High Performance
Real-Time Operating Systems

Device Driver

User's Guide

Copyright

Disclaimer

SCIOPTA Systems AG, makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability of fitness for any particular purpose. Further, SCIOPTA Systems AG, reserves the right to revise this publication and to make changes from time to time in the contents hereof without obligation to SCIOPTA Systems AG to notify any person of such revision or changes.

Trademark

SCIOPTA is a registered trademark of SCIOPTA Systems AG.

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. S02280RL1

SCIOPTA

Table of Contents

1 Introduction

1.1 SCIOPTA Real-Time Operating Systems

SCIOPTA is a high-performance real-time operating system for a variety of target processors. The operating system environment includes:

The operating system environment includes:

- KRN - Pre-emptive Multi-Tasking Real-Time Kernel
- BSP - Board Support Packages
- IPS - Internet Protocols (TCP/IP)
- IPS Applications - Internet Protocols Applications (Web Server, TFTP, DNS, DHCP, Telnet, SMTP etc.)
- SFATFS - FAT File system
- SFFS - FLASH File system, NOR
- SFFSN - FLASH File system, NAND support
- USBD - Universal Serial Bus, Device
- USBH - Universal Serial Bus, Host
- DRUID - System Level Debugger
- SCIOPTA PEG - Embedded GUI
- CONNECTOR - support for distributed multi-CPU systems
- SMMS - Support for MMU
- SCAPI - SCIOPTA API on Windows or LINUX host
- SCSIM - SCIOPTA Simulator

1.2 About this Manual

SCIOPTA device drivers follow a clear and easy to understand concept. This manual includes a description of the SCIOPTA device driver concept and gives an introduction how to use and write **SCIOPTA** device drivers. Detailed descriptions of the device driver structure and interfaces are included.

In the reference part all device driver messages and functions are listed.

Please consult also the other **SCIOPTA** manuals.

2 Installation

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

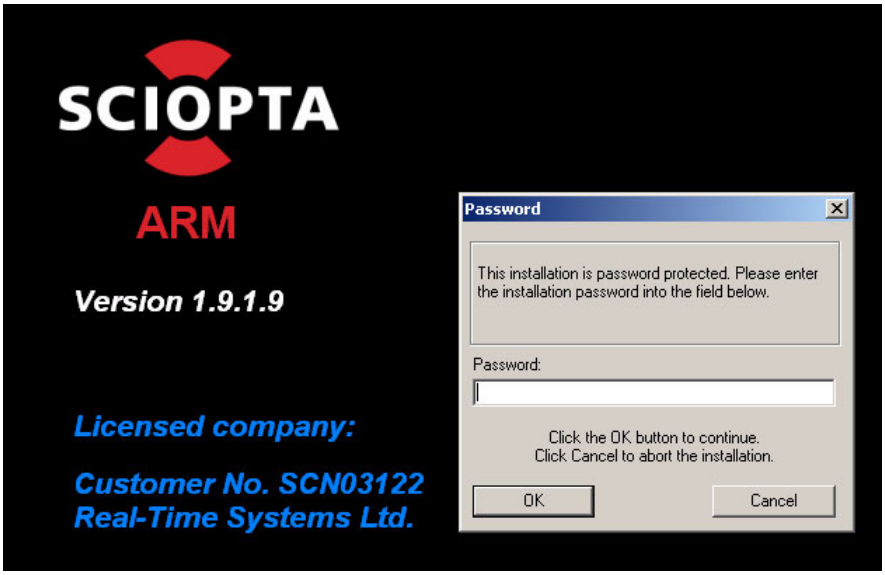

Figure 2-1: Main Installation Window

3 SCIOPTA Device Driver Concept

3.1 Overview

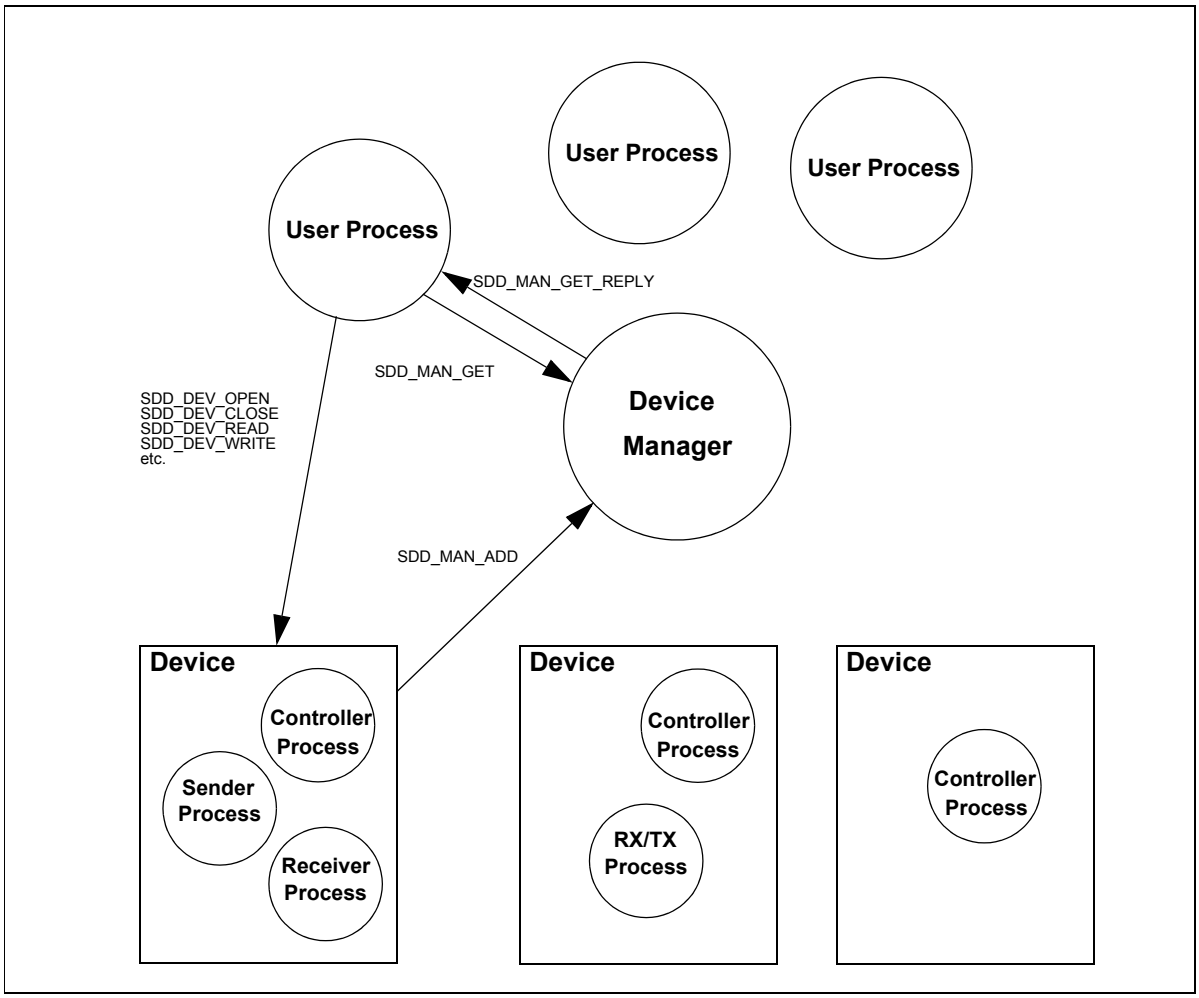

Figure 2-1: SCIOPTA Device Driver Concept

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 and file descriptor interface (posix) 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.

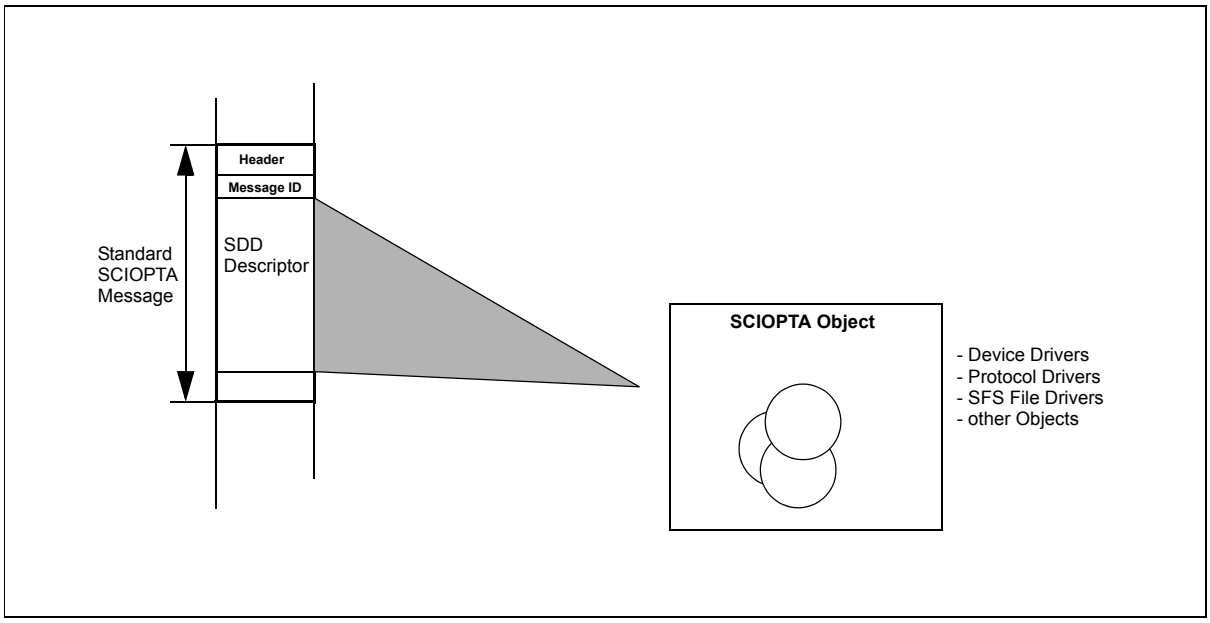

Figure 3-1: SDD Descriptor

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 ID(s) 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

There are three different interfaces which can be used to access the SCIOPTA device driver functionality.

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 Device Driver Function Interface is a function layer on top of the message interface. The message handling and event control are encapsulated within these functions.

Another convenient way is to use the Posix File Descriptor Interface as it is a standardized API.

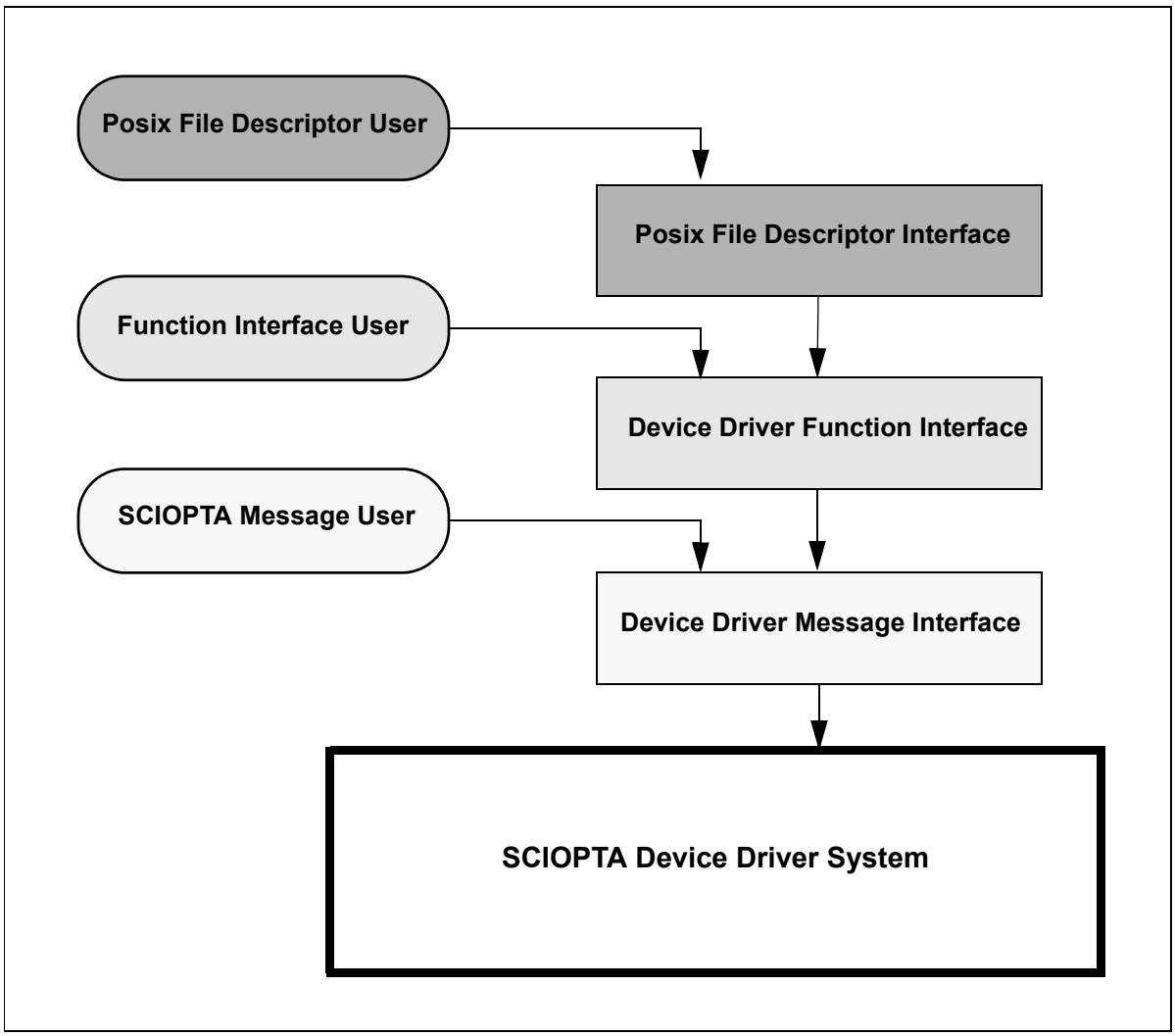

Figure 3-2: SCIOPTA Device Driver System API

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.

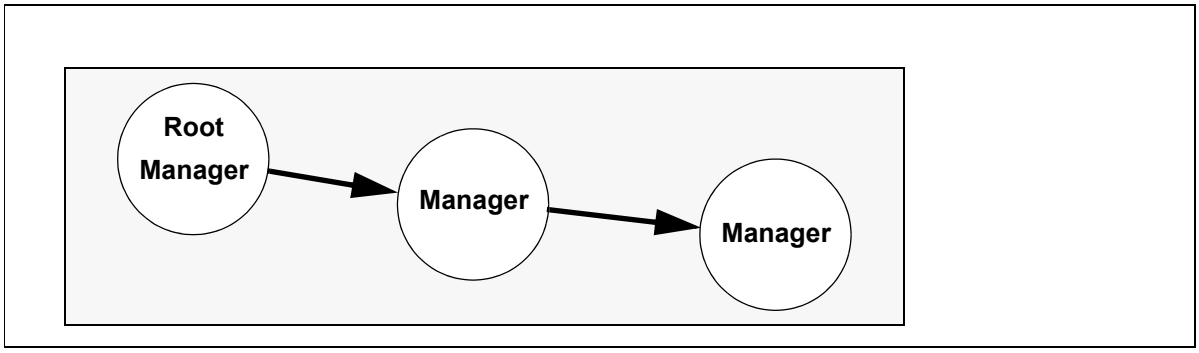

Figure 3-3: SCIOPTA Hierarchical Structured Managers

3.7 Board Support Packages

The description of the board support packages are included in the SCIOPTA Target Manuals for the specific processors. For each officially supported board you will find there:

- Short description of the board including a list of the features.
- Photograph of the board.

List and description of all

- source files for the board setup.
- include files for the board setup.
- project files for the board setup.
- processes of the board setup including information about the process configuration.
- hooks for the board setup.
- source files for all device drivers.
- include files for all device drivers.
- project files for all device drivers.
- processes of the device drivers including information about the process configuration.
- hooks for the device drivers.

4 Using SCIOPTA Device Drivers

4.1 SDD Objects

Please consult chapter 3.2 “SDD Objects” on page 3-2 for more information about SCIOPTA SDD objects.

4.2 Device Driver Application Programmers Interface

There are three different interfaces which can be used to access the SCIOPTA Device Driver functionality.

The SCIOPTA Device Driver Concept 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 devices. Please consult chapter 4.3 “Using the Device Driver Message Interface” on page 4-2 for more information. A detailed description of the messages can be found in chapter 6 “Message Interface Reference” on page 6-1.

The Device Driver Function Interface is a function layer on top of the message interface. The message handling and event control are encapsulated in these functions. Please consult chapter 4.4 “Using the Device Driver Function Interface” on page 4-8 for more information. A detailed list and description of the Function Calls can be found in chapter 7 “Function Interface Reference” on page 7-1.

If you are familiar and comfortable with the POSIX function calls you can use the Device Driver File Descriptor Interface as it meets the POSIX standards. These calls are not yet documented in this manual version.

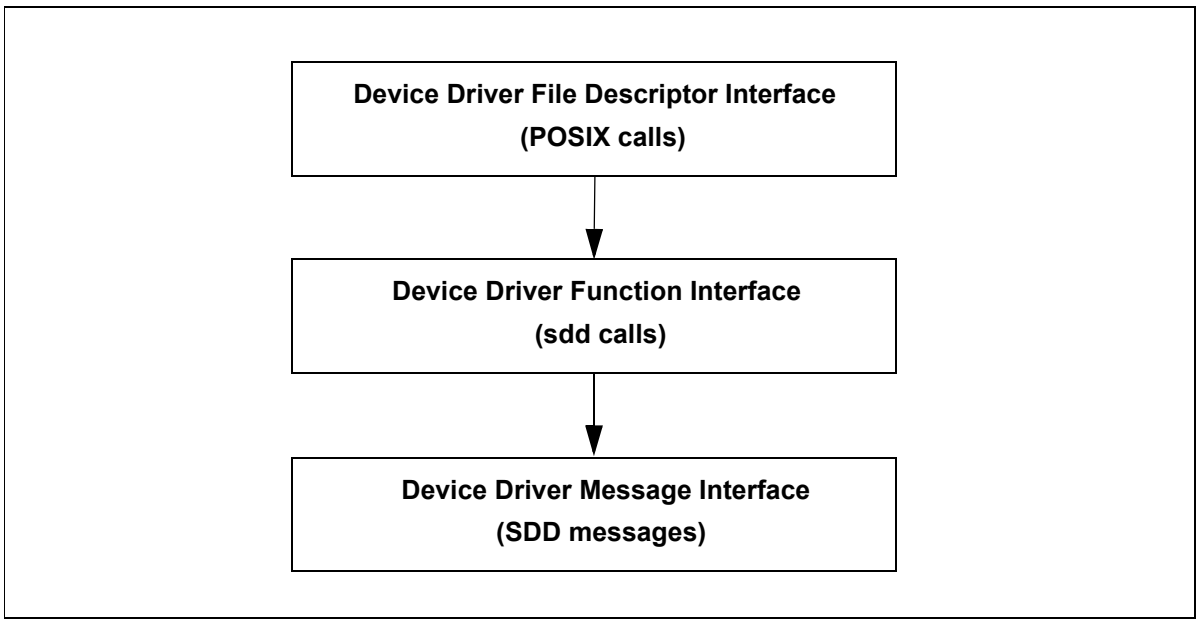

Figure 4-1: SCIOPTA Device Driver API

4.3 Using the Device Driver Message Interface

4.3.1 Introduction

We will just give some simple examples how to use the device driver message interface in a SCIOPTA device driver system. Please consult the SCIOPTA - Kernel, User's Guide and Reference Manual for information how to define and use SCIOPTA messages.

4.3.2 Register a Device

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.

4.3.3 Writing Data Using the SDD Message Interface

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.

4.3.4 Message Sequence Chart Register and Use of a Device

This chart shows a typical MSC where a device will first be registered by a device driver to a manager process. A user process can then use the device (here: getting data from the device). It is also shown that the device generates an error message if a user tries to read from an already closed device.

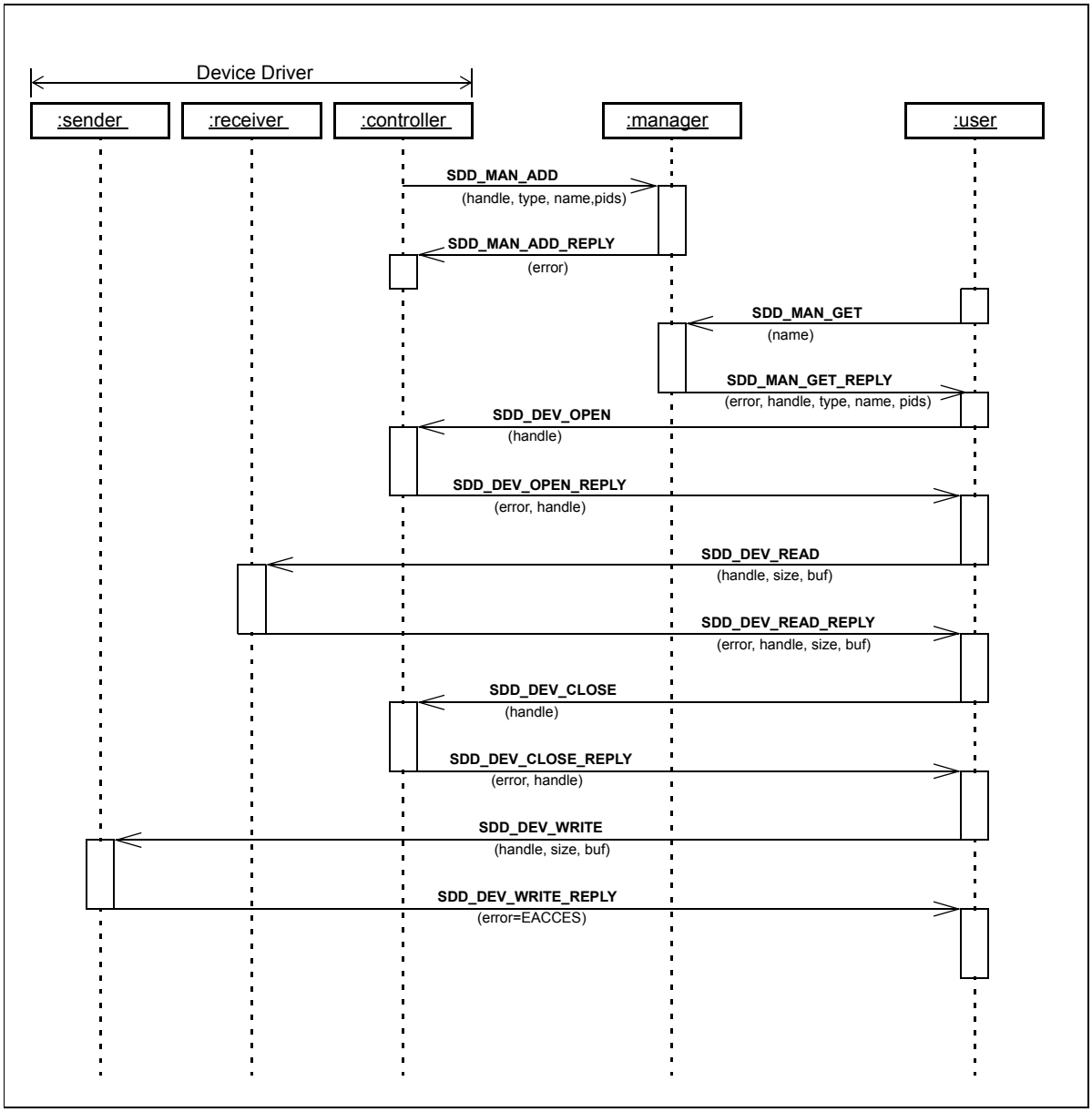

Figure 4-2: MSC Adding a Device and Use It

4.3.5 Using Hierarchical Managers

This chart shows a MSC with two managers. There is a root manager (rootMgr) and a device manager (deviceMgr).

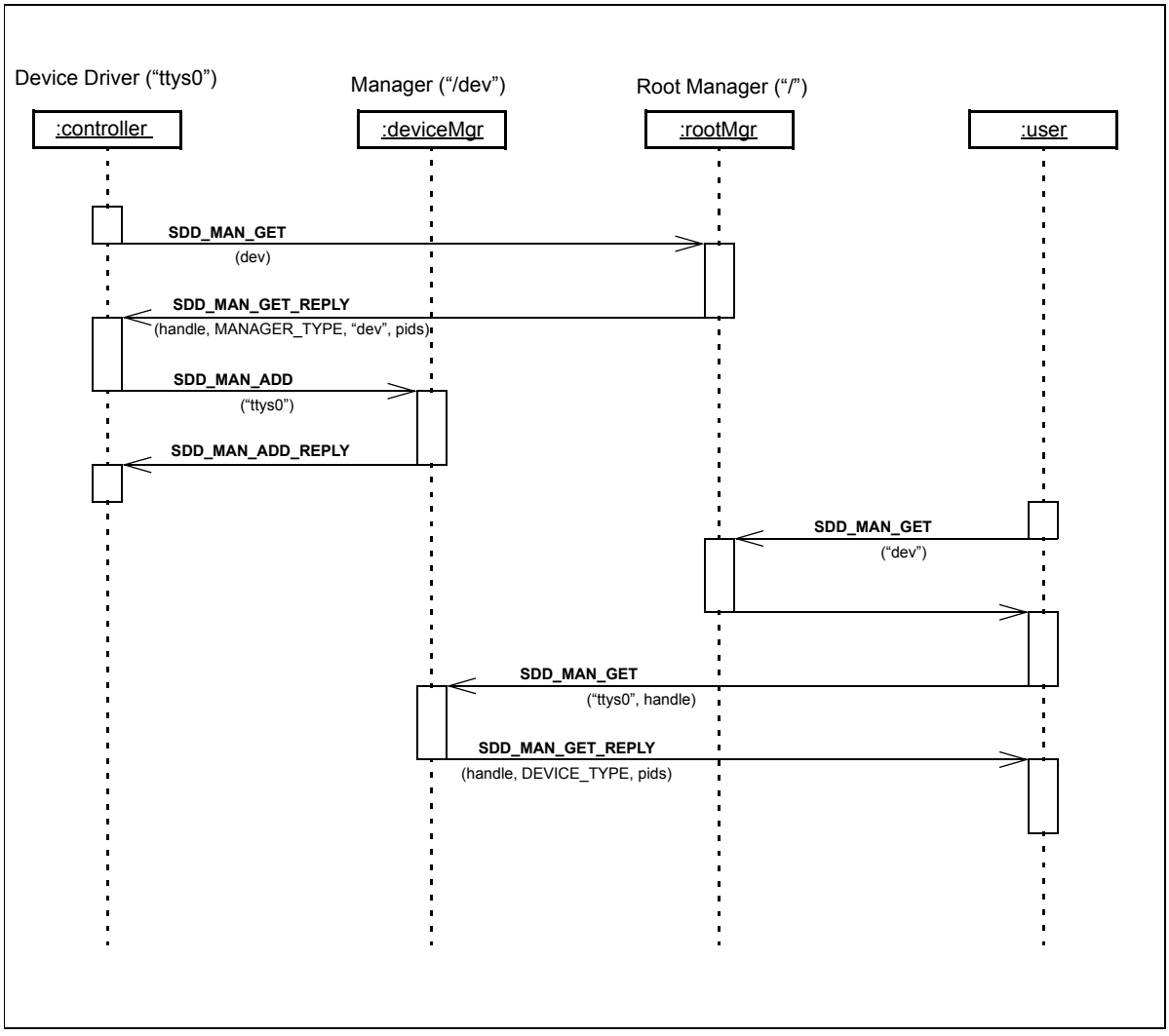

Figure 4-3: MSC Using Hierarchical Managers

4.4 Using the Device Driver Function Interface

4.4.1 Registering a Device

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.

4.4.2 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)
   ```

4.5 Device Manager

4.5.1 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.

4.5.2 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 **4.5.5 "Hierarchical Structured Managers" on page 4-12**) the root manager is the top level and reference manager.

4.5.3 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.

4.5.4 Message Handling in Managers

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

4.5.4.1 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 **6.8 “SDD_MAN_ADD / SDD_MAN_ADD_REPLY” on page 6-12** for the message description.

4.5.4.2 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 **6.12 “SDD_MAN_RM / SDD_MAN_RM_REPLY” on page 6-16** for the message description.

4.5.4.3 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 **6.9 “SDD_MAN_GET / SDD_MAN_GET_REPLY” on page 6-13** for the message description.

4.5.4.4 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 **6.10 “SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY” on page 6-14** for the message description.

4.5.4.5 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 **6.11 “SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY” on page 6-15** for the message description.

4.5.5 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.

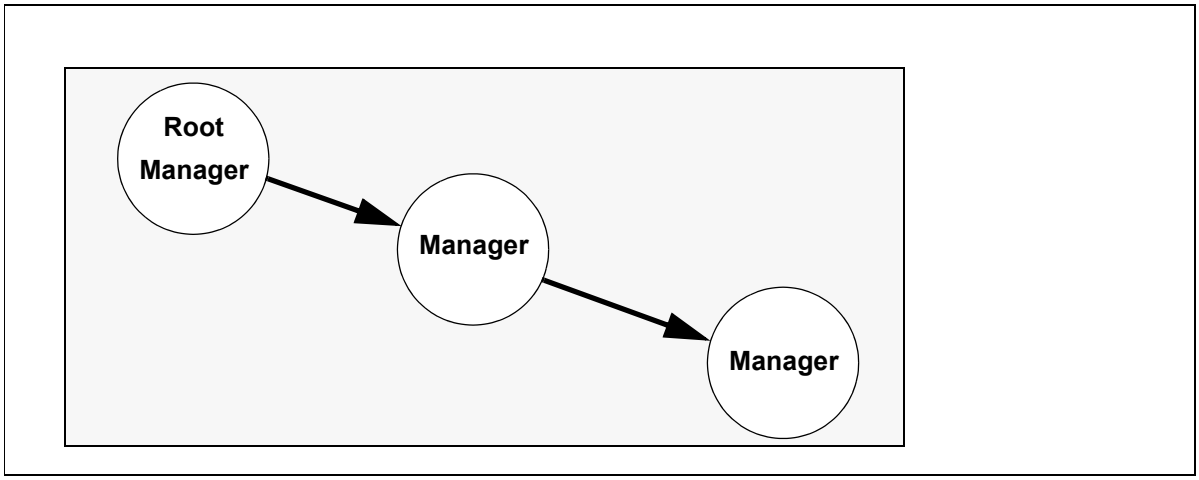

Figure 4-4: SCIOPTA Hierarchical Structured Managers

The main advantage of using hierarchical structured managers is that the SCIOPTA Device Driver File Descriptor Interface (posix calls) 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>)).

4.5.6 On-The-Fly Objects

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

An on-the-fly object is not registering itself at the manager as usual SDD Objects (device drivers) will do. On-the-fly objects are mainly used in file systems such as SCIOPTA SFS.

To remove an on-the-fly 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 on-the-fly objects without releasing (freeing) them. See also chapter 7.17 “sdd_objFree” on page 7-31)

4.5.7 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.

4.5.8 Example of a SCIOPTA Device Manager

The following code gives an example of a manager process in a typical SCIOPTA device driver system.

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_objInfo_t info;
  sdd_obj_t object;
  sdd_manInfo_t managerInfo;
  sc_procPathGetMsgReply_t nameGet;
};
typedef struct __pidEntry_s {
  sc_msgid_t id;
  int ref;
  sc_pid_t pid;
} __pidEntry_t;
static int
entryNameOrderCmp (void NEARPTR left, void NEARPTR right)
{
  return strncmp ((const char *)((sdd_obj_t NEARPTR) left)->name,
          (const char *)((sdd_obj_t NEARPTR) right)->name, SC_NAME_MAX);
}
static void
entryDel (void NEARPTR e)
{
  if (e)
    sc_msgFree ((sc_msgptr_t) &e);
}
SC_PROCESS (SCP_manager)
{
  sc_msg_t msg;
  sc_pid_t to;
  logd_t NEARPTR logd;
  list_t NEARPTR names;
  int s1, s2;
  sc_msg_t e;
  sc_poolid_t pool = SC_DEFAULT_POOL;
  sc_ticks_t timeout = SC_FATAL_IF_TMO;
  logd = logd_new ("/SCP_logd", LOGD_LEVEL_MAX, "manager", SC_DEFAULT_POOL,
           SC_FATAL_IF_TMO);
  msg = sc_procPathGet (SC_CURRENT_PID,0);
  logd_printf (logd, LOGD_INFO, "manager %s start\n",msg->nameGet.path);
  sc_msgFree(&msg);
  names = list_new (entryNameOrderCmp, entryDel, pool, timeout);
  for (;;) {
    msg = sc_msgRx (SC_ENDLESS_TMO, SC_MSGRX_ALL, SC_MSGRX_MSGID);
    to = sc_msgSndGet (&msg);
    switch (msg->id) {
    case SDD_ERROR:
      /* ignore */
      logd_printf (logd, LOGD_INFO, "rev'd SDD_ERROR !\n");
      sc_msgFree(&msg);
      continue;
    case SDD_MAN_ADD:
      {
    if (names->ops->add (names, msg)) {
      msg = sc_msgAllocClr (sizeof (sdd_obj_t), SDD_MAN_ADD, pool,
                timeout);
//->       msg->base.error = 0;
    }
    else {
      /* this means this name allready exist. Names musst be unique in
         given namespace (namespace is the name of the manager).
       */
      msg->base.error = EEXIST;
      /* set the wrong entry to 0 */
      msg->object.name[0] = 0;
    }
      }
      break;
    case SDD_MAN_RM:
      if ((e = names->ops->rm (names, msg))) {
    sc_msgFree (&msg);
    msg = e;
    msg->id = SDD_MAN_RM;
      }
```

```
      else {
    msg->base.error = ENOENT;
      }
      break;
    case SDD_MAN_GET:
      if ((e = names->ops->get (names, msg))) {
    s1 = sc_msgSizeGet (&e);
    s2 = sc_msgSizeGet (&msg);
    if (s1 < s2) {
      msg->base.error = EFAULT;
    }
    else {
      memcpy (msg, e, s2);
      msg->base.error = 0;
    }
    msg->id = SDD_MAN_GET;
      }
      else {
    msg->base.error = ENOENT;
      }
      break;
    case SDD_MAN_GET_FIRST:
      if ((e = names->ops->getFirst (names))) {
    s1 = sc_msgSizeGet (&e);
    s2 = sc_msgSizeGet (&msg);
    if (s1 < s2) {
      msg->base.error = EFAULT;
    }
    else {
      memcpy (msg, e, s2);
      msg->base.error = 0;
    }
    msg->id = SDD_MAN_GET_FIRST;
      }
      else {
    msg->base.error = ENOENT;
      }
      break;
    case SDD_MAN_GET_NEXT:
      if ((e = names->ops->getNext (names, msg))) {
    s1 = sc_msgSizeGet (&e);
    s2 = sc_msgSizeGet (&msg);
    if (s1 < s2) {
      msg->base.error = EFAULT;
    }
    else {
      memcpy (msg, e, s2);
      msg->base.error = 0;
    }
    msg->id = SDD_MAN_GET_NEXT;
      }
      else {
    msg->base.error = ENOENT;
      }
      break;
    case SDD_OBJ_INFO:
      msg->managerInfo.info.ref = 1; /* presistent */
      msg->managerInfo.noOfItems = names->ops->noOfItems (names);
      break;
    case SDD_OBJ_DUP:
      msg->base.error = 0;
      break;
    default:
      msg->base.error = SC_ENOTSUPP;
      break;
    }
    ++msg->id;
    sc_msgTx (&msg, to, 0);
  }
}
```

4.6 Device Driver

4.6.1 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).

4.6.2 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.

4.6.3 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 the Message Interface is 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 **6.8 “SDD_MAN_ADD / SDD_MAN_ADD_REPLY” on page 6-12**.

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 **7.8 “sdd_manAdd” on page 7-13**). 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 **7.14 “sdd_manGetRoot” on page 7-25**).

4.6.4 Message Handling in Device Drivers

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

4.6.4.1 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 **6.4 "SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY" on page 6-5** for the message description.

4.6.4.2 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 **6.2 "SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY" on page 6-1** for the message description.

4.6.4.3 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 **6.5 "SDD_DEV_READ / SDD_DEV_READ_REPLY" on page 6-7** for the message description.

4.6.4.4 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 **6.6 "SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY" on page 6-9** for the message description.

4.6.4.5 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 **6.3 "SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY" on page 6-3** for the message description.

4.6.4.6 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 **6.13 "SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY" on page 6-17** for the message description.

4.6.4.7 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 **6.14 "SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY" on page 6-18** for the message description. It is good practice to release any object before free-ing it.

4.6.4.8 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 **6.7 "SDD_ERROR" on page 6-11** for the message description.

4.6.5 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 7.9 “sdd_manGetByName” on page 7-15) function call. The manager will return the SDD device descriptor including the handle.

4.6.6 Example of a Random Device Driver

```
#include <sciopta.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <sys/errno.h>
#include <sdd/sdd.h>
#include <sdd/sdd.msg>
#include <logd/logd.h>
#include <dev/randdev.h>
/** Local definitions & implementations
*/
union sc_msg {
  sc_msgid_t id;
  sdd_obj_t object;
  sdd_baseMessage_t base;
  sdd_objInfo_t info;
  sdd_devOpen_t open;
  sdd_devRead_t read;
  sdd_devWrite_t write;
  sdd_devIoctl_t ioctl;
  sdd_fileSeek_t seek;
  sdd_fileResize_t resize;
};
typedef struct fildev_s {
  sc_msgid_t id;
  int pos;
} fildev_t;
static void register_dev(const char *name, logd_t NEARPTR logd)
{
  /* registration */
  sdd_obj_t NEARPTR dev;
  sdd_obj_t NEARPTR man;
  dev = (sdd_obj_t NEARPTR) sc_msgAlloc (sizeof (sdd_obj_t), 0,
                                         SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  dev->base.error = 0;
  dev->base.handle = NULL_HANDLE;
  dev->type = SDD_OBJ_TYPE | SDD_DEV_TYPE | SDD_FILE_TYPE;
  strncpy (dev->name, name, SC_NAME_MAX);
  dev->controller = dev->receiver = dev->sender = sc_procIdGet(NULL,SC_NO_TMO);
  /* register to dev man */
  man = sdd_manGetRoot ("/SCP_devman", "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  if (man) {
    if (sdd_manAdd (man, &dev)) {
      logd_printf (logd, LOGD_SEVERE, "Could not add this device \n");
      sc_procKill (SC_CURRENT_PID, 0);
    }
    sdd_objFree (&man);
  }
  else {
    logd_printf (logd, LOGD_SEVERE, "Could not get /SCP_devman\n");
    sc_procKill (SC_CURRENT_PID, 0);
  }
}
/** Mother process */
SC_PROCESS(randdev)
{
  sc_msg_t msg;
  sc_pid_t to;
  int ref = 0;
  logd_t NEARPTR logd;
  logd =
    logd_new ("/SCP_logd", LOGD_INFO, "random", SC_DEFAULT_POOL,
          SC_FATAL_IF_TMO);
  register_dev("random",logd);
  for (;;) {
    msg = sc_msgRx (SC_ENDLESS_TMO, SC_MSGRX_ALL, SC_MSGRX_MSGID);
    to = sc_msgSndGet (&msg);
    switch (msg->id) {
    case SDD_DEV_CLOSE:
      {
    --ref;
    if (ref < 0) {
      sc_miscError (SDD_ERR_BASE + SC_EREFNO, 0);
    }
```

```
  if (ref == 0) {
    /* close it */
  }
  logd_printf (logd, LOGD_FINE, "%8x decr: Reference %d\n", to, ref);
    }
    break;
  case SDD_OBJ_INFO:
    {
  msg->info.ref = ref;
    }
    break;
  case SDD_OBJ_DUP:
  case SDD_DEV_OPEN:
    {
  ++ref;
  if (ref < 0) {
    sc_miscError (SDD_ERR_BASE + SC_EREFNO, 0);
  }
  msg->open.base.error = 0;
  msg->open.base.handle = NULL_HANDLE;
    }
    break;
  case SDD_DEV_READ:
    {
  msg->read.base.error = 0;
  if (!msg->read.outlineBuf) {
    msg->read.outlineBuf = msg->read.inlineBuf;
  }
  srand ((unsigned int)sc_tickGet ());
  msg->read.curpos = 0;
  while (msg->read.curpos < msg->read.size) {
    msg->read.outlineBuf[msg->read.curpos] = (__u8)(rand() & 255);
    ++msg->read.curpos;
  }
    }
    break;
  case SDD_DEV_WRITE:
    {
  msg->write.base.error = 0;
    }
    break;
  case SDD_FILE_SEEK:
    {
  msg->seek.base.error = 0;
    }
    break;
  case SDD_DEV_IOCTL:
    {
  if (!msg->ioctl.outlineArg) {
    msg->ioctl.outlineArg = (__ptrsize_t) msg->ioctl.inlineArg;
  }
  switch (msg->ioctl.cmd) {
  default:
    msg->base.error = EINVAL;
    break;
  }
    }
    break;
  default:
    msg->base.error = SC_ENOTSUPP;
    break;
  }
  ++msg->id;
  sc_msgTx (&msg, to, 0);
}
```

5 Structures

5.1 Base SDD Object Descriptor Structure sdd_baseMessage_t

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.

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

Members

id

Standard SCIOPTA message ID.

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.

Header

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

5.2 Standard SDD Object Descriptor Structure sdd_obj_t

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**.

```
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;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (see chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1**).

manager

Contains a 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 instructions. The values determine the type of messages which are handled by the SDD object.

This member can be one or more of the following values:

Value	Meaning
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

Contains the name of the SDD object. The name must be unique within a domain. A manager corresponds to a domain.

controller

The controller process ID of the SDD object.

sender

The sender process ID 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

The receiver process ID 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.

Remarks

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.

Header

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

5.3 SDD Object Size Structure sdd_size_t

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

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

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.

Header

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

5.4 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)**.

6 Message Interface Reference

6.1 Introduction

In this chapter all SCIOPTA device driver messages are described. Please consult chapter **4.3 "Using the Device Driver Message Interface" on page 4-2** for information how to use the SDD message interface.

Only the generic device driver messages are listed.

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

Please consult the SCIOPTA File System, User's Guide 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.

6.2 SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY

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 **6.14 "SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY" on page 6-18**) to the device or object. This is mainly used to clean on-the-fly created objects. Please consult chapter **4.5.6 "On-The-Fly Objects" on page 4-13** for more information about on-the-fly objects.

Message IDs

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

sdd_devClose_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1** for type information.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EIO	An input/output error occurred.
SC_ENOTSUPP	This request is not supported.

Header

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

6.3 SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY

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.

Message IDs

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

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;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 5-1** for type information.

cmd

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 request message.

ret

Not used in the request message. In the request message this member contains a driver specific value or if a value of minus one if an error was encountered.

outlineArg

Contains a command specific argument. If the member **inlineArg** is used this member must be set to zero. If the value is nonzero this member contains a direct argument. It can also contain a pointer to an argument but this is not recommended as it is not good design practice to use pointers in messages.

inlineArg

Contains a variable sized command specific argument if **outlineArg** is not used. The full argument is included in the message.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

Header

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

6.4 SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY

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.

Message IDs

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

sdd_devOpen_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 5-1** for type information.

flags

Used by the request message and contains BSD conform flags.

This member can be one of the following values:

Value	Meaning
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.

Not modified by the device driver and therefore contains the same value in the request message.

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!).

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOMEM	Not enough memory to open.
SC_ENOTSUPP	This request is not supported.

Header

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

6.5 SDD_DEV_READ / SDD_DEV_READ_REPLY

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.

Message IDs

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

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;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 5-1** for type information.

size

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

Contains the index of the last written byte in the reply message but is also often not used.

outlineBuf

Used by the reply message and can contain the pointer to the read data. If **inlineBuf** is used this member must be set to zero. It is not recommended to use pointers in messages and therefore it is better to use **inlineBuf**. Not used by the request message and can have any value.

inlineBuf

Used by the reply message and contains a variable sized data buffer if **outlineBuf** is not used. The full data is included in the message. Not used by the request message and can have any value.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EIO	An input/output error occurred.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

Header

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

6.6 SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY

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.

Message IDs

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

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;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 5-1** for type information.

size

Contains in the request message the size of the write message buffer. Contains the effective number of written bytes in the request message.

curpos

Not used.

outlineBuf

Used by the request message and can contain the pointer to the data to be written. If **inlineBuf** is used this member must be set to zero. It is not recommended to use pointers in messages and therefore it is better to use **inlineBuf**. Not used by the reply message and can have any value.

inlineBuf

Used by the request message and contains a variable sized write data buffer if **outlineBuf** is not used. The full data is included in the message. Not used by the reply message and can have any value.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure 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.

Header

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

6.7 SDD_ERROR

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 non 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 non existent device.

Message IDs

Request Message **SDD_ERROR**

sdd_error_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1** for type information.

Errors

The following errors can occur. 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.

Value of error	Meaning
SC_ENOPROC	Device or process does not exist.

Header

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

6.8 SDD_MAN_ADD / SDD_MAN_ADD_REPLY

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.

Message IDs

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

sdd_manAdd_t Structure

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

Members

object

SDD object descriptor of the object which will be registered at the manager. Please consult chapter **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** for type information. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EEXIST	Device already exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

Header

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

6.9 SDD_MAN_GET / SDD_MAN_GET_REPLY

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.

Message IDs

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

sdd_manGet_t Structure

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

Members

object

SDD object descriptor of the object which will be get from the manager. Please consult chapter **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** for type information. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

Header

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

6.10 SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY

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).

Message IDs

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

sdd_manGetFirst_t Structure

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

Members

object

SDD object descriptor of the next registered object which will be get from 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.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

Header

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

6.11 SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY

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).

Message IDs

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

sdd_manGetNext_t Structure

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

Members

object

SDD object descriptor of the next registered object which will be get from the manager. Please consult chapter **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** for type information. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOENT	Device does not exists.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

Header

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

6.12 SDD_MAN_RM / SDD_MAN_RM_REPLY

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.

Message IDs

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

sdd_manRm_t Structure

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

Members

object

SDD object descriptor of the object which needs be removed of the manager. Please consult chapter **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** for type information. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
SC_ENOTSUPP	This request is not supported.

Header

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

6.13 SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY

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.

Message IDs

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

sdd_objDup_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1** for type information.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOMEM	Not enough memory.
SC_ENOTSUPP	This request is not supported.

Header

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

6.14 SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY

Description

This message is used to release an on-the-fly object.

An on-the-fly 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 on-the-fly object, this **SDD_OBJ_RELEASE** message must be used. Please consult chapter **4.5.6 "On-The-Fly Objects" on page 4-13** for more information about on-the-fly 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.

Message IDs

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

sdd_objRelease_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1** for type information.

Errors

The following errors can occur. 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.

Value of error	Meaning
ENOENT	Device does not exists.
SC_ENOTSUPP	This request is not supported.

Header

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

6.15 SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY

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.

Message IDs

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

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;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 5-1** for type information.

total

Total size of the SDD object.

free

Free available size of the SDD object.

bad

Not available size of the SDD object.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

Header

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

6.16 SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY

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.

Message IDs

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

sdd_objTime_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1** for type information.

data

Time data in a user defined format.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

Header

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

6.17 SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY

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.

Message IDs

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

sdd_objTime_t Structure

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

Members

base

Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter **5.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 5-1** for type information.

data

Time data in a user defined format.

Errors

The following errors can occur. 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.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
EINVAL	Invalid parameter.
SC_ENOTSUPP	This request is not supported.

Header

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

7 Function Interface Reference

7.1 Introduction

In this chapter all SCIOPTA device driver functions are described. Please consult chapter **4.4 "Using the Device Driver Function Interface" on page 4-8** for a description how to use the function interface.

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 System, 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.

7.2 sdd_devAread

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 user needs to receive explicitly an **SDD_DEV_READ_REPLY** message which contains the data from the device to be read. The caller process will not be blocked and returns immediately.

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.

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

Parameters

self

SDD device descriptor of the device to be read. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

size

Size of the data buffer.

Return Value

This functions actually always returns a zero value.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_devRead_t read;
};
/** Interface implementations
*/
int
sdd_devAread (sdd_obj_t NEARPTR self, ssize_t size)
{
  sc_msg_t msg;
  msg = sc_msgAllocClr (sizeof (sdd_devRead_t) + size * sizeof (unsigned char),
                SDD_DEV_READ,
            sc_msgPoolIdGet ((sc_msgptr_t) &self),
                SC_FATAL_IF_TMO);
  msg->base.handle = self->base.handle;
  msg->read.size = size;
  sc_msgTx (&msg, self->receiver, 0);
  return 0;
}
```

7.3 sdd_devClose

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 on-the-fly created devices. Please consult chapter **4.5.6 "On-The-Fly Objects" on page 4-13** for more information about on-the-fly objects.

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

Parameter

self

SDD device descriptor of the device to be closed. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Headers

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_devClose_t close;
};
/** Interface implementations
*/
int
sdd_devClose (sdd_obj_t NEARPTR self)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_DEV_CLOSE_REPLY, 0
  };
  msg =
    sc_msgAlloc (sizeof (sdd_devClose_t), SDD_DEV_CLOSE,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

7.4 sdd_devIoctl

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.

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

Parameter

self

SDD device descriptor of the device from whom to get and set parameters. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

cmd

Device specific command.

arg

Command specific argument.

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Headers

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_devIoctl_t ioctl;
};
/** Interface implementations
*/
int
sdd_devIoctl (sdd_obj_t NEARPTR self, unsigned int cmd, unsigned long arg)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_DEV_IOCTL_REPLY, 0
  };
  int ret;
  msg =
    sc_msgAlloc (sizeof (sdd_devIoctl_t), SDD_DEV_IOCTL,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  msg->ioctl.cmd = cmd;
  msg->ioctl.ret = 0;
  msg->ioctl.outlineArg = arg;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    ret = msg->ioctl.ret;
    sc_msgFree (&msg);
    return ret;
  }
}
```

7.5 sdd_devOpen

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.

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

Parameter

self

SDD device descriptor of the device to be opened. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

flags

Contains BSD conform flags.

This parameter can be one of the following values:

Value	Meaning
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!).

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Headers

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_devOpen_t open;
};
/** Interface implementations
*/
int
sdd_devOpen (sdd_obj_t NEARPTR self, flags_t flags)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_DEV_OPEN_REPLY, 0
  };
  msg =
    sc_msgAlloc (sizeof (sdd_devOpen_t), SDD_DEV_OPEN,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  msg->open.flags = flags;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  self->base.handle = msg->base.handle;
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

7.6 sdd_devRead

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.

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

Parameter

self

SDD device descriptor of the device to read from. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

buf

Buffer where read data is stored.

ssize

Size of the data buffer.

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_devRead_t read;
};
/** Interface implementations
*/
ssize_t
sdd_devRead (sdd_obj_t NEARPTR self, __u8 *buf, ssize_t size)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_DEV_READ_REPLY, 0
  };
  ssize_t ret;
  msg =
    sc_msgAlloc (sizeof (sdd_devRead_t), SDD_DEV_READ,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  msg->read.size = size;
  msg->read.curpos = 0;
  msg->read.outlineBuf = buf;
  sc_msgTx (&msg, self->receiver, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    ret = msg->read.size;
    sc_msgFree (&msg);
    return ret;
  }
}
```

7.7 sdd_devWrite

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.

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

Parameter

self

SDD device descriptor of the device to write to. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

buf

Buffer where the data to be written is stored.

ssize

Size of the data buffer.

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.
EFBIG	Size of data to be written to big.
SC_ENOTSUPP	This request is not supported.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_devWrite_t write;
};
/** Interface implementations
*/
ssize_t
sdd_devWrite (sdd_obj_t NEARPTR self, const __u8 *buf, ssize_t size)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_DEV_WRITE_REPLY, 0
  };
  int ret;
  msg =
    sc_msgAlloc (sizeof (sdd_devWrite_t), SDD_DEV_WRITE,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  msg->write.size = size;
  msg->write.curpos = 0;
  msg->write.outlineBuf = (__u8 *) buf;
  sc_msgTx (&msg, self->sender, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    ret = msg->write.size;
    sc_msgFree (&msg);
    return ret;
  }
}
```

7.8 sdd_manAdd

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.

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

Parameter

self

SDD manager descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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 object descriptor of the object which will be registered at the manager. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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.

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
};
/** Interface implementations
*/
int
sdd_manAdd (sdd_obj_t NEARPTR self, sdd_obj_t NEARPTR NEARPTR object)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_MAN_ADD_REPLY, 0
  };
  (*object)->base.id = SDD_MAN_ADD;
  (*object)->manager = self->base.handle;
  sc_msgTx ((sc_msgptr_t) object, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

7.9 sdd_manGetByName

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 on-thy-fly devices you should precede the message free by an **sdd_objRelease** function. Please consult chapter **7.17 "sdd_objFree" on page 7-31**.

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

Parameter

self

SDD manager descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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

Path and name of the device.

Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};
sdd_obj_t NEARPTR
sdd_manGetByName (sdd_obj_t NEARPTR self, const char *name)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_MAN_GET_REPLY, 0
  };
  msg =
    sc_msgAlloc (sizeof (sdd_manGet_t), SDD_MAN_GET,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->object.manager = self->base.handle;
  (void)strncpy (msg->object.name, name, SC_NAME_MAX);
  sc_msgTx (&msg, self->controller, 0);
  do {
    if (msg) {
      sc_msgFree (&msg);
    }
    msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  }
  /* could be an early error which is still in the queue */
  while (msg->object.manager != self->base.handle);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return NULL_OBJ;
  }
  else {
    return &msg->object;
  }
}
```

7.10 sdd_manGetByPath

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 path as parameter.

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

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 **7.17 "sdd_objFree" on page 7-31**.

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

Parameter

self

SDD manager descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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.

Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};

sdd_obj_t NEARPTR
sdd_manGetByPath (sdd_obj_t NEARPTR root, const char *path)
{
  const char *name;
  sdd_obj_t NEARPTR folder;

  if (!root) {
    sc_miscErrnoSet (EINVAL);
    return NULL;
  }

  /* open file and read from it at least the elf header */
  folder = sdd_objResolve (root, path, &name);
  if (!folder) {
    sc_miscErrnoSet (ENOENT);
    return NULL;
  }

  if (name && name[0]) {
    return sdd_manGetByName (folder, name);
  }
  else {
    return folder;
  }
}
```

7.11 sdd_manGetFirst

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 7.17 "sdd_objFree" on page 7-31.

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

Parameter

self

SDD manager descriptor. Please consult chapters 5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2 and 5.4 "NEARPTR and FARPTR" on page 5-6 for type information.

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.

Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. Please consult chapters 5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2 and 5.4 "NEARPTR and FARPTR" on page 5-6 for type information. 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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};
sdd_obj_t NEARPTR
sdd_manGetFirst (sdd_obj_t NEARPTR self, int size)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_MAN_GET_FIRST_REPLY, 0
  };
  if (size < sizeof (sdd_obj_t)) {
    size = sizeof (sdd_obj_t);
  }
  msg =
    sc_msgAllocClr (size, SDD_MAN_GET_FIRST,
            sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->object.manager = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return NULL_OBJ;
  }
  else {
    return &msg->object;
  }
}
```

7.12 sdd_manGetNext

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 **7.17 "sdd_objFree" on page 7-31**.

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

Parameter

self

SDD manager descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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.

Return Value

If the functions succeeds the return value is the pointer to the SDD object descriptor of the registered object. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};
sdd_obj_t NEARPTR
sdd_manGetNext (sdd_obj_t NEARPTR self, sdd_obj_t NEARPTR reference,
            int size)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_MAN_GET_NEXT_REPLY, 0
  };
  if (size < sizeof (sdd_obj_t)) {
    size = sizeof (sdd_obj_t);
  }
  msg =
    sc_msgAlloc (size, SDD_MAN_GET_NEXT,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  memcpy (msg, reference, size);
  msg->id = SDD_MAN_GET_NEXT;
  msg->object.manager = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return NULL_OBJ;
  }
  else {
    return &msg->object;
  }
}
```

7.13 sdd_manNoOfItems

The **sdd_manNoOfItems** function is used to get the number of registered devices of the manager device list

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

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

Parameter

self

SDD manager descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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.

Return Value

If the functions succeeds the return value is zero or positive. The returned value contains the number of registered devices of the manager device list.

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

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
SC_ENOTSUPP	This request is not supported.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
  sdd_manInfo_t managerInfo;
};
int
sdd_manNoOfItems (sdd_obj_t NEARPTR self)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_OBJ_INFO_REPLY, 0
  };
  int noOfItems;
  msg =
    sc_msgAlloc (sizeof (sdd_manInfo_t), SDD_OBJ_INFO,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->object.manager = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    noOfItems = msg->managerInfo.noOfItems;
    sc_msgFree (&msg);
    return noOfItems;
  }
}
```

7.14 sdd_manGetRoot

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

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.

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

Parameter

process

Name of the manager process.

name

Name of the manager (can be different than the process name).

plid

Pool ID from where the SDD-Object will be allocated.

tmo

Timeout.

Return Value

If the functions succeeds the return value is the pointer to the created SDD manager descriptor of the root manager. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

If the function fails the return value is NULL.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};
sdd_obj_t NEARPTR
sdd_manGetRoot (const char *process, const char *name, sc_poolid_t plid,
        sc_ticks_t tmo)
{
  sc_pid_t pid;
  sdd_obj_t NEARPTR tmp;
  pid = sc_procIdGet (process, SC_NO_TMO);
  if (pid != SC_ILLEGAL_PID) {
    tmp = (sdd_obj_t NEARPTR) sc_msgAlloc (sizeof (sdd_obj_t), 0, plid, tmo);
    tmp->base.error = 0;
    tmp->base.handle = NULL_HANDLE;
    tmp->type = SDD_OBJ_TYPE | SDD_MAN_TYPE;
    (void)strncpy (tmp->name, name, SC_NAME_MAX);
    tmp->controller = tmp->sender = tmp->receiver = pid;
    return tmp;
  }
  else {
    return NULL_OBJ;
  }
}
```

7.15 sdd_manRm

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

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.

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

Parameter

self

SDD manager descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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 to remove. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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 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.

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**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
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.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};
int
sdd_manRm (sdd_obj_t NEARPTR self, sdd_obj_t NEARPTR reference, int size)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_MAN_RM_REPLY, 0
  };
  if (size < sizeof (sdd_obj_t)) {
    size = sizeof (sdd_obj_t);
  }
  msg =
    sc_msgAlloc (size, SDD_MAN_RM,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  memcpy (msg, reference, size);
  msg->id = SDD_MAN_RM;
  msg->object.manager = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

7.16 sdd_objDup

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.

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

Parameter

self

SDD object descriptor of the object to duplicate. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Return Value

If the functions succeeds the return value is the pointer to the copied SDD object descriptor. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

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

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to duplicate.
SC_ENOTSUPP	This request is not supported.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_objInfo_t info;
  sdd_objDup_t duplicate;
};
/** Interface implementations
*/
sdd_obj_t NEARPTR
sdd_objDup (sdd_obj_t NEARPTR self)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_OBJ_DUP_REPLY, 0
  };
  int size;
  sc_poolid_t plid;
  if (!self) {
    return NULL_OBJ;
  }
  size = sc_msgSizeGet ((sc_msgptr_t) &self);
  plid = sc_msgPoolIdGet ((sc_msgptr_t) &self);
  if (self->controller == sc_procIdGet (NULL,SC_NO_TMO)) {
    /* If it is me, just do a clone */
    /* This should be save, cause a driver should never use himself */
    sc_msg_t dup = sc_msgAlloc (size, SDD_OBJ_DUP, plid, SC_FATAL_IF_TMO);
    memcpy (dup, self, size);
    return (sdd_obj_t NEARPTR) dup;
  }
  msg = sc_msgAlloc(sizeof (sdd_objDup_t), SDD_OBJ_DUP, plid, SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return NULL_OBJ;
  }
  else {
    sc_msg_t dup = sc_msgAlloc (size, SDD_OBJ_DUP, plid, SC_FATAL_IF_TMO);
    memcpy (dup, self, size);
    dup->base.handle = msg->base.handle;
    sc_msgFree (&msg);
    return (sdd_obj_t NEARPTR) dup;
  }
}
```

7.17 sdd_objFree

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

An on-the-fly 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 an on-the-fly device, this **sdd_objRelease** method must be used. It is good practice to release not only on-the-fly devices but also ordinary devices as described. Please consult also chapter **4.5.6 "On-The-Fly Objects" on page 4-13**.

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.

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

Parameter

self

SDD object descriptor of the object to release and to free. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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.

Return Value

None.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
};

static void
sdd_objRelease (sdd_obj_t NEARPTR self)
{
  sc_msg_t msg;

  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_OBJ_RELEASE_REPLY, 0
  };

  if (self->controller == sc_procIdGet (NULL, 0)) {
    return;
  }
  msg =
    sc_msgAlloc (sizeof (sdd_objRelease_t), SDD_OBJ_RELEASE,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  sc_msgFree (&msg);
}

void
sdd_objFree(sdd_obj_t NEARPTR NEARPTR self)
{
  if (self && *self) {
    sdd_objRelease (*self);
    sc_msgFree((sc_msgptr_t) self);
  }
}
```

7.18 sdd_objResolve

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.

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

Parameter

self

SDD manager descriptor of the starting manager. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

pathname

Path name to the SDD object.

last

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.

Return Value

If the functions succeeds the return value is the pointer to the last SDD manager descriptor in the path. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information.

If the function fails the return value is NULL.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_obj_t object;
};
/** Interface implementations
*/
sdd_obj_t NEARPTR
sdd_objResolve (sdd_obj_t NEARPTR self, const char *pathname,
           const char **last)
{
  const char *p;
  char *pt, name[SC_NAME_MAX];
  unsigned int i;
  sdd_obj_t NEARPTR h, *cur;
  /* exception */
  if (pathname[0] == '/' && pathname[1] == '\0') {
    p = pathname;
    ++p;
    *last = p;
    return self;
  }
  cur = self;
  p = pathname;
  if (*p == '/') {
    ++p;
  }
  else {
    /*
       ** get pwd
     */
  }
  for(;;) {
    /* return start of current part */
    *last = p;
    /* copy over next part */
    pt = name;
    for (i = 0; *p && *p != '/' && i <= SC_NAME_MAX; ++i) {
      *pt++ = *p++;
    }
    *pt = 0;
    /* skip '/' if .. */
    if (*p) {
      ++p;
    }
    /* If last part of the path is a manager it. */
    if (!i && !*p) {
      return cur;
    }
    /* Check if the current part is a manager.
       ** If not, return last manager.
     */
    else if (i) {
      h = sdd_manGetByName (cur, name);
      if (!h || !SDD_IS_A (h, SDD_MAN_TYPE)) {
    if (h) {
      if (h != self) {
        /* do never destroy the self pointer!! */
        sdd_objFree (&h);
      }
    }
    return cur;
      }
      /* ok, got a manager, release previous */
      if (cur != self) {
    sdd_objFree (&cur);
      }
      cur = h;
    }
  }
}
```

7.19 sdd_objSizeGet

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.

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

Parameter

self

SDD object descriptor of the object to get the size from. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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. See chapter **5.3 "SDD Object Size Structure sdd_size_t" on page 5-5** for the description of the structure members.

Return Value

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

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to duplicate.
SC_ENOTSUPP	This request is not supported.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_objSize_t size;
};

int
sdd_objSizeGet (sdd_obj_t NEARPTR obj, sdd_size_t * size)
{
  sc_msg_t msg;

  static const sc_msgid_t sel[3] = { SDD_OBJ_SIZE_GET_REPLY, SDD_ERROR, 0 };

  msg = sc_msgAllocClr (sizeof (sdd_objSize_t), SDD_OBJ_SIZE_GET,
            SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  msg->size.base.handle = obj->base.handle;
  sc_msgTx (&msg, obj->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) sel, SC_MSGRX_MSGID);

  if (msg->size.base.error) {
    sc_miscErrnoSet (msg->size.base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    size->total  = msg->size.total;
    size->free = msg->size.free;
    size->bad = msg->size.bad;
    size->used = msg->size.total - size->free - size->bad;
    return 0;
  }
}
```

7.20 sdd_objTimeGet

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.

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

Parameter

self

SDD object descriptor of the object to get the time from. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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.

Return Value

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

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to duplicate.
SC_ENOTSUPP	This request is not supported.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_objTime_t time;
};

int
sdd_objTimeGet (sdd_obj_t NEARPTR obj, __u32 data)
{
  sc_msg_t msg;

  static const sc_msgid_t sel[3] = { SDD_OBJ_TIME_GET_REPLY, SDD_ERROR, 0 };

  msg = sc_msgAllocClr (sizeof (sdd_objTime_t), SDD_OBJ_TIME_GET,
            SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  msg->time.base.handle = obj->base.handle;
  sc_msgTx (&msg, obj->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) sel, SC_MSGRX_MSGID);

  if (msg->time.base.error) {
    sc_miscErrnoSet (msg->time.base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    data = msg->time.data;
    sc_msgFree (&msg);
    return 0;
  }
}
```

7.21 sdd_objTimeSet

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.

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

Parameter

self

SDD object descriptor of the object to set the time to. Please consult chapters **5.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 5-2** and **5.4 "NEARPTR and FARPTR" on page 5-6** for type information. 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.

Return Value

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

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to duplicate.
SC_ENOTSUPP	This request is not supported.

Header

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

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_objTime_t time;
};

int
sdd_objTimeSet (sdd_obj_t NEARPTR obj, __u32 data)
{
  sc_msg_t msg;

  static const sc_msgid_t sel[3] = { SDD_OBJ_TIME_SET_REPLY, SDD_ERROR, 0 };

  msg = sc_msgAllocClr (sizeof (sdd_objTime_t), SDD_OBJ_TIME_GET,
            SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  msg->time.data = data;
  msg->time.base.handle = obj->base.handle;
  sc_msgTx (&msg, obj->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) sel, SC_MSGRX_MSGID);

  if (msg->time.base.error) {
    sc_miscErrnoSet (msg->time.base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    return 0;
  }
}
```

8 Errors

8.1 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

8.2 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

9 Document Revisions

9.1 Manual Version 2.0

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

9.2 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).

9.3 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.

9.4 Manual Version 1.0

- Initial version.

10 Index

R

S

T

U

W