SIEMENS

SIMATIC

System Software for M7–300/400 Program Design

Programming Manual

This manual is part of the documentation package with the order number:

6ES7802–0FA14–8BA0

C79000-G7076-C851-02

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.

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

Siemens Aktiengesellschaft

C79000-G7076-C851-02

Preface

This manual is designed to help you when writing C programs for the automation computer family M7-300 and M7-400 under the operating system M7 RMOS32. It provides you with key information about the features and the functions of M7 RMOS32. It contains information on:

- The M7 RMOS32 hardware and software environment,
- The structure and function of M7 RMOS32,
- The function and services of the RMOS API,
- The function and services of the M7 API,
- Mechanisms for communicating with other S7 and M7 systems,
- Design and development of user programs.

Audience

This manual is mainly intended for those writing C programs for the automation computers M7-300 and M7-400.

Scope of this Manual

This manual is applicable to the automation computers M7-300 and M7-400 with the system software M7-SYS RT V4.0.

What is New?

This manual provides the following new topics on changes and extensions of functions supported by version V4.0 of the system software.

Topic	Chapter
Read and write floating point numbers	7.4
Communication using Configured Connections via PROFIBUS DP (with the IF 964-DP interface submodule) and via Industrial Ethernet (with the CP 1401 communication processor)	8.5
Socket Interface for TCP/IP communication	8.12

The RMOS API provides additional function calls for writing loadable drivers. Information on the operating principle of loadable drivers under M7 RMOS32 and the methods used to program them can be found in the electronic manual *Writing Loadable Drivers* which is part of the M7-SYS RT system software (see below).

Scope of the Documentation Package

The system software for the automation computers M7-300 and M7-400 are documented in several manuals which can be ordered separately from the respective product. The available manuals are listed in the following table.

Manual	Contents
System software for M7-300 and M7-400 Installation and Operation User Manual	Installation on operation of automation computers M7–300/400.
System software for M7-300 and M7-400 Program Design Programming manual	Design and development of C/C++ programs.
System software for M7-300 and M7-400 System and Standard Functions Reference Manual	Detailed information on programming under M7 RMOS32.
System software for M7-300 and M7-400 Writing Loadable Drivers Electronic Manual M7LDRV1B.PDF	Design of loadable drivers under M7 RMOS32, programming and reference manual.

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*, 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.

How to Use this Manual

This manual contains a guide to the key points of program design, program development and program testing.

Product Overview

Chapter 1 gives you an overview of the system environment in which you develop and run your M7 RMOS32 programs. It also describes the technical environment within which an M7 computer can be used to solve automation assignments.

Developing a simple program for the M7

Please read chapter 2 to get an idea of the steps that are necessary to develop a user program. The procedure to be followed is illustrated with a simple example program.

Structure and features of M7 RMOS32

Chapter 3 talks about the system structure and functions of M7 RMOS32. It describes each of the separate components of M7 RMOS32 (M7 RMOS32 kernel, M7 servers) and the interfaces and communication mechanisms between the various components.

RMOS API function calls

Chapter 4 describes the features of the M7 RMOS32 kernel and discusses the functions which the RMOS subsystem provides for task control and task communication. This chapter also provides information and instructions on using the RMOS function calls in your C user program and how you program the calls.

M7 API function calls

Chapters 5, 6 and 7 show you how to access process I/Os using the M7 API function calls provided by the integrated M7 servers. They also contain information on P bus communication with other modules, alarm processing, treatment of time events, working with S7 objects, and describe the services of the operating mode server and the free cycle server.

Communication

Chapter 8 informs you about the various communication mechanisms which the M7 automation computer provides for communicating with other automation systems via configured and non-configured connections.

Loadable Drivers

Chapter 9 describes how to implement programs using loadable drivers.

Design and development of a user program

Chapter 10 uses an example to illustrate how to implement an automation assignment in separate tasks of a C user program for an M7/S7-CPU and for an M7 FM.

Cycle and Reaction Times

Chapter 11 describes how to determine reaction times of an M7-300/400 using measured time values.

Index, Glossary

The index helps you to find descriptions of important keywords, and the glossary provides definitions of technical terms.

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).

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.

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`).

Table of Contents

Product Overview

1

Chapter Overview

1.1 Overview

What is M7 RMOS32?

M7 RMOS32 is the 32 bit real-time multitasking operating system which is used for the modules of the SIMATIC M7 programmable controller system.

Figure 1-1 shows an example of the environment which is used to develop M7 RMOS32 programs and also shows where the finished programs reside and execute.

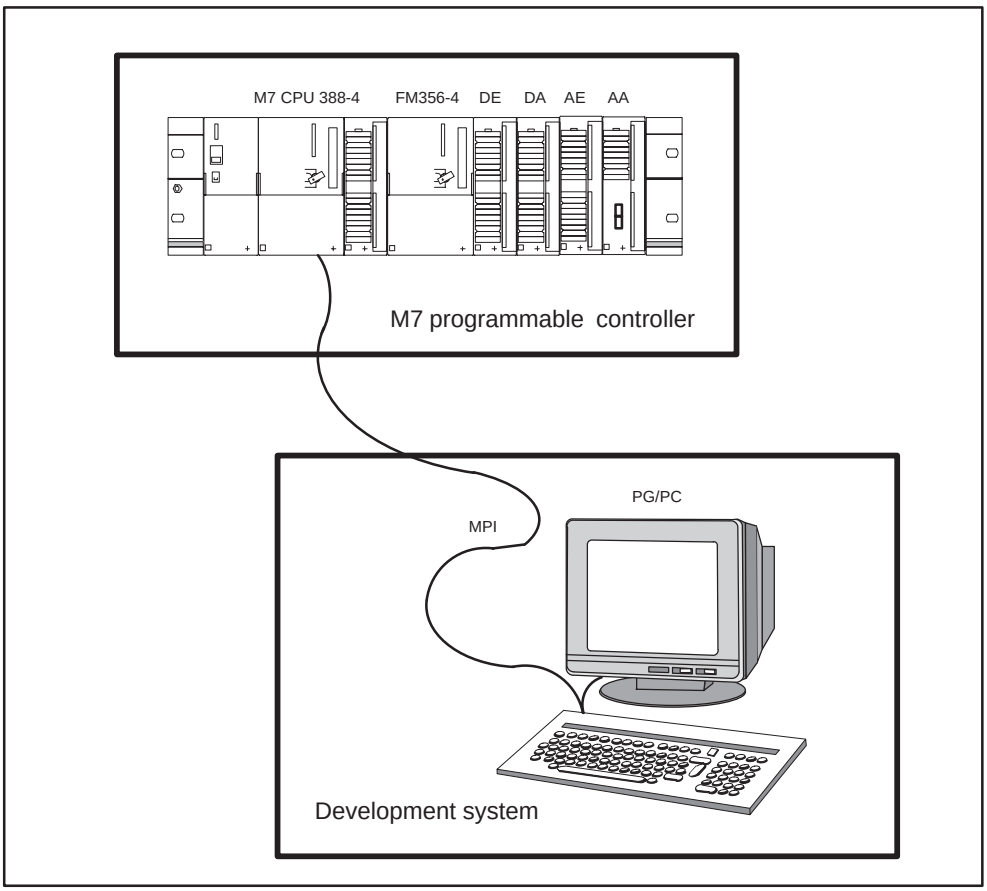


Figure 1-1 System Environment of an M7 RMOS32 Program

M7 Programmable Controller

The hardware platform for M7 RMOS32 programs are the CPU and FM modules of the automation system family SIMATIC M7. All of the modules are AT-compatible and provide nearly all system components of a standard AT compatible PC.

M7 RMOS32 is a 32-bit real-time multitasking operating system (Real-time-Multitasking-Operating-System), which can be used alongside the MS DOS operating system. Communication with further S7 components takes place through integrated M7 servers.

Note

M7-SYS RT does not support further development of MS DOS application programs.

Development System

Program development takes place with the Siemens STEP 7 basic package, which runs on a standard PC or a programming device under the MS Windows 95/NT operating system. The STEP 7 development environment allows you to create automation software without having to become involved with system-specific details such as configuration of rack parameters.

The basic package can be extended with additional tools for programming and developing programs for M7 RMOS32. The basic package can also be combined with the integrated development environment provided by the Borland C/C++ compiler.

The development system is connected to the M7 programmable controller via MPI (Multi-Point Interface).

1.2 Applications of M7 RMOS32

Types of Application

The automation computer SIMATIC M7 and its components lend a high computing speed and overall performance to the SIMATIC S7-300 and S7-400 PLC system family.

The standard PC architecture of SIMATIC M7 and the use of M7 RMOS32 together with the appropriate development tools provide the development engineer with an easily-programmable extension to the SIMATIC family of automation products.

Real-Time Tasks

Typical applications of M7 RMOS32 in M7 automation computers are high-speed technological assignments such as:

- Fast PID controllers (< 10 ms):
  - Synchronizing several drives
  - Winding machines
  - Flying shears
- Rapid positioning tasks:
  - Hydraulic presses
  - Cut positioning on-the-fly
- Rapid data acquisition

Such tasks can be developed and solved rapidly and efficiently in the programming language C/C++.

1.3 Structure of M7 RMOS32 Systems

The structure of M7 RMOS32 allows the cost-saving feature of being able to simultaneously solve process control problems and data processing and/or visualization problems on the same M7 automation computer.

Hardware has now become so powerful that it allows parallel operation of process control tasks and visualization software on the same computer. M7 RMOS32 is provided with suitable mechanisms to allow both types of task to be executed side by side.

The real-time multitasking operating system M7 RMOS32 which executes on the automation computer mainly consists of the RMOS kernel itself and the associated server subsystems for connecting the user tasks to the S7 environment.

Figure 1-2 shows a simplified diagram of the structure of an M7 RMOS32 system.

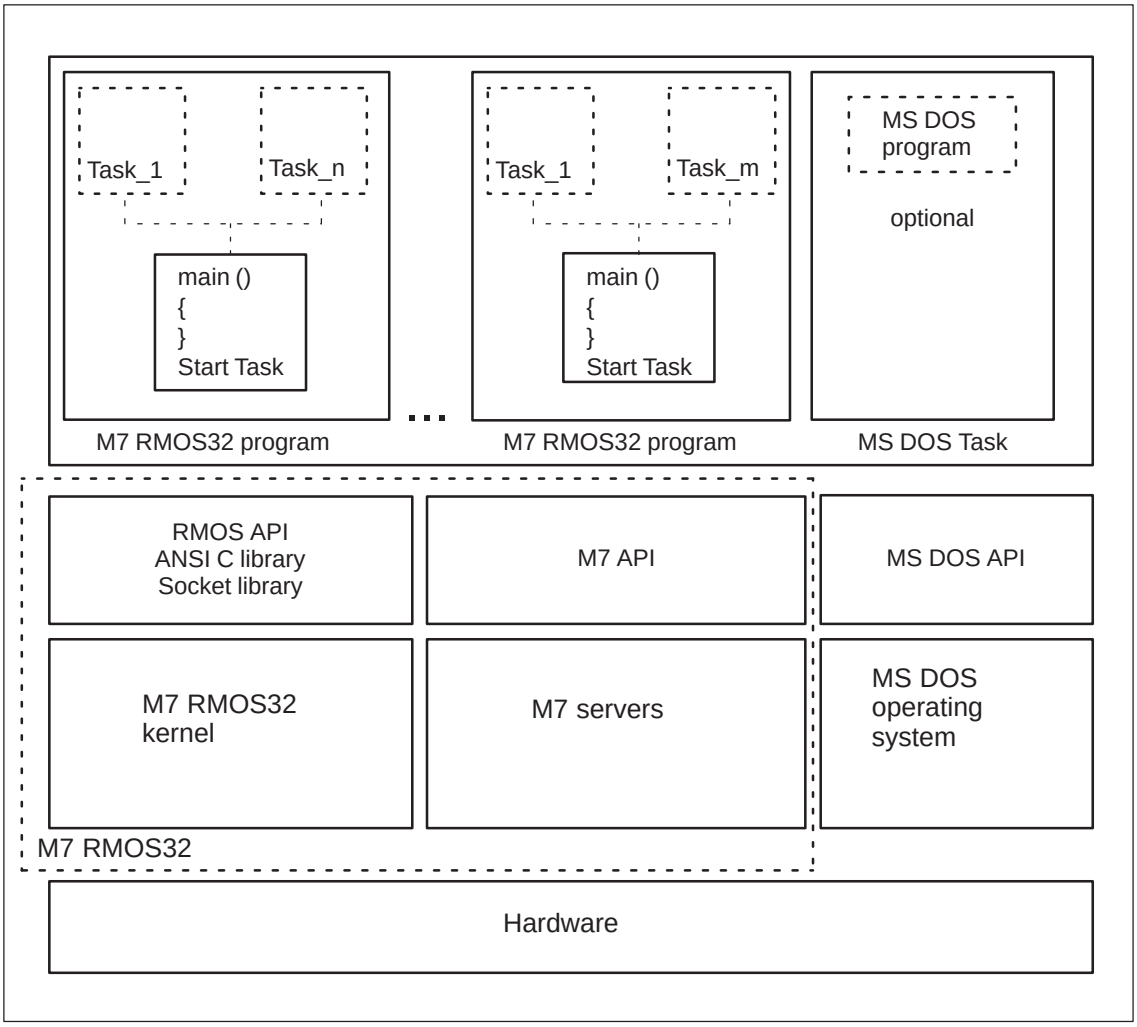


Figure 1-2 Structure of an M7 RMOS32 System

RMOS Subsystem

The kernel of M7 RMOS32 is responsible for multitasking, in other words the control of program execution and the control of the hardware. When considered as a layer model, the kernel is the innermost layer and it communicates directly with the hardware.

User tasks have access to the following functions through the RMOS API, which is a part of the M7 RMOS32 kernel:

- Functions for task management and control (creating, deleting, starting, stopping and terminating tasks, priority control of tasks, status interrogation of tasks)
- Functions for task coordination and communication (administration of semaphores and event flags, message exchange via message queues and mailboxes)
- Functions for resource management (memory management, management of logical resource names)

M7 API

By their very nature, automation programs are closely linked to system components for the input and output of process signals and they also need to communicate with other (intelligent) components of the S7 system. Within the M7 RMOS32 system, such functions are implemented by one or more integrated M7 servers.

User tasks can carry out the following functions and services through the M7 API:

- Accessing process I/Os (inputs and outputs of the signal modules and /or reading from and writing to onboard process I/Os, and sending alarms to CPU modules)
- Communicating with other modules via the P bus
- Creating, reading, writing and deleting S7 objects and/or getting information about access to objects by other components
- Reacting to process and diagnostic alarms
- Reacting to time events
- Reacting to operating modes and/or operating mode transitions
- Communicating with other system components via the communication bus and the MPI

MS DOS Operating System

In addition to the M7 RMOS32 operating system, it is possible to install and run the single-tasking PC-operating system MS DOS on the M7 programmable controller.

The respective MS DOS operating system executes as a low priority task under M7 RMOS32 and simulates a complete operating environment for a DOS program.

Depending on the processor load due to the M7 RMOS32 tasks, the DOS machine gets more or less computing time for executing its programs.

Hardware

The hardware of the M7 programmable controller can be expanded with optional modules (VGA module, floppy disk/hard disk module, modules for serial and parallel ports etc.) to full PC/AT standard. In addition to this, the standard PC hardware of an M7 CPU is provided with special components for interfacing to an S7 system (MPI, P bus and communication bus interface).

- Low level hardware components which are important to the system, such as interrupt control or timer and processor flags, are controlled by the M7 RMOS32 kernel in order to guarantee that the real-time requirements are met.
- Access to S7-specific hardware such as MPI, P bus and communication bus interface takes place solely through the integrated server subsystems. This ensures a transparent integration of the M7 computer in S7 systems.
- Further PC hardware (hard disk, serial and parallel ports, VGA module) can be serviced either by the M7 RMOS32 kernel or by the MS DOS operating system. If they are serviced through MS DOS, this allows the usage of drivers from the respective operating system.

Task Structure of a Program

M7 RMOS32 provides a number of multiprogramming and multitasking functions which allow a complex automation sequence to be subdivided into small, easy to understand subfunctions.

The overall program can be subdivided into separate C programs which can then be separately developed and tested.

Within each C program, each of the separate functions can execute as a so-called “task”. Tasks are self-contained sub-programs which can run in parallel with other tasks.

1.4 Required Skills

What Skills Are Required?

The following skills should be available if you want to develop M7 RMOS32 programs with the help of text-oriented high-level language programming in C/C++:

- Practice in using the SIMATIC Manager

  Well-founded knowledge of the STEP 7 basic package is essential for overall project administration and/or for uploading the program modules from the development system to the M7 programmable controller.

  The relevant information is contained in the user manual “Standard software for S7 and M7, STEP7”.

- Experience with the programming language C and/or C++

  In order to develop programs, experience with the programming language C and /or C++ is required. You should also know how to use the integrated developing environment of Borland C/C++.

  The required information is contained in the documentation provided by Borland with its development environment.

- Knowledge of STEP 7

  The degree of prior experience with STEP7 is highly dependent on the role of your planned application within the overall automation system. The required prior experience can extend from knowledge of the addressing scheme of the process I/Os through design of S7 modules and may need programming experience with STEP 7 itself.

  We recommend the Programming Manual *System Software for S7-300/400, Program Design*. It provides an introduction to S7 programming (commands, data types and structures, functions and function calls etc.).

  However, if you only want to access the signals of the process I/Os and do not need to implement complex communication with other S7 CPU modules, you should be able to find all of the required STEP7 information in this manual.

Developing a Simple Program

2

Chapter Overview

2.1 Overview

In order to develop an M7 RMOS32 program on an M7 programmable controller, we recommend you to start with separate self-contained steps which are then carried out in a particular sequence. According to the results or success of each of the steps, it may be necessary to repeat them (recursive method).

Sequence of the Development Steps

Assuming that the M7 is already configured and parameterized, the development steps should be carried out in the following sequence:

1. Create a C program with M7 ProC/C++.
2. Edit the C program in the Borland environment, compile and link it.
3. Choose an operating system and transfer it to the M7 programmable controller together with the user program.
4. Start user program on the programmable controller.

What is Described in this Chapter?

This chapter uses a simple example program to explain each of the development steps. The example program only requires a very simple configuration of the M7-300 programmable controller and is thus executable on nearly any M7 system.

The aim of this chapter is to develop a program which causes the output signals of a digital output module to change state once per second.

This chapter describes the basic procedure to follow when creating user programs for an automation computer under M7 RMOS32.

2.2 Requirements

Hardware Requirements

In order to be able to carry out each of the steps in developing the example program, the following minimum configuration is required for the M7 programmable controller (see Figure 2-1):

- An M7-300 CPU
- A power supply
- A digital output module

It is also necessary to connect the development system to the M7 programmable controller via MPI. Connecting the two units together and setting the MPI address is described in the STEP7 user manual.

Software Requirements

The following software components are required on the development system:

- STEP 7 basic software V3.2 or higher
- M7 SYS RT system software
- M7 ProC/C++
- Borland C++

It is also assumed that you have already created a suitable project with the SIMATIC Manager and that the hardware has been configured. Please make sure that the logical starting address of the example configuration's digital output module has been set to 12.

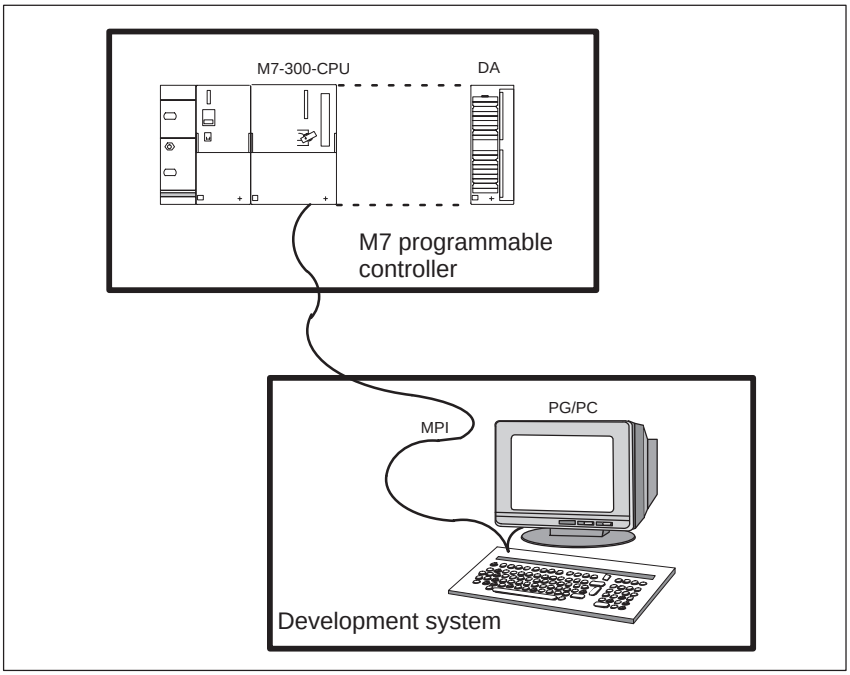


Figure 2-1 Hardware Arrangement for the Example Program

2.3 Creating an M7 RMOS32 Program with M7 ProC/C++

Creating a C Program Framework

Within the already created project, you should first create the M7 software component (in other words a “container” for the C program). Proceed as follows:

1. Select the symbol of the M7 program in the plant mimic of the SIMATIC Manager. Choose the **Insert ▸ M7 Software ▸ C Program** menu command.

   Result: A C-program object is created and is now visible in your project. The C program represents a Borland project that you can edit using the Borland Integrated Development Environment (IDE).

2. Select the created C program and choose the **Edit ▸ Object Properties** menu command. The “Properties” dialog box is opened.

3. Change the name of the C program to *Test program* and click “OK” to exit the dialog box.

   Result: The C-program object will now be shown under the name Test program in the window of the SIMATIC Manager. All necessary header files are already included (see Figure 2-2).

2.4 Editing, Compiling and Linking a C User Program

Editing the C Program

We will now show you how to edit the C program.

1. Select the object named *Test program* and choose the **Edit ▸ Open Object** menu command or simply double click on it. The Borland IDE is started and the project window of the selected software component is displayed. If the source code is not yet shown in the editor window, you can open it for example by clicking twice in the Borland project window on the source code node.

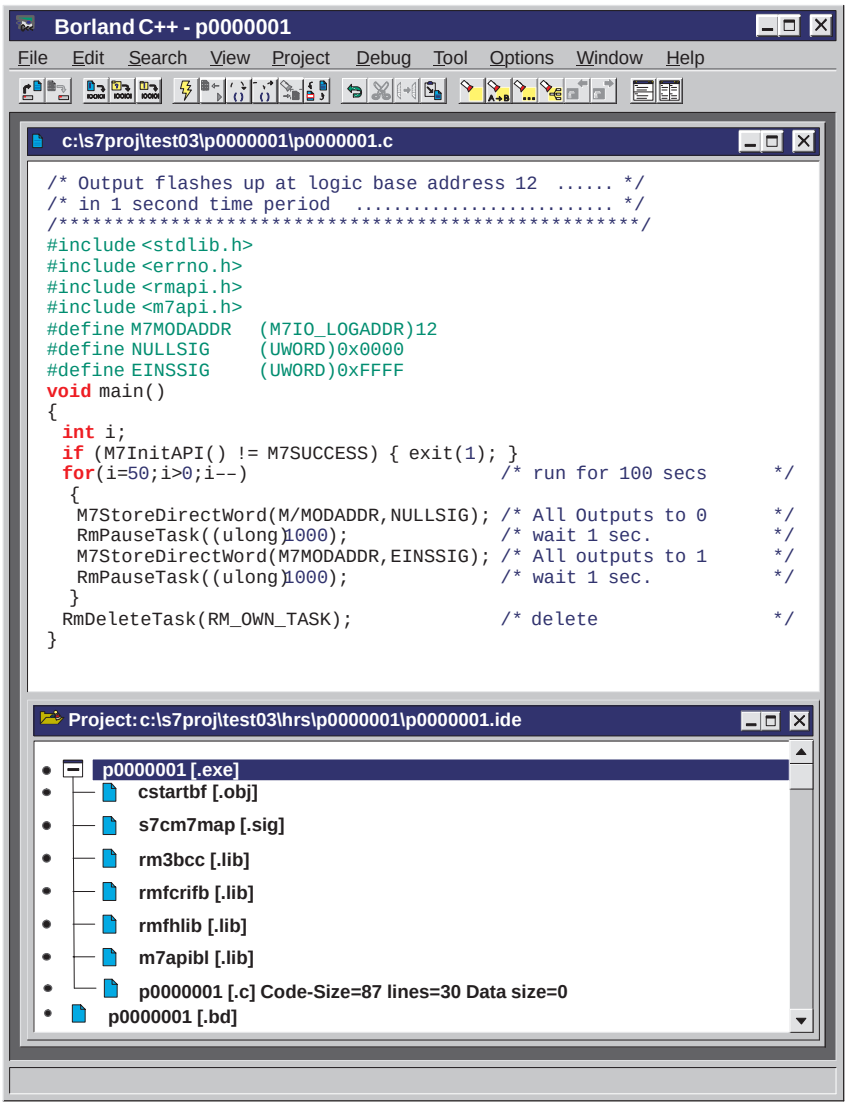


Figure 2-2 An M7 RMOS32 Example Program in the Borland Editor Window

2. A standard source code template has already been created automatically. Edit this file until it corresponds to the example program shown in Figure 2-2. Make sure that you include the header files shown in the figure, since they are required by the example.

Note

Please note that the example program contains no error checking routines.

Compiling and Linking the C Program

After you have edited the program as shown in Figure 2-2, you can compile it and link it. The following two methods are available:

- You select the **Project ▸ Build all** menu command. This command newly compiles and links the entire Borland C project.
- You select the **Project ▸ Make all** menu command. The last modification times of the individual files are analyzed and the relevant compilation and link sequences are carried out.

2.5 Transferring the User Program to the M7 Programmable Controller and Starting It

After you have generated the executable program for your M7 CPU, you can transfer it to the file system on the M7 programmable controller.

Prerequisite: The M7 programmable controller must be started and there must be a functioning MPI connection between the M7 programmable controller and the programming device to transfer the program.

Procedure

To transfer the application program to the M7 programmable controller:

1. Select the “M7 program” object in your project window. Select the **PLC ▸ Manage M7 System** menu command. The dialog box for selecting operating systems and programs is overlaid.
2. Open the “Programs” tab.

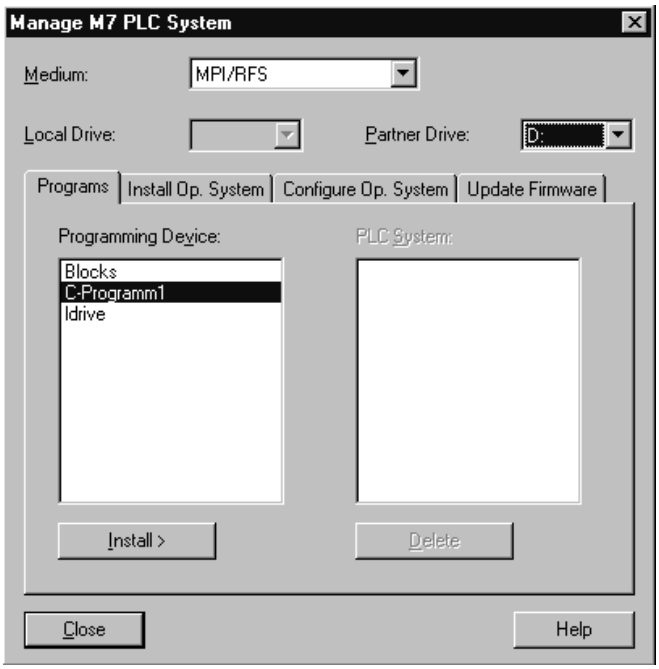


Figure 2-3 Dialog Box: Manage M7 PLC System

3. Select:
   - “MPI/RFS” as the medium
   - Select the first free drive in the list, for example N: as the Local Drive (Windows 95 only). Select “C” as the Partner Drive, which is the hard disk drive of the M7-300.
4. Select your C program from the selection field and click on the “Install” button.

   The program is transferred to the M7 programmable controller and is entered automatically in the \etc\inittab file. The program starts automatically during the next system power up.
5. Then carry out a reset and consequently booting the operating system on the M7 programmable controller.

   Result: The application starts. The E12 input signals are displayed on the A12 outputs and flash at seconds intervals for a duration of 100 seconds.

Structure and Features of M7 RMOS32 3

Chapter Overview

3.1 Overview

M7 RMOS32 provides functions for implementing automation assignments in the SIMATIC S7 system. The available functions are mainly tailored towards open-loop and closed-loop control tasks in the automation field.

In addition to the multitasking and real-time features required for process control, M7 RMOS32 also provides facilities to run data processing applications under the MS DOS operating system.

What is Described in this Chapter?

This chapter first discusses why real-time and multitasking operating systems are needed to solve complex automation assignments. This is followed by an overview of the structure and features of the various components which make up M7 RMOS32.

Note

The standard and system functions of the RMOS API, M7 API, Sockets library and CRUN library are not implemented as C++ class libraries.

3.2 Multitasking Structure and Real-time Features of Technical Processes

When automating technical processes, it is often necessary to be able to monitor and control several separate parts of the process simultaneously. This is because most technical processes can be subdivided into several separate but interlinked tasks which execute in parallel with each other.

Such processes can only be automated successfully with a multitasking operating system, because several processes running in parallel cannot normally be controlled satisfactorily by a sequential program.

Multitasking=Multiprogram Operation

In practice, complex automation processes are usually implemented by subdividing the overall objective into small, easy to handle subfunctions. Each of the subfunctions are then allocated to separate program tasks.

Tasks are self-contained subprograms with their own data, and can be started and executed independently of each other. However tasks must be coordinated and communicate with each other in order to be able to jointly achieve the required result.

Example of Multitasking

Figure 3-1 is a sketch of a typical assignment for a multitasking system: Components B are lifted from conveyor A by gripper C and placed on pallet D.

The control of the conveyor and the control of the gripper movements are both self-contained control functions. However, the movement of the conveyor is dependent on the time point when the gripper lifts a component from the conveyor. Accordingly, the two control tasks must be coordinated with each other.

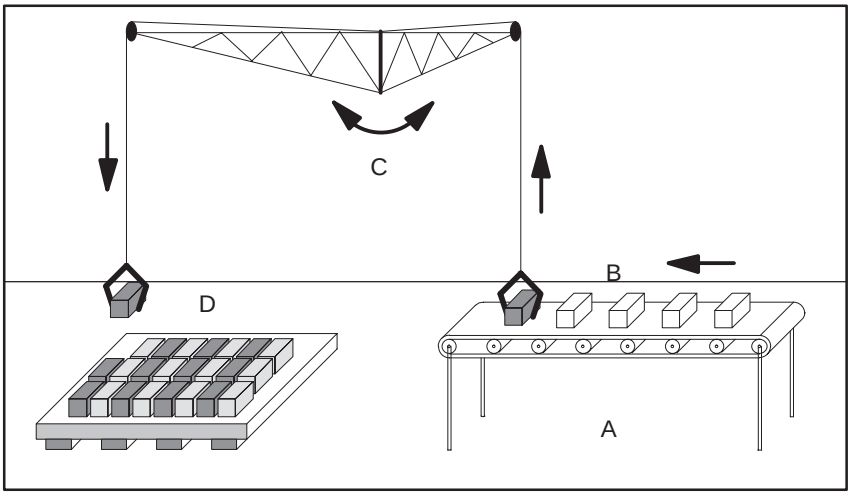


Figure 3-1 Example of a Multitasking Assignment

Real-Time

In order to be able to control the automated process without interruption, the associated operating system must ensure that all necessary tasks take place at the correct time, in other words within a predefined period of time.

Systems which control technical processes must always be closely coupled to the process with respect to timing. In particular the beginning of each processing step is always the result of a signal from the process, and the processing step must be completed by a certain time which is also dependent on the process parameters.

If the result of the processing step is not available in time, this can lead to an incorrect or uncontrolled reaction of the process.

Example

The following is an example of a typical real-time task in a testing station. Workpieces are scanned by a camera while they are moving on a conveyor belt and their quality is checked with an evaluation computer. The defective workpieces are then separated from the good ones using a sorting gate.

In this example, the time period between the recording of the data to be evaluated and the output of the sorting signal must not exceed a predefined value, since otherwise it would be too late to sort out the defective workpieces.

3.3 Features of M7 RMOS32

M7 RMOS32 is an operating system with facilities for pseudo-parallel control of several processes which are running at the same time. It also provides mechanisms for the complete integration of an M7 programmable controller in a SIMATIC S7 system.

Real-Time Features

M7 RMOS32 satisfies the typical real-time requirements which are placed on an operating system for automating technical processes. These requirements include the following:

- Guaranteed maximum reaction time to external asynchronous events such as exceeding a limiting value or reaching a particular position.
- Many tasks don't only depend on external events but also on predefined timing. For example, digital control algorithms require cyclic (iterative) processing and such algorithms are generally very dependent on variations in the cycle time. M7 RMOS32 thus allows a series of input signals to be interrogated and processed at regular intervals, and to convey the resulting control signals to the process.
- Some tasks such as the acquisition of log data must take place at an exact point in time. M7 RMOS32 contains its own clock and a mechanism which allows programs to be started at specified times.

Multitasking Features

In addition to the necessary real-time features, M7 RMOS32 also contains a range of multitasking functions:

- M7 RMOS32 is able to switch the processor between several tasks in order to achieve pseudo-parallel processing of the tasks.
- M7 RMOS32 includes a mechanism (scheduler), which decides which task at which point in time is assigned to the processor, and when the task should relinquish the processor again in favor of another task. The mechanism is designed to ensure that important tasks have a higher priority and less important tasks are made to wait ("preemptive multitasking").
- M7 RMOS32 also provides operating system calls which allow the synchronization of independent tasks.

C Programming Interface

User programs have access to the functions and services of M7 RMOS32 through a C programming Interface.

In addition to the standard system calls, the extensibility of the C language allows you to implement application-specific functions and libraries which are adapted to your own special requirements.

3.4 Components of an M7 RMOS32 System

In order to satisfy the wide range of functions which are required by process automation and to maintain the necessary flexibility for the future, M7 RMOS32 has been designed with a modular structure. It consists of the following main components and subsystems:

- The user program to solve the automation assignment,
- The M7 RMOS32 kernel with the associated programming interface (RMOS API) a socket library and an ANSI C runtime library,
- M7 servers with the associated programming interface (M7 API) for communicating with other modules within the S7 system,
- The MS DOS operating system (optional),
- The hardware of the automation computer.

The following figure shows a simplified structure of an M7 RMOS32 system.

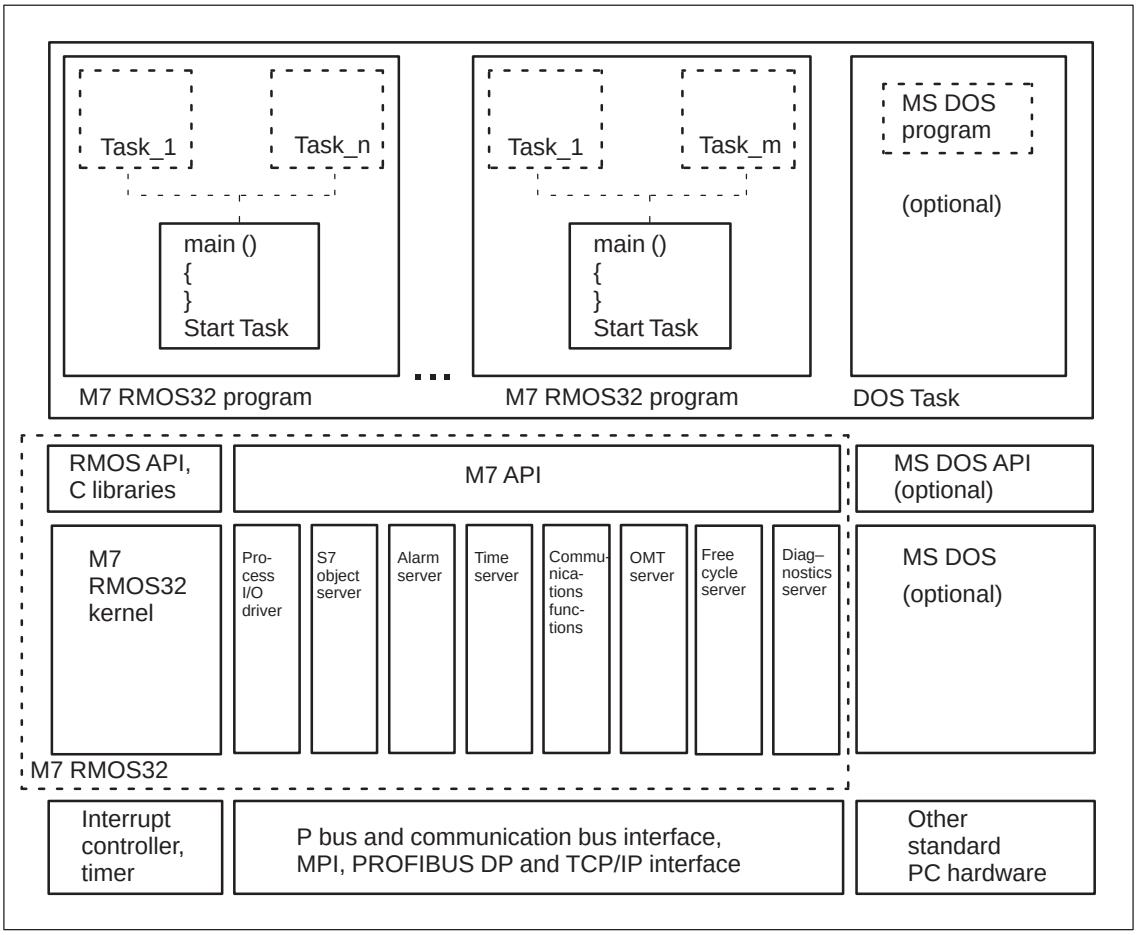


Figure 3-2 Modular Structure of an M7 RMOS32 System

Structure of M7 RMOS32

The real-time multitasking operating system M7 RMOS32 which executes on the M7 programmable controller mainly consists of the M7 RMOS32 kernel itself and the associated server subsystems to allow communication between the user tasks and the S7 environment.

RMOS Subsystem

The kernel of M7 RMOS32 has the job of multitasking, in other words the control of program execution and the handling of the hardware required by multitasking (interrupt controller, timer).

If MS DOS is not being used on the M7, then the M7 RMOS32 kernel is also responsible for controlling the rest of the PC hardware (for example hard disk, parallel and serial interface etc.).

User tasks can access the functions and services of the M7 RMOS32 kernel through the RMOS API. Also the C Runtime Library and the Sockets Library can be used.

M7 API

Process control and automation programs naturally have close ties to system components for the input and output of process signals, and also need to communicate with other (intelligent) components of the S7 system. Within M7 RMOS32, these jobs are carried out by several integrated M7 servers.

A user task can access the functions and services of the M7 servers through the M7 API programming interface.

MS DOS Operating Systems

In addition to the M7 RMOS32 operating system, it is possible to install and run the single-tasking PC-operating system MS DOS V 6.22 on the M7 programmable controller. MS DOS executes as a low priority task under M7 RMOS32 and simulates a complete operating environment for a DOS program.

According to the processor load due to the M7 RMOS32 tasks, the DOS machine gets more or less computing time for executing its programs.

If MS DOS is installed, then it is usually responsible for handling the PC hardware (for example hard disks, serial and parallel ports, VGA module, expansion cards, etc.). This allows the user to integrate low-cost standard MS DOS drivers into the overall system in order to access them from a MS DOS user program.

Hardware

The hardware of the M7 programmable controller can be expanded with optional modules (VGA submodule, floppy disk/hard disk module, submodules for serial and parallel ports etc.) to full PC/AT standard.

In addition to this, the standard PC hardware of an M7 CPU is provided with special components for interfacing to an S7 system such as the MPI, the peripheral bus (P bus), the communication bus (K bus), the TCP/IP and the PROFIBUS DP interfaces.

3.5 Structure and Features of the RMOS Subsystem

The following figure shows the components of the RMOS subsystem.

- M7 RMOS32 kernel: the kernel includes all necessary services for implementing a multitasking system.
- RMOS API: This is the programming interface which allows user tasks to access the services of the M7 RMOS32 kernel.
- ANSI C runtime library: The C runtime support makes available all of the C library functions. The C runtime library is designed as a shared library.
- M7 RMOS32 CLI: M7 RMOS32 includes a user interface CLI (Command Line Interpreter) the command syntax of which is very similar to MS DOS. The user environment can be controlled from the development system via a remote terminal task.

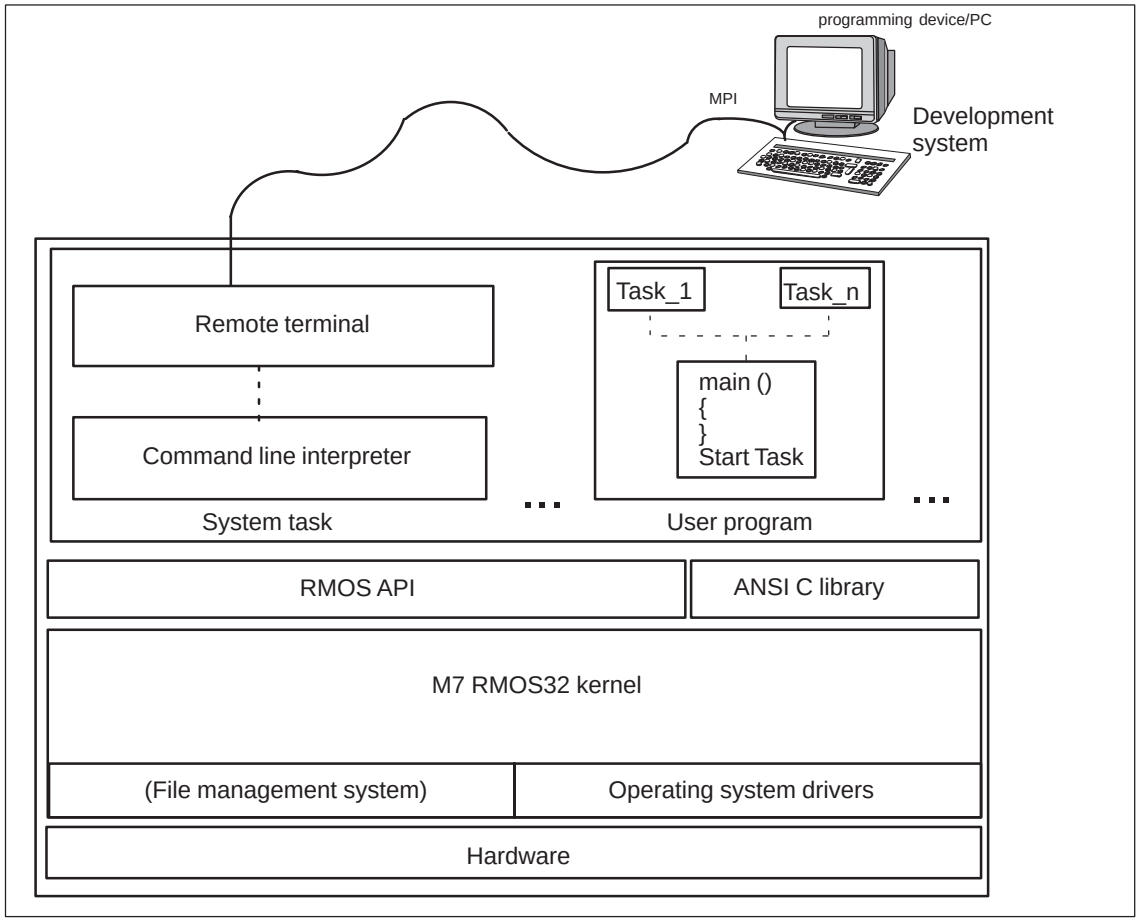


Figure 3-3 RMOS Subsystem

M7 RMOS32 Kernel

The M7 RMOS32 kernel can be considered to be the heart of the overall system. If the system is considered as a layered model, the kernel is the innermost system layer and has access through corresponding drivers to the hardware which is necessary for multitasking (timer, interrupt controller). The kernel also manages system startup and allows programs to access its services through a clearly defined application programming interface (API).

Starting Programs

The kernel also arranges for M7 RMOS32 user programs to be started. Following system boot, the M7 RMOS32 kernel checks the **INITTAB** file in the directory <boot drive>\ETC and starts all programs which this file specifies. This allows the user to automatically start all required M7 RMOS32 programs following a successful system boot by making appropriate entries in the **INITTAB** file.

All M7 RMOS32 programs which are transferred from the development system to the M7 programmable controller using the SIMATIC Manager are automatically entered in the **INITTAB** file.

RMOS API

The RMOS API allows C user programs to access the functions and services of the M7 RMOS32 kernel.

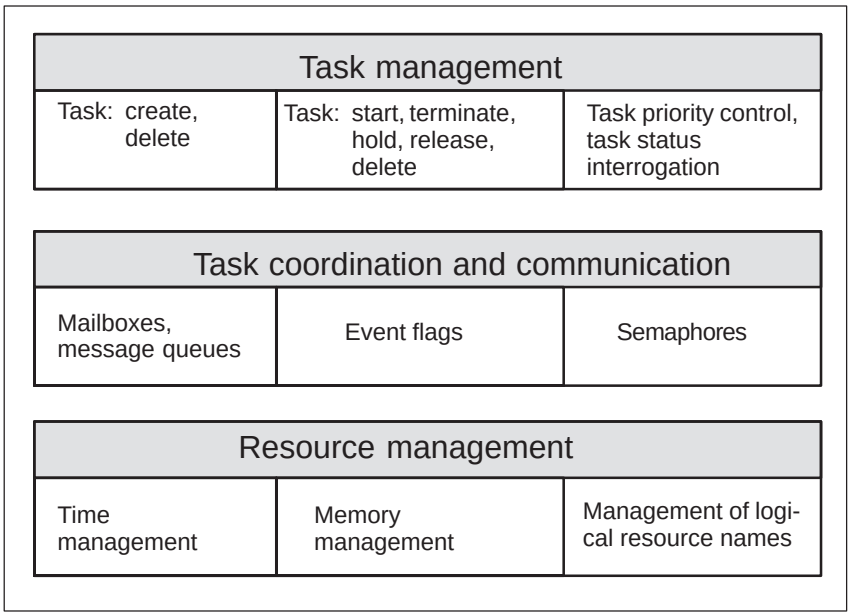


Figure 3-4 System Services of the M7 RMOS32 Kernel

Figure 3-4 shows a simplified overview of the available system services and resources. All resources can be dynamically allocated, in other words they can be defined during program runtime with appropriate RMOS API calls.

User tasks can access the following functions of the M7 RMOS32 kernels via the RMOS API:

- Functions for task management and control:
  - Create, delete, start, terminate and cyclically start tasks
  - Wake up other tasks which are on hold
  - Start tasks from an interrupt handler
  - Change the priority of a task (own or other task)
  - Change the status of a task (own or other task)
  - Disable and enable the scheduler
  - Disable and enable interrupts

- Functions for task coordination and communication:
  - Mailboxes: Executing tasks can send messages to so-called mailboxes and/or collect messages from mailboxes. Mailboxes are used to implement data exchange between several sender and receiver tasks.
  - Message Queues: A message queue is a mailbox which is assigned to one task only and from which only this task can read the messages. A message queue is used to send messages to a specified receiver task. This allows several tasks to send messages to a specified receiver task without having to consider timing.
  - Event flags: event flags are used for task coordination and are an efficient way for exchanging binary messages.

    Event flags consist of logical bits of data, and they are grouped together as event flag groups. Running tasks can set, reset or test one or more event flags immediately or after a defined time. It is also possible to wait for the setting of one or more event flags.
  - Semaphores: When two or more tasks have access to the same resource for example a data structure, semaphores are used to ensure that access is only granted to one task at a time.
- Functions for resource management
  - Time management: Time management is used to implement time-related system services such as the cyclical start or pause and time–out functions for tasks.
  - Memory management: Memory management allows tasks to request and/or release contiguous blocks of memory. This allows tasks to request and/or release dynamic areas of memory, for example to store measured values.
  - Resource management: The management of logical resource names allows each resource (for example task, mailbox, event flag etc.) to be entered in a catalogue as a C character string (max. 15 characters + \0).

    Appropriate RMOS API calls can be used to request the ID (identification number) which is allocated to each resource. This ID is usually needed as a parameter for further RMOS API calls.

Chapter 4 contains a detailed description of the functionality and the uses of the associated API calls.

C Runtime Library

The M7 RMOS32 C runtime library provides user programs in the M7 RMOS32 environment with all C functions specified in the ANSI Draft Standard ISO/IEC DIS 9899. The C runtime library is designed as a shared library and it is reentrant.

The C library supports the following function groups:

- Character handling, string and memory operations
- Input/output operations for example hard disc, terminal, printer etc.
- Memory allocation
- Mathematical functions, for example sin, cos, etc. and floating point arithmetics
- Control functions and error handling

The library calls are made available to M7 RMOS32 programs by including the standard C header files. The required libraries are automatically included when linking the program.

Tasks

Each M7 RMOS32 user program generally consists of several tasks. A task within the user program is a self-contained program unit (function) with associated data and stack area.

Tasks, in other words functions within the same C user program, can jointly access global program variables in order to coordinate with each other and to exchange information.

Tasks in different C user programs can only exchange information with the help of the coordination and communication mechanisms provided by the M7 RMOS32 kernel (for example resource catalog, message queues, mailboxes etc.).

A task can be started and executed by the M7 RMOS32 kernel independently of other tasks. In order to to this, the kernel assigns an internal data structure for each task which stores all of the parameters required for correct multitasking.

Typically, the internal data structure contains the following entries:

- Task ID to identify the task to the kernel
- Start address of the task
- Address of the stack
- Priority of the task on starting

In addition to this, the following variables are saved in the internal data structure when tasks are swapped:

- Current set of registers
- Current priority

These parameters are required by the kernel when the task is reactivated to allow the task to resume processing at the same point where it left off.

If the task uses C library functions to access the standard input or output device or the file system, the assignments which apply are the same as those which were assigned by CLI when starting the associated job.

User Tasks and System Tasks

Two types of tasks can exist in M7 RMOS32:

User tasks that can be created in a user program with th RMOS API calls.

System tasks that 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 Chapter 4.17).

- **Priorities**

  System tasks can run with higher priority than user tasks (see Chapter 4.5).

Command Line Interpreter CLI

The Command Line Interpreter (CLI) provides a user interface for the M7 RMOS32 operating system. CLI allows commands and programs to be started and operated interactively.

The components and responsibilities of CLI include the following:

- ***CLI Task****:* In the M7 RMOS32 system, CLI is a system task which can be started through a remote terminal connection. A running CLI instance is called a session. A CLI session allows interactive commands to be executed and M7 RMOS32 user programs to be started.
- ***Runtime environment for user tasks****:* CLI also provides the runtime environment for user programs. This encompasses:
  - Opening the input and output devices stdin, stdout, stderr while taking account of redirection as necessary.
  - Parsing the command line into the C format (argc/argv).
  - Initializing the C runtime library.
- ***CLI startup batch file CLISTART.BAT****:* When a CLI session starts, it automatically executes a batch file. The batch file is used to make settings to the environment such as the search path, prompt and current directory.

The usage and functionality of CLI is described in the User Manual “Installation and Operation”.

Jobs

All commands and programs started by CLI are called jobs. They can execute in the foreground (in connection with the console) or in the background (without a connection to the console).

Jobs can be:

- External CLI commands (see the User Manual “Installation and Operation”)
- Batch files
- User programs

Jobs can be nested, in other words a job such as a batch file (parent job) can start further jobs (child jobs). All jobs are controlled by the CLI session from which they were started.

CLI manages the following assignments among other things:

- ***Job number:*** Each job has a job number which is assigned by CLI when the job is started.
- ***Current directory:*** Each job is assigned a current directory when the job is started.
- ***Stdin, stdout, stderr:*** Each job is assigned devices for standard input, standard output and error output. For foreground jobs, the CLI console is used for all input and output. Input and output for background jobs is directed to the NULL device. The assignments can be redirected with appropriate commands.

Remote Terminal Task

A remote terminal task provides a connection between the development system and a system task on the M7 programmable controller (for example the CLI task) via MPI. The remote terminal task ensures that input and output devices stdin, stdout and stderr are transparently redirected to the development system via MPI.

Four terminal tasks are started automatically on booting the M7 system to allow a total of four CLI and/or debugger sessions to be established between the development system and the M7 programmable controller.

3.6 Structure and Features of the M7 API

Process control and automation programs naturally have close ties to system components for the input and output of process signals in order to acquire the necessary process parameters and/or to output signals to the process in order to control it. Generally speaking, both analog and digital signals are necessary.

The following figure shows an overview of the components of the M7 subsystem:

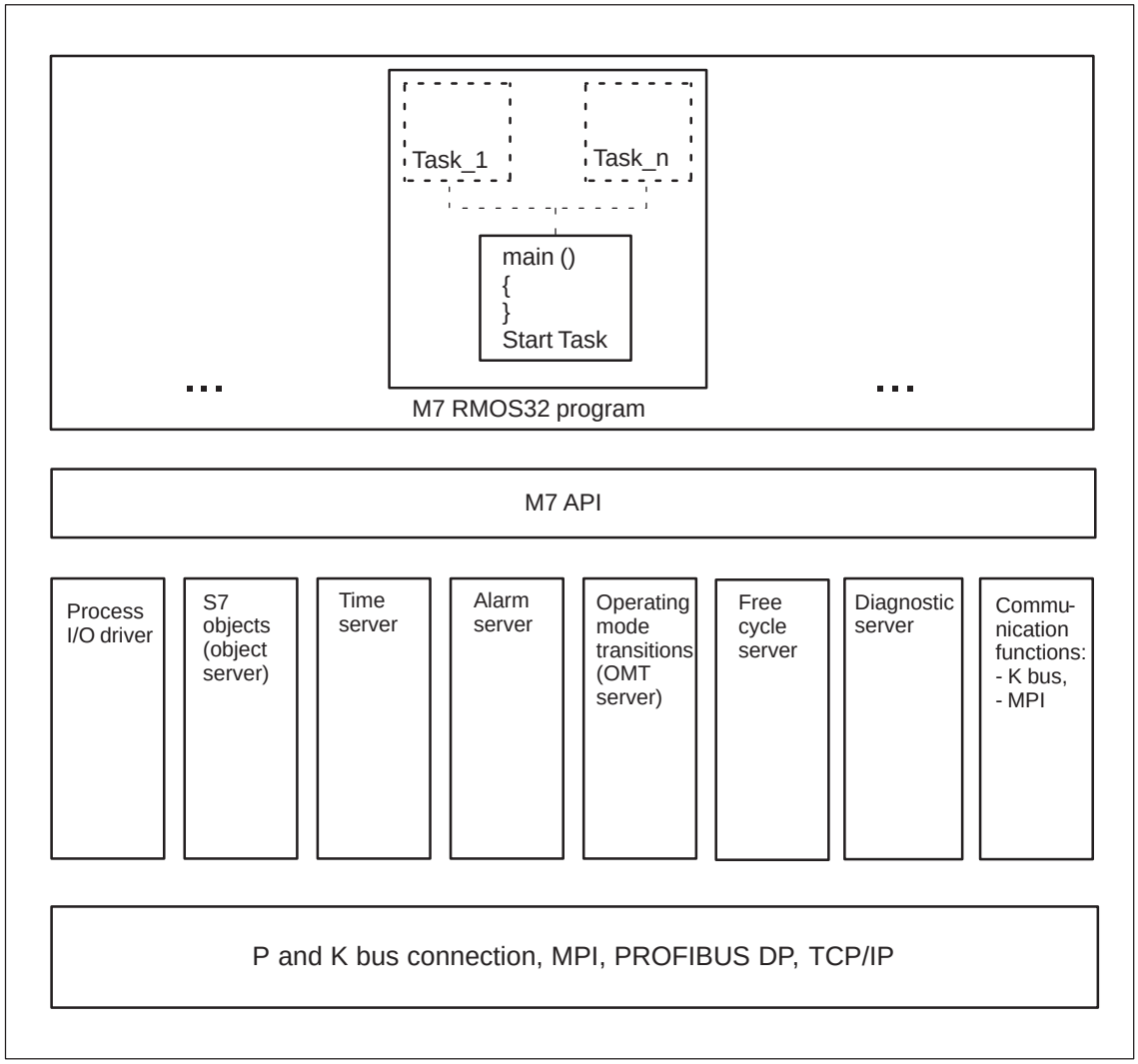


Figure 3-5 Structure of the M7 Subsystem

The Purpose of the Process I/O Driver

The SIMATIC S7 system uses a special I/O bus (P bus) which is connected to the signal modules for input and output of process signals. In addition, the local ISA Bus provides another possibility of connecting process interface submodules by using an EXM module. A connection via PROFIBUS DP is also available.

The function of the P bus can be subdivided into master and slave functions. The master functions include:

- Accessing process I/Os
- Sending parameter records to configure the modules
- Receiving process and diagnostic alarms

The slave functions include:

- Transferring process signal (P bus user data)
- Receiving parameter records from an S7 and/or M7 CPU
- Sending process and diagnostic alarms

The Purpose of the S7 Object Server

The S7 object server coordinates the access to S7 objects between internal programs and external communication partners. The data areas of the S7 object server are comparable with the operand area of an S7 CPU.

The main responsibility of the S7 object server is the management of S7 objects (for example data blocks, flags) which are stored in working memory. In some cases, copies of data blocks are stored in the file system of the mass storage to ensure that their contents are still available even after the module has been switched off.

In addition to the permanent storage of S7 objects on mass storage, the M7 system also provide static RAM to store S7 objects such as data blocks and flag areas. You can specify which data blocks and flags should also be stored in static RAM during configuration with the SIMATIC Manager.

External access to S7 objects takes place via the K bus and P bus, and internal access by user tasks is done with M7 API calls. In addition, M7 RMOS32 tasks can register with the S7 object server to be notified of any access which has been made to S7 objects.

The M7 API Interface provides the programer with the following functions:

- Create and delete S7 objects
- Read and write the contents of S7 objects
- Receive notification of any access which has made to S7 objects
- Redirect and filter access requests to S7 objects

The Purpose of the Time Server

The time server allows M7 API tasks to register for notification of time-related events and to be notified automatically at predefined times. It can generate the following time-related messages:

- Periodic time messages
- Singular time messages
- System time dependent time messages

The time server also provides functions for setting and reading the current system time.

The Purpose of the Alarm Server

The alarm server receives through the P bus and through the ISA bus diagnostic and process alarms which have been triggered by I/O modules and/or by the onboard I/Os. When the alarm server receives an alarm, it identifies the alarm type and informs those M7 RMOS32 tasks which are registered for notification of this type of alarm.

After the alarm has been processed by the task, it is acknowledged by the alarm server to the I/O module which triggered it.

The Purpose of the OMT Server

A SIMATIC S7 CPU defines in its runtime system a particular operating mode model which allows a program to react to a particular operating mode and/or operating mode transition of the S7 CPU and/or the status of the process being controlled (for example automatic restart of a user program when power is restored and/or user program only starts when the SIMATIC S7 CPU has reached the RUN mode, etc.).

In a similar way to the operating system in a SIMATIC S7 CPU, which reacts to operating mode transitions by calling standard OBs, a task running on an M7 automation computer can register with the Operating Mode Transition Server (OMT server) to receive notification of a particular operating mode and/or operating mode transition.

This allows any user program to react in a specific way on reaching a particular operating mode and/or operating mode transition.

The following operating modes are defined within an M7 automation computer:

- STOP
- STARTUP
- RUN
- HOLD
- RESET (memory reset)

OMT requests can be sent to the OMT server in one of the following ways: via P bus and K bus, from the switch on the front panel of the M7 module and from the user program.

- Request via P bus
  Example: An S7/M7 CPU notifies an operating mode transition to an M7 FM. The M7 FM reacts accordingly.
- Request via K bus
  Example: The startup of an operator interface system requests an operating mode transition through a programming device.
- The switch on the front panel of the M7 module
- Request from a user program through an M7 API call

The OMT server carries out the operating mode transition while coordinating with all registered tasks and controls the LEDs on the front panel of the M7.

The Purpose of the FC Server

The FC (Free Cycle) server allows user programs to be synchronized with the operating modes STARTUP, RUN and the cycle control point. In addition to the “free cycle”, which roughly corresponds to the functionality of the background task OB1 in an S7 system, the FC server also generates the cycle control point which is used to to link and unlink S7 objects.

In addition, the FC server provides the same startup functionality which is available on an S7 CPU with the OB100. The startup activities include:

- Clearing the process output image,
- Sending a *startup* message to the registered M7 RMOS32 tasks,
- Waiting for the *“ready”* messages from the registered tasks.

The first cycle is only started after these startup activities have been completed successfully. Within each cycle, the FC server first generates the cycle control point and then the free cycle. The activities of the cycle control point include:

- Sending a *start Cycle* message to the registered M7 RMOS32 tasks.
- Waiting for the *“ready”* messages from the registered M7 RMOS32 tasks.

This is followed by the free cycle. The FC server provides the following services within the free cycle:

- Reading in of the process input image
- Sending a *start Cycle* message to the registered M7 API tasks
- Waiting for the *“ready”* messages from the registered M7 RMOS32 tasks
- Writing the process output image
- Waiting for the specified minimum cycle time
- Monitoring the specified maximum cycle time

User tasks can register themselves with the FC server for the startup, the cycle control point, the free cycle and cycle overflow.

The FC server then notifies the registered tasks at the corresponding times by sending messages. All messages must be acknowledged to the FC server.

If the messages are not acknowledged by the registered tasks in time, in other words within the configured cycle time, the M7 branches by default to the operating mode STOP. However, a branch is not made to STOP mode if a task has registered itself with the FC server for handling cycle time overrun.

The Purpose of the Diagnostics Server

The diagnostics server collects diagnostic messages from the operating system and the programs in a ring buffer. Diagnostic messages include for example operating mode transitions, cycle time errors, I/O errors etc.

The diagnostics buffer can be read by clients which are connected via K bus (for example programming devices). The diagnostics buffer can thus be used to document the incorrect operation of user programs.

Communication

The communication (K) bus is used to communicate with other (intelligent) system components within the S7/M7 station. K bus communication can take place transparently over large distances, whereby data transfer which is outside the S7/M7 station uses the MPI, PROFIBUS DP or Industrial Ethernet (TCP/IP).

MPI (Multi Point Interface) allows the connection of up to 31 devices (programming devices, operator interface systems and other automation systems). Communication is optimized for effective data exchange between individual automation systems and/or between automation systems and programming devices or operator interface systems.

A remote communications partner (for example a programming device) can asynchronously access data areas of an M7 CPU/M7 FM/S7 CPU using the client communication functions. In a similar way, appropriate calls allow an M7 RMOS32 task to access the operand area of a remote M7 CPU/M7 FM/S7 CPU.

The communication functions can be subdivided into:

- Calls for non-configured connections allow an M7 RMOS32 task to access data on a remote M7 CPU/M7 FM/S7 CPU (single-sided communication functions) and/or to communicate with tasks or programs of another M7 CPU/M7 FM/S7 CPU (double-sided communication functions).

- Calls for configured connections (formerly known as PMC calls. Programmed Module Communication or PBK) allow an M7 RMOS32 task to access data on a remote M7 CPU/M7 FM/S7 CPU (single-sided communication functions) and/or to communicate with tasks or programs of another M7 CPU/M7 FM/S7 CPU (double-sided communication functions). Connections can be configured in the following subnet types: MPI/K Bus, Industrial Ethernet (TCP/IP) and PROFIBUS DP.

  It is also possible to interrogate the status of the communication partner and/or to send STOP, RESUME and START requests to the remote device.

- Sockets communication: Sockets are used to communicate with S7 nodes in Industrial Ethernet subnets. Heterogeneous TCP/IP communication is also supported.

The following client functions also require configured connections:

- Calls for the OMS: The M7 API calls for the OMS (Object Management System) allows you to handle the S7 objects on a remote automation system. S7 objects can be for example data blocks or function blocks.

  The M7 API calls allows you to carry out programming device functions such as copying, linking, deleting and uploading S7 objects directly from your user program.

- Calls for the operator interface system: M7 API calls for the operator interface allow you to implement you own the operator interface programs on the M7 automation computer.

  For example, the M7 API provides functions to read and write and/or cyclically read the variables of a remote automation system.

- Calls to read/set the system clock: These calls can be used to read and/or set the system clock of a remote server computer.

- Calls to the diagnostic server: The diagnostic server allows a program to register with the automation computer to receive diagnostic messages which have been generated by a remote automation system.

  User programs can also read out the system status list (SSL) and/or parts of the SSL (diagnostics buffer) from the communication partner. This allows events which have been logged via K bus to be read (cycle times, memory configuration, cycle time errors etc.) in order to react more precisely to malfunctions.

RMOS API Function Calls

4

Chapter Overview

4.1 Overview

The RMOS API (Application Programming Interface) is a C programming interface which provides M7 RMOS32 programs with all necessary functions to implement a multitasking system.

The RMOS API allows you to access all the systems services of the M7 RMOS32 kernel which are needed to subdivide your automation project into small easy to handle sub-functions (tasks). In addition to the task model, the RMOS API provides functions for exchanging messages between tasks, functions for processing interrupts and for managing buffer storage.

What is Described in this Chapter?

This chapter describes how to use each of the RMOS API function calls to solve different programming requirements and what you need to take account of with each call. The usage of the functions is illustrated by simple examples. The examples can be found on the system diskette in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

This chapter only gives a general overview of each function call and does **not** contain **a detailed description** for example of the parameters. This can be found in the Reference Manual.

4.2 General Notes on the RMOS API

Conventions and Header Files for M7 RMOS32 Programs

The names of the RMOS API calls are written in the Hungarian notation and always start with the letters **Rm**...

M7 RMOS32 programs must include the file **RMAPI.H**, which is the header file for the RMOS API function prototypes.

The data types and structure definitions for the RMOS API calls are contained in the header file **RMTYPES.H**.

General definitions, for example error codes, are contained in the header file **RMDEF.H**.

RMAPI.H includes the two files **RMTYPES.H** and **RMDEF.H** and thus only the header file **RMAPI.H** needs to be included in your programs explicitly.

All M7 RMOS32 programs must be created in Windows NT format (**32 bit flat** memory model). Only *near* addresses are used with this memory model, in other words pointers consist solely of a **32 bit offset**.

Caution

M7 RMOS32 applications are created in the FLAT memory model, meaning the system data and user data are located in the same logical address area. Please follow the directions in the Section 4.17 “Memory Protection”.

If you create your M7 RMOS32 program within M7-ProC/C++, the corresponding compiler and linker switches for the Borland C/C++ environment are set automatically.

Data Types of the RMOS API

In order to make it easier to port programs to other systems, the RMOS API environment uses its own type definitions instead of machine-dependent data type designators such as *int* or *long*.

The following table lists the data type designators used in the RMOS API environment. Their definitions are contained in the header file **RMTYPES.H**.

Table 4-1 Data Type Definitions in the RMOS API

Designator	Type Definition	Significance
uchar	unsigned char	unsigned character (value: 0 ... 255)
ushort	unsigned short	unsigned 16 bit integer (value: 0 ... 65 535)
uint	unsigned int	unsigned 32 bit integer (value: 0 ... $2^{32} – 1$)
ulong	unsigned long	unsigned 32 bit integer (value: 0 ... $2^{32} – 1$)
rmproc	void(*rmproc)(void)	pointer to function of type without calling and return parameters

Handling of Error Codes

In principle, any call of an RMOS API function can result in an error. Accordingly, RMOS API function calls include an **error code** in the *return value* which allows you to determine the success or failure of the call. The data type of the *return value* is ***int***.

Successful execution of an RMOS API call is normally indicated by the *return value* **RM_OK** (=0). RMOS API calls which have resulted in errors return **error codes** whose numerical value is a real **positive** number (> 0).

RMOS API calls can also *return values* which do **not** signal an error but instead contain additional **information** on the calling (parent) task. Such messages always have a **negative** value (< 0).

The Reference Manual contains a list of the possible *return values* for each of the RMOS API calls.

Note

For reasons of clarity and to save space, detailed error handling has been omitted from the examples in this manual and from the descriptions of each of the function calls. Nonetheless, your programs should always be designed to react appropriately to any error codes which are returned.

4.3 Structure of an M7 RMOS32 Program

Before your C program can use the services of the M7 RMOS32 runtime system and proceed to carry out the automation assignment, certain steps are required in an initialization section of the program. These steps, which are mainly dependent on the type of program, include not only the initialization of the M7 API runtime system but also the creation of the tasks required within the C user program.

This section describes the steps which are required to create tasks. A small example at the end of the section is used to give you an overview of the initialization section of a typical C program.

Structure of a C User Program

A number of conventions must be followed when structuring a C user program for execution under M7 RMOS32 in order to ensure that the M7 RMOS32 system functions correctly. The following figure shows the general task structure of an M7 RMOS32 user program.

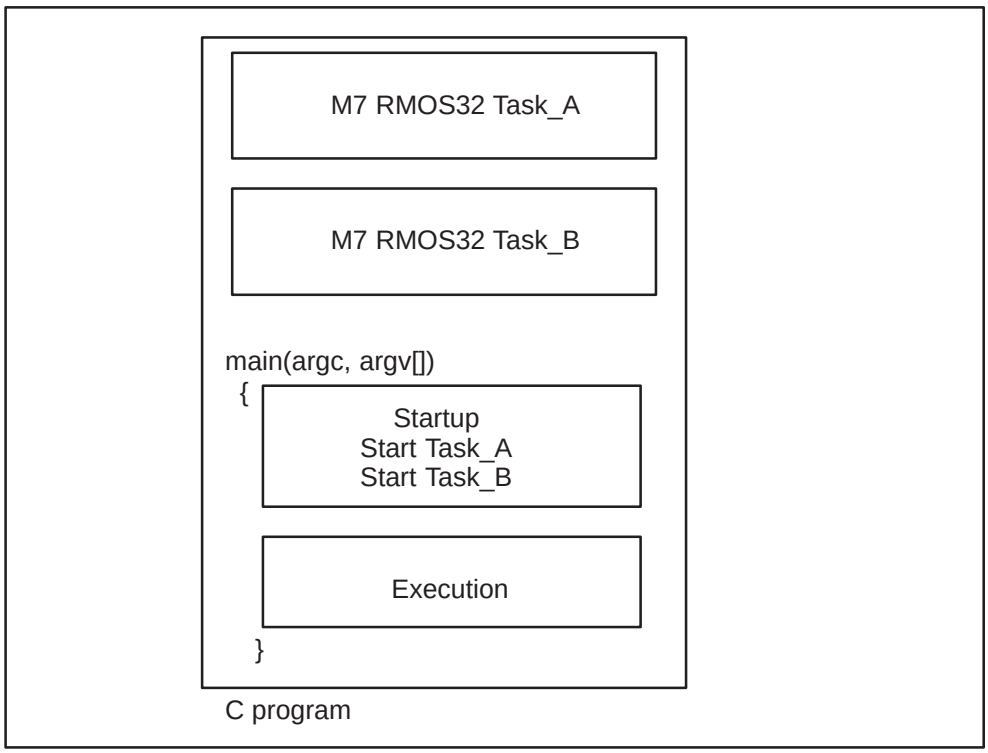


Figure 4-1 General Structure of an M7 RMOS32 User Program

The above example shows a program which consists of three separate tasks (task_A, task_B and task “main”).

Creating a Task

In order to create a task within a C program, you must specify the information required by the RMOS API to manage the task in the initialization section of the main function.

Specifying the Program Code Section

Each task in the system naturally has its own program code section which contains the task-specific program instructions.

In principle, any C function within a C program can be compiled as a self-contained task, provided that it has the following type:

void FunctionName(void)

in other words it does not expect any calling parameters and does not generate a *return value*.

The entry point of the function (***pTaskEntry*** in the following), in other words its address, must be specified as a parameter when registering the task with the RMOS API.

An exception to this rule is the **main()** function, which always executes under M7 RMOS32 as a self-contained task. As with any standard C main function, it is called with the command line parameter which was parsed by the command line interpreter CLI.

A program, in other words the “main task” of the associated C program, can be started automatically on the M7 automation computer by making an appropriate entry in the file CLISTART.BAT.

Assigning the Task Stack

In order to satisfy the requirements of independent and simultaneous execution of several tasks, each task needs its own stack area in addition to its own program code.

Each task requires its own stack, because the stack is used to save the task’s data (processor registers etc.) when the task is swapped out. Furthermore, the task stack is also used to save local variables and data structures and the jump addresses when calling subprograms from within the task.

The stack area is assigned automatically by the RMOS API when creating the task. During registration, the required stack size must be specified to the RMOS API in a calling parameter (***TaskStackSize*** in the example).

Note

You must calculate the required stack size from the memory requirements of the local C variables (care with recursive calls) and calling parameters. The stack size is also dependent on whether you want to use functions from the C runtime library.

For each user task a system stack of 3 Kbyte is assigned additionally to the user stack. The size of the user stack is determined by the ***TaskStackSize*** parameter.

The C function ***printf()*** requires 256 words =1 Kbyte. If you call ***printf()***, the stack size should be at least 256 words (machine words = 32 bit) plus the stack size required by the program.

Caution, stack overflow

If the size defined for the stack is too small, a stack overflow (page fault exception) may occur. In this case the event 2x50, ZI1, ZI2/3 and the task ID are entered to the diagnostics buffer, the task is blocked and the STOP mode is required. No signal handler can be executed at this stage, since the task cannot execute any code at all. In order to react appropriately to such events a second task must be programmed.

Specifying the Priority of the Task

The priority of each task specifies the importance of the job which the task is assigned to carry out. A higher priority task gains access to the CPU as soon as it is ready for execution; a currently executing lower priority task is interrupted.

The priority can be set to a value between 10 and 169 and between 200 and 240, whereby 10 is the lowest priority and 240 is the highest priority.

When creating the task, the task priority is specified to the RMOS API as a calling parameter (***Priority*** in the example). Please refer too to section 4.5.

Specifying the Task Name

The RMOS API registers each of the tasks under a unique task name in the internal resource catalog. Tasks in different C programs (different address space!) use the task name to access task attributes such as the task ID.

When creating the task, the call can be specified with a task name ***Taskname*** to register the task in the resource catalog. If the name is a NULL pointer, no entry is made in the resource catalog.

Creating a Task with the RMOS API

A task must be explicitly created with one of the two following RMOS API calls:

RmCreateTask(pTaskName,TaskStackSize,Priority,TaskEntry,pTaskId)

or

RmCreateChildTask(pTaskName,TaskStackSize,Priority,TaskEntry, pTaskId)

The RMOS API then reserves the required stack area and enters the relevant information in the internal task management table. The task is now known to the system with all the necessary components, for example code area, stack etc. However, the task status (see Section 4.4) is initially DORMANT, in other words it does not yet compete for processor time. The task status can then be changed to READY with the appropriate RMOS API call.

If the task is created with the ***RmCreateChildTask()*** call, then it (in other words the child task) automatically inherits the console, the current working directory and the environment from the calling task (parent).

Storing the Task ID

When you create the task, the RMOS API returns the task ID number in a pointer (***pTaskId***). The task ID is required by further RMOS API calls.

For this reason, you should store the IDs of each of the tasks centrally, for example in the global variables of your C programs, so that each task within your program has access to the IDs of the other tasks.

Starting the Task

After the task has been created with the RMOS API call ***RmCreateTask()***, it can be started by a further RMOS API call, in other words its status can be changed to READY. The only tasks that compete through the scheduler for processor time are those with the status READY and the task which is currently ACTIVE.

The following RMOS API call is used to start a task:

RmStartTask(..,TaskID,Priority,..)

A task can also be started with the following call:

RmQueueStartTask(..,TaskID,Priority,..)

The difference between ***RmStartTask()*** and the second call is that the latter enters the start request into an internal system queue which is executed as soon as the task reaches the DORMANT status.

For example, if the task to be started already has READY status, a further ***RmStartTask()*** call has no effect. In contrast, the ***RmQueueStartTask()*** call causes the task to start again automatically as soon as it terminates, in other words has reached the DORMANT status again.

However, if the task to be started is already in the DORMANT status, then ***RmQueueStartTask()*** is identical to ***RmStartTask()***.

The ***Priority*** parameter is used to specify a priority for the task to be started. If Priority is specified as **RM_TCDPRI**, the task to be started is assigned the priority which was originally specified with ***RmCreateTask()*** during task creation.

Starting of tasks is also a common way of coordinating and/or synchronizing tasks (see section 4.9).

Passing Parameters when Starting a Task

When starting a task with the RMOS API call ***RmStartTask()*** or ***RmQueueStartTask()***, the calling task (parent) can pass two initial values in processor registers to the task to be started.

This is done by specifying the two following 32 bit parameters of type **uint** in the calls:

RmStartTask(..,RegVal1,RegVal2) or
RmQueueStartTask (..,RegVal1,RegVal2)

The task which is started can access the parameters with the following functions:

getdword() / ***getparam()*** or ***get2ndparm()***

Since high level languages like C use the processor registers and thus destroy the original contents, ***getdword()*** or ***getparam()*** must be the first statement in the program code of the task which is started. If a second parameter is passed, then ***get2ndparm()*** must be the second statement, behind ***getdword()*** or ***getparam()***.

Default Environment of the Main Task

The main task of your program cannot create itself with an RMOS API call. Accordingly, “main” is created by the command line interpreter CLI.

CLI uses the following default values for each main task:

- ***TaskName***
  The main Task is entered into the resource catalog with the name CLI_JOB_X together with a task ID which is assigned automatically by CLI. *X* is the job number which is also assigned by CLI.
- ***TaskStackSize***
  The stack is set to the size which was specified in the link information.
- ***Priority***
  By default, CLI sets the priority of the main task to 64. If the main task should execute with a different priority, then the priority must be changed during execution by using the corresponding RMOS API call (see section 4.5).
- ***TaskEntry***
  CLI automatically sets the entry point for the main task to the start address of the main function.
- ***pTaskId***
  The task ID of the main task is assigned automatically by CLI. The main task can request its task ID from the operating system with the following RMOS API call.

Interrogate Task ID

The RMOS API provides the following function call to get the task ID of the current task:

RmGetGetTaskID(RM_OWN_TASK,pTaskID)

The first parameter must be specified as the constant ***RM_OWN_TASK***; the required task ID is returned in the parameter ***pTaskID***.

Example

An example program of a startup section is contained in the file anlauf.c in the directory ..\M7SYSx.yy\EXAMPLES\M7API. It contains the framework of the main task for the example program. It contains only those calls which are relevant for setting up, starting and terminating a task within a C program.

4.4 General Information on the M7 RMOS32 Multitasking Model

Multitasking describes the ability of an operating system to apparently allow execution of several programs (tasks) simultaneously. Under M7 RMOS32, this is implemented by a special task-switching mechanism which is called the **scheduler** in the following.

The scheduler decides which task gets access to the processor and when, and it also decides when the task must relinquish the processor again to another task. The scheduler makes sure that the assignment of processor time takes place in accordance with the current priority of each of the tasks. This ensures that the most important jobs are given a higher priority and the less important jobs are interrupted as necessary (preemptive multitasking).

If a task with a higher priority than the currently ACTIVE task becomes ready to execute (READY status) for example due to an event, the active task is interrupted and the higher priority task is executed instead. The scheduler only allows the interrupted task to resume processing when none of the higher priority tasks are currently READY.

Figure 4-2 shows the task statuses and their admissible transitions. Task status transition takes place as a result of RMOS API calls or other events.

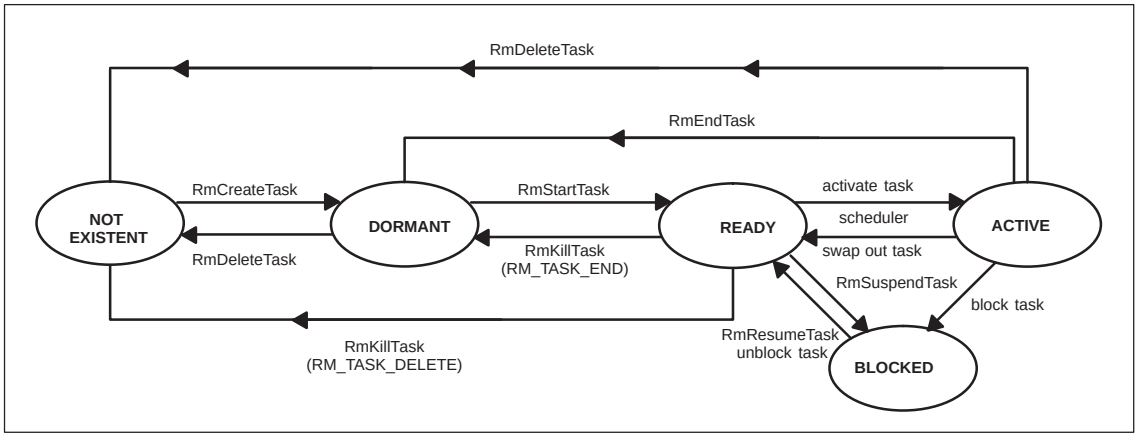


Figure 4-2 Task Statuses and Admissible Status Transitions

Tasks Statuses

In order to ensure the smooth processing of a number of tasks in a multitasking system, the tasks need to be managed by a scheduler, whereby the **status** of each task plays an important role. Since it is not possible to assign more than one task to the processor at once, each of the tasks can be in one of five statuses: ACTIVE, READY, BLOCKED, DORMANT or NOT EXISTENT status.

ACTIVE

The task has been assigned to the processor and the program code is currently being executed. However, the task may need to be interrupted at any time by an READY task with a higher priority.

In a single-processor system such as an M7 automation computer, only one of the tasks can have the ACTIVE status.

Tasks can only reach the ACTIVE status through the READY status and only on request of the scheduler.

READY

The task is not executing and is waiting to be assigned to the processor. If a task is in the READY status, this means that another task of equal or higher priority has already been assigned to the processor.

There are several ways to change the task status to READY:

- The task status can be changed from DORMANT to READY if another active task issues the following RMOS API call:

  RmStartTask()

- If a task is waiting for an event (BLOCKED status), it changes automatically to the READY status when the event occurs. If it is now the highest priority READY task, it will then be assigned to the processor by the scheduler (ACTIVE status).
- If a task is waiting for an event (BLOCKED status), another task can change its status from BLOCKED to READY with following RMOS API call:

  RmActivateTask(TaskID)

- The scheduler changes the task status from ACTIVE to READY. This is done if a higher priority task has become READY.
- The waiting time of a task whose status was changed to BLOCKED with the call ***RmPauseTask(Time)*** can be prematurely ended with the following RMOS API call:

  RmResumeTask()

  The task status also changes from BLOCKED to READY without external influence if the waiting time has expired.

BLOCKED

The task is waiting for an event (for example for a specified time interval to elapse). The task status is changed to READY again by the scheduler when the event occurs.

There are several ways to change the task status to BLOCKED:

- The task itself issues an RMOS API with the wait option. Typical calls are:

  RmReceiveMail()

  RmReadMessage()

  RmGetFlag()

  RmGetBinSemaphore()

- The task changes its own status to BLOCKED with the following call:

  RmPauseTask(Time)

  The task goes back to READY status again when the time has expired. In this case, the task resumes processing at the statement directly after the above RMOS API call.

- The task terminates itself with the following call:

  RmRestartTask(..,Time)

  This causes the task status to change to READY after the time has expired. Processing starts at the beginning of the task.

- The following RMOS API call causes the task status to change from READY to BLOCKED:

  RmSuspendTask()

  A task can also use this call to change its own status to BLOCKED.

DORMANT

A task with DORMANT status is registered with the operating system with all necessary components such as code area, stack etc. and can be started, in other words moved to the READY status, with the RMOS API call ***RmStartTask()***.

The following ways are available to change the task status to DORMANT:

- A task can be created with the following RMOS API call:

  RmCreateTask()

  It is automatically created in the DORMANT status. The function which belongs to the task must already have been uploaded into main memory using the CLI.

  A task which has been created in this way can then be changed to READY status with the call ***RmStartTask()***.

- The following call can be used to terminate a task, in other words change its status from ACTIVE to DORMANT:

  RmEndTask()

  The task still remains in the main memory with its code and data areas.

NOT EXISTENT

The task is not (yet) known to the system, in other words it is not entered in the task management table of the M7 RMOS32 kernel. A task with this status can be for example a file on the mass storage or it may have already been loaded into main memory through the CLI.

The task status can be changed to NOT EXISTENT with the following RMOS API call:

RmDeleteTask()

This call is only executed successfully if the task to be deleted either has the status DORMANT or ACTIVE. In all other cases the call will be rejected with an error message.

If the call is successful, the task management information is deleted from the internal task table and the memory assigned to the task is released.

Interrogating Task Status

The current task status can be requested by other tasks, and by the task itself, with the following RMOS API call:

RmGetTaskState(TaskID,pTaskState)

The current status of the specified task is returned in the parameter ***pTaskState***. The status of the task which issues this call can naturally only be ACTIVE.

Disable and/or Enable Scheduler

Within a task you are able to disable the scheduling mechanism of M7 RMOS32 in order to reserve the processor solely for the ACTIVE task. This is done with the following call:

RmDisableScheduler(void)

If the scheduling mechanism is deactivated, the only task that remains ACTIVE is the one that used the call. Other tasks, even those of a higher priority, do not get any CPU time assignment.

None of the RMOS API or M7 API calls are processed when the scheduling mechanism is deactivated. They are stored in a FIFO queue instead (see section 4.16) and only processed when the scheduling is reactivated again.

Accordingly, the calls ***RmDeleteTask()*** and ***RmStartTask()*** are not allowed with the scheduling deactivated. In addition, with the scheduling deactivated it is recommended to avoid all RMOS API calls where the task may have to wait for another task to run.

Caution

If the scheduling is deactivated for too long a period, the real-time ability of the system can be impaired. Accordingly, the scheduling should only be deactivated by tasks in the initialization phase and/or for emergency stop routines in cases where serious errors have taken place.

The scheduling can be activated again with the following call:

RmEnableScheduler(void)

4.5 Specifying the Task Priority

Priority of a Task

The priority of a task defines the importance of the job which the task carries out. The scheduler always assigns processing time to the task with READY status which has the highest priority.

Each task can be assigned a **priority** either during creation or during runtime. This allows the programmer to specify which of several competing tasks is more important.

If an event takes place while a task is running which causes another task with a higher priority to change to the READY status, the running lower priority task is interrupted. The higher priority task is then assigned to the processor and changes to the ACTIVE status.

Figure 4-3 is an example of the status changes for three competing tasks.

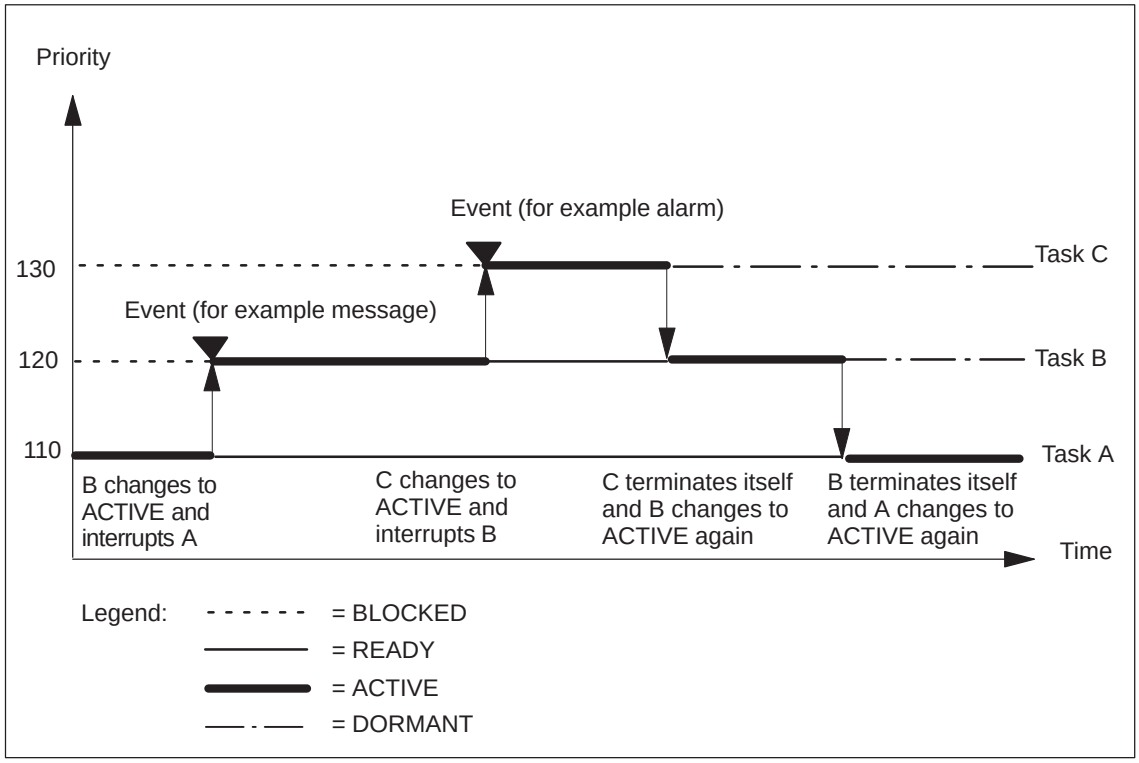


Figure 4-3 Nested Processing of Three Tasks

Specifying Task Priority

In order for a multitasking program to correctly carry out its assigned purpose, the priorities, in other words the importance, of each of the tasks must be correctly specified. Accordingly, please note the following facts:

- A task with a higher priority can interrupt the processing of a task with a lower priority.
- Tasks with the same priority are assigned processor time according to the time slice principle: All tasks with the same priority are assigned time slices of 10 ms for processing. After the time slice has expired, the ACTIVE task is interrupted and another task with the same priority is changed to ACTIVE status.

When creating the task, the task priority is specified to the RMOS API as a calling parameter (***Priority*** in the example).

The priority can be set to a value between 10 and 169 and between 200 and 240, whereby 10 is the lowest priority and 240 is the highest priority.

Priority	Use
0 to 9	Reserved for system services
10 to 169	For user tasks
170 to 199	Reserved for system services
200 to 240	For user tasks
241 to 255	Reserved for system services

The following basic rule should be followed when deciding the priorities:

- Time critical tasks with short reaction times must have a higher priority and a short processing time (200 to 240).
- Tasks whose reaction time is less important can be assigned a medium priority and a longer processing time (100 to 170)
- Background tasks (for example tasks to evaluate large amounts of data) should be given the lowest priority. Their processing time can be as long as required (10 to 99).

Note

You should carefully choose the priorities of each of the tasks since the smooth operation of your multitasking program can be affected adversely by incorrect priority assignment.

Use the range 100 to 169 for higher priority control tasks and use the range from 10 to 99 for lower priority activities.

Controlling the Processing Sequence

In a multitasking program, it is possible to control the sequence of processing of each of the tasks, in other words the sequence of processing the data, by setting the priorities appropriately.

Figure 4-4 shows an example for controlling the processing sequence of two tasks:

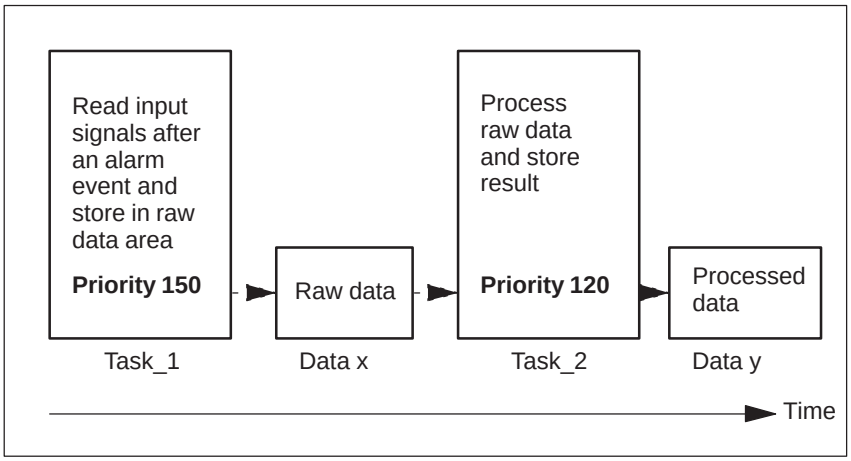


Figure 4-4 Controlling the Processing Sequence of Two Tasks Using Priority

In this example, the two tasks are started simultaneously, in other words moved to the READY status, on receiving an alarm event.

- Since the CPU time is assigned according to priorities, Task _1 is assigned to the processor first. The required signals are read in completely and stored.
- Task_2 is only assigned to the processor when Task_1 is finished. Task_2 can now finish processing the raw data.

In order to ensure correct functioning of this step-by-step processing, it is necessary to ensure that the raw data is completely processed by Task_2 before a new alarm is received and the data to be processed is overwritten by Task_1 with new data.

Note

The maximum processing duration can only be determined to any accuracy for the highest priority tasks. The overall processing time may be very difficult to determine for low priority tasks (for example for Task A in Figure 4-3) if lots of asynchronous alarm events occur.

Pay Attention to Interrupts

When programming a task you should be aware that a task can be interrupted by a higher priority task or an interrupt routine **after each assembler statement** without prior warning!

Data **local** to the task is not affected by the interruption because it is saved on the stack. However, **shared data** (for example global variables) may become overwritten by or incorrectly evaluated by a higher priority task during an interrupt.

Changing the Priority During Runtime

In some cases it may be necessary during program execution to change the priority which was initially assigned to the task when it was created. For example, in some cases it may be necessary to dynamically increase the priority of a task, for example a visualization task which normally has a relatively low priority, due to an interactive user request. A priority change can be used for example to increase the frequency with which the display of particular process data is updated.

One way to do this is to arrange that the task that receives the user request is responsible for increasing the priority of the visualization task. When the faster updating of the process data display is no longer required, the priority of the visualization task can be reduced back to its original value.

Proceed as follows in order to implement these steps in your program:

1. Determine the task ID of the corresponding task. The task ID is returned by the RMOS API when the task is created and can be stored in global variables. Alternatively, (this is the only way to find out the ID of the main task) you can get the task ID with the following function call:

   RmGetTaskID(...,pTaskID)

2. Get the original priority of the task with the following call:

   RmGetTaskPriority(TaskID,pPriorityOld)

3. Set the priority of the task to the new value you require with the call:

   RmSetTaskPriority(TaskID,PriorityNew)

   The task can now execute the required statements with the new priority.

4. When the increased priority is no longer required, you can go back to the original priority with the following call:

   RmSetTaskPriority(TaskID,PriorityOld)

Delay Task Processing

In some cases the reverse effect is required, in other words processing of the task must be delayed for example to allow the processing of a signal by an external device. In this case you can change the task status to BLOCKED with the following call in order to pause task execution for a specified time Time:

RmPauseTask(Time)

After the time has expired, the task resumes processing with the first statement after the ***RmPauseTask()*** statement.

The parameter Time specifies the time in ms. The type of this parameter is ulong and it can accept values from 1..RM_MAXTIME.

If larger delays are required, the parameter time can be set with the following macros:

RM_SECOND(sec) 1..60
(waiting time in seconds)

RM_MINUTE(min) 1..71582 (waiting time in minutes)

RM_HOUR(hour) 1..1193 (waiting time in hours)

The macros for seconds, minutes, and hours can also be combined by addition:

RM_SECOND(sec) + RM_MINUTE(min) + RM_HOUR(hour)

Task suspension can also be ended prematurely by another task before the time has expired with the following RMOS API call:

RmResumeTask(TaskID) mode

This causes the task status to change from BLOCKED to READY, whereupon it once again competes for processor time.

4.6 Terminating Tasks

A task can be terminated in several ways. Depending on the further use for the task, the task can simply be put “on hold” for a specified time. Alternatively, the task can be deleted and the assigned memory can be released for other purposes.

Temporary Termination of Tasks

The RMOS API includes a function which can be used by a task to terminate itself for a specified time. This variant is used when the task is needed again to carry out the same function again at a later time.

An example of such repeating actions include the saving of temporary data in the main memory to mass storage, or the transfer of data to and/or from a remote supervisory computer.

A task can terminate itself temporarily with the following RMOS API call:

RmRestartTask(Mode,Time)

The task changes its status with the above call from ACTIVE to BLOCKED, in other words the task remains in main memory with its code and data areas.

The ***Mode*** parameter is used to specify whether the time interval ***Time*** should refer to the last transition to the READY status or to the current system time.

After the time has expired the task status is automatically changed to READY and the task then competes for processor time again. When it reaches the ACTIVE status, task processing starts from the beginning again.

Terminating Tasks without Deleting Them

In a similar way to the temporary termination of a task, the task can terminated itself without being deleting from the task management. The task changes its status with the following call from ACTIVE to DORMANT:

RmEndTask(void)

In contrast to the time-limited termination, the task can now no longer activate itself again without outside help. The task status can be changed from DORMANT to READY by another task with the call ***RmStartTask()***.

This variant is used where repeating actions need to be carried out which are triggered by other tasks.

Deleting Tasks

A task in the DORMANT (or READY) status can be deleted by another task (or itself) and removed from the main memory with the following call:

RmDeleteTask(TaskID)

This variant is used in particular for tasks whose purpose is solely the initialization of I/O devices or internal data structures. Such tasks can be deleted after initialization in order to release the main memory that they use.

The call ***RmDeleteTask()*** may only be applied to tasks that have the status ACTIVE (in other words the task deletes itself) or READY or DORMANT.

The call ***RmDeleteTask()*** can not be applied to tasks with the BLOCKED status.

Terminating Tasks with RmKillTask()

In order to be able to terminate and/or delete tasks with the BLOCKED status, or other tasks which may have gone wrong in some way, RMOS API provides the following call:

RmKillTask(Mode,TaskID)

This call changes the status of any task (even the calling or parent task) to DORMANT or NOT EXISTENT, depending on the status that it had previously.

The ***Mode*** parameter is used to specify whether the task specified with ***TaskID*** (RM_OWN_TASK=own task) should only be terminated (Mode=RM_TASK_END) or deleted, too (Mode=RM_TASK_DELETE).

Special situations may arise when the specified task has the status BLOCKED (see reference manual). The ***RmKillTask()*** call is inadmissible in the following situations and will be rejected with an error message:

- Debugger breakpoint
- All runtime errors (for example division error),
- Terminate/delete with the ***RmKillTask()*** call was already applied to a task which is waiting for an I/O job to finish (in other words the ***RmKillTask()*** call was applied twice to the same task).

The call ***RmKillTask()*** should only be used by special “monitoring tasks” during program development and/or test in order for example to terminate tasks which have gone wrong in some way.

In a fully tested project, only the calls ***RmEndTask()*** and ***RmDeleteTask()*** should be used.

Note

Resources such as memory pools, mailboxes or semaphores which are still in the ownership of the task are not automatically released after the task is terminated or deleted.

These resources must be released by another task, if possible; otherwise they are no longer accessible for use by other tasks or programs.

Warning

A task must not terminate itself with the statement "**}**" (curly braces=end of function) or **return**, since otherwise unpredictable errors in the system operation may occur.

4.7 Resource Catalog

Which Resources are Entered in the Catalog?

The RMOS API also manages resources, in other words those software aids which the program and/or the system needs to satisfy its assignment. Resources play a central role, since nearly all RMOS API calls access them in some way or another.

A resource is referred to as static or dynamic depending on whether it was registered with the operating system during software configuration (in other words when initializing a data structure) or by an RMOS API call during runtime, respectively.

Access to the resource takes place solely through the resource ID, which is automatically assigned by the system during software configuration (static) and/or when creating the resource during runtime (dynamic).

All resources are automatically entered by the system into the so-called resource catalog together with their resource ID (***ID***), an optional extended ID (***IDEx***), the resource type (***Type***) and an additional symbolic name (***pName***).

Resources which are entered in the resource catalog include:

- Tasks
- Semaphores
- Event flags
- Message queues
- Mail boxes
- Memory pools
- User defined resources

Advantages of the Resource Catalog

The use of symbolic names within the catalog has the advantage above all that tasks that want to access the resource can request the required resource ID through the symbolic resource name.

In particular this means that during the program development phase, only the name and the type of the resource needs to be known but not its ID. This allows a higher degree of independence between resource configuration and user task programming as would be the case if include files were used for this purpose.

Creating Resources

You can create dynamic resources from within a task during runtime by using appropriate RMOS API calls. The general structure of the call is as follows:

RmCreate...(..,pResourceName,pResourceID)

The call can be specified with a resource name ***pResourceName*** (max. 15 characters + \0) to register the resource in the resource catalog. If this parameter is specified as a NULL pointer, no entry is made in the resource catalog. The RMOS API returns the resource ID in ***pResourceID***.

Note

It is recommended to store the returned resource ID in the global variables of your C program in order to make it easier for other tasks in the C program to get access to the resource.

Deleting Resources

Resources created during runtime can be deleted again with the corresponding RMOS API call and thus removed from the resource catalog. The general structure of the call is as follows:

RmDelete...(ResourceID)

This call must be specified with the associated resource ID.

User-Defined Catalog Entries

In addition to the automatic cataloging of resources when they are created, you can also generate your “own” resources and catalog them accordingly. The structure of your own resources (for example monitors, ring memory etc.) from resources which are provided by the system can be tailored to your own requirements.

RMOS API provides the following call to catalog user defined resources:

RmCatalog(Type,ID,IDEx,pName)

The RMOS API enters the specified parameter of type User in the resource catalog.

The following call lets you delete the entry again:

RmUncatalog(pName)

This call deletes the entry with the name ***pName*** from the resource catalog.

Note

The call ***RmCatalog()*** can also be used by a task in order to make any required entry in the resource catalog for information purposes.

This information is then available to all tasks in the system.

Getting Information on Catalog Entries

In order to access resources with an RMOS API call, it is necessary to know the associated resource ID in advance.

If the task which created the resource and the task which needs to access the resource are not within the same C program (different address spaces), then the resource ID cannot be exchanged with the help of global variables.

For such cases the RMOS API provides three calls which you can use to evaluate entries in the resource catalog:

RmGetEntry(..,pName,pEntry)

This call must be specified with the resource name ***pName***. The RMOS API then returns the complete catalog entry in the pointer ***pEntry*** to the structure variable of type ***RmEntryStruct***.

The structure is defined as follows:

```
typedef struct _RmEntryStruct
  {
     uchar   slen;              /* Length of the name */
     char    string[16];        /* Resource name */
     uchar   type;              /* Resource type */
     ulong   ide;               /* Extended ID */
     ushort   id;               /* Resource ID /*
  } RmEntryStruct
```

The following call works the other way round:

RmGetName(Type,ID,IDEx,pName)

On specifying the resource type ***Type***, the resource ***ID*** and/or the extended ID ***IDEx***, this call returns a pointer to the name ***pName*** of the resource.

RmList(Type,Count,pIndex,pNumEntries,pEntry)

The above call returns a specified number ***Count*** of entries of a particular resource type ***Type*** into a previously defined buffer ***pEntry***.

The parameter ***pIndex*** points to a word which contains an internal reference. It is used as the input and output parameter. ***pIndex*** must be initialized with 0 before the first call and must be used without modification for following calls. The end of the resource catalog has been reached when the actual number of entries read ***pNumEntries*** is smaller than the requested number ***Count***.

Example

An example program for reading entries from the resource catalog is contained in the files *mem_a.c* and *mem_b.c* in the directory ...\M7SYSx.yy\EXAMPLES\M7API.

4.8 General Information on Task Coordination, Synchronization and Communication

In order to ensure that the complex nested flow of a multitasking system executes correctly and smoothly, efficient synchronization and communication mechanisms between each of the tasks and/or between tasks and the operating system are essential.

Any multitasking system must provide facilities to enable tasks to wait for other tasks. Accordingly, a task must be able to notify others that a particular function has been completed in order to allow another waiting task to resume processing.

A notification or messaging system of this type needs to provide new, efficient ways for tasks to wait for events without wasting valuable processor time on the overhead.

In addition, the communication concept must allow tasks to influence each other as planned while offering the highest level of system security.

Task Communication

Task communication is normally used to describe all mechanisms with which a task can influence one or more other tasks and these tasks in turn can respond to the original task. Task communication can be subdivided further into:

- Mechanisms to synchronize and coordinate
- Mechanisms for the purpose of communication only

Note

The exchange of user data between tasks by direct access to the data area of another task is allowed for performance reasons.

Synchronization and Coordination

The processes which take place within a multitasking system are dependent above all on the arrival or occurrence of various events, from the results of previous processing steps and from access to common resources. To ensure that the overall system remains under control, the processing of each of the tasks must be coordinated with the other tasks.

What needs to be coordinated?

- Task execution must wait for the availability of input data,
- Task execution must wait for the arrival of events,
- The access to common data must be coordinated.

The simplest form of communication is called synchronization. This is the ability of a task to pause its own execution until another task has completed a particular function or a particular event has occurred.

Tasks must be synchronized with each other if they want to access common resources, for example global data resources, mass storage or I/O devices. The result is generally unusable if two tasks attempt to change data or output data simultaneously.

Mechanisms to synchronize and coordinate tasks are generally considered to be measures which influence the status of tasks.

Communication

Communication is generally considered to be the transfer of messages to other tasks.

During communication, you can differentiate between the sender and the receiver of a message. The sender writes the message and the receiver reads it.

The communication of tasks with each other is always associated with coordination with respect to time.

Examples of Task Communication

The necessity and the type of task communication is usually determined by the automation assignment to be solved.

Figure 4-5 illustrates the subdivision of the automation assignment into separate tasks.

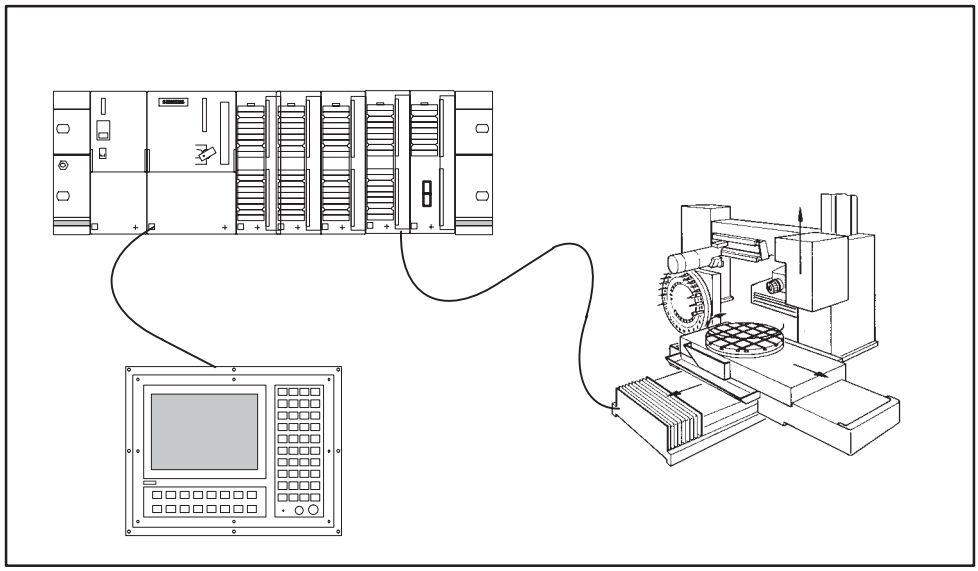


Figure 4-5 Subdivision of the Automation Assignment into Separate Tasks

An intelligent machine control can be implemented by two tasks. One task controls a user terminal and processes user entries and the second task controls the machine.

Each of the tasks needs to transfer information to the other task. The control task needs to know the parameters required for controlling the the machine and the operating task needs to know the status of the machine.

A subdivision of the overall automation assignment between two tasks is advantageous for two reasons:

- The control process must not be influenced (impaired or delayed) by user entries or the need to process user data. Both tasks must be able to carry out their respective functions asynchronously.
- The second reason is to simplify maintenance of the software. One of the tasks only contains the program code for machine control and the other one only needs to deal with the operator panel. Both tasks communicate via a clearly defined interface. Accordingly, the two tasks can be developed by different programmers.

Communication and Synchronization Mechanisms in the RMOS API

The RMOS API provides various mechanisms for communication and synchronization of tasks which you can use according to the problem to be solved.

The following mechanisms are available for task synchronization and communication:

- Coordination by starting another task
- Coordination via semaphores
- Coordination via event flags
- Communication via message queues
- Communication via mailboxes

These mechanisms can be differentiated mainly by the quantity of information which is transferred, by the speed, by complexity and by the intended application.

4.9 Coordinating Tasks by Starting Other Tasks

Coordination by Starting Another Task

This communication mechanism is closely connected with task management itself. A task can be started by another task with the following RMOS API call (see section 4.3):

RmStartTask() or ***RmQueueStartTask()***

Both RMOS API calls also have a wait option ,which causes the calling task (parent) to be halted (synchronous call). The calling task is only changed to the READY status again when the called task (child) has finished. This mechanism also allows a mutual communication and/or synchronization between tasks.

Another synchronization method is for example to interrogate the task status and to issue the call ***RmStartTask()*** only when the other task has the DORMANT status (see Figure 4-6).

The data transfer itself between the calling (parent) and the called (child) task can be implemented with global variables, for example. However, it is important to remember that parallel access to shared data can be unreliable (see semaphores), since the pseudo-parallel processing of tasks and their mutual influence on each other can be very complex.

Example

The following figure shows a simple example for the coordination of tasks by starting another task and then waiting for it to finish.

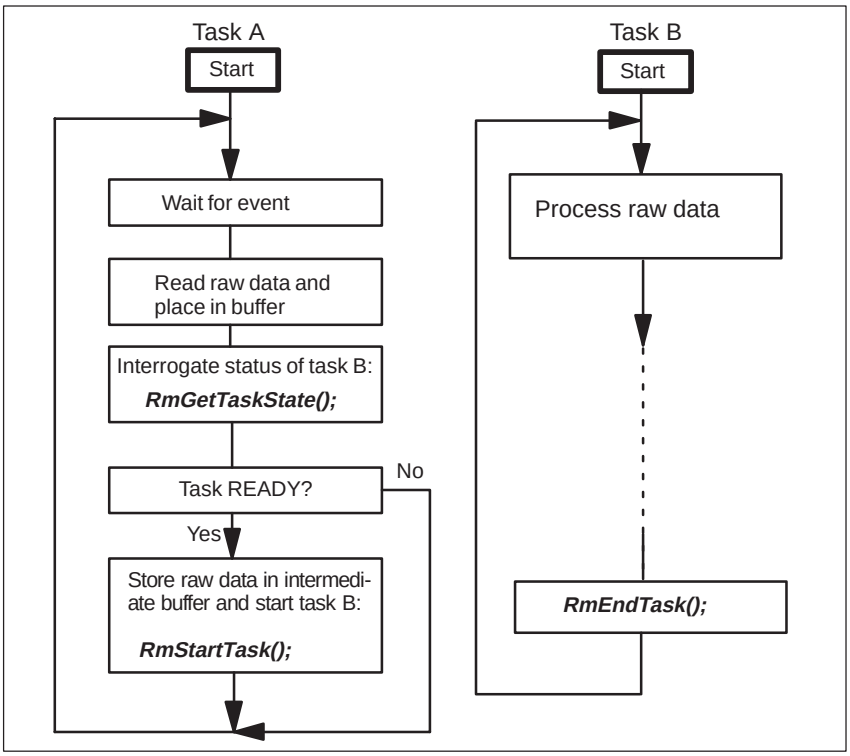


Figure 4-6 Example of Task Coordination Using Task Status

Coordination by Resuming Execution

Closely connected with coordinating tasks by starting another task is the mechanism of resuming the execution of a task on request of another task.

A task can change its status to BLOCKED for a specified time with the RMOS API calls ***RmPauseTask()*** and ***RmRestartTask()*** (see sections 4.5 and 4.6). The waiting period can be expired prematurely with the following call issued by another task:

RmResumeTask(TaskID)

The status of the waiting task then changes to READY.

This method can be used to implement synchronization if one task issues the call ***RmPauseTask()*** or ***RmRestartTask()*** and another task issues the call ***RmResumeTask()***.

Example

The example program *koord_1.c* provides the framework for coordinating tasks in accordance with the example in Figure 4-6. The following processing in Task_B is only indicated schematically. The example program is contained in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

4.10 Coordinating Tasks with Event Flags

Coordination with Flag Groups

An event flag, abbreviated to flag, is a data structure which is implemented by a single bit in memory. In a similar way to semaphores, flags can be used to coordinate various tasks. Flags are arranged in flag groups of 32 bits which are associated with their own resource IDs. Each task has a local flag group which cannot be accessed by other tasks. The local flag group of a task has the ID=0.

Typically, an event flag is set by a task or interrupt routine when a particular condition has been satisfied or an event (interrupt routine) has occurred. A flag represents a binary message with a predefined significance which is determined by the application.

Flags offer an elegant and efficient way to exchange binary messages.

Flag Groups

Flag groups are created during runtime and can be manipulated by all tasks in the system. Flag groups are created with the following call:

RmCreateFlagGrp(pFlagGrpName,pFlagGrpID)

The call can be specified with a flag group name ***pFlagGrpName*** to register the flag group in the resource catalog. If the name is a NULL pointer, no entry is made in the resource catalog.

The RMOS API returns the flag group ID in ***pFlagGrpID***. The ID must be specified in further calls.

The following call is used to delete the flag group again and to release the assigned memory:

RmDeleteFlagGrp(FlagGrpID)

Operations with Flag Groups

The following RMOS API calls can be used to implement coordination with the help of flags:

- ***RmSetFlag(..,FlagMask)*** or ***RmSetFlagDelayed(Time,..,FlagMask)***

  This call is used to set individual flags within a flag group. You must specify the flags in the flag group which you want to set with the 32 bit parameter ***FlagMask***.

  The call ***RmSetFlagDelayed ()*** is used to set the flags after a specified time has elapsed.

- ***RmResetFlag(..,FlagMask)***

  This call is used to reset individual flags within a flag group. You must specify the flags in the flag group which you want to reset with the 32 bit parameter ***FlagMask***.

- ***RmGetFlag(Time,Type,FlagGrpID,TestMask,pFlagMask)***

  This call is used to test the flags ***TestMask*** in the flag group ***FlagGrpID***. The ***Type*** parameter is used to specify which of the two following variants of this call you want to use:

  – Test whether all specified bits are set

  – Test whether at least one bit is set

  If you specify a waiting period ***Time***, the call waits for the bits to be set in other words the task is initially changed to the status BLOCKED. The task is only changed to the status READY again either if the specified bits have been set or the specified time has expired.

  The bits in the flag group are combined logically with the test mask and are returned in the parameter ***pFlagMask***.

Example

Example programs for coordinating tasks with flags are contained in the files *koord_1.c* and *koord_2.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

4.11 Coordinating Tasks with Semaphores

Semaphores offer a simple means of synchronizing tasks. They are often used where individual tasks want to access common resources (global variables etc.). The section of the program in which a task wants to make such an access is called the “critical section”.

If more than one task reaches the critical section at the same time, the pseudo-simultaneous manipulation of common data areas must be prevented to avoid inconsistencies.

Semaphores work like keys or token: before a task reaches the critical section, it must first request the key from the RMOS API. The task can only start to process the critical section and manipulate common data when it is in possession of the key.

If the key has already been assigned to another task, the requesting task is changed to the BLOCKED status until the original task has returned the key.

Coordination of Data Access

The RMOS API supports the “key” concept by implementing binary semaphores. A binary semaphore is a data structure in other words created dynamically and can be occupied by a single task at a time and released again when it is no longer needed.

The following approach should be used when coordinating with semaphores:

1. Create a semaphore for each critical section of your program using the following function call:

   RmCreateBinSemaphore(pSemaphoreName,pSemaphoreID)

   This should be done in the startup part of your C program if possible. The call can be specified with a semaphore name to register the semaphore in the resource catalog. If the name is a NULL pointer, no entry is made in the resource catalog. The RMOS API returns the semaphore ID in the pointer pSemaphoreID.

2. Before you access common data in the critical section of your task, occupy the semaphore with the following call:

   RmGetBinSemaphore(..,SemaphoreID)

   The call must be specified with the semaphore ID. If the semaphore ID is unknown (different C programs), you can request the required semaphore ID from the RMOS API by specifying the semaphore name with the following catalog function:

   RmGetEntry(..,SemaphorepName,pList)

Effect of the call::

If the semaphore is not otherwise occupied, the call returns immediately and the task can access the required data. However, if the semaphore is occupied, the calling (parent) task is changed to the BLOCKED status.

The task only returns to the READY status when the associated semaphore has been released again.

3. After the task has finished accessing the common data, you must release the semaphore again with the following RMOS API call:

   RmReleaseBinSemaphore(SemaphoreID)

   The semaphore is now free again and can be occupied by another task.

Note

If an occupied semaphore is requested by several tasks, when it becomes free again it is assigned to the highest priority task; in the case of several tasks with the same priority it is assigned to the task which has been waiting longest. When a task gets the semaphore it is changed to the READY status. It can access the required common data as soon as it changes to the ACTIVE status.

Priority Change by Semaphore Ownership

In order to prevent a task requesting the semaphore being blocked because the semaphore is already owned by a task with lower priority (deadlock), the priority of the owning task is temporarily increased to equal that of the requesting task until the semaphore is released.

After the semaphore is released, the temporarily increased priority of the task is reduced to the value it had previously.

However, the priority is not returned to its original value if it now no longer has the same increased priority which it was temporarily assigned. The system assumes in this case that the priority has been purposely changed in the meantime with the call ***RmSetTaskPriority()***.

Deleting Semaphores

A semaphore which was created during runtime can be deleted again with the following RMOS API call in order to release the assigned memory:

RmDeleteBinSemaphore(SemaphoreID)

Advantages of the Semaphore Mechanism

Task coordination with the help of semaphores has several advantages when compared to the simple polling of status bits by the user program:

- The waiting period until a resource becomes available takes place under the control of the operating system. This increases system security because status bits are only changed by the system and not by user programs.
- No valuable processor time is wasted by active wait loops.
- The structure of the queuing ensures that when the semaphore becomes available it is assigned to the highest priority task, in other words execution continues with the most important function.

Note

A semaphore does not need to be released by the same task which originally requested it.

For example a semaphore can be requested with the ***RmGetBinSemaphore*** call and then released by another task using the call ***RmReleaseBinSemaphore***.

Difference between Flags and Semaphores

In contrast to semaphores which are an excellent means of distributing exclusive resources between competing parallel tasks, flags provide an efficient mechanism for waiting for events and for testing conditions.

The following two points illustrate the main differences between event flags and semaphores:

- If a task issues a call to set an event flag, the call has no effect if the flag is already set, in other words there is no mechanism to store the call until the flag has been cleared again.

  If a task issued a request to own a semaphore, this request is placed in a queue if the semaphore is already owned by another task. After the semaphore is returned, the operating system immediately sets its status to unavailable again, and it is given to the task which is waiting.

- If several tasks are waiting for an event flag to be set, all tasks are set to the READY status as soon as the flag is set.

  If several tasks are waiting for a semaphore to become available, only one task is set to the READY status when the semaphore becomes available.

Example

The following figure is a somewhat simplified example of a “producer” (Task_A) and a “consumer” (Task_B) with data exchange synchronized with two semaphores.

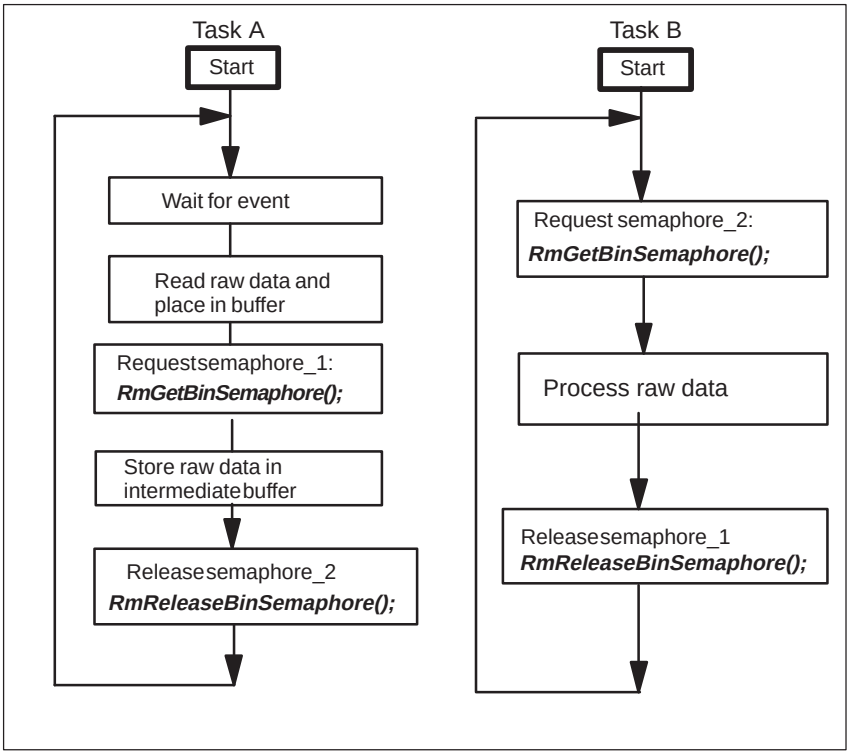


Figure 4-7 Example of Task Coordination Using Semaphores

Example

An example program corresponding to Figure 4-7 is contained in the file koord_2.c in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

4.12 General Information on Message Exchange

Message exchange is a central element in the RMOS API system architecture and has a major effect on the structure of an M7 RMOS32 task. A program to solve an automation assignment typically handles a fairly large number of asynchronous processes which are dictated by the machine to be automated or by other components which are linked to the program by various communication channels.

Generally speaking, a task has a central location where it waits for the occurrence of such events. In the simplest case this may be a time-controlled event which wakes up the task for example every 100 ms. However, a task may also need to handle a process alarm which is triggered by an I/O module or it needs to execute a particular data processing task when it receives raw data from another task.

All of these examples have in common that the triggering events can be converted to appropriate messages which can then be received by the task at a central location. When a message arrives, the task then carries out an action dependent on the message contents, and after finishing the action returns to the central location where it waits for other messages to arrive.

The task only changes to the READY status when an event occurs, and it is then assigned the required processing time in accordance with its priority.

Figure 4-8 shows the basic principle of message exchange between two tasks:

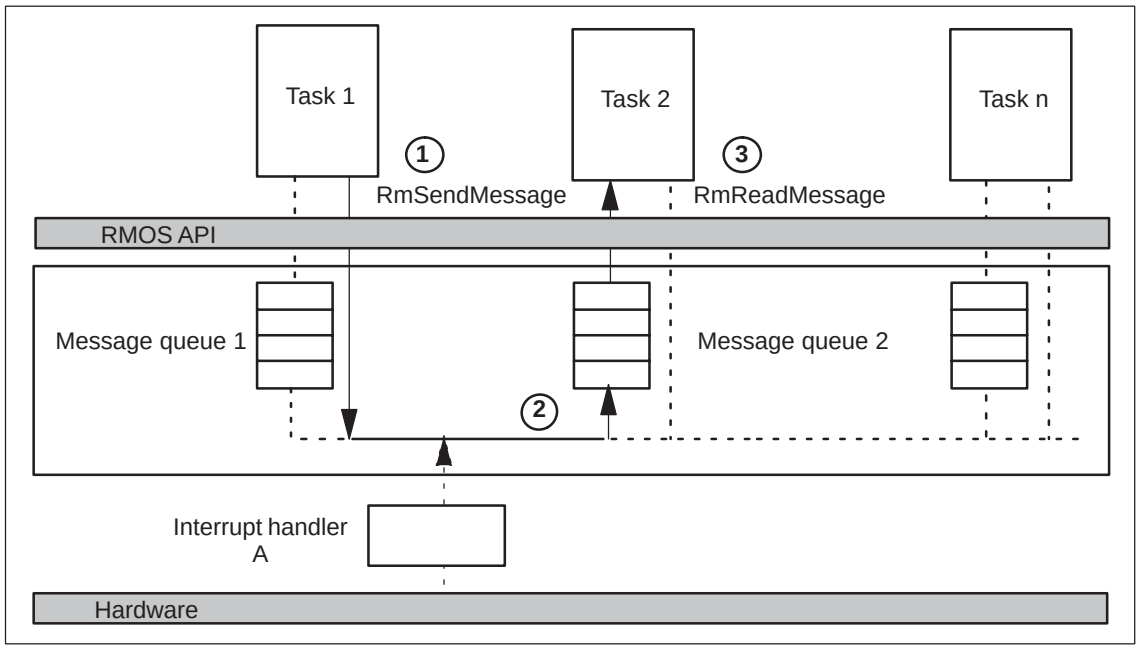


Figure 4-8 Message Exchange between Two Tasks

Creating and Deleting Message Queues

If you want to program a task to receive messages, you need to create a message queue for the task, preferably in the startup section. The queue can only be read from the task itself, but it can be written to from any other task and also from the interrupt handler.

You can create a message queue for a task from any task with the following RMOS API call:

RmCreateMessageQueue(pMessageQueueName,TaskID)

The message queue is entered in the resource catalog with the specified name.

The message queue is implemented as a dynamic list in other words a data structure whose size can change in accordance with requirements. A send request to the queue lengthens the queue by one entry and a read request shortens it correspondingly.

However, if necessary, you can specify the maximum length of the message queue, in other words the number of entries, using the following call:

RmSetMessageQueueSize(TaskID,Limit)

Before the task finally terminates, it is recommended to delete the associated message queue in order to release the assigned memory. This is done with the following call:

RmDeleteMessageQueue(TaskID)

Functional Sequence of Message Exchange

The actual message exchange between two tasks or between an interrupt handler and a task takes place in three steps:

1. The sender task sends a message to the receiver task with the following RMOS API call:

   RmSendMessage(Time,Priority,TaskID,Message,pMessageParam)

2. The operating system accepts the message and passes it to the message queue of the receiver task. If the message queue of the receiver task is already full, the send request is rejected with a corresponding error message.

3. The receiver task can now read the message from its queue with the following RMOS API call:

   RmReadMessage(Time,pMessage,pMessageParam)

   and start processing the message. The receiver task typically waits at a central location in the program in order to receive messages.

   For this reason the call ***RmReadMessage ()*** sets the tasks status to BLOCKED if the message queue is empty and the task only returns to READY when a message is received or the waiting time has expired.

Parameters for a Message

The following parameters from the sender task are specified with the message:

- **Address of the receiver**

  Every message includes the address of the intended receiver specified as the **Task ID** of the receiver task.

- **Message ID**

  A message can be identified by the sender with a message ID (***Message*** parameter) which is a value of type uint (32 bit). According to the agreement between the communicating tasks, the messageID can contain a sequence number or a sender ID or both.

- **Message content**

  The parameter ***MessagePar*** is specified by the sender when the message is sent. This parameter can be used in two ways:

  – The parameter ***MessagePar*** contains the message itself. In this case it can be a maximum of 32 bits in length.

  – The parameter ***MessagePar*** is a pointer to a data area in working memory.

- **Message priority**

  The sender of a message can specify the importance ***Priority*** of the message. The higher the value, the higher the priority. The range of the priority is 0 to 255.

  If a task is waiting for messages and the queue contains several messages, the task is first given the high priority messages followed by the medium priority messages and finally the low priority messages.

  Messages of the same priority are given to the receiver task according to the principle “first in first out”.

  Higher priority messages overtake lower priority messages.

- **Timeout**

  The parameter ***Time*** is used to specify how long to wait before the message is collected. The call used to send the message only returns when the message is collected or the waiting time has expired.

  If the parameter ***RM_CONTINUE*** is specified for ***Time***, the call used to send the message returns immediately.

4.13 Exchanging Messages between Tasks

Program Design Approach

When exchanging **user** messages (the handling of server messages is described elsewhere together with the corresponding calls), we recommend you to take the following approach in the design and programming:

1. Make a list of all messages which need to be exchanged
2. Give each message a messageID
3. Use a #define-statement in your program to replace each message ID by clear text in order to make your program code easier to read and thus to make it easier to make modifications at a later date.
4. Specify the type of each message (pointer or short message in one word).
5. Program the message exchange and evaluation of the messages

Choosing the Message ID

As discussed already above, a message can be sent in various “packages”.

In order to allow the receiver task to know how to interpret each message, you must specify a message type for each message ID.

The values 0x400 to 0x6FFF are available for user-defined IDs.

```
#define  M7MSG_60_ANALOG_SIGNALE      (uint)0x5000
#define  M7MSG_REGEL_PARAMETER        (uint)0x5001
#define  M7MSG_SCHWELLWERT_ERREICHT   (uint)0x5002
.
.
.
.
```

Table 4-2 Proposal for a Table with Message IDs (example).

Message ID	Message type	Comment
M7MSG_60_ANALOG_SIGNALE	pointer	60 analog signals
M7MSG_REGEL_PARAMETER	message	1 analog signal
.....	.....	.....

Exchanging Messages

Proceed as follows to exchange a message between two tasks:

1. Assemble a message in the sender task and determine the task ID of the receiver task.

   If the sender task and the receiver task are both within the same C program, the task ID can be stored in a global variable when creating the task in order to make it available to all other tasks.

   If the sender and the receiver task are in different C programs, the sender task must request the task ID of the receiver task by issuing the call ***RmGetEntry()*** and specifying the cataloged task name of the receiver task.

2. In the sender task, program the function call to send the message:

   RmSendMessage()

3. In the receiver task, first create the message queue and then program the call to wait for a message:

   RmReadMessage(RM_WAIT,..)

4. In the receiver task, evaluate the message ID and - if any - the message content.

Note

If the sender task includes a **pointer** with the message, then it must make sure that the receiver task has evaluated the message before it prepares a new message in the transfer area.

Coordination of message transfer can be done with the so-called handshake method. The receiver task sends an acknowledgement message to the sender task after it has evaluated each message.

The sender task only sends new data to the receiver task after receiving a positive acknowledgement of the previous message.

Checking the Message Queue for New Messages

In order to process messages, it is not necessary for the task to wait until a message arrives. It can carry on with other functions instead and check from time to time whether new messages have arrived.

The checks can also be done with the ***RmReadMessage()*** call.

1. Include a program line similar to the following in the receiver task:

   rm_error_code=RmReadMessage(RM_CONTINUE,pMessage,..)

2. Then evaluate the message ID as follows:

```
    .
if(rm_error_code!=RM_NO_MESSAGE) /*Message available*/

      {
       ........    /* Yes: process message        */
      }
   ......          /* No: do something else      */
```

The Structure of a Task

In order to implement communication with messages, it is usually best to structure M7 RMOS32 tasks with an event loop. The task then has a central waiting position where it waits for events in the form of messages.

The following figure shows the basic structure of an event-controlled M7 RMOS32 task:

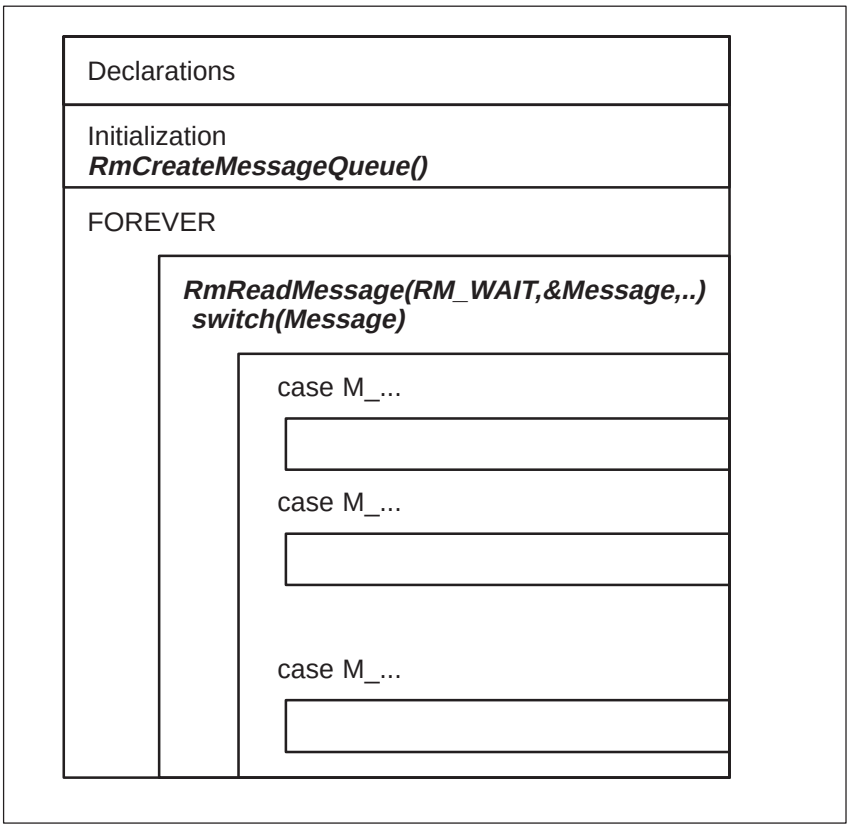


Figure 4-9 Internal Structure of a Task

Example: Structure of Message Exchange

The following figure follows on from Figure 4-8 and illustrates how two tasks can exchange data using message queues together with handshaking.

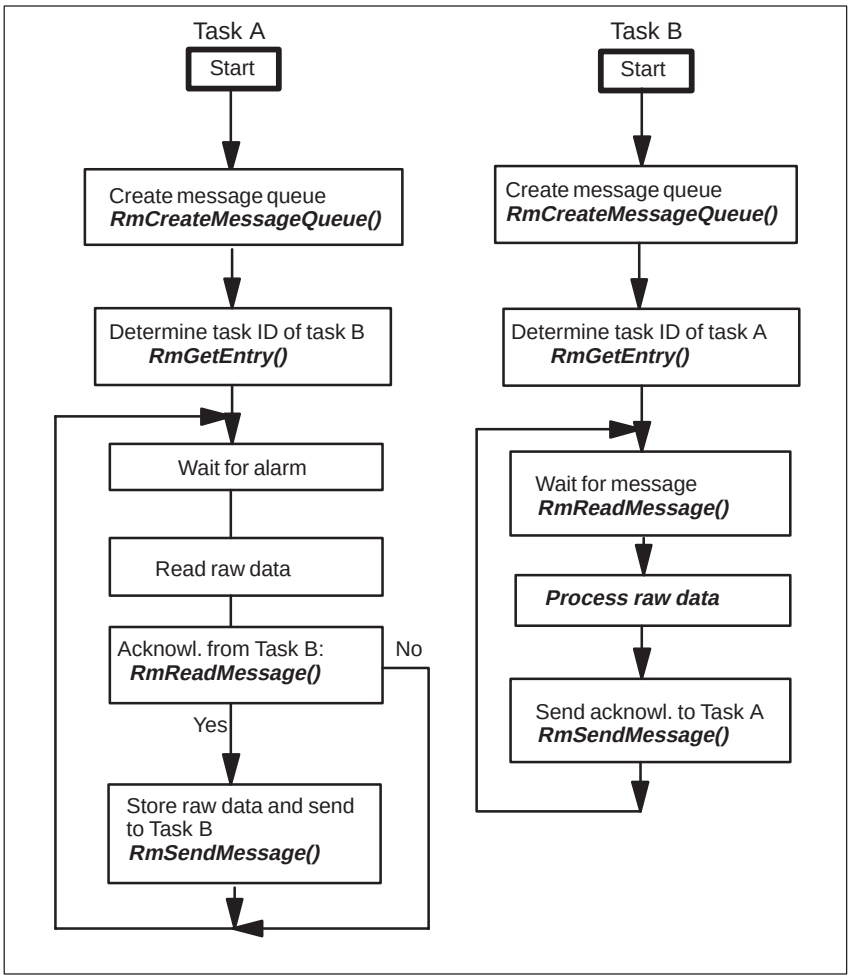


Figure 4-10 Example of Task Coordination Using a Message Queue

Example Program

The example program which implements the method shown in Figure 4-10 assumes that the two tasks are contained in different C programs. The program is contained in the files messqua.c and messqub.c in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

4.14 Data Exchange between Tasks Using Mailboxes

Another way of transferring data from one task to another is to use mailboxes. A mailbox is a global queue for messages which, in contrast to private message queues which belong to a single task, can be used by several receivers and senders. As with message queues, the length of a mailbox changes dynamically. Each send command lengthens the mailbox and each successful receive command shortens it.

Any task (sender task) can send a message to a mailbox which is then stored in the mailbox's own queue. Another task (receiver task) can then collect the message from the mailbox and evaluate it. The message in the mailbox is deleted after it has been collected.

The collection of messages in the mailbox queue is also priority dependent. If a task requests a message from the mailbox, it is given the message with the highest priority, and if several messages are present with the same priority it is given the message which has been stored the longest in the mailbox.

Data exchange with mailboxes is particularly useful if one or more senders want to communicate with several receivers via a central location.

Operations with Mailboxes

Similar calls are available for transferring messages with mailboxes as with communication using message queues:

- ***RmCreateMailbox(pMailboxName,pMailboxID)***

  The above call is issued by a task to create a mailbox. The call must be specified with a mailbox name which is entered in the resource catalog. The RMOS API then returns the mailbox ID which you need for all further sending and receiving calls.

- ***RmSetMailboxSize(MailboxID,Limit)***

  This call is used to limit the length of the mailbox, in other words the maximum number of messages that can be stored simultaneously. If a further message is sent to a full mailbox, the sending call is rejected with an error message.

- ***RmSendMail(Time,Priority,MailboxID,pMail)***

  This call is used to send a message to a mailbox. The position of the message in the mailbox is dependent on its specified priority.

  The parameter **Time** is used to specify how long to wait before the message is collected. The call used to send the message only returns when the message is collected or the waiting time has expired.

  The call must also specify the message priority and a pointer ***pMail*** to a message block of 12 bytes in length.

The format of the message block can be freely chosen. For example, it can contain the message itself (maximum 12 bytes) or the ID, address and length of a message according to the following scheme:

– Byte 1 to 4: Message ID

– Byte 5 to 8: “flat” address of message

– Byte 9 to 12: length of message

- ***RmSendMailDelayed(Time,..,pMailIDStruct)***

  This call is used to send messages (letters) to a mailbox with a specified delay time.

  The parameter ***Time*** is used to specify how long the message transfer should be delayed. The call returns an ID ***pMailIDStruct*** which allows you to cancel the action again if required.

- ***RmSendMailCancel(pMailIDStruct,pMail)***

  This call is used to cancel the action specified with ***RmSendMailDelayed*** within the specified delay time.

- ***RmReceiveMail(Time,MailboxID,pMail)***

  This call copies a 12 byte message from the mailbox specified with ***MailboxId*** to a user buffer whose address is specified with the parameter ***pMail***. The message is then deleted in the mailbox.

  The parameter ***Time*** is used to specify how long the call should wait until it returns.

  If two tasks wait for a message from the same mailbox, the first message to arrive is received by the task with the higher priority. If both tasks have got the same priority, the message is received by the task that issued the ***RmReceiveMail*** call first.

- ***RmDeleteMailbox(MailboxID)***

  This call is used to delete an empty mailbox and to release the assigned memory.

  If messages (letters) are still present in the mailbox or if a task is waiting for a message from the mailbox or if a ***RmSendMailDelayed*** call is still active, the delete call is rejected with an error message.

Example

The following figure shows the producer-consumer example with one producer and two consumers.

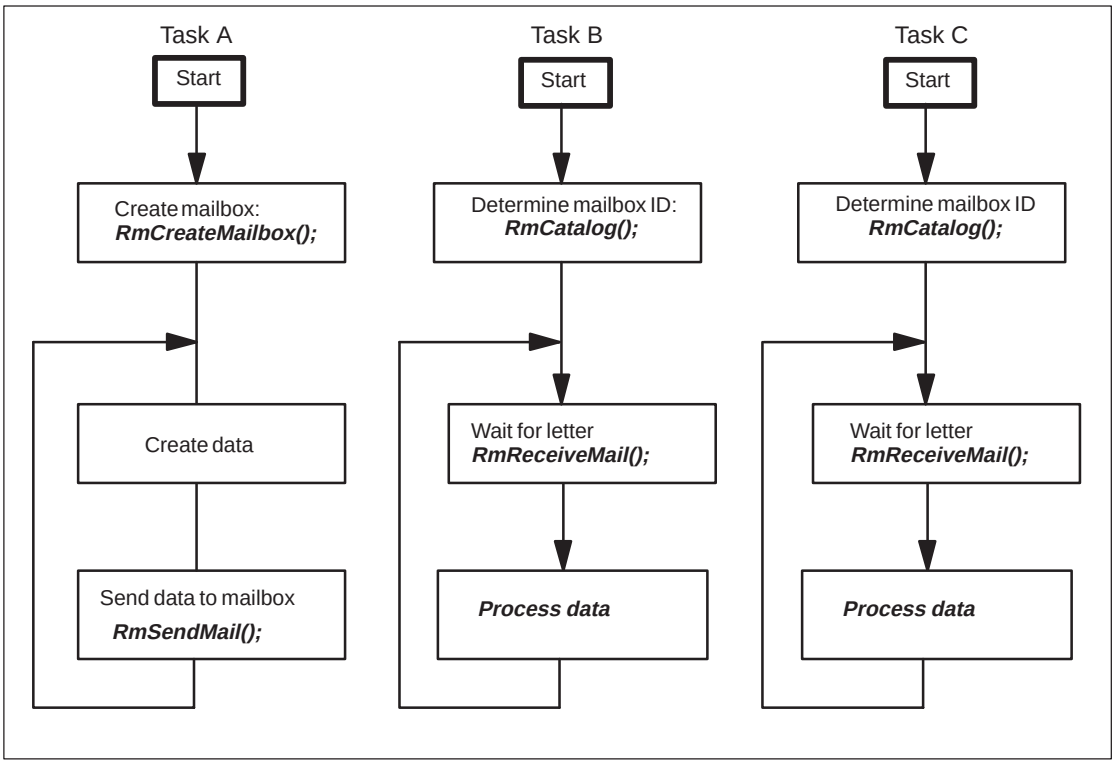


Figure 4-11 Example of Message Exchange Using Mailboxes

4.15 Functions for Memory Management

With the help of memory management, tasks are able to request contiguous memory areas from memory pools and release them to the system again after use. This allows a task to dynamically request memory which is required for example for measured data acquisition or analysis.

The M7 RMOS32 memory management also allows a physical memory area to be mapped to a linear address space.

Memory Areas for Dynamic Data Structures

During system start, memory not required by the operating system up to the memory limit of 16 MB is automatically assigned as a global free memory area (heap). Each of the programs can then request memory blocks from the heap.

Typical examples for reserving memory areas are:

- ***Memory automatically reserved for tasks***
  The system automatically assigns memory areas in the heap when creating a task with ***RmCreateTask()*** to store the task's stack and data segments.

  The assigned memory is released again and returned to the heap when the task is deleted with ***RmDeleteTask()***.

- ***Request from a task for temporary memory***
  The task itself can also request additional memory blocks from the heap using an RMOS API call, for example in order to store temporary data.

  When the memory blocks are no longer required, they can be released with another RMOS API call and returned to the heap.

Creating and/or Releasing Memory Pools

The advantage of several pools compared to a single pool is the improved structuring of the memory assignment to tasks and thus improved control of the overall memory.

For example, if two functionally independent tasks request memory in different pools, one of the tasks will never block the other one as a result of competition for memory. On the other hand, functionally dependent tasks may be able to share temporary memory from the same memory pool for pre-processing and post-processing of process data.

For this reason, you should carefully choose the sizes of the pools and the assignment of the tasks to the pools. The size of each memory pool can be specified when the pool is created.

The following calls are available for creating and/or deleting memory pools.

- ***RmCreateMemPool(pPoolName,pPoolAddress,Size,pPoolID)***

  The above call is used to create a memory pool. It must be specified with the name ***pPoolName*** under which the memory pool should be entered in the resource catalog and the required pool size ***Size***. The RMOS API returns the pool ID ***PoolID*** and a 32 bit flat pointer ***pPoolAddress*** to the start of the memory pool.

- ***RmDeleteMemPool(PoolID)***

  This call is used to delete a previously created memory pool. The call must be specified with the pool ID ***PoolID***.

Getting Information about Memory Pools

Before you request memory blocks from memory pools or from the global heap, you can first get information with following call on the overall size of each pool, the total amount of available memory and the largest available contiguous memory block:

RmGetMemPoolInfo(PoolID,pInfo)

The call must be specified with the ID of the memory pool (specify RM_HEAP for the global heap) and a pointer to a structure of type ***RmMemPoolInfoStruct***.

The RMOS API returns the required parameters in the specified structure.

Requesting Memory Blocks

The following calls are used to request (allocate) memory blocks from a previously created memory pool and/or from the global heap:

- ***RmMemPoolAlloc(Time,Mode,PoolID,Size,ppMemory)***

  The above call requests a memory block of size ***Size*** from the memory pool specified by ***PoolID***. If the requested amount of memory is currently not available, the parameter ***Time*** is used to specify how long should be waited for the assignment of the memory.

  If the parameter ***Size*** is specified as ***-1***, the call requests the largest contiguous memory area from the specified pool. The size of the allocated memory area can then be determined with the call ***RmGetSize()***.

  The call returns a pointer to the requested memory in parameter ***pMemory***. You can optionally specify the access mode (task specific or ##) with the parameter ***Mode*** and/or whether the requested memory should be released again or not with a subsequent ***RmFreeAll()*** call.

- ***RmAlloc(Time,Mode,Size,ppMemory)***

  The above call is used to request a contiguous memory block of size ***Size*** from the global heap. If the requested amount of memory is currently not available, the parameter ***Time*** is used to specify how long should be waited for the assignment of the memory

- ***RmGetSize(pMemory,pSize)***

  The above call returns the size of a previously requested memory area in parameter ***pSize***. You must specify a pointer ***pMemory*** to the memory area whose size you want.

Releasing Memory Blocks

The following calls are used to release memory blocks and return them to memory pools and/or the global heap:

- ***RmFree(pMemory)***

  This call is used to release a specified memory block and return it to the memory pool and/or the heap. You must specify the starting address ***pMemory*** of the memory block.

- ***RmFreeAll(TaskID)***

  This call is used to release all memory from memory pools and the heap which is currently allocated to the task ***TaskID***. Only those areas of memory are released that were not previously protected against global release with the ***Mode*** parameter.

Increase the Size of Memory Blocks

The following call can be used to adjust (increase or decrease) the size of memory blocks without affecting the contents:

RmReAlloc(Time,Mode,newSize,ppMemory)

You must specify the address of a pointer to the previously allocated memory block. The call returns a pointer to the adjusted memory block in the parameter ***ppMemory***. Additional memory which is required is taken from the corresponding memory pool or from the global heap, depending on where the pointer specified with the call was pointing.

The pointer which is returned may not be the same as the pointer which was issued, since it may have been necessary to shift the memory block (and the contents) to a different address.

Note

You can also use the routines ***malloc*** and ***calloc*** of the ANSI C runtime library for memory requests. Internally, the library routines ***malloc*** and ***calloc*** are then implemented with the RMOS API call ***RMAlloc()***.

Addressing Physical Memory

In addition to the functions of memory allocation and release, memory management also allows physical memory areas (for example dual port RAM) to be mapped to the linear address space of a task.

The following call maps a physical memory block to a linear address space:

RmMapMemory(PhysAddress,Length,pPointer)

You must specify the physical starting address ***PhysAdress*** and the length ***Length*** of the memory block to be mapped ,and the address ***pPointer*** of a pointer variable.

The call then returns in the pointer variable the flat starting address of the mapped memory block. The pointer can then be used by user programs to directly address the physical memory.

Example

The following figure shows two communicating tasks, whereby the data which is transferred is stored within a previously requested memory block.

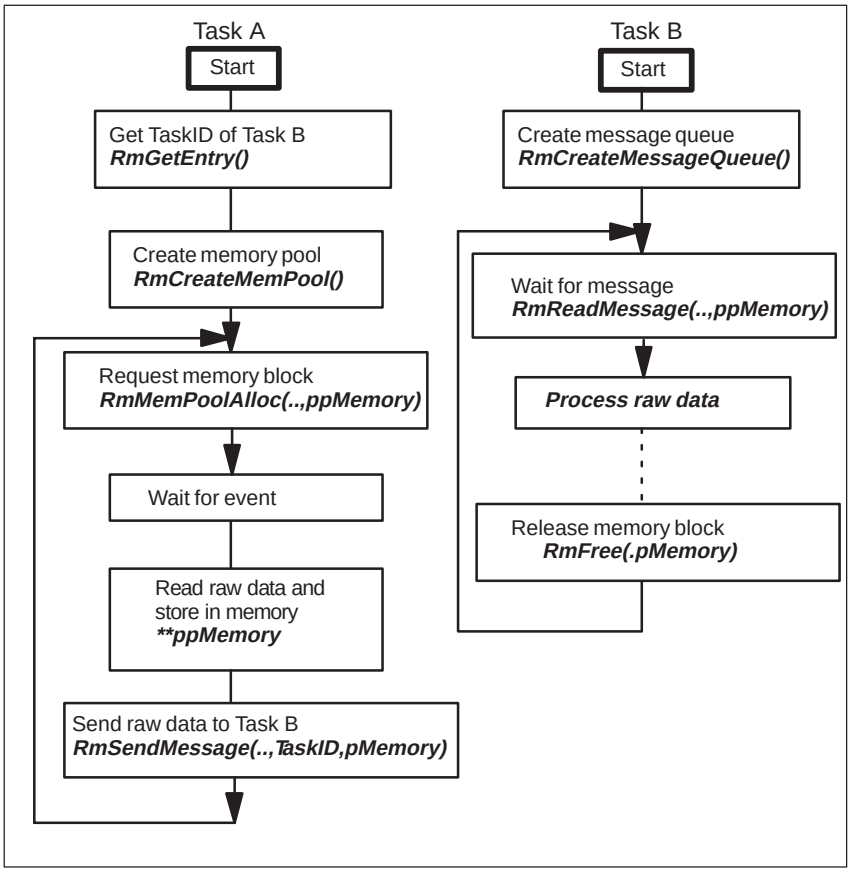


Figure 4-12 Example of Task Communication Using a Ring Buffer

In the above example, task A first creates a memory pool. It then requests a memory block from this pool for use in communication and waits for the next event (alarm).

When an alarm occurs, the raw data is read in and stored in the requested memory. The pointer pMemory to the raw data is then sent to task B for further processing using the RMOS API call ***RmSendMessage()***. When task B has finished processing, it releases the memory which was initially requested by task A.

The advantage of this type of data transfer compared to the method of the original example (Figure 4-6) is the additional buffering of the data to be transferred using memory blocks from the memory pool.

In this case, the explicit memory allocation prevents data which has not yet been fully processed by task B from being overwritten by task A when a new alarm arrives.

The number of buffered alarms, in other words the size of the ring memory, is dependent on the ratio of the overall size of the memory pool to the size of each requested memory block.

Example of Memory Request with Message Exchange

This example program illustrates the statements which are necessary for memory request in conjunction with inter-task communication. The program listings are contained in the files *mem_a.c* and *mem_b.c* in the directory ..\M7SYSx.yy\EXAMPLESM7API.

4.16 Data Security in the Event of Power Failure

The M7-300/400 automation computer has several different mass storage media: hard disk, floppy disk, memory card and OSD, whose file systems are managed by the operating system. Note that if a power failure occurs during a write access to the mass storage medium, the consistency of the file system is no longer guaranteed. As the system software (operating system, configuration files, etc.) is also located on the mass storage medium, a power failure during a write access can result in the system no longer being bootable.

To solve this problem, we recommend that you work with at least two mass storage media (or two partitions on the hard disk):

- One containing the operating system and the files relevant to the system and to which no write accesses are made during operation and
- One containing the user programs and read-only, back-up and load memory areas and to which write accesses during operation are permitted.

Procedure

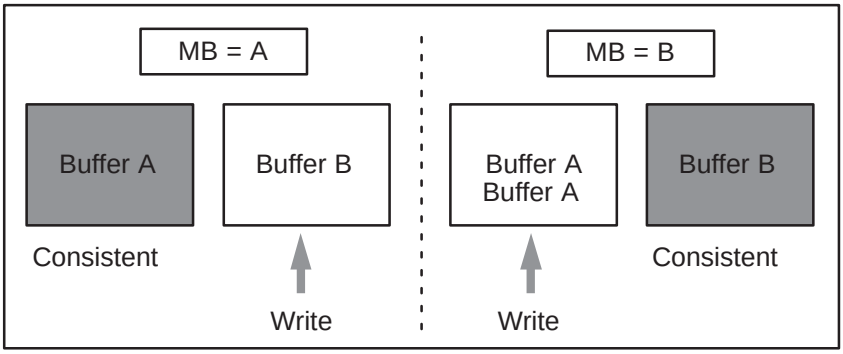
You can use the following procedure to ensure data consistency on the mass storage medium in the event of a power failure:

- Install the operating system on its own partition on the hard disk or on its own mass storage medium. Ensure that no write accesses are made to the partition or the mass storage medium of the operating system during operation. This, in turn, ensures that the operating system and the system data remain intact following a power failure, meaning a complete restart can always be performed.
- Do not create the directories for the back-up memory, the permanent load memory and the read-only memory on the same drive as the operating system but on the drive on which you write during operation. For this purpose, you must assign the relevant path names to the environment variables BACKDIR, RAMDIR and ROMDIR in the \ETC\INITTAB file on the boot drive.
- Do not install the user programs on the same drive as the operating system.
- Store the important data of your user programs retentively in the battery-backed SRAM. This ensures that these data remain accessible and consistent even after a power failure.
- Mark the write accesses to SRAM so that you can identify the consistent data after a power failure.

Working with Retentive Data

Use the battery-backed SRAM to maintain the consistency of important process data even after a power failure. Retentive data have the advantage that the write accesses are faster in comparison to the usual mass storage devices (hard disk, memory card) thus enabling more effective protection in the case of a power failure.

If process data are being continuously updated and must at the same time be consistent, use two or more alternating buffers. One buffer always contains the last consistent process data item while the new data item is being written into the other buffer. The statuses of the buffers are each marked with a memory byte: After the write action is completed, the memory byte is assigned the ID of the buffer which is being 'consistent' at the time.



In this way, you can use the shortest possible write access, that of a retentive memory byte, to protect the whole process of writing a process data item that can contain many write accesses.

Writing Retentive Data Blocks

In order to be able to establish the progress of the program after a power failure, with reference to write accesses, take the following measures in the user program:

1. Generate the required data blocks and memory bytes with the call ***M7CreateObject()***. In the *Part* parameter, specify the number of the block or memory byte configured as retentive.
2. Have each write access validated by a memory byte. Provided that the data in buffer A are consistent, the write access for a process data item in a retentive block should run as follows:
   - Write the process data item to the data block (buffer B) with any number of ***M7Write...*** calls.
   - Set memory byte to B with ***M7WriteByte***.

   After a power failure, you can use the memory byte contents in the SRAM to establish which process data are consistent.

Note

In contrast to retentive data blocks, the retentive memory bytes must be generated again with the call ***M7CreateObject()*** after every complete restart. Only then can their retentive contents be accessed.

4.17 Memory Protection

From Version V2.0, the M7-SYS system software contains a memory protection mechanism that prevents system software code and data areas from being overwritten by faulty user programs. This ensures a higher degree of system software availability and insurance against failure.

This section contains information on measures you must take in your program for the purpose of memory protection.

Memory Protection Levels

There is a distinction between two memory protection levels in executing programs.

- **User level:** Programs executing on the user level must only write to user data areas and user stacks. System data areas and code must only be read and are thus protected from being overwritten by user programs.

  The following programs execute on the user level:

  – User programs

  – CLI commands

  – Library functions, for example, the C runtime library

- **System level:** On the system level, memory protection is removed, that is, all memory areas can be written to.

  The following programs execute on the system level:

  – The operating system

  – The RMOS API and M7 API calls

  – System software servers and drivers

  – Low-level debugger (however, tasks loaded in the debugger execute on the user level)

  – Interrupt handlers

Releasing Memory Areas

Aborting and restarting a program that occupies system resources such as FRBs can sporadically result in system standstills.

Note

Please **always** ensure that a program abort is inhibited in the program (see the **inhibitabort** function), and that as many resources as possible are freed again.

If necessary, restart the system before restarting the aborted program.

Signal Handlers

Standard signal handler:
To provide a response to memory protection violations, a standard signal handler for SIGSEGV, SIGFILL and SIGFPE is set up in the operating system with the ***M7InitAPI*** call. In the standard signal handler, the task that caused the signal is terminated with ***RmEndTask***. If the task crashes, an entry is made in the diagnostics buffer (signal number, task ID). You will find additional information in the Reference Manual under the C Runtime Library (CRUN) function ***signal***.

User-defined signal handler:
Since the operating system does not support several signal handlers for one signal, the standard signal handler can be overwritten by installing a user-defined signal handler (with the CRUN function ***signal***). When setting up a new signal handler via the ***signal*** call, the address of any already installed handler is returned. In this case, you have two possibilities:

- At the end of your handler, you call the standard signal handler with the ***sig*** and ***type*** parameters if you desire a system response for your task, or
- You make the relevant entries to the diagnostics buffer in your program and terminate the task if your program is to respond to a signal itself.

Incorrect Call Parameters

If you use RMOS API and M7 API calls (system calls), that contain pointers or addresses of memory areas as parameters, you must ensure, that no memory areas containing system software code or data are referenced. For performance reasons, no such parameter check is carried out in the system software.

To check whether you specify a system call with a pointer to a valid memory area, you can access the memory area on the user level from the user task before the system call. If the access is successful, the area is valid and you can pass the pointer on to the system call. If the area is invalid, a signal is generated and the signal handler terminates the task (see above).

Interrupt Handlers

There is no memory protection for interrupt handlers.

Caution

Incorrectly programmed interrupt handlers can result in corruption of data and system crashes.

4.18 General Information on the Processing of Interrupts

An important feature of real-time operating systems is its ability to react quickly to external events. This takes place by interrupting the ACTIVE task and reacting to the interrupt by starting another program, the interrupt routine. The cause of an interrupt can be for example a hardware interrupt of a special PC module or a serial interface for rapid data acquisition.

Interrupts in DI and I mode (see explanation below) are processed without needing to change the status of the currently ACTIVE task to READY. It is only necessary to save the appropriate CPU registers to the ACTIVE task's stack. The ACTIVE task resumes processing again after returning from the interrupt routine.

Caution

The synchronization between largely independent tasks and interrupt routines is one of the most difficult features to handle in the multitasking programming.

Accordingly, you should only program interrupt routines if you have sufficient experience in the field of multitasking programming and also in using Petri networks to analyze deadlocks.

Furthermore, interrupt routines which are incorrect or too long can cause the real-time response to the overall system to be dramatically impaired.

What is an Interrupt Handler?

An interrupt handler is a special function which is installed by a task. An interrupt handler is assigned to either a hardware or a software interrupt. In general, C functions can be directly used as interrupt handlers under M7 RMOS32.

The interrupt handler is activated when the hardware or software interrupt event occurs. In the I or S mode, the program context is automatically saved and the interrupt handled accordingly. At the same time, the system automatically creates the correct context for the interrupt handler in order to allow it to properly access the variables within its address space.

A hardware interrupt handler is activated by a hardware interrupt, for example an interrupt which is triggered by an interface ASIC. A software interrupt handler is activated by an appropriate C call from a task.

The code of the software interrupt handler executes within the context of the calling (parent) task. For example, if the own TaskID is requested within a software interrupt handler, the TaskID of the calling task is returned. The software interrupt handler can thus be considered to be a global subprogram of a task.

Limitations within an Interrupt Handler

Whether or not RMOS API functions and functions of the C runtime library can be called depends on the operating mode in which the program code of the interrupt handler is executing.

No M7 API calls can be issued within an interrupt handler.

M7 RMOS32 System Modes

When code is being processed by the processor, the system may be operating in one of processing four modes:

- Application mode (A mode)
- Disabled interrupt mode (DI mode)
- Interrupt mode (I mode)
- System mode (S mode)

The processing mode is directly related to how and whether the program can be interrupted by other tasks and/or events.

With the exception of A mode, all other modes are reserved for execution of code within the system kernel and/or within interrupt handlers.

The processor mode within which each interrupt handler is executed is defined when installing the handler (see section 4.17).

A mode

The A mode is directly associated with the execution of program code within normal user tasks. The assignment of processor time is is the responsibility of the scheduler, in other words task execution can be interrupted either by a higher

priority task or by an interrupt.

DI mode
If the interrupt handler has been assigned to DI mode, an interrupt causes the installed routine to be executed directly. In this case, after automatic storage of the flag registers and the return address, the interrupt routine itself must save the rest of the registers (this is normally done by the compiler if the function has been declared as an interrupt routine).

During its execution, the interrupt routine cannot be interrupted by other interrupts.

The DI mode must only be used for interrupt processing for extremely time-critical functions. The total number of processing steps must not exceed ca. 25 assembler commands.

When the interrupt routine is finished, the saved registers must be restored and the routine must be exited with IRET.

API calls and runtime library functions must not be used when processing an interrupt in DI mode.

I mode
If more than 25 assembler commands are required to process the interrupt, processing must take place in I mode. In contrast to interrupt processing in DI mode, the installed interrupt handler is embedded in an interrupt driver which is provided by the system.

The interrupt driver saves all registers, services the interrupt controller to enable all higher priority interrupts and switches to the system stack before the user routine is allowed to execute.

With interrupt processing in the I mode, the interrupt routine can be interrupted at any time by another interrupt of higher priority. The total number of assembler commands in the interrupt routine must not exceed 100 in order not to impair the real-time response of the system, in other words the reaction time to interrupts of lower priority.

After the interrupt has been processed, the interrupt driver restores the original context and resumes program execution at the place where it was interrupted.

RMOS API calls can be issued during execution of the interrupt handler. However, these calls are not executed immediately but they are placed in an internal queue instead. They are executed in S mode when the interrupt routine, in other words the I mode, has been terminated.

S mode
If more than about 100 assembler commands are required to process the interrupt, processing must take place in S mode. Execution in S mode differs substantially from the two previous interrupt modes. In this case, the scheduler is responsible for assigning processor time.

With interrupt processing in S mode, a separate system task is created for the interrupt routine which is then placed in a FIFO queue (first in first out). If several system tasks have been created, they are processed one after the other. Before the system task status is changed to ACTIVE, the scheduler first changes the

status of a ACTIVE user task (if any) to READY.

System tasks have a higher priority than all user tasks, and are thus processed first. All other interrupts are also enabled in the S mode.

Note

The processing duration in S mode is not unlimited. Other system processes, if any (FIFO processing), the M7 server and all programs are blocked during this time.

Caution

Incorrectly programmed interrupt handlers can result in corruption of data and system crashes.

Additional Functions

In addition to designing your own interrupt routines, it is possible to arrange for other functions to take place if an interrupt occurs. The RMOS API offers the following functions for this purpose:

- Task start by interrupt

  This call is used to start a task if an interrupt occurs. Each interrupt cause the system call ***RmQueueStartTask(..,TaskID,..)*** to be issued. If the specified task has the DORMANT status, it is started immediately if the priority is high enough.

- Mailbox message by interrupt

  If the associated interrupt occurs, the system sends a message to the specified mailbox. This allows external events to trigger unique messages which are queued in the mailbox. The messages can then be processed by tasks which are servicing the mailbox.

  If the size of the specified mailbox is limited by the ***RmSetMailboxSize()*** call, the message is not sent if the mailbox is already full. In this case the interrupt is ignored.

Note

The associated interrupt handler must be uninstalled before the mailbox and/or the task is deleted.

4.19 Installing Interrupt Handlers

Interrupt routines are ideally suited for reacting quickly to external events. Accordingly, the RMOS API provides various calls with which you can install your own interrupt handlers for processing interrupts in the various processing modes.

Interrupt Number

When installing the interrupt handler, the associated interrupt number ***IntNum*** can be assigned in two ways:

- ***SW Interrupt:*** if the specified interrupt number lies between 0 and 255, the associated interrupt is treated as a software interrupt. In this case, the system does not carry out any enabling and/or further servicing of the interrupt controller.
- ***HW Interrupt:*** to specify an HW interrupt, you must specify an interrupt number *x* between 1 and 15 with the macro ***IRQ****(x)* which is defined in the include file **RMTYPES.H**. In this case, in addition to the automatic conversion for the entry in the interrupt vector table, the interrupt is treated internally as a hardware interrupt.

Interrogation and Installation of Interrupt Handlers

The RMOS API provides the following calls for interrogating and installing interrupt handlers:

- ***RmGetIntHandler(IntNum,PHandlerEntry)***

  The above call is used to determine the presently installed interrupt handler. You must specify the interrupt number ***IntNum*** and the address ***pHandlerEntry*** of a pointer.

  The call returns the entry point of the currently installed interrupt handler in the pointer variable.

- ***RmSetIntDIHandler(IntNum,DIHandlerEntry)***

  The above call installs the C function as an interrupt handler with the interrupt number ***IntNum***. The entry point for the C function is ***DIHandlerEntry*** and the interrupt is executed in the DI mode (disable interrupt).

- ***RmSetISHandler(IntNum,IHandlerEntry,SHandlerEnty)***

  The above call installs an interrupt handler with the interrupt number ***IntNum***. When an interrupt occurs, the subprogram specified by ***IHandlerEntry*** is executed in the I mode. Following this and if the return value is not 0, the subprogram specified by ***SHandlerEntry*** is executed in the S mode.

  If the return value of the I mode routine is 0, the S mode routine is not executed.

- ***RmSetIntTaskHandler(IntNum,TaskID)***

  The above call is used to install the system interrupt handler for starting a specified task. If an interrupt ***IntNum*** occurs, task ***TaskID*** is started.

- ***RmSetIntMailboxHandler(IntNum,MailboxID)***

  The above call is used to install the system interrupt handler for sending a message. If an interrupt IntNum occurs, a message is sent to the mailbox ***MailBoxID***.

  The main contents of the message are the interrupt number, the interrupt vector and the interrupt type (SW or HW interrupt). The exact structure of the message is described in the Reference Manual.

- ***RmSetIntDefHandler(IntNum)***

  The above call is used to uninstall a custom interrupt handler for the interrupt ***IntNum*** and to reinstall the default interrupt handler.

Example of Installing an Interrupt Handler

In this example program, a HW interrupt handler, which consists of the two functions IHandlerEntry and SHandlerEntry, is installed in the main task. The HW interrupt handler is uninstalled again at the end of the main task. The example program to is contained in the file *setinthd.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

M7 API: General Information, Events and Alarms

5

Chapter Overview

5.1 Overview

The M7 API (Application Programming Interface) is a C programming interface which provides M7 RMOS32 programs with all necessary functions to solve your automation assignments.

The M7 API provides all functions which you need to access the process I/Os in order to ensure correct control of the process.

In addition to calls for accessing the process I/Os, the M7 API provides functions for managing internal S7 objects, calls for communicating with other automation components and further functions to integrate your M7 automation computer transparently in an S7 automation system.

What is Described in this Chapter?

This chapter contains general information on the M7 API, as for example, data formats and converting between Intel and SIMATIC byte order. It also contains information on the following topics:

- Using the system servers
- Responding to events such as alarms or operating mode transitions
- Getting information
- Time-controlled program execution

You learn how to use each of the M7 API function calls to solve different programming requirements and what you need to take account of with each call. The usage of the functions is illustrated by simple example programs. The examples can be found on the system diskette in the directory ...\M7SYSx.yy\EXAMPLES\M7API.

At the start of each section you will find a summary of the structure and function of each of the servers. This is followed by a description of the associated calls.

This chapter only gives a general overview of each function call and does **not** contain **a detailed description** for example of the parameters. This can be found in the Reference Manual “System Software for M7-300/400, System and Standard Functions”.

5.2 General Notes on the M7 API

Conventions and Header Files for M7 RMOS32 Programs

The names of the M7 API calls are written in the Hungarian notation and always start with the letters **M7**...

M7 RMOS32 programs must include the file **M7API.H**, which is the header file for the M7 API function prototypes.

This file also contains the data types, structure definitions and error codes.

Data Types of the M7 API

In order to make it easier to port programs to other systems, the M7 API environment uses its own type definitions instead of machine-dependent data type designators such as *int* or *long*.

The following table lists the data type designators used in the M7 API environment. Their definitions are contained in the header file **M7API.H**.

Table 5-1 Data Type Definitions in the M7 API

Designator	Type definition	Significance
UBYTE	unsigned char	unsigned character (value: 0...255)
UWORD	unsigned short	unsigned 16 bit integer (value: 0...65 535)
UDWORD	unsigned long	unsigned 32 bit integer (value: 0... 2^{32} - 1)
SBYTE	signed char	signed character (value: -128...127)
SWORD	signed short	signed 16 bit integer (value: -32 768...32 767)
SDWORD	signed long	signed 32 bit integer (value: -2^{31} ... 2^{31} - 1)
BOOL	unsigned int	unsigned int Boolean value
BYTE	UBYTE	unsigned character (value: 0...255)
WORD	UWORD	unsigned 16 bit integer (value: 0...65 535)
DWORD	UDWORD	unsigned 32 bit integer (value: 0... 2^{32} - 1)
REAL	float	32 bit real value (value: +/– 1,1754x10^{-38} ... +/– 3,4028x10^{38})
M7ERR_CODE	int	error code

Initializing the M7 API

The M7 API must be initialized before your C program can use the M7 API runtime system and start to process the automation assignment itself. This is done with the call:

M7InitAPI(void)

This function must be called once at the beginning of each C program that uses M7 API functions.

Handling of Error Codes

M7 API function calls include **error codes** on the return value, or - in contrast to RMOS API function calls - in a pointer. The error code allows you to determine the success or failure of the call.

Note

For reasons of clarity and to save space, detailed error handling has been omitted from the examples in this manual and from the descriptions of each of the function calls. Nonetheless, your programs should always be designed to react appropriately to any error codes which are returned.

5.3 Using Servers to get Notification of Events

Functional Sequence

Each task which wants to be notified about the occurrence of an event (access to an S7 object, process alarm, time event, etc.), must register itself with the server which is responsible for the particular event.

If the event occurs, the server then automatically sends an appropriate notification message to the task's message queue.

Figure 5-1 shows the functional sequence when registering for alarms and sending server messages.

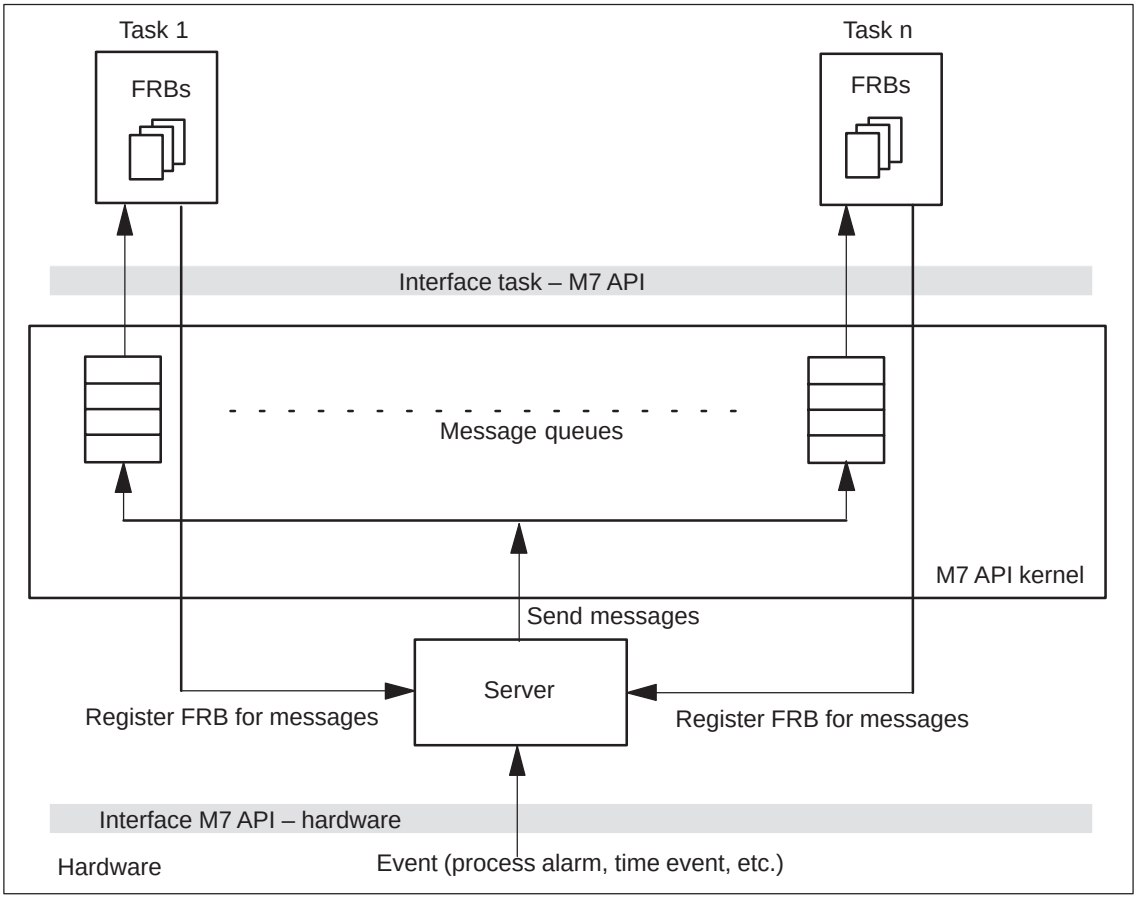


Figure 5-1 Sending Server Messages

Function Request Block

When the task registers itself with the server, the call must specify a pointer to an FRB (Function Request Block). When the FRB is registered, all of the FRB entries are initialized too, in other words they are initialized with parameters to specify the event.

While processing the registration request, the FRB is used by the server to store intermediate results which can occur during processing.

During this time, the FRB is under the control of the server and must **not** be modified by the user program.

If a program wants to register itself for notification by a server, it must first allocate (define) an associated FRB variable in its data area. The data type of the variable and thus the associated FRB parameters are function specific, in other words they depend on the type of request (see “Reference Manual, System Software for M7-300/400, System and Standard functions).

Note

Never allocate the data area for the FRB from your program stack; it should always be allocated in the global variable area or dynamically on the heap or in another memory pool.

In addition to the function-specific parameters, each FRB contains a user field (“tag” field) which can be used by the task to manage the requests, in other words to uniquely identify each FRB which is returned.

The following macro is used to write user-defined information into the tag field:

M7SetFRBTag(pxxFRB,FRBTag)

The call must be specified with a pointer ***pxxFRB*** to the FRB to be processed. ***FRBTag*** is the user-defined tag.

Analogously, the following macro is used to read the user-defined tag in an FRB:

M7GetFRBTag(pxxFRB)

Registering with a Server for Notification of Specified Events

Use the following procedure to register a task with a server to receive automatic notification of events:

1. Define an FRB variable in your program for each registration request (FRBs, which are C data structures, are defined in the file **M7API.H** with Typedef statements).
2. Write a user-defined tag into the FRB. The tag is used by the task to uniquely identify each FRB which is returned. Specify the tag with the following macro:

   M7SetFRBTag(pxxFRB,FRBTag)

   Specify the address of the FRB to be registered in parameter ***pxxFRB*** and specify the FRB tag in ***FRBTag***.
3. Register the FRB with the server for notification messages with the following M7 API call:

   M7Link...(pxxFRB,..)

   Specify the address of the FRB to be registered in parameter ***pxxFRB***.

Note

Each task must only register one FRB for each event.

Identification of the FRB

If you have registered several events of the same type with the same server, for example several time-dependent messages, you can identify each message using the FRB tags. You do this as follows:

1. Use the following call to read the message from the task's message queue:

   RmReadMessage(RM_WAIT,&Message,&pMessageParam)
2. If the message originated from an M7 server, after reading the message queue the variable ***pMessageParam*** points to a valid ***FRB***.

   Use the following macro to get the FRB tag:

   frbTag = M7GetFRBTag(pMessageParam)
3. Branch in your program according to the message ID ***Messag***e and the FRB tag ***frbTag*** using the following method:

```
if (Message == M7MSG_TIMESERVER)
 { switch (frbTag)
  {
 case: ...
   }   /* End switch */
}      /* End if */
```

Interrogating the Error Code

While processing the registration request, the FRB is used by the server to store intermediate results which can occur during processing.

If an error occurs during processing, the corresponding error code is also stored in the associated FRB. This error code can be read after reading the message queue using the following macro:

M7GetFRBErrCode(pFRB)

The macro must be specified with a pointer ***pFRB*** to the associated FRB. The error code is returned in the return value.

Program Examples

Program examples for handling FRBs and registering and deregistering with a server can be found in the previous and/or the following chapters.

5.4 General Information on Alarm Handling

Alarm (interrupt) handling concerns alarms which are triggered by the process I/Os and/or by an M7 FM module (slave) and are sent to the module (master) which processes the alarm via an interrupt line of the P bus and/or module bus. The module which processes the alarm is always a CPU module or an FM module.

Alarm Server

With the M7 automation computer, alarms are handled with the help of the alarm server. The alarm server can receive alarms, identify them and acknowledge them. However, the alarm server only has the task of managing the alarms and does not itself deal with (**process**) the alarm. The latter is generally the responsibility of a user task.

Alarm Types

The following alarm types can be processed by a user task:

- Process alarm:

  Process alarms are module-specific. For example, a module can trigger a process alarm when a particular signal edge is detected at a digital input. Process alarms can also be triggered by an IP or a CP. In these cases, the reason for the process alarm is usually more complex than with simple signal modules.

  The modules also send a so-called *alarm descriptor* of 4 bytes in length.

- Diagnostic alarms:

  As with process alarms, diagnostics alarms are also module-specific. For example, an I/O module can trigger a diagnostic alarm when it detects a wire break or an internal error (defective component).

  In the System S7-300, in addition to the alarm descriptor of 4 bytes, diagnostic-enabled modules also generate 16 bytes of extended diagnostic information in data record 1 of the module which can be read by the user program with the function ***M7LoadRecord()***.

  If a diagnostic alarm is triggered by a station attached to the PROFIBUS DP bus, in addition to the S7-compliant extended diagnostic information the DP station can request a complete DP standard diagnostic telegram.

- Remove/insert alarms:

  If a configuration change takes place in the RUN, STOP and RESTART operating states (for example, a module becomes defective or is removed), a remove/insert event is generated and sent to the alarm server. This alarm results in an entry in the diagnostics buffer and the system status list in the relevant CPU (see section 5.7).

- Battery fault alarms (see section 5.14).

Both process and diagnostics alarms can also be triggered with corresponding M7 API calls by a user task running on an M7 FM (see section 5.6). In this case, the additional information which is sent with the alarm can be specified as needed.

In this case, the significance and interpretation of the additional information must be explicitly coded in your program.

Alarm Identification

A process alarm from a module can be uniquely assigned to one of up to 32 channels. In this case, the alarm is associated with an **alarm descriptor** which identifies the channel which triggered the alarm. The structure of the descriptor is module-specific. It normally uses a 1 to many coding - see Figure 5-2.

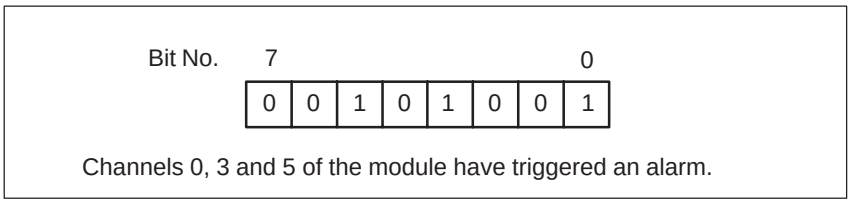


Figure 5-2 Example for an Alarm Descriptor from a Signal Module

Alarm Handling by a Task

When a task wants to handle alarms, it registers an FRB (**function request block**) with the alarm server. When the registered alarm occurs, the alarm server then sends a message to the task. After the task has carried out the alarm specific actions, it requests the alarm server to acknowledge the alarm.

The following figure shows the functional sequence of alarm handling by a task:

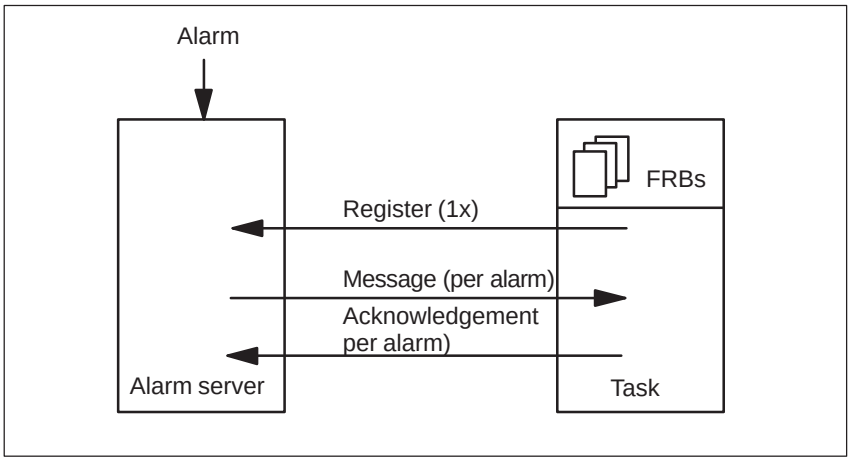


Figure 5-3 Principle of Alarm Handling by a Task

FRB for Alarms

In order to redirect and acknowledge alarms, the alarm server needs a so-called *function request block* (FRB) which is used to store the parameters of the alarm.

Tasks handling alarms must register a **separate FRB** with the alarm server for each **module** from which you want to receive alarms and for each **alarm type** (please see section 5.3 for information on the general usage of FRBs).

Registering FRBs with the Alarm Server

The following calls are available to register an FRB with the alarm server:

- ***M7LinkIOAlarm(pPAFRB,PType,Addr,Alarmmask,Mprio)***
- ***M7LinkDiagAlarm(pDAFRB,PType,Addr,Mprio)***

These calls are used to register process and diagnostic alarms, respectively. In addition to a pointer ***pPAFRB*** to the associated FRB, you must specify the logical base address ***Addr*** and the type ***pType*** (input module: ***M7IO_IN***, output module: ***M7IO_OUT***) of the module which may trigger the alarm.

The module type needs to be specified since an input module and an output module may lie on the same logical address (but not with the default addressing!).

Furthermore, when registering a process alarm you need to specify the parameter ***Alarmmask*** to indicate which of the 32 channels you want to monitor. The parameter ***Mprio*** specifies the required priority of the message from the the alarm server.

Evaluating the Alarm Message

After the corresponding alarm has been received, the alarm server sends a message to the task with the message ID M7MSG_IO_ALARM or M7MSG_DIAG_ALARM (***pMessage*** parameter).

The following macros make it easier for the user program to evaluate the parameters of the FRB which is referenced in the message (parameter ***pMessageParam*** contains a pointer to the FRB):

- ***M7GetIOAlarmPType(pPAFRB)***
- ***M7GetDiagAlarmPType(pDAFRB)***

This macro returns the type of the module which triggered the alarm (input or output module).

- ***M7GetIOAlarmAddr(pPAFRB)***
- ***M7GetDiagAlarmAddr(pDAFRB)***

This macro returns the logical base address of the module which triggered the alarm.

- ***M7GetIOAlarmMask(pPAFRB)***

In the case of a process alarm, this macro is used to read the alarm mask which was specified when the alarm was registered.

- ***M7GetIOAlarmState(pPAFRB)***
- ***M7GetDiagAlarmInfo(pDAFRB, pInfo)***

For both process and diagnostic alarms, this macro is used to read the alarm descriptor.

Diagnostic Data Records DS0 and DS1

The diagnostic data of a module are found in the data records 0 and 1 of the system data area:

- Data record 0 contains 4 bytes of diagnostic data which describe the current status of a module.
- Data record 1 contains
  - The 4 bytes of diagnostic data which are also in data record 0 **and**
  - The module-specific diagnostic data

The structure and the contents of the diagnostic data records is described in the User Manual.

Process Alarm Information of IF 961-DiO

The process alarm information for the IF 961-DIO interface submodule comprises 4 bytes of which only byte 0 is relevant. The cause of interrupt is stored as follows in byte 0: bit n set means ‘Level change at channel n‘ (n = 0 to 7). You will also find this information in the hardware manuals in the ‘Interrupt Register’ section of the ‘IF 961-DIO Interface Submodule Addressing and Interrupt‘ chapter.

Format of the Alarm Information

The supplementary alarm information for diagnostic and hardware interrupts must be converted to the SIMATIC or Intel format in the following situations:

- When an alarm is initiated by an FM 356-4 for an S7-300 CPU, use the *M7_SWAP_DWORD* function to convert to the SIMATIC format.
- When an alarm is initiated by an alarm-capable 16-bit signal module for a CPU 488-4/5, use the *M7_SWAP_WORD* function to convert to the Intel format.

Getting the Standard Diagnosi

If a diagnostic alarm is triggered by a station attached to the PROFIBUS DP bus, in addition to the S7-compliant extended diagnostic information the DP station can request a complete DP standard diagnostic telegram.

The following call is used to read the DP standard diagnostic telegram:

M7DPNormDiagnose(Baddr,pBuffer)

The call must be specified with the logical base address ***Baddr*** of the associated ET-ER and the address ***pBuffer*** of a buffer. The call then returns the complete standard diagnostic information in the buffer.

The structure and format of the standard diagnostic information is described in detail in DIN19245T3.

Acknowledging Alarms

When a task has finished processing an alarm, the alarm must be acknowledged to the alarm server. This is done with the following calls for a process alarm and for a diagnostic alarm, respectively:

M7ConfirmIOAlarm (pPAFRB)

or

M7ConfirmDiagAlarm (pDAFRB);

You must specify the pointer to the FRB which was included in the message.

Deregistering the Notification of Alarms

If, in a particular process configuration, a task no longer needs to process a particular alarm, it can deregister the associated FRB with the alarm server with the following calls for a process alarm and for a diagnostic alarm, respectively:

M7UnLinkIOAlarm (pPAFRB)

or

M7UnLinkDiagAlarm (pDAFRB)

The task can then for example re-register an alarm for the module using a different alarm mask.

Note

Before a task which handles alarms terminates, it must deregister all alarms which are registered with the alarm server.

5.5 Procedure to Follow for Alarm Handling

Configuring Alarms

Before an alarm can be generated by a local I/O module, the module which triggers the alarm must be suitably configured.

Configuration is described in the user manual “Standard Software for S7 and M7 STEP 7”. The parameters required to enable alarm generation are module specific and are contained in the corresponding module description.

Registering for Notification of Alarms

Before a task can receive and process alarms from a particular module, the task must register the alarm with the alarm server. Registration is required for each module from which you want to receive alarms and for each alarm type.

Reacting to Alarms

After the task which processes the alarms has registered all required alarm types, it should wait for alarms to arrive with the call ***RmReadMessage()***.

If one of the registered process or diagnostic alarms occurs, the task receives a message from the alarm server with the message type M7MSG_IO_ALARM or M7MSG_DIAG_ALARM, respectively, and can then process the alarm accordingly. The message contains a pointer to the FRB of the alarm event which has been triggered.

Each alarm which is received should be handled according to the following method. The alarm should first be analyzed by the task with the help of the FRB as follows:

1. Read the the signal module’s alarm descriptor from the FRB (you must not change the alarm descriptor in the FRB!).
2. Determine which channel has triggered the alarm (the alarm descriptor is module specific!).
3. In your task, call the section for processing the alarm type (in other words react to the signal change) that has occurred.
4. Acknowledge the alarm with the alarm server.

Alarm Processing Priority

You can determine freely the priority with which alarms are processed. If alarms are processed by the task logged at the alarm server, the alarms are processed according to the priority of this task. If this priority is too low or too high, alarm processing can be relocated to another task whose priority you can define accordingly.

Example of Alarm Handling

In the example program in the file *alarm.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API, the main task registers with the alarm server for two process alarms and one diagnostics alarm.

The task then waits in the main loop for messages from the alarm server and branches accordingly.

On receiving the terminate task message, alarm notification is deregistered and the task is terminated.

5.6 Sending Alarms to a CPU

A CPU or FM module (master) can receive an interrupt request, in other words an alarm, via the P bus.

In one possible scenario, a signal module (slave) triggers an alarm which is then sent to an FM (master). In another scenario, an FM (slave) can trigger an alarm which is then sent to a CPU (S7 CPU or M7 CPU).

The M7 API provides calls to allow an M7 RMOS32 task running on an FM to send process and diagnostic alarms to a CPU.

Functional Sequence of Alarm Handling

If a task on the FM triggers an alarm, the interrupt line on the P bus is activated and an interrupt is triggered on the CPU. In the case of an S7 CPU, current execution is interrupted and the OB which is assigned to alarm handling is called.

The S7 CPU then acknowledges the alarm to the task which triggered it. The triggering task can interrogate the status of the alarm handling with appropriate M7 API calls.

If the alarm receiver (master) is an M7 CPU or M7 FM, tasks registered with the alarm server are notified using messages.

After the alarm handling is finished, the tasks acknowledge the alarm to the alarm server. The triggering task can interrogate the status of the alarm handling with appropriate M7 API calls. The calls used to handle alarms are described in section 5.5.

Sending Alarms to a CPU

In many applications, it is necessary to immediately notify the CPU (master) about an event on the FM (slave), for example if the FM has reached a certain stage of processing or a fault has occurred.

The following M7 API calls allow an M7 RMOS32 task to trigger an alarm and/or to interrogate the status of the alarm handling:

- ***M7SendIOAlarm(Alarminfo)***

  This function triggers an alarm of type process alarm. The parameter AlarmInfo can be used to send an optional 4 byte coded message containing **additional information on the alarm** to the CPU.

  The following call can be used by the triggering task to interrogate the status of the alarm handling:

  M7GetIOAlarmBusy()

  The return value of this function (busy or idle) indicates whether the alarm handling on the master is still running and/or is finished.

- ***M7SendDiagAlarm(...,pAlarminfo)***

  This function triggers an alarm of type diagnostic alarm. The parameter ***pAlarminfo*** is a pointer to diagnostics information on the alarm. You must provide these diagnostic data. The structure and the contents of the diagnostic data records is described in the User Manual.

  The following call can be used by the triggering task to interrogate the status of the alarm handling:

  M7GetDiagAlarmBusy()

  The return value of this function (busy or idle) indicates whether the alarm handling on the master is still running and/or is finished.

Example for Exchange of User Data with Handshake

This example program illustrates how to send information from an M7 CPU to an FM through user data and using handshake. The example is contained in the files *nutz_fm.c* and *nutz_cpu.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

The FM acknowledges the processing of the user data by sending a diagnostic alarm. The CPU indicates when new user data is available by acknowledging the alarm.

5.7 Registering for Remove/Insert Events

In the SIMATIC S7/M7-400 system, the configuration of I/O modules is monitored automatically once per second. If a configuration change takes place in the RUN, STOP and RESTART operating states (for example, a module becomes defective or is removed), a remove/insert event is generated and sent to the alarm server. This event results in an entry in the diagnostics buffer and the system status list in the relevant CPU.

The alarm server then sends a corresponding message to all registered tasks. The tasks must now evaluate the message.

Note

Power supply modules, CPUs, adapter casings, IMs and interface submodules must not be removed!

At least 2 s must elapse between removing and inserting a module in order to be certain that the removal and insertion is detected by the CPU.

Since removal is only monitored once a second, an access error can be detected when accessing a removed module before this period has elapsed.

Registering and deregistering for insert/remove events is not supported in the S7/M7-300 system.

Responding to Insert/Remove Events

To be able to respond to insert/remove events in the user program, you must register the M7/S7-400 programmable controller mounting rack with the ***M7LinkZSAlarm*** call and respond accordingly to receipt of an insert/remove event. If a mounting rack has not been registered for the insert/remove event, the CPU changes from RUN to STOP if an insert/remove event occurs in this mounting rack.

Registering for Notification of Insert/Remove Events

The following M7 API call is available for registering an FRB for insert/remove events:

M7LinkZSAlarm(pZSFRB,IMRBaddr,MPrio)

This call registers the FRB with the address ***pZSFRB*** with the alarm server for notification of insert/remove events. In ***IMRBaddr***, you must also pass to the call either the starting address (logical base address) of the IMR interface module or the ***M7CR_BADDR*** ID for the central mounting rack and the ***MPrio*** priority of the returned message.

If an insert/remove event occurs after registering the FRB, the task is sent a message from the alarm server with the message ID M7MSG_ZS_ALARM.

Macros for Evaluating the FRB

The referenced FRB contains a data structure with information on the module in each slot within the specified rack.

The M7 API provides the following macros to evaluate the parameters of the FRB:

- ***M7GetZSAlarmIMRBaddr(pZSFRB)***

  This macro returns ***IMRBaddr*** (starting address of the IMR interface module) which was specified when registering the insert/remove FRB. The macro must be specified with a pointer ***pZSFRB*** to the associated FRB.

- ***M7GetZSAlarmMode(pZSFRB,SlotNum)***

This macro must be specified with a pointer ***pZSFRB*** to the associated FRB and the slot number ***SlotNum***, and returns the mode of the module in the specified slot. Your user program must then compare the returned value with the following constants:

  – M7DEV_OK (module okay)

  – M7DEV_REM (module has been removed)

  – M7DEV_PUT (module has been inserted)

The parameter ***SlotNum*** must lie within the range 1 ... MAX_SLOT_400. The constant MAX_SLOT_400 specifies the maximum number of slots in the System S7-400.

- ***M7GetZSAlarmPType(pZSFRB,SlotNum)***

This macro must be specified with a pointer ***pZSFRB*** to the associated FRB and the slot number ***SlotNum***, and returns the type (input or output module) of the module in the specified slot.

- ***M7GetZSAlarmAddr(pZSFRB,SlotNum)***

This macro must be specified with a pointer ***pZSFRB*** to the associated FRB and the slot number ***SlotNum***, and returns the logical base address of the module in the specified slot.

- ***M7GetZSAlarmIdent(pZSFRB,SlotNum)***

This macro must be specified with a pointer pZSFRB to the associated FRB and the slot number ***SlotNum***, and returns the ID number of the module in the specified slot. The ID of each module is specified in the associated hardware description.

Acknowledging the Insert/Remove Event

After evaluation of the message, you must acknowledge the event so that the FRB allocated by the system can be enabled again. The following is available for acknowledgment:

M7ConfirmZSAlarm(pZSFRB)

The call must be specified with a pointer ***pZSFRB*** to the FRB referred to when registering.

Deregistering the Notification of Insert/Remove Events

If a task no longer requires notification from the alarm server of insert/remove events, it can deregister the associated FRB with the alarm server with the following call:

M7UnLinkZSAlarm(pZSFRB)

This call must be specified with the pointer ***pZSFRB*** to the associated FRB.

Parameterizing a Newly Inserted Module

If a module is inserted in RUN mode, the CPU checks that the module type of the newly inserted module agrees with the configured module. If the module types agree, parameterization is carried out. Either the default parameters or the parameters you assigned with STEP 7 are then transferred.

5.8 Handling Process Image Transfer Errors

Transfer errors can occur in free cycle during the cyclic update of the process image if a change in the configuration is detected (for example, module defective or removed). In this case, a process image transfer error event is generated and passed on to the free cycle server.

The free cycle server then sends the relevant signal to the registered tasks.

Responding to Process Image Transfer Errors

To be able to respond in the user program to process image transfer errors, you must register the task with the ***M7LinkPIError*** call and then make the relevant response when a process image transfer error is received. If a task has been registered for process image transfer errors, the CPU changes from RUN to STOP when a process image transfer error occurs.

Registering for Notification of Process Image Transfer Errors

The following M7 API call is available for registering an FRB for process image transfer errors:

M7LinkPIError(pPIEFRB,MPrio)

The call registers an FRB with the address ***pPIEFRB*** to receive notification of process image transfer errors at the free cycle server. You must also specify the ***MPrio*** of the returned message.

If a process image transfer error occurs after registering the FRB, the task is sent a message from the free cycle server with the message ID M7MSG_PI_ERROR. The message contains the process image type and the process image address where the transfer error has occurred.

Macros for Evaluating the FRB

M7 API has the following macros to provide you with information on the contents of the process image transfer error message:

- ***M7GetPIErrorPIType(PIErrMsgBuf,PIType)***

  This macro determines from the M7MSG_PI_ERROR message the process image type at which the transfer error has occurred and returns it in the ***PIType*** variable. The call must be specified with a pointer to the message buffer.

- ***M7GetPIErrorAddr(PIErrMsgBuf,Addr)***

  This macro determines from the M7MSG_PI_ERROR message the address at which the transfer error has occurred and returns it in the ***Addr*** variable. The call must be specified with a pointer to the message buffer.

Deregistering the Notification of Process Image Transfer Errors

If a task no longer requires notification of process image transfer errors, you can deregister the FRB again. The following call is available for deregistration:

M7UnLinkPIError(pPIEFRB)

This call must be specified with a pointer ***pPIEFRB*** to the FRB referred to in the registration.

5.9 General Information on Time-Controlled Program Execution

The time server has the purpose of sending M7 RMOS32 tasks an automatic message at a specified time of day (optionally on a specified day or date) or after a specified time interval has elapsed. The task must first register itself with the time server to receive time-dependent messages. The time server also provides functions for setting and reading the current system time.

If a task needs to receive a message after a particular time interval has expired and/or on reaching a particular system time, it first needs to register an FRB with the time server.

The time server then sends a message to the task when the time-dependent event occurs. In the case of operation with handshake, the task must acknowledge the message to the time server before it can receive another time-dependent message.

Figure 5-4 shows the main features of time-dependent program execution.

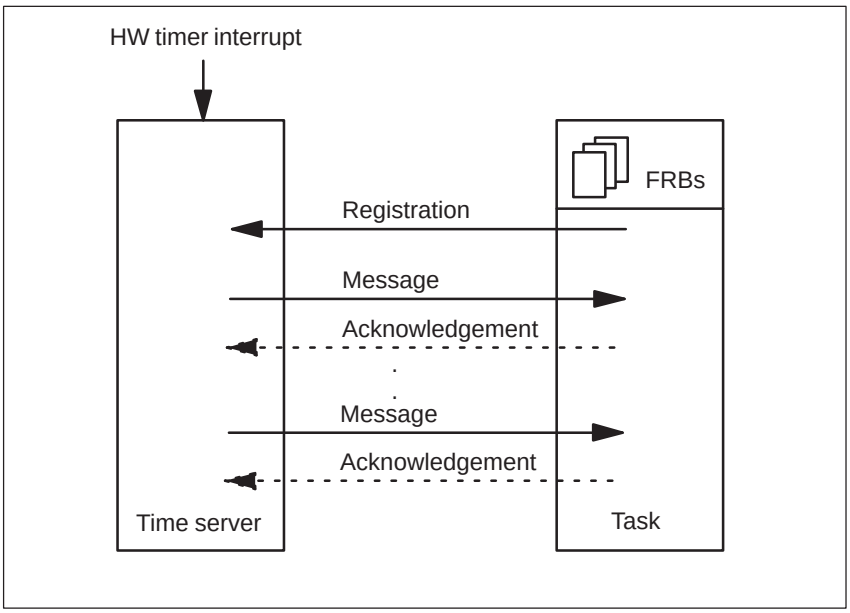


Figure 5-4 Main Features of Time-Dependent Program Execution.

Time Message Types

The following types of time messages are available:

- Periodical time messages– are sent to the task by the time server **periodically** each time a specified **time interval** has elapsed.

  Periodic time messages are used for functions that need to be called periodically when a specified constant time interval has elapsed. Typical applications are for example controlling algorithms or data acquisition functions.

- Singular time messages – are sent by the time server to a task **once** only after a specified **time duration** has expired (for example 100 ms).

  Singular time messages are used for functions that must be executed with a specified time delay after an event occurs (for example the time between reaching the maximum temperature and switching off the plant) or following an action (for example the time between switching on a drive until the running speed has been reached.

- System time dependent time messages – are sent by the time server to a task at a specified time once per day or on specified dates.

  System time dependent time messages are used for functions that need to execute at a particular time of day or at a particular time on a particular date, for example in order to heat up the process before the shift starts or to switch from Winter time to Summer time or vice versa.

Note

Time messages are sent only in the RUN operating mode.

Operating Modes for Periodic Time Messages

Periodic time messages can be specified for the following operating modes:

- Operation without handshake

  When operating without handshake, the time server sends the time message when the specified time has been reached, independently of whether the task has acknowledged receipt of the previous message or not.

- Operation with handshake

  With handshake mode, the task must acknowledge receipt of each periodic time message before the next periodic time message is sent.

5.10 Programming Time-Dependent Processing

Registering for Notification of Time Events

Before a task can receive time messages it must register itself for notification by the time server.

To allow it to deliver time messages and process the acknowledgements, the time server needs a so-called *function request block* (**FRB**) to store the parameters for each message. The task must specify a pointer to an FRB in its own data area for each registered time message.

Registering for Notification of Periodic Time Messages

Periodic time messages are registered with the following call:

M7LinkPeriodicTimer(pTFRB,TimeBase,Period,Handshake,MPrio)

The time interval with which the message is periodically sent is specified in the two parameters ***TimeBase*** and ***Period***.

TimeBase specifies the base time and ***Period*** specifies the associated multiplication factor. The following values can be specified for time base: 1ms, 10ms, 100ms and 1s.

Example: For a time interval of 300 ms, you should specify ***TimeBase*** as TB_100ms (for 100ms) and ***Period*** as 3.

The ***Handshake*** parameter is used to specify whether you want handshake mode (M7WITH_HANDSHAKE) or operation without handshake (M7NO_HANDSHAKE). The parameter ***Mprio*** specifies the required priority of the message sent by the time server.

Registering for Notification of Singular Time Messages

Singular time messages are registered with the following call:

M7LinkOneShotTimer(pTFRB,TimeBase,Time,MPrio)

The time duration after which the message is sent is specified in the two parameters ***TimeBase*** and ***Period***.

TimeBase specifies the base time and ***Period*** specifies the associated multiplication factor. The following values can be specified for time base: 1ms, 10ms, 100ms and 1s.

The parameter ***Mprio*** specifies the required priority of the message sent by the time server.

Registering for Notification of System Time-Dependent Time Messages

System time controlled time messages are registered with the following call:

M7LinkDate(pTFRB,pDateTime,Periodic,MPrio)

The required time (and the required date if necessary) is encoded in the data structure ***DateTime***. The call is specified with a pointer ***pDateTime*** to ***DateTime***.

If you want a time message to be sent **daily** at the specified time independent of the date, the parameter ***Periodic*** should be specified as M7DAILY. In this case, the date specified is not evaluated.

If you want the message to be sent at a specified **time** on a specified date, the parameter ***Periodic*** should be specified as M7ONCE.

The parameter ***Mprio*** specifies the required priority of the message sent by the time server.

Analyzing Time Messages

After a task has registered all required time messages, it should wait for time messages to arrive with the call ***RmReadMessage()***.

If one of the registered time events occurs, the task receives a message from the time server with the message type M7MSG_TIMESERVER (parameter ***pMessage***).

The message contains a pointer to the FRB of the time event event which has been triggered (parameter ***pMessageParam***). In the case of periodic and singular time messages, you can interrogate the time base ***TimeBase*** and the multiplication factor ***Period*** which were specified with the registration using one of the following two macros:

M7GetTimeBase(pTFRB) or ***M7GetPeriod(pTFRB)***

Both macros must specify the pointer pTFRB to the FRB which was included in the message.

Each time message which is received should be handled according to the following method:

1. Get the FRB tag which was specified when registering the FRB (see section 5.3) with the following call:

   frbTag = M7GetFRBTag(pTFRB);

   You must specify the pointer ***pTFRB*** to the FRB which was included in the message.

2. Use the variable ***frbTag*** to program a branch (switch) which contains routines for handling each registered time message.

3. If the received time message was registered for handshake mode, acknowledge receipt of the message with the time server.

Acknowledging Time Messages

Periodic time messages registered for handshake mode must be acknowledged after receipt before the next periodic time message is sent.

A periodic time message is acknowledged with the following call:

M7ConfirmPeriodicTimer(pTFRB)

You must specify the pointer ***pTFRB*** to the FRB which was included in the message.

Determining Lost Periodic Time Messages

If a task processes periodic time messages, it may happen that if processing takes a long time one or more of the following time messages get lost.

If you are handling the periodic time messages **with handshake**, this allows you to check whether periodic time messages have become lost. In handshake mode, the time server keeps an internal counter of periodic time messages which have not been acknowledged.

You can find out how many time messages have been lost with the following call:

M7GetLostPeriods(pTFRB)

You must specify the pointer ***pTFRB*** to the FRB which was included in the message.

When the lost periodic messages are interrogated, the internal counter in the time server is set to 0 again.

Deregistering the Notification of Time Events

If a task no longer requires notification of time events, it can deregister them again with the time server depending on their type. The deregistering is done with the following calls:

- deregistering periodic time messages:

  M7UnLinkPeriodicTimer(pTFRB)

- deregistering singular time messages:

  M7UnLinkOneShotTimer(pTFRB)

- deregistering system time controlled time messages:

  M7UnLinkDate(pTFRB)

All of the calls must be specified with a pointer to the FRB to be deregistered.

Note

Before a task which handles time events terminates, it must deregister all events which are registered with the time server.

Reading and Setting the System Time

The time server provides functions to read and/or set the current system time. The current system time is needed for example in order to output a time stamp with a message or with measurement data.

M7GetTime(pDateTime)

This call must be specified a pointer ***pDateTime*** to a data structure **DateTime** which you have provided in your data area.

Analogously, the system time can be set by a task with the following call:

M7SetTime(pDateTime)

This call must be specified with a pointer ***pDateTime*** to an initialized data structure of type M7TIME_DATE.

Example for Handling Time Messages

In the example program *timer.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API, the main task registers with the time server to receive a periodic time message, a message at a specified time and a message after a specified time duration has elapsed.

In the processing loop, the task waits for messages from the time server and branches correspondingly.

Following receipt of the end of task message, no more messages are sent and the task is terminated.

5.11 Reading and Setting the System Clock of a Communication Partner

The two M7 API calls ***M7KReadTime()*** and ***M7KWriteTime()*** are used to read and set the system clock of a communication partner. These functions require S7 single -system connections that are configured with STEP 7 (see chapter 8).

Procedure: Reading the Clock

Proceed as follows to read the time from the internal system clock of a communication partner:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   The ***M7KInitiate*** call establishes an application link to the communication partner. You must specify the host address ***pHostAddr*** of the remote partner as a string. If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. The following call is used to read the time:

   M7KReadTime(ConnID,pBuffer,nBufsiz,pnBytes)

   This call reads the time of the server programmable controller into the data structure of type M7KTIME which is referenced by pointer ***pBuffer***.

   You also specify the application link ID ***ConnID*** and the length of the buffer ***nBufsiz***.

   The call returns the number of actually read bytes in ***pnBytes***.
4. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Procedure: Setting the Clock

Proceed as follows to set the internal system clock of a communication partner:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   The ***M7KInitiate*** call establishes an application link to the communication partner. You must specify the host address ***pHostAddr*** of the remote partner as a string. If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. The following call is used to set the system clock:

   M7KWriteTime(ConnID,pBuffer,nBufsiz,pnBytes)

   This call sets the clock of the server programmable controller to the time specified in a data structure of type M7KTIME. You must specify a pointer ***pBuffer*** to the data structure.

   The call requires in addition the application link ID ***ConnID*** and the length ***nBufsize*** of the data structure.
4. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

5.12 General Information on Operating Modes

This section first explains why it is useful to implement operating mode awareness in your program. It then gives a summary of possible operating modes and the functions of the OMT server, and then explains how to use OMT functions in your program.

Why are Operating Modes Necessary?

When a program runs on programmable logic controller (PLC), there are several different possible phases of operation and operating mode transitions (OMT). Starting out with the RESET or POWER ON situation, the first status to be reached is the STOP mode, which is used for initializing the operating system. The PLC then starts up the process (STARTUP mode) and then executes its normal automation assignment (RUN mode). It remains in the RUN mode until it returns to a passive condition as a result of an event. The PLC can also be put in the HALT mode for test and debugging purposes.

On an S7 CPU, the requirement to implement the various operating modes is directly implemented in hardware and software. The execution of a user program in an S7 CPU is very closely coupled to the operating modes. The operating system of an S7 CPU is even able to stop the user program at any place and to resume processing again at the same place.

In order to allow the integration of an M7 into an S7 system, the M7 also supports this operating mode model. The OMT server (Operating Mode Transition) is the central mechanism in the system for processing operating modes and their transitions. It allows user programs to recognize the various system operating modes and to get notification of operating mode transitions.

In addition, it is possible to directly influence the operating mode from a user program with appropriate calls.

Which Operating Modes are Available?

The following figure shows the possible operating modes of the M7 automation computer and the transitions between them which are supported.

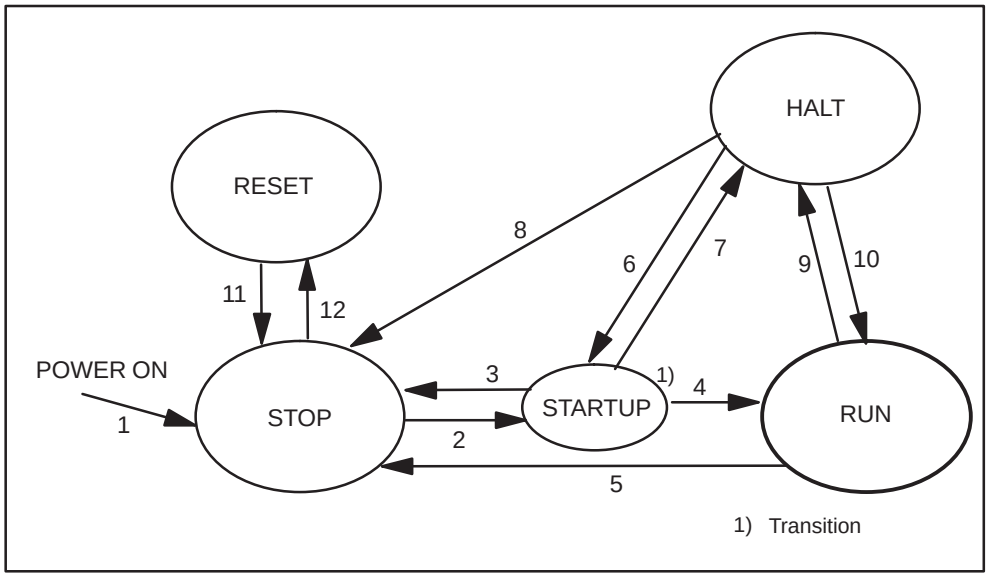


Figure 5-5 M7 Operating Modes and Operating Mode Transitions which are Supported by the OMT Server

Note

If the M7 is operated as a function module (FM), the operating modes which the OMT server will allow the FM to reach (with the exception of STOP) are dependent on the operating mode (RUN or STOP) of the associated CPU (S7 CPU or M7 CPU).

When the CPU changes to STOP mode, this forces the FM to change to STOP mode too. The OMT server only allows the other operating modes to be reached on the FM when the CPU is in the RUN mode.

STOP

STOP mode has the following characteristics:

- The operating mode STOP does not affect the execution of the operating systems M7 RMOS32, MS-DOS or the running tasks.
- However, running tasks are not sent any time-dependent messages, process or diagnostic alarms or notification of the cycle control point and/or free cycle.
- Signals **are not output** from the signal modules to the process I/Os (OD signal - output disable - is active). However, peripheral inputs can be read in and evaluated in the program.
- The transition to stop mode can be caused by the operating mode switch on the module, on request of a program, using communication functions for configured connections (for example from programming device), as a reaction to a system error and in the case of an FM by a corresponding operating mode transition of the associated CPU.
- A transition to STOP mode can be made from all other operating modes.

STARTUP

STARTUP mode has the following characteristics:

- The STARTUP mode represents the transition between STOP and RUN (if an error occurs the system may return to STOP mode again: transition 3 in Figure 5-6). In STARTUP mode, the user program should carry out all startup measures such as testing and initializing.
- Tasks which want to carry out initialization in the STARTUP mode should register for the STARTUP function of the free cycle server.
- The OD signal is still active, in other words in STARTUP mode signals are not output from signal modules to the I/O modules. However, peripheral inputs can be read in and evaluated by the program.
- The STARTUP mode can only be reached on an FM if the associated CPU is in the RUN mode.

RUN

RUN mode has the following characteristics:

- In the RUN mode, all signals are output from the signal modules to the I/O modules (OD Signal is inactive), and all services of the M7 servers are available. All of the functions execute which are necessary for process control.
- The RUN mode can only be reached by an FM if the associated CPU is in the RUN mode and the operating mode switch of the FM is in the position RUN or RUNP.
- RUN mode is reached by successfully completing the STARTUP mode (transition 4 in Figure 5-6).

HALT

HALT mode can be reached on request of a programming device for test purposes. It has the following characteristics:

- All program times and monitoring times are halted.
- Alarms which occur in halt mode are not delivered.
- The OD signal is active, in other words no signals are output from the signal modules to the I/O modules. However, peripheral inputs can be read in and evaluated by the program.
- programming device and operator interface functions are still supported in HALT mode.

RESET

RESET mode is used to set up the original status of the M7 following serious errors. The RESET mode is reached as a result of the following functions:

- Programmed Memory Reset request from a programming device or from a user program either locally or using communication functions for configured connections (see chapter 8.11)
- Triggered RESET function with the operating mode switch (MRES)

Programmed Memory Reset Request

The following actions are carried out:

– All S7 objects in the working memory the static RAM, the backup memory and the permanent loading memory are deleted.

– All S7 objects in the constant memory are copied to the working memory and activated. The original status of the S7 object server is restored.

Note

Executing user programs are not halted or terminated but continue to run unless an explicit request is programmed. A task that is not programmed to respond to operating mode transitions can be assigned a modified hardware configuration causing serious faults.

So that a task can give the relevant response to such events, it must register with the OMT server for the operating mode transition STOP to MRES with the *M7LinkTransition* call.

However, if the system software locks up, the restoration of valid S7 objects is not sufficient alone to bring the M7 to an executable condition. In this case, the system software also needs to be reloaded. This can be accomplished with the following function:

Memory Reset with the Operating Mode Switch

The following actions are carried out:

- The processor and all hardware modules of the S7 are reset and/or reinitialized.
- The entire operating system M7 RMOS32 is rebooted.
- All S7 objects in the working memory the static RAM, the backup memory and the permanent loading memory are deleted.
- All S7 objects in the constant memory are copied to the working memory and activated. The original status of the S7 object server is restored.
- The file INITTAB is processed, in other words all programs listed in this file are loaded and started.

Functions of the OMT Server

The control and display of the operating modes is carried out by the so-called OMT server. It processes requests for mode changes (transitions) and notifies registered tasks about transitions and newly reached modes. Furthermore, the OMT server controls the LED lamps and interrogates the position of the operating mode switch. It also generates the OD signal in order to disable the peripheral outputs.

In addition to managing operating modes, the OMT server also monitors the battery voltage of the buffer battery and sends a message to registered tasks when the voltage drops below a limiting value.

Figure 5-6 shows the main functions of the OMT server.

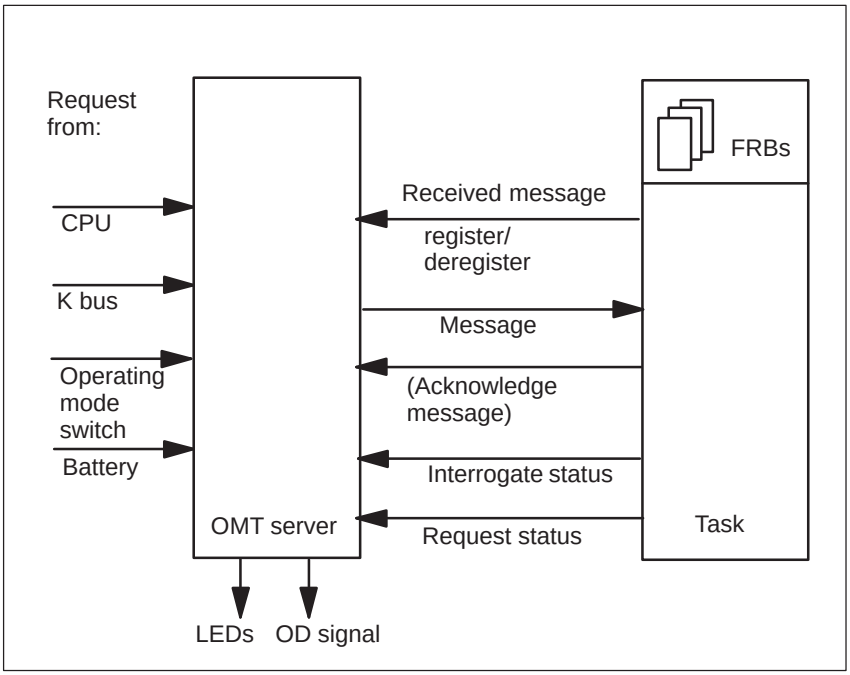


Figure 5-6 Overview of the Functions of the OMT Server

5.13 Interrogating and Changing Operating Modes

Interrogating the Current Operating Mode

The following call is used to interrogate the current operating mode:

M7GetState(void)

The call returns an identifier for the current operating mode.

Requesting an Operating Mode

You can request various operating modes with your user program. However, you should be aware that the operating mode RUN can only be reached by an FM on request from a task if the associated CPU is also in the RUN mode and the operating mode switch of the FM is in the position RUN or RUNP.

If a request is made for the operating modes RUN or STARTUP, other tasks which are registered for the corresponding operating mode transition can reject the request. In this case, the requested transition is not made (in other words the RUN or STARTUP mode is not reached).

A request for the operating mode STOP is always accepted, independently of the current operating mode.

Programming an Operating Mode Request

Proceed as follows to request a particular operating mode from your user task:

1. Set up a function request block (FRB) in your data area for the message. The OMT server requires the FRB for the request. If necessary, identify (tag) the FRB with the following call:

   M7SetFRBTag

2. Request a new operating mode from the OMT server with the following call:

   M7RequestState(pTSFRB,State,Reason,MPrio)

   You must specify the address ***pTSFRB*** of the associated FRB. The parameter ***State*** specifies the identifier of the required operating mode and ***Reason*** is an identifier for the reason for the request. The parameter ***Mprio*** specifies the required priority of the message from the OMT server.

Note

The value of the ***Reason*** parameter of the call ***M7RequestState()*** is entered into the diagnostic buffer as the event ID.

3. Create a message queue for your program (if unavailable) and wait for the message with the following call:

   RmReadMessage

   The message M7MSG_REQ_FINISHED is sent by the OMT server if the requested operating mode has been reached and/or the request was rejected.

4. The call returns an error code which indicates whether the request was successful or not. Use the following macro to evaluate the error code:

   M7GetFRBErrCode(pTSFRB)

Registering for Notification of an Operating Mode Transition

Proceed as follows to register your task with the OMT server to receive automatic notification when a particular operating mode has been reached:

1. Set up a function request block (FRB) in your data area for the message. The OMT server requires the FRB for the request. Identify (tag) the FRB with the following call

   M7SetFRBTag

2. Register your task with the OMT server for notification of the OMT event with the following call:

   M7LinkState(pTSFRB,State,MPrio)

   The call must be specified with a pointer ***pTSFRB*** to the associated FRB. The ***State*** parameter specifies the mode which you want to be notified about. The parameter ***Mprio*** specifies the required priority of the message from the OMT server.

3. Create a message queue for your program (if unavailable) and wait for the message with the following call:

   RmReadMessage

   The message M7MSG_REQ_STATE is sent by the OMT server when the requested operating mode has been reached.

4. Evaluate the received message and branch accordingly in your task to react to the operating mode which has been reached.

   The following macros make it easier to evaluate the message sent by the OMT server:

   M7GetTSType(pTSFRB) or ***M7GetTSReason(pTSFRB)***

These calls return the current operating mode and/or the reason for the new operating mode (diagnostic information), respectively.

5. Repeat steps 3 and 4 (loop in your C program).

Registering for Notification of an Operating Mode Transition Request

Proceed as follows to register your task with the OMT server to receive automatic notification when a request for a particular operating mode transition has been received by the OMT server, for example the transition STOP -> STARTUP:

1. Set up a function request block (FRB) in your data area for the message. The OMT server requires the FRB for the request. Identify (tag) the FRB with the following call

   M7SetFRBTag

2. Register your task with the OMT server for notification of the OMT event with the following call:

   M7LinkTransition(pTSFRB,Transition,MPrio)

   The call must specified with a pointer ***pTSFRB*** to the associated FRB. The ***Transition*** parameter specifies the operating mode transition which you want to be notified about. The parameter ***Mprio*** specifies the required priority of the message from the OMT server.

3. Create a message queue for your program (if unavailable) and wait for the message with the following call:

   RmReadMessage

   The message M7MSG_REQ_TRANSITION is sent by the OMT server when the requested operating mode has been reached.

4. Evaluate the received message and branch accordingly in your task to react to the operating mode transition request which has been requested (for example the transition STOP -> STARTUP).

   The following macros make it easier to evaluate the message sent by the OMT server:

   M7GetTSType(pTSFRB) or ***M7GetTSReason(pTSFRB)***

   These calls return the requested operating mode transition and/or the reason for the request, respectively.

5. Acknowledge the message with the following call:

   M7ConfirmTransition(pTSFRB,AllowTransition)

   The parameter ***AllowTransition*** (TRUE or FALSE) is used to acknowledge the transition. This flag can be used to prevent a transition to the operating modes STARTUP or RUN.

The OMT server only makes the change to STARTUP or RUN if you acknowledge the message with ***AllowTransition = TRUE***. If you specify AllowTransition=FALSE, the transition is not allowed and the original operating mode is retained.

Requests for a transition to the operating mode STOP are always accepted independently of the acknowledgement.

6. Repeat steps 3 to 5 (loop in your C program)

Notification of Aborted Transitions

When the STOP-to-STARTUP transition is rejected (***M7ConfirmTransition(.. AllowTransition=FALSE***), then no M7MSG_STATE message is issued upon reaching the STARTUP state.

Tasks that have registered with the OMT Server are sent an M7TRANS_STARTUPSTOP and an M7STATE_STOP message if a STOP-STARTUP transition could not be executed.

Tasks that have registered with the OMT Server are sent an M7TRANS_RUNSTOP message if a STARTUP-RUN transition could not be executed.

Deregistering the Notification of OMT Events

If a task no longer requires notification from the OMT server, it can deregister notification with one of the following calls:

M7UnLinkState(pTSFRB) or ***M7UnLinkTransition(pTSFRB)***

The two calls are used to deregister the notification of an operating mode transition and/or an operating mode transition request, respectively.

Note

Before a task which handles OMT events terminates, it must deregister all events which are registered with the OMT server.

Determining the Reset Cause

With the following call a task can determine the cause of the M7's last reset:

M7GetResetCause(pState)

The call must be specified with a pointer to ***pState***, which will contain the returned reset cause. The reset may be caused by power off, mode switch operation or watchdog.

Example of Using the OMT Server Functions

In the example program contained in the file *bzue.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API, the main task registers with the OMT server for notification of a transition to the operating mode RUN and a request for an operating mode transition from RUN to STOP.

In the processing loop, the task waits for messages from the OMT server and branches correspondingly.

On receiving a terminate task message, the task requests STOP mode from the OMT server. After reaching STOP mode, notification is deregistered and the task is terminated.

5.14 Registering and Deregistering for Battery Alarms

The functions of the OMT server also include the cyclic monitoring of the buffer battery. If the battery voltage drops below a limiting value, the OMT server enables the BAF LED (battery fault LED). The BAF LED is extinguished again when the battery is replaced.

In addition, your user task can get information from the OMT server on the battery condition in order to react accordingly.

Registering for Notification of Battery Faults

Proceed as follows to register your task with the OMT server to receive automatic notification when the battery voltage drops below a limiting value:

1. Set up a function request block (FRB) in your data area for the message. The OMT server requires the FRB for the request. Identify (tag) the FRB with the following call
2. Register your task with the OMT server for notification of battery faults with the following call:

   M7LinkBatteryFailure(pBAFFRB,MPrio)

   The call must specified with a pointer ***pTSFRB*** to the associated FRB. The Transition parameter specifies the operating mode transition which you want to be notified about. The parameter ***Mprio*** specifies the required priority of the message from the OMT server.
3. Create a message queue for your program (if unavailable) and wait for the message with the following call:

   RmReadMessage

   If the battery voltage drops below a limiting value before or during processing of the FRB, the task receives a message of type M7MSG_BATTERY_FAILURE.
4. Check whether the received message is of the above type and branch in your task accordingly. The message M7MSG_BATTERY_FAILURE does not need to be acknowledged to the OMT server.

Deregistering the Notification of Battery Errors

If a task no longer requires notification from the OMT server of battery faults, it can deregister notification with the following call:

M7UnLinkBatteryFailure(pBAFFRB)

The call must be specified with a pointer ***pBAFFRB*** to the FRB to be deregistered.

Note

Before a task which handles OMT events terminates, it must deregister all events which are registered with the OMT server.

5.15 General Information on the Free Cycle Server

The free cycle server provides the functionality of cyclic processing for the M7 system, in other words it provides equivalent functionality to OB1 in a SIMATIC system. Tasks can register equivalent functions with the FC server.

In addition, the FC server also generates the cycle control point, which is used to link and unlink S7 objects and to update process images. It also monitors the maximum admissible cycle time (cycle overflow) and controls the minimum cycle time.

The FC server is also responsible for the STARTUP functions.

Functional Sequence of the Free Cycle

The following figure shows the functional sequence of the free cycle:

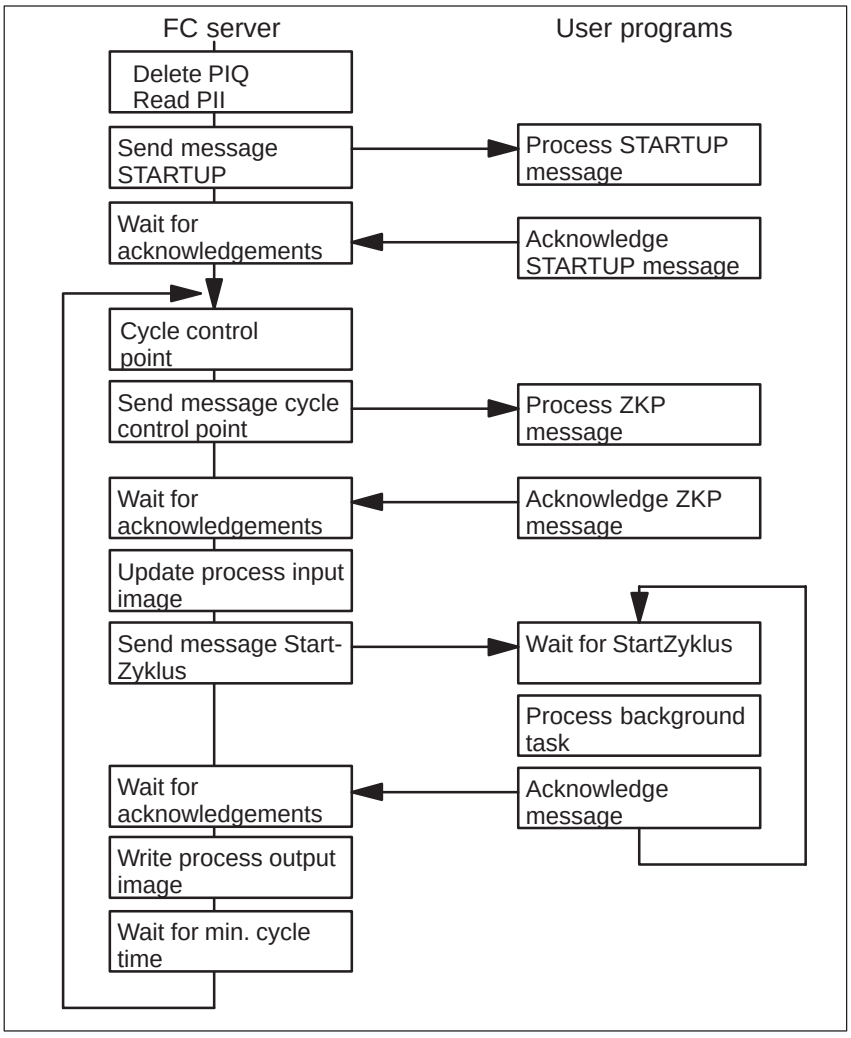


Figure 5-7 Functional Sequence of the Free Cycle

Responsibilities of the FC Server

The FC server carries out the following tasks, in this order:

- Implementation of the STARTUP functions:
  STARTUP is the operating mode where registered tasks carry out initialization and other checks. The FC server initiates the following activities during STARTUP:
  - Delete PIQ and update PII:
    The first thing that the FC server does during STARTUP is to delete the process output image and update the process input image.
  - Send STARTUP messages:
    The FC server then sends the start up message to all registered programs, causing them to execute their initialization routines.
  - Wait for acknowledgements:
    All registered programs must acknowledge the end of their initialization sequences by sending and end-of-STARTUP message to the FC server. The FC server only makes the transition to the RUN mode and provides the first cycle control point is when all registered programs have acknowledged STARTUP.
- Provision of the cycle control point (CCP):
  In the functional sequence of the SIMATIC S7 system, the cycle control point defines that point where S7 objects can be linked or unlinked (in other words activated or deactivated) by the object management system.

  In an M7 system, the linking of objects takes place under control of the FC server as follows: programs which are registered for the cycle control point receive a message from the FC server to indicate the time point when they are allowed to create new S7 objects and to modify or delete existing ones.

  The following activities are carried out by the FC server during the CCP:
  - Send the CCP message:
    The FC server sends the CCP message to all registered programs. The programs are now allowed to create, modify or delete S7 objects.
  - Wait for acknowledgements:
    All registered programs must acknowledge the end of their processing to the FC server after they have completed the required actions.
- Provision of the free cycle
  The free cycle starts after the CCP. The FC server carries out the following activities within the free cycle:
  - Read (update) the process input image:
    After all programs have acknowledged the CCP message, the FC server updates the process input image.

– Send the StartZyklus (start cycle) message:
  After updating the process input image, the FC server sends the StartZyklus message to all registered programs to request them to carry out their background activities.

– Wait for acknowledgements:
  All registered programs must acknowledge the end of their activities with an end of cycle message to the FC server. In order to be able to detect programs which are not reacting properly, waiting for the acknowledgements has a timeout equal to the maximum cycle time.

  If the maximum cycle time is exceeded (cycle overflow), a transition is made from RUN to STOP. An FC server call is available for retriggering the cycle time.

– Update the process output image:
  After the FC server has received acknowledgements from all registered programs, it sends the process output image to the I/O modules.

– Waiting for the minimum cycle time:
  In order to guarantee the minimum cycle time, after updating the process output image, the FC server waits until the jump to the next cycle control point takes place.

– Monitoring the cycle time:
  During the free cycle, the FC server continuously monitors the elapsed cycle time. If the maximum cycle time is exceeded (cycle overflow), by default an automatic transition is made to STOP mode.

  The default reaction (transition to STOP) can be prevented by registering a task with the FC server for cycle overflow.

Note

With the M7 system, there is no guarantee of consistency of the process I/O images for tasks which are not registered with the free cycle server.

5.16 Registering and Deregistering with the Free Cycle Server

Registering for Notification by the FC Server

Proceed as follows to register your task with the OMT server to receive automatic notification of the cycle control point and/or the free cycle:

1. Set up a function request block (FRB) in your data area for the message. The FC server requires the FRB for the request. Identify (tag) the FRB with the following call

   M7SetFRBTag

2. Register your task with the FC server for notification of the cycle control point and/or the free cycle with the following call:

   M7LinkCycle(pFSCFRB,Cycle,MPrio)

   The parameter ***MPrio*** specifies the required priority of the message from the FC server.

   The call must specified with a pointer ***pFSCFR*** to the associated FRB. The ***Cycle*** parameter specifies the FC event which you want to be notified about.

   The following values can be specified for ***Cycle***:

   M7S_CYCLECONTROLPOINT: Register the task for the cycle control point.

   M7S_FREECYCLE: Register the task from the free cycle.

   M7S_STARTUPCYCLE: Register the task for the STARTUP mode.

   M7S_CYCLEOVERFLOW: Register the task for cycle overflow.

   If you register at least one task for cycle overflow, the system will no longer make an automatic transition to STOP mode if cycle overflow occurs. If you want to go to STOP mode nonetheless, you must request STOP mode explicitly from the OMT server with the call ***M7RequestState()***.

3. Create a message queue for your program (if unavailable) and wait for the message with the following call:

   RmReadMessage (see chapter 4.12).

4. Evaluate the received message and branch accordingly in your C program.

   All messages from the FC server have the message type M7MSG_CYCLE. The following macro makes it easier to evaluate the message sent by the OMT server:

   M7GetFSCType((M7FSCFRB_PTR)pmessage_par)

   After calling ***RmReadMessage*** the parameter ***pmessage_par*** points to an FRB registered with a previous call to ***M7LinkCycle. pmessage_par*** is then passed to the ***M7GetFSCType*** macro in order to evaluate the FRB. The macro returns the parameter ***Cycle*** from the FRB in the message.

5. After processing you must acknowledge the message with the following function:

   M7ConfirmCycle(pFSCFRB)

6. Repeat steps 3 to 5 (loop in your C program).

Deregistering the Notification by the FC Server

If a task no longer requires notification from the FC server, it can deregister notification with the following call:

M7UnLinkCycle(pFSCFRB)

The call must be specified with a pointer ***pFSCFR***B to the FRB to be deregistered.

Note

Processing of the cycle overflow event should always be carried out by a task with sufficient priority which is not registered for the free cycle. This allows appropriate and successful reaction to cycle overflow caused by locked-up cyclic tasks.

Further cycle time monitoring is disabled until your task has acknowledged the cycle overflow message. This prevents the cycle overflow message from being sent repeatedly, which could result in overflow of the message queue.

Retriggering the Free Cycle

A program is also able to retrigger the free cycle in order to prevent a cycle overflow error, which could otherwise cause a transition to the STOP mode.

The following call is used to retrigger the free cycle:

M7RetriggerCycle()

This call resets the cycle time counter to 0, in other words cycle time monitoring starts at the beginning again.

Note

Before a task which handles FC events terminates, it must deregister all events which are registered with the FC server.

Example Program for the FC Server

In the example program in the file *fzserv.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API, the initialization section in the main task registers the task with the FC server for notification of STARTUP and of the free cycle.

In the main processing loop, the task waits for an M7MSG_CYCLE message and branches to the appropriate routine according to the message type (STARTUP or free cycle). On receiving the terminate task message, the task is deregistered with the FC server, the message queue is deleted and the task is terminated.

5.17 General Information on the Diagnostic Server

Modules which support diagnostics can recognize, notify and protocol errors within the automation system, its wiring and the wiring to the process being automated.

This ensures that dangerous conditions of the plant can be avoided as far as possible and also reduces downtime due rapid fault detection and analysis.

The following figure shows the diagnostic concept of the M7. It only shows those components which are involved in fault recognition and further processing of diagnostic events.

The central location for handling diagnostic events is the diagnostic server. It receives and stores all diagnostic events and processes them.

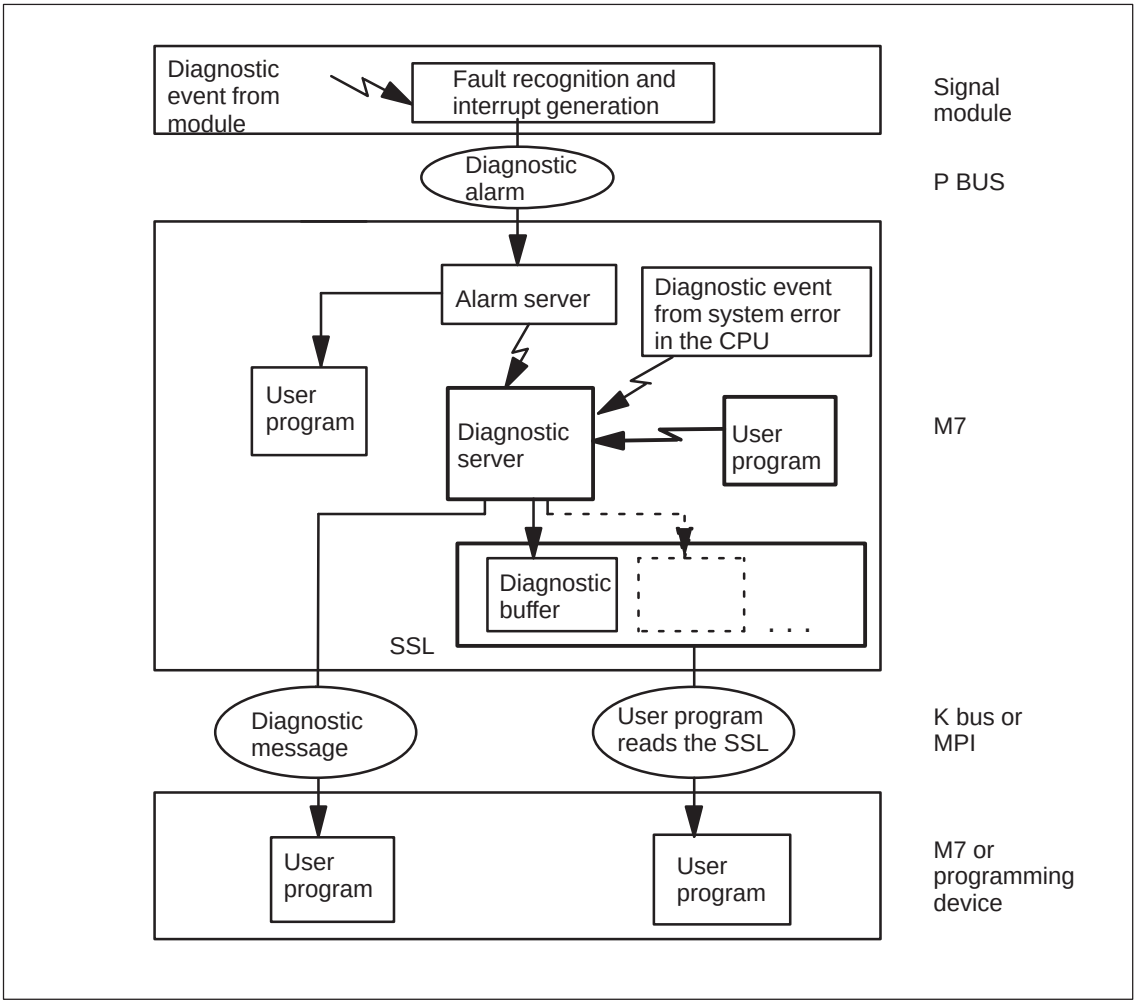


Figure 5-8 Diagnostic Concept of the M7

Sources of Diagnostic Events

Diagnostic events can be generated in three places:

- I/O modules: Diagnostic events from I/O modules, for example “wire break”, are notified by a diagnostic alarm to the M7 alarm server (see section 5.3). This notifies the diagnostic server and notifies the programs which have registered for this type of diagnostic alarm.
- Operating system of the CPU: System errors on the CPU, for example buffer voltage is too low or operating mode transitions are notified by the module’s operating system to the diagnostic server directly.
- User program: You can also notify the alarm server directly from your user program with the M7 API call ***M7WriteDiagMode()*** in order to send a user message (for example valve 5 does not open) (see section 5.18).

Reaction of the Diagnostic Server

After the diagnostic server receives an event message, it carries out the following actions:

- The event is given a time stamp
- The event is entered in the diagnostic buffer. Further diagnostic status data in the SSL (System Status List) is also updated.
- The diagnostic entry is sent using configured connections via MPI or K bus to the registered programs.

  The M7 API call ***M7DiagMode()*** is used by a program on the M7 to register itself with a communication partner to receive diagnostic event messages (see section 5.19).

Diagnostic Buffer

The diagnostic buffer is a buffered memory area on the M7 programmable controller which is organized as a ring buffer and is not cleared by RESET. It contains all diagnostic events in chronological order. This allows for example system errors to be evaluated a long time after they have occurred. The diagnostic buffer is part of the SSL.

System Status List

The SSL (System Status List) contains all available information on the current status of an S7 CPU or an M7 programmable controller. The SSLs are virtual lists, in other words particular parts of them are available statically (for example the diagnostic buffer) and other parts are only created dynamically when the information is requested.

The SSL and thus the diagnostic buffer too can be read using configured connections via MPI from a communication partner (programming device, S7 CPU or M7). The M7 API call ***M7SZLRead()*** is used for reading the SSL (see section 5.20).

5.18 Writing a User Entry to the Diagnostic Buffer

The call ***M7WriteDiagnose()*** is used to enter user information into the diagnostic buffer. You can also specify whether the user entry should be sent to all registered tasks or not.

The contents of the diagnostic buffer of an M7 or an S7 can be read with the call ***M7SZLRead()*** (see Section 5.20).

These functions require S7 single-system connections that are configured with STEP 7 (see chapter 8).

Structure of an Entry in the Diagnostic Buffer

An entry in the diagnostic buffer of an M7 programmable controller is structured as follows:

Table 5-2 Structure of an Entry in the Diagnostic Buffer of an M7

Byte	Contents
1 and 2	Event ID
3, 4, 5 and 6	Reserved
7 and 8	Extended information 1
9, 10, 11 and 12	Extended information 2
13 to 20	Time stamp

Event ID

Each event is assigned an event ID. The event ID contains the following information:

- Event class (4 bit): The event class specifies for example whether the event was triggered synchronously (I/O access error) or asynchronously (cycle overflow), from within the system or by a user task (valid are only the values 0X0A and 0X0B), etc.
- Identifier (4 bit): The identifier gives on the one hand information on the location of the error:
  - External errors (for example distributed battery failure in ER)
  - Internal errors (for example cycle overflow)

  and on the other hand information on the type of event. A differentiation is made between
  - New events (arriving)
  - Old events (departing)
- Event number (8 bit): Within each event class the event number uniquely specifies every possible event which can occur.

Extended Information

The extended information contains additional information on the event. The contents of the extended information can be different for each event. When you create a user entry, you are free to determine the structure of the entry as required.

Time Stamp

The time stamp has the type M7TIME_DATE. It characterizes the time point of the event which has occurred.

Writing a User Entry

The following M7 API call is used to write a user entry to the diagnostic buffer:

M7WriteDiagnose(Type,Eventnumber, Direction, ZI1,ZI23,Send)

You must specify the event type (internal or external) in ***Type*** and a unique event number in ***Eventnumber***.

The parameter ***Direction*** is used to differentiate between events which are departing (***Direction***=FALSE) and events which are arriving (***Direction***=TRUE).

ZI1 and ***ZI23*** contain the extended information. You can specify with the ***Send*** parameter whether the diagnostic entry should be sent to registered programs (***Send***=TRUE) or not (***Send***=FALSE).

5.19 Registering and Deregistering Notification of Diagnostic Messages

Programs on the M7 programmable controller can register themselves with communication partners to receive notification of diagnostic events. If the specified diagnostic event occurs, the diagnostic entry is sent to all registered programs in addition to storing it in the diagnostic buffer.

This allows a central module to get notification of operating errors throughout the plant (for example module defect) in order to monitor the correct functioning of the whole plant.

Program Steps: Registering for Notification of Diagnostic Events

This function requires a S7 single-system connection configured with STEP7 (see chapter 8). Proceed as follows to register a program to receive diagnostic messages from a communication partner:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   The function ***M7KInitiate()*** opens an application link to the remote communication partner. You must enter the host address of the partner in the ***pHostAddr*** parameter.

   Following successful application link establishment, the parameter ***pConnID*** contains a valid application link ID.
3. Create an FRB CommFRB of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag()
4. Register for notification of diagnostic events with the following call:

   M7DiagMode(flags,ConnID,pCommFRB,pszUserName,MPrio)

   This call registers the task with username ***pszUserName*** with the server programmable controller.

   You must also specify the application link ID ***ConnID*** and the address of the associated FRB ***pCommFRB***.

   The parameter ***flags*** (can be OR'ed) specifies the type(s) of the diagnostic events to notify. Flags which are not set mean that this event type is not notified.

   A_SYSMSG: All system diagnostic entries are notified (no user specific entries).

   A_USERMSG: Only user entries are notified.

   A_BESYMSG: Only operating mode transitions with the associated "virtual device status" are notified. This corresponds to the service "Unrequested notification of device status" (see section 8.11).

Receiving Diagnostic Messages

After you have registered your task for notification of diagnostic events, the communication partner sends diagnostic entries to the communication driver of the local programmable controller asynchronously.

Proceed as follows to copy this data, which is notified asynchronously to the user program, from the address area of the communication driver to the prepared receive buffer in the program:

1. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

   The communication driver generates a message of type M7MSG_DIAG_MSG for each received data packet. The parameter ***pMessageParam*** refers to the FRB which was specified when registering for notification.

2. Determine the associated request number ***nRequest*** and the user-defined tag from the referenced FRB using the macros ***M7GetCommRequest()*** and ***M7GetFRBTag()***.

   The request number is used to differentiate between the different diagnostic events.

   DIAG_SYS_USER_MSG: Request number for system diagnostic entries and user-specific diagnostic entries.

   DIAG_USERMSG: Request number for user entries.

   DIAG_BESYMSG: Request number for " Not requested notification of device status".

3. Use the following synchronous call to copy the received data from the address area of the communication driver to the receive buffer in your user program:

   M7KEvent(ConnID,nRequest,pBuffer,nBufsiz,pnBytes)

   You must specify a valid application link ID ***ConnID***, the request number ***nRequest***, the address ***pBuffer*** and the size ***nBufsiz*** of the receive buffer.

   The call returns the number of actually copied bytes in ****pnBytes***.

   If a message is not present for the specified application link ID ConnID and request number ***nRequest***, the call returns without error with ****pnBytes***=zero.

4. Repeat steps 1 to 3 (loop in your C program).

Deregistering Notification of Diagnostic Messages

If a task no longer requires notification of diagnostic events, it can deregister the associated FRB again with the diagnostic server with the following call:

M7DiagMode(flags,ConnID,pCommFRB,pszUserName)

This call deregisters the task which was registered with the server with user name ***pszUserName***. In the call you must specify ***flags=A_ZERO_FLAG***.

You must also specify the application link ID ***ConnID*** and the address of the associated FRB ***pCommFRB***.

Closing the Application Link

When the application link to the remote station is no longer required, you can close it again with the following call:

M7KAbort(ConnID)

You must specify the ID ***ConnID*** of the application link to be closed.

Note

Before a task which receives diagnostic messages terminates, it must deregister all events which are registered with the diagnostic server.

5.20 Reading the System Status List (SSL)

Programs on the M7 can read the system status list from a communication partner. In addition to the diagnostic buffer, the system status list contains current status data for all modules and connected I/O modules.

This allows a central module to get information on all modules throughout the plant in order to check the correct functioning of the whole plant.

Structure of the SSL

An SSL generally consists of several SSL partial lists, whereby each partial list can contain several sections.

A section within an SSL partial list consists of the following two components:

- Header,
- Data record.

Structure of the Header

The header of a section in an SSL partial list is structured as follows (see the Appendix of the User Manual for further information):

Table 5-3 Header Structure of a Section in an SSL Partial List

Byte	Contents
1 and 2	SSL ID
3 and 4	Index
5 and 6	Length of a data record
7 and 8	Number of data records

SSL ID

The SSL ID is one word in length and uniquely identifies an SSL partial list for a particular module within the S7 system. The SSL ID has the following structure:

- Module type ID (4 bit): The module type ID specifies which module (CPU, CP or FM) you want to access within the S7 system.
- Partial list number (4 bit): The partial list numbers and their significance are dependent on the respective SSL. The partial list number is used to specify which part of the SSL you want to read.
- SSL number (8 bit): The SSL number is used to specify which SSL you want to read.

  In general, depending on the type of module, not all of the possible SSLs will be available.

Index

If the SSL partial list contains several sections, the index is used for addressing purposes. The index is 1 word in length (16 bit) and its significance is dependant on the respective SSL partial list.

If the SSL partial list only has one section (for example the SSL partial list with the number 0xF), the address index is not evaluated when reading the data records.

Length of a Data Record

This entry specifies the length of a data record in words (16 bit). The length of a data record is constant within the same SSL, in other words it is independent of the particular SSL partial list and/or section. The length of a data record is dependent on the type of information which is stored in the SSL.

Note

In each SSL, an exception is made by the SSL partial list with the number 0xF.

Within each SSL, this SSL partial list, which only has one section, contains the sections headers of all SSL partial lists in the SSL.

Accordingly, a data record in this SSL partial list is always 4 words long (8 bytes) and has the structure shown in Table 5-3.

Number of Data Records

This entry specifies the number of data records in the respective section.

Data Records

The data records contain the actual information units. The structure and significance of the data words within the data records are dependent on the respective SSL.

Note

A detailed description of the system status list of an M7 or S7 programmable controller can be found in the manuals:

“System Software for M7-300/400 Installation and Operation” and “System Software for S7-300/400 System and Standard functions”.

Procedure: Reading the SSL

This function requires a S7 single -system connection configured with STEP7 (see chapter 8). Proceed as follows to read parts of the SSL of a remote communication partner:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   The function ***M7KInitiate()*** opens an application link to the remote communication partner. You must enter the host address of the partner in the ***pHostAddr*** parameter.

   Following successful application link establishment, the parameter ***pConnID*** contains a valid application link ID.
3. The following call is used to read the SSL:

   M7SZLRead(flags,ConnID,pBuffer,nBufsiz,SZLID,index,pnBytes)

   This call reads the part of the SSL specified with ***SZLID*** and index from the communications partner specified with ***ConnID***.

   The parameter ***flags*** is used to specify whether the data records which are read should be stored in a file or in the main memory.

   If the flag A_FILE is set, then parameter ***pBuffer*** specifies the path and name of a file. If A_FILE is not set, then ***pBuffer*** specifies the address and ***nBuffer*** the length of a receiving buffer in the main memory.

   The call returns the number of actually read bytes in the parameter ***pnBytes***.
4. If data no longer needs to be exchanged with the remote communication partner, you can close the application link. To do this, program the function call:

   M7KAbort(ConnID)

   You must transfer the ID ***ConnID*** of the application link to the call.

5.21 Controlling the User LEDs

M7-300 modules have a special user LED for display purposes; M7-400 modules have a second user LED.

You can control these LEDs from your user program in order for example to indicate a particular processing step.

Controlling the User LEDs

The following M7 API call is used to control the user LEDs:

M7SetUserLED(Led,Mode)

The parameter ***Led*** is used to specify the number of the user ***LED***. For the M7-400, ***LED*** can be either M7USERLED1 or M7USERLED2; for the M7-300 only M7USERLED1 is valid.

The ***Mode*** parameter is used to switch on or off the specified LED with the constants M7LED_ON and M7LED_OFF. The ***Mode*** parameter can also be used to control the blinking frequency by ORing with M7LED_FLASHSLOW or M7LED_FLASHFAST.

For example, the command to switch on the LED and activate fast blinking (2 Hz) is:

M7SetUserLED(M7USERLED1,M7LED_ON | M7LED_FLASHFAST)

Table 5-4 LED Control Modes

Mode	Effect
M7LED_OFF	switch off LED
M7LED_ON	switch on LED continuously
M7LED_ON | M7LED_FLASHSLOW	switch on LED, blinking at 0.5 Hz
M7LED_ON | M7LED_FLASHFAST	switch on LED, blinking at 2 Hz

5.22 Converting the Data Format between Intel and SIMATIC Byte Order

When processing S7 objects or data from process I/Os, conversion between Intel and SIMATIC data format is necessary when accessing larger data structures (for example with the calls ***M7Read***, ***M7Write***, ***M7LoadDirect*** or ***M7Storedirect***).

It is particulartly important to ensure that data within S7 objects which will be used for communication services (for example operator interface) is always stored in SIMATIC data format (Motorola byte order).

This section explains the difference between these two data formats and describes C macros for converting the data format in both directions.

Representation of S7 Data in M7 Memory

In an M7 automation computer, data is stored in Intel byte order. For data records of more than 1 byte in length (for example word, double word), the structure of the data differs from that in S7 objects (including data from process I/Os).

If it is necessary to exchange or compare data between S7 objects (for example the data record for an alarm descriptor) and M7 memory, the different data formats must be taken into account.

The following figure shows how data is stored in word and double word format in an M7 automation computer and in an S7 object:

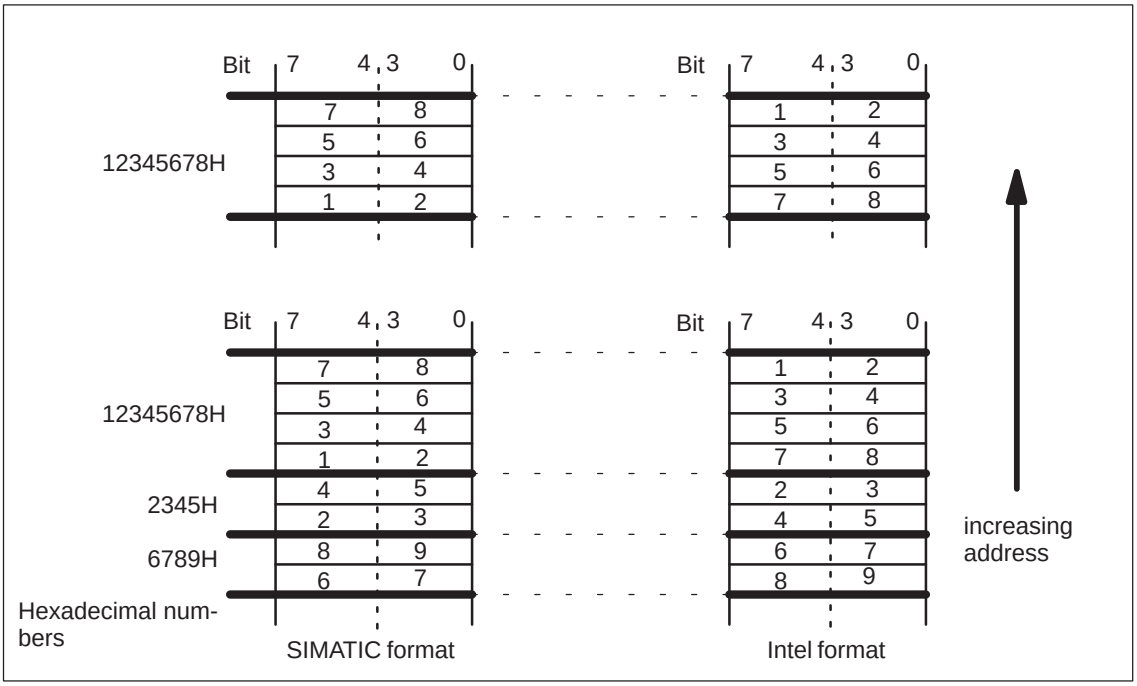


Figure 5-9 Representation of Data in SIMATIC and Intel Formats

Converting the Data Format

The following C macros are provided for converting data from Intel to the SIMATIC byte order and vice versa:

- ***M7_SWAP_WORD (x)*** for converting a data **word** in both directions
- ***M7_SWAP_DWORD (x)*** for converting a **double word** in both directions

Example of Conversion

The following example showing part of a program listing illustrates conversion of the data in an Intel field to the SIMATIC data format and the conversion of the data in an S7 field in the Intel data format.

```
#include    <m7api.h>
.
.
UWORD  Intel_w_feld[10];
UWORD  S7_w_feld[10];
UDWORD Intel_dw_feld[10];
UDWORD S7_dw_feld[10];

int i;                  /* Loop variable          */

main ()
{
                        /* Convert Intel field    */
                        /* and store in           */
                        /* S7 field               */
 for (i = 0; i < 10; i++)
 {
  s7_w_feld [i] = M7_SWAP_WORD(Intel_w_feld [i]);
  s7_dw_feld [i]= M7_SWAP_DWORD(Intel_dw_feld [i]);
 }
 .
 .
                        /* Convert S7 field       */
                        /* and store in           */
                        /* Intel field            */
 for (i = 0; i < 10; i++)
 {
  Intel_w_feld [i]  = SWAP_WORD(S7_w_feld [i]);
  Intel_dw_feld [i] = SWAP_DWORD(S7_dw_feld [i]);
 }
 .
 .
}
```

Accessing Process I/Os

6

Chapter Overview

6.1 General Information on Communication with Process I/Os

Process control and automation applications naturally have close ties to system components for the input and output of process signals in order to acquire the necessary process parameters and/or to output signals to the process in order to control it. Generally speaking, both analog and digital signals are necessary.

The SIMATIC S7 system and thus an M7 automation computer too are provided with a special I/O bus for the input and output of process signals (P bus) which is connected to the I/O modules. Further I/O modules can connected to the local SIMATIC module bus (ISA Bus) using an EXM extension module. The P bus offers a number of other functions in addition to accessing process I/Os, for example processing alarms and configuring modules.

In addition to mechanisms for accessing local process I/Os (P bus I/O modules and IF submodules), SIMATIC M7 also allows transparent connection to distributed I/O modules using the PROFIBUS DP bus system. From the point of view of administration, file transfer and alarm handling, distributed I/O modules are handled in the same way as local I/Os.

Those special functions of the distributed I/Os which are not supported by local I/Os are handled by additional M7 API calls.

Structure of an S7/M7-300 System

When categorizing the P bus functions, it should be noted that the modules attached to the P bus can be separated into two groups. One group is for CPU modules with master functions on the P bus and the other group is for I/O modules with purely slave functions.

The following figure uses the System S7-300 to illustrate the structure and interconnection of process I/Os. In a system like this, a differentiation is made between the CPU operation and operation as a function module (FM) only, both with and without a local bus segment. With the System S7-300, only one local bus segment can be configured per row of modules.

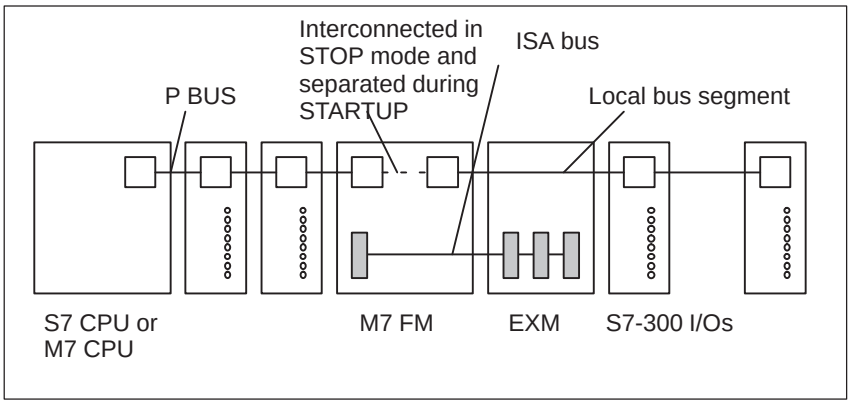


Figure 6-1 The Basic Structure of an S7-300 System

Master and Slave Behavior

If the M7 automation computer functions as a master (M7 CPU) in an S7-300 system, it has the same rights and duties as an S7 CPU and is responsible for example for initializing all of the modules on the right hand side during startup.

An M7 FM normally has both master and slave functionality. After initializing the system modules, the FM detaches its local bus segment and then operates in accordance with its configuration to the right hand side as master (pseudo-CPU) and to the left hand side as a slave.

This allows an M7 FM to access the standard I/O modules via its local bus segment just like a normal CPU module. Although the FM is still subordinated to the CPU, in normal operation the CPU can no longer access the modules which are now “owned” by the FM.

If the FM does not have a local bus segment, it functions as a master towards its local I/O modules (interface submodules) and as a slave on the P bus.

In either case the FM makes available the user data of its parent CPU in the form of input and output signals.

P Bus Master Functions

The P bus functions of a master include:

- Access to the process I/Os (see section 6.3)
- Initializing the transfer of data records (see section 6.7)
- Receiving and processing process and diagnostic alarms (see section 5.5)

P Bus Slave Functions

The P bus functions of a slave include:

- The provision of process signals (user data, see section 6.5)
- Data record transfer to/from an S7 and/or M7 CPU (see section 6.6)
- Sending process and diagnostic alarms (see section 5.6).

6.2 Accessing the Process I/Os

Internally, access to process I/Os takes place via the process I/O driver. It hides the different characteristics of each of the I/O bus systems.

The following figure is a schematic diagram of the structure of the process I/O driver:

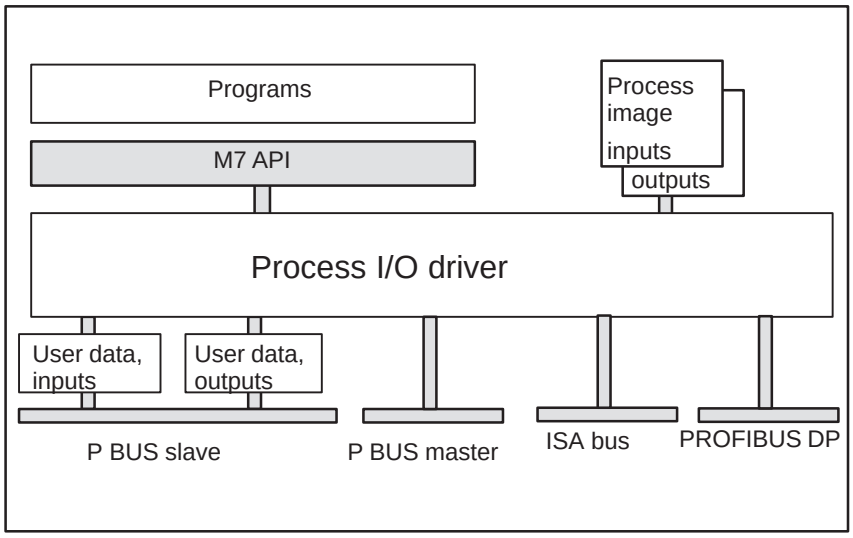


Figure 6-2 Accessing the Process I/Os

Direct Access to the Process I/Os

The process I/O driver implements the direct access to the signal converters of the I/O modules. Within the driver, an internal check is made to determine if the process signal can be accessed or not and the specified logical address is converted to the corresponding geographical address. The process I/O driver then determines whether the specified signal converter can be accessed via the P bus, ISA bus or PROFIBUS DP. Finally, access to the I/O module's signal converter takes place through the appropriate bus system.

Particularly when accessing local process I/Os on the ISA bus, the calculating times needed to convert the logical address to the associated geographical address is not insignificant in comparison to the time required to access the I/O module's signal converter itself.

For this reason, in addition to the direct I/O access to interface submodules, the M7 automation computer also allows access via I/O descriptors. I/O descriptors are used to store the address information required for direct access.

Process I/O Images

In addition to directly accessing the signals on the process I/Os, the M7 automation computer is also able to access the process I/O images. Process I/O images are data structures in memory.

Process input images:
At the begin of each CPU cycle, the status of the physical inputs of the I/O modules is interrogated and then stored in the process input image. In this way, in addition to the direct but slow access to the process I/Os, you can also reference process data via the process image.

The address of the process input image corresponds to the start of the I/O area of the inputs. For System 300/400 the size of the process input image is 256/512 bytes, respectively.

Process output images:
Instead of the direct but slow access to the physical outputs of the I/O modules, process data can be initially stored in the process output image and then transferred to the I/O modules at the end of each CPU cycle.

The address of the process output image corresponds to the start of the I/O area of the outputs. For System 300/400 the size of the process output image is 256/512 bytes, respectively.

Note

The inputs and outputs of modules having addresses higher than
256 in M7-300 and
512 in M7-400
are not situated within the process image. Only direct access operations can be used for these signals.

If you do not use process I/O image operations in your program, we recommend you to disable the update of the process image table in the SIMATIC Manager (in the Module's Properties "Cycle/Clock Memory" Tab). So the repeat accuracy of high priority tasks which execute direct I/O access is increased.

Process Image Segments

System S7-400 also implements the concept of process image segments for inputs and outputs. Process image segments are separate segments of the overall process image which can be independently updated, in other words they can be read in from process I/Os and/or written to process I/Os independently.

With a System S7-400 it is possible to configure up to 8 process image segments for input and 8 process image segments for output. Process image segments are set up using the graphical user interface of the SIMATIC Basic Package software.

Process I/O images have the advantage that the status of all inputs is determined at the same time point and the status of all outputs can be transferred to the process I/Os at the same time point. The reading of the input signals and the writing of the output signals can be initiated either by the free cycle server or by a user program.

User Data

User data provide the simplest way of communication between the CPU module and the underlying FM.

The user data area is subdivided into two parts. The input area can be read by an S7 CPU with a load command and the output area can be written with transfer commands.

An M7 RMOS32 program can access the user data area with appropriate M7 API calls in the same way as the I/O area of a process I/O module.

From the physical point of view, the user data area is situated in the dual port RAM of the M7 FM.

6.3 Reading and Writing Process I/O Data

The following section explains how to read or write process I/O data directly or by means of process I/O images.

Logical Starting Addresses

The logical starting addresses of the signal modules can be configured with the SIMATIC Manager. If configuration is not carried out explicitly, the default settings apply.

Please refer to the corresponding device manuals for information on default settings for the logical starting addresses of the signal modules in the System S7/M7-300.

SIMATIC/Intel Conversion

All process data is stored in the M7 automation computer in **SIMATIC** data format (Motorola byte order). For data records more than 1 byte in length, the structure of the data differs from that of the **Intel** data format (in other words Intel byte order) used for data in your C user program (see section 5.22).

When reading and writing process data in word and double word format, the corresponding M7 API functions convert the data automatically between the two formats. Accordingly, you can process this data just like any other data in your C user program.

In the case M7 API calls which specify the address of a buffer for the source and/or destination address, no automatic conversion is made between the two data formats. Accordingly, in before you can use this data in your C user program, you first need to carry out the necessary conversion explicitly using appropriate macros (see section 5.22).

Significance of the Process I/O Images

S7 programs are usually designed to access the process signals through the process I/O images which are stored in RAM. This allows faster processing than is possible with direct access to the I/O signals through the process I/O driver.

The operating system of an S7 CPU reads at the begin of each CPU cycle, in other words the beginning of the OB1, the current process input signals from the process modules and stores their status in the process input image. At the end of a CPU cycle, in other words after processing the OB1, the operating system transfers the current state of the process output image to the signal outputs of the I/O modules.

The process input image represents a snapshot of the process I/Os at a specified time and ensures that all programs get the same consistent image of the I/O signals.

Deleting and Updating the Process I/O Images

The M7 automation computer offers the same possibilities as an S7 CPU. The M7 API automatically sets up the two process images in memory during startup.

For all tasks that have registered themselves for the free cycle, the FC server provides the same services as the operating system of an S7 CPU. The process input image is automatically updated at the start of the free cycle by the FC server and the process output image is automatically transferred to the I/O modules at the end of the cycle.

However, analogously to an S7 CPU, you can also use appropriate M7 API calls in your user program to update the process input image and/or transfer the process output image to the I/O modules “manually”. This allows you to freely choose the time and also the locations in your program at which the updating and transferring is carried out.

If you don’t want to register any of your tasks with the free cycle server, you should reset the process images to a defined condition before the first access. Otherwise you run the risk of issuing random signals when the process output image is transferred to the I/O modules.

The following call is used to reset the process I/O images:

M7ClearPI(PIType)

The parameter ***PIType*** is used to specify whether you want to reset the process input image (***PIType*=M7IO_PII**) or the process output image (***PIType*=M7IO_PIQ**).

The process input image can be updated with the following call:

M7LoadPII(PIINo)

For an S7-400 system, the parameter ***PIINo*** is used to specify the number of the process input image segment (1...8) to update. If 0 is specified, the entire process input image is updated.

This parameter must always be specified as 0 for System S7-300.

The process output image can be output to the process I/Os with the following call:

M7StorePIQ(PIQNo)

For an S7-400 system, the parameter ***PIQNo*** is used to specify the number of the process output image segment (1...8) to output. If 0 is specified, the entire process output image is output.

This parameter must always be specified as 0 for System S7-300.

Accessing the Process I/O Images

The following M7_API calls are available for accessing process I/O images:

- Request/write bit:

  M7LoadBit ***/* request bit status */***

  M7StoreBit(..,Value) ***/* set bit to "Value" */***

- Read/write byte:

  M7LoadByte ***/* read byte status */***

  M7StoreByte(..,Value) ***/* set byte to "Value""Value" */***

- Read/write word:

  M7LoadWord ***/* read word status */***

  M7StoreWord(..,Value) ***/* set word to "Value" */***

- Read/write double word:

  M7LoadDWord; ***/* read double word status */***

  M7StoreDWord(..,Value); ***/* set double word to "Value" */***

The above M7 calls use logical addresses to access the process signals of the process images.

You can also use the M7 API calls of the S7 object server in order to read from or write to a process I/O image. In this case, you can also use the other services provided by the S7 object server (see section 7.4).

Direct Access to the Process I/Os

Alternatively, your user program can also access process data in the I/O modules directly. This is done with the following M7 API calls:

- Read/write block:

  M7LoadDirect(..,pBuffer) /* read block to "Buffer" */

  M7StoreDirect(..,pBuffer) /* write "Buffer" to block */

- Read/write byte:

  M7LoadDirectByte /* read byte status */

  M7StoreDirectByte(..,Value) /* set byte to "Value" */

- Read/write word:

  M7LoadDirectWord /* read word status */

  M7StoreDirectWord(..,Value) /* set word to "Value" */

- Read/write double word:

  M7LoadDirectDWord /* read double word status */

  M7StoreDirectDWord(..,Value) /* set double word to "Value" */

  The functions ***M7LoadDirect(..,pBuffer)*** and ***M7StoreDirect(..,pBuffer)*** do not carry out an automatic conversion of the data format from the SIMATIC byte order to the Intel byte order or vice versa.

The above M7 calls use logical addresses to access the process signals of the process images. Please refer to the corresponding device manual for information on the default assignment of logical addresses to each of the racks, slots and signal inputs/outputs in System S7/M7-300.

Automatic Updating of the Process Output Image

When directly accessing the output signals in the process I/Os, the associated process output image is automatically updated. The function ***M7StoreDirect()*** also causes the process output image to be updated simultaneously. In this case the process output image serves as a write-through cache for the process I/Os.

The following figure illustrates the function of the calls:

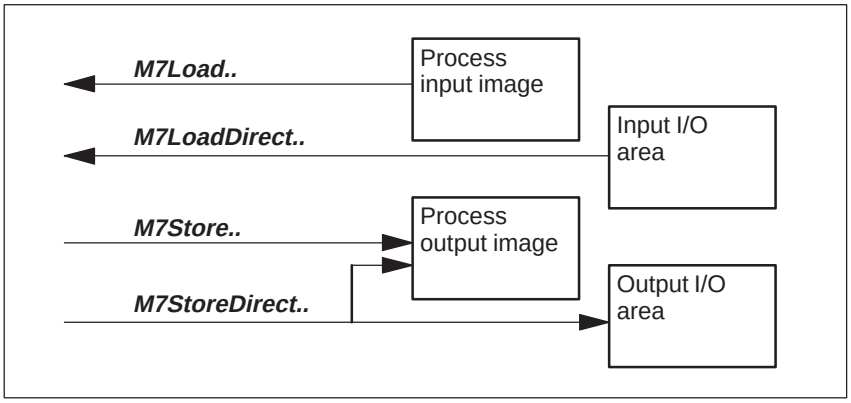


Figure 6-3 Automatic Update of the Process Output Image

I/O Access Error

You must implement responses to I/O access errors in your C program:

- With SIMATIC S7 the appropriated OB must be loaded
- With SIMATIC M7 I/O access errors are returned by the M7 API calls. The response must be included in a program routine.

If no processing routine is found, the S7 CPU and the M7 CPU behave differently:

- SIMATIC S7 reaction: the CPU goes to STOP and the event is entered in the diagnostic buffer.
- SIMATIC M7 reaction: the CPU remains in RUN and no event is entered in the diagnostic buffer.

Example of Accessing Process Data

The example program illustrates the programming of simple access to the process images and/or directly to the process I/Os, in other words without the use of symbolic variables. The program is contained in the file p_access.c in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

6.4 Access to the Process I/Os of the Interface Submodules Using I/O Descriptors

I/O descriptors are used for fast access to signals of the process I/Os in the case of interface submodules.

Each time you issue a call ***M7LoadDirect()*** or ***M7StoreDirect()*** the driver first checks whether the process signal can be accessed or not and then converts the specified logical address to the corresponding geographical address. The conversion of logical address is associated with processing time and thus with an increase of the overall access time.

I/O descriptors can be used to bypass the conversion of the logical address and thus reduce the associated processor load. I/O descriptors store all the information in other words needed to access the process signals.

The savings in processor time are particularly relevant in the case of access to local process I/Os (interface submodules). In this case the overall access time can be lowered very significantly compared to direct access.

Setting up I/O Descriptors

Before you can reference the process signals of the interface submodules using I/O descriptors, I/O descriptors must be initialized for the required address range of the ISA bus process I/Os.

The following M7 API calls are available for initializing I/O descriptors:

M7InitISADesc(Addr,PType,Len,pIODesc)

This function calculates an I/O descriptor for the specified I/O area from the logical based address ***Addr*** and length ***Len***. The parameter ***PType*** is used to specify whether the I/O range to be addressed contains inputs (***PType=M7IO_IN***) or outputs (***PType=M/IO_OUT***).

The parameter ***pIODesc*** points to the I/O descriptor that was initialized by the call.

Accessing I/O Signals Using I/O Descriptors

After setting up the corresponding descriptor, macros are used to carry out fast access to the signals of the interface submodule.

The following macros are available for fast input and output of signals from/to the interface submodules:

- Read/write byte:

  M7LoadISAByte(pIODesc,pError) /* read byte status */

  M7StoreISAByte(pIODesc,Value) /* set byte to "Value" */

- Read/write word:

  M7LoadISAWord(pIODesc,pError) /* read word status */

  M7StoreISAWord(pIODesc,Value) /* set byte to "Value" */

- Read/write double word:

  M7LoadISADWord(pIODesc,pError) /* read double word status */

  M7StoreISADWord(pIODesc,Value); /* set double word to "Value" */

The macros ***M7LoadISA...(pIODesc,..)*** carry out direct access to the input signals in the I/O area specified by the I/O descriptor ***pIODesc***.

The return value of the call contains the value which was read. The parameter ***pError*** indicates whether the call was successful or not.

The macros ***M7StoreISA...(pIODesc,val)*** carry out direct access to the I/O modules with the help of the I/O descriptors. The data to be written is specified in the parameter ***val***.

The address of the I/O area is specified in the I/O descriptor ***pIODesc***. The process output image is updated automatically (see Figure on Page 6-10). The return value of the call contains an error code which indicates whether the call was successful or not.

All calls carry out automatic conversion of the data format from the Intel byte order to the SIMATIC byte order.

Note

I/O accesses to the **IF 961-AIO** interface submodule are only possible using I/O descriptors (macros *M7LoadISA*... and *M7StoreISA*...) if the submodule has been configured for cyclic scanning.

Example of Handling Process Data

As an extension to the previous example, this example program shows how to program access to the process I/Os with the help of I/O descriptors. The example is contained in the file *isa.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

6.5 P Bus Communication via User Data

This section describes how you can transfer small amounts of data (max. 16 bytes) between an M7 FM and a CPU module (S7 CPU or M7 CPU) via Pbus.

User data is the simplest method of data transfer between a CPU (S7 CPU or M7 CPU) and an underlying function module (FM).

With respect to user data, an FM behaves like a signal module from the viewpoint of the CPU. Accordingly, the CPU can access the user data area of the FM using the calls for I/O modules.

The user data area is subdivided into two parts, one for inputs and one for outputs. The input area can be read by the CPU and the output area can be written to. The maximum size of each of these areas is:

- 16 bytes for FM 356-4
- 128 bytes for FM 456-4

From the viewpoint of the CPU, the address of the input and output areas are both mapped to the logical starting address which is assigned to the FM's slot. From the viewpoint of the C user program, the input area and the output area for the user data are mapped to the **logical starting address** of the input I/O area and/or the output I/O area, respectively. With default addressing (only for M7-300) the starting address is 240. Otherwise the starting addresses can be assigned with STEP 7.

An M7 RMOS32 program can access the user data area using the same M7 API calls which are used to directly access the I/O area of a process I/O module.

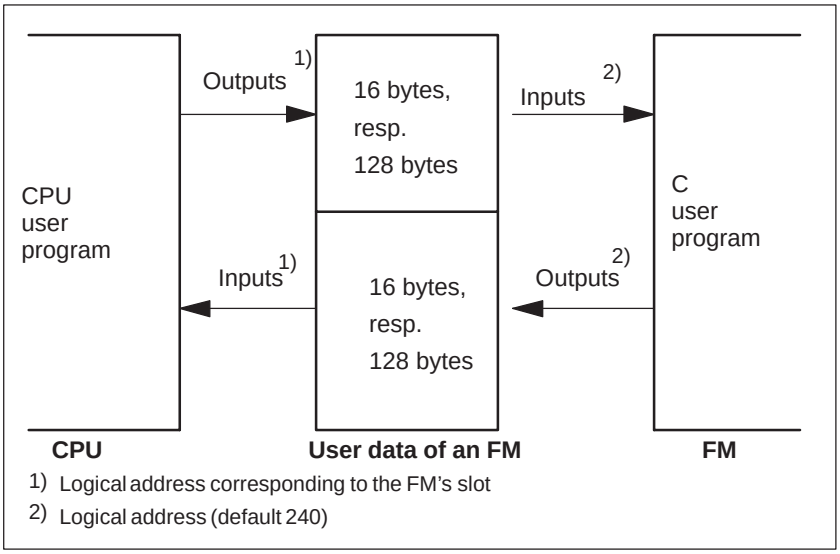


Figure 6-4 Access to the User Data of an FM

Programming the Exchange of User Data

Proceed as follows to use the user data area for data transfer:

The user data area is subdivided into two parts.

1. Design the structure of the FM user data with separate parts for input and output. Each part can be structured as required. **Please note that inputs from the FM's viewpoint are outputs for the CPU, and outputs from FM's viewpoint are inputs for the CPU.**
2. Program the required access to the user data area for the FM and for the CPU:
   - On the FM: To access the FM user data, your C program should use the M7 API functions for accessing process I/Os.

     You can use the function ***M7LoadDirect..*** to read the user data from the output area and the function ***M7StoreDirect..***. to write the user data to the input area. With the default addressing, the base address of the input area and the output area are both specified as the **logical starting address 240** in case of default addressing.
   - On an S7 CPU: You read and write the FM user data in your STEP 7 user program in the same way that you access the data of an analog signal module.

     The input area can be read on the S7 CPU with the load command (for example ***L PW..***) and the output area can be written to with the transfer command (for example ***T PW..***). In both cases, the logical starting address of the input and the output area is the address to which the FM's slot is assigned.
   - On an M7 CPU: You read and write the FM user data in your M7RMOS32 user program in the same way that you access the data of an analog signal module.

     The input area can be read on the M7 CPU with the load command (for example ***M7LoadDirect..***) and the output area can be written to with the transfer command (for example ***M7StoreDirect..***). In both cases, the logical starting address of the input and the output area is the address to which the FM's slot is assigned.

Alarms at Transferring User Data

Alarms can be initiated upon transferring user data from the CPU to the FM 356-4 or FM 456-4 with the ***M7StoreDirect..*** calls. Use the ***M7LinkIOAlarm*** call in order to process these alarms.

Example of Transferring User Data

An example program for data transfer between a CPU and an FM is contained in the files *nutz_fm.c* and *nutz_cpu.c* in the directory ..\M7SYSx.yy\EXAMPLES\M7API.

6.6 P Bus Communication via Data Records

Data Exchange Using M7 Data Records

Another way of communicating between a function module (FM) and a CPU (S7 CPU or M7 CPU) is to use data records.

With respect to data records, an FM behaves like a signal module from the viewpoint of the CPU. Accordingly, the CPU can send records to the FM and/or read them from the FM (see section 6.7).

The P bus protocol transfers the data records synchronously to the CPU user program, in other words all of the data in the record has been transferred when the function ***SFC 58/59*** (S7 CPU) or the call ***M7StoreRecord()/M7LoadRecord()*** or ***M7LoadRecordEx()*** (M7 CPU) has completed successfully.

Characteristics of M7 User Data Records

Each user data record is addressed using a so-called record number between 2 and 127. A particular record number can be assigned to two different records, one for reading and one for writing.

On the FM side, the records are stored in the memory area for records of the S7 object server. The M7 RMOS32 program can access the data records using the M7 calls for the S7 object server (see section 7.4).

Proceed as follows to use FM data records for data transfer:

1. Choose the record numbers that you want to use; M7 RMOS32 reserves the numbers 2 to 127 for user programs.
2. Define the structure of the data records. The records can be structured as required.
3. Specify the access to each of the records by the CPU (reading, writing).
4. Program the required access to the records, both on the FM side and on the CPU side.

Note

When reading and writing records, please carry out conversion between Intel and SIMATIC data format within your C user program as necessary (see section 5.22).

Diagnostic Data Records DS0 and DS1

The diagnostic data records DS0 and DS1 contain the diagnostic data of a module. In the memory area of the S7 object server on an FM 356/456-4 these data records are assigned the “read” attribute, since the CPU can only read them. On the FM the diagnostic data records can be read or written. The FM 456/356-4 can read the diagnostic data records of its subordinated modules, e.g. of another FM, and it can write the diagnostic data records which are to be read by the CPU.

Parameter Data Records DS0 and DS1

The parameter data records DS0 and DS1 contain the configuration data of a module.

- DS0 contains static parameters, that are assigned using STEP 7. These parameters are downloaded from the programming device to the CPU and the CPU transfers them to the modules.
- DS1 contains dynamic parameters, that can be assigned from the user program during execution.

In the memory area of the S7 object server on an FM 356/456-4 these data records are assigned the “write” attribute, since the CPU can only write them. On the FM the diagnostic data records can be read or written. The FM 356-4 can write the parameter data records of the modules located in its local bus segment. Generally an FM can read the parameter data records DS0 and DS1 received from the CPU.

Programming Access to the Records on the FM

Proceed as follows to program the access on the FM:

1. Create the required records as S7 objects on the S7 object server (see section 7.4). Observe the different type code for reading and writing.
2. In order to read from and write to the records, use the functions for accessing S7 objects (see section 7.4).
3. If necessary, you can register a task with the S7 object server to receive a message when the CPU accesses the record in a reading or writing mode.

Programming Access to Records on the CPU

Proceed as follows to program the access on the CPU:

On an S7 CPU: In your STEP 7 user program, use the system function ***SFC 58*** for writing the record and the system function ***SFC 59*** for reading the record.

The base address in the SFC call should be specified as the starting address assigned to the FM's slot.

On an M7 CPU: In your C user program, use the M7 API call ***M7StoreRecord()*** for writing the record and the calls ***M7LoadRecord()*** or ***M7LoadRecordEx()*** for reading the record.

Please use the starting address assigned to the slot of the FM for the base address parameter to be specified in the M7 API call.

6.7 Configuring I/O Modules

Initial configuration of the signal modules usually takes place during system startup. Some of the signal modules and the M7 interface submodules allow the initial configuration made with the SIMATIC Manager to be changed dynamically later, during runtime. The M7 API provides functions for transferring data records.

Calls for Data Transfer

I/O modules are configured with module-specific data records. The parameter records are identified by a record number. The maximum length of a parameter record is 240 bytes.

The following calls are used to transfer parameter records to the modules:

M7StoreRecord(RecordNum,pBuffer,..,PType,Addr)

M7LoadRecord(RecordNum,pBuffer,..,PType,Addr) or

M7LoadRecordEx(RecordNum,pBuffer,..,PType,Addr) or

The function ***M7StoreRecord()*** transfers a data record to an I/O module. The record number is specified in ***RecordNum***. The parameter ***PType*** specifies whether the module is an input module or an output module.

The parameter ***Addr*** specifies the logical starting address of the signal module within the S7 system.

The module type ***PType*** must also be specified since an input module and an output module can lie on the same logical address (but not with the default addressing!).

The functions ***M7LoadRecord()*** and ***M7LoadRecordEx()*** transfer a data record from an I/O module into a buffer in working memory. the call ***M7LoadRecordEx()***additionally returns the number of bytes transferred.

SIMATIC/Intel Conversion

The parameter records of the signal modules are stored in **SIMATIC** data format (Motorola byte order). For data records of more than 1 byte in length, the structure of the data differs from that of the **Intel** data format (in other words Intel byte order) used for data in your C user program. The M7 API provides C macros to convert the data format from the Intel byte order to the SIMATIC byte order or vice versa (the data formats and the conversion are described in 5.22).

Procedure to Follow when Configuring

We recommend the following procedure to configure an I/O module:

1. Get the following information from the data sheet for the signal module:
   - Which record numbers are already assigned to the signal module?
   - What is the structure of each data record?

   For information on the M7 interface submodule parameters please see Chapter 3 of the Reference Manual.
2. Decide which record numbers you require for configuration.
3. Define a C structure for the configuration data in your C program which matches the record structure. Assemble a data record in the buffer in Intel data format.
4. Define an identical C structure for the configuration data in SIMATIC data format.
5. Use the conversion macro to convert each individual component of the Intel C structure into the SIMATIC C structure.
6. Transfer the data record in SIMATIC data format to the signal module using the following call:

   M7StoreRecord(RecordNum,pBuffer,...)

Working with S7 Objects

7

Chapter Overview

7.1 General Information on S7 Objects

S7 objects are data areas whose structure follows the SIMATIC data conventions. S7 objects largely correspond to the operand area of an S7 CPU, in other words the data types which can be accessed with the command set of an S7 processor (process I/O module, process image etc.).

Just as with an S7 CPU, S7 objects on the M7 can be addressed directly using the system's built-in communication services - in other words with the hardware and software tools from the SIMATIC S7 system such as the operator interface - without needing any explicit code in the C user program.

Which S7 Objects are Available?

The M7 API provides the S7 objects listed in Table 7-1. The type ID shown in the table is needed by the M7 API call to access each S7 object type.

Table 7-1 S7 Objects Available on the M7

S7 Object	Type ID	Created (by)
Process input image	M7D_PII	automatically
Process output image	M7D_PIQ	automatically
Peripheral input area	M7D_IO	automatically
Peripheral output area	M7D_IO	automatically
Flag area	M7D_M	C user program
Data block	M7D_DB	C user program
Data record, read	M7D_PAR_READ	C user program
Data record, write	M7D_PAR_WRITE	C user program

Note

The “read” and “write” attributes for data records are considered on an FM from the CPU’s point of view.

Usually an FM reads the data records (type identifier M7D_PAR_WRITE) - for example parameter data records - which were written by the CPU with the call ***M7Readxx***. On the other hand an FM writes data records (type identifier M7D_PAR_READ) - for example diagnostic data records - that are to be read by the CPU with the call ***M7Writexx*** (see also chapters 6.6 and 7.4).

S7 Object Server

Local management of S7 objects and access to S7 objects is the responsibility of the S7 object server.

The main function of the object server is the management of S7 objects, in other words the coordination of access between programs and external communication partners. Furthermore, the S7 object server provides the following functions through the M7 API:

- Creating and deleting S7 objects
- Copying data (reading/writing)
- Detecting and coordinating access
- Redirecting and filtering requests

Note

In contrast to an S7 system, the M7 system has no FB (function block) and no OB (organization module).

Communication Services for S7 Objects

S7 objects located on remote modules can be accessed using the following functions:

- M7 API calls for the operator interface (OI), see section 7.7.
- M7 API calls for the object management system(OMS), which allow you to access the S7 objects of a remote programmable controller the way a programming device does.

The M7 API calls for the OI and the OMS require that the connections to the communication partners have been configured using the SIMATIC Manager (see also Sections 8.5 and 8.6).

7.2 The Memory Model of the S7 Object Server

The functional behavior of an S7 CPU allows the user to activate and deactivate S7 objects such as data blocks during program execution, in order for example to make them available for usage by an S7 program or to prevent their usage.

The runtime environment of an S7 CPU allows for example that an S7 object is made accessible to an S7 user program during program execution, either through an external communication interface or from mass storage, allowing the user program to process the object. Similarly, access to the S7 object can be removed again during execution of an S7 user program and the object returned to mass storage.

In a similar way to the S7 CPU, an M7 automation computer also allows dynamic handling of S7 objects.

Storage of S7 Objects

Analogously to an S7 CPU, in order to implement this dynamic object transfer the M7 automation computer has access to a special memory model for storing S7 objects. The following storage areas are available in this memory model (see Figure 7-1):

- Working memory
- Static RAM
- Temporary load memory
- Backup memory
- Permanent load memory
- Read-only memory

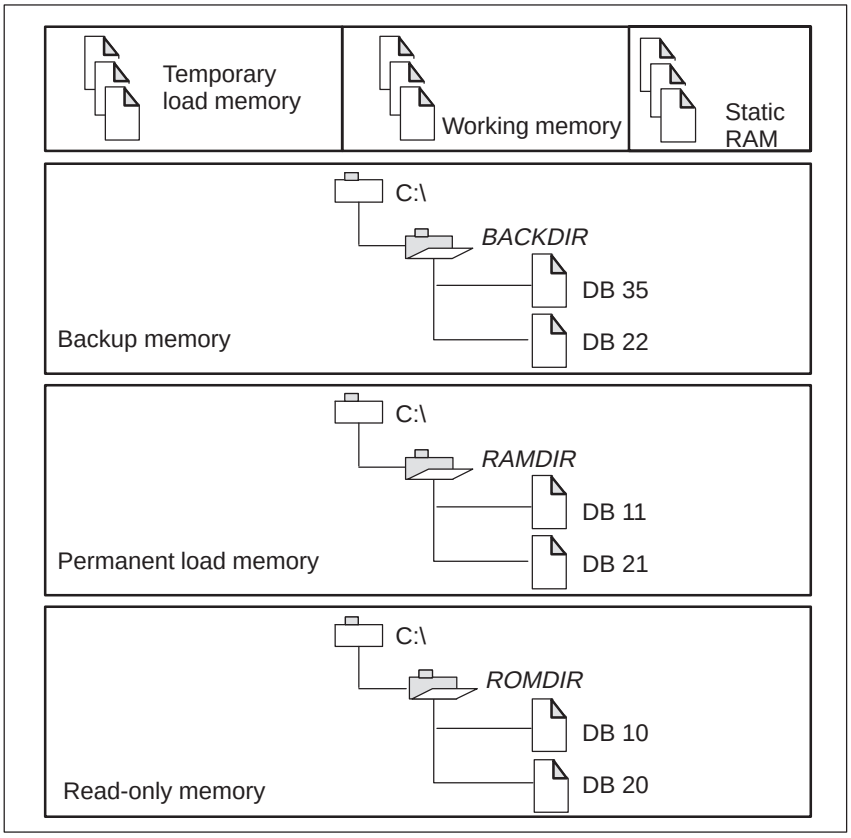


Figure 7-1 Storage of S7 Objects

The working memory and the temporary load memory are both located in the main memory RAM of the automation computer and the backup memory, the permanent loading memory and the read-only memory are implemented by directories in the M7 file system.

Working Memory

All processing of S7 objects takes place in the working memory. Only those S7 objects which are currently stored in working memory can be accessed by the user program for reading and writing through the associated M7 API calls.

S7 objects that you create with the M7 API call ***M7CreateObject()*** are automatically placed in the working memory.

Static RAM

In addition to the main memory, the M7 automation computer also has static RAM (SRAM) to store flag areas and data blocks. System configuration with the SIMATIC manager is used to specify which flag areas and/or which data blocks are kept in the static RAM area.

This allows you to store certain important process data in buffered memory to enable the M7 to resume processing after powering up the system at the same place where the system was halted on being switched off (or as a result of sudden power loss).

Temporary Load Memory

The temporary load memory is used for temporary storage of S7 objects which you want to load into main memory via an external communication interface in order to activate (link) them using the object management system (OMS).

However, before an object can be activated (linked) with the object management system, the object must first be completely read into the main memory in other words into the temporary loading memory.

Programs do not have direct access to S7 objects in the temporary loading memory.

Backup Memory

The backup memory is used for long term storage of M7 objects (independently of the M7 power supply) which have been locally created on the M7 with the call ***M7CreateObject()***.

The call ***M7StoreObject()*** can then be used to copy the locally created S7 objects from the working memory to the backup memory. This call can be used to synchronize locally created S7 objects in working memory with the associated S7 objects in the backup memory.

The backup memory is implemented as a directory in the M7 file system. The directory is specified with the environment variable *BACKDIR*.

Permanent Load Memory

The permanent loading memory is used for long term storage of S7 objects (independently of the M7 power supply) which have been transferred to the M7 from an external source via communication services.

Whenever S7 objects are loaded into working memory via communication services and activated with the object management system, a copy of the S7 object is automatically stored in the permanent load memory.

In this way, the object management system automatically synchronizes S7 objects which are loaded via communication services and activated in the M7 with the associated S7 objects in the permanent load memory.

The permanent load memory is implemented as a directory in the M7 file system. The directory is specified with the environment variable *RAMDIR*.

Read-only Memory

The read-only memory is used to store those S7 objects which are needed to recreate the original status of the automation computer. The original status is reached with the RESET function.

S7 objects which have been locally created on the M7 can be copied from the working memory to the read-only memory with the call ***M7StoreObject()***.

All active (in other words linked) S7 objects which were not created locally are automatically copied from the working memory to the read-only memory when the memory mode is switched from RAM to EPROM. S7 objects created locally with ***M7CreateObject()*** are not copied into the read-only memory.

This allows the stored original status of the M7 to be dynamically modified, in other words to define a new configuration for the RESET function.

The read-only memory is implemented as a directory in the M7 file system. The directory is specified with the environment variable *ROMDIR*.

Note

Number of data blocks:

If too many data blocks are copied and linked in the permanent load memory area (RAMDIR), in the backup memory area (BACKDIR) or in the read-only memory area (ROMDIR) of the object server, resource problems may result.

We recommend that you store a maximum of 128 objects (data blocks) in the above memory areas.

Memory Reset

The memory reset function is used to recreate the original status of the M7 automation computer (see section 5.12). First of all, all S7 objects in working memory (buffered and non-buffered), temporary and permanent load memory are deleted.

Following this, all S7 objects in the read-only memory are copied into the working memory and activated. The original status is thus recreated, at least from the viewpoint of the S7 objects.

Power On / Hardware Reset

During power on and/or hardware reset of the M7, first of all the S7 objects in working memory which are not backed up are deleted. Following this, the S7 objects in read-only memory, the S7 objects in the permanent load memory and the S7 objects in the backup storage are copied into the working memory in this order and activated.

S7 objects with the same name are overwritten by subsequent copying actions, in other words S7 objects in the read-only memory are overwritten by objects of the same name in the permanent load memory, and S7 objects in the permanent load memory are overwritten by objects with the same name in the backup memory.

Accordingly, assuming correct shut down of the system, the power on and/or hardware reset function allows the M7 automation computer to be brought back to the same status that it had before shut down.

7.3 Preparation for the Use of S7 Objects

Planning the Required S7 Objects

Before you want to access S7 objects in your C user program, you should decide which type of S7 objects you need and their required purpose. Generally speaking, the objects that are needed can be subdivided into two categories:

- S7 objects with dynamic data:

  These are used for all S7 data that you want to process with your user program and/or objects which are necessary for external access by operator interface functions.

- Data blocks with static data

  These objects are used for static S7 data that you want to process with your C user program (for example static parameters, constants) which are stored in the M7 file system.

Specifying BACKDIR, RAMDIR and ROMDIR

These three environment variables are used to specify to the runtime environment on which logical devices and under which directories you want to store the backup memory, permanent load memory and read-only memory respectively.

The following SET commands are included in the file \ETC\INITTAB in order to specify the required directories:

SET BACKDIR=<path_back> for the *BACKDIR* directory,

SET RAMDIR=<path_ram> for the *RAMDIR* directory,

SET ROMDIR=<path_rom> for the *ROMDIR* directory.

The path names <path_back>, <path_ram> and <path_rom> must specify the complete path including the drive letter.

Creating Data Blocks

Data blocks with static data can be created with a self-contained utility program as follows (the M7 API calls are discussed in 7.4):

1. Create the required data block in your program with the function ***M7CreateObject()***. The data block is stored by the S7 object server in the working memory.
2. Initialize the contents of the data block with the memory function ***M7Write***.. for bits, bytes, words or double words.
3. Copy the data block into the back memory*(BACKDIR)* or the read-only memory *(ROMDIR)* with the call ***M7StoreObject()***.
4. Enter the name of the utility program in the CLISTART.BAT file. Make sure that the directories *BACKDIR* and/or *ROMDIR* have been created before the utility is executed.
5. Load the utility into the M7 with the SIMATIC Manager and start it using the file CLISTART.BAT.

7.4 Creating and Working with S7 Objects

This section explains how to create and work with S7 objects in your C user program.

Creating an S7 Object

With the exception of the objects E, A, PE and PA, each S7 object with dynamic data must be created in your C program. The M7 API provides the following call for creating objects:

M7CreateObject(ObjType,Part,count,Ptr);

The call must be specified with the required object type ***ObjType*** (see Table 7-1), the segment or part number ***Part*** (for example DB number) and the number of elements ***count*** that you want to create.

The parameter ***Ptr*** is used to specify whether the object to be recreated should be stored in the program's own memory area or the memory area of the S7 object server.

If you create an object in the program's own memory area, you can manipulate it without the help of M7 API calls. If it is created in the memory area of the S7 object server, you can only read the data and/or overwrite the data using the associated M7 calls.

If you want to create the object in the program's own memory area, ***Ptr*** specifies the address of the previously assigned memory area that you want to use. If you don't want to get involved with memory management, specify Ptr as a null pointer. In this case, the S7 object server carries out the memory management for the object and chooses the starting address.

Note

The M7 API function ***M7CreateObject()*** is used to create an S7 object in working memory. If you want to store the data from this object permanently, you must then explicitly copy the object with the M7 API call ***M7StoreObject()*** from the working memory to *BACKDIR* or *ROMDIR*.

The S7 objects M7D_I, M7D_Q and M7D_IO (inputs and outputs) can be accessed both with calls to access process I/Os (see section 6.3) and with functions of the S7 object server.

Getting Information on S7 Objects

The following function is used to get all available information on the data structure of an S7 object:

M7GetObjectInfo(ObjType,Part,pObjInfo)

This call must be specified by the object type ***ObjType***, the part number ***Part*** (for example DB number) and the address ***pObjInfo*** of a data structure of type M7OBJ_INFO. This data structure must be previously assigned in your data area.

If the call is successful, the memory area specified by ***pObjInfo*** contains the following information:

- Size: Object size in bytes
- Data: Pointer to the object data
- External: TRUE if the object lies outside the object server, FALSE if the object lies within the object server.

Moving S7 Objects to User Memory

S7 objects can also be moved within memory. This is done with the following call:

M7LocateObject(ObjType,Part,..,Ptr,Copy)

This call moves the S7 object specified by the object type ***ObjTyp*** and the part number ***Part*** into a user-defined memory area. The call must also specify the starting address ***Ptr*** of the new memory area to which the object should be moved. The ***Copy*** parameter is used to specify whether the object's user data should be also transferred to the new area or not.

Returning S7 Objects to the Object Server

S7 objects that have been moved with the ***M7LocateObject*** call can be returned to the S7 object server. This is done with the following call:

M7RelocateObject(ObjType,Part,Copy)

This call returns the S7 object specified by the object type ***ObjTyp*** and the part number ***Part*** to the S7 object server. The ***Copy*** parameter is used to specify whether the object's user data should be also returned to the object server or not.

Deleting S7 Objects

The following call is used to delete an S7 object:

M7DeleteObject(ObjType,Part)

This call deletes the S7 object specified by the object type ***ObjType*** and the part number ***Part*** from working memory **and** from the *BACKDIR* directory.

The following function is used to delete an existing S7 object from *BACKDIR* **or** *ROMDIR* (but not from the working directory):

M7RemoveObject(ObjType,Part,Rom)

The parameter ***Rom*** is used to specify whether the S7 object should be deleted from *BACKDIR (Rom=FALSE)* or from *ROMDIR (Rom=TRUE)*.

Note

Objects located in the SRAM area can be deleted only in the STOP state.

Saving in *BACKDIR/ROMDIR*

In addition to storing data blocks and data records in working memory, you can also copy them to the backup memory (*BACKDI*R) and/or the read-only memory (*ROMDIR*). This is done with the following call:

M7StoreObject(ObjType,Part,Rom)

This call copies the object specified by object type ***ObjTyp***e and part number ***Part*** from working memory to the *BACKDIR* directory (Rom=FALSE) or *ROMDIR* (Rom=TRUE). The call returns with an error if the S7 object server cannot find the associated directory.

Accessing S7 Objects

The following M7 API calls to the S7 object server are provided for accessing data in S7 objects:

- Request/write bit:

  M7ReadBit ***/* request bit status */***

  M7WriteBit(..,Value) ***/* set bit to "Value" */***

- Read/write byte:

  M7ReadByte ***/* read byte status */***

  M7WriteByte(..,Value) ***/* set byte to "Value" */***

- Read/write word:

  M7ReadWord ***/* read word status */***

  M7WriteWord(..,Value) ***/* set word to "Value" */***

- Read/write double word:

  M7ReadDWord; ***/* read double word status */***

  M7WriteDWord(..,Value); ***/* set double word to "Value" */***

- Read/write floating point number:

  M7ReadReal; ***/* read floating point number */***

  M7WriteReal(..,Value); ***/* set floating point to "Value" */***

- Read/write data area:

  M7Read ***/* read data area */***

  M7Write ***/* write data area */***

Intel/SIMATIC Conversion

All S7 data is stored in *SIMATIC* data format (Motorola byte order). The structure of the data differs from that of the *Intel* data format (in other words Intel byte order) used for data in your C user program.

When reading and writing data in **word** and **double word** format, the corresponding M7 API functions convert the data automatically between the two formats. When reading and writing bits and bytes there is no need for conversion, since they are identical in both data formats.

When reading and writing **data areas**, you need to carry out the necessary conversion explicitly using appropriate macros. This is described in section 5.22.

7.5 Registering for Notification Messages when S7 Objects are Accessed

In some cases it may be necessary for a task to be notified if another task or external communication partner accesses S7 objects within the S7 object server.

For example, a task may need to be notified if an S7 module sends a data block to the M7 with new runtime parameters.

In this case, the task which needs to be notified can register itself for a particular S7 object with the S7 object server. In this case it will automatically receive a message in its message queue from the S7 object server after an access to the registered S7 object has occurred.

Registering for Notification

The following call is used to register an FRB to to be notified by the S7 object server when an F7 object is accessed:

M7LinkDataAccess(pOBJFRB,ObjType,Part,Flags,Mprio)

The call registers the FRB with the address ***pOBJFRB*** to receive notification on access to the S7 object specified by ***ObjType*** and ***Part***.

The parameter ***Flags*** is used to specify **which type of access** should result in notification by the S7 object server (writing, reading, creating, deleting, linking). Only one access type can be specified.

A list of the various possibilities for the ***Flags*** parameter can be found in the Reference Manual.

The call stores the parameters ***ObjType***, ***Part*** and ***Flags*** in the specified FRB. This information can be accessed and evaluated later - messages which are received from the S7 object server also specify the associated FRB.

The parameter ***Mprio*** specifies the required priority of the message from the S7 object server.

Section 5.3 contains a detailed description of the functional sequence when registering to receive messages of all types.

Evaluating the Message if the Object is Accessed

After access takes place to the S7 object which was specified when registering the FRB, the S7 object server sends a message with the appropriate identification.

The message identification (for example M7MSG_DATA_ACCESS_R: reading access, M7MSG_DATA_ACCESS_W: writing access etc.) specifies which type of access took place to the registered S7 object.

The following macros make it easier for the user program to evaluate the parameters of the FRB which is referenced in the message. These parameters were specified when registering the FRB.

- ***M7GetObjType(pOBJFRB)***

This macro returns the object type ***ObjType*** of the S7 object which was registered.

- ***M7GetPart(pOBJFRB)***

This macro returns the part number ***Part*** of the S7 object which was registered.

- ***M7GetFlags(pOBJFRB)***

This macro returns the access type ***Flags*** which was specified when registering the S7 object.

Deregistering the Notification of Access to S7 Objects

If a task no longer requires notification when an S7 object is accessed, it can deregister the associated FRB with the S7 object server with the following call:

M7UnLinkDataAccess(pOBJFRB)

The call must be specified with the address of the FRB to be deregistered.

Note

Before a task terminates, it must deregister all FRBs registered with the S7 object server.

Example of Handling S7 Objects

The example program in the file *objacces.c* in the directory ..\M7SYx.yy\EXAMPLES\M7API first of all creates an S7 object (DB 200) in the start up section and registers itself with the M7 object server to be notified if write access takes place to the DB 200 object.

In the main program loop, the task waits for messages from the S7 object server and branches correspondingly.

When the terminate task message is received, the registration with the S7 object server is cancelled and the task is terminated.

7.6 Registering Callback Functions for S7 Object Access

In an analogous way as registering itself for a notification a message, a task can also register a callback function with the S7 object server. The callback function allows tasks to directly control access to S7 objects. Callback functions are often used for conversion between SIMATIC and Intel data formats.

If a request to access the registered S7 object is received, the registered callback function is executed **before** the object is processed by the S7 object server. The callback function is given a pointer to the FRB which was specified in the registration (see section 7.5) allowing it to get information about the type of access.

The return value of the callback function can be used to specify whether the final processing of the access request by the S7 object server should be allowed or not. This allows a task to have complete control over access to S7 objects and can also allow the task to reserve the sole rights to process the access request itself.

Note

A callback function must not disturb the execution of the user program, so it must have a very short execution time. Callback functions must contain no interrupt handling, no blocking function calls and no RMOS API calls.

Registering a Callback Function for Object Access

The following M7 API call is used to register a callback function for object access:

M7LinkDataAccessCB(pCBFRB,pCallBack,ObjType,Part,Flags)

The call registers an FRB with address ***pOBJFRB*** for access to the S7 object specified by ***ObjType*** and ***Part***. You must also specify a pointer ***pCallBack*** to the callback function to be executed.

The parameter ***Flags*** is used to specify **which type of access** should result in notification by the S7 object server (writing, reading, creating, deleting, linking). Only one access type can be specified.

A list of the various possibilities for the ***Flags*** parameter can be found in the Reference Manual.

Notes on the Callback Function

The specified callback function must be declared as follows:

UDWORD CallBack(M7CBFRB_PTR pCBFRB)

When the callback function is called, the M7 API provides a pointer ***pCBFRB*** to the associated FRB in which the registration information (type identifier, access type etc.) is stored.

The FRB also contains the address of a buffer which contains the data to be written and/or to receive the data to be read from the S7 object. This information allows the callback function to carry out the data transfer itself, in other words without the help of the S7 object server.

The return value of the callback function can be used to specify whether the final processing of the access request by the S7 object server should be allowed or not.

Macros for Evaluating the FRB

The following macros make it easier for the callback function to evaluate the parameters of the FRB which is referenced in the call:

- ***M7GetCBObjType(pCBFRB)***

This macro returns the object type ***ObjType*** of the S7 object which was registered.

- ***M7GetCBPart(pCBFRB)***

This macro returns the part number ***Part*** of the S7 object which was registered.

- ***M7GetCBFlags(pCBFRB)***

This macro returns the access type ***Request*** which was specified when registering the S7 object.

- ***M7GetCBBuffer(pCBFRB)***

This macro returns a pointer to a buffer. For writing access, the buffer contains the data to be written to the S7 object. For reading access, it is used to receive the data to be read from the S7 object.

- ***M7GetCBDataType(pCBFRB)***

This macro is used to specify the data type (M7_DTxxx) which should be used to access the S7 object.

- ***M7GetCBByteOffset(pCBFRB)***

This macro specifies the byte base address of the first element in the S7 object to be operated on (read, write, delete etc.).

- ***M7GetCBBitOffset(pCBFRB)***

This macro specifies the bit base address of the first element in the S7 object to be operated on (read, write, delete etc.).

- ***M7GetCBCount(pCBFRB)***

This specifies the number of elements to be operated on (read, write, delete etc.) in the S7 object.

Deregistering the Callback Function

When the registration of the callback function for access to the S7 object is no longer required, the callback function can be deregistered with the following call:

M7UnLinkDataAccessCB(pCBFRB)

You must transfer the address of the FRB to be deregistered to the call.

Note

Before a task terminates, it must deregister all FRBs registered with the S7 object server.

Example

A sample program for monitoring accesses to S7 objects using a callback function can be found under M7SYS2.00\EXAMPLES\M7API\CALLBACK.C.

7.7 Calls for the Operator Interface

Operator interface systems allow process steps and machines to be observed and controlled. The integration of the operator interface mechanism in the operating systems of the S7 CPU and the M7 programmable controller allows the interconnection of operator interface systems (clients) to be implemented using system services (servers).

In addition, the M7 API calls for the operator interface allow you to implement your own operator interface programs (clients) on the M7 programmable controller.

These calls require that the connections to the communication partners have been configured using the SIMATIC Manager (see also Sections 8.5 and 8.6).

Functional Sequence: Reading/Writing Once Only

The M7 API calls for the operator interface (OI) provide functions to read data from and write data to a remote communication partner once only using a configured connection.

As with the unidirectional communications functions for configured connections, here too the functionality required in the remote communication partner is implemented internally in the operating system of the S7 CPU or M7 CPU and does not require explicit user programming.

In contrast to the unidirectional functions for configured connections ***M7PBKPut()*** and the ***M7PBKGet()***, the corresponding OI functions ***M7BUBWrite()*** and ***M7BUBRead()*** are executed synchronously. Transfer of the data is finished after the function returns to the calling task (parent), in other words if the call is successful, valid data has already been copied at this point to the destination programmable controller.

Figure 7-2 shows the functional sequence of the (optionally repeated) “once only” reading from/writing to a remote communication partner.

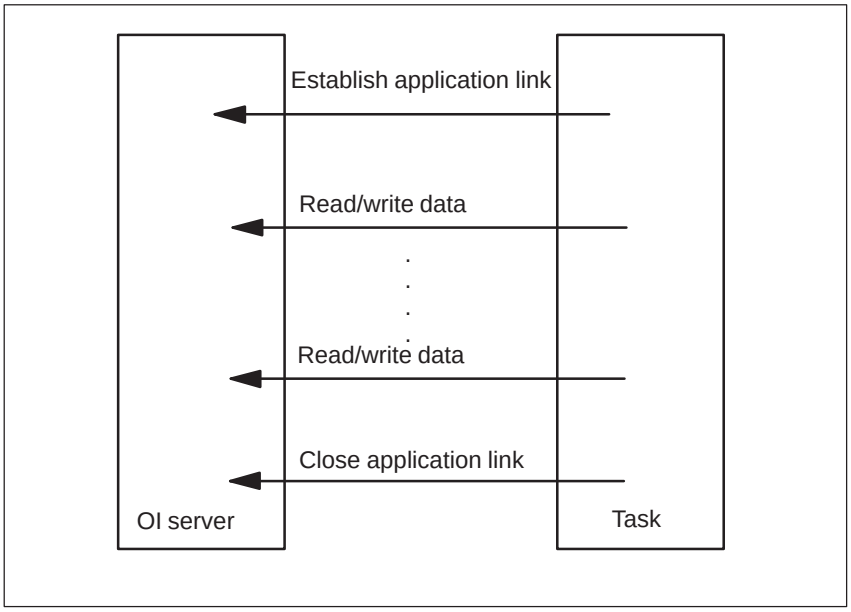


Figure 7-2 Functional Sequence of the “Once Only” Reading/Writing of Data

Functional Sequence: Cyclic Reading

In addition to the calls for once only reading and writing, the M7 API also offers calls for cyclic reading of data from a remote communication partner (OI server).

After establishing the application link to the remote communication partner, the M7 program must register an OI request for cyclic reading of data. The request specifies the variables required and also the duration of the cycle.

The cyclic reading function is started either directly on registering the request, or

with an extra start command.

From this point in time, the OI server sends data cyclically to the communication driver of the receiving programmable controller. Each time a data record is received, the K bus driver of the receiving programmable controller sends a message to the receiver task.

The receiver task must then copy the received data from the address area of the communication driver to a buffer within its own address area.

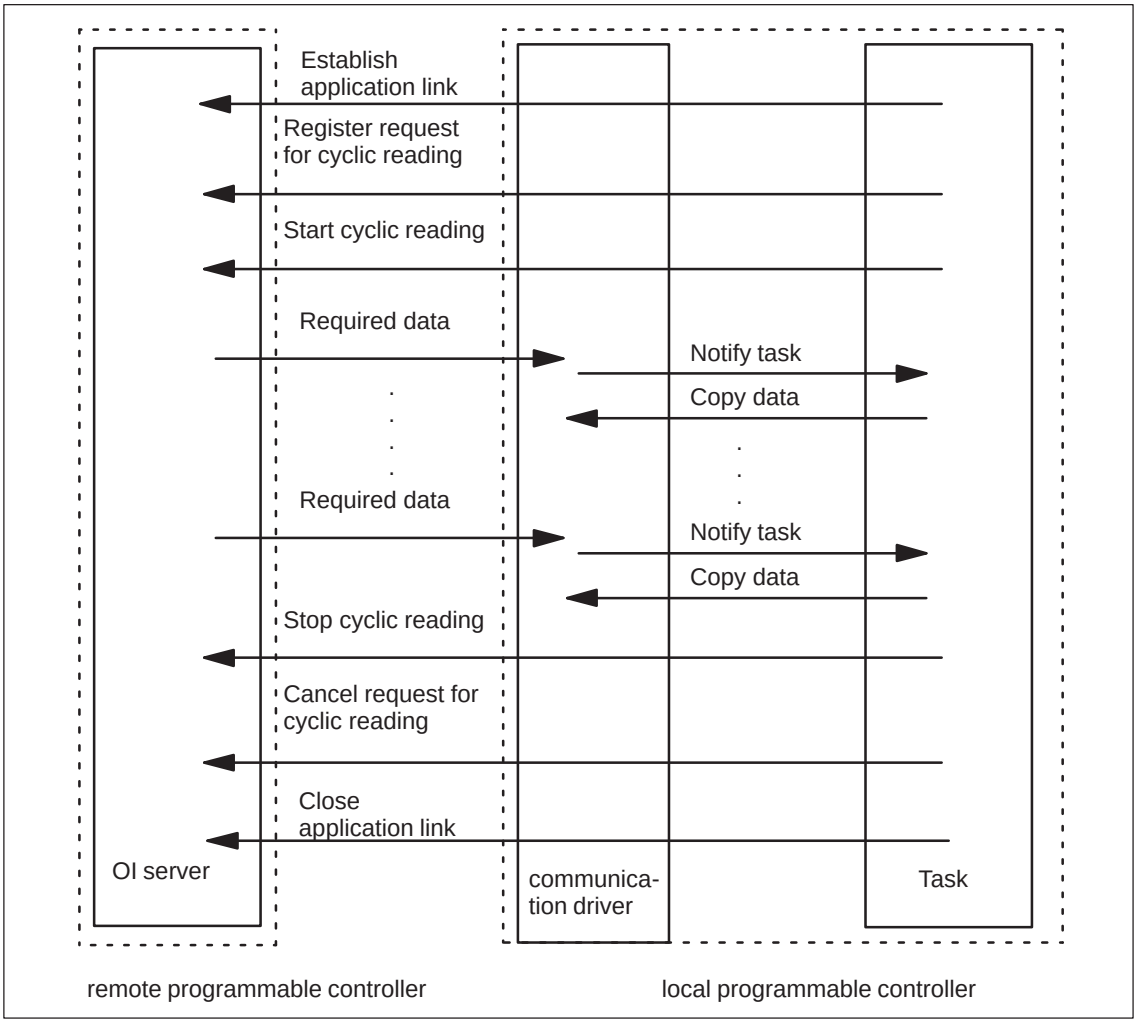


Figure 7-3 Functional Sequence of the “Cyclic” Reading/Writing of Data

7.8 Programming Once Only Reading and Writing

Reading Variables from the Communication Partner Once Only

Proceed as follows if you want your C program to read data from the S7 data area of a communication partner using operator interface (OI) functions:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Specify the number ***nVars***, data type and address of the variables to be read from the communication partner.

   Create an array ***AddrBuffer[nVars]*** of size ***nVars*** for the variables to be read. The array should have elements of type ***M7VARADDR***.

   The type ***M7VARADDR*** is defined in M7API.H and specifies a consecutive number of items within an S7 object. Initialize the elements of the array with the addresses of the variables to be read.

   On the receiver side, create a field ***DataBuffer[nVars]*** with ***nVars*** elements of type ***M7VARDATA***. The type ***M7VARDATA*** is defined in M7API.H and specifies a single buffer (address, size, etc.) to store a consecutive number of items.

   Create in the global data or on the heap ***nVars*** data buffers (for example DataBuffer[nVars] if all the variables are the same size) to receive the ***nVars*** variables.

   Initialize the elements of the array ***DataBuffer[nVars]*** with the addresses of the data buffers.
4. Read the variables from the remote station with the following M7 API call:

   M7BUBRead(ConnID,nVars,pAddrBuffer,pDataBuffer,pnBytes)

   You must specify a valid application link ID ***ConnID***, the number ***nVars*** of variables to be transferred, the address ***pAddrBuffer*** of the array with the addresses of the variables and the address ***pDataBuffer*** of the array with the address of the receive buffer.

   The return value indicates whether the call was successful or not. The call returns the number of actually copied bytes in ***pnBytes***.
5. Repeat the step 3 (loop in your C program).
6. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Note

The maximum amount of data which can be transferred with an OI request is limited by the maximum PDU size (protocol data unit) of the application link.

Writing Variables to the Communication Partner Once Only

Proceed as follows if you want your C program to send data to the S7 data area of a communication partner using operator interface (OI) functions:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Specify the number ***nVars***, data type and address of the variables to be written to the communication partner.

   Create an array ***AddrBuffer***[***nVars***] of size ***nVars*** for the variables to be written. The array should have elements of type ***M7VARADDR***.

   The type ***M7VARADDR*** is defined in M7API.H and specifies a consecutive number of items within an S7 object. Initialize the elements of the array with the addresses of the variables to be written.

   Create a field ***DataBuffer[nVars]*** with ***nVars*** elements of type ***M7VARDATA***. The type ***M7VARDATA*** is defined in M7API.H and specifies a single buffer (address, size, etc.) to store a consecutive number of items.

   Initialize the elements of the array ***DataBuffer[nVars]*** with the addresses of the variables whose contents you want to send. The variables to be sent must be stored in the global data or on the heap.
4. Send the variables to the remote station with the following M7 API call:

   M7BUBWrite(ConnID,nVars,pAddrBuffer,pDataBuffer)

   You must specify a valid application link ID ***ConnID***, the number ***nVars*** of variables to be transferred and the address ***pAddrBuffer*** of the array with the addresses of the destination variables and the address ***pDataBuffer*** of the field with the addresses of the variables to send.

   The return value indicates whether the call was successful or not.

5. Repeat the step 3 (loop in your C program).
6. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

7.9 Programming Cyclic Reading

Registering a Request for Cyclic Reading

If you want your C program to read data cyclically from a communication partner, you must first register a corresponding operator interface (OI) request for cyclic reading as follows:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Create an FRB ***CommFRB*** of type **COMMFRB** in your data area (global data or heap) for the communication feedback. Tag the FRB and thus your communication request with the function:

   M7SetFRBTag(&CommFRB, Tag)
4. Specify the number ***nVars***, data type and address of the variables to be read cyclically from the communication partner.

   Create an array ***AddrBuffer[nVars]*** of size ***nVars*** for the variables to be read. The array should have elements of type ***M7VARADDR***.

   The type ***M7VARADDR*** is defined in M7API.H and specifies a consecutive number of items within an S7 object. Initialize the elements of the array with the addresses of the variables to be read.

   Create a field ***DataBuffer[nVars]*** with ***nVar***s elements of type ***M7VARDATA***. The type ***M7VARDATA*** is defined in M7API.H and specifies a single buffer (address, size, etc.) to store a consecutive number of items.

   Create in the global data or on the heap ***nVars*** data buffers (for example DataBuffer[nVars] if all the variables are the same size) to receive the ***nVars*** variables.

   Initialize the elements of the array ***DataBuffer[nVars]*** with the addresses of the data buffers.

5. Read the variables from the remote station with the following M7 API call:

   M7BUBCycRead(flags,ConnID,pCommFRB,nVars,pAddrBuffer, pDataBuffer,CycTime,pnRequest, MPrio)

   You must specify a valid application link ID ***ConnID***, the address ***pCommFRB*** of the FRB you have created, the number ***nVars*** of variables to be transferred, the address ***pAddrBuffer*** of the array with the addresses of the variables, the address ***pDataBuffer*** of the array with the address of the receive buffer and the cycle time ***CycTime***.

   The parameter ***flags*** specifies whether the request should be started automatically after registration (A_IMMEDIATE) or not.

   The request returns in ***pnRequest*** the internally assigned request number. ***Mprio*** specifies the required priority of the returned message.

Starting a Request for Cyclic Reading

After you have registered a request for cyclic reading, you can start it with the following call:

M7BUBCycReadStart(ConnID,nRequest)

You must specify a valid application link ID ***ConnID*** and the request number ***nRequest*** which was assigned when creating the request.

Receiving Cyclic Data

After your task has registered a request for cyclic reading of data, the communication partner sends data repeatedly at the specified interval (cycle time) to the communication driver of the local programmable controller.

Proceed as follows in order to copy this data, which is received asynchronously of the user program, from the address area of the communication driver to the receive buffer which you created in your program:

1. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

   The communication driver generates a message of type M7MSG_BUB_NDR for each received data packet. The parameter ***pMessageParam*** refers to the FRB which was specified when registering the request.

2. Determine the job number with the following macro:

   M7GetCommRequest((M7COMMFRB_PTR)pMessageParam)

3. Use the following synchronous call to copy the received data from the address area of the communication driver to the receive buffer in your user program:

   M7KEvent(ConnID,nRequest,pBuffer,nBufsiz,pnBytes)

   You must specify a valid application link ID ***ConnID***, the request number ***nRequest***, the address ***pBuffer*** and the size ***nBufsiz*** of the array referred to with ***pBuffer*** with the specified receive buffer information.

   The call then returns the number of actually copied bytes in ***pnBytes***.

   If a message cannot be found for the specified application linkID ***ConnID*** and request number ***nRequest***, the call returns without error with ****pnBytes***=0.

4. Repeat the steps 1 to 3 (loop in your C program).

Stopping a Request for Cyclic Reading

A currently running request for cyclic reading can be stopped with the following call:

M7BUBCycReadStop(ConnID,nRequest)

You must specify the application linkID ***ConnID*** and the request number ***nRequest*** which was returned when registering the request.

This call only stops the running request, in other words the cyclic transfer of data is halted. The request still remains stored in the request queue of the K bus driver and can be started again with another call of ***M7BUBDycReadStart()***.

Deleting a Request for Cyclic Reading

A request for cyclic reading can be deleted from the request queue of the K bus driver with the following call:

M7BUBCycReadDelete(ConnID,nRequest)

You must specify the application link ID ***ConnID*** and the request number ***nRequest*** which was returned when registering the request.

Example of Cyclic Reading

The example program in the file *mpicyc.c* in the directory ..\M7SYx.yy\EXAMPLES\M7API illustrates the procedure to follow for cyclic reading of data.

7.10 General Information on the Object Management System

The M7 API calls to the OMS (Object Management System) allow a program on the M7 programmable controller to operate on S7 objects on a remote automation system in the same way as a programming device.

The functions of the OMS which can be implemented with the help of a programming device and/or with M7 API calls are closely connected with the general memory model of an S7 CPU and/or with the equivalent memory model of the S7 object server of an M7 programmable controller.

These calls require that the connections to the communication partners have been configured using the SIMATIC Manager (see also Sections 8.5 and 8.6).

S7 CPU Memory Model

An S7 CPU has three logically separate memory areas:

- Load memory: The load memory is used to store, in other words load user programs (OBs, FBs and FCs) or data blocks without symbolic operand assignment or comments.

  The blocks stored in the load memory are called passive objects since they cannot be directly accessed by user programs.

  The load memory is subdivided into a read/write memory area and a read-only memory area. Both the read/write memory area and the read-only memory area (with limitations) can be accessed with programming device functions and/or with M7 API calls.

- Working memory: The working memory contains the active S7 objects, in other words those components which are operated on directly during the execution of user programs.

  The OMS function “linking” is used to transfer S7 objects from the load memory into the working memory and to activate them.

  The working memory only contains those parts of S7 objects which are necessary for execution of the user programs. Parts of objects which are not relevant for execution, for example block headers, remain in the load memory.

- System memory: The system memory contains the additional memory elements, for example flags, counters, times, stacks etc., which each S7 CPU make available to user programs.

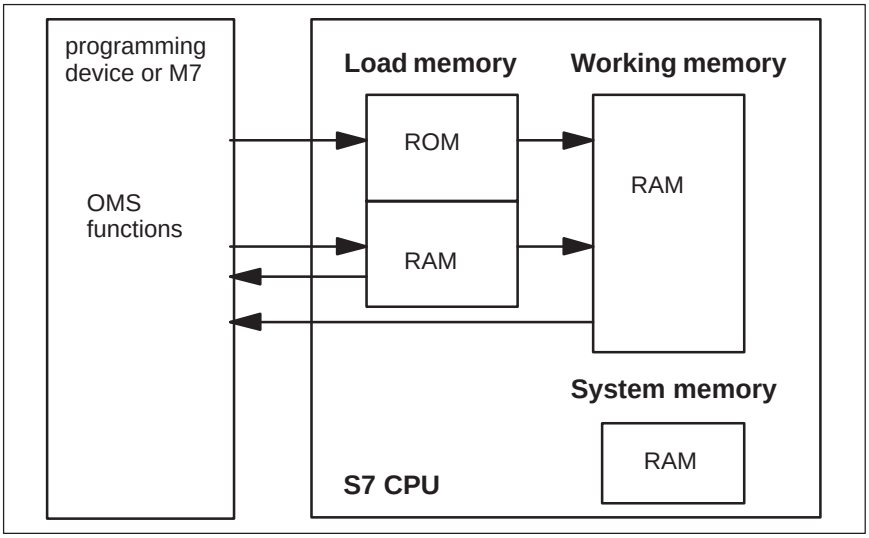


Figure 7-4 Memory Model of an S7 CPU

Note

The differences between the S7 object memory model of an M7 automation computer and the memory model of an S7 CPU are described in section 7.2.

Functions of the OMS

The following OMS functions can be executed from an M7 programmable controller using M7 API calls:

- ***M7OVSWrite()***: This function (upload) reads an active or passive block from the working memory or load memory
- ***M7OVSRead()***: This function (download) copies the specified block to the load memory of an S7 CPU and/or in the temporary load memory of an M7.
- ***M7OVSLinkIn()***: This function links the specified block, in other words it is copied from the load memory to the working memory and activated.
- ***M7OVSDelete()***: This function deletes the specified block. Both active and passive blocks can be deleted.
- ***M7OVSCompress()***: This function carries out a memory compression of the S7 CPU load memory (RAM).
- ***M7OVSFindFirst()***/***M7OVSFindNext()***: These functions are used to read the block list.

7.11 Uploading, Loading, Linking and Deleting Blocks

Uploading Blocks

The uploading function can be used for example to transfer individual code or blocks and data blocks from an S7 CPU or an M7 to an M7 programmable controller.

This function allows you for example to transfer and then evaluate the current contents of blocks in the working memory of a remote communication partner to your M7 programmable controller.

Due to the memory model of the S7 object server, uploading from an M7 programmable controller is limited to data blocks only.

Program Steps: Uploading Blocks

Proceed as follows to upload a block from an M7 programmable controller or an S7 CPU to an M7:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. The following call is used to upload the required block:

   M7OVSRead(flags,ConnID,pBitmap,pBuffer,nBufsiz,BlkTyp, BlkNum, pnBytes)

   The call ***OVSRead()*** copies the specified block from the working memory or load memory of the communication partner specified with the application link ID ***ConnID*** into the M7.

   The parameter ***flags*** is used to specify whether the block should be copied from the load memory (A_PASSIV) or from the working memory (A_LINKED_IN).

   Furthermore, ***flags*** can be used to specify whether the entire block (A_SSB and A_HEADER not set) or only the interface description (A_SSB) or only the block header (A_HEADER) should be copied, respectively.

   Flags is also used to specify whether the uploaded block should be stored in a buffer in main memory (A_FILE not set) or in a file on mass storage (A_FILE set).

   If the header flag is set, the returned one-byte bitmap specifies the location (ROM, RAM load memory, working memory) from where the block was uploaded.

If A_FILE is not set, then ***pBuffer*** specifies the address and ***nBufsiz*** the length of the buffer in which the uploaded block should be stored. If A_FILE is set, ***pBuffer*** specifies the path and name of the file in which the block will be stored; ***nBufsiz*** is not evaluated in this case.

The parameters ***BlkTyp*** and ***BlkNum*** are used to specify the block type (DB, OB, FB etc.) and the block number.

The number of actually uploaded bytes is returned in ***pnBytes***.

The return value indicates whether the call was successful or not.

4. Repeat step 3 in your user program until all required blocks have been transferred.
5. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Downloading Blocks

The downloading function can be used for example to transfer individual code blocks or data blocks from an M7 programmable controller to an S7 CPU or an M7. Due to the memory model of the S7 object server, downloading to an M7 programmable controller is limited to data blocks only.

The blocks are always stored in the RAM load memory (S7 CPU) or in the temporary load memory (M7).

Before copying the block into the load memory of the remote partner, a block header must be created with an appropriate call.

Program Steps: Downloading Blocks

Proceed as follows to upload a block from an M7 programmable controller or an S7 CPU to an M7:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   Following successful establishment of the application link the ***pConnID*** parameter points to the valid application link ID.
3. Create a memory buffer to store the block. First create a suitable block header in the local buffer with the following call:

   M7OVSSetObjectHeader(ptr,BlkNum,nLength,Language,Type, Attribute,ProtectionLevel)

   You must specify the address ***ptr*** and the length ***nLength*** in bytes of the memory area to receive the block. The length must be at least S7_OBJECT_HEADER_LENGTH in order to be big enough for the entire header.

   You must also specify the type ***Type*** and the number ***BlkNum*** of the block and the creation language ***Language*** (only for OB, FB and FC), the object attribute ***Attribute*** and the allowed access level ***ProtectionLevel***.
4. After you have created the module header, you can fill the user data area with the required information.
5. Upload the block with the following call:

   M7OVSWrite(flags,ConnID,pBuffer,nBufsiz,BlkTyp,BlkNum)

   This call copies the specified block to the load memory of the communication partner specified with application link ID ***ConnID***.

   The parameter ***flags*** is used to specify whether an existing block of the same type and the same number should be overwritten (A_UNCONDITIONAL set) or not (A_UNCONDITIONAL not set).

   Flags is also used to specify whether the block to be downloaded is stored in a buffer in main memory (A_FILE not set) or in a file on mass storage (A_FILE set).

   If A_FILE is not set, then ***pBuffer*** specifies the address and nBufsiz the length of the buffer in which the block to be downloaded is stored. If A_FILE is set, pBuffer specifies the path and name of the file in which the block is stored; ***nBufsiz*** is not evaluated in this case.

   The parameters ***BlkTyp*** and ***BlkNum*** are used to specify the block type (DB, OB, FB etc.) and the block number.

   The return value indicates whether the call was successful or not.
6. Repeat steps 2 to 4 in your user program until all required blocks have been transferred.

Note

The blocks are initially loaded in the RAM load memory (S7 CPU) or in temporary load memory (M7) and are not yet active.

The loaded blocks are only activated in a subsequent linking action.

Linking Blocks

The function ***M7OVSLinkIn()*** is used to transfer several code blocks and data blocks from the load memory (RAM or read-only memory) of an S7 CPU to the working memory.

Due to the memory model of the S7 object server, linking of blocks in an M7 programmable controller is limited to data blocks only. The OMS copies the required block from temporary load memory into the working memory at the next cycle control point. In addition, copies of the blocks to be linked are placed in the permanent load memory (RAMDIR).

Program Steps: Linking Blocks

Proceed as follows in order to link a block, in other words to copy it from load memory to working memory and activate it:

M7OVSLinkIn(ConnID,nBlks,pBlkList)

You must specify the application link ID ***ConnID*** of the communication partner.

You must also specify the number ***nBlks*** and the address ***pBlkList*** of a block list which lists the type and number of each of the blocks to be linked.

Deleting Blocks

The function ***M7OVSDelete()*** is used to delete several code blocks and data blocks in RAM load memory or in the working memory of an S7 CPU.

Deleting blocks releases memory for other purposes. Memory fragmentation which can arise due to repeated deleting and loading of blocks can be minimized with the compressing memory function (see section 7.12).

Due to the memory model of the S7 object server, deleting on an M7 programmable controller is limited to data blocks only.

Program Steps: Deleting Blocks

The following call is used to delete blocks in a remote communication partner:

M7OVSDelete(flags,ConnID,nBlks,pBlkList)

This function deletes blocks in the communication partner referred to by the application link ID ***ConnID***.

You must also specify the number ***nBlks*** and the address ***pBlkList*** of a block list which lists the type and number of each of the blocks to be deleted.

The parameter ***flags*** is used to specify whether the block should be deleted in load memory (A_PASSIV) and/or in working memory (A_LINKED_IN). Blocks in the read-only memory and/or in EPROM cannot be deleted with this call.

7.12 Compressing Memory and Setting the Memory Mode

Compressing Memory

If blocks in the user memory (RAM load memory and working memory) of an S7 CPU are repeatedly deleted and loaded, unusable memory gaps can occur that reduce the amount of available memory. This is called memory fragmentation.

The function ***M7OVSCompress()*** is used to rearrange the blocks in the S7 CPU user memory without gaps in order to maximize the amount of contiguous free memory.

The compress function can be executed in the operating mode RUN and STOP. All memory gaps are closed in both cases.

Program Steps: Compressing Memory

Proceed as follows to compress the user memory of an S7 CPU:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Compress the memory with the following M7 API call:

M7OVSCompress(ConnID)

You must specify a valid application link ID ***ConnID***.

4. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Note

The compress function is not supported on an M7.

Changing the Memory Mode

With an M7/S7 CPU, the function “change memory mode” is used to store a new initial condition, in other words the initial configuration which is used by the RESET function.

When changing the memory mode from RAM to EPROM, all active in other words linked S7 objects which were not created locally are copied to the ROM and/or changed to the EPROM mode. The memory mode of locally created S7 objects is not changed.

Blocks in ROM and/or in EPROM mode cannot be deleted. In order to delete them, they must first be copied again into the load memory and/or changed to the RAM mode.

Note

Owing to a system definition, data blocks with the attribute “dynamic” cannot be switched to EPROM mode. If you want to switch data blocks to EPROM mode from the programming device, you must ensure that the attribute “M7LANGTYP_CPU” (block was created dynamically by the CPU) does not appear in the object header but “M7LANGTYP_DB” (block created with the block editor). Use the call **M7OVSSetObjectHeader** to set the block header.

Program Steps: Changing the Memory Mode

In order to change the memory mode of an S7 CPU proceed as follows:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   The function ***M7KInitiate()*** opens an application link to a server via MPI. You must enter the host address of the partner in the ***pHostAddr*** parameter.

   After return of the call you can read the success of the action from the return value. Following successful application link establishment, the parameter ***pConnID*** contains a valid application link ID, which must be given via this application link in the case of further communication calls.

3. The following M7 API call is used to change the memory mode of an S7 CPU:

   M7OVSMemMode(flags,ConnID)

   The parameter ***flags*** specifies the required mode (A_PLC_RAM) or A_PLC_EPROM). Only one of the two flags should be set.

   You must also specify the application link ID ***ConnID***.

4. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

7.13 Reading the Block List of a Communication Partner

The functions ***M7OVSFindFirst()*** and ***M7OVSFindNext()*** are used to list the type and number of the blocks stored in an S7 CPU or M7.

The list contains information on which blocks of your communications partner are stored in which memory area (load or working memory).

Program Steps: Reading the Block List

The block list (block directory) is read one item at a time starting with the call ***M7OVSFindFirst()***, which searches for the first entry, and followed by a loop with the call ***M7OVSFindNext()***.

In order to read the entries in the block list of a remote communication partner proceed as follows:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Start the search process with the following M7 API call:

   M7OVSFindFirst(flags,ConnID,BlkTyp,Language,pFFBlkInfo)

   You must specify a valid application link ID ***ConnID***.

   The parameter ***flags*** specifies whether you are looking for directory entries of blocks in the load memory (A_PASSIV) or the working memory (A_LINKED_IN).

   If the flag A_DIRECTORY is set, the search is first made for blocks with the lowest block type numbers, in other words for OBS. The parameter ***BlkTyp*** is not evaluated in this case. This method is the best way to get a list of all available blocks.

   The flag A_LANGUAGE is used to search for blocks of the specified programming language only (parameter ***Language***).

   If the flag A_DIRECTORY is not set, the parameter ***BlkTyp*** is used to specify the require block type (OB, FB, DB, etc.). If the flag A_LANGUAGE is set, the parameter ***Language*** is used to specify the programming language (KOP, AWL, etc.) of the block to be searched.

You must also specify the address ***pFFBlkInfo*** of a find-first block structure.

If the call is successful, in other words if the return value is 0, the function enters the first entry found in the find-first info structure. This entry can then be read and processed by your user program.

If no entry is found, the function returns a return value of M7E_KSUB_EOF.

4. You can now look for the next directory entry with the following call:

   M7OVSFindNext(flags, ConnID, pFFBlkInfo)

   You must specify the same value for the ***flags*** parameter as in the previous ***M7OVSFindFirst()*** call. You must also specify the application link ID ***ConnID*** and the address ***pFFBlkInfo*** of a previously defined find-first block structure.

   If an appropriate entry is found (return value = M7SUCCESS), the call returns the entry in the find-first block structure. If no further entry is found, the call returns the return value M7_KSUB_EOF.

5. Repeat step 4 in your user program until the call does not return any further entries.

6. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Communication

8

Chapter Overview

8.1 Communication Mechanisms in SIMATIC S7/M7

The M7 API contains the following communication mechanisms for data exchange between CPUs and FMs of SIMATIC S7/M7 programmable controllers:

- Data exchange via non-configured connections
- Data exchange via configured connections (known as PBC (Programmed Block Communication from earlier M7-SYS versions)
- Communication via sockets

The communication type you choose depends primarily on the amount of data to be transmitted. The following table shows the differences between the communication mechanisms.

Table 8-1 Comparison between the Communication Mechanisms in SIMATIC M7

Criterion	Communication via non-configured connections	Communication via configured connections	Communication via sockets
Availability of the function calls	M7-300 and M7-400		
Communication connections	The connection is not configured but it is set up automatically while the function calls are being executed. After data transfer has finished, the connection either remains set or it is closed depending on parameter assignment.	Static or dynamic connections are configured in the SIMATIC Manager of STEP 7 during system configuration.	Connections are set up and closed dynamically in the user program.
Change to the STOP mode	Upon a transition to STOP the connection remains set.		
Number of connections to a communication partner	At any one time, a maximum of one connection is possible to a communication partner.	You can establish several connections to a communication partner.	You can establish several connections to a communication partner.
Address range	Modules can be addressed in the local S7/M7 station or in the MPI subnet.	Modules can be addressed in the MPI subnet, on PROFIBUS or on Industrial Ethernet.	Modules can be addressed in the Industrial Ethernet (TCP/IP) subnet.

Table 8-1 Comparison between the Communication Mechanisms in SIMATIC M7

Criterion	Communication via non-configured connections	Communication via configured connections	Communication via sockets
Maximum user data length	A user data length of 76 bytes is guaranteed.	The maximum transferrable user data length depends on the function call type and on the communication partner (S7/M7-300 or S7/M7-400).	The maximum transferrable user data length is 1024 bytes.
Number of variables transferred per call	You can transfer one variable.	With one function call you can transfer any number of variables. The number is limited by the maximum amount of transferred user data.	With one function call you can transfer any number of variables. The number is limited by the maximum amount of transferred user data.

Unidirectional and Bidirectional Communications

Communication via configured and non-configured connections allows you to:

- communicate with tasks/programs of other M7/S7-CPU (bidirectional communications functions, for example BSend/BRcv) or
- read or write the data of a remote M7/S7-CPU from your program (unidirectional communications functions, for example Put/Get)

Unidirectional communications functions initiated on the M7 side by the relevant M7-API call, do not require user programming on the opposite side. The relevant functions are integrated in the operating system of the communications partner. The operating system writes (reads) the desired data into (from) the addressed data area.

Bidirectional communications functions require user programming on both sides. For this reason, bidirectional communication can only be carried out with a suitable partner at the other end, and the two programs must be synchronized with each other to be ready to communicate at the same time.

Concept of Protection Levels

SIMATIC S7 provides the following protection level concept to protect user programs against unauthorized changes or the theft of know-how.

This is implemented on an M7 or S7-CPU with three protection levels (Level 1-3) which are associated with the disabling or enabling of functions.

In general, the higher the level number the more the limitations.

With bidirectional communication, all functions are allowed independently of the currently set protection level on the M7 or S7.

With unidirectional communication, the function can only be executed when the currently assigned protection level on the S7 or M7 CPU is less or equal to the protection level assigned to the function.

The protection levels for an M7 and/or and S7-CPU have the following characteristics:

- Level 1: No limitations, all functions can be executed
- Level 2: All functions for process control, monitoring and communication and functions for getting information are allowed. In the case of TIS (Test and commissioning) and OMS (Object Management System) functions, only reading is allowed.
- Level 3: All functions for process control, monitoring and communication and functions for getting information are allowed. No TIS and OMS functions are allowed.

Protection Levels of the Communication Functions

The following table is a summary of the protection levels which are required to execute the M7-API calls for communication via configured and non-configured connections. Part of the functions listed below are described in other chapters.

Table 8-2 Protection Levels of the Functions via Configured Connections

M7-API call	Admissible in the protection level
M7BUBCycRead()	1,2,3
M7BUBCycReadDelete(,,)	1,2,3
M7BUBCycReadStart()	1,2,3
M7BUBCycReadStop()	1,2,3
M7BUBRead()	1,2,3
M7BUBWrite()	1
M7DiagMode()	1,2,3
M7KAbort()	1,2,3
M7KEvent()	1,2,3

Table 8-2 Protection Levels of the Functions via Configured Connections

M7-API call	Admissible in the protection level
M7KInitiate()	1,2,3
M7KPassword()	1,2,3
M7KReadTime()	1,2,3
M7KWriteTime()	1,2,3
M7OVSCompress()	1
M7OVSDelete()	1
M7OVSFindFirst()	1,2,3
M7OVSFindNext()	1,2,3
M7OVSLinkIn()	1
M7OVSMemMode	1,2,3
M7OVSRead()	2
M7OVSWrite()	1
M7PBKBrcv()	1,2,3
M7PBKBsend()	1,2,3
M7PBKCancel()	1,2,3
M7PBKGet()	1,2,3
M7PBKPut()	1,2,3
M7PBKPrint()	1,2,3
M7PBKResume()	1,2,3
M7PBKStart()	1,2,3
M7PBKStatus()	1,2,3
M7PBKStop()	1,2,3
M7SZLRead()	1,2,3
M7PBKURcv()	1,2,3
M7PBKUSend()	1,2,3
M7WriteDiagnose()	1,2,3

If all functions need to be allowed on the remote station regardless of the currently assigned protection level, the application link set up with the ***M7KInitiate()*** call must be authenticated for this (see section 8.6).

Table 8-3 Protection Levels of the Functions via Non-Configured Connections

M7-API call	Admissible in the protection level
M7PBKXGet()	1,2,3
M7PBKXPut()	1,2,3
M7PBKXRcv()	1,2,3
M7PBKXSend()	1,2,3
M7PBKXCancel()	1,2,3
M7PBKXAbort()	1,2,3
M7PBKIGet()	1,2,3
M7PBKIPut()	1,2,3
M7PBKIAbort()	1,2,3

Protection Levels of an M7 or S7-CPU

The currently assigned protection level on an M7 or S7-CPU is mainly affected by two settings:

- Configured protection level: the protection level and password parameters are stored encoded in SDB0 of the module.
- Selected protection level: the protection level is also determined by the position of the module's operating mode switch as follows:
  - RESET position: Protection level 3
  - STOP position: Protection level 1
  - RUN position: Protection level 2
  - RUN_PROG position: Protection level 1

If the protection level and the password have been entered in SDB0, the currently assigned protection level of an M7 or S7-CPU is the numerically higher value of the configured and the selected protection level.

If no password is entered into SDB0, the currently assigned protection level is equal to the selected protection level.

8.2 Communication via Non-Configured Connections

There is a simple way of programming communication within the local SIMATIC station or between two subnet nodes **without having to configure connections**: using communication functions for non-configured connections.

You can transfer up to 76 bytes of data within the local station, an MPI subnet or on PROFIBUS.

Concept

When using communication functions for non-configured connections, the connection is set and an application link is established while the call is being executed.

Depending on the ***flags*** parameter (whether the CONT flag is set or not), the connections and application links remain established or are terminated after data transfer.

An application link that is not terminated after data transfer (CONT flag set), continues to be available to all function calls for non-configured connections that whish to address the same communication partner. This application link can be used by any task. The first call to this communication partner, for which the CONT **flag** is not set, causes the connection and the application link to be terminated after data transfer.

This results in the following communication properties:

- The number of communication partners that can be reached one after the other is higher than the number of communication partners that can be reached simultaneously (see the hardware manuals and the User Manual for module-specific data).
- If at the moment no connection or application link can be established due to lack of resources, this is indicated in an error code. You must call this function later in the program, but even then it is not guaranteed that the connection will be established. Please check the resources required by your program and use a CPU with a higher number of connection resources if needed.

Application links already established by function calls for configured connections cannot be used by function calls for non-configured connections.

The application link that has been established for a communication function being executed, cannot be used by other function calls at the same time. Other requests to the same communication partner can be executed only after the current one has finished.

Note

If your program contains several requests for the same communication partner, you must ensure that the functions that return with the M7E_KSUB_CONN_ACTIVE error code are called again at a later time.

Inside or Outside the Local Station?

With communication functions for non-configured connections data can be transferred either inside the local station or between two stations (subnet nodes). There are two ways of addressing:

- Inside the local station the partner is addressed by the start I/O address (logical base address), for example
  - – Between an S7-/M7-CPU and an FM 356/456-4 or
  - – Between an S7-/M7-CPU and an S7-200 within a DP master system.

  Note that distributed I/Os are treated the same way as I/Os in the central assembly.
- Partners outside the local station are addressed by the node address of the remote station, for example between an S7-/M7-300 and an S7-/M7-400 in an MPI subnet.

Communication Inside the Local Station

Table 8-4 contains the M7-API functions for communication inside the local station followed by a short description.

Table 8-4 M7-API Function for Communication inside the local Station (non-configured connections)

Function Call	Description
M7PBKIPut()	Start asynchronous variable writing (unidirectional communication function)
M7PBKIGet()	Start asynchronous variable reading (unidirectional communication function)
M7PBKIAbort()	Close an application link

Communication between Two Stations

Table 8-5 contains the M7-API functions for communication outside the local station followed by a short description.

Table 8-5 M7-API Function for Communication outside the local Station (non-configured connections)

Call	Meaning
M7PBKXPut()	Start asynchronous variable writing (unidirectional communication function)
M7PBKXGet()	Start asynchronous variable reading (unidirectional communication function)
M7PBKXAbort()	Close an application link

Table 8-5 M7-API Function for Communication outside the local Station (non-configured connections)

Call	Meaning
M7PBKXSend()	Send data (bidirectional communication function)
M7PBKXRcv()	Receive data (bidirectional communication function)
M7PBKXCancel()	Cancel running receive request

Information on Communication Functions

See the Reference Manual “System Software for M7-300/400 Standard and System Functions” for detailed information on the communication functions (M7-API calls).

This chapter only makes qualitative reference to the functions and does not provide detailed description of parameters.

See chapter 8.7 for general information on unidirectional and bidirectional communication functions.

8.3 Programming Unidirectional Communication via Non-Configured Connections

Unidirectional communications functions initiated on the M7 side by the relevant M7-API call, do not require user programming on the opposite side. The relevant functions are integrated in the operating system of the communications partner. The operating system writes (reads) the desired data into (from) the addressed data area.

Unidirectional Communication Functions

Basic sequence of unidirectional communication functions:

1. The unidirectional function (M7PBK...**Put**.. or M7PBK...**Get**) is called on the client side.
2. The user program is sent a message when the operation is finished. The success or failure of the action can be determined from the FRB referenced in the message.
3. On the remote side, the corresponding communication function is carried out by an internal operating system function asynchronously of the user program. The operating system reads (writes) the required data to the (from the) specified S7 operand area (S7-CPU) or S7 object area (M7).

Writing Data to and Reading Data from the Communication Partner

Proceed as follows if you want your C program to overwrite variables in the S7 data area of the remote station with data from variables of the local S7 object server or to read variables from the S7 data area of the remote station:

1. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)

2. Specify the data type and addresses of the variables. Create elements of type ***M7VARADDR*** in the global data or on the heap.

   – Use ***SrcVar*** to store the address of the local source variable for write requests.

   – Use ***DstVar*** to store the address of the local destination variable for read requests.

   – Use ***RemoteVar*** to store the address of the remote variable to be overwritten or read.

   The type ***M7VARADDR*** is defined in M7API.H and specifies a consecutive number of items within an S7 object.

   Initialize the elements containing the addresses of the variables to be transferred. Make sure that the data types of the source/destination and remote variables are compatible.

3. Start the asynchronous data transfer with the following M7-API calls:

   – Write inside the local station:

   M7PBKIPut(flags,IOID,LADDR,pRemoteVar,pSrcVar,pCommFRB,MPrio)

   – Read inside the local station:

   M7PBKIGet(flags,IOID,LADDR,pRemoteVar,pDstVar,pCommFRB, MPrio)

   – Write to a partner outside the local station:

   M7PBKXPut(flags,DEST_ID,pRemoteVar,pSrcVar,pCommFRB, MPrio)

   – Read from outside the local station:

   M7PBKXGet(flags,DEST_ID,pRemoteVar,pDstVar,pCommFRB, MPrio)

   The parameter ***flags*** specifies whether the application link is to remain (=CONT set) or to be terminated (=CONT not set). If the application link is to remain, it must be terminated at a later point with the call ***M7PBKIAbort()*** or ***M7PBKXAbort()*** respectively.

   In order to identify the communication partner inside the local station, you must specify a valid start address composed of ***IOID*** and ***LADDR***. In order to identify the communication partner outside the local station, you must specify the node address (MPI address) of the remote station.

   pRemoteVar and ***pSrcVar*** point to the addresses of the remote and source variables for write requests.

pRemoteVar and ***pDstVar*** point to the addresses of the remote and destination variables for read requests.

You must also specify the address of the associated FRB ***pCommFRB***. The parameter ***Mprio*** specifies the required priority of the returned message.

4. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

5. Evaluate the received message. The system notifies the end of data transfer to the task with a message of the type:

   – M7MSG_PBK_DONE for write requests

   – M7MSG_PBK_NDR for read requests

   If you have requested several transfers, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer can be determined with the following call:

   M7GetCommStatus((M7COMMFRB_PTR)pMessageParam)

   If the transfer was successful, a valid value should now be present:

   – For PUT requests: in the referenced variables of the remote station, where it and can be read and processed by the remote user program

   – For GET requests: in the referenced variables of the S7 object server where it can be read and processed by your user program with the ***M7Read..()*** call

6. Repeat the steps 3. to 5. (loop in your C program) if you want to send the source data area cyclically to the communication partner.

7. When no further data transfer is needed and if the communication function has been called with the CONT flag set, you can close the application link to the communication partner with one of the following calls:

   – Inside the local station:

     M7PBKIAbort(IOID,LADDR)

   – Outside the local station:

     M7PBKXAbort(DEST_ID)

   You must specify the address of the communication partner: the start address ***IOID,LADDR*** or the node address ***DEST_ID*** respectively.

8.4 Programming Bidirectional Communication via Non-Configured Connections

Bidirectional communication functions need user programming on both sides. For this reason, bidirectional communication can only be carried out with a suitable partner at the other end, and the two programs must be synchronized with each other to be ready to communicate at the same time.

Bidirectional Communication Functions

Basic sequence of bidirectional communication functions:

1. The bidirectional function (***M7PBKXSend...***) is called on the sender side.
2. The user program is sent a message when the operation is finished. The success or failure of the action can be determined from the FRB referenced in the message.
3. The complementary function (***M7PBKXRcv...***) is called on the receiver side.
4. Upon successful data receipt the user program is sent a message.

Rule

If both send and receive requests are to be executed in your user program, the receive requests must be called first, before the send requests.

Receiving Data

Proceed as follows if you want your C program to receive data from a remote station using the bidirectional communication function ***M7PBKXRcv()***:

1. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)

2. Now decide whether the data to be transferred should be received to the program's own data area or in a data area of the S7 object server.

   In the data area of the receiver task: in the global data area or on the heap create a buffer ***DstVar[nLength]*** (formal data type ***M7VARADDR***) for the data to be received.

   In the data area of the S7 object server: specify the area in the S7 object in which you want to receive the data. Create a variable ***DstVar*** of type ***M7VARADDR*** in the global data area or on the heap. Initialize this variable with the address information of the area in the S7 object.

3. Start data transfer from the remote station using the following M7-API call:

   M7PBKXRcv(flags,R_ID,pDstVar,nLength,pCommFRB,MPrio)

   The call starts to receive data sent from the remote station using the ***M7PBKXSend***.

   You must specify the receiver ID ***R_ID***. The call only accepts data from the sending station if the ID is the same as the receiver ID specified in the send request.

   The parameter ***flags*** is used to specify whether the data to be transferred should be copied to the data area of the sender task (***flags=A_USER***) or to the data area of the S7 object servers (***flags=A_ZERO_FLAG***).

   The pointers ***pDstVar*** and ***nLength*** specify:

   – The starting address and length of the buffer in which the received data should be written if ***flags=A_USER***,

   – The address information of the S7 object variable if ***flags=A_ZERO_FLAG***.

4. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

5. Evaluate the received message. The system notifies the end of data transfer to the task with the message M7MSG_PBK_NDR.

   If you have requested several transfers, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer and/or an error in the asynchronous part of the send request can be determined with the following macro:

   M7GetCommStatus((M7COMMFRB)pMessageParam)

   The number of bytes received can be determined with the following macro:

   M7GetCommRcvLen((M7COMMFRB)pMessageParam)

6. Repeat the steps 3. to 5. (loop in your C program) if you want to receive the source data cyclically from the communication partner.

7. A running receive request ***M7PBKXRcv*** can be cancelled with the following call:

   M7PBKXCancel(pCommFRB)

   You must specify the address ***pComFRB*** of the request's FRB.

Sending Data

Proceed as follows if you want your C program to send data to a remote station using the bidirectional communication function ***M7PBKXSend()***:

1. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)

2. Now decide whether the data to be transferred should sent from the program's own data area or in a data area of the S7 object server.

   In the data area of the sender task: in the global data area or on the heap create a buffer ***SrcVar[nLength]*** (formal data type ***M7VARADDR***) for the data to be sent. Initialize the buffer with the data to be sent.

   In the data area of the S7 object server: specify the variables of the S7 objects whose contents you want to transfer. Create a variable ***SrcVar*** of type ***M7VARADDR*** in the global data area or on the heap. Initialize this variable with the address information of the items to be transferred.

3. Start data transfer to the remote station using the following M7-API call:

   M7PBKXSend(flags,DEST_ID,R_ID,pSrcVar,nLength,pCommFRB, MPrio)

   The parameter ***flags*** is used to specify whether the data to be transferred is contained in the data area of the sender task (***flags=A_USER***) or in the data area of the S7 object server (***flags=A_ZERO_FLAG***).

   Additionally you specify with the parameter ***flags***, what happens to the application link after the data transfer is finished:

   – ***flags=CONT*** the application link is maintained

   – ***flags≠CONT*** the application link is closed

   You must also specify the node address ***DEST_ID*** of the communication partner, the ID ***R_ID*** of the remote receiver block and/or receive call and the starting address ***pSrcVar*** and the length ***nLength*** of the buffer.

   Depending on whether the ***A_USER*** flag is set or not, the following happens:

   – If ***flags*** = ***A_USER***, the contents of the buffer are transferred.

   – If ***flags*** = ***A_ZERO_FLAG***, the call determines the address of the item in the S7 object server from the address variable in the buffer and then transfers it.

   If neither the A_USER flag nor the CONT flag is set, you must set ***flags*** = ***A_ZERO_FLAG***.

   You must also specify the address of the associated FRB ***pCommFRB***. The parameter ***Mprio s***pecifies the required priority of the returned message.

4. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

5. Evaluate the received message. The system notifies the end of data transfer to the task with the message with the ID M7MSG_PBK_DONE.

   If you have requested several transfers, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer and/or an error in the asynchronous part of the send request can be determined with the following macro:

   M7GetCommStatus((M7COMMFRB)pMessageParam)

6. Repeat the steps 3. to 5. (loop in your C program) if you want to send the source data cyclically to the communication partner (with the CONT flag set).
7. When no further data transfer is needed and if the communication function has been called with the CONT flag set, you can close the application link to the communication partner with the following call:

   M7PBKXAbort(DEST_ID)

   You must specify the node address ***DEST_ID*** of the communication partner.

8.5 Communication via Configured Connections

The functions for communication via configured connections can be called on all M7-300/400-CPUs and -FMs to transfer data to and from S7-300/400 CPUs, M7-300/400 CPUs and M7 FMs. With these functions larger data amounts up to max. 64 kbytes can be transmitted via various subnets (MPI, PROFIBUS, Industrial Ethernet) as well as inside a node via communication bus (K bus). These functions require S7 single -system connections that are configured with STEP 7. This allows programming devices (PGs) or devices installed some distance away to be connected to the automation system.

The communication functions via configured connections are oriented towards the characteristics of an S7-CPU. For this reason, programming the communication functions with the M7-API is described similar with the S7 programming.

Functions Overview

The M7 API function calls requiring configured connections can be divided into the following groups:

- General calls for application links: These functions allow establishing, closing and getting information on application links (see chapter 8.6).
- Calls for sending and receiving data: These functions allow M7 RMOS32 tasks to access data on remote M7/S7-CPUs (unidirectional communication function) and/or to communicate directly with tasks/programs on another M7/S7-CPU (bidirectional communication function).

- Control and information calls: These functions allow you to interrogate the device status of the communication partner and/or to send a request to change the operating mode of the remote device to STOP, START or RESUME (see chapter 8.11).

This chapter describes only the function groups listed above. The following function groups are described in other chapters of the manual.

- Calls for the OMS: The M7-API calls for the OMS (Object Management System) allow handling of S7 objects on a remote automation system. S7 objects include for example data blocks or function blocks.

  The calls allow execution of “programming device functions” such as copying, linking, deleting and uploading S7 objects directly from your user program. These functions are described in chapters 7.10 to 7.13.

- Calls for the OI: M7 calls for the OI (Operator Interface) allow the implementation of your own OI programs on the M7 programmable controller.

  For example, the M7-API provides functions to read and write and/or cyclic reading of variables in a remote automation system. These functions are described in chapters 7.7 to 7.9.

- Calls to read/set the time: These calls can be used to read and/or set the system time of a remote server (see chapter 5.11).

- Calls for the diagnostic server: The diagnostic server allows a program on the M7 programmable controller to register for diagnostic messages which are output by a remote automation system.

  Furthermore, the program can read the system status list (SSL) and/or parts of the SSL (such as the diagnostics buffer) of the communication partner. This allows evaluation of events which are communicated by the communication bus (for example cycle times, memory configuration, cycle overflow, etc.) in order to react appropriately to faults.

  These functions are described in chapters 5.17 to 5.20.

8.6 Establishing and Authenticating Application Links via Configured Connections

Before it is possible to communicate with another partner, it is necessary to establish an application link between the two programs. The application link is based on a configured connection.

All connections must be configured with the SIMATIC Manager when commissioning the automation system, irrespective of when the connections need to be used. It is only possible to open an application link between partners, to send requests and to exchange data with a remote partner after appropriate configuration has taken place.

Active and Passive Connections

The connection can be configured to be active or passive.

When setting the connection, the passive station always waits for a connection attempt by the active station, in other words the active station sends a CONNECTION REQUEST which can be accepted or rejected by the passive station.

Dynamic and Static Connections

The connection can be configured to be dynamic or static.

In the case of **dynamic** connections, the connection is set with the call ***M7KInitiate()***. If for an active connection no partner answers, the call returns with an error message. For a passive connection, the ***M7KInitiate()*** call waits until a connection request has been received.

In the case of **static** connections, connection establishment is controlled by the system. The connections are generally available permanently. The connection is re-established automatically following interruptions. The call ***M7KInitiate()*** always returns a ConnID reference for further communication calls, even when a connection to the remote partner has not yet been established. However, productive send and receive requests can only be handled by a connection which is already established.

Configuring Connections with the SIMATIC Manager

Before using the communication functions in the application program, you must configure the connections to the communication partner in the SIMATIC Manager. You must assign a connection table to each programmable module (CPU/FM) that takes part in the communication via a configured subnet. All connections configured for one module must be entered to its connection table. The procedure is described in the online help of the SIMATIC Manager's Connection Configuration and in the STEP 7 User Manual.

Establishing an Application Link

The following call is used to establish an application link between a user program and a communication partner within a SIMATIC station or via MPI:

M7KInitiate(pConnID,pHostAddr)

You must specify the host address ***pHostAddr*** of the remote partner as a string. ***pHostAddr*** contains the decimal connection number (for example “17”) from the connection configuration table as a string. The link between the two programs can take place either via configured or free connections.

If you specify the parameter ***pHostAddr*** as the string “local”, this opens a loopback application link to the local host.

After the call returns, the return value indicates whether the call was successful or not. If the call was successful, the parameter ***pConnID*** points to a valid application link ID which must be specified for all further communication requests for this application link.

An application link can be established on a dynamic as well as on a static connection.

Note

The ***M7KInitiate()***function is executed synchronously without time-out. Because of that, when establishing more than one application link using dynamic connections , the program of the passive partner can be blocked if no partner is available. In order to avoid possible blockings you can do one of the following:

– Use static connections instead of dynamic ones or

– Implement one task for each application link to be established.

Authenticating the Application Link

Regardless of the currently assigned protection level of the remote station, if it is required to allow all functions, the application link which was established with the call ***M7KInitiate()*** must be authenticated with the following call:

M7KPassword(flags,ConnID,pszPassword)

You must specify a valid application link ID ***ConnID*** and the password ***pszPassword*** of the remote station.

If the ***flags*** parameter is specified as SET_PASSWORD and the correct password is specified, all functions are allowed if the call is successful.

If no flags are set in the ***flags*** parameter, then only the functions of the currently assigned protection level of the partner station are allowed.

Closing the Application Link

If the application link between two communication partners is no longer required, it can be closed again with the following call:

M7KAbort(ConnID)

You must specify a valid application link ID ***ConnID***. All waiting asynchronous requests are automatically deleted.

Inquiring the Status of the Application Link

The status of an application link can be determined with the following call:

M7GetConnStatus(ConnID, pConnState)

You must specify a valid application link ID ***ConnID***. After the call returns, the return value indicates whether the call was successful or not. If the call was successful, the parameter ***pConnState*** points to the current state of the application link.

8.7 General Information on Communication Functions for Configured Connections

The communication functions for configured connections are used for data exchange between two communication partners. A differentiation is made between unidirectional and bidirectional functions. The program must specify the quantity of data and the data transfer method in the M7-API calls.

Unidirectional Communication Functions

Unidirectional communication functions which are initiated on the M7 side by an appropriate M7-API call do not require any user programming on the other station. The appropriate functions are integrated in the operating system of the communication partner.

On the client side, the unidirectional communication function is started asynchronously. The user program is sent a message when the operation is finished. The success or failure of the action can be determined from the FRB referenced in the message.

On the remote side, the corresponding communication function is carried out by an internal operating system function asynchronously of the user program. The operating system reads (writes) the required data to the (from the) specified S7 operand area (S7-CPU) or S7 object area (M7).

Functional Sequence of *M7PBKPut()*

Figure 8-1 illustrates the functional sequence of unidirectional communication using the example of the M7-API call ***M7PBKPut()***. The call transfers the data via communication driver to the operating system on the S7/M7 side.

In the remote system, the data is then written to the specified S7 data area asynchronously of the cyclic user program, and can then be be read and processed by the user program.

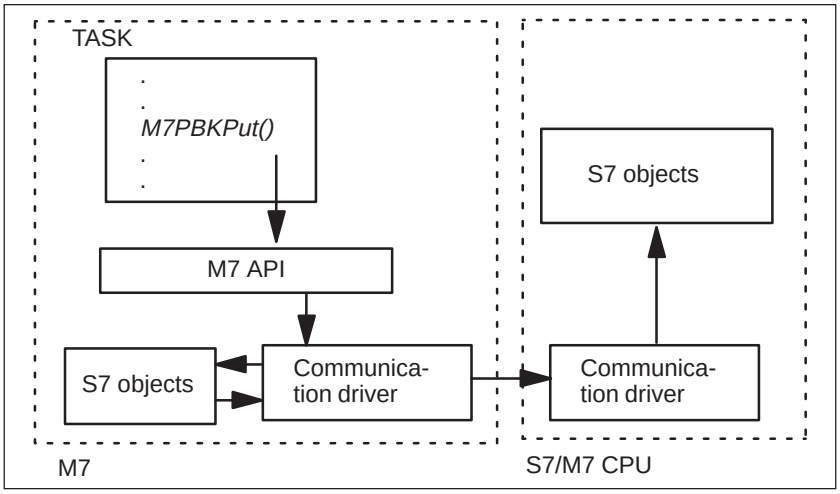


Figure 8-1 Unidirectional Communication between an M7 and an S7/M7-CPU using ***M7PBKPut***

Functional Sequence of *M7PBKGet()*

Figure 8-2 illustrates the functional sequence of unidirectional communication using the example of the M7-API call ***M7PBKGet()***. The call fetches the data from the operating system on the remote S7/M7 side.

In the remote system, the data is read from the specified S7 data area and transmitted to the communication driver on the M7 side, where it can then be be read and processed by the task.

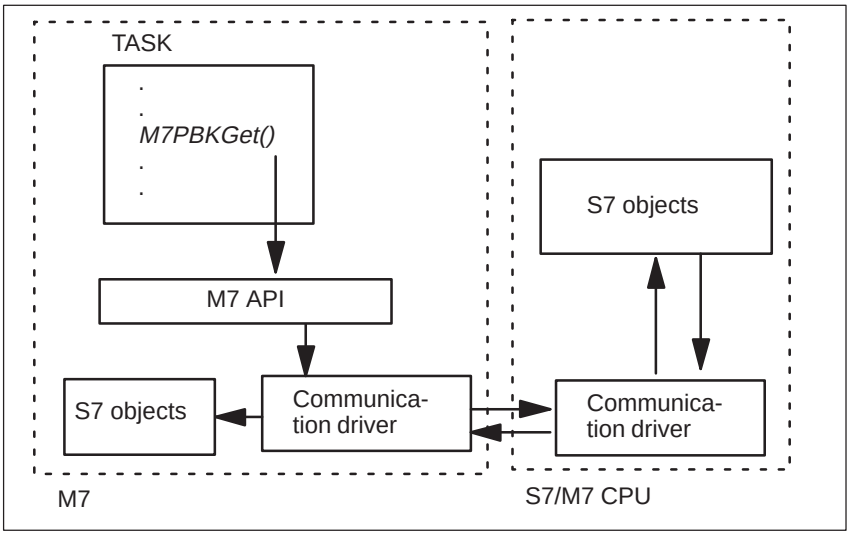


Figure 8-2 Unidirectional Communication between an M7 and an S7/M7-CPU using ***M7PBKGet***

Functional Sequence of the S7 SFB PUT

Figure 8-3 illustrates the functional sequence of the SFB PUT on an S7-CPU. The SFB PUT transfers the data to the communication driver on the M7 side, where the data is written to the S7 data area managed by the Object Server asynchronously of the user task.

User tasks on the M7 side can use callback functions to register themselves with the Object Server for notification when the specified data area is accessed (see chapter 7.6). On receiving the notification message, the task can then read the data from the corresponding S7 object with the M7-API call ***M7Read()***.

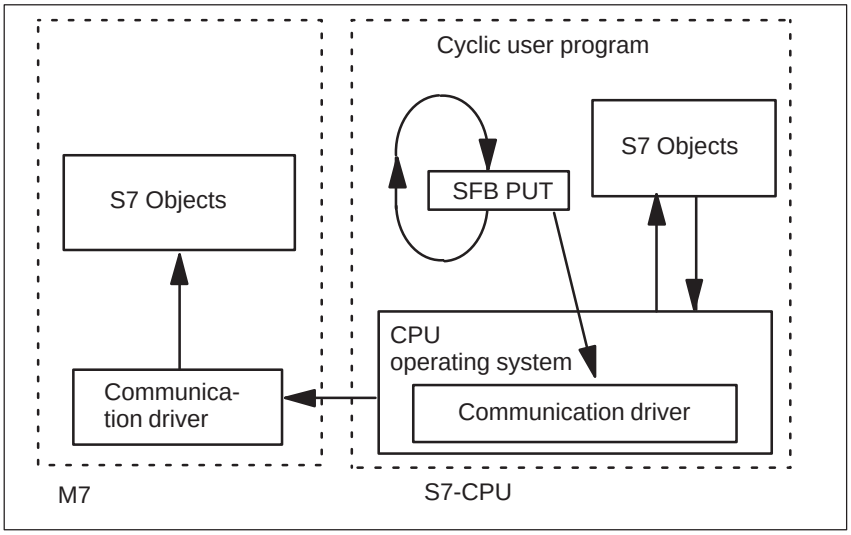


Figure 8-3 Unidirectional Communication between an S7-CPU and an M7 using ***PUT***

Functional Sequence of the S7 SFB GET

Figure 8-4 illustrates the functional sequence of the SFB GET on an S7-CPU. The SFB GET reads the data from the communication driver on the M7 side.

The user tasks on the M7 side must have prepared the required data. The data must have been written to the appropriate S7 object with one of the M7-API calls ***M7Write...()***. The data is then read asynchronously of the user task from the specified S7 data area.

Subsequently the cyclic user program on the S7 side can process the read data.

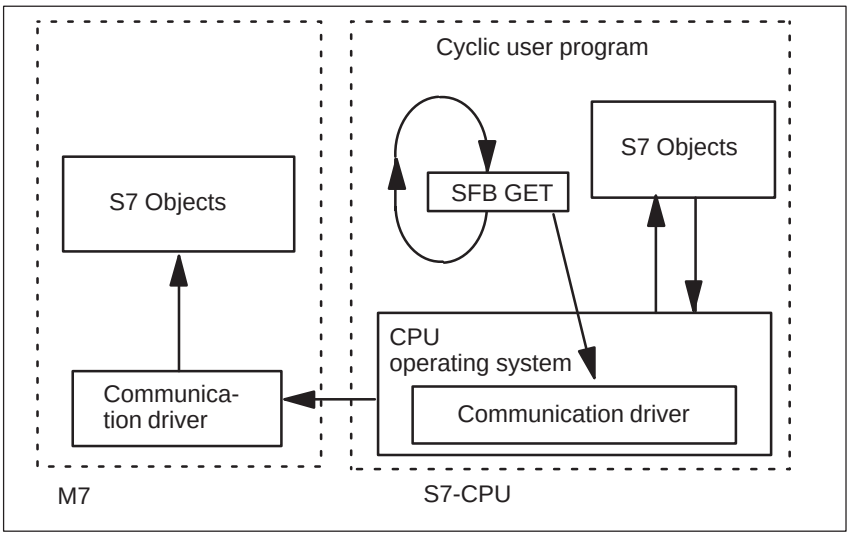


Figure 8-4 Unidirectional Communication between an S7-CPU and an M7 using GET

Bidirectional Communication Functions

Bidirectional communication functions need user programming on both sides. For this reason, bidirectional communication can only be carried out with a suitable partner at the other end, and the two programs must be synchronized with each other to be ready to communicate at the same time.

The bidirectional communication for configured connections is started asynchronously on the client side. The user program is sent a message when the operation is finished. The success or failure of the action can be determined from the FRB referenced in the message.

Functional Sequence of M7PBKBsend and M7PBKBrcv

Figure 8-5 illustrates the functional sequence of bidirectional communication using the example of the M7-API call ***M7PBKBsend()***. This function transfers data to a BRCV block on the S7-CPU side. The data is only accepted when the BRCV block is explicitly called on the S7-CPU side.

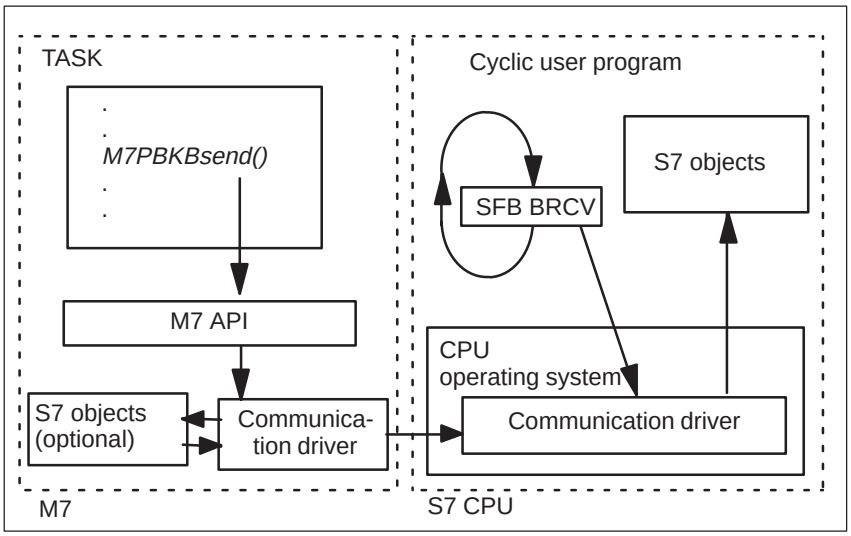


Figure 8-5 Bidirectional Communication between an M7 and an S7-CPU

Figure 8-6 illustrates the functional sequence of the BSEND call on an S7-CPU. The BSEND call transfers the data to the communication driver on the M7 side, where the data is received by the user task using the M7-API call ***M7PBKBrcv()***.

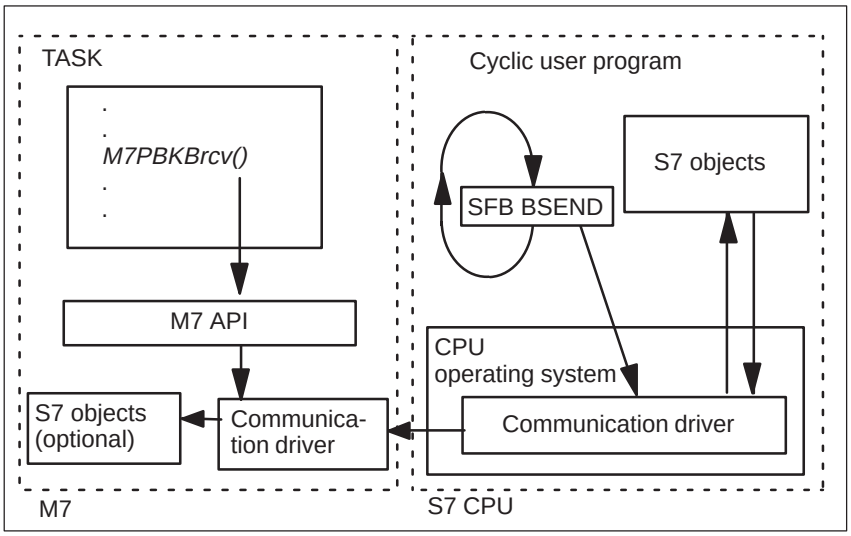


Figure 8-6 Bidirectional Communication between an S7-CPU and an M7

8.8 Programming Unidirectional Communication Functions

Sending Data to the Communication Partner

Proceed as follows if you want your C program to overwrite variables in the S7 data area of the remote station with data from variables of the local S7 object server:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)
4. Specify the number ***nVars***, data type and source and remote address of the variables which are used. Create two arrays in the global data or on the heap of size ***nVars*** with elements of type ***M7VARADDR*** (***SrcVar***[***nVars***] and ***RemoteVar***[***nVars***] respectively).

   The arrays ***SrcVar[nVars]*** and ***RemoteVar[nVars]*** are used to store the addresses of the source and remote variables in the local and/or remote station, respectively.

   The type ***M7VARADDR*** is defined in M7API.H and specifies a consecutive number of items within an S7 object.

   Initialize the elements of the source and remote arrays with the addresses of the data to be transferred. Make sure that the data types of the source and remote variables are compatible.
5. Start the data transfer to the remote station with the following M7-API call:

   M7PBKPut(ConnID,nVars,pRemoteVar,pSrcVarAddr,pCommFRB,MPrio)

   In order to identify the communication partner, you must specify a valid application link ID ***ConnID***, the number ***nVars*** of variables to be transferred and pointers ***pSrcVar***, ***pRemoteVar*** to the arrays with the addresses of the source and remote variables.

   You must also specify the address of the associated FRB ***pCommFRB***. The parameter ***Mprio*** specifies the required priority of the returned message.

6. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

7. Evaluate the received message. The system notifies the end of data transfer to the task with the message with the ID M7MSG_PBK_DONE.

   If you have requested several transfers for configured connections, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer can be determined with the following call:

   M7GetCommStatus((M7COMMFRB_PTR)pMessageParam)

   If the transfer was successful, valid value should now be present in the referenced variables of the remote station and can be read and processed by the remote user program.

8. Repeat the steps 4 to 6 (loop in your C program) if you want to send the source data area cyclically to the communication partner.

9. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Reading Data from the Communication Partner

Proceed as follows if you want your C program to read data from a remote station with the help of unidirectional communication functions and store it in the S7 data area of the S7 object server:

1. Configure a connection to the communication partner using the SIMATIC Manager.

2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.

3. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)

4. Specify the number ***nVars***, data type and source and remote address of the variables which are used. Create two arrays in the global data or on the heap of size ***nVars*** with elements of type ***M7VARADDR*** (***DstVar***[***nVars***] and ***RemoteVar***[***nVars***] respectively).

   The arrays ***DstVar[nVars]*** and ***RemoteVar[nVars]*** are used to store the addresses of the destination and remote variables in the local and/or remote station, respectively.

   The type ***M7VARADDR*** is defined in M7API.H and specifies a consecutive number of items within an S7 object.

   Initialize the elements of the destination and remote arrays with the addresses of the data to be transferred. Make sure that the data types of the destination and remote variables are compatible.

5. Start the data transfer to the remote station with the following M7-API call:

   M7PBKGet(ConnID,nVars,pRemoteVar,pDstVar,pCommFRB,MPrio)

   In order to identify the communication partner, you must specify a valid application link ID ***ConnID***, the number ***nVars*** of variables to be transferred and pointers ***pDstVar***, ***pRemoteVar*** to the arrays with the addresses of the destination and remote variables.

   You must also specify the address of the associated FRB ***pCommFRB***. The parameter ***Mprio*** specifies the required priority of the returned message.

6. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

7. Evaluate the received message. The system notifies the end of data transfer to the task with the message with the ID M7MSG_PBK_NDR.

   If you have requested several transfers for configured connections, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer can be determined with the following call:

   M7GetCommStatus((M7COMMFRB_PTR)pMessageParam)

   If the transfer was successful, valid value should now be present in the referenced variables of the S7 object server and can be read and processed by your user program with the following call:

   M7Read..()

8. Repeat the steps 4 to 6 (loop in your C program) if you want to receive the remote data cyclically from the communication partner.
9. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Note

The maximum amount of data which can be transferred with an **M7PBKPut()** or an **M7PBKGet()** call is limited by the maximum PDU size (protocol data unit) of the application link.

Determining the PDU Size

The maximum PDU size for an application link can be determined with the following call:

M7GetPduSize(ConnID,pnPduSize)

You must specify a valid application link ID ***ConnID***. The call returns the max. PDU size in bytes in the parameter ***pnPduSize***.

Sending Data with Format Specification

Proceed as follows if you want your C program to send data with a specific format to the communication partner:

M7PBKPrint(..., ConnID, ...)

You must specify a valid application link ID ***ConnID***. The other parameters depend on your destination system's requirements.

This function is unidirectional; this means that the receive function is carried out by the operating system of the communication partner. On the M7 side no receive function is implemented for such send requests. The function ***M7PBKPrint()*** is used for example for assigning parameters to certain Point-To-Point CPs.

8.9 Structure of Bidirectional Communication Functions for Configured Connections

Overview

Bidirectional communication functions are used for data transfer between communication partners. Appropriate calls must be issued by user programs on both sides of the connection.

The following calls are available for bidirectional communication on an M7 system:

M7PBKBsend()/M7PBKBrcv()– block-oriented send/receive

M7PBKUSend()/M7PBKURcv()– uncoordinated send/receive

The two call types differ by the following:

Function	***M7PBKBsend()*** ***M7PBKBrcv()***	***M7PBKUSend()*** ***M7PBKURcv()***
Acknowledge-ments	An acknowledgement is sent upon data transfer. This guarantees, for example upon the return of ***M7PBKBsend***, that on the receive side an ***M7PBKBrcv*** call or a **B_RCV** block have been issued.	No acknowledgement is sent upon data transfer.
Amount of transferred data	Up to 64 kbytes of data can be transferred with one call.	Maximum one PDU can be trans-mitted with one call. The PDU size depends on the communication partner and must be determined with the ***M7GetPDUSize*** call.

Functional Sequence

In combination with the corresponding SEND and RCV blocks on an S7-CPU, these M7-API calls allow a block-oriented or an uncoordinated data transfer between sender and receiver.

The data is transferred from source to destination asynchronously; for larger quantities of data this generally takes several CPU cycles.

The communication process is closed directly after transferring all of the data. The sender is notified of the end of the transfer with the message M7MSG_PBK_DONE. No acknowledgement is sent of whether the received data has been processed by the partner or not.

On the partner station, the transferred data is received with the BRCV block (S7-CPU) or ***M7PBKBrcv()*** (M7-CPU) and copied to the corresponding destination data area. The receipt of new data is notified with the message M7MSG_PBK_NDR.

Determining the PDU Size

The maximum PDU size for an application link can be determined with the following call:

M7GetPduSize(ConnID,pnPduSize)

You must specify a valid application link ID ***ConnID***. The call returns the max. PDU size in bytes in the parameter ***pnPduSize***.

Sending Data on an M7-CPU

The following calls are used on an M7-CPU for sending the data:

M7PBKBsend()

M7PBKUSend()

The ***M7PBKBsend()*** call initiates block-oriented data transfer and the ***M7PBKUSend()*** call initiates an uncoordinated data transfer. After data transfer is finished, the sending task is notified with the message M7MSG_PBK_DONE.

Note

All of the data is transferred to the remote station.

The calls ***M7PBKBsend()*** and respectively ***M7PBKUSend()*** are only successful if a corresponding BRCV/URVC block or ***M7PBKBrcv()/M7PBKURcv()*** call is waiting on the remote side. Each BRCV/URCV block or ***M7PBKBrcv()/M7PBKURcv()*** call causes the entire data area to be received.

Receiving Data on an M7-CPU

The following call is used on an M7-CPU for receiving the data:

M7PBKBrcv()

M7PBKURcv()

The successful transfer of all the data to the destination data area is notified to the receiver task with the message M7MSG_PBK_NDR.

Rule

If both send and receive requests are to be executed in your user program, the receive requests must be called first, before the send requests.

8.10 Programming Bidirectional Communication Functions for Configured Connections

Block-oriented Send with *M7PBKBsend()*

Proceed as follows if you want your C program to send data to a remote station using the bidirectional communication functions ***M7PBKBsend()*** and ***M7PBKBrcv()***:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter pConnID points to a valid application link ID.
3. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)
4. Now decide whether the data to be transferred should sent from the program's own data area or in a data area of the S7 object server.

   In the data area of the sender task: create a buffer ***SrcVar[nLength]*** (formal data type ***M7VARADDR***) for the data to be sent in the global data area or on the heap. Initialize the buffer with the data to be sent.

   In the data area of the S7 object server: specify the variables of the S7 object whose contents you want to transfer. Create a variable ***ScrVar*** of type ***M7VARADDR*** in the global data area or on the heap. Initialize this variable with the address information of the items to be transferred.
5. Start data transfer to the remote station using the following M7-API call:

   M7PBKBsend(flags,ConnID,R_ID,pSrcVar,nLength,pCommFRB,MPrio)

   The parameter ***flags*** is used to specify whether the data to be transferred is contained in the data area of the sender task (***flags=A_USER***) or in the data area of the S7 object server (***flags=A_ZERO_FLAG***).

   You must also specify the application link ID ***ConnID***, the ID ***R_ID*** of the remote receiver block and/or receive call and the starting address ***pSrcVar*** and the length ***nLength*** of the buffer.

   If ***flags*** = A_USER, the contents of the buffer are transferred. If ***flags*** = ***A_ZERO_FLAG***, the call determines the address of the item in the S7 object server from the address variable in the buffer and then transfers it.

   You must also specify the address of the associated FRB ***pCommFRB***. The parameter ***Mprio s***pecifies the required priority of the returned message.

6. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

7. Evaluate the received message. The system notifies the end of data transfer to the task with the message with the ID M7MSG_PBK_DONE.

   If you have requested several transfers for configured connections, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer and/or an error in the asynchronous part of the send request can be determined with the following macro:

   M7GetCommStatus((M7COMMFRB)pMessageParam)

8. Repeat the steps 4 to 7 (loop in your C program) if you want to send the source data cyclically to the communication partner.

9. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Block-oriented Receive with *M7PBKBrcv()*

Proceed as follows if you want your C program to receive data from a remote station using the bidirectional communication functions ***M7PBKBsend()*** and ***M7PBKBrcv()***:

1. Configure a connection to the communication partner using the SIMATIC Manager.

2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.

3. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)

4. Now decide whether the data to be transferred should be received to the program's own data area or in a data area of the S7 object server.

   In the data area of the receiver task: create a buffer ***DstVar[nLength]*** (formal data type ***M7VARADDR***) for the data to be received in the global data area or on the heap.

   In the data area of the S7 object server: specify the area in the S7 object in which you want to receive the date. Create a variable ***DstVar*** of type ***M7VARADDR*** in the global data area or on the heap. Initialize this variable with the address information of the area in the S7 object.

5. Start data transfer from the remote station using the following M7-API call:

   M7PBKBrcv(flags,ConnID,R_ID,pDstVar,nLength,pCommFRB,MPrio)

   The call starts to receive data from the remote station via the application link ***ConnID***.

   You must specify the receiver ID ***R_ID***. The call only accepts data from the sending station if the ID is the same as the receiver ID specified in the send request.

   The parameter ***flags*** is used to specify whether the data to be transferred should be copied to the data area of the sender task (***flags=A_USER***) or to the data area of the S7 object servers (***flags=A_ZERO_FLAG***).

   The pointers ***pDstVar*** and ***nLength*** specify the starting address and length of the buffer in which the received data should be written (***flags=A_USER***), and/or the address information of the S7 object variable (***flags=A_ZERO_FLAG***).

6. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

7. Evaluate the received message. The system notifies the end of data transfer to the task with the message M7MSG_PBK_NDR.

   If you have requested several transfers for configured connections, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer and/or an error in the asynchronous part of the send request can be determined with the following macro:

   M7GetCommStatus((M7COMMFRB)pMessageParam)

   The number of bytes received can be determined with the following macro:

   M7GetCommRcvLen((M7COMMFRB)pMessageParam)

8. Repeat the steps 4 to 7 (loop in your C program) if you want to receive the source data cyclically from the communication partner.

9. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Uncoordinated Send with *M7PBKUSend()*

Proceed as follows if you want your C program to send data to a remote station using the bidirectional communication functions ***M7PBKUSend()*** and ***M7PBKURcv()***:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter pConnID points to a valid application link ID.
3. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function:

   M7SetFRBTag(&CommFRB, Tag)

   The data is sent from data area of the sender task: create a buffer ***SrcVar[nLength]*** (formal data type ***M7VARADDR***) for the data to be sent in the global data area or on the heap. Initialize the buffer with the data to be sent.
4. Start data transfer to the remote station using the following M7-API call:

   M7PBKUSend(flags,ConnID,R_ID,nVars,pSrcVar,pCommFRB,MPrio)

   The parameter ***flags*** is set to ***A_ZERO_FLAG*** to specify that the data to be transferred is contained in the data area of the sender task.

   You must also specify the application link ID ***ConnID***, the ID ***R_ID*** of the remote receiver block and/or receive call and the starting address ***pSrcVar*** and the number ***nVars*** of data to be sent.

   You must also specify the address of the associated FRB ***pCommFRB***. The parameter ***Mprio s***pecifies the required priority of the returned message.
5. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)
6. Evaluate the received message. The system notifies the end of data transfer to the task with the message with the ID M7MSG_PBK_DONE.

   If you have requested several transfers for configured connections, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

The success or failure of the data transfer and/or an error in the asynchronous part of the send request can be determined with the following macro:

M7GetCommStatus((M7COMMFRB)pMessageParam)

7. Repeat the steps 4 to 7 (loop in your C program) if you want to send the source data cyclically to the communication partner.
8. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Uncoordinated Receive with *M7PBKURcv()*

Proceed as follows if you want your C program to receive data from a remote station using the bidirectional communication functions ***M7PBKUSend()*** and ***M7PBKURcv()***:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Create an FRB ***CommFRB*** of data type **COMMFRB** in your data area (global data or heap) for the control data. Tag the FRB and thus your communication request with the following function.

   M7SetFRBTag(&CommFRB, Tag)

   The data is received in the data area of the receiver task: create a buffer ***DstVar[nLength]*** (formal data type ***M7VARADDR***) for the data to be received in the global data area or on the heap.
4. Start data transfer from the remote station using the following M7-API call:

   M7PBKURcv(flags,ConnID,R_ID,nVars,pDstVar,pCommFRB,MPrio)

   The call starts to receive data from the remote station via the application link ***ConnID***.

   You must specify the receiver ID ***R_ID*** in parameter R_ID. The call only accepts data from the sending station if the ID is the same as the receiver ID specified in the send request.

   The parameter ***flags*** is set to ***A_ZERO_FLAG*** to specify that the data to be transferred should be copied to the data area of the receiver task.

   The pointers ***pDstVar*** and ***nLength*** specify the starting address and length of the buffer in which the received data should be written.
5. Wait for a message with the following call:

   RmReadMessage(..,pMessageParam)

6. Evaluate the received message. The system notifies the end of data transfer to the task with the message M7MSG_PBK_NDR.

   If you have requested several transfers for configured connections, the following call can be used to identify the request:

   M7GetFRBTag(pMessageParam)

   The success or failure of the data transfer and/or an error in the asynchronous part of the send request can be determined with the following macro:

   M7GetCommStatus((M7COMMFRB)pMessageParam)

   The number of bytes received can be determined with the following macro:

   M7GetCommRcvLen((M7COMMFRB)pMessageParam)

7. Repeat the steps 4 to 7 (loop in your C program) if you want to send the source data cyclically to the communication partner.
8. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Cancelling Running Requests

A running ***M7PBKBrcv()***, ***M7PBKBSend()*** or ***M7PBKURcv()***job can be cancelled with the following call:

M7PBKCancel(ConnID,pCommFRB)

To identify the request, you must specify the application link ID ***ConnID*** and the address ***pCommFRB*** of the associated FRB.

8.11 Inquiry and Control Functions for Configured Connections

The inquiry and control functions for configured connections allow you to get information on the current status of a remote communication partner and/or to control program execution in the remote partner (for example request a transition to STOP mode).

As with the unidirectional communication functions for configured connections, with the inquiry and control functions the required operation on the remote station (server) is implemented by the operating system without needing explicit programming.

Overview

The M7-API provides the following inquiry and control functions:

- ***M7PBKStatus()***: check the current status of the remote station
- ***M7PBKStop()***: request a transition to STOP mode (stop all user programs) on the remote station
- ***M7PBKStart()***: send a cold restart request to the remote station
- ***M7PBKResume()***: send a warm restart (**RESUME)** request to the remote station.

Table 8-6 shows the possibilities which are available to change the current operating mode of the remote S7/M7-CPU using calls for configured connections.

Table 8-6 Changing the Operating Modes

Current operating mode	You can change the current operating mode with:		
	COLD RESTART	WARM RESTART (RESUME)	STOP
HALT			X
STOP	X	X	
STARTUP			X
RUN			X

Note

In contrast to an S7-CPU, an M7 automation computer (programmable controller) only supports cold restart. A RESUME request is rejected with an error message.

Interrogate Status of a Remote Station

Proceed as follows to get information on the current status of a remote communication partner:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Get the status with the following M7-API call:

   M7PBKStatus(ConnID,pBuffer,nBufsiz,pnBytes)

   The call requests status information from the remote station specified by the application link ID ***ConnID***.

   You must specify the starting address ***pBuffe***r and the length ***nBufSiz*** of the buffer into which the received status information should be copied.

   The call returns the actual number of bytes copied in the parameter ***pnBytes***.

Note

The structure and significance of the status information is described in the Reference Manual “System Software for S7-300/400 System and Standard Functions”.

4. If required, repeat the step 2 (loop in your C program) to continuously update the status information in your buffer.
5. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

Note

The M7-API also supports “unrequested notification of device status”. However, this function is not initiated by a call in the user program (for example ***M7PBKStatus()***), but is implemented within the framework of diagnostic messages (see section 5.19).

Changing the Operating Mode of a Remote Station

Proceed as follows if you want to request a change of the current operating mode of the remote communication partner to START, RESUME, or STOP mode:

1. Configure a connection to the communication partner using the SIMATIC Manager.
2. Establish an application link to the remote communication partner with the following call:

   M7KInitiate(pConnID,pHostAddr)

   If the application link was established, the parameter ***pConnID*** points to a valid application link ID.
3. Start the appropriate request with one of the following M7-API calls:

   M7PBKStart(ConnID) (request for START), or

   M7PBKStop(ConnID) (request for STOP), or

   M7PBKResume(ConnID) (request for RESUME).

   Each call must be specified with a valid application link ID ***ConnID*** for the remote station.

Note

In contrast to an S7-CPU, an M7 automation computer (programmable controller) only supports cold restart. A RESUME request is rejected with an error message.

4. When the application link to the remote station is no longer required, you can close it again with the following call:

   M7KAbort(ConnID)

   You must specify the ID ***ConnID*** of the application link to be closed.

8.12 Communication via Sockets

The socket interface in M7-SYS RT V4.0 offers function calls for communication in subnetworks of the Industrial Ethernet type via TCP/IP protocols. It is a prerequisite that a CP 1401 communication processor is installed in the M7-300/400. This facilitates communication in homogeneous S7 systems as well as heterogeneously with any other nodes in the Industrial Ethernet subnetwork.

The socket interface forms a separate library. For the function prototypes, the header file **SOCKET.H** from the M7 RMOS32 programs must be included. The RMFSK2IB.LIB library also has to be linked in.

You must configure the network using STEP 7.

The following sections provide an overview of the features and characteristics of the socket interface under M7-SYS V4.0.

Further Information

You will find instructions for programming with the socket interface in the manuals that are listed in the “Bibliography”. The socket calls are described in Chapter 2 of the reference manual “Standard Functions and System Functions”, Volume II.

Services and Protocols

The socket interface provides access to the following TCP/IP protocols:

TCP (Transport Control Protocol)
TCP is a connection-oriented protocol. It offers error-free data transmission, i.e. the transmitted data is received in the same sequence in which it is sent; otherwise an error is reported.

UDP (User Datagram Protocol)
The datagram service UDP is connectionless and does not provide error-free data transfer. Programs that use UDP must ensure that data is received correctly.

The data is transmitted as a stream of bytes with both protocols.

Socket Types

Sockets are the termination points of communication at which functions such as sending and receiving can be executed. Sockets are created by the communicating nodes and individual names can be assigned to them. Sockets always exist within a domain that supports specific protocols and a specific address family.

Sockets have types for the purpose of classifying their communication characteristics for the user. It is only possible for communication tasks to use sockets of the same type.

The following socket types are available

- SOCK_STREAM
  A stream socket supports bidirectional, reliable, sequential and non-duplicated data flow. A stream socket requires an error-free connection. When a connection has been created with the help of a stream socket, information can flow from one point to another point of the link in a data stream.
- SOCK_DGRAM
  A datagram socket supports bidirectional data flow, but it cannot be guaranteed that it is reliable, sequential and non-duplicating. This means that a task that receives messages via a datagram socket can receive these messages in duplicate and in a different sequence from the one in which they were sent.

Addresses

M7-SYS RT V4.0 supports sockets of the Internet domains. The address family of these sockets is always AF_INET. A name can be assigned to a socket. The name comprises the following components:

- Identifier for the address family (AF_INET)
- Port number identifier for a communication service or an application
- Internet (IP) address

  The IP address comprises 32 bits (4 octets separated by points) e.g. 142.120.12.44

The HOSTS File

The file \ETC\HOSTS on the boot drive of the M7-300/400 is used for administration of the communication nodes. The file is in ASCII format. One line corresponds to one node entry and is in the following form:

```
<Address> <Hostname> <Alias> <Comment>
```

The individual fields of an entry are separated by blanks or TABs and have the following meaning:

`<Address>`	IP address
`<Hostname>`	Name of the node
`<Alias>`	Optional, one or more aliases for the node
`<Comment>`	Optional, begins with a “#” character and is applicable as far as the end of the line

Example entry for a node called “M7alpha”

```
120.142.5.12       M7alpha      M7a12    #in Nuremberg
```

In this file, you can enter all nodes that you want to access via the socket interface. The socket interface provides calls for opening the HOSTS file and for querying entries.

You can also modify the HOSTS file in the SIMATIC Manager with the menu command **PLC Manage M7 System**, under the “Configure Operating System” tab.

The SERVICES File

The file \ETC\SERVICES on the boot drive of the M7-300/400 is used for administration of the communication service providers. Fixed port numbers are assigned to communication services or servers in the SERVICES file. The file is in ASCII format. One line corresponds to one node entry and is in the following form:

```
<Service> <PortNo/Protocol> <Alias> <Comment>
```

The individual fields of an entry are separated by blanks or TABs and have the following meaning:

`<Service>` Name of service

`<PortNo/Protocol>`
Port number and protocol for the service, separated by an oblique character (/)

`<Alias>` Optional, one or more aliases for the service

`<Comment>` Optional, begins with a “#” character and is applicable as far as the end of the line

Example entry for a service called “NeWS”

```
NeWS          144/tcp       news          #Window System
```

In this file, you can enter all communication services that you want to access via the socket interface. The socket interface provides calls for opening the SERVICES file and for querying entries.

You can also modify the SERVICES file in the SIMATIC Manager with the menu command **PLC Manage M7 System**, under the “Configure Operating System” tab.

Connection-Oriented Communication

The program layout for connection-oriented communication is shown schematically in the following diagram.

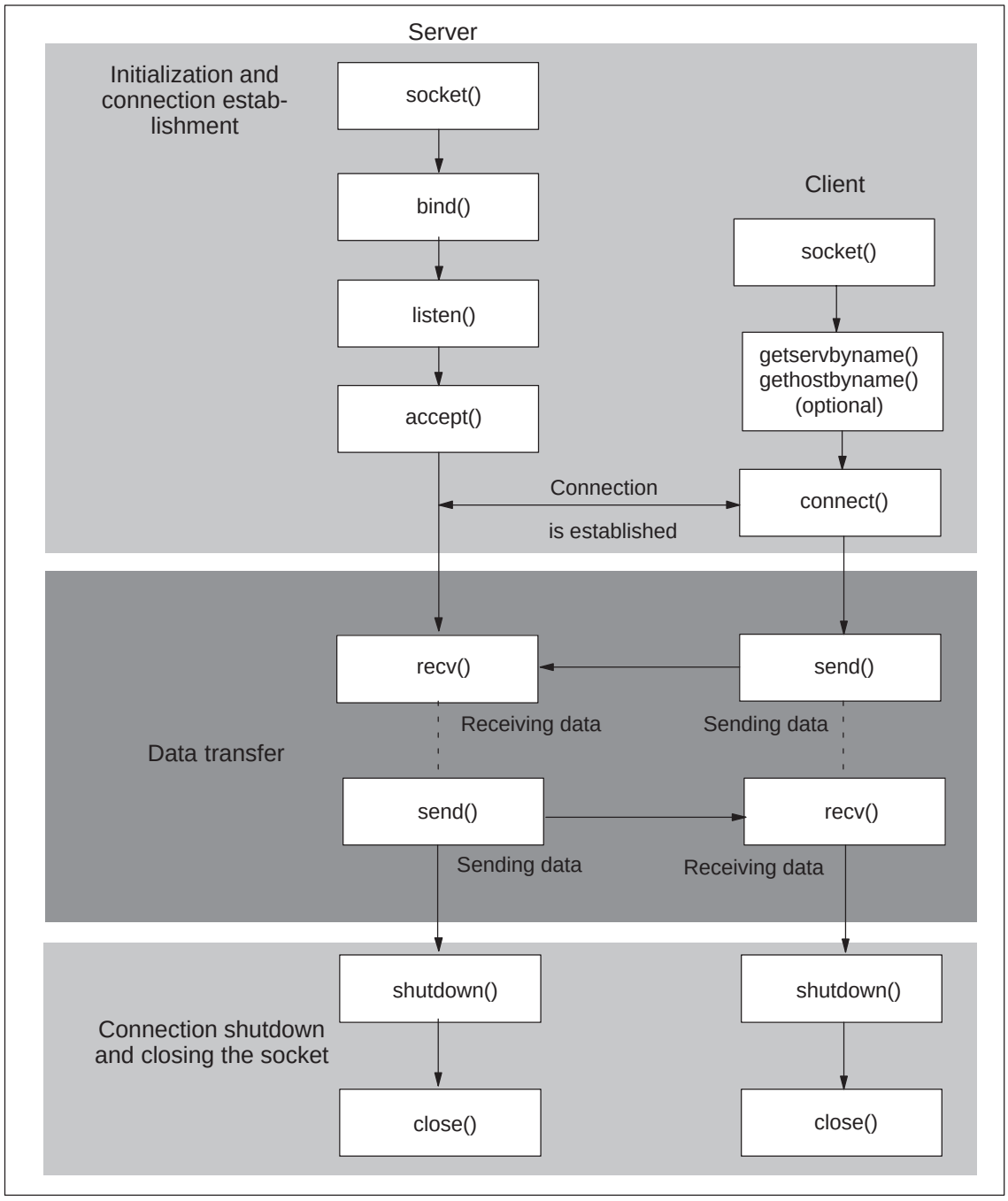


Figure 8-7 Connection-Oriented Communication – Program Layout

Connectionless Communication

The program layout for connectionless communication is shown schematically in the following diagram.

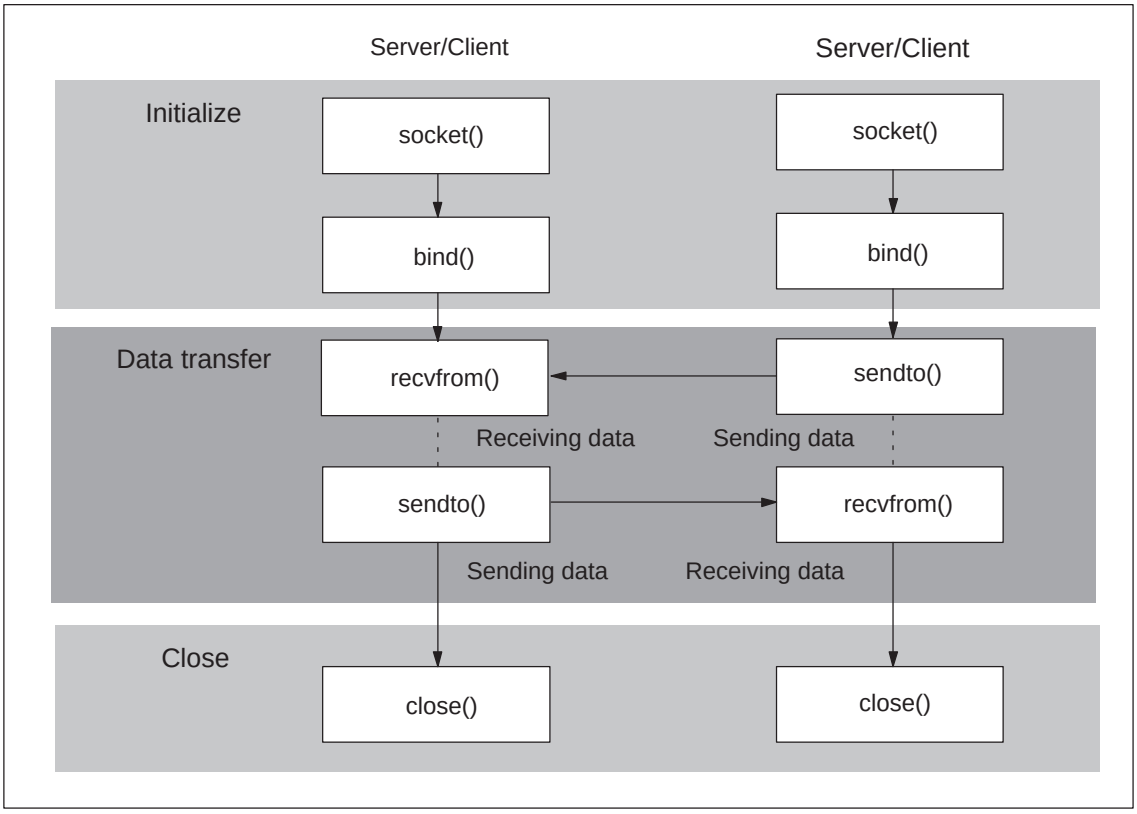


Figure 8-8 Connectionless Communication – Program Layout

Blocking and Non-Blocking Mode

The communication functions can be implemented in blocking or non-blocking mode. The mode can be specified for a socket with the help of the C runtime function ***ioctl()***.

- In **blocking** mode, a call waits until either the function can be executed completely or an error occurs. The blocking call ***connect()***, for example, only returns when the connection has been established or when an error has occurred.
- In **non-blocking** mode, a call is either executed immediately or it returns and sets the external variable *errno* = EWOULDBLOCK. Then the task must issue the call ***nselect()*** in order to wait for events arriving at the socket. When the expected event occurs, the original call can be repeated.

  The non-blocking call ***recv()***, for example, returns with *errno* = EWOULDBLOCK immediately when no data is present in the receive buffer. The call ***nselect()*** must be used to wait for the READ_NOTIFY event. Once the expected event has occurred, the ***recv()*** call can be repeated.

Urgent Data

Urgent data (out-of-band data) can be sent and received using sockets of the SOCK_STREAM type. The following options must be used in this case:

- *flags*=MSG_OOB in calls ***send()***, ***sendto()***, and
- *flags*=MSG_URGENT in calls ***recv()*** and ***recvfrom()***

Options

The following options are supported for sockets:

Call	Option	Meaning
ioctl()	SLiocNBIO	Switch blocking mode on and off
setsockopt()	SO_REUSEADDR	Permit a local address to be assigned more than once using ***bind()***.
	SO_KEEPALIVE	Test that the connection is still available at fixed intervals.

Bibliography

On the subjects of TCP/IP and socket programming, we recommend the following literature:

1. Comer, D. E., *Internetworking with TCP/IP Bd.1 Principles, protocols, and architecture* . Prentice-Hall 1991 ISBN 0-13-474321-0; 0-13-474321-0
2. Comer, D. E.; Stevens, D. L., *Internetworking with TCP/IP Bd.2 Design, implementation and internals*
   Prentice-Hall 1991 ISBN 0-13-465378-5; 0-13-472242-6
3. Comer, D. E.; Stevens, D. L., *Internetworking with TCP/IP Bd.3 Client-server programming and applications. BSD socket version*
   Prentice-Hall 1996 ISBN 0-13-262148-7
4. Feit, S., *TCP/IP Achitecture, Protocols and Implementation*
   McGraw-Hill 1993 ISBN 0-07-020346-6
5. Martin, J.; Leben, J., *TCP/IP Networking. Architecture, Administration and Programming* Prentice Hall 1994 IS: ISBN 0-13-642232-2
6. Siyan, K. S., *Inside TCP/IP. A Comprehensive Indroduction to Protocols and Concepts* New Riders 1997 ISBN 1-56205-714-6
7. Furrer, F. J., *Ethernet-TCP/IP fuer die Industrieautomation. Grundlagen und Praxis* (User Manual) Huethig 1998 ISBN 3-7785-2641-3

Loadable Drivers

9

Chapter Overview

9.1 Overview

M7 RMOS32 provides a selection of various device drivers (referred to simply as drivers) for programming hardware components such as serial interfaces.

You must load the drivers you want to use for your application before the actual communication call, hence the name “loadable drivers”. Only those drivers which are actually required have to be loaded into the work memory of the module.

Basic Terms

The terms **device** and **unit** appear often in the following paragraphs. The device is the driver for a hardware component (e.g. SER8250 for serial ports) and “unit” is the name given to a specific hardware component (such as the serial port COM1). Both resources are entered in the resource catalog when the driver is loaded or the unit is generated.

The phrase “generating a unit” in this context always means that a “management unit” for a hardware component is created in a driver, and is recorded in the resource catalog. The unit (name) is always required for access to the hardware component.

Which Loadable Drivers Are Available?

From Version V2.0 of the system software for M7-300/400, the following loadable drivers are available:

- 3964 driver – for accessing serial ports via 3964 protocol
- SER8250 driver – for simple access to serial ports.

Information about RMOS API Calls

Detailed information about RMOS API calls which address loadable drivers (e.g. for reading a character over a serial interface), and about the required data structures and libraries, can be found in the Reference Manual “System Software for M7-300/400, Standard and System Functions”.

This chapter only makes qualitative reference to the functions and does not provide detailed description of parameters.

Writing Your Own Driver

For information on how to design and write your own loadable drivers please refer to the electronic manual “System Software for M7-300/400, Writing Loadable Drivers”. You can open this manual on your programming device by calling **Start ▸ Simatic ▸ S7 Manuals ▸ M7-SYS RT Loadable Device Drivers** from the task bar.

9.2 Loading a Driver and Generating a Unit

In the user program, you can load a driver with the following call:

RmLoadDevice(pDeviceName,pArguments)

Described below are the three options allowed by this call:

- Load the driver only
- Load the driver and simultaneously generate a unit of the driver
- Generate a unit (when the driver is loaded)

Requirements

The library RMFCRIFB.LIB is required when the application is linked.

Loading a Driver

If you only want to load one driver and the driver is not yet entered in the resource catalog, the parameters should be assigned as follows: ***pDeviceName*** is the **path name** for the driver, ***pArguments*** is a NULL pointer. Example:

RmLoadDevice("\M7RMOS32\3964.drv", NULL)

The call causes the 3964 driver to be loaded. The driver is entered in the resource catalog under the name "3964".

Loading a Driver and Generating a Unit

If you want to load a driver and simultaneously generate a unit (and no entry exists for either in the resource catalog), the parameters should be assigned as follows: ***pDeviceName*** is the **path name** for the driver, ***pArguments*** is a string with parameters for the unit. The string always begins with the unit name. Example:

RmLoadDevice("\M7RMOS32\SER8250.drv", "COM_1 IRQ:4 BASE:0x3F8")

The call causes the driver to be loaded and entered in the resource catalog under the name SER8250. A unit named COM_1 is also generated with the specified parameter values for the IRQ number of the interface or the I/O base address. Default settings are used for further communication parameters, such as the transmission rate.

Generating a Unit

If the driver is already loaded and entered in the resource catalog, you can generate further units. The parameters should be assigned as follows: ***pDeviceName*** is the **name** of the driver, as entered in the resource catalog; ***pArguments*** is a string with configuration parameters for the unit. The string always begins with the unit name, followed by the individual parameters separated by spaces. Example:

RmLoadDevice("SER8250", "COM_1 IRQ:4 BASE:0x3F8 MODE:19200-n-8-1 BUFFER:512")

The call causes a unit named "COM_1" to be generated with the specified parameters and entered in the resource catalog.

Configuring a Unit

When a unit is generated, the interface is configured (***pArguments*** parameter).These configuration parameters depend on the driver used.

If you want to reconfigure the interface, the following call is available:

RmIOControl(...,Control,pBuffer,..)

Control function **RM_IOCTL_INIT_ASCII** or **RM_IOCTL_INIT** must be used for the ***Control*** parameter. In the case of control function RM_IOCTL_INIT_ASCII ***pBuffer*** points to an array of pointers which point to the configuration parameters (ASCII strings) for the interface. In the case of control function RM_IOCTL_INIT, ***pBuffer*** points to a structure that depends on the type of driver.

Note

All driver-specific parameters are described in the reference manual under the ***RmIOControl*** call, control function RM_IOCTL_INIT_ASCII of the relevant driver.

Further Options

Outside the user program, you have the option of loading drivers and generating units:

- Using the CLI (command line interpreter), during normal operation
- By means of entries in the RMOS.INI file, on system power-up.

You will find detailed information on the above in the User Manual entitled “System Software for M7-300/400, Installation and Operation”.

9.3 Communication with Loadable Drivers

When the driver is loaded, and a unit with suitable configuration parameters is generated, tasks can send read and write requests to this unit. The following function calls are available:

- 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 basic sequence of RMOS API calls required is described below; you will find more details of the relevant call in the Reference Manual.

Note

The ***RmIO*** calls described below are valid for all loadable drivers. Special control functions for the ***RmIOContro***l call, and special calls provided for individual drivers in the function libraries are described in the section on the relevant driver.

Sending to / Receiving from

In order to send characters to or receive characters from a unit, proceed as follows:

1. Set up sufficient buffers for send and receive data in the memory.
2. Fill the send buffer with the data to be transmitted.
3. Open the corresponding unit with the call

   RmIOOpen(pUnitName,Mode,pHandle)

   pUnitName is the name of the unit in the RMOS resource catalog, as defined when you generated the unit. ***Mode*** specifies whether the unit is to be opened for read or write access; use the value RM_IO_WRITE for sending or RM_IO_READ for receiving.

   If you want to open the unit for sending **and** receiving, you must combine the values for the mode with an OR operation; that is RM_IO_READ | RM_IO_WRITE.

   The call returns a descriptor ***Handle*** to which the pointer ***pHandle*** points. The descriptor is required for subsequent read or write access.

4. Read from the unit with the call

   RmIORead(Wait,FlagMask,Handle,Length,pBuffer,...)

   Wait specifies whether (=RM_WAIT) or not (=RM_CONTINUE) program execution waits until the read task is finished. If the program does not wait, that is if the read access takes place asynchronously to program execution, ***FlagMask*** (a bit mask in the local flag group) indicates the end of the read access.

   Handle is the descriptor for the unit. ***Length*** indicates the number of bytes in the memory area identified by ***pBuffer*** to be read in over the serial interface.

5. Write to the unit with the call

   RmIOWrite(Wait,FlagMask,Handle,Length,pBuffer,...)

   Wait specifies whether (=RM_WAIT) or not (=RM_CONTINUE) program execution waits until the write task is finished. If the program does not wait, that is if the write access takes place asynchronously to program execution, ***FlagMask*** (a bit mask in the local flag group) indicates the end of the write access.

   Handle is the descriptor for the unit. ***Length*** indicates the number of bytes to be sent via serial interface from the memory area identified by ***pBuffer***.

6. Close the unit with the call

   RmIOClose(Handle)

Note

Only 3964 Driver:

If a program both writes to and reads from a unit, the read request must be always programmed before the the write request. When data arrives and no read request exists, the incoming data are discarded.

Summary

All loaded drivers should be handled as follows:

- Generate unit: a “management unit” is generated in the driver, and the name of the unit (e.g. serial interface) is defined in the resource catalog.
- The name of the unit is required in order to open the unit for send and receive tasks (***RmIOOpen()*** call).
- The ***RmIOOpen()*** call returns a descriptor **Handle**.
- The **Handle** descriptor identifies the unit for the subsequent send or receive tasks. The data area where the data to be transmitted or received are stored must always be specified with ***pBuffer***.

9.4 Driver 3964

Driver 3964 is a character-oriented driver, sometimes referred to as a character device driver. The name is derived from the 3964(R) procedure. This procedure controls data transfer via a point-to-point connection between two communication partners. The 3964(R) procedure contains the security layer (layer 2), in addition to the bit transfer layer (layer 1).

The 3964(R) procedure appends control characters to the user data during data transfer (security layer). These control characters enable the communication partners to check whether arriving data are complete and free of errors.

Procedure 3964(R) evaluates the following control characters:

- STX Start of Text
- DLE Data Link Escape
- ETX End of Text
- BCC Block Check Character (3964R only)
- NAK Negative Acknowledge

Please refer to the documentation of the device you want to connect via the loadable 3964(R) driver (e.g. CP with serial interface) for detailed information about how the 3964 procedure functions.

Information about the Protocol

Mode: Half-duplex (data are either sent or received at one time).

Data transfer: Asynchronous (start and stop bit required for transfer of the characters); code-transparent (any sequence of characters allowed in the user data)**.**

Control Functions for *RmIOControl*

The following call is available in addition to the ***RmIO*** calls described in Section 9.3:

RmIOControl(Wait,FlagMask,Handle,Control,pBuffer,..)

This call allows you to call up the function code of a control function, specified by ***Control***. The meaning of ***Wait***, ***FlagMask*** and ***Handle*** is similar to the calls described in Section 9.3. ***pBuffer*** points to a parameter block for the control function.

You can use the control functions for:

- Initializing/configuring the unit
- Request control

Configuring the Unit

The following table provides a brief description of the control functions available for initializing and configuring the unit.

Table 9-1 Control Functions for Initializing/Configuring the Unit

Control Function for Control	Meaning
RM_IOCTL_INIT_ASCII	Configure unit, parameters are stored as ASCII strings
RM_IOCTL_INIT	Configure unit, parameters are stored in structure (structure type ***Rm3964InitStruct***)
RM_IOCTL_MODE	Configure unit with individual communication parameters (parameters are stored in structure of type ***RmIOCTLModeSerialStruct***)
RM_IOCTL_INIT_GET	Read in unit configuration in structure of type ***Rm3964InitStruct***

Request Control

The following table provides a brief description of the control functions available for request control.

Table 9-2 Control Functions for Request Control

Control Function for Control	Meaning
RM_IOCTL_CANCEL	Cancel current send/receive request
RM_IOCTL_RESET	Reset unit and restart (unit must subsequently be reconfigured)
RM_IOCTL_RESERVE	Reserve unit for calling task. If other tasks issue requests to the “reserved” unit, they are not executed until the unit is released (***RM_IOCTL_RELEASE***)
RM_IOCTL_RELEASE	Release the unit
RM_IOCTL_GET_PROPERTIES	Get the function scope of the driver. The result is stored in a structure of type ***RmIOCTLPropertiesStruct***
RM_IOCTL_GET_VERSION	Get the version of the driver. The result is stored in a structure of type ***RmIOCTLVersionStruct***

9.5 Driver SER8250

Driver SER8250 is a character-oriented driver, sometimes referred to as a character device driver, for simple access to the serial interface (without using a protocol). The driver supports both bytewise communication and the transfer of character strings.

Control Functions for *RmIOControl*

The following call is available in addition to the ***RmIO*** calls described in Section 9.3:

RmIOControl(Wait,FlagMask,Handle,Control,pBuffer,..)

This call allows you to call up the function code of a control function, specified by ***Control***. The meaning of ***Wait***, ***FlagMask*** and ***Handle*** is similar to the calls described in Section 9.3. ***pBuffer*** points to a parameter block for the control function.

You can use the control functions for:

- Initializing/configuring the unit
- Request control
- Flow control
- Handling the “background buffer”

Configuring the Unit

The following table provides a brief description of the control functions available for initializing and configuring the unit.

Table 9-3 Control Functions for Initializing/Configuring the Unit

Control Function for Control	Meaning
RM_IOCTL_INIT_ASCII	Configure unit, parameters are stored as ASCII strings
RM_IOCTL_INIT	Configure unit, parameters are stored in structure (structure type ***Ser8250InitStruct***)
RM_IOCTL_MODE	Configure unit with individual communication parameters (parameters are stored in structure of type ***RmIOCTLModeSerialStruct***)
RM_IOCTL_INIT_GET	Read in unit configuration in structure of type ***Ser8250InitStruct***

Request Control

The following table provides a brief description of the control functions available for request control.

Table 9-4 Control Functions for Request Control

Control Function for Control	Meaning
RM_IOCTL_CANCEL	Cancel current send/receive request
RM_IOCTL_RESET	Reset unit and restart (unit must subsequently be reconfigured)
RM_IOCTL_RESERVE	Reserve unit for calling task. If other tasks issue requests to the “reserved” unit, they are not executed until the unit is released (***RM_IOCTL_RELEASE***)
RM_IOCTL_RELEASE	Release the unit
RM_IOCTL_GET_PROPERTIES	Get the function scope of the driver. The result is stored in a structure of type ***RmIOCTLPropertiesStruct***
RM_IOCTL_GET_VERSION	Get the version of the driver. The result is stored in a structure of type ***RmIOCTLVersionStruct***

Flow Control

The following table provides a brief description of the control functions available for flow control.

Table 9-5 Control Functions for Flow Control

Control Function for Control	Meaning
RM_IOCTL_READSTOP and further control functions for defining the number of characters, stop character, and wait time	Define the end condition for read requests: end character, number of characters read or expiry of a defined wait time
RM_IOCTL_READSTOP_GET	Read the end condition defined by ***RM_IOCTL_READSTOP***
RM_IOCTL_WRITESTOP and further control functions for defining the stop character and character delay time	Define the end condition for write requests: end character or expiry of a defined character delay time
RM_IOCTL_WRITESTOP_GET	Read the end condition defined by ***RM_IOCTL_WRITESTOP***

Handling of the “Background Buffer”

The “background buffer” is a memory area used by driver SER8250 for the temporary storage of data.

The following table lists the control functions available for handling the background buffer.

Table 9-6 Control Functions for the Background Buffer

Control Function for Control	Meaning
RM_IOCTL_BUFFER_SETSIZE	Set the size of the background buffer
RM_IOCTL_BUFFER_GETSIZE	Get the size of the background buffer
RM_IOCTL_BUFFER_USED	Get the number of characters in the background buffer
RM_IOCTL_BUFFER_FLUSH	Flush the background buffer

Runtime Library for SER8250

With the loadable SER8250 driver, you have the option of using additional or alternative calls by linking the library RMFSERB.LIB. The **header file serial.h** should be included for these calls.

The library is based on the ***RmIO*** calls (see Section 9.3); however the control functions are implemented differently (***SerialInit*** and ***SerialInitEx*** for initialization of the unit).

Please ensure that access to the unit is only “reserved” for one task at a time. If several tasks access the unit, the accesses must be coordinated such that they take place successively.

The following table lists the library functions and provides references to similar ***RmIO*** calls.

Table 9-7 Library Functions for Driver SER8250

Library Function	Meaning
SerialInit	Initialize unit (transmission rate, parity, number of data bits, number of stop bits)
SerialInitEx	Initialize unit (transmission rate, parity, number of data bits, number of stop bits, size of background buffer, number of characters to be read for receive requests, declaration of start and stop characters and character delay time)
SerialOpen	Open unit (see ***RmIOOpen***)
SerialCose	Close unit (see ***RmIOClose***)

Table 9-7 Library Functions for Driver SER8250

Library Function	Meaning
SerialCheckChar	Read individual character from unit (do not wait for character) - if the background buffer is empty, the call is terminated with the error message “data not found” (see ***RmIORead***).
SerialCheckString	Read string from unit (do not wait for characters) - if the background buffer is empty, the call is terminated with the error message “data not found” (see ***RmIORead***).
SerialPutChar	Write individual character to unit (see ***RmIOWrite***)
SerialGetChar	Read individual character from unit (wait for character, see ***RmIORead***)
SerialPutString	Write string to unit (see ***RmIOWrite***)
SerialGetString	Read string from unit (wait for character, see ***RmIORead***)

Sending to/ Receiving from the Unit using Library Functions

In order to send to or receive characters from the unit (serial interface), proceed as follows:

1. Set up sufficient buffers for send and receive data in the memory.
2. Fill the send buffer with the data to be transmitted.
3. Open the corresponding unit with the call

   SerialOpen(pUnitName,pHandle)

   pUnitName is the name of the unit in the RMOS resource catalog, as defined when you generated the unit.

   The call returns a descriptor ***Handle*** to which the pointer ***pHandle*** points. The descriptor is required for subsequent read or write access.
4. Initialize the unit with the call

   SerialInit(Handle,Baud,Data,Parity,Stop) or

   SerialInitEx(Handle,Baud,Data,Parity,Stop,BufferSize,...)

   The call ***SerialInitEx()*** supports **extended** initialization of the unit, such that, in addition to the parameters for the transmission rate, number of data bits, parity check and number of stop bits, you can also specify the size of the background buffer, and further control character declarations, etc.
5. Write to the unit with the call

   SerialPutChar(Handle,Char) or

   SerialPutString(Handle,pString,MaxLen)

   SerialPutChar writes a single character ***Char*** over the serial interface; ***SerialPutString*** write a string of length ***MaxLen*** addressed by ***pString***.
6. Read from the unit with the call

   SerialGetChar(Handle,Char) or

   SerialGetString(Handle,MaxLen,pString,pCount)

   Both calls wait for the arrival of characters; the user program does not continue execution until the receive data have been read in from the background buffer of the unit.

   In order to terminate the read request immediately if the background buffer is empty, use the calls

   SerialCheckChar(Handle,Char) or

   SerialCheckString(Handle,MaxLen,pString,pCount)

   MaxLen is the maximum number of characters of a string to be read. ***pString*** is the address of a memory area where the characters are to be stored, and ***pCount*** points to a `ulong` containing the number of characters read.
7. Close the unit with the call

   SerialClose(Handle)

Note

In order to read data from the serial interface within a program please use:

- either the ***RmIOxxx*** RMOS API calls or
- the calls of the **RMFSERB.LIB** library

Design and Development of a User Program 10

Chapter Overview

10.1 Overview

This chapter uses a concrete example which solves a typical automation assignment to illustrate the individual functions of M7 RMOS32.

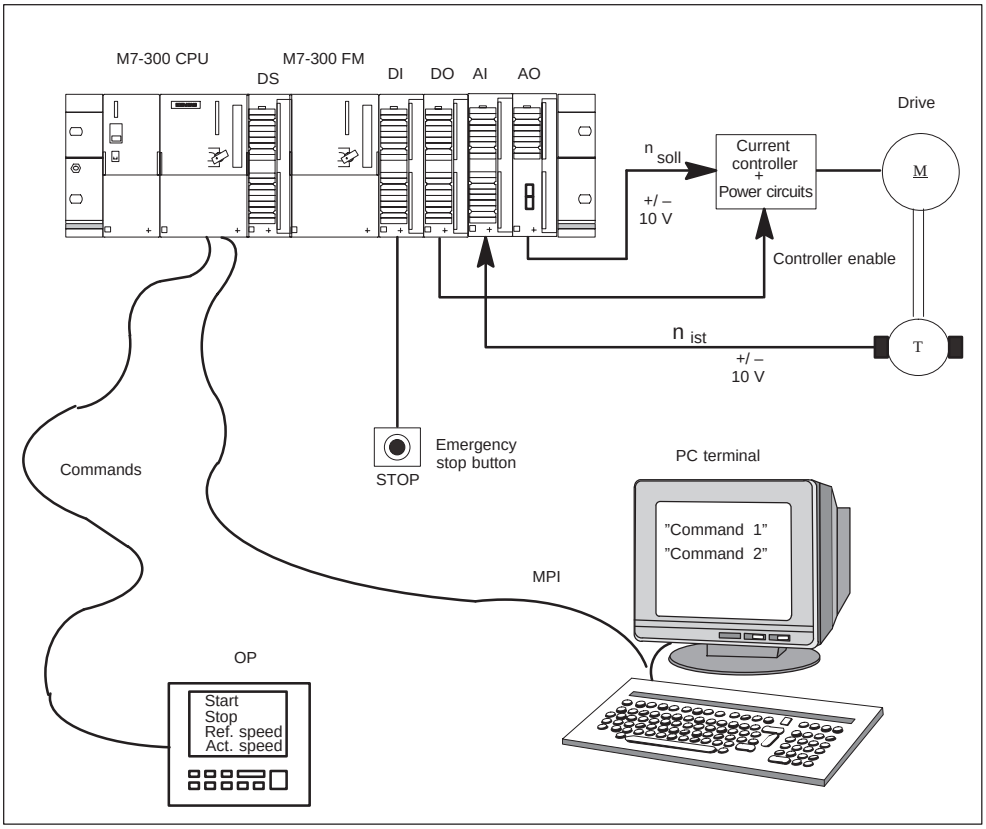


Figure 10-1 Hardware Configuration for the Example Automation Assignment

Hardware Configuration

The example automation assignment makes use of the following hardware modules:

- A CPU module to control and monitor the overall process. Apart from the FM, the CPU also needs to control other components in the plant. However, these other components are not shown in Figure 10-1 and will not be discussed further in our example.

  The CPU also controls the operator interface, which is implemented with an operator panel (OP).

  The CPU module can be either an S7 CPU or an M7 CPU. In the following example, both of these CPU module types will be discussed.

- An M7 FM to do the actual controlling of the plant components shown in Figure 10-1. It has access through its local bus segment to the required control and feedback signals from the process I/Os.

Process I/Os of the FM

To satisfy the hardware requirements of the example, the following signals are connected to the local process I/Os of the FM:

- One analog input to the signal module AI for interrogating the actual speed,
- One analog output from the signal module AO for the control signal to the current controller,
- One digital input to the signal module DI for the STOP button,
- One digital output from the signal module DO for the output of an enable signal for the current controller.

Automation Problem to be Solved in the Example

In the example, the following automation assignment needs to be solved (Figure 10-1):

- It is necessary to be able to switch on and switch off a drive M from an operator panel (OP) which is connected to the CPU module.
- It is necessary to control the rotary speed of the drive. The required speed is input on the operator panel. A tachometer T is provided to interrogate the actual speed.
- The switching on and off of the drive should also be controlled with an enable signal from a DS simulation module of the CPU (logical AND with the start/stop command).
- It is necessary to display the current speed on the OP.
- A STOP button is necessary for stopping the drive manually in an emergency.
- During the development and commissioning phase, a PC will be connected to the CPU module through an MPI interface to upload the program and to debug it. This PC will also be used to display RMOS consoles and to enter control parameters for the purpose of optimization.

What is Described in this Chapter?

This chapter uses the concrete assignment above to describe each of the steps of work. It also shows how each of the separate requirements of the assignment are converted to the corresponding flow diagrams for C programs.

10.2 Procedure for Program Development

Development Steps

When solving automation assignments, we recommend to start out by making decisions on technological aspects before converting each step into a separate program. Accordingly, we recommend the following procedure:

1. Determine the technological requirements which are placed on the CPU module. Determine which programming approach can be best used to solve the problems.
2. Determine which S7 objects for operator interface (OI) devices your program will need on the CPU module and list them according to type and extent if necessary.
3. Determine the technological requirements which are placed on the FM and the data exchange with the CPU module.
4. Make a list of all process data, alarms (including FM alarms for the CPU module), FM user data and configuration data.
5. Assign the process data, alarms and FM user data to their logical addresses.
6. Check whether your program on the M7 FM needs further S7 objects for OI devices in addition to the process data, and list them according to type and extent if necessary.
7. Determine the required communication mechanisms between the CPU module and the M7 FM.
8. Choose the reaction of your program to the operating modes STARTUP, STOP and HALT. Allow for messages in case of operating mode changes and/or interrogation of the operating mode.
9. Determine the logical structure of your C program. Decide on the best task structure for your program.
10. Determine the most suitable coordination and communication mechanisms between each of the tasks.
11. Choose the program sequence for each of the tasks.

You may find when refining your program design that you did not take account of all necessary facts in a previous step. In this case repeat one or more planning steps (iterative approach).

Software Components of the Automation Assignment

Figure 10-2 shows, from the programming viewpoint, the most important software components that you need to design an M7 RMOS32 program for a CPU module and an M7 FM.

The significance of each of the elements is described in previous chapters.

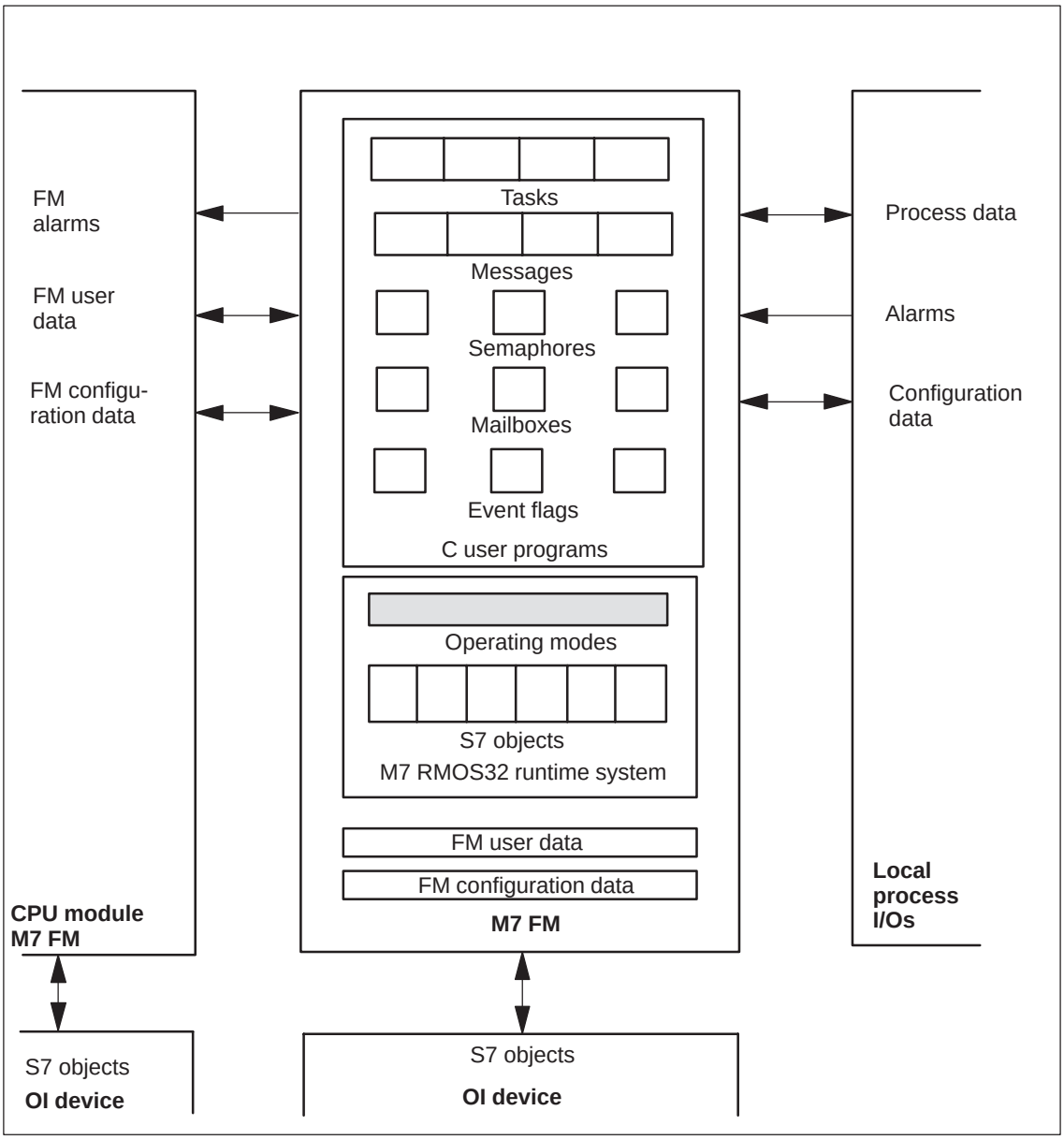


Figure 10-2 Software Components of an M7 RMOS32 Program

10.3 Determining the Technological Aspects of the Assignment

Sketch an Overview of the Overall System

In order to determine the technological aspects of the automation assignment, an overview diagram showing the hardware components of the subsystems can be very useful.

Initially, the diagram should only show components which are relevant for each of the subtasks to be implemented. In our example, this encompasses on the one hand the responsibilities of the FM, in other words its communication with the process I/Os and the overlying CPU, and the responsibilities of the CPU module.

The following description applies to both the M7 CPU and the S7 CPU. Later, when describing the responsibilities of the CPU, the corresponding S7 FC and/or S7 SFC calls are described alongside the M7 API calls.

Determine the Functional Sequence

Use the figure to make a note of each of the individual functions which will comprise the overall responsibilities of the FM and the CPU, including the associated data exchange.

Define the Responsibilities of the CPU

Determine what the CPU needs to do to help the FM to solve the overall assignment. This could include:

- Determining the initial data for the FM
- Evaluating result data from the FM
- Exchanging data with the FM
- Exchanging date with the OP

Define the Responsibilities of the FM

Make a detailed list of the FM's responsibilities, including the necessary communication with the CPU.

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Sketch an overview of the overall system

The overview diagram for the example program is shown in Figure 10-1.

Determine the functional sequence

- The drive is switched on and off from the OP
- The required rotary speed is entered
- The actual speed is determined
- The control current is calculated
- The control current is output
- The STOP button is interrogated and the drive is switched off if the STOP button is pressed

Define the responsibilities of the CPU:

- OP inputs are copied directly to an S7 data area (for example flag area)
- Interrogate the enable signal of the simulation module
- Transfer new values from the OP inputs to the FM
- Read actual speed from the FM and store it in a flag word for displaying on the OP

Define the responsibilities of the FM:

- Read and interpret data from the CPU (“required speed“, “enable“, “start/stop”)
- Interrogate STOP button
- Switch drive controller on and off
- Process required speed value for controller
- Read in actual speed as an analog value and process it
- Calculate controlling algorithm
- Output analog value for control current
- Notify status of drive to the CPU
- Notify actual speed to the CPU
- React to operating mode change to STOP or HALT

10.4 Summary of the Alarms and Process Data

Alarm Types

The following alarm types are available under M7 RMOS32:

- Process alarms

  These are triggered by alarm-enabled signal modules if either a digital input signal has changed or an analog input signal has moved outside an upper or lower limit.

- Diagnostic alarms:

  These are triggered by alarm-enabled signal modules if a fault occurs on the module, for example wire break.

Summarizing the Alarms

Proceed as follows when summarizing the alarms:

1. Create a list of all signal modules which are attached to the local process I/Os of the CPU (see table 10-1).
2. Create a list of all signal modules which are attached to the local process I/Os of the FM (see table 10-1).
3. Make a note in the list of whether the signal module is capable of generating an alarm (“alarm-enabled”) and note down the alarm type.
4. Decide whether the FM should send a process or diagnostics alarm to the CPU module and make a note of this in your design documentation.

Note

An alarm-enabled signal module can only set off a process or diagnostic alarm if it has been appropriately configured with the SIMATIC Manager (Basic Configuration).

Table 10-1 Example List of Module Names

Module name	Module type	Alarm type

Summarizing the Process Data

Extend the list of module names by adding lines for the signal names (see Table 10-2).

Table 10-2 Example List of Module and Signal Names

Module name	Module type	Alarm type
Signal name	**Signal type**	**Associated module**

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Summary of alarms and process data for the CPU:

Module name	Module type	Alarm type
DS	Digital input	no alarm
Signal name	**Signal type**	**Associated module**
FREIGABE	Digital input	DS

Summary of alarms and process data for the FMs:

Module name	Module type	Alarm type
DI	Digital input	no alarm
DO	Digital output	no alarm
AI	Analog input	no alarm
AO	Analog output	no alarm
Signal name	**Signal type**	**Associated module**
STOP_TASTER	Digital input	DI
FREI_AUS	Digital output	DO
IST_DREHZ	Analog input	AI
STELL_STR	Analog output	AO

10.5 Assigning the Logical Addresses

Starting Addresses of the Local Process I/Os

In the case of an M7, the SIMATIC Manager allows you to independently configure the logical starting address of the local process I/Os for the CPU (signal modules to the right of the CPU and to the left of the FM) and for the FM (signal modules to the right of the FM). However, the example program will use the default addressing for the M7 because most S7-300 CPU modules only support the default addressing of the System S7-300.

Starting Address of the FM

The FM can use any of the slots intended for the signal modules. In the example program, the FM has the **default starting address for analog modules**.

Address of the FM User Data

The FM user data is accessed with two different logical addresses:

- From the viewpoint of the CPU, the logical address of the FM user data corresponds to the default starting address of the FM for the chosen slot.
- From the viewpoint of the FM (access through the C user program), the logical address of the FM user data is **set to 240 with the default addressing.**

Logical Address of a Signal

The logical address of signals can be calculated as follows:

- Address of a binary signal:

  Module starting address + byte address of the signal byte within the module and the bit number of the signal within the signal byte.

- Address of an analog signal:

  Module starting address + byte address of the analog signal within the module.

In System S7-300, digital modules occupy 4 byte addresses by default and analog modules occupy 16 byte addresses by default.

Table of Assigned Starting Addresses

We recommend you to extend the list with the module and signal names to include the logical addresses of the modules and/or the signals. This will make it easier for you later on to design and test your program.

Table 10-3 Example List of Module and Signal Names and Logical Addresses

Module name	Module type	Alarm type	Logical starting address
Signal name	**Signal type**	**Associated module**	**Logical address**

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Slots for the modules:

- Slot DS: 4
- Slot FM: 5
- Slot DI: 6
- Slot DO: 7
- Slot AI: 8
- Slot AO: 9

Logical addresses::

The following logical addresses can be calculated from the slot assignment:

- Logical start address of the FM from the viewpoint of the CPU: 272
- Logical addresses of the signal modules:

  The logical default addresses of the DI module of the local process I/Os of the CPU and the associated process input signal are shown in the following table.

Module name	Module type	Alarm type	Logical starting address
DS	Digital input	no alarm	0
Signal name	**Signal type**	**Associated module**	**Logical address**
FREIGABE	Digital input	DS	0 (DI+0)

The logical default addresses of the signal modules of the local process I/Os of the FM are shown in the following table.

Module name	Module type	Alarm type	Logical starting address
DI	Digital input	no alarm	8
DO	Digital output	no alarm	12
AI	Analog input	no alarm	320
AO	Analog output	no alarm	336
Signal name	**Signal type**	**Associated module**	**Logical address**
STOP_TASTER	Digital input	DI	8 (DI+0)
FREI_AUS	Digital output	DO	12 (DO+0)
IST_DREHZ	Analog input	AI	320 (AI+0)
STELL_STR	Analog output	AO	336 (AO+0)

10.6 Data Exchange between the FM and the Overlying CPU Module

Is Communication Necessary?

First decide whether cooperation with the CPU module is in fact necessary, or whether the FM can operate autonomously. If cooperation is not necessary, you can skip the rest of this section.

Choose Communication Type

Decide which of the following possibilities is most appropriate for the cooperation in your program:

- The CPU module only sends data to the FM during STARTUP and the data is transferred after booting.
- Data exchange (in one or both directions) between the CPU and the FM **needs to be coordinated**.

  If P bus communication is used to transfer FM user data or FM data records, appropriate coordination mechanisms (handshake) must be explicitly implemented in your program code (see example program in section 6.2).

  The P bus system itself does not provide any special function call for coordinating data exchange.
- Data exchange (in one or both directions) between the CPU and the FM **does not need to be coordinated**.

  Uncoordinated data exchange does not require any special measures, and can take place either through the P bus (FM user data or FM data records) or through the communication bus.

Choose the Resources for the Data Exchange

The P bus allows data to be exchanged between the CPU and the FM using FM user data or FM data records.

The K bus allows data to be exchanged between the CPU and the FM using the S7 data area of the object server.

Choose Coordination for P Bus Communication

The following resources from M7 RMOS32 can be used to coordinate the exchange of data via the P bus from the CPU to the FM:

- Communication via user data:
  - The FM sends a process or diagnostic alarm to the CPU module when it has processed the received data and is ready to receive new data.
  - The CPU is registered to receive alarm messages from the alarm server, and acknowledges the alarm to the alarm server after it has sent new data to the FM. The acknowledgement is the signal to the FM that new data has been transferred.
  - The FM polls the status of the alarm processing in order to find out whether the CPU has sent new user data.
- Communication via data records:
  - The C user program on the FM is registered to receive messages from the object server, and is notified when the CPU module has sent a new FM data record.
  - If necessary, the FM can send a process or diagnostic alarm to the CPU module as notification that the received data records have been completely read and that the FM is ready to receive new data.
  - The CPU is registered to receive alarm messages from the alarm server, and sends new data to the FM on receiving the alarm (alarm acknowledgement is not processed further in this case).

Note

Coordination of data transfer through the P bus in the reverse direction (in other words from the FM to the CPU) can take place using the same method in reverse.

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Communication type:

- The required speed should be transferred cyclically from the CPU to the FM during program execution. Since only a single value is transferred, simple coordination using notification of access by the object server should be sufficient.
- The start/stop commands from the OP and the enable signal from the simulation module should be transferred cyclically from the CPU module to the FM.
- The FM notifies the CPU cyclically about the operating state (on/off). Further coordination is not planned.
- The FM cyclically stores the actual speed value in a data record. The CPU reads this record cyclically. Further coordination of both processes is not planned.

Resource for the communication:

- The required speed and the actual speed are each stored in a data record. Data exchange takes place through the P bus.
- The start/stop commands of the OP and the enable signal of the simulation module are stored cyclically by the CPU module in the user data inputs (from the FM's viewpoint).
- The drive status is stored by the FM cyclically in the user data outputs (from the FM's viewpoint).

Coordination between CPU and FM:

When the CPU module transfers a data record with the new required speed, the C user program on the FM must be notified by the object server.

Communication measures on the CPU module:

- Transfer "enable" and "start/stop" signal to the FM user data:
- Send the new required speed cyclically to the FM data records,
- Read the "drive status" signal cyclically from the FM user data and transfer to the assigned flag area,
- Read the actual speed cyclically from the FM data records and transfer to the assigned flag area.

Communication measures on the FM:

- Create data record for the required speed as an S7 object,
- Create data record for the actual speed as an S7 object,
- Register with the object server for notification of write access to the data record,
- Interrogate “enable” and “start/stop” signals in the user data and react accordingly,
- Read new required speed from the data record,
- Write the drive status signal cyclically to the FM user data,
- Write the actual speed cyclically to the data record.

10.7 Summary of the FM User Data

Access to the User Data

16 bytes each for input and output are available for the FM user data.

The memory area for the user data is implemented on the FM as so-called dual-port-ram. This means that both the FM processor and the CPU can access this memory area.

However, the significance of the user data is different from the CPU's viewpoint and the FM's viewpoint:

Output from the CPU's viewpoint (and thus for the CPU user program) are inputs from the FM's viewpoint (and thus for the FM user program) and vice versa.

Figure 10-3 summarizes once again the access mechanisms for the user data:

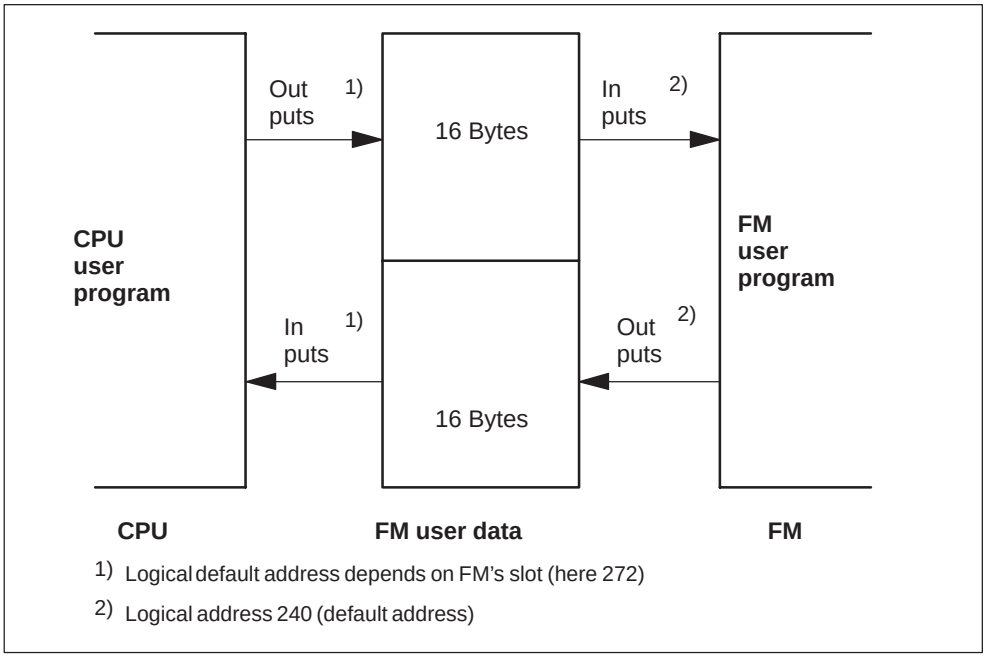


Figure 10-3 Accessing the FM User Data

Structure of the User Data

You can structure the FM user data as required provided that you code the CPU user program accordingly.

Procedure

Prepare an assignment list for the **FM** user data inputs and the **FM** user data outputs.

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Structure of the FM user data inputs (CPU outputs):

The inputs of the FM user data are assigned to the control signals START/STOP and the enable signal FREI_EIN from the CPU. Byte 0 of the user data inputs has the following bit assignment from the FM's viewpoint:

Bit 7						Bit 1	Bit 0
						START/STOP	FREI_EIN

Structure of the FM user data outputs (CPU inputs):

The outputs of the FM user data are assigned to the drive status signal (STATUS). It is used by the FM to inform the CPU of the drive status. Byte 0 of the user data outputs has the following bit assignment from the FM's viewpoint

Bit 7							Bit 0
							STATUS

10.8 Choice of the S7 Objects

Which S7 Objects are Available?

Under M7 RMOS32, the following S7 objects are available you C user program:

- Process data, including FM user data (they are automatically configured by M7 RMOS32 as S7 objects during hardware configuration),
- FM data records,
- Flag areas,
- Data blocks,
- Timers,
- Counters.

Usage of Data Records

You can use data records for the following applications:

- To send configuration data for the FM from the CPU to the FM,
- To exchange other data between the FM and the CPU.

Specifying the Data Records for the FM

Data record numbers 2 to 127 are reserved for user programs such as your FM user program; each of these records has a maximum length of 240 bytes but can otherwise be structured as required. Make a list of the records that you need in your program (for example one record for FM configuration data) and choose the structure of each of the records.

Decide on the following parameters for each record:

- Record number,
- Record length,
- Data structure,
- Intended access by the FM (reading, writing, reading/writing).

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

In the example, apart from the process data you only need data records for the CPU to transfer the control data as S7 objects:

- One data record for the required speed,
- One data record for the actual speed.

Record numbers:

The requires speed has the record number “10” and the actual speed has the record number “11”.

Length of the records:

Each of these record (required speed and actual speed) is one (SIMATIC) word in length (16 bit).

Record structure:

Record number	Allocation
10	Word 0 = required speed as an integer
11	Word 0 = actual speed as an integer

Access to the records by the FM and the CPU:

Record number	Access by the FM	Access by the CPU
10	Reading	Writing
11	Writing	Reading

10.9 Deciding on the Reaction to Operating Modes and Operating Mode Transitions

The OMT server allows user programs to interrogate operating modes. Operating modes are determined either by the position of the mode switch on the FM or by the operating mode of the CPU.

In addition, user programs can register with the OMT server for notification of operating mode transitions.

User programs can also register with the free cycle server to allow it to carry out cyclic background tasks or initialization activities during STARTUP mode.

User programs can execute independently of the defined operating modes (STOP, STARTUP, RESET, and RUN). However, the physical signals are only output from the local I/O modules to the process actuators during RUN mode.

Program Reaction in STARTUP Mode

In the operating mode STARTUP, you should carry out all programming measures which are necessary for the **technological startup** of your program. If the program is split between several tasks, you can provide an appropriate program section in each task for this purpose.

Program Reaction in RUN Mode

In the operating mode RUN, you should implement all programming measures that are necessary for the process control assignment itself. If the program is split between several tasks, you can provide an appropriate program section in each task for this purpose.

Program Reaction to the Transition to STOP or HALT Mode

On detecting a transition from RUN to STOP or HALT mode, you should pause all control processes in your FM program. Although signals are still read from the I/O modules in the STOP or HALT mode, no signals are output.

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Program reaction in STARTUP mode

In the example program, initialization takes place in STARTUP mode. The enable signal is sent to the drive, the controller task is started and the required speed is initially set to 0.

Program reaction in RUN mode

All technological activities of the CPU and FM execute in RUN mode

Program reaction to the transition to STOP or HALT mode

In the transition from RUN to STOP or HALT the drive is braked, the drive enable signal is cleared and the controller task is terminated.

10.10 Choosing the Logical Program Structure

A prerequisite for choosing the program structure is that the technological subdivision of responsibilities between the CPU and the FM has already been decided. If this is not the case, you must first choose which responsibilities should be undertaken by the FM and which of them by the CPU.

When deciding, consider both the FM user program and also the modules which cooperate with the FM and the tasks of your CPU user program.

How should the FM Work?

There are several ways that the FM can carry out its assignment. Use the following method to decide the most appropriate approach for your program:

- With fixed parameters,
- With dynamic parameters that are continuously specified by the CPU module,
- Without result data for the CPU module,
- With result data for the CPU module,
- With permanent data on mass storage,
- Without permanent data on mass storage,
- With non-volatile data in static RAM,
- Without non-volatile data in static RAM,
- Operates in a continuous loop,
- Controlled by events (alarms, time messages),
- Operates in a continuous loop **and** controlled by events.

Design the Cooperation CPU-FM into Your CPU Program

Since your FM probably doesn't function autonomously, but cooperates with the CPU module, you must consider the cooperation CPU-FM when planning and implementing your CPU program.

In order to clearly delineate in the CPU program the operations needed for cooperation with the FM, you should for example implement those parts of the CPU program in an FC and/or in a separate task.

Choosing Tasks

First decide which tasks are necessary on the FM to solve your automation objective. If several tasks are necessary, then decide on the subdivision of work between the tasks.

The result of this development step should be a sketch of the task structure of your FM program.

Determine the Functional Blocks

Subdivide each task into functional blocks and define the functions of each block.

The result of this development step should be a block diagram for the functional sequence in each task.

Planning the Functions

After you have chosen the functional blocks, you can probably recognize whether there may be subtasks which can either be implemented by a function which is already available, or for which it is meaningful to program a new function.

List the required functions with a short description of their purpose.

Result

The result of this development step should provide sufficient information for planning the task coordination and determining the program execution.

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Function of the FM:

- The FM uses both **dynamic** runtime parameters (required speed etc.) and also **static** parameters (control parameters) which are specified in the commissioning phase.
- The responsibility of the FM program is to **report** the actual drive status and the actual speed to the CPU module.

- **Non-volatile data** in static RAM is not necessary.
- OP commands in the FM user data and the STOP button are interrogated **cyclically** (**free cycle message**).
- By default, the speed control takes place with a **cycle time of 1ms** (**time message**).
- The actual speed and the actual drive status are written **cyclically** to a data record and/or the user data (**free cycle message**).
- The need to **transfer a new required speed data record** is notified by the object server with a message to the FM user program (the FM user program registers with the object server to receive **access messages**).
- The **operating modes STOP** and **HALT** are notified by the OMT server with a message to the FM user program (the FM user program registers with the OMT server to receive **OMT messages**).

Extensions to the CPU program:

In the example program, the CPU program has been specially designed for the FM program. The following features are planned:

- Flag area for the OP commands “start”, “stop” and “required speed” and for the results “actual speed“ and “drive status”.
- A separate section within the OB1 module (S7 program) and/or a separate task (M7) which is registered with the free cycle server for notification of FC events. This section and/or task is responsible for communication with the FM.

Planned tasks on the FM:

All functions are implemented within a single C program. The program is subdivided into two tasks, the MAIN_TASK and the CONTROLLER_TASK.

- The MAIN_TASK has the following functions:
  - Activity in initialization section (STARTUP):

    Create the CONTROLLER_TASK

    Create records 10 and 11 and register DS 10 with the object server for notification of writing access.

    Register with the free cycle server for notification of the operating mode STARTUP and for the free cycle.

    Register with the time server to receive cyclic messages to be used for keyboard interrogation.

    Register with the OMT server for notification of the operating modes STOP and HALT to allow an appropriate reaction to be implemented.

  - Activities at the 200 ms time message:

    Output error messages to the console

    Interrogate keyboard and change the control parameter

  - Activities during STARTUP mode:

    Set required speed to 0

    Enable drive controller and set drive enable

    Start CONTROLLER_TASK

  - Activities during the free cycle:

    Interrogate OP commands “start/stop“ and “enable”

    Interrogate STOP button

    Set or clear drive enable

    Write drive status and actual speed value

  - Activities on access to DS 10 (in other words new required speed from the CPU program):

    Read DS 10 and set new required speed

  - Activities on receiving terminate task message:

    Deregister notification messages from all servers

    Delete data records

    Delete CONTROLLER_TASK

    Terminate MAIN_TASK

- The CONTROLLER_TASK has the following functions:
  - Activities in initialization section:

    Register with time server for 1 ms time message

  - Activities at 1 ms time message:

    Control drive if no time messages have got lost and drive enable is set by the MAIN_TASK

    Otherwise brake drive with maximum braking current and clear drive enable after 200 ms.

  - Activities on receiving terminate task message:

    Deregister notification messages from the time server

    Terminate CONTROLLER_TASK

Logical blocks for the tasks:

The following figure shows a subdivision of MAIN_TASK and CONTROLLER_TASK in logical processing blocks.

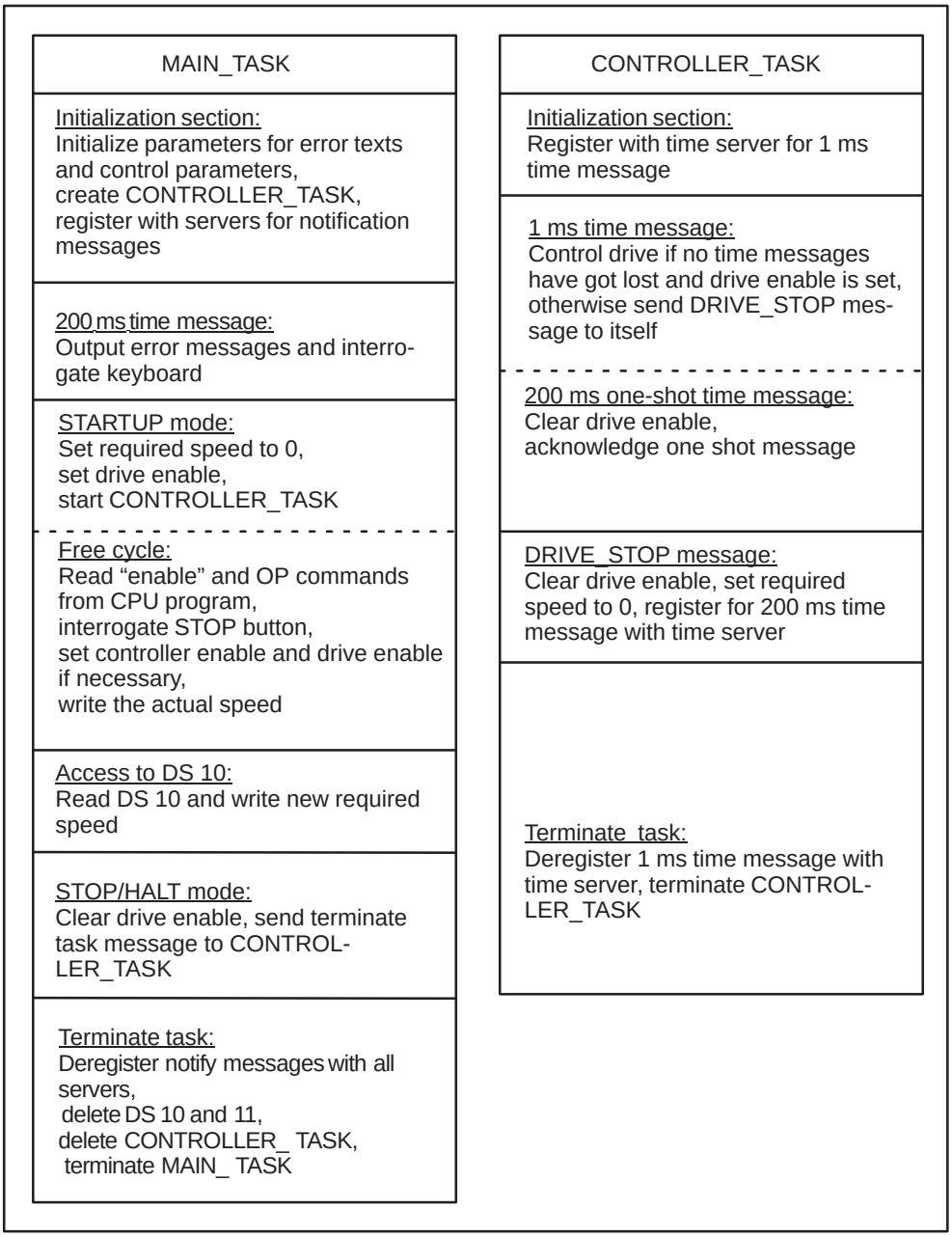


Figure 10-4 Logical Blocks of the Tasks in the FM Program

Functions:

C functions are used for calculation the parameters for drive speed control, to output the control value and read in the actual speed.

10.11 Deciding on Task Coordination and Listing Global Data

Analyze Dependencies

Proceed as follows:

1. Investigate which dependencies are present between each of the tasks in the planned multitasking program,
2. List the affected tasks and describe their dependencies.

Choose Type of Coordination

There are several ways of coordinating tasks. Choose the most suitable method for your program from:

- Global variables:

  The controlling task sets a global variable, the dependent task interrogates this variable and reacts accordingly.

- Event flags:

  The controlling task sets an event flag and the dependent task interrogates this flag or waits for the flag to be set and reacts accordingly.

- Semaphores:

  In order for example to protect a common data areas against inadmissible overwriting, all of the tasks which have access to the data area should share a semaphore. Only one task at a time can occupy the semaphore. The semaphore is released again when the access is finished.

  If a semaphore is already occupied by a task, another task which wants access is halted until the task with the semaphore releases it again.

- Messages:

  The controlling task sends the dependent task a message. The dependent task waits for the message and reacts when the message arrives in accordance with the message contents.

Specify Resources for Coordination

After choosing the type of coordination, specify the resources with which you want to carry out the coordination:

- Event flags and their significance.
- Semaphore names and the assigned common data areas and/or other objects to be protected.
- Message codes and their significance.

Choose Resource Names for the Resource Catalog

Depending on the extent of the automation assignment, the program code may need to be distributed between one or more C programs. Different C programs have different address spaces and cannot thus exchange information via global variables.

Information exchange between tasks in different C programs can only take place via entries in the resource catalog. Accordingly, you should choose names for all required resources (task names, mailbox names etc.) and enter them into the resource catalog to make the resources globally accessible.

Choose Global Data for the C User Program

Depending on the extent of the automation assignment, your C user program may require global data in C notation to make it accessible to all of your programs. If possible, decide on the type and structure of this data during the program design phase.

Decisions Which were Taken at this Stage of Development

The following decisions were taken in this development stage:

Dependencies:

- The MAIN_TASK determines whether the drive controller is enabled or not (interrogation of the CPU “enable” signal, the OP command “start/stop” and the STOP button).

  If the drive controller is disabled, the CONTROLLER_TASK carries out an emergency stop of the drive and clears the drive enable signal after 200 ms.

- The MAIN_TASK interrogates the PC command “terminate” and the STOP/HALT mode, and causes the CONTROLLER_TASK to brake the drive and to terminate if necessary.

Type of coordination:

The above dependencies are coordinated by the following program mechanisms:

- The MAIN_TASK sets the status of the global variable “run” according to the status of the CPU “enable” signal, the OP command START/STOP and the status of the STOP button, and stops the drive (controller output=0) and disables the controller accordingly.

  On detecting “run = false”, the CONTROLLER_TASK sets the control variable for the current controller to 0 and clears the drive enable signal after 200 ms.

- On detecting the PC command “terminate” and/or “new sampling period” together with STOP/HALT mode, the MAIN_TASK sets the global variable “run” to false in order to stop the drive (controller output=0) and to disable the controller.

  After a pause of 250 ms, the MAIN_TASK then sends the TASK_END message to the CONTROLLER_TASK. The higher priority of the CONTROLLER_TASK allows it to complete its clearing up work before terminating.

  In the case of the PC command “terminate”, the MAIN_TASK then deletes the CONTROLLER_TASK and terminates itself.

  In the case of “new sampling period”, STARTUP mode is requested after the CONTROLLER_TASK has terminated in order to re-initialize the CONTROLLER_TASK (this is necessary to change the sampling period).

Global data for the FM program:

Since both tasks can be created within the same C program, information exchange between the tasks can take place using global variables.

The following common data - which should be declared as global data - is required for the MAIN_TASK and the CONTROLLER_TASK :

- The variable “run”, which specifies the controller status. The variable run is “set” or cleared in the MAIN_TASK and evaluated by the CONTROLLER_TASK.
- Task ID of the CONTROLLER_TASK, to allow the main task to send a TASK_END message to the CONTROLLER_TASK.
- Error flag, which can be set by the CONTROLLER_TASK and is interrogated cyclically by the MAIN_TASK are output to the console.

The following global variables are also used:

- FRBs for registering for notification messages from the servers. The memory for FRBs must not be allocated on the stack; it must either be defined globally or via explicit memory request from the heap and/or other memory pools.
- Other global variables (error texts, sampling period), names of the external C functions.

10.12 Specifying the Program Execution

Requirement

If you have designed your user program in accordance with the above procedure, you can now specify the program structure.

If you are not clear about the usage of the M7 RMOS32 functions, please refer again to the corresponding sections in chapters 6 and 7.

Procedure

The basis for this development step are the decisions that you made for the logical program blocks and the task coordination. Develop a flow diagram for each logical block which you can also refine in several passes if necessary (iterative method).

Result

The result of this development step should be suitable as the basis for coding your C user program and for extending the CPU user program.

Task Sequence of FM Program

Figures 10-5 and 10-6 show the sequence of the tasks in the example FM program.

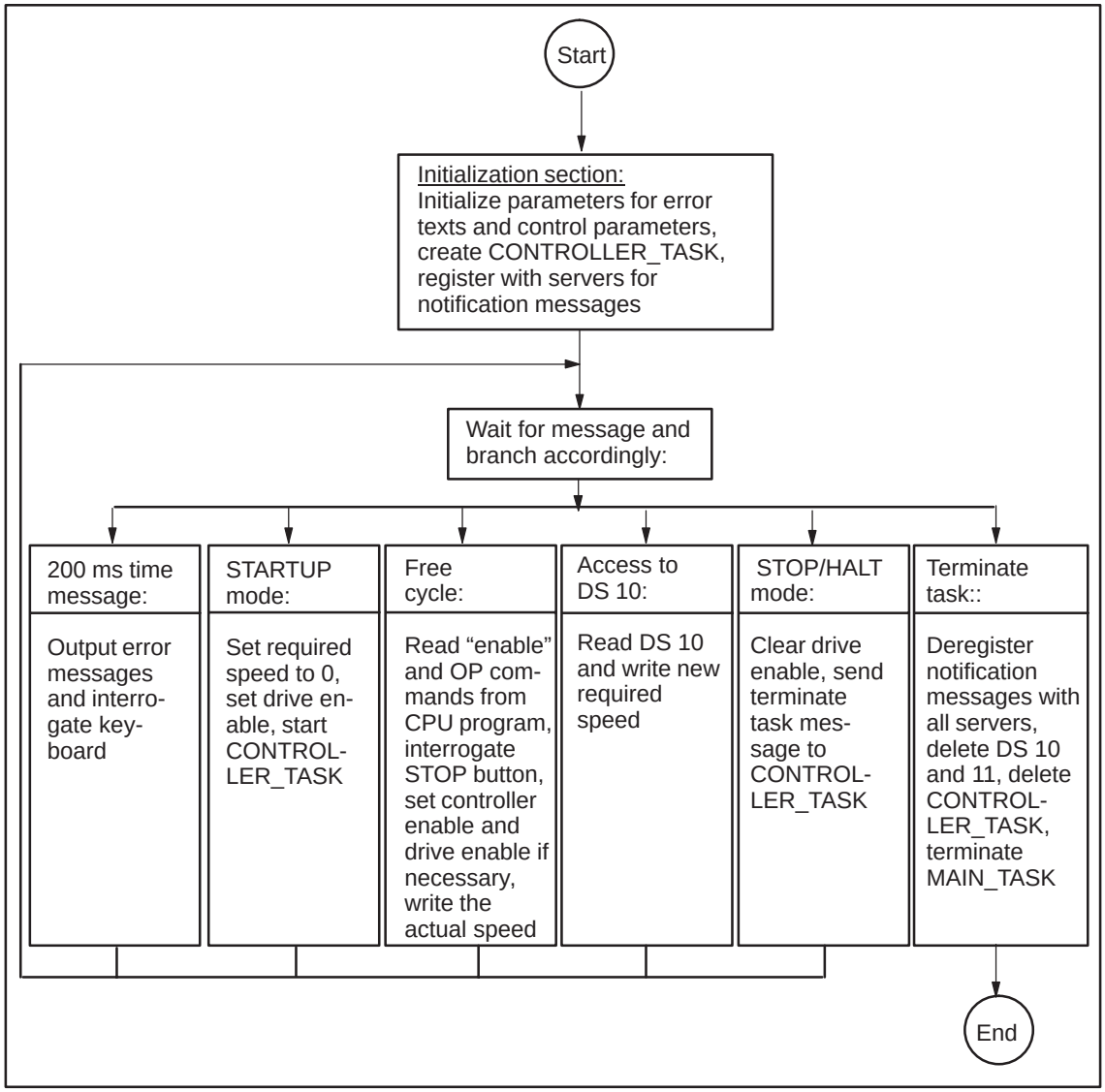


Figure 10-5 Functional Sequence of the CONTROLLER_TASK on the FM

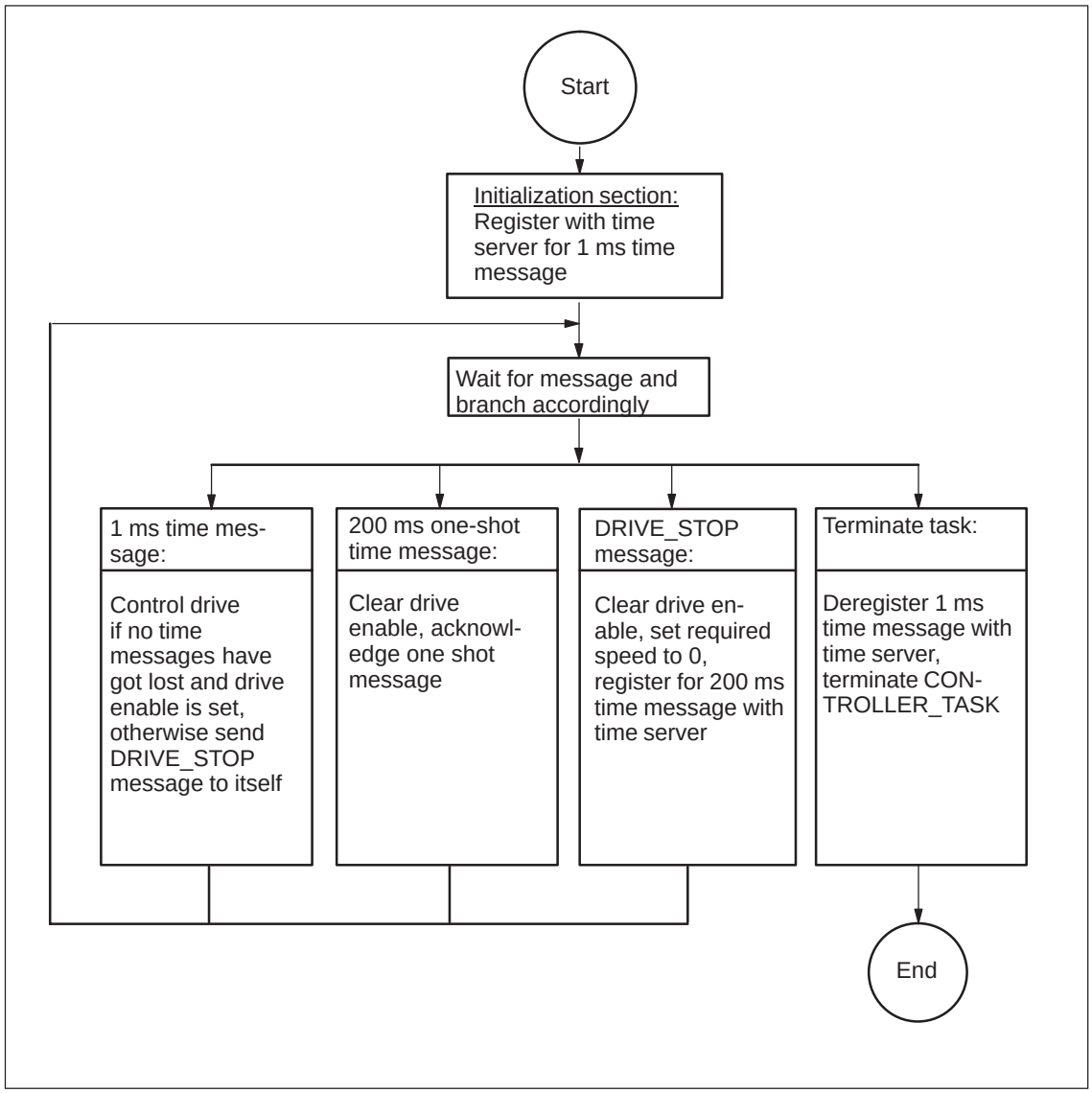


Figure 10-6 Functional Sequence of the CONTROLLER_TASK on the FM

Task Sequence of CPU Program

Figures 10-7 and 10-8 show the sequence of the tasks in the CPU user program.

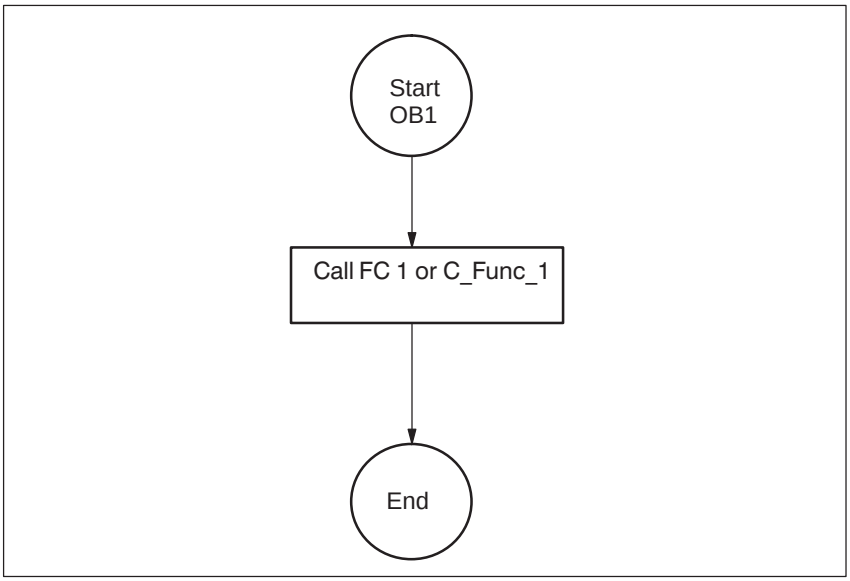


Figure 10-7 Functional Sequence of the OB1 and/or the Corresponding Task on an M7 CPU

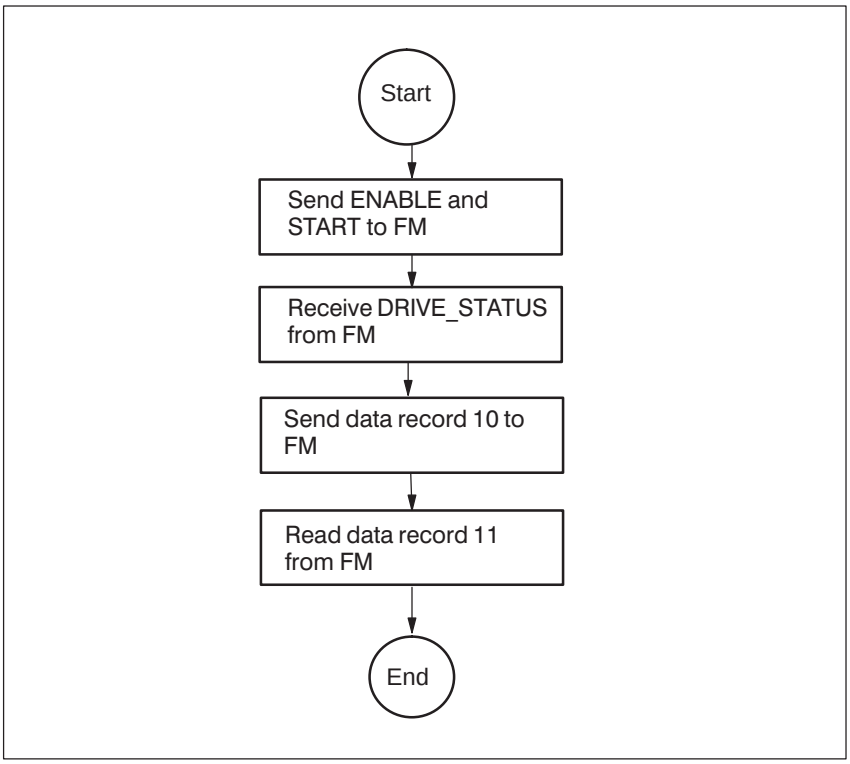


Figure 10-8 Functional Sequence of the FC1 and/or the Task C_Func_1 on an M7 CPU

10.13 Summary of the Design of the Example Program

This section summarizes the main design features of the example program.

Overview of the Software Components

Figure 10-9 gives an overview of the software components which were chosen for solving the example automation problem.

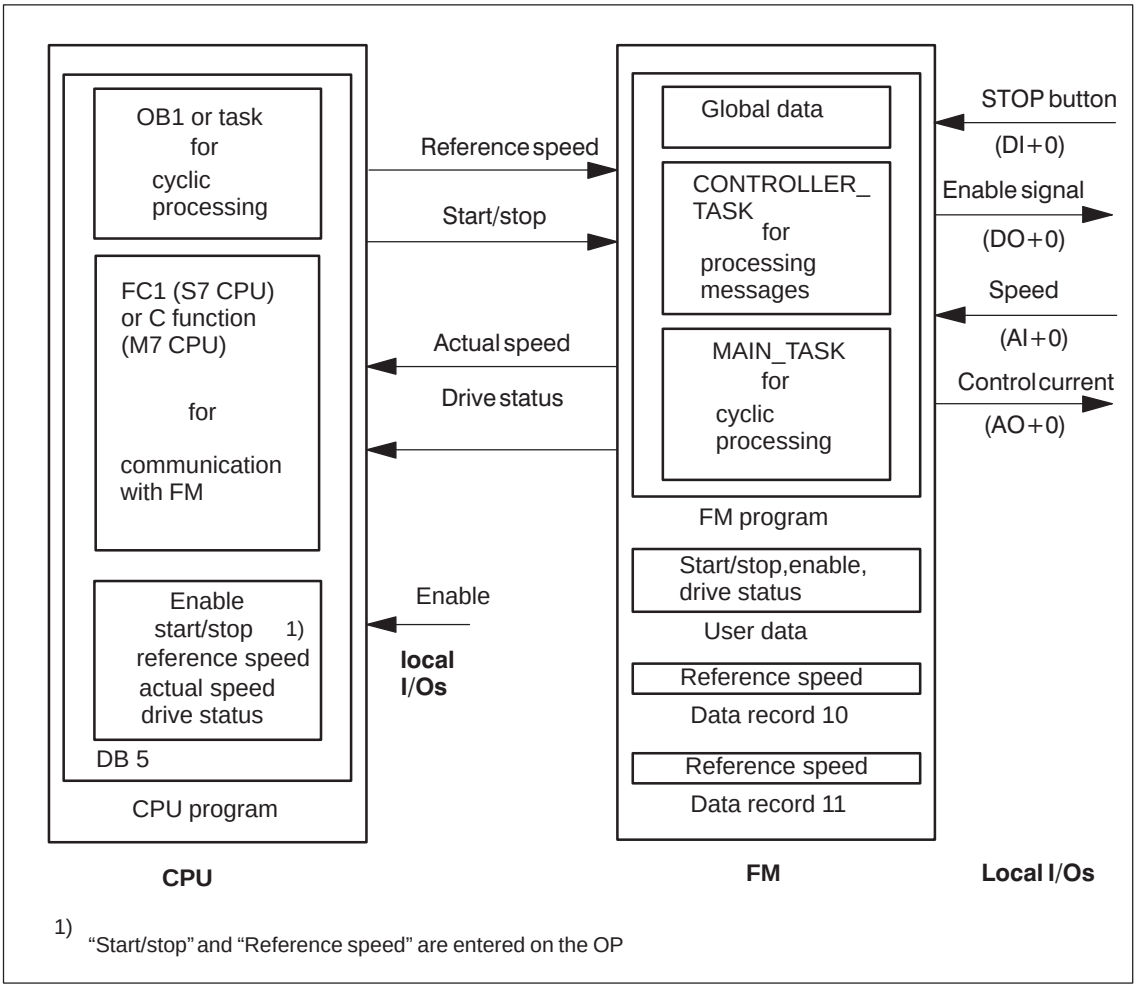


Figure 10-9 Summary of the Software Components for Implementing the Example Program

Process Data

The CPU module uses the following process data

- Enable signal from the simulation module: digital input on the simulation module DI with the symbolic name ENABLE.

*The FM uses the following process data:

- STOP button: Digital input on the signal module DI with the symbolic name STOP_BUTTON.
- Enable signal for the current controller: Digital output on the signal module DO with the symbolic name ENABLE_OFF.
- Current value of drive speed: Analog input on the signal module AI with the symbolic name ACT_SPEED.
- Control current: Analog output on the signal module AO with the symbolic name CONTROL_CURR.
- FM user data:
  - Inputs: byte 0/bit 0 for the signal ENABLE_ON, bit 1 for the signal START/STOP
  - Outputs: byte 0/bit 0 for the signal STATUS.

Communication with the CPU

The following data is transferred between the CPU module and the FM:

The CPU module writes the command signals START/STOP and ENABLE to the FM user data and transfers the reference speed to the FM in a data record.

The FM then writes the drive status to the user data and writes the actual speed to the FM data record. The actual speed in this record is read cyclically by the CPU module.

S7 Objects

The following S7 objects are required:

- Record 10 for the reference speed. The CPU module has write access to this record and the FM has read access to this record.
- Record 11 for the actual speed. The FM has write access to this record and the CPU module has read access to this record.

Reaction to Operating Modes

The FM user program gets notification of STOP/HALT mode from the OMT server and disables the drive controller accordingly.

Logical Program Structure

The user programs are subdivided into the following logical sections:

- FM user program:

  The C user program for the FM contains the logical parts MAIN_TASK, CONTROLLER_TASK and C functions for the controller algorithms.

  MAIN_TASK handles the cyclic interrogation of signals and commands and processes the terminate task event.

  The CONTROLLER_TASK is time controlled and handles the drive speed regulation and carries out an emergency stop if necessary.

  The C function INPUT is used to read in the required speed, CONTROLLER is used to calculate the controlling parameters and OUTPUT is used to output the controlling current to the drive.

- CPU user program:

  The OB1 calls FC1 cyclically (S7 CPU) and/or the free cycle server calls the task C_FUNC_1 cyclically (M7 CPU).

  FC1 and/or C_FUNC_1 transfer the OP command START/STOP M98.0 and the signal ENABLE M96.0 to the FM user data.

  In addition, FC1 and/or C_FUNC_1 read the drive status STATUS from the FM user data and stores it in M92.0 of the CPU module.

  The reference speed is transferred from MW 94 on the CPU module to FM data record 10.

  The actual speed is transferred from FM data record 11 to MW 90 on the CPU module.

Programming Examples

Program source code for the example described in this chapter can be found in the file *cpubeisp.c* and *fmbeisp.c* in the directory ..\M7SYS\EXAMPLES\M7API. These files form the framework required to develop the example automation assignment. Various parts, for example the controller algorithm, will need to be added to implement a fully functional system.

Cycle and Reaction Times of the M7-300/400

11

This chapter explains how the cycle and reaction times of the M7-300/400 are comprised.

You can display the cycle time of your user program on the relevant CPU using the programming device (see *STEP 7 User Manual*).

An example is used to illustrate how the cycle time is calculated.

More important for monitoring a process is the reaction time. This chapter explains in detail how you calculate the reaction time. If you are using a CPU/FM as the master in a PROFIBUS DP network, you must also take the bus operating times into account (see Section 11.2).

Chapter Overview

11.1 Cycle Time

Cycle Time Definition

The cycle time or scan cycle time is the time which elapses while a program cycle is run through.

Parts of the Cycle Time

The cycle time comprises:

Factors	Remark
Operating system processing time	See Section 11.2
Process image transfer time (PIQ and PII)	
User program processing time	This is calculated from the execution times of the individual calls
Communication	Communication can increase the load and the cycle time.
Interrupt load and higher priority tasks	See Section 11.4

Note

On M7-300/400 several user programs may run in the free cycle.

Figure 11-1 shows the parts of the cycle time:

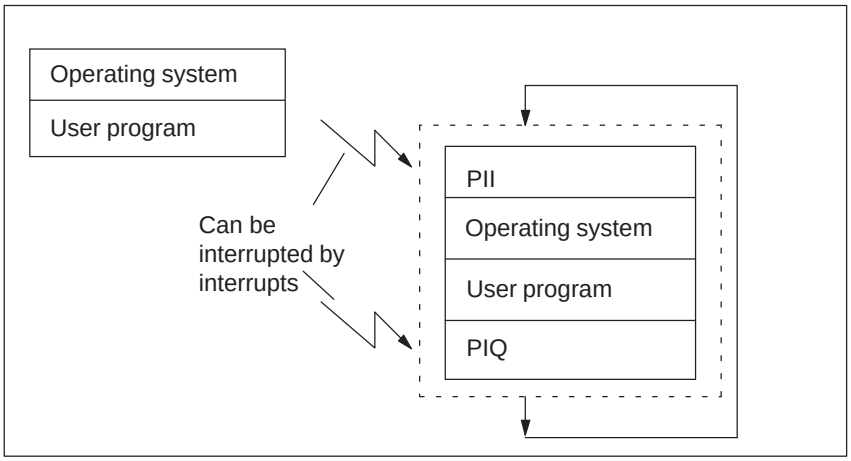


Figure 11-1 Parts of the Cycle Time

Increasing the Cycle Time

You should note that the cycle time of a user program is increased by the following:

- Time-driven interrupt processing
- Hardware interrupt processing (see also Section 11.4)
- Diagnostics and error processing (see also Section 11.5)
- Communication via the MPI and communication bus
- Other tasks with higher priority

Note

If you have configured a minimum cycle time, an actual cycle time shorter than this set minimum is increased by a wait time to the minimum cycle time. This means that the time between the update of the inputs and outputs is the actual (shorter) cycle time, but the time between the subsequent update of the inputs will be at least the minimum cycle time.

11.2 Reaction Time

Reaction Time Definition

The reaction time is the time from an input signal being recognized to changing an output signal linked to it.

Factors

The reaction time depends on the cycle time and on the following factors:

Factors	Remark
Delay in the inputs and outputs	You will find the delay times in the technical specifications for the signal modules.
Additional bus operating times in the PROFIBUS DP network	

Variations

The actual reaction time lies between a shortest and a longest reaction time. When you configure your plant, you should always calculate using the longest reaction time.

In the following sections, the shortest and longest reaction times are explained so that you have a better overview of the possible range of variation in reaction times.

Shortest Reaction Time

Figure 11-2 shows you which conditions are required to achieve the shortest reaction time.

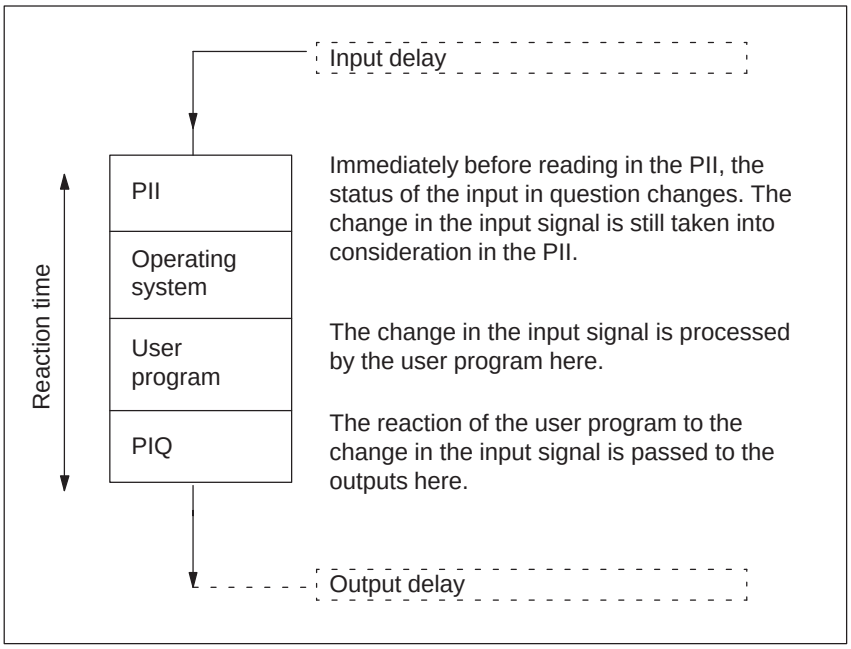


Figure 11-2 Shortest Reaction Time

Calculation

The (shortest) reaction time is made up as follows:

- 1 × process image transfer time of the inputs +
- 1 × operating system processing time +
- 1 × program processing time +
- 1 × process image transfer time of the outputs +
- Delay in the inputs and outputs

The corresponds to the sum of the cycle time and the delay in the inputs and outputs.

Longest Reaction Time

Figure 11-3 shows you how the longest reaction time results.

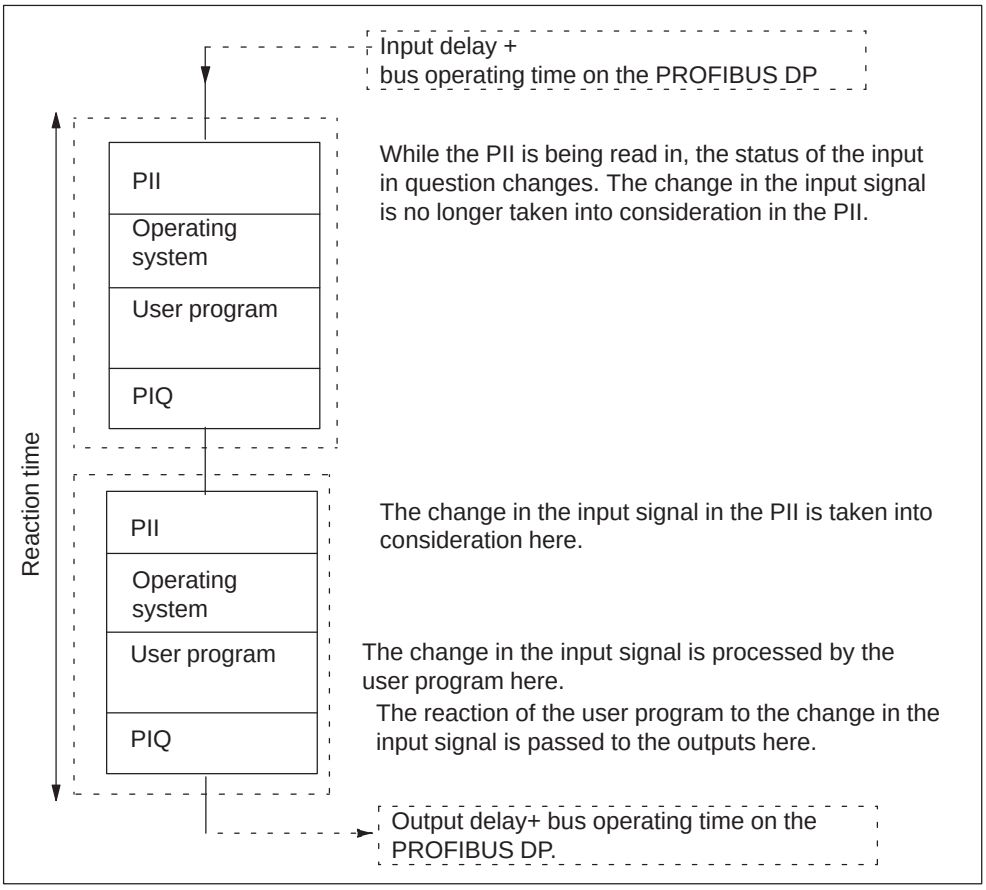


Figure 11-3 Longest Reaction Time

Calculation

The (longest) reaction time is made up as follows:

- 2 × process image transfer time of the inputs +
- 2 × process image transfer time of the outputs +
- 2 × operating system processing time +
- 2 × program processing time +
- 2 × bus operating times in the PROFIBUS DP network +
- Delay in the inputs and outputs

This corresponds to the sum of twice the cycle time and the delay in the inputs and outputs plus twice the bus operating time.

Operating System Processing Time

Table 11-1 lists the operating system processing times for the CPUs. The influence of test functions such as monitoring variables, of block functions such as downloading/deleting/compressing, and of removing/inserting during operation is not taken into consideration.

Table 11-1 Operating System Processing Times of the CPU/FM

Processing Time	CPU 388-4	CPU 488-3	CPU 486-3	FM 356-4	FM 456-4
Min.	approx. 645 µs	approx. 140 µs	approx. 260 µs	approx. 661 µs	approx. 539 µs
Max.	approx. 777 µs	approx. 185 µs	approx. 314 µs	approx. 791 µs	approx. 691 µs
Typ.	approx. 663 µs	approx. 150 µs	approx. 268 µs	approx. 677 µs	approx. 589 µs

Updating the Process Image

Table 11-2 contains the CPU/FM times for the process image update (process image transfer time). The values listed are 'ideal' values which are increased by interrupts occurring or by the CPU communicating.

The M7-300 CPU/FM times for the process image update are calculated as follows:

K + no. of bytes in the PI in rack "0" ×A
+ no. of bytes in the PI in the expansion racks "1" to "3" ×B
+ no. of bytes in the PI via DP ×D
= CPU time

The M7-400 CPU/FM times for the process image update are calculated as follows:

K + no. of bytes in the PI in the central rack×A
+ no. of bytes in the PI in the expansion unit with local link ×B
+ no. of bytes in the PI in the expansion unit with remote link ×C
+ no. of bytes in the PI via DP ×D
= CPU time

Table 11-2 Process Image Update of the CPUs/FMs

	Parts	CPU 388-4	CPU 488-3	CPU 486-3	FM 356-4	FM 456-4
K	Basic load	172 µs	24 µs	56 µs	172 µs	66 µs
A	Per byte in central rack and rack “0” resp.	8.2 µs	5.2 µs	3.3 µs	8.2 µs	14.3 µs
B	Per byte in expansion unit with local link and in racks “1” to “3” resp.	8.2 µs	12.7 µs	15.3 µs	–	–
C	Per byte in expansion unit with remote link	–	not available	not available	–	–
D	Per byte in DP area	7 µs	3 µs	4 µs	7 µs	5 µs

Input/Output Delays

You should note the following delay times depending on the module:

- For digital inputs: the input delay time
- For digital outputs: delay times can be ignored
- For relay outputs: typical delay times of 10 ms to 20 ms. The delay in the relay outputs is also dependent on temperature and voltage.
- For analog inputs: cycle time of the analog input
- For analog outputs: response time of the analog output

Bus Operating Times in the PROFIBUS DP Network

If you configured your PROFIBUS DP network with STEP 7, STEP 7 calculates the typical bus operating time you can expect. You can then display the bus operating time of your configuration on the programming device (see *STEP 7 User Manual*).

Figure 11-4 shows an overview of the bus operating time. In this example, it is assumed that each DP slave has 4 bytes of data on average.

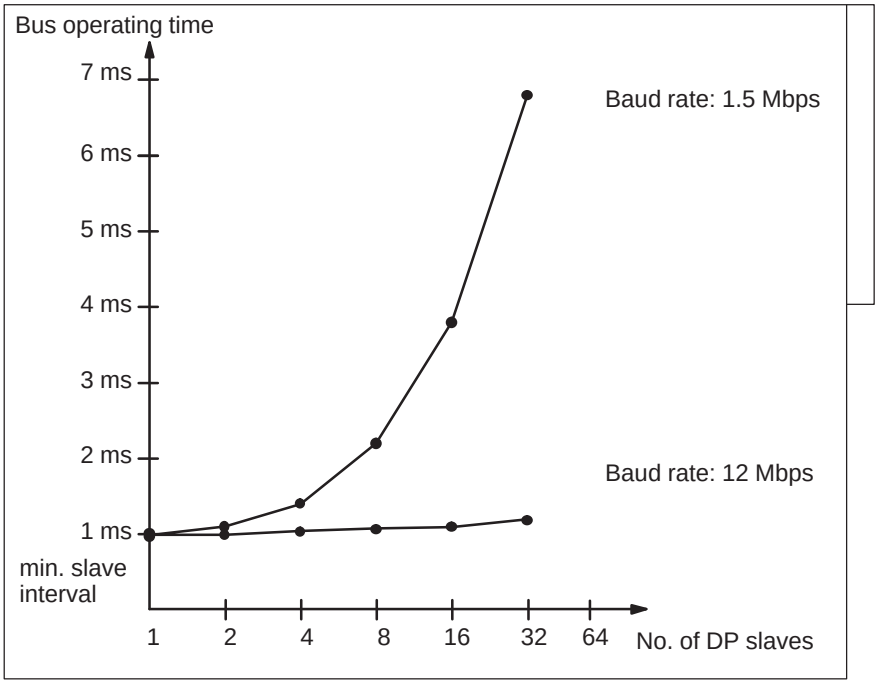


Figure 11-4 Overview of the Bus Operating Times for PROFIBUS DP at 1.5 Mbps and 12 Mbps for Integrated DP Interface

If you run a PROFIBUS DP network with a number of masters, you must take into account the bus operating time for each master. This means the total bus operating time = bus operating time $\times$ no. of masters.

Increasing the Cycle Time by Nesting Interrupts

Table 11-3 shows how the cycle time is typically increased by nesting an interrupt. In addition to the increase in the cycle time, there is also the program execution time in the interrupt level. If a number of interrupts are nested, their times must be added together.

Table 11-3 Increase in Cycle Time by Nesting Interrupts

CPU/FM	Hardware Interrupt	Diagnostic Interrupt alarm	Time-of-Day Interrupt	Time-Delay Interrupt
CPU 388-4	approx. 626 µs	approx. 877 µs	approx. 107 µs	approx. 107 µs
CPU 488-3	approx. 154 µs	approx. 203 µs	approx. 69 µs	approx. 69 µs
CPU 486-3	approx. 279 µs	approx. 399 µs	approx. 162 µs	approx. 162 µs
FM 356-4	approx. 626 µs	approx. 877 µs	approx. 107 µs	approx. 107 µs
FM 456-4	approx. 319 µs	not available	approx. 110 µs	approx. 110 µs

11.3 Calculation Example for the Cycle Time and Reaction Time

Parts of the Cycle Time

As a reminder, the cycle time is made up of the following:

- Process image transfer time
- Operating system processing time
- User program processing time

Example for the Cycle Time

You have configured an M7-300 with a CPU 388-4 and 2 digital input modules with 2 bytes each in PI in the central rack. The user program has an assumed run time of 1.5 ms

Since the M7-300 is a single station, there are no communication activities at the scan cycle checkpoint.

Calculation

For the example, the cycle time is made of the following times:

- Process image transfer time

  Process image: 172 µs + 4 × 2 × 8.2 µs = approx. **0.24 ms**

- Operating system run time

  Cycle control: approx. **663** µs ≈ **0.66 ms**

- User program processing time: approx. **1.5 ms**

The cycle time is the sum of these times:

Cycle time = 0.24 ms + 0.66 ms + 1.5 ms **= 2.4 ms**

Parts of the Reaction Time

As a reminder, the reaction time is the sum of the following:

- 2 × process image transfer time of the inputs +
- 2 × process image transfer time of the outputs +
- 2 × operating system processing time +
- 2 × program processing time +
- Delay in the inputs and outputs

Example for the Reaction Time

You have configured an M7-300 with a CPU 388-4 and 4 digital input modules with 2 bytes each in PI in the central rack. The user program has an assumed run time of 1.5 ms

According to the example above a cycle time of 2.6 ms results.

Calculation

For the example, the reaction time is made of the following times:

- Double cycle time: 2 × 2.4 ms = 4.8 ms
- I/O delay time
  - the digital input module SM 321; DI 16 × DC 24 V has an input delay of max. **4.8 ms** per channel
  - the digital output module SM 322; DO 16 × DC 24 V output delay is not relevant

  Reaction time = 4.8 ms + 4.8 ms **= 9.6 ms**.

11.4 Interrupt Reaction Time

Interrupt Reaction Time Definition

The interrupt reaction time is the time from when an interrupt signal first occurs to calling the first instruction in the activated task. This applies both to hardware interrupts and to diagnostic interrupts.

As a rule, the following applies: interrupts with higher priority classes have priority. This means the interrupt reaction time is increased by the program processing time of the activated higher priority task and tasks with the same priority not yet processed.

Calculation

The interrupt reaction time is made up as follows:

Shortest interrupt reaction time =
minimum interrupt reaction time of the CPU +
minimum interrupt reaction time of the signal modules +
bus operating time on the PROFIBUS DP

Longest interrupt reaction time =
maximum interrupt reaction time of the CPU +
maximum interrupt reaction time of the signal modules +
2 × bus operating time on the PROFIBUS DP

Hardware Interrupt Reaction Times of CPUs/FMs

Module	Interrupt Reaction Time	
	min.	max.
CPU 388-4	approx. 290 µs	approx. 485 µs
CPU 488-3	approx.84 µs	approx. 152 µs
CPU 486-3	approx. 132 µs	approx. 210 µs
FM 356-4	approx. 290 µs	approx. 485 µs
FM 456-4	not available	not available

Diagnostic Interrupt Reaction Times of CPUs/FMs

Module	Interrupt Reaction Time	
	min.	max.
CPU 388-4	approx. 302 µs	approx. 346 µs
CPU 488-3	approx. 87 µs	approx. 220 µs
CPU 486-3	approx. 138 µs	approx. 507 µs
FM 356-4	approx. 302 µs	approx. 346 µs
FM 456-4	not available	not available

Signal Modules

The interrupt reaction time for signal modules is the time which elapses between the signal module recognizing an interrupt event and the signal module triggering the interrupt. You will find the hardware and diagnostic interrupt reaction times in the data sheet for the respective signal module.

11.5 Calculation Example for the Interrupt Reaction Time

Parts of the Interrupt Reaction Time

As a reminder, the hardware interrupt reaction time is made up of the following:

- Hardware interrupt reaction time of the CPU
- Hardware interrupt reaction time of the signal module
- 2 × bus operating time on the PROFIBUS DP

Example for the Longest Process Interrupt Reaction Time

You have configured an M7-300 comprising a CPU 388-4 and 4 digital modules in the central rack. A digital input module is the SM 321; DI 16× DC 24 V; with hardware and diagnostic interrupt. In the parameter assignment of the CPU and the SM, you have only enabled the hardware interrupt. You have assigned an input delay of 0.1 ms for the digital input module.

Calculation

For the example, the hardware interrupt reaction time is made of the following times:

- Hardware interrupt reaction time of the CPU 388-4: approx. 0.5 ms
- Hardware interrupt reaction time of the SM 421; DI 16 × 24/60 VUC (from data sheet):
  - Internal interrupt preprocessing time: 0.2 ms
  - Input delay: 0.1 ms
- As the signal modules are inserted in the central rack, the bus operating time on the PROFIBUS DP is not relevant.

The hardware interrupt reaction time is the sum of these times:

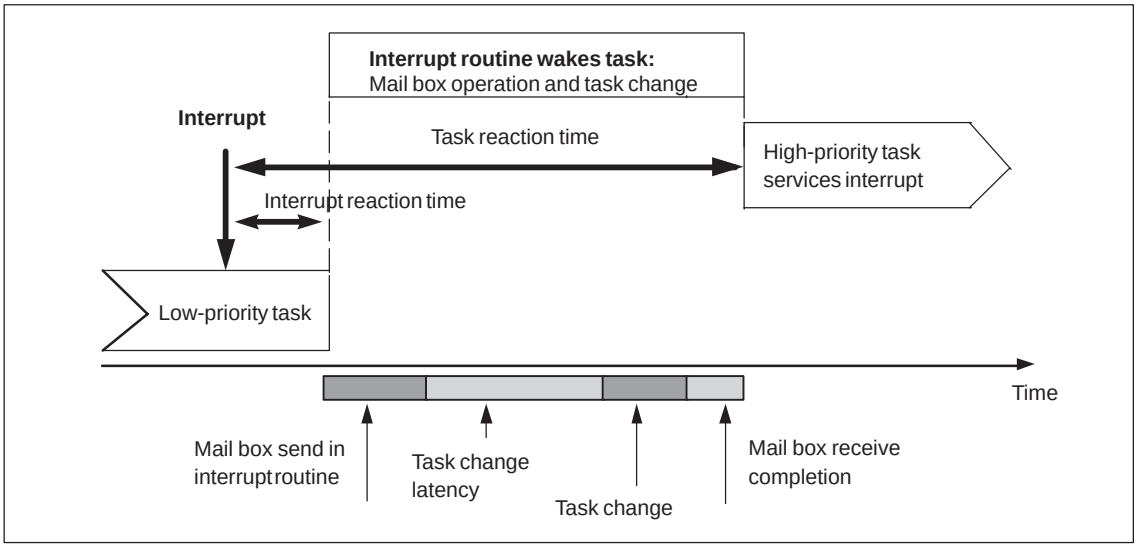
Hardware interrupt reaction time = 0.5 ms + 0.2 ms + 0.1 ms = **approx. 0.8 ms**

11.6 Operating System-Specific Reaction Times

Definition of Terms

This section describes the operating system-dependent reaction times that you must take into account in your C program. The terms are defined as follows:

Term	Definition
Task reaction time	Time which elapses between the occurrence of an interrupt and executing of the first operation in the task (without high-priority interrupts pending). It is the sum of the following times • Interrupt reaction time • Time for the execution of the system call that unblocks the high-priority task (e.g. send to a mail box) • Task change latency • Task change time • Completing execution of the blocked system call in the high-priority task (e.g. receive from a mail box)
Interrupt reaction time	Time which elapses between the occurrence of an interrupt and executing of the first operation in the interrupt service routine (without high-priority interrupts pending).
Task change latency	Time which elapses between the high-priority task getting ready and reenabling the system scheduling. These times can be increased by scheduler disabling in the user program.
Task change time	Time which elapses between the high-priority task getting ready and executing of the first operation in the task.



In a real-time operating system the interrupt reaction time and the task reaction

time are the most important measured values that are decisive for the quality of the operating system. The interrupt reaction time is responsible for how fast the operating system can react to external events such as completion of D/A conversion. The task reaction time is responsible for how fast the a high-priority task can react to external events.

Task Reaction Time

Module	Task Reaction Time		
	min.	typ.	max.
CPU 388-4	approx. 80 µs	approx. 85 µs	approx. 100 µs
CPU 486-3	approx. 30 µs	approx. 40 µs	approx. 80 µs
CPU 488-3	approx. 15 µs	approx. 25 µs	approx. 60 µs
FM 356-4	approx. 80 µs	approx. 85 µs	approx. 100 µs
FM 456-4	approx. 55 µs	approx. 65 µs	approx. 90 µs

Interrupt Reaction Time

Module	Interrupt Reaction Time		
	min.	typ.	max.
CPU 388-4	approx. 8 µs	approx. 12 µs	approx. 30 µs
CPU 486-3	approx. 5 µs	approx. 8 µs	approx. 15 µs
CPU 588-3	approx. 4 µs	approx. 7 µs	approx. 12 µs
FM 356-4	approx. 8 µs	approx. 12 µs	approx. 30 µs
FM 456-4	approx. 6 µs	approx. 10 µs	approx. 25 µs

Rules for Interrupt Handlers

When deciding whether a routine should be programmed as an interrupt handler or a task, you should consider the following (see also chapter 4.17):

1. Interrupt handler must node exceed a certain run time, since it can affect the real-time behavior and the performance of the whole system. As a thumb rule, such routines must have a run time of max. 20 µs to 40 µs.
2. Interrupt handler are executed on system level. This means that no memory protection is active for this code, and erroneous programming may therefor cause the whole system to hang up.
3. No M7 API and RMOS API system calls must be used in interrupt handlers.

You must structure your C program so that hardware-oriented requests are serviced directly in the interrupt handler while the subsequent processing of the data is implemented in one or more tasks. A task can be assigned a priority according to the M7 SYS priority model (see chapter 4.4).

Note

Variations of the run time can occur because of the PC-compatible hardware architecture and especially because of the use of cache memory.

Glossary

A

Acknowledgement

Under → M7 RMOS32, acknowledgement refers to the associated M7 API call which a task issues to notify an M7 server that a → message which was sent was successfully received and evaluated.

Alarm

Message from an alarm-enabled signal module to an S7 CPU or to the → M7 (Interrupt) to notify that a signal change or fault has occurred in the module (→ process alarm, → diagnostic alarm, → TS alarm).

Alarm-enabled

A module which is capable of generating alarm messages.

Alarm-enabled Module

A module which is capable of generating an alarm.

Alarm Server

Part of the → M7 RMOS32 operating system which notifies a → task that an alarm has been received.

Analog Module

Analog modules are signal converters for analog process signals.

Analog input modules convert analog measurement parameters to digital values to allow them to be processed by an → M7 or a → S7 CPU.

Analog output modules convert digital values into analog control signals (voltages or current).

AT

Advanced Technology. AT refers to the PC standard of the 2nd generation, in other words PCs with an ISA bus (16 bit data and 24 bit address lines), Intel processor >= 80286, two interrupt controllers and a hard disk.

Automation Computer

M7 CPU or function module with an operating system that allows including standard PC software such as data base processing or operator interface software into the programmable controller.

B

BACKDIR

Environment variable. Specifies the name of the directory on the → onboard silicon disk or the → memory card or hard disk which is used for the → backup memory for → S7 objects

Backup Memory

The backup memory is used for long term storage of → S7 objects which were created locally on the FM. The storage is independent of the power supply of the → M7 (→ non-volatile).

During system startup, all S7 objects are copied from the backup memory into the working memory and activated.

In M7-300/400: The backup memory is implemented by a directory (→ BACKDIR) in the M7 file system.

BIOS

Basic Input Output System

BIOS is that part of the → software which provides the connection between the hardware and the drivers of the operating system. BIOS makes the computer's software less dependent on the type of hardware being used; the BIOS software is contained in an EPROM.

Important components of the BIOS are for example the loader for the boot sector and the hardware setup program to configure the hardware and to adjust the clock.

Bus Segment

→ Local bus segment

C

CLI

CLI (abbreviation for Command Line Interpreter) implements a user environment for the → M7 RMOS32 operating system. The CLI allows commands to be entered and programs to be started and operated interactively.

C Library

→ C runtime library

Communication Bus

→ K bus

Communication via Configured Connections

With the communication functions for configured connections, you can exchange data between an S7/M7 CPU/FM and other modules with communication functionality, if the communication partners are connected via K bus or MPI. You can either read or write data of another module or send data to and receive data from a communication partner.

Additionally the operating mode of the communication partner can be interrogated and changed. These communication functions require connections configured with the SIMATIC Manager.

Communication via Configured Connections

With the communication functions for non-configured connections, you can exchange data between an S7/M7 CPU/FM and other modules with communication functionality, if the communication partners are connected to the same MPI subnet or if they belong to the same S7/M7 station.

These communication functions don't require connections configured with the SIMATIC Manager as opposed to the communication functions for configured connections.

Configuration

Within the context of PLC hardware, configuration is the interconnection of separate modules to assemble the required automation system.

C Program

→ C user program

CPU

Central Processing Unit. In this manual, the term CPU is used to describe the CPU module of an S7/M7 programmable controller. The term CPU is synonymous for S7 CPU and/or M7 CPU.

In this manual, CPU does not refer to the CPU chip contained in each module.

C Runtime Library

The C runtime library provides all C functions which are defined in the ANSI draft standard ISO/IEC DIS 9899 to→ M7 RMOS32 user programs.

C User Program

In the present manual, C user program is also abbreviated to C program (→ M7 RMOS32 user program, → MS DOS user program).

Cycle Control Point

The cycle control point is that point in time within a CPU cycle when → S7 objects are → linked or unlinked by the → OMS.

D

Debugger

→ Remote debugger

Default Configuration

The default configuration is a carefully chosen basic configuration which is always used when no other value has been specified. In the case of the S7/M7-300, a special application of the default configuration is the assignment of the → logical addresses of the signal modules → geographical addresses.

Diagnostic Alarm

In the case of signal modules which are capable of generating alarms (→ alarm-enabled signal modules), a diagnostic alarm is generated if a fault occurs, for example a wire break (→ alarm).

Diagnostic Buffer

The diagnostic buffer is a memory area on a → M7 or S7 CPU. The diagnostic buffer stores all → diagnostic events in the order of their occurrence. The diagnostic buffer is part of the → SSL.

Diagnostic-enabled

A module which is capable of generating diagnostic messages.

Diagnostic Event

Diagnostic events are for example faults on a → module, system faults of the CPU module, corresponding messages from a user program or operating mode transitions.

Diagnostic events are stored in the → diagnostic buffer and/or sent automatically to registered communication participants. The → diagnostic server is responsible for managing diagnostic events.

Diagnostic Functions

The diagnostic functions encompass the entire system diagnostics. This includes the detection, evaluation and notification of faults within the automation system.

Diagnostic Server

→ M7 server to manage → diagnostic events.

Digital Module

Digital modules are signal converters for binary process signals

Dynamic Parameters

In contrast to → static parameters, dynamic parameters of modules can be changed by the → M7 RMOS32 user program during program runtime.

E

Event Flag

A → resource which is provided by the → M7 RMOS32 kernel. Event flags can be used to coordinate several tasks and/or to exchange binary messages.

F

FC Server

→ M7 server in the → M7 RMOS32 operating system. The FC server (free cycle server) allows all tasks to synchronize themselves with the free cycle. It also provides the → cycle control point and monitors the maximum cycle time.

File

A file contains a set of data required for a particular task or for a particular reason. Files are stored on → mass storage.

FM

Abbreviation for function module. An M7 programmable controller can be operated as a function module or as a CPU. When operating as an FM, from the CPU module's viewpoint it behaves like an I/O module with associated → user data → parameter records.

At the same time, the FM can access I/O modules through its own → local bus segment.

Free Cycle

The free cycle is used to simulate the timing sequence of a cyclic S7 user program (OB1) on an M7 programmable controller. At the start of the free cycle, the process input images are updated and the registered tasks are notified. The process output image is transferred to the I/O modules at the end of the free cycle after all registered tasks have notified end of processing.

The → FC server is responsible for updating the process images and notifying the tasks. All tasks which participated in the free cycle have the same consistent view of the process signals.

Function Request Block (FRB)

Each "link" call to an → M7 server (for example alarm server) is assigned to an FRB. Storage for the FRB must be provided by the M7 RMOS32 user program in the global data or on the heap.

M7 servers use the FRB to store data which arises during processing and/or to return the results of processing to the user program.

G

Geographical Address

The geographical address is the physical assignment of an I/O signal in the PLC system. The geographical address is made up of the rack, slot and channel number.

The geographical address is only used by the system software. User programs use → logical addresses to access I/O signals. The assignment of the → logical base address to the geographical address is dependent on the default configuration but can be changed with configuration software.

H

Hardware

Hardware encompasses all of the physical and technical equipment of the automation system.

Heap

The heap is a global memory area which is managed by the → M7 RMOS32 kernel. → M7 RMOS32 user programs can request memory blocks from the heap during runtime and/or can return memory blocks which are no longer required. The heap is also called the global heap.

Hungarian Notation

Convention for variable and function names within a C program. The naming convention specifies both data type and usage, and was called Hungarian notation to honor the Hungarian Charles Simonyi.

I

Interrupt

Interrupt is the term for a request to interrupt program processing by an external event (for example → alarm). An interrupt request can be triggered for example by a hardware interrupt from an interface chip.

IR Alarm

Insert/Remove alarm. In the System S7/M7-400, the correct function of all modules is monitored cyclically. If a module is removed or inserted, this event is notified by the → alarm server to all registered → tasks.

ISA Bus

Industrial Standard Architecture bus with 16 bit data lines and 24 bit address lines which is used in PCs.

K

K Bus

The communication bus (K bus) is part of the backplane of the S7/M7-300/400 programmable logic controllers. It speeds up communication between programmable modules, the CPU and the programming device.

Key Switch

→ Operating mode switch

L

Linking

In the context of the M7 runtime environment, linking is the transfer of → S7 objects from the → temporary loading memory into the working memory. S7 objects are only active (in other words they can only be processed by → M7 RMOS32 user programs) after they are linked.

M7 RMOS32 user programs have no way of accessing unlinked (in other words passive) S7 objects.

In the context of software development, linking is a process which is closely connected with compiling source code.

Load Memory

→ Permanent load memory, → temporary load memory

Logical Address

The address used by a user program in a PLC system to access an I/O signal.

Logical Base Address

The logical address for the first I/O signal of a module.

Local Bus Segment

A bus segment is the contiguous (undivided) P bus in the S7-300 backplane bus of a module rack. If necessary, the bus can be logically separated after a function module (for example FM356-4). The part of the divided backplane bus from the function module to the end of the rack is called the local bus segment.

M

M7

A module from the family of automation computers for the systems SIMATIC S7-300 and SIMATIC S7-400. An M7 module is implemented with → AT architecture (PC standard) and is fully IBM compatible.

An M7 module can be either an → FM or an → M7 CPU.

M7 API

M7 API (Application Programming Interface) is the calling interface which an M7 RMOS32 user program uses to access the services of the → M7 server.

M7 CPU

M7 Central Processing Unit. M7 CPU is used to describe the → M7 if it implements the functions of the central processing module in an S7 automation system, rather than for example an FM.

The term M7 CPU is used in the present manual to differentiate it from other → CPUs (for example → S7 CPU).

M7 RMOS32

M7 RMOS32 is the 32 bit real-time multitasking operating system for the M7 automation computer. M7 RMOS32 consists of the → M7 RMOS32 kernel and the → M7 servers. It also includes utilities (RTI, RFS, CLI, debugger) and libraries (C runtime library, Socket library).

M7 RMOS32 Kernel

The kernel of M7 RMOS32 implements the basic services for task management, task coordination and communication and resource management. A C user program can access the services of the M7 RMOS32 kernel using the → RMOS API.

M7 RMOS32 Task

→ Task which execute under the operating system → M7 RMOS32. This term is sometimes abbreviated to RMOS tasks in programming manuals.

M7 RMOS32 User Program

A C program which executes under the → M7 RMOS32 operating system and accesses the operating system services using the RMOS API and M7 API.

An M7 RMOS32 program generally consists of several tasks and is started and managed with the CLI as a job.

M7 Server

System program of → M7 RMOS32 that provides particular M7 API services and processes requests from the tasks.

An M7 server provides the function calls which are used by an → M7 RMOS32 user program for example to access process I/Os or to communicate with other SIMATIC modules.

M7-SYS RT

System software for M7-300/400.

Mailbox

A mailbox is used to store messages which are sent by M7 RMOS32 tasks. A single mailbox can be shared between several sender and receiver tasks.

Accordingly, a mailbox can be used to implement data exchange between several sender tasks and several receiver tasks.

Main Memory

Main memory is all of the RAM memory of an → M7 which can be directly addressed by the processor.

Main Task

Under M7 RMOS32, the main function of an → M7 RMOS32 user program is executed as a separate task (main task). The main task is automatically registered with the → M7 RMOS32 kernel by the command line interpreter → CLI when the program is started.

Mass Storage

Memory for long term → non-volatile storage of larger amounts of data. In contrast to the → main memory, the mass storage cannot be directly addressed by the processor.

Typical types of mass storage in an M7 are → hard disc, → memory card and → OSD.

Memory Card

The memory card is a pluggable memory module for an M7 which can be used for → non-volatile storage of parts or all of the → software on the M7 and also for static data.

A memory card is accessed by the system in a similar way to a diskette; accordingly, another name for memory card is silicon disk.

Memory Pool

A memory area which can be assigned by an → M7 RMOS32 user program using RMOS API calls. During runtime, an M7 RMOS32 user program can then request dynamic memory blocks from the memory pool or return blocks which are no longer required.

Message

Predefined data structure that a → task or an M7 server can send to another task.

Message Queue

→ Memory area which is assigned to a → task for buffering → messages that are received. Each message is buffered in the message queue until it is read by the receiver task.

A message queue can be used to implement data exchange between several sender tasks and one receiver task.

Module

Modules are printed circuit boards fitted with digital chips which are plugged into an S7/M7 programmable controller.

Module Parameters

Module parameters are values which can be configured to change the behavior of the module. A differentiation is made between static and dynamic module parameters.

MPI

Multi point Interface for communication between S7 systems, → PGs and → operator interface systems.

MS DOS

Microsoft **D**isk **O**perating **S**ystem, an operating system from the Microsoft company.

Under M7 RMOS32, the MS DOS operating system and an MS DOS program can execute as a separate lower priority M7 RMOS32 task.

MS DOS Program

Program which executes under the operating system MS DOS.

Multitasking Operating System

Operating system that allows the pseudo-parallel execution of several → tasks.

Typical example of multitasking operating systems are UNIX, Window 3.x, Windows 95 and M7 RMOS32.

N

Non-volatile Data

Non-volatile (retentive) data is data which does not get erased if the power supply is turned off. Non-volatile data can be stored on mass storage or in static RAM.

O

OD Signal

The OD signal (output disable) can be output by an S7/M7 to the process I/Os to disable the output signal drivers.

Signal output from the I/O modules to the process peripherals only takes place (again) when the OD signal has been disabled by the S7/M7.

OMS

The OMS (Object Management System) of an S7 CPU manages the local S7 objects. In an M7, this job is done by the S7 object server.

Communication functions for configured connections allow an M7 RMOS32 program to communicate with the OMS (or S7 object server) of a remote S7 CPU (or M7) in order to operate on remote S7 objects.

OMT Server

→ The OMT server is an → M7 server that interrogates the current → operating mode of the → M7 and can also change the mode on receiving a request through the → P bus, K bus, from the → operating mode switch or from a → M7 RMOS32 user program. The OMT server is also responsible for notifying a new operating mode or an operating mode transition to M7 RMOS32 user programs on request.

Onboard Silicon Disk

The onboard silicon disk (OSD) is a permanently installed mass storage device in an M7. It can be used for non-volatile storage of data.

Operator Interface System

Operator interface systems have access to data areas of an S7 CPU and/or to S7 objects of an M7 programmable controller. Operator interface systems are used to visualize process data and to allow an operator to control the plant.

Operating Mode

The SIMATIC S7 family of automation computers features the following five operating modes: RESET, STOP, STARTUP, HALT and RUN.

In the → M7 the operating modes are managed by the → OMT server. If a → M7 RMOS32 user program needs to know the current operating mode of the M7, it must register itself to receive notification from the OMT server or the → FC server (the latter only for STARTUP and RUN).

Operating Mode Switch

The operating mode switch of the → M7 is used to boot or reset the M7 and to select the required operating mode. The current operating mode is interrogated by the → OMT server and is notified to → M7 RMOS32 user programs on request.

The operating mode switch is provided with a removable key.

OSD

→ Onboard Silicon Disk

P

Parameter

A differentiation is made between two types of parameters:

- Parameters of a C function call,
- Parameters of a → parameter block.

A parameter for a C function call is an item of data or a → pointer to (address of) a data value or field of the operands to be processed.

A parameter of a parameter block influences the behavior (configuration) of a module.

Parameter Block

A parameter block is a group of logically-connected parameters (system diagnostics, retentive memory parameters, etc.).

The parameters of a parameter block are accessed and configured with the SIMATIC Manager.

Parameterization

Within the context of the modules of an automation computer, parameterization is adjusting the behavior of the → module.

Parameter Record

A parameter record is used to store configuration data when → configuring a module.

A parameter record has a maximum length of 240 bytes and is uniquely identified by a number.

Permanent Load Memory

The permanent loading memory is used for long term storage of → S7 objects which were loaded into the M7 using communication functions for configured connections. The storage is independent of the power supply of the → M7 (“non-volatile”).

During system startup, all S7 objects are copied from the permanent load memory into the working memory and activated.

The permanent load memory is implemented by a directory (→RAMDIR) in the M7 file system.

PG

Programming device (programmer) or development system. A PG is a computer similar to a PC, and is used to develop automation software and to control the operator environment (→ CLI) of the → M7.

PLC

→ Programmable logic controller

PMC Functions

PMC functions (Programmed Module Communication) → Communication via configured/non-configured connections.

Pointer

A pointer is a special variable in a C program to store an address.

Priority

Under → M7 RMOS32, the term priority is used in the following situations:

1. The priority that you assign to a → **task** affects whether and how often the task is interrupted; higher priority tasks are able to interrupt lower priority tasks.
2. The priority that you assign to a → **message** affects the way that it is stored in a → message queue: higher priority messages are stored in the queue in front of the lower priority messages; messages of the same priority are stored in the queue in chronological order.

Process Alarm

A process alarm can be triggered by an alarm-enabled signal module after a certain event has arisen (→ alarm).

Process Image

A process image (process I/O image) is a memory area in the main memory of the → M7 which is used to store images of the input and output signals of the process I/Os. The address of the process image corresponds to the start of the I/O area.

Process I/O Driver

The process I/O driver allows a C program to access signals of the process I/Os through the → P bus. The process I/O driver also allows transparent communication with local process I/Os through the → ISA bus and with remote I/O modules through the → PROFIBUS DP Bus.

PROFIBUS DP

PROFIBUS DP is the serial bus used to interconnect distributed I/O modules. It was specially developed by Siemens for field applications.

Programmable Logical Controller

A programmable logic controller (PLC) is an electronic control circuit whose automation function is stored as a software program. Accordingly, the configuration and wiring of the PLC are not dependent on the automation assignment.

The PLC is constructed as a computer; it consists of a CPU module with memory, I/O modules and an internal bus system. The I/O modules and the programming language are tailored to the needs of automation programs.

R

Rack

A rack is a mechanical construction which contains slots (connectors) for modules.

RAMDIR

Environment variable. Specifies the name of the directory on the → onboard silicon disk or the → memory card or hard disk which is used for the → permanent loading memory for → S7 objects.

Real-time Operating System

An operating system that allows tasks to be carried out in a specified time. Real-time operating systems guarantee a maximum reaction time to external and internal events and allow prediction of the time required for any particular (→ tasks).

Remote Debugger

Debugger with a remote component which is installed on the M7 programmable controller, in other words on the M7, in order to test an M7 RMOS32 user program running on the M7 with the help of a development system (PG/PC).

Under M7 RMOS32, the connection between M7 and the development system takes place through MPI.

Resources

Resources (BM) are all software structures which are made available by → M7 RMOS32 to → M7 RMOS32 user programs to allow them to fulfill their automation assignment.

A user program can request the M7 RMOS32 kernel to create, catalog and manage resources. Typical resources are → tasks, → semaphores, → mailboxes, → event flags, → message queues, → memory pools etc.

RFS

Remote file system (RFS). The system program RFS is used to map drives of the mass storage on the M7 to local drives on the development system.

RMOS

RMOS (Real-time Multitasking Operating System) is a 32 bit, → real-time, → multitasking operating system developed by Siemens. The M7 RMOS32 kernel mainly consists of the RMOS kernel.

RMOS API

Calling interface for → M7 RMOS32 programs to allow them to access the services of the → M7 RMOS32 kernel.

RMOS Tasks

→ M7 RMOS32 tasks.

ROM

ROM is used to store those → S7objects which are required to recreate the original status of an M7 programmable controller. The original status is created with the RESET function.

The backup memory is implemented by a directory (→ ROMDIR) in the M7 file system.

ROMDIR

Environment variable. Specifies the name of the directory on the → onboard silicon disk or the → memory card or hard disk which is used for the → constant memory for → S7 objects.

RTI

Remote terminal Interface (RTI). The system program RTI is used to establish an MPI connection between the programming device or PC and the M7 programmable controller. This connection can then be used for example to control the operator environment (→ CLI) of the M7.

S

S7 CPU

Central Processing Unit of an S7 automation system.

The term S7 CPU is used in the present manual to differentiate it from other → CPUs (for example → M7 CPU).

S7 Object

S7 objects on an M7 represent the operand area of an S7 CPU. S7 objects allow an M7 to communicate transparently with other SIMATIC components (for example operator interface system, PG, S7 CPU).

S7 objects are created and managed by the → S7 object server. S7 objects on an M7 are for example all process data, data blocks or flag areas.

S7 Object Server

→ M7 server for managing → S7 objects. The S7 object server provides → M7 API calls for creating, storing, deleting and accessing S7 objects.

Segment

A (bus) segment is the contiguous (undivided) P bus of a module rack. A differentiation is made between:

- The undivided bus segment between the → CPU and the → FM, and
- The local bus segment, in other words a logically separate segment of the bus between the → FM and the end of the module rack.

A segment is also a specified part of the process input image or process output image.

Semaphore

Synchronization mechanism for coordinating shared access of tasks to a resource which requires exclusive usage (for example a common memory area). Tasks accessing the shared resource can coordinate their functional sequence by reserving and releasing semaphores.

Server

→ M7 server

Signal Module

Signal modules provide the connection between an S7 CPU or M7 and the sensors and actuators of the process being controlled. Signal modules can be either:

- Digital I/O modules,
- Analog I/O modules.

SIMATIC Manager

The graphical user interface of STEP 7.

Simulator Module

A simulator module is a module

- that can simulate digital input signals using operator elements, or
- can display digital output signals.

Single-tasking Operating System

An operating system which only allows a single task (C program) to be executed at any one time. A typical example of a single-tasking operating system is → MS DOS.

Software

All programs which can be executed on a computer are called software. This includes both the operating system and the user programs.

SSL

SSL (System Status List) contains all available information on the current status of a → S7 CPU or an → M7. The SSL can be interrogated by a communication partner (PG, S7 CPU or M7) using communication functions for configured connections.

Stack (of a Task)

The stack of a task is automatically assigned by the → M7 RMOS32 kernel when the task is created. The stack size is specified as a parameter.

The stack is used to store local task variables and return addresses of function calls.

Start Address

Basic address used to determine all I/O addresses.

STARTUP

The → operating mode STARTUP is the transition from the STOP mode to the RUN mode. Among other things, the STARTUP mode is responsible for initializing the program modules which are responsible for process control in the RUN mode.

Static Parameters

In contrast to → dynamic parameters, static parameters of modules can be assigned using the SIMATIC Manager.

STEP 7

STEP 7 is the programming software for the SIMATIC S7/M7/C7.

System Clock

Smallest interval of time in an operating system or CPU. All other times are programmed as integer multiples of the system clock.

T

Task

Part of a C program which processes a particular function and which is known to the → M7 RMOS32 kernel as a separate component of the program.

A task is the smallest unit of execution in a → multitasking operating system, in other words a task can be started and stopped independently of other tasks.

Task ID

Identification of a task which is assigned when creating the task and must be specified for example when sending a message to the task.

Task Stack

→ Stack

Temporary Load Memory

The temporary load memory is used as a temporary store for → S7 objects which are loaded into the → main memory through the → K bus or from → mass storage and activated (in other words → linked) by the object management system (→ OMS).

Time Server

→ M7 server under → M7 RMOS32 for processing date and time and time events.

Transfer Buffer

Memory area below the 1 Mbyte limit which is used to exchange data between an → M7 RMOS32 program and an → MS DOS program.

TSR Program

A TSR program is an MS DOS program which remains resident in working memory after termination (TSR=terminate and stay resident).

U

User Data

User data can be exchanged between a CPU and a signal module, a function module or a communication processor via process image or via direct access. User data can be digital and analog input/output signals from signal modules or monitor and modify information from function modules.

W

Working Memory

Working memory is part of the → main memory of the → M7. Working memory is used to process S7 objects. An M7 RMOS32 user program can only access those S7 objects which are currently stored in working memory.

Index

D

E

F

G

H

N

O

P

R

S

T

U

W

Siemens AG
A&D AS E48
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:

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _

_ _