High Performance
Real-Time Operating Systems

Device Driver

User's and Reference Manual

Copyright

Disclaimer

Trademark

Headquarters

SCIOPTA Systems AG
Fiechthagstrasse 19
4103 Bottmingen
Switzerland
Tel. +41 61 423 10 62
Fax +41 61 423 10 63
email: sales@sciopta.com
www.sciopta.com

Document No. S10096RL1

Table of Contents

1 SCIOPTA System

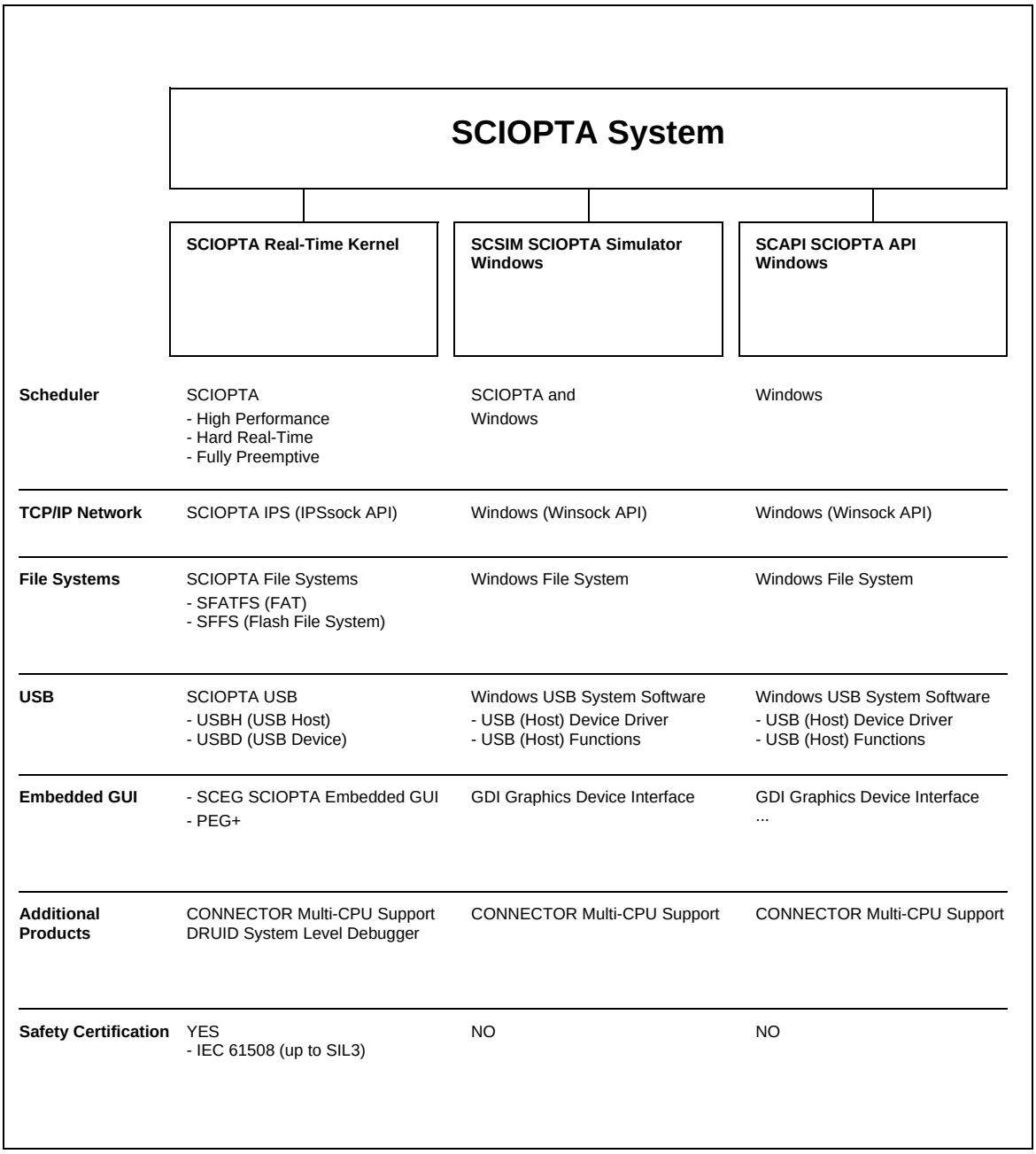


Figure 1-1: The SCIOPTA System

1.1 The SCIOPTA System

1.1.1 SCIOPTA System

SCIOPTA System is the name for a SCIOPTA framework which includes design objects such as SCIOPTA modules, processes, messages and message pools. SCIOPTA is designed on a message based architecture allowing direct message passing between processes. Messages are mainly used for interprocess communication and synchronization. SCIOPTA messages are stored and maintained in memory pools. The kernel memory pool manager is designed for high performance and memory fragmentation is avoided. Processes can be grouped in SCIOPTA modules, which allows you to design a very modular system. Modules can be static or created and killed during run-time as a whole. SCIOPTA modules can be used to encapsulate whole system blocks (such as a communication stack) and protect them from other modules in the system.

1.1.2 SCIOPTA Real-Time Kernels

SCIOPTA System Framework together with specific SCIOPTA scheduler results in very high performance real-time operating systems for many CPU architectures. The kernels and scheduler are written 100% in assembler. SCIOPTA is the fastest real-time operating system on the market. The SCIOPTA architecture is specifically designed to provide excellent real-time performance and small size. Internal data structures, memory management, interprocess communication and time management are highly optimized. SCIOPTA Real-Time kernels will also run on small single-chip devices without MMU.

1.1.3 SCIOPTA Simulator and API for Windows

The **SCIOPTA System** is available on top of Windows. **SCSIM** is a SCIOPTA simulator including SCIOPTA scheduling together with the Windows scheduler. This allows realistic system behaviour in a simulated environment.

SCAPI is a SCIOPTA API allowing message passing in a windows system. SCAPI is mainly used to design distributed systems together with CONNECTOR processes. Scheduling in SCAPI is done by the underlying operating system.

1.2 About This Manual

The purpose of this **SCIOPTA - Real-Time Kernel, User´s Manual** is to give all needed information how to use SCIOPTA Real-Time Kernel in an embedded project. Also the **SCIOPTA SCSIM Simulator** product is covered in this manual.

After a description of the installation procedure, detailed information about the technologies and methods used in the SCIOPTA Kernel are given. Descriptions of Getting Started examples will allow a fast and smooth introduction into SCIOPTA. Furthermore you will find useful information about system design and configuration. Also target specific information such as an overview of the system building procedures and a description of the board support packages (BSP) can be found in the manual.

Please consult also the **SCIOPTA - Kernel, Reference Manual** which contains a complete description of all system calls and error messages.

1.3 Supported Processors

1.3.1 Architectures

SCIOPTA - supports the following processor architectures. Architectures are referenced in this document as **<arch>**):

- arm
- ppc (power pc)
- coldfire
- win32 (SCIOPTA SCSIM Simulator and SCAPI SCIOPTA API, Windows Host)

1.3.2 CPU Families

SCIOPTA - Kernel supports the following CPU families. CPU families are referenced in this document as **<cpu>**):

Architecture **<arch>**	CPU Family **<cpu>**	Description
arm	at91sam7	**Atmel AT91SAM7 (ARM7)** Atmel AT91SAM7S, AT91SAM7SE, AT91SAM7X, AT91SAM7A3 and all other derivatives of the Atmel AT91SAM7 family.
	at91sam9	**Atmel AT91SAM9 (ARM9)** Atmel AT91SAM9260, AT91SAM9261, AT91SAM9263 and all other derivatives of the Atmel AT91SAM9 family.
	lpc21xx	**NXP LPC21xx/22xx (ARM7)** NXP LPC21xx and NXP LPC22xx and all other derivatives of the NXP LPC21xx/22xx family.
	lpc24xx_lpc23xx	**NXP LPC23xx/24xx (ARM7)** NXP LPC21xx and NXP LPC22xx and all other derivatives of the NXP LPC21xx/22xx family.
	str7	**STMicroelectronics STR710 (ARM7)** STMicroelectronics STR71x and all other derivatives of the STMicroelectronics STR710 family.
	str9	**STMicroelectronics STR910 (ARM9)** STMicroelectronics STR91x and all other derivatives of the STMicroelectronics STR910 family.
	stm32	**STMicroelectronics STM32 (ARM Cortex M3)** All derivatives of the STMicroelectronics STM32 family.

Architecture <arch>	CPU Family <cpu>	Description
arm	imx27	**Freescale i.MX2x (ARM9)** Freescale i.MX21, i.MX23, i.MX25, i.MX27 and all other derivatives of the Freescale i.MX2x family.
	imx35	**Freescale i.MX3x (ARM1136JF)** Freescale i.MX31, i.MX35, i.MX37 and all other derivatives of the Freescale i.MX3x family.
	stellaris	**Texas Instrument Stellaris (ARM Cortex M3)** All derivatives of the Texas Instrument Stellaris family.
	tms570	**Texas Instrument TMS570 (ARM Cortex R4F)** All derivatives of the Texas Instrument TMS570 family.
	stm32	**STMicroelectronics STM32 (ARM Cortex M3)** All derivatives of the STMicroelectronics STM32 family.
	pxa270	**Marvell PXA270 (XScale)** All derivatives of the Marvell PXA270 family.
	pxa320	**Marvell PXA320 (XScale)** All derivatives of the Marvell PXA320 family.
ppc	mpx5xx	**Freescale PowerPC MPC500** MPC53x, MPC55x, MPC56x and all other derivatives of the Freescale MPC500 family.
	mpc5500	**Freescale PowerPC MPC55xx** MPC5516, MPC5534, MPC5554, MPC5567 and all other derivatives of the Freescale MPC55xx family.
	mpc8xx	**Freescale PowerPC PowerQUICC I** MPC823, MPC850, MPC852T, MPC855T, MPC857, MPC859, MPC860, MPC862, MPC866 and all other derivatives of the Freescale MPC8xx family.
	mpc82xx	**Freescale PowerPC PowerQUICC II** MPC8250, MPC8255, MPC8260, MPC8264, MPC8265, MPC8266 and all other derivatives of the Freescale MPC82xx family.
	mpc83xx	**Freescale PowerPC PowerQUICC II Pro** MPC8313, MPC8314, MPC8315 and all other derivatives of the Freescale MPC83xx family.
	mpc52xx	**Freescale PowerPC MPC5200** MobileGT MPC5200 and all other derivatives of the Freescale MPC52xx and 51xx family.
	ppc4xx	**AMCC PowerPC 4xx** PowerPC 405, 440, 460 and all other derivatives of the AMCC PowerPC 4xx family.

Architecture <arch>	CPU Family <cpu>	Description
coldfire	mcf521x	**Freescale Coldfire MCF521x (V2)** MCF5213 and all other derivatives of the Freescale ColdFire MCF521x family.
	mcf523x	**Freescale Coldfire MCF523x (V2)** MCF5235 and all other derivatives of the Freescale ColdFire MCF523x family.
	mcf525x	**Freescale Coldfire MCF525x (V2)** MCF5253 and all other derivatives of the Freescale ColdFire MCF525x family.
	mcf532x	**Freescale Coldfire MCF532x (V3)** MCF5329 and all other derivatives of the Freescale ColdFire MCF532x family.
	mcf548x	**Freescale Coldfire MCF548x (V4e)** MCF5485 and all other derivatives of the Freescale ColdFire MCF548x family.
	mcf5223	**Freescale Coldfire MCF5223x (V2)** MCF52233 and all other derivatives of the Freescale ColdFire MCF5223x family.
	mcf5272	**Freescale Coldfire MCF527x (V2)** MCF5272 and all other derivatives of the Freescale ColdFire MCF527x family.
	mcf5282	**Freescale Coldfire MCF528x (V2)** MCF5282 and all other derivatives of the Freescale ColdFire MCF528x family.
	mcf54455	**Freescale Coldfire MCF54455x (V4)** MCF54455 and all other derivatives of the Freescale ColdFire MCF54455x family.
win32	---	**For Windows based PCs and workstations**

2 Installation

Please consult chapter 2 Installation of the SCIOPTA - Kernel, User's Guides for a detailed description and guidelines of the SCIOPTA installation.

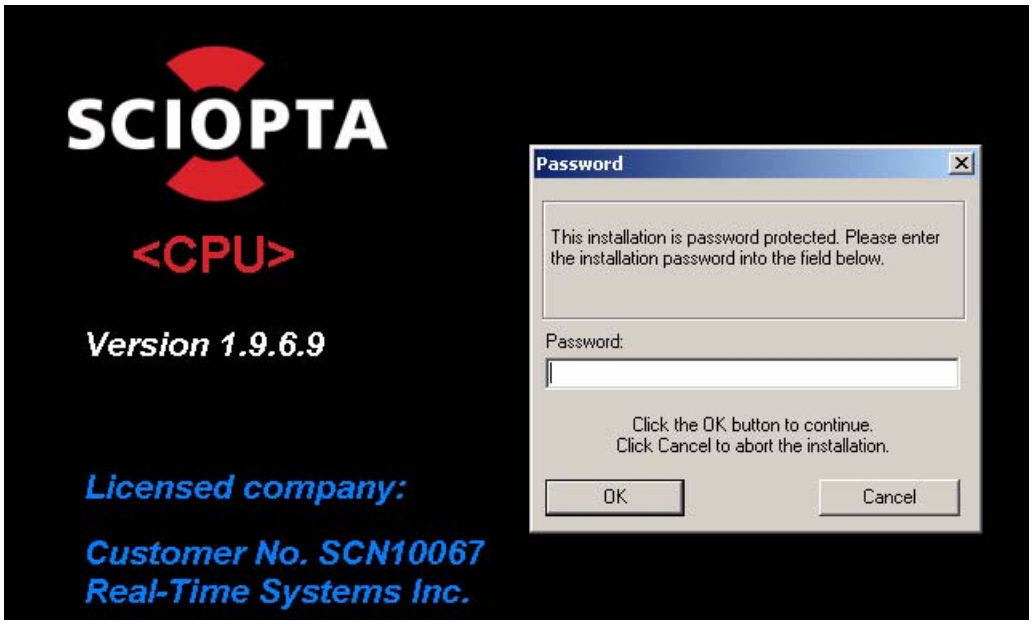


Figure 2-1: Main Installation Window

3 SCIOPTA Device Driver Concept

3.1 Overview

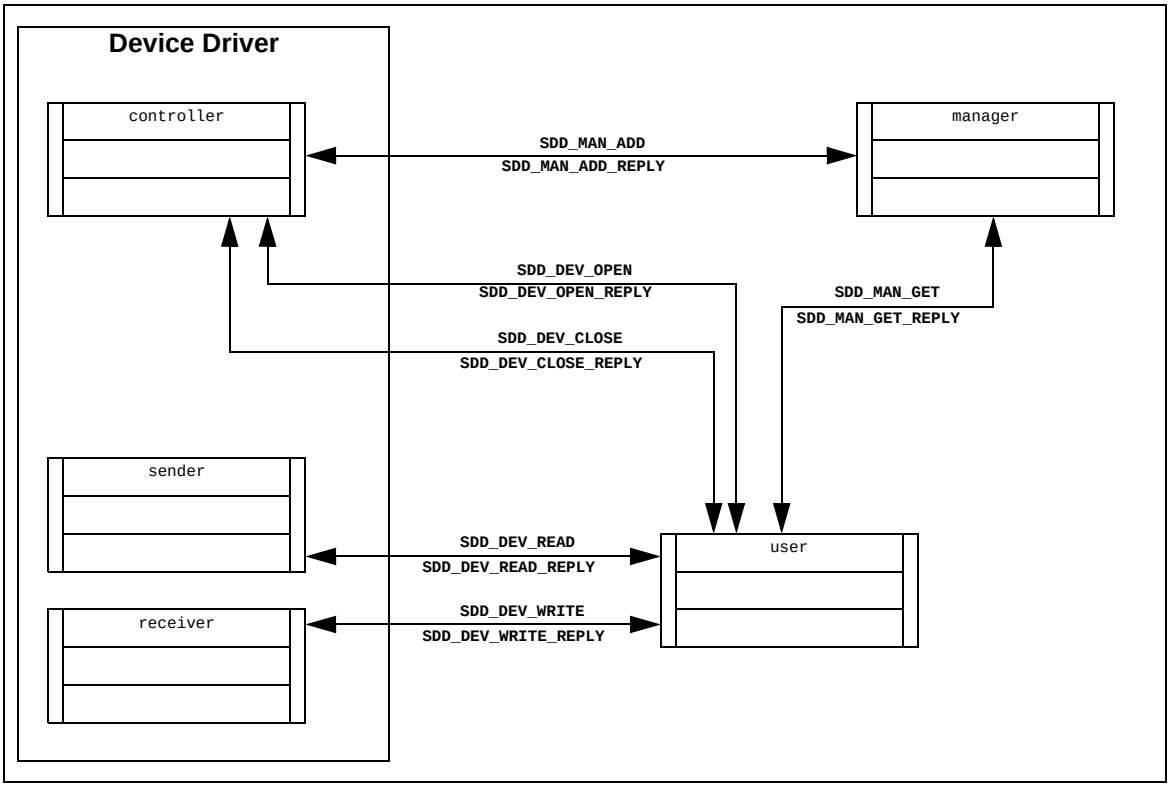


Figure 3-1: Process-Message Diagram Device Driver Concept

Devices are managed in device manager processes which maintain a device data base. In a SCIOPTA system there can be more than one device manager processes.

A standard SCIOPTA devices driver consists of at least one process. For more complex devices there is often a controller, a sender and receiver process handling device control and data receiving and transmitting. Additionally SCIOPTA interrupt processes are implemented to handle the device interrupts.

The user process is getting information about the device from the device manager and communicates directly with the device processes.

SCIOPTA is a message based system all communication is done by SCIOPTA messages. But there is also a function interface interface available.

3.2 SDD Objects

SDD objects are specific system objects in a SCIOPTA real-time operating system such as:

SDD devices and SDD device drivers	Objects and methods controlling I/O devices
SDD managers	Objects and methods managing other SDD objects. SDD managers are maintaining SDD object databases. There are for instance **SDD device managers** which managing **SDD devices** and **SDD device drivers** and **SDD file managers** which are managing **files** in the SCIOPTA SFS file system.
SDD protocols	Objects and methods representing network protocols such as SCIOPTA IPS TCP/IP internet protocols.
SDD directories and files	Objects and methods representing files and directories in the SCIOPTA SFS file system.

3.2.1 SDD Object Descriptors

SDD object descriptors are data structures in SCIOPTA containing information about **SDD objects**.

SDD object descriptors are stored in standard SCIOPTA messages. Therefore, SDD object descriptors contain a message ID structure element.

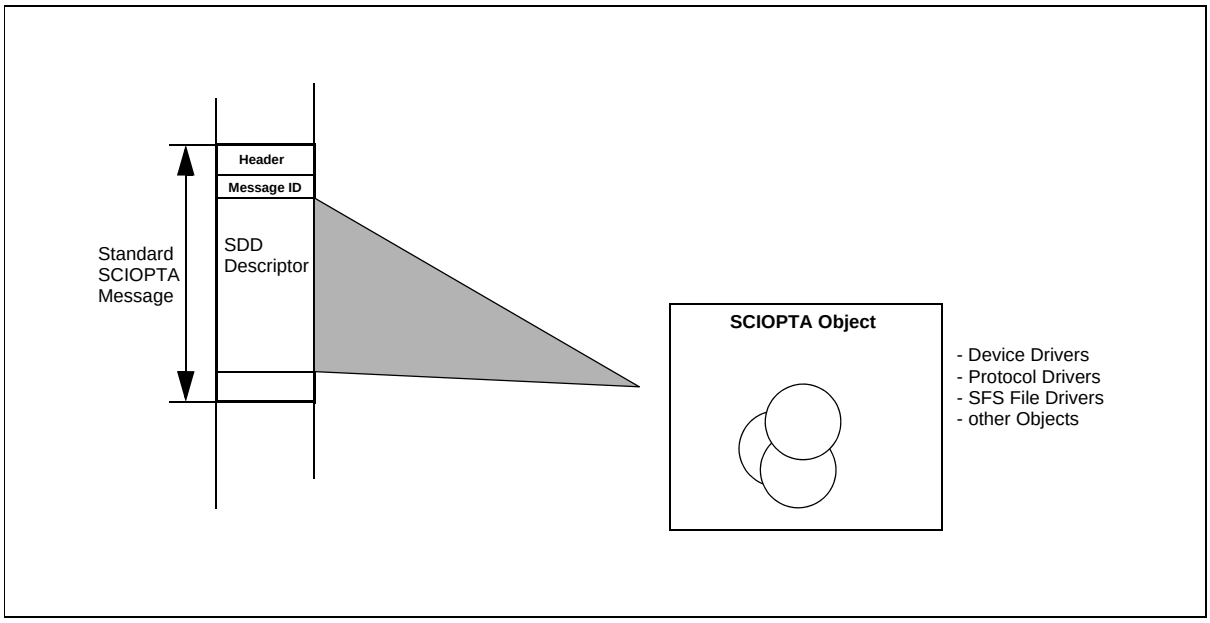


Figure 3-2: SDD Descriptor

3.2.2 Specific SDD Object Descriptors

- **SDD device descriptors** contain information about **SDD devices**.
- **SDD device manager descriptors** contain information about **SDD device managers**.
- **SDD network device descriptors** contain information about **SDD network devices**.
- **SDD protocol descriptors** contain information about **SDD protocols**.
- **SDD file manager descriptors** contain information about **SDD file managers**.
- **SDD file device descriptors** contain information about **SDD file devices**.
- **SDD directory descriptors** contain information about **SDD directories**.
- **SDD file descriptors** contain information about **SDD files**.

Please consult chapter **5 "Structures" on page 5-1** for more information about SDD object descriptor structures.

3.3 Registering Devices

Before a device can be used it must be registered. The device driver needs to register the device directly at the device manager.

Using the SCIOPTA message interface this is done by sending a **SDD_MAN_ADD** message.

Using the SCIOPTA function interface, the **sdd_manAdd** function will be used by the device driver to register the device to a manager.

The device manager will enter the device in its device database.

3.4 Using Devices

User processes will communicate directly with device drivers.

Before a user process can communicate with a device it must get the device descriptor from the device manager.

Using the SCIOPTA message interface the user process can request a device by sending a **SDD_MAN_GET** message to the device manager which responds with a **SDD_MAN_GET_REPLY** message. This reply message contains the device descriptor with full information about the device including:

- Process IDs of all processes (controller, sender and receiver)
- Device handle (pointer to a data structure of the device which holds additional device information such as unit numbers etc.)

Using the SCIOPTA function interface the device descriptor can be retrieved from the device manager with the **sdd_manGetByName** function. The return value of this function is the pointer to the device descriptor including the same full information about the device as above.

The User Process can now communicate to the device driver using **SDD_DEV_XXX** messages or **sdd_devXxx** functions.

3.5 Device Driver Application Programmers Interface

The SCIOPTA device driver system is based on the SCIOPTA message passing technology. You can access the device driver functionality by exchanging messages. This results in a very efficient, fast and direct way of working with SCIOPTA. An application programmer can use the SCIOPTA message passing to send and receive data for high speed communication.

The standard device driver API is a function layer on top of the message interface. The message handling and event control are encapsulated within these functions.

3.6 Hierarchical Structured Managers

In a SCIOPTA system there can be more than one manager and managers can be organized in a hierarchical structure. This can already be seen as the base of a file system. In a hierarchical manager organization, managers reside below the root managers and have a nested organization. Hierarchical organized manager systems are mainly used in file systems such as the SCIOPTA SFS.

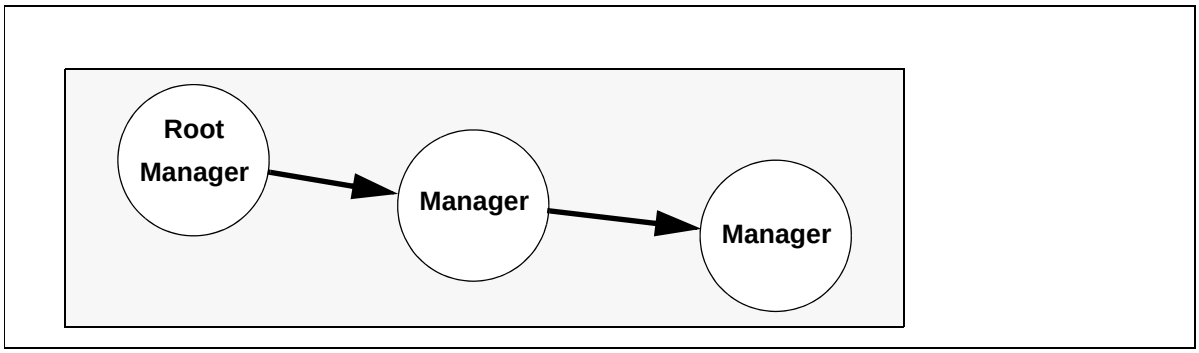


Figure 3-3: SCIOPTA Hierarchical Structured Managers

4 Data Structures

4.1 Base SDD Object Descriptor Structure

4.1.1 Description

The base SDD object descriptor structure is the basic component of all SDD object descriptors. It is inherited by all other specific SDD object descriptors and represents the smallest common denominator.

It contains the message ID (SDD object descriptors are SCIOPTA messages), an error variable and the handle of the SDD object.

4.1.2 sdd_baseMessage_t Structure

```
typedef struct sdd_baseMessage_s {
   sc_msgid_t              id;
   sc_errorcode_t          error;
   void                    *handle;
} sdd_baseMessage_t;
```

4.1.3 Structure Members

id Message ID.

Standard SCIOPTA message identity.

error Error code.

handle Handle of the SDD object.

This is usually a pointer to a structure which further specifies the SDD object.

The user of a device object which is opening and closing the device, reading from the device and writing to the device does not need to know the handle and the handle structure. The user will usually get the SDD device descriptor by using the **sdd_manGetByName** function call. The SDD device manager will return the SDD device descriptor including the handle.

Only processes inside the SDD object (the device driver) may access and use the handle.

4.1.4 Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

4.2 Standard SDD Object Descriptor Structure

4.2.1 Description

This structure contains more specific information about SDD objects such as types, names and process IDs. It is an extension of the base SDD object descriptor structure **sdd_baseMessage_t**.

4.2.2 sdd_obj_t Structure

```
typedef struct sdd_obj_s {
   sdd_baseMessage_t    base;
   void                 *manager;
   sc_msgid_t           type;
   unsigned char        name[SC_NAME_MAX + 1];
   sc_pid_t             controller;
   sc_pid_t             sender;
   sc_pid_t             receiver;
} sdd_obj_t;
```

4.2.3 Structure Members

base — Base SDD object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

manager — Manager access handle.

It is a pointer to a structure which further specifies the manager. This is only used if the SDD object descriptor describes an SDD manager and is only used in SDD manager messages (SDD_MAN_XXX). For SDD file managers a 0 defines an SDD root manager.

You do not need to write anything in the manager handle if you are using the function interface as this is done in the interface layer.

type — Type of the SDD object.

More than one value can be defined and must be separated by OR. The values determine the type of messages which are handled by the SDD object.

SDD_OBJ_TYPE — General SDD object type. Handles the following messages:
SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY
SDD_OBJ_DUPLICATE / SDD_OBJ_DUPLICATE_REPLY
SDD_OBJ_INFO / SDD_OBJ_INFO_REPLY

SDD_MAN_TYPE — The SDD object is an SDD manager. It handles the following manager messages:
SDD_MAN_ADD / SDD_MAN_ADD_REPLY
SDD_MAN_RM / SDD_MAN_RM_REPLY
SDD_MAN_GET / SDD_MAN_GET_REPLY
SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY
SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY

SDD_DEV_TYPE	The SDD object is an SDD device. It handles the following device messages: SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY SDD_DEV_DUALOPEN / SDD_DEV_DUALOPEN_REPLY SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY SDD_DEV_READ / SDD_DEV_READ_REPLY SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY
SDD_FILE_TYPE	The SDD object is an SDD file. It handles the following file messages: SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY
SDD_NET_TYPE	The SDD object is an SDD protocol or network device. It handles the following network messages: SDD_NET_RECEIVE / SDD_NET_RECEIVE_REPLY SDD_NET_RECEIVE_2 / SDD_NET_RECEIVE_2_REPLY SDD_NET_RECEIVE_URGENT / SDD_NET_RECEIVE_URGENT_REPLY SDD_NET_SEND / SDD_NET_SEND_REPLY
name	Name of the SDD object.
	The name must be unique within a domain. A manager corresponds to a domain.
controller	ID of the controller process of the SDD object.
sender	ID of the sender process of the SDD object.
	If the SDD object is a device driver, the sender process sends the data to the physical layer. It usually receives SDD_DEV_WRITE or SDD_NET_SEND messages and can reply with the corresponding reply messages.
receiver	ID of the receiver process of the SDD object.
	If the SDD object is a device driver, the receiver process receives the data from the physical layer. In passive synchronous mode the receiver process receives the SDD_DEV_READ messages and replies with the SDD_DEV_READ_REPLY message. In active asynchronous mode (used by network devices) the receiver process sends a SDD_NET_RECEIVE, SDD_NET_RECEIVE_2 or SDD_NET_RECEIVE_URGENT message.

Please Note: For specific or simple SDD objects the process IDs for controller, sender and receiver can be the same. These SDD objects contain therefore just one process.

4.2.4 Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

4.3 SDD Object Name Structure

4.3.1 Description

This structure contains a name string additional to the base object descriptor. It is an extension of the base SDD object descriptor structure **sdd_baseMessage_t**.

4.3.2 sdd_name_t Structure

```
typedef struct sdd_name_s {
   sdd_baseMessage_t    base;
   char                 name[SC_NAME_MAX + 1];
} sdd_name_t;
```

4.3.3 Structure Members

base	Base SDD object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.
name	Name of the SDD object.
	The name must be unique within a domain.

4.3.4 Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

4.4 SDD Object Size Structure

4.4.1 Description

This structure contains information about SDD object sizes. This can be cache sizes, file sizes or any sizes an object could have.

4.4.2 sdd_size_t Structure

```
typedef struct sdd_size_s {
   size_t               total;
   size_t               free;
   size_t               used;
   size_t               bad;
} sdd_size_t;
```

4.4.3 Structure Members

total	Total size of the SDD object.
free	Free available size of the SDD object.
used	Used size of the SDD object.
bad	Not available size of the SDD object.

4.4.4 Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

4.5 NEARPTR and FARPTR

Some 16-bit kernels need near and far pointer defines.

In 32-bit kernels this is just defined as a pointer type (*):

```
#define FARPTR   *
#define NEARPTR  *
```

This mainly to avoid cluttering up sources with #if/#endif.

These target processor specific data types are defined in the file **types.h** located in **sciopta\<cpu>\arch**.

File location: <install_folder>\sciopta\<version>\include\sciopta\<cpu>\arch.

This file will be included by the main type file **(types.h** located in **ossys**).

5 Messages

5.1 Introduction

SCIOPTA is a message based real-time operating system. Interprocess communication and coordination is done by messages. Message passing is a very fast, secure, easy to use and a good to debug method.

Messages are the preferred tool for interprocess communication in SCIOPTA. SCIOPTA is specifically designed to have a very high message passing performance. Messages can also be used for interprocess coordination or synchronization duties to initiate different actions in processes. For this purposes messages can but do not need to carry data.

A message buffer (the data area of a message) can only be accessed by one process at a time which is the owner of the message. A process becomes owner of a message when it allocates the message by the **sc_msgAlloc** system call or when it receives the message by the **sc_msgRx** system call.

Message passing is also possible between processes on different CPUs. In this case specific communication process types on each side will be needed called **SCIOPTA CONNECTOR Processes**.

In this chapter all SCIOPTA device driver messages are described. Only the generic device driver messages are listed.

Please consult the SCIOPTA IPS Internet Protocols, User's and Reference Manual for a description of the specific network device messages (**SDD_NET_XXX**).

Please consult the SCIOPTA File System, User's and Reference Manual for a description of the specific file system device messages (**SDD_FILE_XXX**).

The messages are listed in alphabetical order. The request and reply message are described together.

5.1.1 Error Code

The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

5.1.2 Header File

The messages are defined in the following header file:

<installation_folder>\sciopta\<version>\include\sdd\sdd.msg

5.2 SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY

5.2.1 Description

This message is used to close an open device. If a device is not used any more it should be closed by the user process.

The user process sends an **SDD_DEV_CLOSE** request message to the controller process of the device driver. The controller process sends an **SDD_DEV_CLOSE_REPLY** reply message back

If a device or an object is not needed any more you should send a release message (see 5.16 “SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY” on page 5-21) to the device or object. This is mainly used to clean temporary objects. Please consult chapter 7.6 “Temporary Objects” on page 7-5 for more information about temporary objects.

5.2.2 Message IDs

Request Message	**SDD_DEV_CLOSE**
Reply Message	**SDD_DEV_CLOSE_REPLY**

5.2.3 sdd_devClose_t Structure

```
typedef struct sdd_devClose_s {
   sdd_baseMessage_t           base;
} sdd_devClose_t;
```

5.2.4 Structure Members

base Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

5.2.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EIO	An input/output error occurred.
SC_ENOTSUPP	This request is not supported.

5.3 SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY

5.3.1 Description

This message is used to set or get specific parameters to/from device drivers on the hardware layer. It is mainly included here to be compatible with the BSD API.

The user process sends an **SDD_DEV_IOCTL** request message to the controller process of the device driver. The controller process sends an **SDD_DEV_IOCTL_REPLY** reply message back.

The size of the allocated **SDD_DEV_IOCTL** message must be big enough to contain the specific command. The user process needs to add this in addition to the size of the **sdd_devIoctl_s** structure at message allocation.

5.3.2 Message IDs

Request Message	**SDD_DEV_IOCTL**
Reply Message	**SDD_DEV_IOCTL_REPLY**

5.3.3 sdd_devIoctl_t Structure

```
typedef struct sdd_devIoctl_s {
   sdd_baseMessage_t            base;
   unsigned   int               cmd;
   int                          ret;
   unsigned   long              outlineArg;
   unsigned   char              inlineArg[1];
} sdd_devIoctl_t;
```

5.3.4 Structure Members

base — Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

cmd — Device specific command.

Used by the **request message** and contains a device specific command. Not modified by the device driver and therefore contains the same value in the **reply message**.

ret — Return value. Only used in the reply message.

<value>	In the **reply message** this member contains a driver specific value.
-1	An error was encountered.

outlineArg	Command specific argument.
<value>	Argument.
0	The member **inlineArg** is used.

inlineArg	Command specific included argument.
	Argument if **outlineArg** is not used. The size is variable and the whole argument is included.

5.3.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

5.4 SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY

5.4.1 Description

This message is used to open a device for read, write or read/write.

The user process sends an **SDD_DEV_OPEN** request message to the controller process of the device driver including the access type in the **flag** data. The controller process sends an **SDD_DEV_OPEN_REPLY** reply message back. The reply message contains the access handle of the device driver. This is the handle which must be used for all further device accesses.

5.4.2 Message IDs

Request Message	**SDD_DEV_OPEN**
Reply Message	**SDD_DEV_OPEN_REPLY**

5.4.3 sdd_devOpen_t Structure

```
typedef struct sdd_devOpen_s {
   sdd_baseMessage_t            base;
   flags_t                      flags;
} sdd_devOpen_t;
```

5.4.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.
flags	Device open flags.
	Used by the request message and contains BSD conform flags. Not modified by the device driver and therefore contains the same value in the request message.
O_RDONLY	Opens the device for read only. Cannot be ored with O_WRONLY.
O_WRONLY	Opens the device for write only.
O_RDWR	Opens the device for read and write.
O_TRUNC	Decrease a file to length zero. Can be ored with O_RDONLY and O_WRONLY.
O_APPEND	Sets the read/write pointer to the end of the file. Can be ored with O_RDONLY and O_WRONLY.

5.4.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
ENOMEM	Not enough memory to open.
SC_ENOTSUPP	This request is not supported.

5.5 SDD_DEV_READ / SDD_DEV_READ_REPLY

5.5.1 Description

This message is used to read data from a device driver. It can only be used if the device was first successful opened for read.

The user sends an **SDD_DEV_READ** request message to the device driver receiver process. The device driver receiver process replies with the **SDD_DEV_READ_REPLY** reply message which contains the read data.

For simpler devices with just one process the controller process can also act as device receiver process.

The size of the allocated **SDD_DEV_READ** message must be big enough to contain the requested size of the read data. The user process needs to add this in addition to the size of the **sdd_devRead_s** structure at message allocation.

5.5.2 Message IDs

Request Message	**SDD_DEV_READ**
Reply Message	**SDD_DEV_READ_REPLY**

5.5.3 sdd_devRead_t Structure

```
typedef struct sdd_devRead_s {
   sdd_baseMessage_t            base;
   ssize_t                      size;
   ssize_t                      curpos;
   unsigned char                *outlineBuf;
   unsigned char                inlineBuf[1];
} sdd_devRead_t;
```

5.5.4 Structure Members

base — Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

size — Size of message data buffer.

Contains in the **request message** the requested size of the read message buffer.
In the **reply message** this member contains the size of the message data buffer.

curpos — Index into the message buffer.

Contains the index of the last written byte in the reply message.
In most of the devices not used.

outlineBuf	Pointer to referenced read data.
<readptr>	Used by the **reply message** and can contain the pointer to the read data. Not recommended as pointers should not be used in messages. Rather use **inlineBuf.** Not used by the **request message** and can have any value.
0	The member **inlineBuf** is used.
inlineBuf	Included read data.
	Read data used by the **reply message** if **outlineBuf** is not used. The size is variable and the whole data is included. Not used by the **request message** and can have any value.

5.5.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EIO	An input/output error occurred.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

5.6 SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY

5.6.1 Description

This message is used to write data to a device driver. It can only be used if the device was first successful opened for write.

The user sends an **SDD_DEV_WRITE** request message which contains the data to be written to the device driver sender process. The device driver sender process replies with the **SDD_DEV_WRITE_REPLY** reply message.

For simpler devices with just one process the controller process can also act as device sender process.

The size of the allocated **SDD_DEV_WRITE** message must be big enough to contain the requested size of the data to write. The user process needs to add this in addition to the size of the **sdd_devWrite_s** structure at message allocation.

5.6.2 Message IDs

Request Message	**SDD_DEV_WRITE**
Reply Message	**SDD_DEV_WRITE_REPLY**

5.6.3 sdd_devWrite_t Structure

```
typedef struct sdd_devWrite_s {
   sdd_baseMessage_t            base;
   ssize_t                      size;
   ssize_t                      curpos;
   unsigned char                *outlineBuf;
   unsigned char                inlineBuf[1];
} sdd_devWrite_t;
```

5.6.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.
size	Size of message data buffer.
	Contains in the **request message** the size of the message buffer. In the **reply message** this member contains the number of written bytes.
curpos	Not used.

outlineBuf	Pointer to referenced data to be written.
<readptr>	Used by the **request message** and can contain the pointer to the read data. Not recommended as pointers should not be used in messages. Rather use **inlineBuf.** Not used by the **reply message** and can have any value.
0	The member **inlineBuf** is used.
inlineBuf	Included data to be written.
	Data to be written used by the **request message** if **outlineBuf** is not used. The size is variable and the whole data is included. Not used by the **reply message** and can have any value.

5.6.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EIO	An input/output error occurred.
EINVAL	Invalid parameter.
EFBIG	Size of data to be written to big.
SC_ENOTSUPP	This request is not supported.

5.7 SDD_ERROR

5.7.1 Description

This message is mainly used by device driver and other processes which do not use reply messages as answer of request messages for returning error codes.

Connector processes are using these messages for instance to inform users about not existing device drivers. The connector process sends an **SDD_ERROR** message to the user process.

In addition to receive the usual reply messages a user process should always also receive a possible **SDD_ERROR** message. This will inform the user of a not existent device.

5.7.2 Message ID

Message **SDD_ERROR**

5.7.3 sdd_error_t Structure

```
typedef struct sdd_error_s {
   sdd_baseMessage_t            base;
} sdd_error_t;
```

5.7.4 Structure Members

base Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

5.7.5 Errors

base.error Error code.

SC_ENOPROC Device or process does not exist.

5.8 SDD_MAN_ADD / SDD_MAN_ADD_REPLY

5.8.1 Desciption

This message is used to add a new device in the device driver system.

The device driver controller process sends an **SDD_MAN_ADD** request message to the manager process. The manager process registers the new device in its device database and replies with the **SDD_MAN_ADD_REPLY** reply message.

The **SDD_MAN_ADD** message contains the SDD device descriptor of the device to add to the manager.

5.8.2 Message IDs

Request Message	**SDD_MAN_ADD**
Reply Message	**SDD_MAN_ADD_REPLY**

5.8.3 sdd_manAdd_t Structure

```
typedef struct sdd_manAdd_s {
   sdd_obj_t                    object;
} sdd_manAdd_t;
```

5.8.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.

5.8.5 Errors

object.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EEXIST	Device already exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.9 SDD_MAN_GET / SDD_MAN_GET_REPLY

5.9.1 Description

This message is used to get the SDD device descriptor (including the process IDs and handle) of a registered device.

The user sends an **SDD_MAN_GET** request message to the SDD device manager. The device manager replies with the **SDD_MAN_GET_REPLY** reply message which contains all information about the device (including all process IDs).

The **SDD_MAN_GET_REPLY** message contains the device descriptor of the registered device.

5.9.2 Message IDs

Request Message	**SDD_MAN_GET**
Reply Message	**SDD_MAN_GET_REPLY**

5.9.3 sdd_manGet_t Structure

```
typedef struct sdd_manGet_s {
   sdd_obj_t                    object;
} sdd_manGet_t;
```

5.9.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.

5.9.5 Errors

object.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.10 SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY

5.10.1 Description

This message is used to get the device descriptor (including the process IDs and handle) of the first registered device from the SDD manager's device list.

The user sends an **SDD_MAN_GET_FIRST** request message to the device manager process. The device manager replies with the **SDD_MAN_GET_FIRST_REPLY** reply message which contains all information about the device (including all process IDs).

5.10.2 Message IDs

Request Message	**SDD_MAN_GET_FIRST**
Reply Message	**SDD_MAN_GET_FIRST_REPLY**

5.10.3 sdd_manGetFirst_t Structure

```
typedef struct sdd_manGetFirst_s {
   sdd_obj_t                    object;
} sdd_manGetFirst_t;
```

5.10.4 Structure Members

base	Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

5.10.5 Errors

object.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.11 SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY

5.11.1 Description

This message is used to get the device descriptor (including the process IDs and handle) of the next registered device from the SDD manager's device list.

The user sends an **SDD_MAN_GET_NEXT** request message to the device manager process. The device manager replies with the **SDD_MAN_GET_NEXT_REPLY** reply message which contains all information about the device (including all process IDs).

5.11.2 Message IDs

Request Message	**SDD_MAN_GET_NEXT**
Reply Message	**SDD_MAN_GET_NEXT_REPLY**

5.11.3 sdd_manGetNext_t Structure

```
typedef struct sdd_manGetNext_s {
   sdd_obj_t                  object;
} sdd_manGetNext_t;
```

5.11.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.

5.11.5 Errors

object.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.12 SDD_MAN_NOTIFY_ADD / SDD_MAN_NOTIFY_ADD_REPLY

5.12.1 Desciption

This message is used to request a message from the manager when a device or SDD object is registered.

The user sends an **SDD_MAN_NOTIFY_ADD** request message containing the device name to the device manager process. The device manager replies with the **SDD_MAN_NOTIFY_ADD_REPLY** reply message when the device has been registered by the manager.

The notify add request can contain a timeout.

5.12.2 Message IDs

Request Message	**SDD_MAN_NOTIFY_ADD**
Reply Message	**SDD_MAN_NOTIFY_ADD_REPLY**

5.12.3 sdd_manNotify_t Structure

```
typedef struct sdd_manNotify_s {
   sdd_name_t                   name;
   sc_ticks_t                   tmo;
   sc_tmoid_t                   tmoid;
} sdd_manNotify_t;
```

5.12.4 Structure Members

name.base Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

name.name Name of the device or SDD object.

Used in the request message to specify the name of the device.

tmo Time-out.

If the manager does not respond within this time-out it will reply with the **SDD_MAN_NOTIFY_ADD_REPLY** message containing an error code.

tmoid Internal use.

Used by the manager for its own purposes.

5.12.5 Errors

name.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EEXIST	Device already exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.13 SDD_MAN_NOTIFY_RM / SDD_MAN_NOTIFY_RM_REPLY

5.13.1 Desciption

This message is used to request a message from the manager when a device or SDD object is removed.

The user sends an **SDD_MAN_NOTIFY_RM** request message containing the device name to the device manager process. The device manager replies with the **SDD_MAN_NOTIFY_RM_REPLY** reply message when the device has been registered by the manager.

The notify remove request can contain a timeout.

5.13.2 Message IDs

Request Message	**SDD_MAN_NOTIFY_RM**
Reply Message	**SDD_MAN_NOTIFY_RM_REPLY**

5.13.3 sdd_manNotify_t Structure

```
typedef struct sdd_manNotify_s {
   sdd_name_t                   name;
   sc_ticks_t                   tmo;
   sc_tmoid_t                   tmoid;
} sdd_manNotify_t;
```

5.13.4 Structure Members

name.base Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

name.name Name of the device or SDD object.

Used in the request message to specify the name of the device.

tmo Time-out.

If the manager does not respond within this time-out it will reply with the **SDD_MAN_NOTIFY_RM_REPLY** message containing an error code.

tmoid Internal use.

Used by the manager for its own purposes.

5.13.5 Errors

name.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EEXIST	Device already exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.14 SDD_MAN_RM / SDD_MAN_RM_REPLY

5.14.1 Description

This message is used to remove a device from the device driver system.

The process sends an **SDD_MAN_RM** request message to the device manager process. The device manager process removes the device from its device database and replies with the **SDD_MAN_RM_REPLY** reply message.

5.14.2 Message IDs

Request Message	**SDD_MAN_RM**
Reply Message	**SDD_MAN_RM_REPLY**

5.14.3 sdd_manRm_t Structure

```
typedef struct sdd_manRm_s {
   sdd_obj_t                    object;
} sdd_manRm_t;
```

5.14.4 Structure Members

base Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

5.14.5 Errors

object.base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
SC_ENOTSUPP	This request is not supported.

5.15 SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY

5.15.1 Description

This message is used to create a copy of a device with identical data structures.

The user process sends an **SDD_OBJ_DUP** request message to the controller process of the device. The controller process sends an **SDD_OBJ_DUP_REPLY** reply message back.

5.15.2 Message IDs

Request Message	**SDD_OBJ_DUP**
Reply Message	**SDD_OBJ_DUP_REPLY**

5.15.3 sdd_objDup_t Structure

```
typedef struct sdd_objDup_s {
   sdd_baseMessage_t            base;
} sdd_objDup_t;
```

5.15.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.

5.15.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

5.16 SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY

5.16.1 Description

This message is used to release a temporary object.

A temporary object is an SDD object which is created by the manager without involving a real device. This is mainly used in the file system. To free such an object, this **SDD_OBJ_RELEASE** message must be used. Please consult chapter 7.6 “Temporary Objects” on page 7-5 for more information about temporary objects.

The user process sends an **SDD_OBJ_RELEASE** request message to the SDD manager. The SDD manager sends an **SDD_OBJ_RELEASE_REPLY** reply message back.

5.16.2 Message IDs

Request Message	**SDD_OBJ_RELEASE**
Reply Message	**SDD_OBJ_RELEASE_REPLY**

5.16.3 sdd_objRelease_t Structure

```
typedef struct sdd_objRelease_s {
   sdd_baseMessage_t            base;
} sdd_objRelease_t;
```

5.16.4 Structure Members

base	Base object descriptor.

Specifies the base SDD object descriptor structure of an SDD object.

5.16.5 Errors

base.error	Error code.
ENOENT	Device does not exists.
SC_ENOTSUPP	This request is not supported.

5.17 SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY

5.17.1 Description

This message is used to get the size of an SDD object. This can be cache sizes, file sizes or any sizes an object could have.

The user process sends an **SDD_OBJ_SIZE_GET** request message to the controller process of the device driver. The controller process sends an **SDD_OBJ_SIZE_GET_REPLY** reply message back.

5.17.2 Message IDs

Request Message	**SDD_OBJ_SIZE_GET**
Reply Message	**SDD_OBJ_SIZE_GET_REPLY**

5.17.3 sdd_objTime_t Structure

```
typedef struct sdd_objSize_s {
   sdd_baseMessage_t            base;
   size_t                       total;
   size_t                       free;
   size_t                       bad;
} sdd_objSize_t;
```

5.17.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.
total	Total size of the SDD object.
free	Free available size of the SDD object.
bad	Not available size of the SDD object.

5.17.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

5.18 SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY

5.18.1 Description

This message is used to get the time from device drivers.

The user process sends an **SDD_OBJ_TIME_GET** request message to the controller process of the device driver. The controller process sends an **SDD_OBJ_TIME_GET_REPLY** reply message back.

5.18.2 Message IDs

Request Message	**SDD_OBJ_TIME_GET**
Reply Message	**SDD_OBJ_TIME_GET_REPLY**

5.18.3 sdd_objTime_t Structure

```
typedef struct sdd_objTime_s {
   sdd_baseMessage_t            base;
   __u32                        date;
} sdd_objTime_t;
```

5.18.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.
date	Date and time in a user defined format.

5.18.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

5.19 SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY

5.19.1 Description

This message is used to set the time of device drivers.

The user process sends an **SDD_OBJ_TIME_SET** request message to the controller process of the device driver. The controller process sends an **SDD_OBJ_TIME_SET_REPLY** reply message back.

5.19.2 Message IDs

Request Message	**SDD_OBJ_TIME_SET**
Reply Message	**SDD_OBJ_TIME_SET_REPLY**

5.19.3 sdd_objTime_t Structure

```
typedef struct sdd_objTime_s {
   sdd_baseMessage_t            base;
   __u32                        date;
} sdd_objTime_t;
```

5.19.4 Structure Members

base	Base object descriptor.
	Specifies the base SDD object descriptor structure of an SDD object.
date	Date and time in a user defined format.

5.19.5 Errors

base.error	Error code.
EBADF	The member **handle** of the base object descriptor is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

6 Application Programmer Interface

6.1 Introduction

In this chapter all SCIOPTA device driver functions are described.

Only the generic device driver functions are listed.

Please consult the SCIOPTA IPS Internet Protocols, User's Guide and Reference Manual for a description of the specific network device functions (**sdd_net***).

Please consult the SCIOPTA File Systems (FAT and FLASH), User's Guide and Reference Manual for a description of the specific file system device functions (**sdd_file***).

The functions are listed in alphabetical order.

6.1.1 Header File

The function prototypes are defined in the following header file:

<installation_folder>\sciopta\<version>\include\sdd\sdd.h

6.1.2 Kernel Libraries

The following API functions are included in the kernel libraries. The kernel libraries are build for the supported compilers and can be found here:

<installation_folder>\sciopta\<version>\lib\<arch>\

Please consult the kernel libraries sections of the "Building SCIOPTA Systems" chapter of the SCIOPTA - Real-Time Kernel User's Manual for more information about kernel libraries.

6.1.3 Source Code

Instead of using the gdd (sdd) libraries you could include the source code in your project. This would be useful if other compiler switches are needed as the ones used for building the libraries.

The source codes of all API functions are supplied in the SCIOPTA kernel delivery and can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\

6.2 sdd_devAread

6.2.1 Description

The **sdd_devAread** function is used to read data in an **asynchronous mode** from a device driver. It can only be used if the device was first successfully opened for read.

An **SDD_DEV_READ** message will be allocated and sent to the **receiver** process of the device. The caller process will not be blocked and returns immediately.

The user needs to receive explicitly an **SDD_DEV_READ_REPLY** message which contains the data from the device to be read.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.2.2 Syntax

```
int sdd_devAread (
   sdd_obj_t NEARPTR    self,
   ssize_t              size
);
```

6.2.3 Parameters

self	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device to be read from.
size	Read data size.
	Size of the data buffer where the SDD device will put the read data.

6.2.4 Return Value

This functions actually always returns a zero value.

6.2.5 Source Code

The source code of the **sdd_devAread** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\devaread.c

6.3 sdd_devClose

6.3.1 Description

The **sdd_devClose** function is used to close an open device. If a device is not used any more it should be closed by the user process.

The function sends an **SDD_DEV_CLOSE** message to the **controller** process of the device driver and waits on an **SDD_DEV_CLOSE_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

If a device is not needed any more you should call the **sdd_objRelease** method. This mainly allows to clean all temporary created devices. Please consult chapter 7.6 “Temporary Objects” on page 7-5 for more information.

6.3.2 Syntax

```
int sdd_devClose (
   sdd_obj_t NEARPTR    self
);
```

6.3.3 Parameter

self	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device to be closed.

6.3.4 Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.3.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_devClose** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EIO	An input/output error occurred.
SC_ENOTSUPP	This request is not supported.

6.3.6 Source Code

The source code of the **sdd_devClose** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\devclose.c

6.3.7 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.4 sdd_devIoctl

6.4.1 Description

The **sdd_devIoctl** function is used to get and set specific parameters in device drivers on the hardware layer. It is mainly included to be compatible with the BSD API.

The function sends an **SDD_DEV_IOCTL** message to the **controller** process of the device driver and waits on an **SDD_DEV_IOCTL_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.4.2 Syntax

```
int sdd_devIoctl (
   sdd_obj_t NEARPTR    self,
   unsigned int         cmd,
   unsigned long        arg
);
```

6.4.3 Parameter

self	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device from whom to get and set parameters.
cmd	Device specific command.
arg	Command specific argument.

6.4.4 Return Value

If the functions succeeds the return value is zero or positive. The returned value is device specific.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.4.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_devIoctl** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

6.4.6 Source Code

The source code of the **sdd_devIoctl** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\devioctl.c

6.4.7 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.5 sdd_devOpen

6.5.1 Description

The **sdd_devOpen** function is used to open device for read, write or read/write, or to create a device (file).

The function sends an **SDD_DEV_OPEN** message to the **controller** process of the device driver and waits on an **SDD_DEV_OPEN_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.5.2 Syntax

```
int sdd_devOpen (
   sdd_obj_t NEARPTR    self,
   flags_t              flags
);
```

6.5.3 Parameter

self	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device to be opened.
flags	Contains BSD conform flags.
O_RDONLY	Opens the device for read only.
O_WRONLY	Opens the device for write only.
O_RDWR	Opens the device for read and write.
O_TRUNC	Decrease a file to length zero.
O_APPEND	Sets the read/write pointer to the end of the file.

O_TRUNC and O_APPEND can be ored with O_RDONLY and O_WRONLY.
O_RDONLY cannot be ored with O_WRONLY (as it is not equal to O_RDWR!).

6.5.4 Return Value

If the functions succeeds the return value is zero or positive. The returned value is device specific.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.5.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_devOpen** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.

EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

6.5.6 Source Code

The source code of the **sdd_devOpen** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\devopen.c

6.5.7 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.6 sdd_devRead

6.6.1 Description

The **sdd_devRead** function is used to read data from a device driver. It can only be used if the device was first successfully opened for read.

The function sends an **SDD_DEV_READ** message to the **receiver** process of the device driver and waits on an **SDD_DEV_READ_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.6.2 Syntax

```
ssize_t sdd_devRead (
   sdd_obj_t NEARPTR    self,
   char                 *buf,
   ssize_t              size
);
```

6.6.3 Parameter

self	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device to read from.
buf	Buffer where read data is stored.
ssize	Size of the data buffer.

6.6.4 Return Value

If the functions succeeds the return value is zero or positive. The returned value contains the number of read bytes.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.6.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_devRead** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EIO	An input/output error occurred.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

6.6.6 Source Code

The source code of the **sdd_devRead** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\devread.c

6.6.7 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.7 sdd_devWrite

6.7.1 Description

The **sdd_devWrite** function is used to write data to a device driver. It can only be used if the device was first successful opened for write.

The function sends an **SDD_DEV_WRITE** message to the **sender** process of the device driver and waits on an **SDD_DEV_WRITE_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.7.2 Syntax

```
ssize_t sdd_devWrite (
   sdd_obj_t NEARPTR    self,
   char                 *buf,
   ssize_t              size
);
```

6.7.3 Parameter

self	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device to write to.
buf	Buffer where the data to be written is stored.
ssize	Size of the data buffer.

6.7.4 Return Value

If the functions succeeds the return value is zero or positive. The returned value contains the number of written bytes.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.7.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_devWrite** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EIO	An input/output error occurred.
EFBIG	Size of data to be written to big.
EINVAL	Invalid parameter.

SC_ENOTSUPP This request is not supported.

6.7.6 Source Code

The source code of the **sdd_devWrite** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\devwrite.c

6.7.7 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.8 sdd_manAdd

6.8.1 Description

The **sdd_manAdd** function is used to add a new device in the device driver system by registering it at a device manager process.

The function sends an **SDD_MAN_ADD** message to the **controller** process of the manager and waits on an **SDD_MAN_ADD_REPLY** message. The caller process will be blocked until this message is received.

6.8.2 Systax

```
int sdd_manAdd (
   sdd_obj_t NEARPTR         self,
   sdd_obj_t NEARPTR NEARPTR object
);
```

6.8.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
object	SDD device descriptor.
	Specifies the base SDD object descriptor of the SDD device which will be registered at the manager. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object. Please note the pointer to a pointer type.

6.8.4 Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.8.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manAdd** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EEXIST	Device already exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.8.6 Source Code

The source code of the **sdd_manAdd** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\manadd.c

6.8.7 Example

```
    .
    .
sdd_obj_t *dev;
sdd_obj_t *man;

unit = procNumGet();
async_t * const me = &async[unit];

dev = (sdd_obj_t *) sc_msgAlloc (sizeof (sdd_obj_t), 0,
                                 SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
me->dev = dev;
dev->base.error = 0;
dev->base.handle = me;
dev->type = SDD_OBJ_TYPE | SDD_DEV_TYPE;
strncpy (dev->name, "uart0", SC_NAME_MAX);
dev->name[strlen("uart0") - 1] += unit;
dev->controller = sc_procIdGet (0, SC_NO_TMO);

/*
** Search interrupt process
*/
msg = sc_msgAllocClr(SC_NAME_MAX,0,SC_DEFAULT_POOL,SC_FATAL_IF_TMO);
strcpy((char *)msg,"SCI_serial0");
((char *)msg)[10] += unit;
dev->sender = dev->receiver = sc_procIdGet ((char *)msg, SC_NO_TMO);
sc_msgFree(&msg);

/* register to dev man */
man = sdd_manGetRoot (MANAGER_NAME, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
if (man){
    if (sdd_manAdd (man, &dev)) {
        PRINTF ("Could not add this device \n");
        sc_procKill (SC_CURRENT_PID, 0);
    }
    sdd_objFree (&man);
}
else {
    PRINTF ("Could not get manager\n");
    sc_procKill (SC_CURRENT_PID, 0);
}
    .
    .
```

6.9 sdd_manGetByName

6.9.1 Description

the **sdd_manGetByName** function is used to get the SDD device descriptor of a registered device from the manager's device list by giving the name as parameter.

The function sends an **SDD_MAN_GET** message to the **controller** process of the manager and waits on an **SDD_MAN_GET_REPLY** message.The caller process will be blocked until this message is received.

The returned SDD device descriptor is a SCIOPTA message. If you do not need to use the device any longer you should free this message buffer by a **sc_msgFree** system call. For temporary devices you should precede the message free by an **sdd_objRelease** function. Please consult chapter 6.18 "sdd_objFree" on page 6-33.

6.9.2 Syntax

```
sdd_obj_t NEARPTR sdd_manGetByName (
   sdd_obj_t NEARPTR       self,
   const char              *name
);
```

6.9.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
name	Name of the device.

6.9.4 Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.9.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manGetByName** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.9.6 Source Code

The source code of the **sdd_manGetByName** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\mangetbyname.c

6.9.7 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.10 sdd_manGetByPath

6.10.1 Description

The **sdd_manGetByPath** function is used to get the SDD device descriptor of a registered device from the manager's device list by giving the path as parameter.

The function is first executing an **sdd_objResolve** function and then calls **sdd_manGetByPath**.

The returned SDD device descriptor is a SCIOPTA message. If you do not need to use the device any longer you should free this message buffer by a **sc_msgFree** system call. For on-thy-fly devices you should precede the message free by an **sdd_objRelease** function. Please consult chapter 6.18 "sdd_objFree" on page 6-33.

6.10.2 Syntax

```
sdd_obj_t NEARPTR sdd_manGetByPath (
   sdd_obj_t NEARPTR       self,
   const char              *path
);
```

6.10.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
path	Path and name of the device.

6.10.4 Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.10.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manGetByPath** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.10.6 Source Code

The source code of the **sdd_manGetByPath** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\mangetbypath.c

6.10.7 Example

```
SC_PROCESS(SCP_fatfsTest)
{
    logd_t *logd;
    sdd_obj_t *fd;
    sdd_obj_t *cur;
    sdd_obj_t *root;

    char verify[100];
    int len;
    int error;

    static const char *testfile = "/Test.txt";
    static const char *testfile2 = "/test.txt";
    static const char *buf = "Hello Sciopta, this is a test text\n";

    logd = logd_new ("/SCP_logd",
        LOGD_INFO,
        "fatfsTest",
        SC_DEFAULT_POOL,
        SC_FATAL_IF_TMO);

    /* Increase log-level */
    logd_levelSet("/SCP_logd",LOGD_INFO);

    logd_printf(logd,LOGD_INFO,"started\n");

/*
** wait for SCP_devman == root manager
*/
    if (!(root = sdd_manGetRoot ("/SCP_devman",
                 "/",
                 SC_DEFAULT_POOL,
                 SC_ENDLESS_TMO)))
    {
        logd_printf (logd, LOGD_SEVERE, "Could not get devman.\n");
        goto error;
    }

    sdd_manNotifyAdd(root, MP_NAME, SC_ENDLESS_TMO);

    logd_printf(logd,LOGD_INFO,"List root:\n");
    dir(root,NULL, logd);
    logd_printf(logd,LOGD_INFO,"------------\n");

    cur = sdd_manGetByPath(root,"/"MP_NAME);

    sdd_objFree(&root);

/*
** set new root
*/
    root = cur;
        .
        .
```

6.11 sdd_manGetFirst

6.11.1 Description

The **sdd_manGetFirst** function is used to get the SDD device descriptor of the first registered device from the manager's device list.

The function sends an **SDD_MAN_GET_FIRST** message to the **controller** process of the manager and waits on an **SDD_MAN_GET_FIRST_REPLY** message. The caller process will be blocked until this message is received.

The returned SDD device descriptor is a SCIOPTA message. If you do not need to use the device any longer you should free this message buffer by a **sc_msgFree** system call. It is good practice to precede the message free by an **sdd_objRelease** function. Please consult chapter 6.18 "sdd_objFree" on page 6-33.

6.11.2 Syntax

```
sdd_obj_t NEARPTR sdd_manGetFirst (
   sdd_obj_t NEARPTR       self,
   int                     size
);
```

6.11.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
size	Size of the expected SDD device descriptor.
	This is usually sizeof(sdd_obj_t) but could also be sizeof (sdd_myobject_t) if you extend the standard SDD descriptor structure.

6.11.4 Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.11.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manGetFirst** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.

SC_ENOTSUPP This request is not supported.

6.11.6 Source Code

The source code of the **sdd_manGetFirst** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\manget1st.c

6.11.7 Example

```
static void dir(sdd_obj_t *folder, const char * mask,logd_t *logd)
{
  sdd_obj_t *cur;
  sdd_obj_t *old;
  sdd_size_t size;

  if ( mask ){
    cur = sfs_findFirst (folder, mask);
  } else {
    cur = sdd_manGetFirst(folder, sizeof(sdd_obj_t));
  }
  if ( !cur ) return;
  logd_printf(logd,LOGD_INFO, "type size\n");
  while (cur) {
    /* test type
     */
    if (cur->type == SFS_ATTR_FILE) {
      if (sdd_objSizeGet (cur, &size) == -1) {
   size.total = 0;
      }
      logd_printf(logd,LOGD_INFO, "FILE %6d %s\n", size.total,cur->name);
    }
    else if (cur->type == SFS_ATTR_DIR) {
      logd_printf(logd,LOGD_INFO, "DIR         %s\n", cur->name);
    }
    else if (cur->type & (SDD_DEV_TYPE|SFS_MOUNTP_TYPE)) {
      if (sdd_objSizeGet (cur, &size) == -1) {
   size.total = 0;
      }
      logd_printf(logd,LOGD_INFO, "DEV  %6d %s\n", size.total,cur->name);
    }
    else if (cur->type & SDD_NET_TYPE) {
      logd_printf(logd,LOGD_INFO, "NET         %s\n", cur->name);
    }
    else if (cur->type & SDD_MAN_TYPE) {
      logd_printf(logd,LOGD_INFO, "MAN         %s\n", cur->name);
    }
    else {
      logd_printf(logd,LOGD_INFO, "OBJ         %s\n", cur->name);
    }
    old = cur;
    cur = sdd_manGetNext (folder, old, sizeof (sdd_obj_t));
    sdd_objFree (&old);
  }
}
```

6.12 sdd_manGetNext

6.12.1 Description

The **sdd_manGetNext** function is used to get SDD device descriptor of the next registered device from the manager's device list.

The function sends an **SDD_MAN_GET_NEXT** message to the **controller** process of the manager and waits on an **SDD_MAN_GET_NEXT_REPLY** message.The caller process will be blocked until this message is received.

The returned SDD device descriptor is a message. If you do not need to use the device any longer you should free this message buffer by a **sc_msgFree** system call. It is good practice to precede the message free by an **sdd_objRelease** function. Please consult chapter 6.18 "sdd_objFree" on page 6-33.

6.12.2 Syntax

```
sdd_obj_t NEARPTR sdd_manGetNext (
   sdd_obj_t NEARPTR    self,
   sdd_obj_t NEARPTR    reference,
   int                  size
);
```

6.12.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
reference	SDD object descriptor of a registered object.
	The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.
size	Size of the expected SDD device descriptor.
	This is usually sizeof(sdd_obj_t) but could also be sizeof (sdd_myobject_t) if you extend the standard SDD descriptor structure.

6.12.4 Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.12.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manGetNext** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.12.6 Source Code

The source code of the **sdd_manGetNext** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\mangetnext.c

6.12.7 Example

```
static void dir(sdd_obj_t *folder, const char * mask,logd_t *logd)
{
  sdd_obj_t *cur;
  sdd_obj_t *old;
  sdd_size_t size;

  if ( mask ){
    cur = sfs_findFirst (folder, mask);
  } else {
    cur = sdd_manGetFirst(folder, sizeof(sdd_obj_t));
  }
  if ( !cur ) return;
  logd_printf(logd,LOGD_INFO, "type size\n");
  while (cur) {
    /* test type
     */
    if (cur->type == SFS_ATTR_FILE) {
      if (sdd_objSizeGet (cur, &size) == -1) {
   size.total = 0;
      }
      logd_printf(logd,LOGD_INFO, "FILE %6d %s\n", size.total,cur->name);
    }
    else if (cur->type == SFS_ATTR_DIR) {
      logd_printf(logd,LOGD_INFO, "DIR         %s\n", cur->name);
    }
    else if (cur->type & (SDD_DEV_TYPE|SFS_MOUNTP_TYPE)) {
      if (sdd_objSizeGet (cur, &size) == -1) {
   size.total = 0;
      }
      logd_printf(logd,LOGD_INFO, "DEV  %6d %s\n", size.total,cur->name);
    }
    else if (cur->type & SDD_NET_TYPE) {
      logd_printf(logd,LOGD_INFO, "NET         %s\n", cur->name);
    }
    else if (cur->type & SDD_MAN_TYPE) {
      logd_printf(logd,LOGD_INFO, "MAN         %s\n", cur->name);
    }
    else {
      logd_printf(logd,LOGD_INFO, "OBJ         %s\n", cur->name);
    }
    old = cur;
    cur = sdd_manGetNext (folder, old, sizeof (sdd_obj_t));
    sdd_objFree (&old);
  }
}
```

6.13 sdd_manGetRoot

6.13.1 Description

The **sdd_manGetRoot** function is used to get (create) an SDD object descriptor of a root manager process. It will set a process as root manager.

An SDD object descriptor is created with **base.handle** = NULL defining the object as a root manager. The created and returned SDD object descriptor is used to address root managers for getting devices.

The created SDD object descriptor of the root manager is a message. If you do not need to use the manager any longer you should free this message buffer by a **sc_msgFree** system call.

6.13.2 Syntax

```
sdd_obj_t NEARPTR sdd_manGetRoot(
   const char         *process,
   const char         *name,
   sc_poolid_t        plid,
   sc_ticks_t         tmo
);
```

6.13.3 Parameter

process	Name of the manager process.
name	Name of the manager.
	The name can be different than the process name.
plid	Pool ID.
	Pool ID from where the SDD-Object will be allocated.
tmo	Time-out.
	Maximum time to wait for the manager to be created.

6.13.4 Return Value

If the functions succeeds the return value is the pointer to the created SDD manager descriptor of the root manager.

If the function fails the return value is NULL.

6.13.5 Source Code

The source code of the **sdd_manGetRoot** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\mangetroot.c

6.13.6 Example

```
SC_PROCESS(SCP_fatfsTest)
{
    logd_t *logd;
    sdd_obj_t *fd;
    sdd_obj_t *cur;
    sdd_obj_t *root;

    char verify[100];
    int len;
    int error;

    static const char *testfile = "/Test.txt";
    static const char *testfile2 = "/test.txt";
    static const char *buf = "Hello Sciopta, this is a test text\n";

    logd = logd_new ("/SCP_logd",
        LOGD_INFO,
        "fatfsTest",
        SC_DEFAULT_POOL,
        SC_FATAL_IF_TMO);

    /* Increase log-level */
    logd_levelSet("/SCP_logd",LOGD_INFO);

    logd_printf(logd,LOGD_INFO,"started\n");

/*
** wait for SCP_devman == root manager
*/
    if (!(root = sdd_manGetRoot ("/SCP_devman",
                 "/",
                 SC_DEFAULT_POOL,
                 SC_ENDLESS_TMO)))
    {
        logd_printf (logd, LOGD_SEVERE, "Could not get devman.\n");
        goto error;
    }

    sdd_manNotifyAdd(root, MP_NAME, SC_ENDLESS_TMO);

    logd_printf(logd,LOGD_INFO,"List root:\n");
    dir(root,NULL, logd);
    logd_printf(logd,LOGD_INFO,"------------\n");

    cur = sdd_manGetByPath(root,"/"MP_NAME);

    sdd_objFree(&root);

/*
** set new root
*/
    root = cur;
        .
        .
```

6.14 sdd_manNotifyAdd

6.14.1 Description

The **sdd_manNotifyAdd** function is used to wait until a device or SDD object has been registered by the manager.

The function sends an **SDD_MAN_NOTIFY_ADD** message to the **controller** process of the manager and waits on an **SDD_MAN_NOTIFY_ADD_REPLY** message. The caller process will be blocked until this message is received or the time-out is expired.

The notify add request can contain a timeout.

6.14.2 Syntax

```
int sdd_manNotifyAdd (
   sdd_obj_t NEARPTR    self,
   const char           *name,
   sc_ticks_t           tmo
);
```

6.14.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
name	Name of the device or SDD object.
	Specifies the name of the SDD object to wait for registering.
tmo	Time-out.
	If the manager does not respond within this time-out it will return an error.

6.14.4 Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.14.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manNotifyAdd** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EEXIST	Device already exists.

ENOENT	Device does not exists.
SC_ENOTSUPP	This request is not supported.

6.14.6 Source Code

The source code of the **sdd_manNotifyAdd** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\mannotifyadd.c

6.14.7 Example

```
SC_PROCESS(SCP_fatfsTest)
{
    logd_t *logd;
    sdd_obj_t *fd;
    sdd_obj_t *cur;
    sdd_obj_t *root;

    char verify[100];
    int len;
    int error;

    static const char *testfile = "/Test.txt";
    static const char *testfile2 = "/test.txt";
    static const char *buf = "Hello Sciopta, this is a test text\n";

    logd = logd_new ("/SCP_logd",
        LOGD_INFO,
        "fatfsTest",
        SC_DEFAULT_POOL,
        SC_FATAL_IF_TMO);

    /* Increase log-level */
    logd_levelSet("/SCP_logd",LOGD_INFO);

    logd_printf(logd,LOGD_INFO,"started\n");

/*
** wait for SCP_devman == root manager
*/
    if (!(root = sdd_manGetRoot ("/SCP_devman",
                 "/",
                 SC_DEFAULT_POOL,
                 SC_ENDLESS_TMO)))
    {
        logd_printf (logd, LOGD_SEVERE, "Could not get devman.\n");
        goto error;
    }

    sdd_manNotifyAdd(root, MP_NAME, SC_ENDLESS_TMO);

    logd_printf(logd,LOGD_INFO,"List root:\n");
    dir(root,NULL, logd);
    logd_printf(logd,LOGD_INFO,"------------\n");

    cur = sdd_manGetByPath(root,"/"MP_NAME);

    sdd_objFree(&root);

/*
** set new root
*/
    root = cur;
        .
        .
```

6.15 sdd_manNotifyRm

6.15.1 Description

The **sdd_manNotifyRm** function is used to wait until a device or SDD object has been removed by the manager.

The function sends an **SDD_MAN_NOTIFY_RM** message to the **controller** process of the manager and waits on an **SDD_MAN_NOTIFY_RM_REPLY** message. The caller process will be blocked until this message is received or the time-out is expired.

The notify add request can contain a timeout.

6.15.2 Syntax

```
int sdd_manNotifyAdd (
   sdd_obj_t NEARPTR    self,
   const char           *name,
   sc_ticks_t           tmo
);
```

6.15.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
name	Name of the device or SDD object.
	Specifies the name of the SDD object to wait for removing.
tmo	Time-out.
	If the manager does not respond within this time-out it will return an error.

6.15.4 Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.15.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manNotifyRm** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EEXIST	Device already exists.

ENOENT	Device does not exists.
SC_ENOTSUPP	This request is not supported.

6.15.6 Source Code

The source code of the **sdd_manNotifyRm** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\mannotifyrm.c

6.15.7 Example

```
SC_PROCESS(test)
{
  int error;
  logd_t *logd;
  sdd_obj_t *root;

  logd = logd_new ("/SCP_logd", LOGD_INFO, "msc-demo", SC_DEFAULT_POOL,
         SC_FATAL_IF_TMO);


  logd_printf(logd, LOGD_INFO, "Waiting for drv0\n");

  /* First get root manager */
  root = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_ENDLESS_TMO);
  if ( !root ){
    logd_printf(logd, LOGD_SEVERE, "Error could not get root-manager\n");
    sc_sleep(10);
    sc_miscError(SC_ERR_SYSTEM_FATAL,1);
    sc_procKill(SC_CURRENT_PID,0);
  }

  for(;;){
    char help[12];
    /* wait for FS to appear */
    error = sdd_manNotifyAdd(root, "drv0", SC_ENDLESS_TMO);

    logd_printf(logd, LOGD_INFO, "Found device...\n");

    f_enterFS();
    f_chdrive(0);

    if ( !f_getlabel(0,help,11)){
      logd_printf(logd, LOGD_INFO,"0:->%s<\n",help);
    }

    dir("*.*");

    f_releaseFS(sc_procIdGet(NULL, SC_NO_TMO));

    logd_printf(logd,LOGD_INFO,"Ok to remove USB-stick now ....\n");
    /* Wait until drv0 is detached */
    error = sdd_manNotifyRm(root, "drv0", SC_ENDLESS_TMO);
  }
}
```

6.16 sdd_manRm

6.16.1 Description

The **sdd_manRm** function is used to remove registered device, files and directories from a manager.

The function sends an **SDD_MAN_RM** message to the **controller** process of the manager and waits on an **SDD_MAN_RM_REPLY** message.The caller process will be blocked until this message is received.

6.16.2 Syntax

```
int sdd_manRm (
   sdd_obj_t NEARPTR    self,
   sdd_obj_t NEARPTR    reference,
   int                  size
);
```

6.16.3 Parameter

self	SDD manager descriptor.
	Specifies the base SDD object descriptor of a device manager. If the system is using a non-hierarchical manager layout, all managers are configured as root managers. The SDD object descriptor of such a root manager can be collected by calling the **sdd_manGetRoot** function.
reference	SDD object descriptor of the registered object to remove.
	The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.
size	Size of the SDD object descriptor.
	This is usually sizeof(sdd_obj_t) but could also be sizeof (sdd_myobject_t) if you extend the standard SDD descriptor structure.

6.16.4 Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.16.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_manRm** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOENT	Device does not exists.
SC_ENOTSUPP	This request is not supported.

6.16.6 Source Code

The source code of the **sdd_manRm** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\manrm.c

6.16.7 Example

```
static void unregisterDev(const char * name, logd_t *logd)
{
  /* registration */
  sdd_obj_t NEARPTR dev;
  sdd_obj_t NEARPTR man;

  logd_printf (logd, LOGD_INFO, "Unregister %s\n", name);

  dev = (sdd_obj_t NEARPTR) sc_msgAllocClr(sizeof (sdd_obj_t),
                   SDD_OBJ,
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);
  strncpy (dev->name, name, SC_NAME_MAX);

  /* unregister from dev man */
  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  if (man) {
    if (sdd_manRm (man, dev, sizeof(sdd_obj_t))) {
      logd_printf (logd, LOGD_SEVERE, "Could not remove %s\n",name);
      sc_procKill (SC_CURRENT_PID, 0);
    }
    sdd_objFree (&man);
    sc_msgFree((sc_msgptr_t)&dev);
  }
  else {
    logd_printf (logd, LOGD_SEVERE, "Could not get /SCP_devman\n");
    sc_procKill (SC_CURRENT_PID, 0);
  }
}
```

6.17 sdd_objDup

6.17.1 Description

The **sdd_objDup** function is used to create a copy of the SDD device descriptor of a device by adopting the same state and context as the original device.

The function sends an **SDD_OBJ_DUP** message to the **controller** process of the device and waits on an **SDD_OBJ_DUP_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.17.2 Syntax

```
sdd_obj_t NEARPTR sdd_objDup (
   sdd_obj_t NEARPTR         self
);
```

6.17.3 Parameter

self	SDD object descriptor.
	Specifies the SDD descriptor of the registered object to remove. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

6.17.4 Return Value

If the functions succeeds the return value is the pointer to the copied SDD object descriptor.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

6.17.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_objDup** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.17.6 Source Code

The source code of the **sdd_objDup** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\objdup.c

6.17.7 Example

```
int mount(logd_t *logd,
     const char *root,
     const char *device,
     const char *filesystem,
     const char *mountpoint,
     int volume,
     sc_ticks_t tmo)
{
  sdd_obj_t NEARPTR rt;
  sdd_obj_t NEARPTR dev;
  sdd_obj_t NEARPTR fs;
  sc_msg_t msg;
  int error;
  /*
  ** get SCP_devman
  */
  if (!(rt = sdd_manGetRoot( root,
               "/",
               SC_DEFAULT_POOL,
               SC_FATAL_IF_TMO)))
  {
    logd_printf(logd,LOGD_INFO,"Could not get devman.\n");
    return -1;
  }

  if ( sdd_manNotifyAdd(rt,device,tmo) < 0 ){
    logd_printf(logd,LOGD_INFO, "%s did not appear.\n",device);
    return -1;
  }
  /*
  ** get ram drive
  */
  if (!(dev = sdd_manGetByName (rt, device))) {
    logd_printf(logd,LOGD_INFO, "Could not get <%s> device.\n",device);
    return -1;
  }

  /*
  ** Mount drive
  */

  if ( sdd_manNotifyAdd(rt,filesystem,tmo) < 0 ){
    logd_printf(logd,LOGD_INFO, "%s did not appear.\n",filesystem);
    return -1;
  }

  if (!(fs = sdd_manGetByName (rt, filesystem) )){
    logd_printf(logd,LOGD_INFO, "Could not get FATFS driver.\n");
    return -1;
  }

  msg = sc_msgAlloc(sizeof(hcc_fatfs_mount_t),
           HCC_FATFS_MOUNT,
           SC_DEFAULT_POOL,
           SC_FATAL_IF_TMO);
  msg->mount.dev = sdd_objDup(dev);
  msg->mount.drive = volume;
  strcpy(msg->mount.mp, mountpoint);
  sc_msgTx(&msg,fs->controller,0);
  sdd_objFree(&dev);

  msg = sc_msgRx(SC_ENDLESS_TMO,NULL,0);
  error = msg->mount.error;
  sc_msgFree(&msg);

  sdd_objFree(&rt);
  sdd_objFree(&fs);
  return error;
}
```

6.18 sdd_objFree

6.18.1 Description

The **sdd_objRelease** function is used to release and to free an SDD object mainly an on-the-fly object.

A temporary object is an SDD object which is created by the manager after an **SDD_MAN_GET** request without involving a real device. This is mainly used in the file system. To free such a temporary device, this **sdd_objRelease** method must be used. It is good practice to release not only temporary devices but also ordinary devices as described. Please consult also chapter 7.6 “Temporary Objects” on page 7-5.

The function sends an **SDD_OBJ_RELEASE** message to the **controller** process of the device and waits on an **SDD_OBJ_RELEAS_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function.

6.18.2 Syntax

```
void sdd_objFree (
   sdd_obj_t NEARPTR NEARPTR    self
);
```

6.18.3 Parameter

self	SDD object descriptor.
	Specifies the SDD descriptor of the object to release and to free. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

6.18.4 Return Value

None.

6.18.5 Source Code

The source code of the **sdd_objFree** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\objfree.c

6.18.6 Example

```
#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE         "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t NEARPTR man;
  sdd_obj_t NEARPTR dev;
  __u8 buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  /* open device for the shell, fd will be 0 == stdin
   */
  dev = sdd_manGetByName(man,DEVICE);
  if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
    sc_msgFree ((sc_msgptr_t) &man);
    sc_miscError (0x1001, sc_miscErrnoGet());
  }

  for (;;) {
    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((sc_msgptr_t) &man);
      sc_miscError (0x2001, sc_miscErrnoGet());
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);
    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);
    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
   n = 0;
      } else if ( n > 0 ){
   buf[n] = 0;
   n = sdd_devWrite(dev,buf,n);
      } else if ( n < 0 ){
   kprintf(0,"Error: %d\n",sc_miscErrnoGet());
      }
    }
    sdd_devClose(dev);
  }
  sdd_objFree(&dev);
}
```

6.19 sdd_objResolve

6.19.1 Description

The **sdd_objResolve** function is used to return the last struct manager in a given path for hierarchical organized managers. This is mainly used in the file system.

If the system is using a non-hierarchical device manager layout, all device managers are configured as root managers. The SDD object descriptor of such a root manager must be collected by calling the **sdd_manGetByName** function.

6.19.2 Syntax

```
sdd_obj_t NEARPTR sdd_objResolve(
   sdd_obj_tNEARPTR          self,
   const char                *pathname,
   const char                **last
);
```

6.19.3 Parameter

self	SDD manager descriptor.
	SDD manager descriptor of the starting manager.
pathname	Path name to the SDD object.
last	Name of the last SDD object.
	Name of the last SDD object (which is not a manager) in the list. Will only be valid after execution. Please note the pointer to a pointer type.

6.19.4 Return Value

If the functions succeeds the return value is the pointer to the last SDD manager descriptor in the path.

If the function fails the return value is NULL.

6.19.5 Source Code

The source code of the **sdd_objResolve** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\objresolve.c

6.20 sdd_objSizeGet

6.20.1 Description

The **sdd_objSizeGet** function is used to get the size of an SDD object. This can be cache sizes, file sizes or any sizes an object could have.

The function sends an **SDD_OBJ_SIZE_GET** message to the **controller** process of the device and waits on an **SDD_OBJ_SIZE_GET_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.20.2 Syntax

```
int sdd_objSizeGet (
   sdd_obj_t NEARPTR          self,
   sdd_size_t                 *size
);
```

6.20.3 Parameter

self	SDD object descriptor.
	Specifies the SDD descriptor of the object to get the size from. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.
size	Size of the SDD object.

6.20.4 Return Value

If the function fails the return value is 0. To get the error information call **sc_miscErrnoGet**.

6.20.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_objSizeGet** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.20.6 Source Code

The source code of the **sdd_objSizeGet** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\objsizeget.c

6.20.7 Example

```
static void dir(sdd_obj_t *folder, const char * mask,logd_t *logd)
{
  sdd_obj_t *cur;
  sdd_obj_t *old;
  sdd_size_t size;

  if ( mask ){
    cur = sfs_findFirst (folder, mask);
  } else {
    cur = sdd_manGetFirst(folder, sizeof(sdd_obj_t));
  }
  if ( !cur ) return;
  logd_printf(logd,LOGD_INFO, "type size\n");
  while (cur) {
    /* test type
     */
    if (cur->type == SFS_ATTR_FILE) {
      if (sdd_objSizeGet (cur, &size) == -1) {
   size.total = 0;
      }
      logd_printf(logd,LOGD_INFO, "FILE %6d %s\n", size.total,cur->name);
    }
    else if (cur->type == SFS_ATTR_DIR) {
      logd_printf(logd,LOGD_INFO, "DIR         %s\n", cur->name);
    }
    else if (cur->type & (SDD_DEV_TYPE|SFS_MOUNTP_TYPE)) {
      if (sdd_objSizeGet (cur, &size) == -1) {
   size.total = 0;
      }
      logd_printf(logd,LOGD_INFO, "DEV  %6d %s\n", size.total,cur->name);
    }
    else if (cur->type & SDD_NET_TYPE) {
      logd_printf(logd,LOGD_INFO, "NET         %s\n", cur->name);
    }
    else if (cur->type & SDD_MAN_TYPE) {
      logd_printf(logd,LOGD_INFO, "MAN         %s\n", cur->name);
    }
    else {
      logd_printf(logd,LOGD_INFO, "OBJ         %s\n", cur->name);
    }
    old = cur;
    cur = sdd_manGetNext (folder, old, sizeof (sdd_obj_t));
    sdd_objFree (&old);
  }
}
```

6.21 sdd_objTimeGet

6.21.1 Description

The **sdd_objTimeGet** function is used to get the time of an SDD device.

The function sends an **SDD_OBJ_TIME_GET** message to the **controller** process of the device and waits on an **SDD_OBJ_TIME_GET_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.21.2 Syntax

```
int sdd_objTimeGet (
   sdd_obj_t NEARPTR          self,
   __u32                      data
);
```

6.21.3 Parameter

self	SDD object descriptor.
	Specifies the SDD descriptor of the object to get the time from. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.
data	Time data in a user defined format.

6.21.4 Return Value

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.21.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_objTimeGet** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.21.6 Source Code

The source code of the **sdd_objTimeGet** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\objtimeget.c

6.22 sdd_objTimeSet

6.22.1 Description

The **sdd_objTimeSet** function is used to get the time of an SDD device.

The function sends an **SDD_OBJ_TIME_SET** message to the **controller** process of the device and waits on an **SDD_OBJ_TIME_SET_REPLY** message. The caller process will be blocked until this message is received.

The SDD device descriptor must be collected from a device manager by calling the **sdd_manGetByName** function and it might be valid only after the device was successfully opened after calling the **sdd_devOpen** function.

6.22.2 Description

```
int sdd_objTimeSet (
   sdd_obj_t NEARPTR          self,
   __u32                      data
);
```

6.22.3 Parameter

self	SDD object descriptor.
	Specifies the SDD descriptor of the object to set the time to. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.
data	Time data in a user defined format.

6.22.4 Return Value

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

6.22.5 Errors

error code	Return value of **sc_miscErrnoGet**
	After an **sdd_objTimeSet** error.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

6.22.6 Source Code

The source code of the **sdd_objTimeSet** function can be found here:

<installation_folder>\sciopta\<version>\gdd\sdd\objtimeset.c

7 Device Manager

7.1 Description

The Device Manager Process is the main process of a SCIOPTA device driver system. It maintains the list of all registered device in its device database.

User processes and devices communicate with the device manager process with specific message types to register and remove devices or to get information about registered devices.

7.2 Root Manager

In SCIOPTA systems without the need for file system functionality there are usually only root managers. Root managers are managing devices in a system. There can be more than one root managers in a SCIOPTA system.

In a hierarchical organized device manager system (see chapter 7.5 “Hierarchical Structured Managers” on page 7-4) the root manager is the top level and reference manager.

7.3 Manager Duties

A manager has the following jobs and duties:

- Maintaining the list of registered devices.
- Adding new devices in the list as required by device drivers.
- Removing existing devices from the list.
- Returning information about registered devices to inquiring processes.

7.4 Message Handling in Managers

A user written device manager should be able to receive and handle the following messages.

7.4.1 SDD_MAN_ADD

This message is received from a device driver. The manager will register the device in its device database and replies with an **SDD_MAN_ADD_REPLY** message. If the manager encounters an error, the error code will be included in the reply message. Please consult chapter 5.8 “SDD_MAN_ADD / SDD_MAN_ADD_REPLY” on page 5-11 for the message description.

7.4.2 SDD_MAN_RM

This message is received from a device driver. The manager will remove the device from its device database and replies with an **SDD_MAN_RM_REPLY** message. If the manager encounters an error, the error code will be included in the reply message. Please consult chapter 5.14 “SDD_MAN_RM / SDD_MAN_RM_REPLY” on page 5-19 for the message description.

7.4.3 SDD_MAN_GET

This message is received from a user process which needs information about a registered device. The message contains the name of the device. The manager will search the device in its device database and fill all device information into the **SDD_MAN_GET_REPLY** message if the device was found. If the manager encounters an error it will send the reply message including the error code. Please consult chapter 5.9 “SDD_MAN_GET / SDD_MAN_GET_REPLY” on page 5-12 for the message description.

7.4.4 SDD_MAN_GET_FIRST

This message is received from a user process which wants to scan through the device registry of a manager. This is mainly used in file systems. The manager will return the first entry in its device driver database in the **SDD_MAN_GET_FIRST_REPLY** message. If the manager encounters an error it will send the reply message including the error code. Please consult chapter 5.10 “SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY” on page 5-13 for the message description.

7.4.5 SDD_MAN_GET_NEXT

This message is received from a user process which wants to scan through the device registry of a manager. This is mainly used in file systems. The manager will return the next entry in its device driver database in the **SDD_MAN_GET_NEXT_REPLY** message. If the manager encounters an error it will send the reply message including the error code. Please consult chapter 5.11 “SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY” on page 5-14 for the message description.

7.4.6 SDD_MAN_NOTIFY_ADD

This message is received from a user process which wants to wait on a device to be registered by the manager. The manager will return a **SDD_MAN_NOTIFY_ADD_REPLY** message after it has registered the device in its device database. If the manager encounters an error it will send the reply message including the error code. Please consult chapter 5.12 “SDD_MAN_NOTIFY_ADD / SDD_MAN_NOTIFY_ADD_REPLY” on page 5-15 for the message description.

7.4.7 SDD_MAN_NOTIFY_RM

This message is received from a user process which wants to wait on a device to be removed from the device database. The manager will return a **SDD_MAN_NOTIFY_RM_REPLY** message after it has removed the device. If the manager encounters an error it will send the reply message including the error code. Please consult chapter 5.13 “SDD_MAN_NOTIFY_RM / SDD_MAN_NOTIFY_RM_REPLY” on page 5-17 for the message description.

7.5 Hierarchical Structured Managers

In a hierarchical manager organization, managers reside below the root managers and have a nested organization. Hierarchical organized manager systems are mainly used in file systems such as the SCIOPTA SFS.

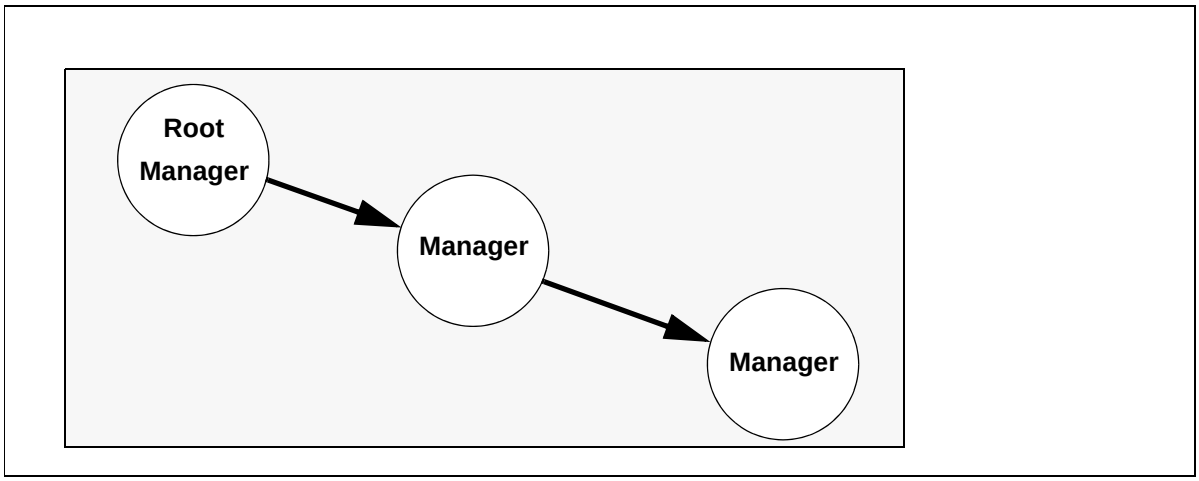


Figure 7-1: SCIOPTA Hierarchical Structured Managers

The main advantage of using hierarchical structured managers is that the SCIOPTA Device Driver File Descriptor Interface can perform single file tree accesses.

The user can build hierarchical structured managers by getting the SDD object descriptor of the root manager and the other managers by using **sdd_manGetRoot** functions and then register the next manager at the root manager by **sdd_manAdd** functions:

(sdd_manAdd(<SDD object descriptor of the root manager>, <SDD object descriptor of the next manager>)).

7.6 Temporary Objects

Temporary objects are SDD objects which are created by managers upon request from a user. Temporary objects can be devices, files, directories or even other managers. User processes are usually communicating with temporary object through the manager which has created the object.

A temporary object is not registering itself at the manager as usual SDD objects (device drivers) will do. Temporary objects are mainly used in file systems such as SCIOPTA SFS.

To remove a temporary object the function **sdd_objRelease** should be used. This is the only way to release the access handle which is owned by the manager. Mainly in file systems it should be avoided to create lots of temporary objects without releasing (freeing) them. See also chapter 6.18 “sdd_objFree” on page 6-33)

7.7 Opaque Manager Handle

Message data which is received or sent by managers include a data element called **manager**.

Example of the structure of a **SDD_MAN_ADD** message which registers a device at the device manager:

```
typedef struct sdd_manAdd_s {
   struct sdd_obj_s {
      struct sdd_baseMessage_s {
         sc_msgid_t                id;
         sc_errorcode_t            error;
         void                      *handle;
      } base;
      void                         *manager; /* Opaque manager handle */
      sc_msgid_t                   type;
      unsigned char                name[SC_NAME_MAX + 1];
      sc_pid_t                     controller;
      sc_pid_t                     sender;
      sc_pid_t                     receiver;
   } object;
} sdd_manAdd_t;
```

The opaque manager handle (**manager**) is a pointer to a structure which further specifies the manager.

This handle is only used in all manager messages (**SDD_MAN_XXX**).

You do not need to write anything in the opaque manager handle if you are using the function interface as this is done in the interface layer.

7.8 Example of a SCIOPTA Device Manager

You will find the code for the standard device manager used in the SCIOPTA examples here:

<installation_folder>\sciopta\<version>\gdd\sdd\manager.c

8 Device Driver

8.1 Description

Device driver processes are managing and controlling the devices in a SCIOPTA system.

A SCIOPTA device driver can contain one or more processes. The controller process is initializing the process, performing some system tasks and also responsible for shut down the device. For executing the specific data input and output tasks more processes may be added to a device driver such as a sender process and receiver process. Some more complex drivers may even require more processes. A simple SCIOPTA device driver usually will have only one process (controller process).

8.2 Device Driver Processes

Typical device driver have three processes:

1. Controller process.
2. Sender process.
3. Receiver process

For interrupt handling there might be some additional interrupt processes or the sender and/or the receiver process are implemented as interrupt processes.

Simple devices can have just one process. Its up to the device driver designer to implement the number of device driver processes suitable for the device.

8.3 Register a Device

If a user needs to work with a device, he usually will get the SDD device descriptor from a device manager. Then he can open and close the device and can send and receive data to and from the device. Therefore the device must register itself at a device manager.

If directly messages are used the device can allocate an **SDD_MAN_ADD** message, fill the data with all device information and sent it to the manager. See also chapter 5.8 “SDD_MAN_ADD / SDD_MAN_ADD_REPLY” on page 5-11.

If the Function Interface is used the device driver will allocate an SDD device descriptor of type sdd_obj_t and fill the structure with the device data. Then the function sdd_manAdd can be called (see chapter 6.8 “sdd_manAdd” on page 6-13). The sdd_manAdd function needs two parameters, one is the SDD device descriptor and the other is the SDD object descriptor of the device manager (all information such as process IDs and handles of the manager). If the device manager is a root manager the function sdd_manGetRoot must be used to get the SDD object descriptor of the manager (see chapter 6.13 “sdd_manGetRoot” on page 6-23).

8.3.1 Register a Device Using Messages

Before a device can be used it must be registered. The device driver needs to register the device directly at the device manager. Using the message interface this is done by sending a **SDD_MAN_ADD** message. The device manager will enter this device in its device database.

1. Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_obj_t          dev;
};

sc_msg_t     msg;
```

2. Allocate an SDD_MAN_ADD message of type sdd_obj_t.

```
msg = sc_msgAlloc( sizeof (sdd_obj_t),
                   SDD_MAN_ADD,
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);
```

3. Fill the SDD device descriptor (message body).

msg->dev.base.id	SDD_MAN_ADD (Filled by sc_msgAlloc)
msg->dev.base.error	0
msg->dev.base.handle	Access handle of the device driver
msg->dev.manager	0 (SCP_netman is a Root Manager)
msg->dev.type	Type of the device
msg->dev.name	Name string of the device
msg->dev.controller	Device driver controller process ID
msg->dev.sender	Device driver sender process ID
msg->dev.receiver	Device driver receiver process ID

4. Send the message to the device manager process (i.e. /**SCP_devman**). If SCP_devman is a static process you can address it by just append **_pid** to the process name. If the manager is a dynamic process you must use the sc_procIdGet to get the manager process ID.

```
sc_msgTx (&ipv4msg, SCP_devman_pid, 0);
```

5. Receive the SDD_MAN_ADD_REPLY message from SCP_devman.

```
static const sc_msgid_t select[2] = { SDD_MAN_ADD_REPLY, 0 };

msg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The SDD_MAN_GET_REPLY message is sent by SCP_devman and received.

Check msg->dev.base.error for a returned error condition.

8.3.2 Registering a Device Using the Function Interface

Before a device can be used it must be registered. The device driver needs to register the device directly at the device manager. Using the SDD function interface this is done by setting-up an SDD device descriptor and using the sdd_manAdd function. The device manager will enter this device in its device database.

1. The device driver allocates first SCIOPTA message of type **sdd_obj_t** to be used as SDD device descriptor.

```
sdd_obj_t NEARPTR dev;
dev = (sdd_obj_t NEARPTR) sc_msgAlloc(  sizeof (sdd_obj_t),
                                        0,
                                        SC_DEFAULT_POOL,
                                        SC_FATAL_IF_TMO );
```

2. The device driver fills the data structure of the SDD-Object with the device data:

dev->base.id	not used
dev->base.error	0
dev->base.handle	Access handle of the device driver
dev->manager	0 (SCP_netman is a Root Manager)
dev->type	Type of the device
dev->name	Name string of the device
dev->controller	Device driver controller process ID
dev->sender	Device driver sender process ID
dev->receiver	Device driver receiver process ID

3. Before registering the device the device driver needs to get the SDD object descriptor of the device manager. In this case the device manager is a root manager.

```
sdd_obj_t NEARPTR man;
man = sdd_manGetRoot ("SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
```

   The return value man->base.error can be tested for a possible error condition.

4. Now the device can be registered:

```
ret = sdd_manAdd (man, &dev)
```

   The return value can be tested for a possible error condition.

8.4 Message Handling in Device Drivers

A device driver should be able to receive and handle the following messages.

8.4.1 SDD_DEV_OPEN

This message will open the device. The device driver can reply with an **SDD_DEV_OPEN_REPLY** message including a possible error condition. If no reply message is used the error can be sent back by a specific **SDD_ERROR** message. Please consult chapter 5.4 “SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY” on page 5-5 for the message description.

8.4.2 SDD_DEV_CLOSE

This message will close the device. The device driver can reply with an **SDD_DEV_CLOSE_REPLY** message including a possible error condition. If no reply message is used the error can be sent back by a specific **SDD_ERROR** message. Please consult chapter 5.2 “SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY” on page 5-2 for the message description.

8.4.3 SDD_DEV_READ

After receiving this message the device driver will send back the read data in an **SDD_DEV_READ_REPLY** message. If the device driver encounters an error, the error code can be included in the reply message. Please consult chapter 5.5 “SDD_DEV_READ / SDD_DEV_READ_REPLY” on page 5-6 for the message description.

8.4.4 SDD_DEV_WRITE

This message includes data to be written to the device. The device driver can reply with an **SDD_DEV_WRITE_REPLY** message including a possible error condition. If no reply message is used the error can be sent back by a specific **SDD_ERROR** message. Please consult chapter 5.6 “SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY” on page 5-8 for the message description.

8.4.5 SDD_DEV_IOCTL

This message will set specific device driver parameters. The **SDD_DEV_IOCTL_REPLY** message returns device parameters from the device driver. If the device driver encounters an error, the error code can be included in the reply message. Please consult chapter 5.3 “SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY” on page 5-3 for the message description.

8.4.6 SDD_OBJ_DUP

After receiving this message the device driver will duplicate the access handle (i.e increase a reference counter) and send back the duplicated access handle in an **SDD_OBJ_DUP_REPLY** message. If the device driver encounters an error, the error code can be included in the reply message. Please consult chapter 5.15 “SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY” on page 5-20 for the message description.

8.4.7 SDD_OBJ_RELEASE

This message will release an object (usually an on-the-fly object). An on-the-fly object is an SDD object created by a manager without involving a real device. This is mainly used in the file system. The SDD object can reply with an **SDD_OBJ_RELEASE_REPLY** message including a possible error condition. If no reply message is used the error can be sent back by a specific **SDD_ERROR** message. Please consult chapter 5.16 “SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY” on page 5-21 for the message description. It is good practice to release any object before free-ing it.

8.4.8 SDD_ERROR

This message is mainly used by device drivers which do not use reply messages as answer of request messages for returning error codes. If the device driver encounters an error, the error code will be included in the **SDD_ERROR** message. SDD_ERROR is also used if the device driver receives unknown messages. Please consult chapter 5.7 “SDD_ERROR” on page 5-10 for the message description.

8.5 Opaque Device Handle

Message data which are received or sent by devices include a data element called **handle**.

Example of the structure of a **SDD_MAN_ADD** message (SDD device descriptor) which registers a device at the device manager:

```
typedef struct sdd_manAdd_s {
   struct sdd_obj_s {
      struct sdd_baseMessage_s {
         sc_msgid_t                id;
         sc_errorcode_t            error;
         void                      *handle;  /* Opaque device handle */
      }  base;
      void                         *manager;
      sc_msgid_t                   type;
      unsigned char                name[SC_NAME_MAX + 1];
      sc_pid_t                     controller;
      sc_pid_t                     sender;
      sc_pid_t                     receiver;
   } object;
} sdd_manAdd_t;
```

The handle (or opaque device handle, which would be the correct name) is a pointer to a structure which further specifies the device.

The user of a device which is opening and closing the device, reading from the device and writing to the device does not need to know the handle and the handle structure. The user will usually get the SDD device descriptor by using the **sdd_manGetByName** (see chapter 6.9 “sdd_manGetByName” on page 6-15) function call. The manager will return the SDD device descriptor including the handle.

8.6 Device Driver Examples

Device driver examples can be found in the SCIOPTA delivery

8.6.1 CPU Families Driver Source Files

Typical CPU family files:

druid_uart.c	Druid UART driver.
simple_uart.c	Simple polling UART function for printf debugging or logging.
serial.c	Serial driver.
<driver_name>.c	CPU family specific device drivers. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\src\

8.6.2 Chip Driver Source Files

Typical chip and device files:

<driver_name>.c	Specific device drivers for controllers and devices which are not board specific. File location: <install_folder>\sciopta\<version>\bsp\common\src\

9 Using Device Drivers

9.1 Writing to Device Drivers

9.1.1 Writing Data Using SCIOPTA Messages

1. Message definition:

```
union sc_msg {
   sc_msgid_t        id;
   sdd_obj_t         dev;
};

sc_msg_t     ddmsg;
```

2. First we need to get the SDD device descriptor from the device manager SCP_devman.

   Allocate an SDD_MAN_GET message of type sdd_obj_t.

```
ddmsg = sc_msgAlloc( sizeof (sdd_obj_t),
                     SDD_MAN_GET,
                     SC_DEFAULT_POOL,
                     SC_FATAL_IF_TMO);
```

3. Enter the device name in the SDD_MAN_GET message.

ddmsg->dev.base.id	SDD_MAN_GET (Filled by sc_msgAlloc)
ddmsg->dev.base.error	not used
ddmsg->dev.base.handle	not used
ddmsg->dev.manager	not used
ddmsg->dev.type	not used
ddmsg->dev.name	Name string of the device
ddmsg->dev.controller	not used
ddmsg->dev.sender	not used
ddmsg->dev.receiver	not used

4. Send the message to the device manager process (i.e. /**SCP_devman**). If SCP_devman is a static process you can address it by just append **_pid** to the process name. If the manager is a dynamic process you must use the sc_procIdGet to get the manager process ID.

```
sc_msgTx (&ddmsg, SCP_devman_pid, 0);
```

5. Receive the SDD_MAN_GET_REPLY message from SCP_devman.

```
static const sc_msgid_t select[2] = { SDD_MAN_GET_REPLY, 0 };

ddmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

   The SDD_MAN_GET_REPLY message is sent by SCP_devman and received. The received message is the SDD device descriptor and contains all information how to access the device driver.

ddmsg->dev.base.id	SDD_MAN_GET_REPLY
ddmsg->dev.base.error	Possible error returned by SCP_devman
ddmsg->dev.base.handle	Handle of the device driver.
ddmsg->dev.manager	not used
ddmsg->dev.type	Type of the device
ddmsg->dev.name	Name string of the device (not modified)
ddmsg->dev.controller	controller process ID of the device driver
ddmsg->dev.sender	sender process ID of the device driver

ddmsg->dev.receiver	receiver process ID of the device driver

6. To be able to communicate with the device driver we need to open it. This will return the device driver access handle.

   Message definition:

```
union sc_msg {
   sc_msgid_t        id;
   sdd_devOpen_t     devOpen;
};

sc_msg_t     openmsg;
```

7. Allocate a SDD_DEV_OPEN message of type sdd_devOpen_t.

```
openmsg = sc_msgAlloc (sizeof (sdd_devOpen_t),
                  SDD_DEV_OPEN,
                  SC_DEFAULT_POOL,
                  SC_FATAL_IF_TMO);
```

8. Fill the message body.

openmsg->devOpen.base.id	SDD_DEV_OPEN (Filled by sc_msgAlloc)
openmsg->devOpen.base.error	0
openmsg->devOpen.base.handle	handle of the device driver (copied fromddmsg->dev.base.handle).
openmsg->devOpen.flags	device driver flags.

9. Send this message to the controller process of the device driver.

```
sc_msgTx (&openmsg, ddmsg->dev.controller, 0);
```

10. Receive the SDD_DEV_OPEN_REPLY message from the device driver.

```
static const sc_msgid_t select[2] = { SDD_DEV_OPEN_REPLY, 0 };

openmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

11. The SDD_DEV_OPEN_REPLY message is sent by the device driver and received. The received message contains the access handle of the device driver.

openmsg->devOpen.base.id	SDD_DEV_OPEN_REPLY
openmsg->devOpen.base.error	Possible error returned by the device driver
openmsg->devOpen.base.handle	Access handle of the device driver
openmsg->netOpen.flags	not modified

We have now all information to access the device driver.

- Device processes of the device driver:

  ddmsg->dev.controller
  ddmsg->dev.sender
  ddmsg->dev.receiver

- Access handle: **`openmsg->devOpen.base.handle`**

12. Writing to the device.

    Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_devWrite_t     devWrite;
};

sc_msg_t     writemsg;
```

13. Allocate a SDD_DEV_WRITE message of type sdd_devWrite_t.

```
writemsg = sc_msgAlloc( sizeof (sdd_devWrite_t) + size of data,
                        SDD_DEV_WRITE,
                        SC_DEFAULT_POOL,
                        SC_FATAL_IF_TMO);
```

14. Fill the message body.

writemsg->devWrite.base.id	SDD_DEV_WRITE (Filled by sc_msgAlloc)
writemsg->devWrite.base.error	0
writemsg->devWrite.base.handle	access handle of the device driver (copied fromopenmsg->devOpen.base.handle).
writemsg->devWrite.size	Size of the data buffer.
writemsg->devWrite.curpos	Not used.
writemsg->devWrite.outlineBuf	NULL
writemsg->devWrite.inlineBuf	Data Buffer (copied data)

15. Send this message to the sender process of the device driver.

```
sc_msgTx (&writemsg, ddmsg->dev.sender, 0);
```

16. Receive the SDD_DEV_WRITE_REPLY message from the device driver.

```
static const sc_msgid_t select[2] = { SDD_DEV_WRITE_REPLY, 0 };

writemsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

17. The SDD_DEV_WRITE_REPLY message is sent by the device driver and received.

    Check writedmsg->devWrite.base.error for a returned error condition.

9.1.2 Writing Data Using the SDD Function Interface

1. If a device is registered, a user process can use the device. Also the user process needs to get the SDD object descriptor of the device manager before the SDD device descriptor can be get from the manager. In this case the device is registered at the root manager.

   ```
   sdd_obj_t NEARPTR man;
   man = sdd_manGetRoot ("SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
   ```

   The return value man->base.error can be tested for a possible error condition.

2. The user can now get the SDD device descriptor from the device manager:

   ```
   sdd_obj_t NEARPTR dev
   dev = sdd_manGetByName (man, "DeviceName");
   ```

   The return value dev->base.error can be tested for a possible error condition.

3. Before using the device the user process needs to open it (i.e. for read and write):

   ```
   ret = sdd_devOpen (dev, O_RDWR)
   ```

   The return value can be tested for a possible error condition.

4. Now the device can be used by the user process e.g. for writing:

   ```
   sizez = sdd_devWrite (dev, dataBuffer, noOfBytes)
   ```

10 System Start and Setup

10.1 Standard SCIOPTA Start Sequence

After a system hardware reset the following sequence will be executed from point 1.

In the SCIOPTA SCSIM Simulator after Windows has started the SCIOPTA application by calling the **sciopta_start** function inside the **WinMain** function the sequence will be executed from point 4.

1. The kernel calls the function **reset_hook**.
2. The kernel performs some internal initialization.
3. The kernel calls **cstartup** to initialize the C system.
4. The kernel calls the function **start_hook**.
5. The kernel calls the function **TargetSetup**. The code of this function is automatically generated by the **SCONF** configuration utility and included in the file sconf.c. **TargetSetup** creates the system module.
6. The kernel calls the dispatcher.
7. The first process (init process of the system module) is swapped in.

The code of the following functions is automatically generated by the **SCONF** configuration utility and included in the file sconf.c.

8. The init process of the system module creates all static modules, processes and pools.
9. The init process of the system module calls the **system module start function**. The name of the function corresponds to the name of the system module.
10. The process priority of the init process of the system module is set to 32 and loops for ever.
11. The init process of each created static module calls the **user module start function** of each module. The name of the function corresponds to the name of the respective module.
12. The process priority of the init process of each created static module is set to 32 and loops for ever.
13. The process with the highest system priority will be swapped-in and executed.

Please consult the SCIOPTA - Kernel, User's Manual for a detailed description of above steps and the generic SCIOPTA startup objects.

10.2 Declaring the Device Manager

If you want to use the SCIOPTA Device Manager concept you need a device manager (see chapter 7 “Device Manager” on page 7-1) which must be declared in the SCIOPTA SCON configuration. It is usually placed in the system module.

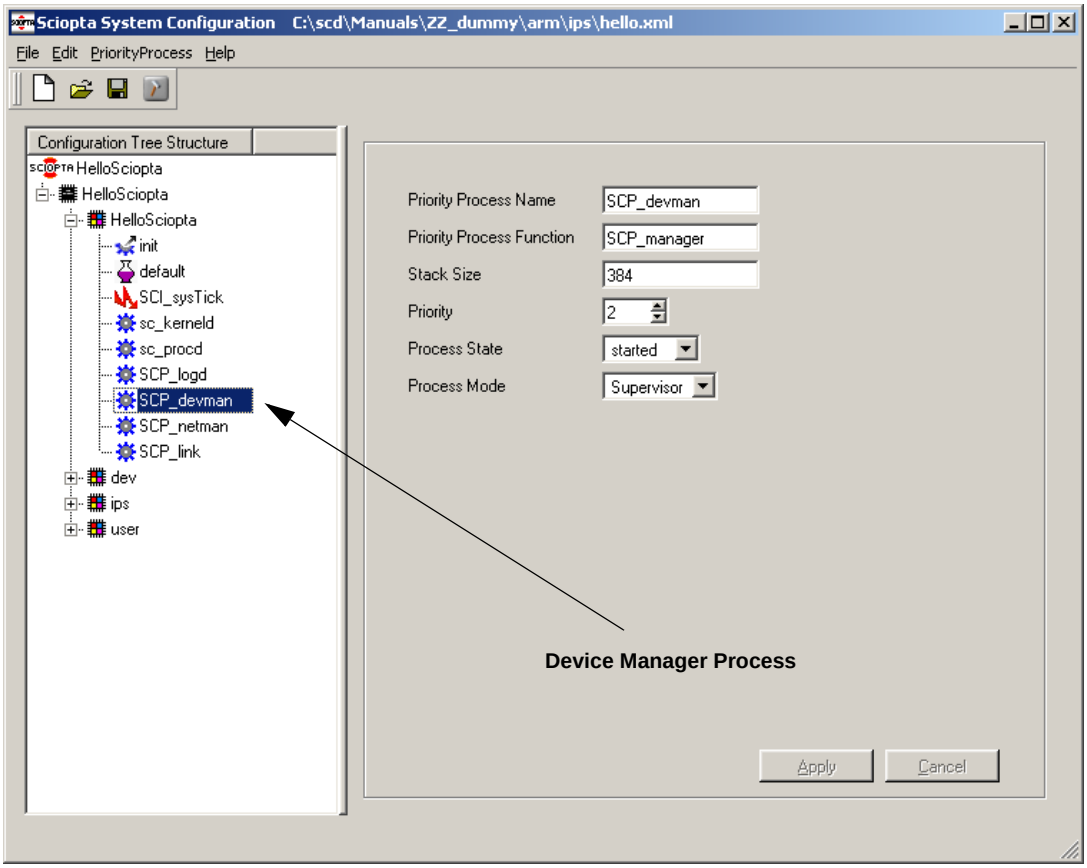
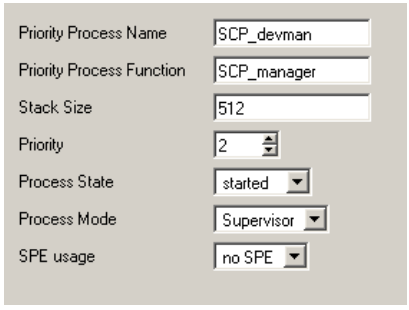


Figure 10-1: SCONF Window of a Typical System Module

10.2.1 Device Manager Process Parameter for All Architectures

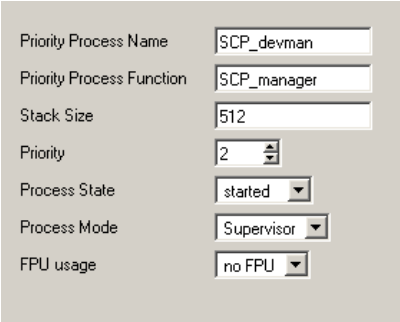
Priority Process Name	Process name.
	SCP_devman is the name of the device manager process in this example.
Priority Process Function	Process function entry.
	SCP_manager is the function name of the device manager. This is the address where the created device manager process will start execution. The code of the standard device manager is included in the (sdd) kernel library.
Stack Size	Process stack size.
	Enter a big enough stack size for the device manager process in bytes.
Priority	Priority of the process.
	The priority of the device manager process must fit your system priority structure.
Process State	Starting state of the device manager process.
started	The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes.
Processor Mode	Selects device manager processor mode.
Supervisor	The device manager process runs in CPU supervisor mode.

10.2.2 Additional Parameters for PowerPC



SPE Usage	Selects if PowerPC Signal Process Engine is used or not.
no SPE	SPE not used.
SPE	SPE is used.

10.2.3 Additional Parameters for ColdFire



FPU usage	Selects if a Floating Point Unit exists and will be used.
no FPU	No FPU in the system
FPU	System includes an FPU

11 Building SCIOPTA Systems

11.1 Introduction

In a new project you have first to determine the specification of the system. As you are designing a real-time system, speed requirements needs to be considered carefully including worst case scenarios. Defining function blocks, environment and interface modules will be another important part for system specification.

Systems design includes defining the modules, processes and messages. SCIOPTA is a message based real-time operating system therefore specific care needs to be taken to follow the design rules for such systems. Data should always be maintained in SCIOPTA messages and shared resources should be encapsulated within SCIOPTA processes.

To design SCIOPTA systems, modules and processes, to handle interprocess communication and to understand the included software of the SCIOPTA delivery you need to have detailed knowledge of the SCIOPTA application programming interface (API). The SCIOPTA API consist of a number of system calls to the SCIOPTA kernel to let the SCIOPTA kernel execute the needed functions.

The SCIOPTA kernel has over 80 system calls. Some of these calls are very specific and are only used in particular situations. Thus many system calls are only needed if you are designing dynamic applications for creating and killing SCIOPTA objects. Other calls are exclusively foreseen to be used in CONNECTOR processes which are needed in distributed applications.

One of the strength of SCIOPTA is that it is easy-to-use. A large part of a typical SCIOPTA application can be written by using the system calls which are handling the interprocess communication: **sc_msgAlloc**, **sc_msgTx**, **sc_msgRx** and **sc_msgFree**. These four system calls together with **sc_msgOwnerGet** which returns the owner of a message and **sc_sleep** which is used to suspend a process for a defined time, are often sufficient to write whole SCIOPTA applications.

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

The SCIOPTA building procedure consists of the following steps:

- Configuring the system with the **SCONF** configuration tool (sconf.exe).
- Locate the include files and define the include paths.
- Assemble the kernel.
- Locate and get all assembler source files and assemble it.
- Locate and get all C/C++ source files and compile them.
- Design the linker script to map your system into the target memory.
- Select and define the correct libraries for the SCIOPTA **Generic Device Driver** (gdd) and **Utilities** (util) functions.
- Link the system.

Please consult the SCIOPTA - Kernel, User's Manual for a detailed description of the SCIOPTA building procedures.

12 Errors

12.1 Standard Error Reference

EPERM	Operation not permitted
ENOENT	No such file or directory
ESRCH	No such process
EINTR	Interrupted system call
EIO	I/O error
ENXIO	No such device or address
E2BIG	Arg list too long
ENOEXEC	Exec format error
EBADF	Bad file number
ECHILD	No child processes
EAGAIN	Try again
ENOMEM	Out of memory
EACCES	Permission denied
EFAULT	Bad address
ENOTBLK	Block device required
EBUSY	Device or resource busy
EEXIST	File exists
EXDEV	Cross-device link
ENODEV	No such device
ENOTDIR	Not a directory
EISDIR	Is a directory
EINVAL	Invalid argument
ENFILE	File table overflow
EMFILE	Too many open files
ENOTTY	Not a typewriter
ETXTBSY	Text file busy
EFBIG	File too large
ENOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system

EMLINK	Too many links
EPIPE	Broken pipe
EDOM	Math argument out of domain of func
ERANGE	Math result not representable
EDEADLK	Resource deadlock would occur
ENAMETOOLONG	File name too long
ENOLCK	No record locks available
ENOSYS	Function not implemented
ENOTEMPTY	Directory not empty
ELOOP	Too many symbolic links encountered
EWOULDBLOCK	Operation would block
ENOMSG	No message of desired type
EIDRM	Identifier removed
ECHRNG	Channel number out of range
EL2NSYNC	Level 2 not synchronized
EL3HLT	Level 3 halted
EL3RST	Level 3 reset
ELNRNG	Link number out of range
EUNATCH	Protocol driver not attached
ENOCSI	No CSI structure available
EL2HLT	Level 2 halted
EBADE	Invalid exchange
EBADR	Invalid request descriptor
EXFULL	Exchange full
ENOANO	No anode
EBADRQC	Invalid request code
EBADSLT	Invalid slot
EDEADLOCK	Resource deadlock would occur
EBFONT	Bad font file format
ENOSTR	Device not a stream
ENODATA	No data available
ETIME	Timer expired
ENOSR	Out of streams resources

ENONET	Machine is not on the network
ENOPKG	Package not installed
EREMOTE	Object is remote
ENOLINK	Link has been severed
EADV	Advertise error
ESRMNT	Srmount error
ECOMM	Communication error on send
EPROTO	Protocol error
EMULTIHOP	Multihop attempted
EDOTDOT	RFS specific error
EBADMSG	Not a data message
EOVERFLOW	Value too large for defined data type
ENOTUNIQ	Name not unique on network
EBADFD	File descriptor in bad state
EREMCHG	Remote address changed
ELIBACC	Can not access a needed shared library
ELIBBAD	Accessing a corrupted shared library
ELIBSCN	lib section in a.out corrupted
ELIBMAX	Attempting to link in too many shared libraries
ELIBEXEC	Cannot exec a shared library directly
EILSEQ	Illegal byte sequence
ERESTART	Interrupted system call should be restarted
ESTRPIPE	Streams pipe error
EUSERS	Too many users
ENOTSOCK	Socket operation on non-socket
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
EPROTOTYPE	Protocol wrong type for socket
ENOPROTOOPT	Protocol not available
EPROTONOSUPPORT	Protocol not supported
ESOCKTNOSUPPORT	Socket type not supported
EOPNOTSUPP	Operation not supported on transport endpoint
EPFNOSUPPORT	Protocol family not supported

EAFNOSUPPORT	Address family not supported by protocol
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Cannot assign requested address
ENETDOWN	Network is down
ENETUNREACH	Network is unreachable
ENETRESET	Network dropped connection because of reset
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by peer
ENOBUFS	No buffer space available
EISCONN	Transport endpoint is already connected
ENOTCONN	Transport endpoint is not connected
ESHUTDOWN	Cannot send after transport endpoint shutdown
ETOOMANYREFS	Too many references: cannot splice
ETIMEDOUT	Connection timed out
ECONNREFUSED	Connection refused
EHOSTDOWN	Host is down
EHOSTUNREACH	No route to host
EALREADY	Operation already in progress
EINPROGRESS	Operation now in progress
ESTALE	Stale NFS file handle
EUCLEAN	Structure needs cleaning
ENOTNAM	Not a XENIX named type file
ENAVAIL	No XENIX semaphores available
EISNAM	Is a named type file
EREMOTEIO	Remote I/O error
EDQUOT	Quota exceeded
ENOMEDIUM	No medium found
EMEDIUMTYPE	Wrong medium type
EHASHCOLLISION	Number of hash collisions exceeds maximum generation counter value

12.2 Specific SCIOPTA Error Reference

SC_EBADBAD	A programming fault
SC_EREFNO	Illegal reference number
SC_ESTATIC	Error due to a statical system
SC_ENOPROC	Requested process does not exist
SC_ENOTIMPL	This request is not implemented
SC_ENOTSUPP	This request is not supported
SC_ENOENT	This entry does not exist
SC_EEXIST	This entry does already exist
SC_ERANGCHK	Range check
SC_ECOULDNOTREG	Range check

13 Board Support Packages

13.1 Introduction

A SCIOPTA board support package (BSP) consists of number of files containing device drivers and project files such as makefiles and linker script for specific boards.

The BSPs are included in the delivery in a specific folder and organized in different directory levels depending on CPU dependency:

1. General System Functions
2. Architecture System Functions
3. CPU Family System Functions
4. Board System Functions

All BSP files can be found at the following top-level location after you have installed SCIOPTA:

File location: <install_folder>\sciopta\<version>\bsp\

Please consult also the SCIOPTA - Device Driver, User's and Reference Manual for information about the SCIOPTA device driver concept.

13.2 General System Functions

General System Functions are functions which are common to all architectures, all CPUs and all boards.

It contains mainly include and source device drivers files for external (not on-chip) controllers.

Generic debugger files might also be placed here.

File location: <install_folder>\sciopta\<version>\bsp\common\include\
File location: <install_folder>\sciopta\<version>\bsp\common\src\

13.3 Architecture System Functions

Architecture System Functions are functions which are architecture (<arch>) specific (please consult chapter 1.3.1 "Architectures" on page 1-3 for the list of supported architectures) and are common to all CPUs and all boards.

It contains generic linker script include files (module.ld), C startup files (cstartup.S) and other architecture files.

File location: <install_folder>\sciopta\<version>\bsp\<arch>\include\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\src\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\src\<compiler>\

13.4 CPU Family System Functions

CPU Family System Functions are functions which are architecture (<arch>) specific and CPU family specific (please consult chapter 1.3.2 “CPU Families” on page 1-3 for the list of supported CPU families) and are common to all boards.

It contains mainly include and source device drivers files for on-chip controllers.

File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\include\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\src\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\src\<compiler>\

13.5 Board System Functions

Board System Functions are functions which are architecture (<arch>) specific, CPU family specific and board specific.

It contains mainly include, source and project files for board setup.

Debugger initialization files and linker scripts might also placed here.

File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\include\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\src\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\src\<compiler>\

14 Manual Versions

14.1 Manual Version 3.0

- Manual completely redesigned containing a new structure.

14.2 Manual Version 2.0

- Front page, Litronic AG changed to SCIOPTA Systems AG
- Chapter 2 Installation added.

14.3 Manual Version 1.2

- Chapter 2 Sciopta Device Driver Concept, former chapter “Standard SDD Descriptor Structure” removed from this chapter.
- Chapter 2.1 Sciopta Device Driver Concept, Overview, former chapters 2.1 and 2.2 merged into one chapter.
- Chapter 2.2 SDD Objects, new chapter.
- Chapter 2.2.1 SDD Object Descriptors, former chapter SDD Descriptors modified and rewritten.
- Chapter 2.2.2 Specific SDD Object Descriptors, rewritten.
- Chapter 2.3 Registering Devices, function sddManAdd added.
- Chapter 2.4 Using Devices, function sddManGetByName added.
- Chapter 3 Using SCIOPTA Device Drivers, header renamed from System Design. Descriptions of the device descriptor structures removed and included in new chapter.
- Former chapter 3.5.4.6 SDD_MAN_INFO, removed.
- Former chapters 3.5.4.7 and 3.6.4.7 SDD_OBJ_INFO, removed.
- Former chapter 3.5.6, process supervision removed.
- Chapter 4 Structures, new chapter.
- Chapter 4.3 SDD Object Size Structure sdd_size_t, added.
- Chapter 4.4 NEARPTR and FARPTR, added.
- Chapter 5 Message Interface Reference, messages SDD_FILE_RESIZE and SDD_FILE_SEEK removed from this manual to file system manual. Messages SDD_NET_RECEIVE, SDD_NET_RECEIVE_URGENT and SDD_NET_SEND removed from this manual to IPS internet protocols manual.
- Chapter 5 Message Interface Reference, whole chapter redesigned.
- Former chapter 5.12 SDD_MAN_INFO / SDD_MAN_INFO_REPLY, removed.
- Former chapter 5.15 SDD_OBJ_INFO, removed.
- Chapters 5.15 SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY, 5.16 SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY and 5.17 SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY, added.
- Chapter 6 Function Interface Reference, functions sdd_fileResize and sdd_fileSeek removed from this manual to file system manual. All network device functions sdd_net* removed from this manual to IPS internet protocols manual.
- Chapter 6 Function Interface Reference, whole chapter redesigned.

- Chapters 6.10 sdd_manGetByPath, 6.17 sdd_objFree, 6.19 sdd_objSizeGet, 6.20 sdd_objTimeGet and 6.21 sdd_objTimeSet, added.
- Former chapter 6.16 sdd_objInfo, removed.
- Former chapter 6.17 sdd_objRelease, removed (replaced by sdd_objFree).

14.4 Manual Version 1.1

- All sdd_obj_t * changed to sdd_obj_t NEARPTR to support SCIOPTA 16 Bit systems.
- All sdd_netbuf_t * changed to sdd_netbuf_t NEARPTR to support SCIOPTA 16 Bit systems.

14.5 Manual Version 1.0

- Initial version.

15 Index

F

G

H

I

K

L

S

T

U

W