
**High Performance**
**Real-Time Operating Systems**

---

# Kernel V2

# Reference Manual

# Copyright



# Disclaimer



# Trademark



**Headquarters**

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

Document No. S09332RL1

# Table of Contents

# 1 SCIOPTA System

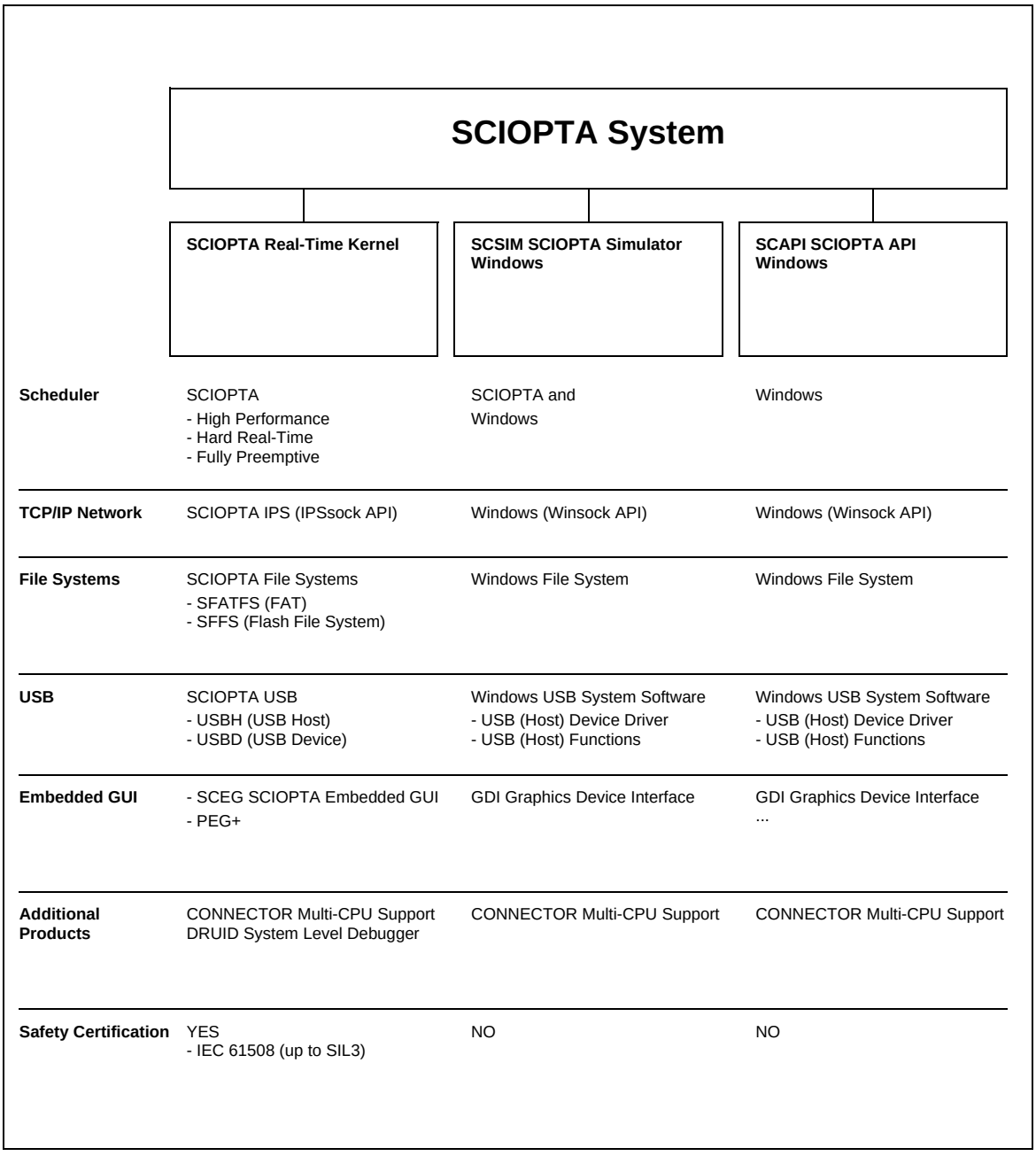


**Figure 1-1: The SCIOPTA System**

## 1.1 The SCIOPTA System

### 1.1.1 SCIOPTA System

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

### 1.1.2 SCIOPTA Real-Time Kernels

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

### 1.1.3 SCIOPTA Simulator and API for Windows

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

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

## 1.2 About This Manual

This SCIOPTA - Kernel V2, Reference Manual contains a complete description of all system calls and error messages.

Please consult the SCIOPTA - Kernel V2, User's Manual for detailed information about the SCIOPTA installation, technologies and methods.

## 1.3 Supported Processors

### 1.3.1 Architectures

SCIOPTA - Kernel V2 supports the following processor architectures. Architectures are referenced in this document as **<arch>**):

- ppc (power pc)
- rx (Renesas RX600)

### 1.3.2 CPU Families

SCIOPTA - Kernel V2 supports the following CPU families. CPU families are referenced in this document as **<cpu>**):

| Architecture **<arch>** | CPU Family **<cpu>** | Description |
|---|---|---|
| ppc | mpx5xx | **Freescale PowerPC MPC500**<br>MPC53x, MPC55x, MPC56x and all other derivatives of the Freescale MPC500 family. |
| | mpc5500 | **Freescale PowerPC MPC55xx**<br>MPC5516, MPC5534, MPC5554, MPC5567 and all other derivatives of the Freescale MPC55xx family. |
| | mpc8xx | **Freescale PowerPC PowerQUICC I**<br>MPC823, MPC850, MPC852T, MPC855T, MPC857, MPC859, MPC860, MPC862, MPC866 and all other derivatives of the Freescale MPC8xx family. |
| | mpc82xx | **Freescale PowerPC PowerQUICC II**<br>MPC8250, MPC8255, MPC8260, MPC8264, MPC8265, MPC8266 and all other derivatives of the Freescale MPC82xx family. |
| | mpc83xx | **Freescale PowerPC PowerQUICC II Pro**<br>MPC8313, MPC8314, MPC8315 and all other derivatives of the Freescale MPC83xx family. |
| | mpc52xx | **Freescale PowerPC MPC5200**<br>MobileGT MPC5200 and all other derivatives of the Freescale MPC52xx and 51xx family. |
| | ppc4xx | **AMCC PowerPC 4xx**<br>PowerPC 405, 440, 460 and all other derivatives of the AMCC PowerPC 4xx family. |
| rx | rx | **Renesas RX600**<br>RX610, RX621, RX62N, RX62T and all other derivatives of the Renesas RX600 family. |

## 1.4 SCIOPTA Real-Time Kernel V2

SCIOPTA Real-Time Kernel V2 (Version 2) is the successor to the SCIOPTA standard kernel.

Differences between SCIOPTA Real-Time Kernel V2 and SCIOPTA Real-Time Kernel Standard:

- Module priority: No process inside a module is allowed to have a higher priority than modules's maximum priority.
- Module friendship concept has been removed.
- System calls **sc_procPathGet** and **sc_procNameGet** return NULL if PID valid, but does not exist anymore.
- Time slice for prioritized processes is now full supported and documented.
- The system call **sc_procSliceSet** is only allowed within same module.
- Parameter n in system call **sc_procVarInit** is now real maximum number of process variables. It was n+1 before.
- Calls **sc_msgTx** and **sc_msgTxAlias** sets the activation time.
- Flag **SC_MSGTX_RTN2SND** introduced for **sc_msgTx** and **sc_msgTxAlias** system calls.
- The activation time is saved for **sc_msgRx** in prioritized processes.
- System call **sc_sleep** returns now the activation time.
- The error handling has been modified.
- Error hook API changed.
- Error process and error proxy introduced.
- New system calls:
    - **sc_moduleCreate2** call replaces the call sc_moduleCreate. It gets the module parameters now from a module descriptor block (mdb).
    - **sc_modulePrioGet**. Returns the priority of a module.
    - **sc_moduleStop**. Stops a whole module.
    - **sc_procAtExit**. Register a function to be called if a prioritized process is called.
    - **sc_procAttrGet**. Returns specific process attributes.
    - **sc_procCreate2** call replaces the calls sc_procPrioCreate, procIntCreate and procTimCreate. It gets the process parameters now from a process descriptor block (pdb).
    - **sc_procWakeupDisable**. Disables the wakeup of a timer or interrupt process.
    - **sc_procWakeupEnable**. Enables the wakeup of a timer or interrupt process.
    - **sc_msgHdChk**. Integrity check of message header.
    - **sc_msgFind**. Finds messages which have been allocated or already received.
    - **sc_tickActivationGet**. Returns the tick time of last activation of the calling process.
    - **sc_miscCrc32**. Calculates a 32 bit CRC over a specified memory range.
    - **sc_miscCrc32Contd**. Calculates a 32 bit CRC over an additional memory range.

# 2 System Calls Overview

## 2.1 Introduction

This chapter list all SCIOPTA system calls in functional groups. Please consult chapter 3 “System Calls Reference” on page 3-1 for an alphabetical list.

## 2.2 Message System Calls

| | |
|---|---|
| sc_msgAlloc | Allocates a memory buffer of selectable size from a message pool.<br>Chapter 3.21 “sc_msgAlloc” on page 3-30. |
| sc_msgAllocClr | Allocates a memory buffer of selectable size from a message pool and initializes the message data to 0.<br>Chapter 3.22 “sc_msgAllocClr” on page 3-33. |
| sc_msgTx | Sends a message to a process.<br>Chapter 3.33 “sc_msgTx” on page 3-53. |
| sc_msgTxAlias | Sends a message to a process by setting any process ID.<br>Chapter 3.34 “sc_msgTxAlias” on page 3-56. |
| sc_msgRx | Receives one ore more defined messages.<br>Chapter 3.29 “sc_msgRx” on page 3-44. |
| sc_msgFree | Returns a message to the message pool.<br>Chapter 3.24 “sc_msgFree” on page 3-36. |
| sc_msgFind | Find a message in the allocated-messages queue of a process.<br>Chapter 3.23 “sc_msgFind” on page 3-34. |
| sc_msgAcquire | Changes the owner of the message. The caller becomes the owner of the message.<br>Chapter 3.19 “sc_msgAcquire” on page 3-26. |
| sc_msgAddrGet | Returns the process ID of the addressee of the message.<br>Chapter 3.20 “sc_msgAddrGet” on page 3-28. |
| sc_msgHdCheck | The message header will be checked for plausibility.<br>Chapter 3.25 “sc_msgHdCheck” on page 3-38. |
| sc_msgHookRegister | Registers a message hook.<br>Chapter 3.26 “sc_msgHookRegister” on page 3-39. |
| sc_msgOwnerGet | Returns the process ID of the owner of the message.<br>Chapter 3.27 “sc_msgOwnerGet” on page 3-41. |
| sc_msgPoolIdGet | Returns the pool ID of a message.<br>Chapter 3.28 “sc_msgPoolIdGet” on page 3-42. |
| sc_msgSizeGet | Returns the size of the message buffer.<br>Chapter 3.30 “sc_msgSizeGet” on page 3-47. |
| sc_msgSizeSet | Modifies the size of a message buffer.<br>Chapter 3.31 “sc_msgSizeSet” on page 3-49. |
| sc_msgSndGet | Returns the process ID of the sender of the message.<br>Chapter 3.32 “sc_msgSndGet” on page 3-51. |

## 2.3 Process System Calls

| | |
|---|---|
| sc_procAtExit | Register a function to be called if a prioritized process is killed.<br>Chapter 3.42 “sc_procAtExit” on page 3-72. |
| sc_procAttrGet | Returns specific process attributes.<br>Chapter 3.43 “sc_procAttrGet” on page 3-73. |
| sc_procCreate2 | Requests the kernel daemon to create process.<br>Chapter 3.44 “sc_procCreate2” on page 3-75. |
| sc_procDaemonRegister | Registers a process daemon which is responsible for pidGet request.<br>Chapter 3.45 “sc_procDaemonRegister” on page 3-82. |
| sc_procDaemonUnregister | Unregisters a process daemon.<br>Chapter 3.46 “sc_procDaemonUnregister” on page 3-83. |
| sc_procHookRegister | Registers a process hook.<br>Chapter 3.47 “sc_procHookRegister” on page 3-84. |
| sc_procIdGet | Returns the process ID of a process.<br>Chapter 3.48 “sc_procIdGet” on page 3-86. |
| sc_procKill | Requests the kernel daemon to kill a process.<br>Chapter 3.49 “sc_procKill” on page 3-88. |
| sc_procNameGet | Returns the full name of a process.<br>Chapter 3.50 “sc_procNameGet” on page 3-90. |
| sc_procObserve | Request a message to be sent if the given process pid dies (process supervision).<br>Chapter 3.51 “sc_procObserve” on page 3-92. |
| sc_procPathCheck | Checks if the construction of a path is correct.<br>Chapter 3.53 “sc_procPathGet” on page 3-95. |
| sc_procPathGet | Returns the path of a process.<br>Chapter 3.53 “sc_procPathGet” on page 3-95. |
| sc_procPpidGet | Returns the process ID of the parent of a process.<br>Chapter 3.54 “sc_procPpidGet” on page 3-97. |
| sc_procPrioGet | Returns the priority of a process.<br>Chapter 3.55 “sc_procPrioGet” on page 3-99. |
| sc_procPrioSet | Sets the priority of a process.<br>Chapter 3.56 “sc_procPrioSet” on page 3-101. |
| sc_procSchedLock | Locks the scheduler and returns the number of times it has been locked before.<br>Chapter 3.57 “sc_procSchedLock” on page 3-103. |
| sc_procSchedUnlock | Unlocks the scheduler by decrementing the lock counter by one.<br>Chapter 3.58 “sc_procSchedUnlock” on page 3-104. |
| sc_procSliceGet | Returns the time slice of a timer process.<br>Chapter 3.59 “sc_procSliceGet” on page 3-105. |
| sc_procSliceSet | Sets the time slice of a timer process.<br>Chapter 3.60 “sc_procSliceSet” on page 3-106. |

| | |
|---|---|
| sc_procStart | Starts a process.<br>Chapter 3.61 “sc_procStart” on page 3-107. |
| sc_procStop | Stops a process.<br>Chapter 3.62 “sc_procStop” on page 3-109. |
| sc_procUnobserve | Cancels the observation of a process.<br>Chapter 3.63 “sc_procUnobserve” on page 3-111. |
| sc_procVarDel | Deletes a process variable.<br>Chapter 3.64 “sc_procVarDel” on page 3-113. |
| sc_procVarGet | Returns a process variable.<br>Chapter 3.65 “sc_procVarGet” on page 3-114. |
| sc_procVarInit | Initializes a process variable area.<br>Chapter 3.66 “sc_procVarInit” on page 3-115. |
| sc_procVarRm | Removes a process variable area.<br>Chapter 3.67 “sc_procVarRm” on page 3-117. |
| sc_procVarSet | Sets a process variable.<br>Chapter 3.68 “sc_procVarSet” on page 3-118. |
| sc_procVectorGet | Returns the interrupt vector of an interrupt process.<br>Chapter 3.69 “sc_procVectorGet” on page 3-119. |
| sc_procWakeupEnable | Enabes the wakeup of a timer or interrupt process.<br>Chapter 3.70 “sc_procWakeupEnable” on page 3-120. |
| sc_procWakeupDisable | Disables the wakeup of a timer or interrupt process.<br>Chapter 3.71 “sc_procWakeupDisable” on page 3-121. |
| sc_procYield | Yields the CPU to the next ready process within the current process's priority group.<br>Chapter 3.72 “sc_procYield” on page 3-122. |

## 2.4 Module System Calls

| | |
|---|---|
| sc_moduleCreate2 | Creates a module.<br>Chapter 3.12 “sc_moduleCreate2” on page 3-12. |
| sc_moduleIdGet | Returns the ID of a module.<br>Chapter 3.13 “sc_moduleIdGet” on page 3-17. |
| sc_moduleInfo | Returns a snap-shot of a module control block.<br>Chapter 3.14 “sc_moduleInfo” on page 3-18. |
| sc_moduleKill | Kills a module.<br>Chapter 3.15 “sc_moduleKill” on page 3-21. |
| sc_moduleNameGet | Returns the full name of a module.<br>Chapter 3.16 “sc_moduleNameGet” on page 3-23. |
| sc_modulePrioGet | Returns the priority of a module.<br>Chapter 3.17 “sc_modulePrioGet” on page 3-24. |
| sc_moduleStop | Stops a whole module.<br>Chapter 3.18 “sc_moduleStop” on page 3-25. |

## 2.5 Message Pool Calls

| | |
|---|---|
| sc_poolCreate | Creates a message pool.<br>Chapter .3.35 “sc_poolCreate” on page 3-58 |
| sc_poolDefault | Sets a message pool as default pool.<br>Chapter 3.36 “sc_poolDefault” on page 3-61. |
| sc_poolHookRegister | Registers a pool hook.<br>Chapter 3.37 “sc_poolHookRegister” on page 3-62. |
| sc_poolIdGet | Returns the ID of a message pool.<br>Chapter 3.38 “sc_poolIdGet” on page 3-64. |
| sc_poolInfo | Returns a snap-shot of a pool control block.<br>Chapter 3.39 “sc_poolInfo” on page 3-65. |
| sc_poolKill | Kills a whole message pool.<br>Chapter 3.40 “sc_poolKill” on page 3-69. |
| sc_poolReset | Resets a message pool in its original state.<br>Chapter 3.41 “sc_poolReset” on page 3-71. |

## 2.6 Timing Calls

| | |
|---|---|
| sc_sleep | Suspends a process for a defined time.<br>Chapter 3.73 “sc_sleep” on page 3-123. |
| sc_tmoAdd | Request a time-out message after a defined time.<br>Chapter 3.80 “sc_tmoAdd” on page 3-131. |
| sc_tmoRm | Remove a time-out job.<br>Chapter 3.81 “sc_tmoRm” on page 3-133. |

## 2.7 System Tick Calls

| | |
|---|---|
| sc_tick | Calls the kernel tick function. Advances the kernel tick counter by 1.<br>Chapter 3.74 “sc_tick” on page 3-125. |
| sc_tickActivationGet | Returns the tick time of last activation of the calling process.<br>Chapter 3.75 “sc_tickActivationGet” on page 3-126. |
| sc_tickGet | Returns the actual kernel tick counter value.<br>Chapter 3.76 “sc_tickGet” on page 3-127. |
| sc_tickLength | Returns/sets the current system tick-length.<br>Chapter 3.77 “sc_tickLength” on page 3-128. |
| sc_tickMs2Tick | Converts a time from milliseconds into ticks.<br>Chapter 3.78 “sc_tickMs2Tick” on page 3-129. |
| sc_tickTick2Ms | Converts a time from ticks into milliseconds.<br>Chapter 3.79 “sc_tickTick2Ms” on page 3-130. |

## 2.8 Process Trigger Calls

| | |
|---|---|
| sc_trigger | Signals a process trigger.<br>Chapter 3.82 “sc_trigger” on page 3-135. |
| sc_triggerValueGet | Returns the value of a process trigger.<br>Chapter 3.83 “sc_triggerValueGet” on page 3-137. |
| sc_triggerValueSet | Sets the value of a process trigger.<br>Chapter 3.84 “sc_triggerValueSet” on page 3-138. |
| sc_triggerWait | Waits on its process trigger.<br>Chapter 3.85 “sc_triggerWait” on page 3-139. |

## 2.9 CONNECTOR Process Calls

| | |
|---|---|
| sc_connectorRegister | Register Registers a connector process.<br>Chapter 3.2 “sc_connectorRegister” on page 3-1. |
| sc_connectorUnregister | Removes a registered connector process.<br>Chapter 3.3 “sc_connectorUnregister” on page 3-3. |

## 2.10 Miscellaneous and Error Calls

| | |
|---|---|
| sc_miscCrc | Calculates a CRC over a specified memory range.<br>Chapter 3.4 “sc_miscCrc” on page 3-4. |
| sc_miscCrcContd | Calculates a CRC over an additional memory range.<br>Chapter 3.5 “sc_miscCrcContd” on page 3-5. |
| sc_miscCrc32 | Calculates a 32 bit CRC over a specified memory range.<br>Chapter 3.6 “sc_miscCrc32” on page 3-6. |
| sc_miscCrc32Contd | Calculates a 32 bit CRC over an additional memory range.<br>Chapter 3.7 “sc_miscCrc32Contd” on page 3-7. |
| sc_miscErrnoGet | Returns the error code.<br>Chapter 3.8 “sc_miscErrnoGet” on page 3-8. |
| sc_miscErrnoSet | Set Sets the error code.<br>Chapter 3.9 “sc_miscErrnoSet” on page 3-9. |
| sc_miscError | Error call.<br>Chapter 3.10 “sc_miscError” on page 3-10. |
| sc_miscErrorHookRegister | Registers an Error Hook.<br>Chapter 3.11 “sc_miscErrorHookRegister” on page 3-11. |

# 3 System Calls Reference

## 3.1 Introduction

This chapter contains a detailed description of all SCIOPTA - Kernel V2 system calls in alphabetical order.

Please consult the SCIOPTA - Kernel V2, User's Manuals for more information on how to use the system calls and about the objects and behaviour of the SCIOPTA system.

## 3.2 sc_connectorRegister

### 3.2.1 Description

This system call is used to register a connector process. The caller becomes a connector process.

Connector processes are used to connect different target in distributed SCIOPTA systems. Messages sent to external processes (residing on remote target or CPU) are sent to the local connector processes.

### 3.2.2 Syntax

```
sc_pid_t sc_connectorRegister (
   int     defaultConn
);
```

### 3.2.3 Parameter

| **defaultConn** | Defines the type of registered CONNECTOR. |
|---|---|
| == 0 | The caller becomes a connector process. The name of the process corresponds to the name of the target. |
| != 0 | The caller becomes the default connector for the system. The name of the process corresponds to the name of the target. |

### 3.2.4 Return Value

Process ID which is used to define the process ID for distributed processes.

### 3.2.5 Example

```
sc_pid_t connid;

connid = sc_connectorRegister(1);
```

### 3.2.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_SYSTEM_FATAL | |
| Caller is not a prioritized process. | e0 = Process type (see pcb.h). |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_SYSTEM_FATAL | |
| Default CONNECTOR is already defined: | e0 = pid |
| Process is already a CONNECTOR: | e0 = 0 |
| KERNEL_ENO_MORE_CONNECTOR | |
| The maximum number of CONNECTORS is reached. | |

## 3.3 sc_connectorUnregister

### 3.3.1 Description

This system call is used to remove a registered connector process. The caller becomes a normal prioritized process.

### 3.3.2 Syntax

```
void sc_connectorUnregister (void);
```

### 3.3.3 Parameter

None

### 3.3.4 Return Value

None

### 3.3.5 Example

```
sc_connectorUnregister;
```

### 3.3.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_CONNECTOR ¦ SC_ERR_SYSTEM_FATAL<br>Caller is not a connector process. | e0 = 0 |

## 3.4 sc_miscCrc

### 3.4.1 Description

This function calculates a 16 bit cyclic redundancy check value (CRC-16-CCITT) over a specified memory range.

The start value of the CRC is 0xFFFF.

### 3.4.2 Syntax

```
__u16 sc_miscCrc (
   __u8            *data,
   unsigned int    len
);
```

### 3.4.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |

### 3.4.4 Return Value

The 16 bit CRC value.

### 3.4.5 Example

```
typedef struct ips_socket_s {
   sc_msgid_t id;
   __u16 srcPort;
   __u16 dstPort;
   ips_addr_t srcIp;
   ips_addr_t dstIp;
   dbl_t list;
}ips_socket_t;

__u16 crc;
ips_socket_t *ref;

crc = sc_miscCrc (ref->srcPort, 4);
```

### 3.4.6 Errors

None

## 3.5 sc_miscCrcContd

### 3.5.1 Description

This function calculates a 16 bit cyclic redundancy check value (CRC-16-CCITT) over an additional memory range.

The variable **start** is the CRC start value.

### 3.5.2 Syntax

```
__u16 sc_miscCrcContd (
   __u8         *data,
   unsigned int len,
   __u16        start
);
```

### 3.5.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |
| **start** | CRC start value. |

### 3.5.4 Return Value

The 16 bit CRC value.

### 3.5.5 Example

```
crc2 = sc_miscCrcContd (ref1->srcPort, 4, crc);
```

### 3.5.6 Errors

None

## 3.6 sc_miscCrc32

### 3.6.1 Description

This function calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3, polynominal: 0x04C11DB7) over a specified memory range.

The start value of the CRC is 0xFFFFFFFF.

### 3.6.2 Syntax

```
__u32 sc_miscCrc32 (
   __u8            *data,
   unsigned int    len
);
```

### 3.6.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |

### 3.6.4 Return Value

The inverted 32 bit CRC value.

### 3.6.5 Example

```
__u32   bcrc;
__u32   burst[4];

bcrc = sc_miscCrc32 (burst, 16);
```

### 3.6.6 Errors

None

## 3.7 sc_miscCrc32Contd

### 3.7.1 Description

This function calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3, polynomial: 0x04C11DB7) over an additional memory range.

The variable **init** is the CRC32 start value.

### 3.7.2 Syntax

```
__u32 sc_miscCrc32Contd (
   __u8         *data,
   unsigned int len,
   __u32        init
);
```

### 3.7.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |
| **init** | CRC32 start value. |

### 3.7.4 Return Value

The inverted 32 bit CRC value.

### 3.7.5 Example

```
__u32   b2crc;
__u32   burst2[4];

b2crc = sc_miscCrc32Contd (burst2, 16, b2crc);
```

### 3.7.6 Errors

None

## 3.8 sc_miscErrnoGet

### 3.8.1 Description

This system call is used to get the process error number (errno) variable.

Each SCIOPTA process has an errno variable. This variable is used mainly by library functions to set the errno variable.

The errno variable will be copied into the observe messages if the process dies.

### 3.8.2 Syntax

```
sc_errcode_t sc_miscErrnoGet (void);
```

### 3.8.3 Parameter

None

### 3.8.4 Return Value

Process error code.

### 3.8.5 Example

```
if ( sc_miscErrnoGet () != 104 ){
   kprintf (0, "Can not connect: %d\n",sc_miscErrnoGet ());
}
```

### 3.8.6 Errors

None

## 3.9 sc_miscErrnoSet

### 3.9.1 Description

This system call is used to set the process error number (errno) variable.

Each SCIOPTA process has an errno variable. This variable is used mainly by library functions to set the errno variable.

The errno variable will be copied into the observe messages if the process dies.

### 3.9.2 Syntax

```
void sc_miscErrnoSet (
   sc_errcode_t    err

);
```

### 3.9.3 Parameter

| | |
|---|---|
| **err** | User defined error code. |

### 3.9.4 Return Value

None.

### 3.9.5 Example

```
sc_miscErrnoSet (ENODEV);
```

### 3.9.6 Errors

None

## 3.10 sc_miscError

### 3.10.1 Description

This system call is used to call the error hooks with an user error.

The SCIOPTA error hooks is usually called when the kernel detects a system error. But the user can also call the error hook and including own error codes and additional information.

This system call will not return if there is no error hook. If an error hook is available the code of the error hook can decide to return or not.

### 3.10.2 Syntax

```
void sc_miscError (
   sc_errcode_t      err,
   sc_extra_t        misc
);
```

### 3.10.3 Parameter

| **err** | User defined error code. |
|---|---|
| <error> | User error code. |
| SC_ERR_SYSTEM_FATAL | Declares error to be system fatal. Must be ored with <error> |
| SC_ERR_MODULE_FATAL | Declares error to be module fatal. Must be ored with <error> |
| SC_ERR_PROCESS_FATAL | Declares error to be process fatal. Must be ored with <error> |

| **misc** | Additional data to pass to the error hook. |
|---|---|

### 3.10.4 Return Value

None.

### 3.10.5 Example

```
sc_miscError (MY_ERR_BASE + MY_ER001, (sc_extra_t) "/SCP_myproc");
```

### 3.10.6 Errors

None

## 3.11 sc_miscErrorHookRegister

### 3.11.1 Description

This system call will register an error hook

Each time a system error occurs the error hook will be called if there is one installed.

### 3.11.2 Syntax

```
sc_errHook_t *sc_miscErrorHookRegister (
   sc_errHook_t       *newhook
);
```

### 3.11.3 Parameter

| **newhook** | Function pointer to the hook. |
|---|---|
| <fptr> | Function pointer. |
| NULL | Will remove and unregister the hook. |

### 3.11.4 Return Value

Function pointer to the previous error hook if error hook was registered.

0 if no error hook was registered.

### 3.11.5 Example

```
sc_errHook_t error_hook;

sc_miscErrorHookRegister(error_hook);
```

### 3.11.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_SYSTEM_FATAL | |
| Process ID of caller is not valid (SC_ILLEGAL_PID) | e0 = pid |

## 3.12 sc_moduleCreate2

### 3.12.1 Description

This system call is used to request the kernel daemon to create a module. The standard kernel daemon (**sc_kerneld**) needs to be defined and started at system configuration.

When creating a module the maximum number of pools and processes must be defined. There is a maximum number of 128 modules per SCIOPTA system possible.

Each module has also a priority which can range between 0 (highest) to 31 (lowest) priority. This module priority defines a maximum priority level for all processes contained inside that module. The kernel will generate an error, if a process is created which has a higher priority than the module priority.

Each module contains an init process with process priority = 0 which will be created automatically.

The start address of dynamically created modules must reside in RAM.

### 3.12.2 Syntax

```
sc_moduleid_t sc_moduleCreate (
   sc_mdb_t           *mdb
);
```

### 3.12.3 Parameter

| | |
|---|---|
| **mdb** | Pointer to the module descriptor block (mdb) which defines the module to create. |
| | See chapter 3.12.5 “Module Descriptor Block mdb” on page 3-13. |

### 3.12.4 Return Value

Module ID.

### 3.12.5 Module Descriptor Block mdb

The module descriptor block is a structure which is defining a module to be created.

It is included in the header file modules.h.

```
struct sc_mdb_s {
  char name[SC_MODULE_NAME_SIZE+1];
  sc_module_addr_t *maddr;
  sc_prio_t maxPrio;
  unsigned int maxPools;
  unsigned int maxProcs;
  void (*init)(void);
  sc_bufsize_t stacksize;
  __u8 safetyFlag;
  __u8 spare_b;
  __u16 spare_h;
  __u32 *pt;
};
typedef struct sc_mdb_s sc_mdb_t;
```

### 3.12.6 Structure Members

| | |
|---|---|
| **name** | Name of the module to create. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore. |
| **maddr** | Pointer to a structure containing the module addresses and size. |
| | See chapter 3.12.7 “Module Address and Size” on page 3-14. |
| **maxPrio** | Maximum module priority. |
| | The priority of the module which can be from 0 to 31. 0 is the highest priority. |
| **maxPools** | Maximum number of pools in the module. |
| | The kernel will not allow to create more pools inside the module than stated here. Maximum value is 128. |
| **maxProcs** | Maximum number of processes in the module. |
| | The kernel will not allow to create more pools inside the module than stated here. Maximum value is 16384. |
| **init** | Function pointer to the init process function. |
| | This is the address where the init process of the module will start execution. |
| **stacksize** | Stack size of the module init process. |

| | |
|---|---|
| **safetyFlag** | Module safety flag. |
| FALSE | None-safety module. |
| TRUE | Safety module. |
| **spare_b** | Spare, write 0. |
| **spare_h** | Spare, write 0. |
| **pt** | Pointer to the page table for MMU. |
| <pageptr> | Pointer to the MMU page table |
| NULL | If no MMU in the system. |

### 3.12.7 Module Address and Size

This is a structure which is defining the module addresses and the module size to be created. It is usually generated by the linker script.

It is defined in the header file modules.h.

```
typedef struct sc_module_addr_s {
  char *start;
  __u32 size;
  __u32 initsize;
} sc_module_addr_t;
```

### 3.12.8 Structure Members

| | |
|---|---|
| **start** | Start address of the module in RAM. |
| **size** | Size of the module in bytes (RAM). |
| | The minimum module size can be calculated according to the following formula (bytes):<br>size_mod = p * 256 + stack + pools + mcb + initsize<br>p = Number of static processes.<br>stack = Sum of stack sizes of all static processes.<br>pools = Sum of sizes of all message pools.<br>mcb = Module control block (see below)<br>initsize = Code (see parameter initsize) below<br>where: mcb = 200 |
| **initsize** | Size of the initialized data. |

### 3.12.9 Example

```
extern sc_module_addr_t M2_mod;

static const sc_mdb_t mdb = {
   /* module-name      */ "M2",
   /* module addresses */ &M2_mod, /* => linker-script */
   /* max. priority    */ 0,
   /* max. pools       */ 2,
   /* max. procs       */ 3,
   /* init-function    */ M2_init,/* init-stacksize */512,
   /* safety-flag      */ SC_KRN_FLAG_TRUE,
   /* spare values     */ 0,0,0
};

(void) sc_moduleCreate2 (&mdb,0);
```

### 3.12.10 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter **mdb** not valid. 0 or SC_NIL. | |
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EMODULE_TOO_SMALL ¦ SC_ERR_SYSTEM_FATAL<br>Process control blocks and pool control blocks<br>do not fit in module size. | e0 = Module size |
| KERNEL_EILL_NAME ¦ SC_ERR_SYSTEM_FATAL<br>Requested name does not comply with SCIOPTA naming rules<br>or does already exist. | |
| KERNEL_ENO_MORE_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Maximum number of modules reached. | e0 = Number of modules |

| Error Code ¦ Error Type | Extra Value |
|---|---|

KERNEL_EILL_PARAMETER ¦ SC_ERR_SYSTEM_FATAL

Parameter of module descriptor block not valid.
- maddr 0, SC_NIL or unaligned.
- maxPrio > 31
- maxPools > 128
- maxProcs > (MAX_PID+1)
- init = 0
- init not 4-byte alligned
- stacksize < SC_MIN_STACKSIZE
- safetyFlag neither true nor false
- pt not valid
- spares not 0

KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL

Module addresses or sizes not valid.
Start address, size or initsize unaligned.
initsize > size.

KERNEL_EMODULE_OVERLAP ¦ SC_ERR_SYSTEM_FATAL

Modules do overlap.

e0 = Requested start address
e1 = Module start address

## 3.13 sc_moduleIdGet

### 3.13.1 Description

This system call is used to get the ID of a module

In contrast to the call sc_procIdGet, you can just give the name as parameter and not a path.

### 3.13.2 Syntax

```
sc_moduleid_t sc_moduleIdGet (
   const char         *name
);
```

### 3.13.3 Parameter

| **name** | Module name. |
|---|---|
| <name> | Pointer to the 0 terminated name string. |
| NULL | Current module. |

### 3.13.4 Return Value

Module ID if the module name was found.

Current module ID (module ID of the caller) if parameter **name** is NULL.

SC_ILLEGAL_MID if the module name was not found.

### 3.13.5 Example

```
sc_moduleid_t mid;

mid = sc_moduleIdGet ("user_01");
sc_moduleStop (mid);
```

#### 3.13.5.1 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE_NAME ¦ SC_ERR_PROCESS_WARNING<br>String pointed to by name too long. | e0 = name |

## 3.14 sc_moduleInfo

### 3.14.1 Description

This system call is used to get a snap-shot of a module control block (mcb).

SCIOPTA maintains a module control block (mcb) per module and a process control block (pcb) per process which contains information about the module and process. System level debugger or run-time debug code can use this system call to get a copy of the control blocks.

The caller supplies a module control block structure in a local variable. The kernel will fill the structure with the module control block data.

The structure content will reflect the module control block data at a certain moment which will be determined by the kernel. It is therefore a data snap-shot of which the exact time cannot be retrieved. You cannot directly access the module control blocks.

The structure of the module control block is defined in the module.h include file.

### 3.14.2 Syntax

```
int sc_moduleInfo (
   sc_moduleid_t      mid,
   sc_moduleInfo_t    *info
);
```

### 3.14.3 Parameter

| | |
|---|---|
| **mid** | Module ID. |
| <mid> | mid. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |
| **info** | Pointer to a local structure of a module control block. |
| | See chapter 3.14.5 “Module Info Structure” on page 3-19. |

### 3.14.4 Return Value

1 if the module was found. In this case the **info** structure is filled with valid data.

0 if the module was not found.

### 3.14.5 Module Info Structure

The module info is a structure containing a snap-shot of the module control block.

It is included in the header file modules.h.

```
typedef struct sc_moduleInfo_s {
  sc_moduleid_t mid;
  sc_pid_t ppid;
  char name[SC_MODULE_NAME_SIZE+1];
  char *text;
  sc_modulesize_t textsize;
  char *data;
  sc_modulesize_t datasize;
  sc_modulesize_t freesize;
  unsigned int max_process;
  unsigned int nprocess;
  unsigned int max_pools;
  unsigned int npools;
} sc_moduleInfo_t;
```

### 3.14.6 Structure Members

| | |
|---|---|
| **mid** | Module ID. |
| **ppid** | ID of the parent process. Process from where the module was created. |
| **name** | Name of the module. |
| **text** | Current address into module text segment. |
| **textsize** | Size of module text segment. |
| **data** | Current address into module data segment. |
| **datasize** | Size of module data segment. |
| **freesize** | Free size of module. |

| | |
|---|---|
| **max_process** | Maximum defined number of processes in the module. |
| **nprocess** | Actual number of processes. |
| **max_pools** | Maximum defined number of pools in the module. |
| **npools** | Actual number of pools. |

### 3.14.7 Example

```
sc_moduleid_t mid;
sc_moduleInfo_t usr_info;
int check;

mid = sc_moduleIdGet ("user_01");
check = sc_moduleInfo (mid, &usr_info);
```

### 3.14.8 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter **info** not valid (info == 0). | |
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |

## 3.15 sc_moduleKill

### 3.15.1 Description

This system call is used to dynamically kill a whole module.

The standard kernel daemon (**sc_kerneld**) needs to be defined and started at system configuration.

All processes and pools in the module will be killed and removed. The system call will return when the whole kill process is done. The system module cannot be killed.

### 3.15.2 Syntax

```
void sc_moduleKill (
   sc_moduleid_t      mid,
   flags_t            flags
);
```

### 3.15.3 Parameter

| **mid** | Module ID. |
|---|---|
| <mid> | Module ID of the module to be killed and removed. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

| **flags** | Module kill flags. |
|---|---|
| 0 | A cleaning up will be executed. |
| SC_MODULEKILL_KILL | No cleaning up will be done. |

### 3.15.4 Return Value

None.

### 3.15.5 Example

```
sc_moduleid_t mid;
sc_moduleInfo_t usr_info;
int check;

mid = sc_moduleIdGet ("user_01");
sc_moduleKill (mid, 0);
```

### 3.15.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Module to be killed is the system module.<br>MID is not valid (mid >= SC_MAX_MODULE).<br>MID is not valid ( mcb ==SC_NIL). | e0 = mid. |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_SYSTEM_FATAL<br>Caller is not a prioritized process. | e0 = pcb. |

## 3.16 sc_moduleNameGet

### 3.16.1 Description

This system call is used to get the name of a module.

The name will be returned as a 0 terminated string.

### 3.16.2 Syntax

```
const char *sc_moduleNameGet (
   sc_moduleid_t        mid
);
```

### 3.16.3 Parameter

| **mid** | Module ID. |
|---|---|
| <mid> | Module ID. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

### 3.16.4 Return Value

Name string of the module if the module was found.

0 if the module was not found.

### 3.16.5 Example

```
sc_moduleid_t mid;

mid = sc_moduleIdGet ("user_01");
printf ("Module :%s\n", sc_moduleNameGet (mid));
```

### 3.16.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |

## 3.17 sc_modulePrioGet

### 3.17.1 Description

This system call is used to get the priority of a module.

### 3.17.2 Syntax

```
sc_prio_t sc_modulePrioGet (
   sc_moduleid_t        mid
);
```

### 3.17.3 Parameter

| **mid** | Module ID. |
|---|---|
| <mid> | Module ID. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

### 3.17.4 Return Value

Module priority if the module was found and was valid.

SC_ILLEGAL_PRIO if the module was not valid (mcb == SC_NIL).

### 3.17.5 Example

```
sc_moduleid_t mid;

mid = sc_moduleIdGet ("user_01");
printf ("Module Priority :%u\n", sc_modulePrioGet (mid));
```

### 3.17.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |

## 3.18 sc_moduleStop

### 3.18.1 Description

This system call is used to stop a module.

It will stop all processes in a module.

The process stop will be done in the order of their process ID. First all interrupt and timer processes will be stopped and then all prioritized processes are stopped.

The stop behaves identically as the sc_procStop system call for the respective process types.

### 3.18.2 Syntax

```
void sc_moduleStop (
   sc_moduleid_t        mid
);
```

### 3.18.3 Parameter

| **mid** | ID of module to be stopped. |
|---|---|
| <mid> | Module ID. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

### 3.18.4 Return Value

None.

### 3.18.5 Example

```
sc_moduleid_t mid;

mid = sc_moduleIdGet ("user_01");
sc_moduleStop (mid);
```

### 3.18.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Stopcounter overrun. | e0 = pcb |

## 3.19 sc_msgAcquire

### 3.19.1 Description

This system call is used to change the owner of a message. The caller becomes the owner of the message.

The kernel will copy the message into a new message buffer allocated from the default pool if the message resides not in a pool of the callers module and the callers module is not friend to the module where the message resides. In this case the message pointer (**msgptr**) will be modified.

Please use sc_msgAcquire with care. Transferring message buffers without proper ownership control by using sc_msgAcquire instead of transmitting and receiving messages with sc_msgTx and sc_msgRx will cause problems if you are killing processes.

### 3.19.2 Syntax

```
void sc_msgAcquire (
   sc_msgptr_t       msgptr
);
```

### 3.19.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to the message buffer pointer. |

### 3.19.4 Return Value

None.

### 3.19.5 Example

```
/* Change owner of a message */

sc_msg_t msg;
sc_msg_t msg2;

msg = sc_msgRx(SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

msg2 = msg->transport.msg; /* receive msg indirect */
sc_msgAcquire(&msg2);      /* become owner of the message */
```

### 3.19.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EMSG_HD_CORRUPT ¦ SC_ERR_MODULE_FATAL | |
| Message header is corrupt. Kernel is the message owner. | e0 = Pointer to message. |
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL | |
| Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL | |
| Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL | |
| Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.20 sc_msgAddrGet

### 3.20.1 Description

This system call is used to get the process ID of the addressee of a message.

The kernel will examine the message buffer to determine the process to which the message was originally transmitted.

This system call is mainly used in communication software of distributed multi-CPU systems (using connector processes). It allows to store the original addressee when you are forwarding a message by using the sc_msgTxAlias system call.

### 3.20.2 Syntax

```
sc_pid_t sc_msgAddrGet (
   sc_msgptr_t          msgptr
);
```

### 3.20.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 3.20.4 Return Value

Process ID of the addressee of the message.

### 3.20.5 Example

```
/* Get original addressee of a message */

sc_msg_t msg;
sc_pid_t addr;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);
addr = sc_msgAddrGet(&msg);
```

### 3.20.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.21 sc_msgAlloc

### 3.21.1 Description

This system call will allocate a memory buffer of selectable size from a message pool.

SCIOPTA supports ownership of messages. The new allocated buffer is owned by the caller process. The owner of the message will change to the receiver process if the message is sent to another process. If you need to define a new owner without sending the message you could use the sc_msgAcquire system call. The call sc_msgAcquire must be used very carefully as this will pass messages around in a disorderly manner.

SCIOPTA is not returning the exact amount of bytes requested but will select one of a list of fixed buffer sizes which is large enough to contain the requested number of bytes. This list can contain 4, 8 or 16 fixed sizes which will be defined when a message pool is created. The content of the allocated message buffer is not initialized and can have any random value.

As SCIOPTA supports multiple pools the caller has to state the pool ID (**plid**) from where to allocate the message. The pool can only be in the same module as the caller process.

The caller can define how the system will respond to some limiting system states such as memory shortage in message pools and reply delays due to dynamic system behaviour (**tmo**).

### 3.21.2 Syntax

```
sc_msg_t sc_msgAlloc (
   sc_bufsize_t         size,
   sc_msgid_t           id,
   sc_poolid_t          plid,
   sc_ticks_t           tmo
);
```

### 3.21.3 Parameter

| | |
|---|---|
| **size** | The requested size of the message buffer. |
| **id** | Message ID. |
| | The message ID which will be placed at the beginning of the data buffer of the message. |
| **plid** | Pool ID from where the message will be allocated. |
| <pool id> | Pool ID from where the message will be allocated. |
| SC_DEFAULT_POOL | Message will be allocated from the default pool. The default pool can be set by the system call sc_poolDefault. |

| **tmo** | Allocation timing parameter. |
|---|---|
| SC_ENDLESS_TMO | Timeout is not used. Blocks and waits endless until a buffer is available from the message pool. |
| SC_NO_TMO | A NIL pointer will be returned if there is memory shortage in the message pool. |
| SC_FATAL_IF_TMO | A (fatal) kernel error will be generated if a message buffer of the requested size is not available. |
| 0 < tmo =< SC_TMO_MAX | Timeout value in system ticks. Alloc with timeout. Blocks and waits the specified number of ticks to get a message buffer. |

### 3.21.4 Return Value

| | |
|---|---|
| Pointer to the allocated buffer if: | **tmo** = SC_ENDLESS_TMO or |
| | **tmo** = SC_NO_TMO and if a buffer of the requested size is available or |
| | **tmo** > 0 and the system responds within the timeout period. |
| NIL pointer if: | **tmo** = SC_NO_TMO and if a buffer of the requested size is not available or |
| | **tmo** = > 0 and the system does not respond within the timeout period. |

### 3.21.5 Example

```
/* Allocate TEST_MSG from default pool */

sc_msg_t msg;

msg = sc_msgAlloc(sizeof(test_msg_t), /* size */
                  TEST_MSG,           /* message id */
                  SC_DEFAULT_POOL,    /* pool index */
                  SC_FATAL_IF_TMO);   /* timeout */
```

### 3.21.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_POOL_ID ¦ SC_ERR_PROCESS_FATAL<br>Pool index is not available. | e0 = Pool index |
| KERNEL_EILL_BUFSIZE ¦ SC_ERR_PROCESS_FATAL<br>Illegal message size was requested. | e0 = Requested size<br>e1 = Pool CB (or -1) |
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL<br>Request for number of bytes could not be fulfilled. | e0 = size<br>e1 = Pool CB |
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>tmo value not valid. | e0 = tmo value |
| KERNEL_EILL_DEFPOOL_ID ¦ SC_ERR_PROCESS_WARNING<br>Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook. | e0 = Pool index |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Process would swap but interrupts and/or scheduler are/is locked. | e0 = Lock counter value or<br>-1 if interrupt are locked. |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>tmo-flag with wrong value. Likely system is corrupt. | e0 = tmo-flag |

## 3.22 sc_msgAllocClr

### 3.22.1 Description

This system call works exactly the same as sc_msgAlloc but it will initialize the data area of the message to 0.

### 3.22.2 Syntax

```
sc_msg_t sc_msgAllocClr (
   sc_bufsize_t         size,
   sc_msgid_t           id,
   sc_poolid_t          plidx,
   sc_ticks_t           tmo
);
```

### 3.22.3 Parameter

Parameter values are the same as in chapter 3.21 “sc_msgAlloc” on page 3-30.

### 3.22.4 Return Value

Return values are the same as in chapter 3.21 “sc_msgAlloc” on page 3-30.

### 3.22.5 Example

```
/* Allocate TEST_MSG from default pool and clear its content*/

sc_msg_t msg;

msg = sc_msgAllocClr(sizeof(test_msg_t), /* size */
                     TEST_MSG,           /* message id */
                     SC_DEFAULT_POOL,    /* pool index */
                     SC_FATAL_IF_TMO);   /* timeout */
```

### 3.22.6 Error

same as in chapter 3.21.6 “Errors” on page 3-32.

## 3.23 sc_msgFind

### 3.23.1 Description

This system call is used to find messages which have been allocated or already received. The allocated-messages queue of the caller will be searched for the desired messages. This call is similar to **sc_msgRx** (see chapter 3.29 “sc_msgRx” on page 3-44) but instead of searching the messages-input queue the allocated-messages queue will be scanned. This queue holds the messages which have already been received (by **sc_msgRx**) or messages which have been allocated by the caller process.

If a message matching the conditions is found the kernel will return to the caller. If the allocated-messages queue is empty or no wanted messages are available in the queue a NULL will be returned. The call sc_msgFind will not block the system.

A pointer to an array (**wanted**) containing the messages (and/or process IDs) which will be scanned by sc_msgFind must be given. The array must be terminated by 0. The kernel stores the pointer to the array in the process control block for debugging.

A parameter flag (**flag**) controls different searching methods:

1. The messages to be searched are listed in a message ID array.
2. The array can also contain process IDs. In this case all messages sent by the listed processes are searched.
3. You can also build an array of message ID and process ID pairs to find specific messages sent from specific processes.
4. Also a message array with reversed logic can be given. In this case any message is searched except the messages specified in the array.

If the pointer **wanted** to the array is **NULL** or the array is empty (contains only a zero element) all messages will be searched.

### 3.23.2 Syntax

```
sc_msg_t sc_msgFind (
   sc_msgptr_t        mp,
   void               *wanted,
   flags_t            flag
);
```

### 3.23.3 Parameter

| | |
|---|---|
| **mp** | Pointer to a message in the allocated-messages queue. |
| | Pointer to the message in the queue from where the find scanning will start. |
| NULL | Scanning starts from the head of the allocated-messages queue. |

| **wanted** | Pointer to the message (or pid) array. |
|---|---|
| <ptr> | Pointer to the message (or process ID) array. |
| SC_MSGRX_ALL | All messages will be searched. |

| **flag** | Find flag. |
|---|---|
| | More than one value can be defined and must be separated by OR instructions. |
| SC_MSGRX_MSGID | An array of wanted message IDs is given. |
| SC_MSGRX_PID | An array of process ID´s from where sent messages are received is given. |
| SC_MSGRX_NOT | An array of message ID's is given which will be excluded from the search. |
| SC_MSGRX_BOTH | An array of pairs of message ID´s and process ID´s are given to search for specific messages from specific transmitting processes. |

### 3.23.4 Return Value

Pointer to the received message if the message has been received. The caller becomes owner of the received message.

NULL if no messages are in the allocated-messages queue or no messages can be found which match the wanted flag conditions.

### 3.23.5 Examples

TBD

### 3.23.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL | |
| Illegal flags. | e0 = flags |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL | |
| tmo-flag with wrong value. Likely system is corrupt. | e0 = tmo-flag |

## 3.24 sc_msgFree

### 3.24.1 Description

This system call is used to return a message to the message pool if the message is no longer needed. Message buffers which have been returned can be used by other processes.

Only the owner of a message is allowed to free it by calling sc_msgFree. It is a fatal error to free a message owned by another process. If you have, for example transmitted a message to another process it is the responsibility of the receiving process to free the message.

Another process actually waiting to allocate a message of a full pool will become ready and therefore the caller process pre-empted on condition that:

1. the returned message buffer of the caller process has the same fixed size as the one of the waiting process and
2. the priority of the waiting process is higher than the priority of the caller and
3. the waiting process waits on the same pool as the caller will return the message.

### 3.24.2 Syntax

```
void sc_msgFree (
   sc_msgptr_t       msgptr
);
```

### 3.24.3 Parameter

**msgptr** Pointer to message pointer.

### 3.24.4 Return Value

None.

### 3.24.5 Example

```
/* Free a message */

sc_msg_t msg;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);
sc_msgFree(&msg);
```

### 3.24.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Message is a timout message. | |

## 3.25 sc_msgHdCheck

### 3.25.1 Description

It is used to check the message header. The header will be checked for plausibility.

Checks include message ownership, valid module, message endmarks, size and others.

### 3.25.2 Syntax

```
int sc_msgHdCheck (
   sc_msgptr_t       msgptr,
);
```

### 3.25.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 3.25.4 Return Value

!= 0 if the message header is ok.

== 0 if the message header is corrupted.

### 3.25.5 Example

```
sc_msg_t msg:
msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

if (sc_msgHdCheck(&msg)) {
   printf ("Message ok!"
} else {
   printf ("Message corrupted!"
};
```

### 3.25.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message pointer or pointer to message are zero. | |

## 3.26 sc_msgHookRegister

### 3.26.1 Description

This system call will register a global or module message hook.

There can be one module message hook of each type (transmit/receive) per module.

If sc_msgHookRegister is called from within a module a module message hook will be registered.

A global message hook will be registered when sc_msgHookRegister is called from the start hook function which is called before SCIOPTA is initialized.

Each time a message is sent or received (depending on the setting of parameter **type**) the module message hook of the caller will be called if such a hook exists. First the module and then the global message hook will be called.

### 3.26.2 Syntax

```
sc_msgHook_t *sc_msgHookRegister (
   int             type,
   sc_msgHook_t    *newhook
);
```

### 3.26.3 Parameter

| **type** | Defines the type of registerred CONNECTOR. |
|---|---|
| SC_SET_MSGTX_HOOK | Registers a message transmit hook. Every time a message is sent, this hook will be called. |
| SC_SET_MSGRX_HOOK | Registers a message receive hook. Every time a message is received, this hook will be called. |

| **newhook** | Message hook function pointer. |
|---|---|
| <funcptr> | Function pointer to the message hook. |
| NULL | Removes and unregisters the message hook. |

### 3.26.4 Return Value

Function pointer to the previous message hook. if the message hook was registered.

0 if no message hook was registered.

### 3.26.5 Example

```
sc_msgHook_t oldMsgHook;

oldMsgHook = sc_msgHookRegister(SC_SET_MSGRX_HOOK,rxHook);
oldMsgHook = sc_msgHookRegister(SC_SET_MSGTX_HOOK,txHook);
```

### 3.26.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Wrong type (unknown or not active) | e0 = Requested message hook type. |

## 3.27 sc_msgOwnerGet

### 3.27.1 Description

This system call is used to get the process ID of the owner of a message.

The kernel will examine the message buffer to determine the process who owns the message buffer.

### 3.27.2 Syntax

```
sc_pid_t sc_msgOwnerGet (
   sc_msgptr_t          msgptr
);
```

### 3.27.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 3.27.4 Return Value

Process ID of the owner of the message.

### 3.27.5 Example

```
/* Get owner of received message (will be caller) */

sc_msg_t msg;
sc_pid_t owner;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);
owner = sc_msgOwnerGet(&msg);
```

### 3.27.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.28 sc_msgPoolIdGet

### 3.28.1 Description

This system call is used to get the pool ID of a message.

When you are allocating a message with sc_msgAlloc you need to give the ID of a pool from where the message will be allocated. During run-time you sometimes need to this information from received messages.

### 3.28.2 Systax

```
sc_poolid_t sc_msgPoolIdGet (
   sc_msgptr_t       msgptr);
```

### 3.28.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 3.28.4 Return Value

Pool ID where the message resides if the message is in the same module than the caller.

SC_DEFAULT_POOL if the message is not in the same module than the caller.

### 3.28.5 Example

```
/* Retrieve the pool-index of a message */

sc_msg_t msg:
sc_poolid_t idx;
msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

idx = sc_msgPoolIdGet(&msg);
```

### 3.28.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.29 sc_msgRx

### 3.29.1 Description

This system call is used to receive messages. The receive message queue of the caller will be searched for the desired messages.

If a message matching the conditions is received the kernel will return to the caller. If the message queue is empty or no wanted messages are available in the queue the process will be swapped out and another ready process with the highest priority will run. If a desired message arrives the process will be swapped in and the **wanted** list will be scanned again.

A pointer to an array (**wanted**) containing the messages (and/or process IDs) which will be scanned by **sc_msgRx**. The array must be terminated by 0. The kernel stores the pointer to the array in the process control block for debugging.

A parameter flag (**flag**) controls different receiving methods:

1. The messages to be received are listed in a message ID array.
2. The array can also contain process IDs. In this case all messages sent by the listed processes are received.
3. You can also build an array of message ID and process ID pairs to receive specific messages sent from specific processes.
4. Also a message array with reversed logic can be given. In this case any message is received except the messages specified in the array.

If the pointer **wanted** to the array is **NULL** or the array is empty (contains only a zero element) all messages will be received.

The caller can also specify a timeout value **tmo**. The caller will not wait (swapped out) longer than the specified time. If the timeout expires the process will be made ready again and sc_msgRx will return with **NULL**.

The activation time is saved for **sc_msgRx** in prioritized processes. The activation time is the absolute time (tick counter) value when the process became ready.

### 3.29.2 Syntax

```
sc_msg_t sc_msgRx (
   sc_ticks_t        tmo,
   void              *wanted,
   int               flag
);
```

### 3.29.3 Parameter

| **tmo** | Timeout parameter. |
|---|---|
| SC_ENDLESS_TMO | Blocks and waits endless until the message is received. |
| SC_NO_TMO | No timeout, returns immediately. Must be set for interrupt and timer processes. |
| 0 < tmo =< SC_TMO_MAX | Timeout value in system ticks. Receive with timeout. Blocks and waits a specified maximum number of ticks to receive the message. If the timeout expires the process will be made ready again and sc_msgRx will return with NULL. |

| **wanted** | Pointer to the message (or pid) array. |
|---|---|
| <ptr> | Pointer to the message (or process ID) array. |
| SC_MSGRX_ALL | All messages will be received. |

| **flag** | Receive flag. |
|---|---|
| | More than one value can be defined and must be separated by OR instructions. |
| SC_MSGRX_MSGID | An array of wanted message IDs is given. |
| SC_MSGRX_PID | An array of process ID´s from where sent messages are received is given. |
| SC_MSGRX_NOT | An array of message ID's is given which will be excluded from receive. |
| SC_MSGRX_BOTH | An array of pairs of message ID´s and process ID´s are given to receive specific messages from specific transmitting processes. |

### 3.29.4 Return Value

Pointer to the received message if the message has been received. The caller becomes owner of the received message.

NULL if timeout expired. The process will be made ready again.

### 3.29.5 Examples

```
/* wait max. 1000 ticks for TEST_MSG */

sc_msg_t msg;
sc_msgid_t sel[2] = { TEST_MSG, 0 };
msg = sc_msgRx( 1000,            /* timeout in ticks */
                sel,             /* selection array, here message IDs */
                SC_MSGRX_MSGID); /* type of selection */


/* wait endless for a message from processes other than sndr_pid */

sc_msg_t msg;
sc_pid_t sel[2];
sel[0] = sndr_pid;
sel[1] = 0;
msg = sc_msgRx( SC_ENDLESS_TMO,        /* timeout in ticks, here endless*/
                sel,                   /* selection array, here process IDs */
                SC_MSGRX_PID|SC_MSGRX_NOT); /* type of selection, inverted */
```

```
/* wait for message from a certain process */

sc_msg_t msg;
sc_msgrx_t sel[3];
sel[0].msgid = TEST_MSG;
sel[0].pid = testerA_pid;
sel[1].msgid = TEST_MSG;
sel[1].pid = testerB_pid;
sel[2].msgid = 0;
sel[2].pid = 0;
msg = sc_msgRx( SC_ENDLESS_TMO,      /* timeout in ticks, here endless */
                sel,                 /* selection array, here process IDs */
                SC_MSGRX_PID|SC_MSGRX_MSGID); /* type of selection */

/* Wait for any message */

sc_msg_t msg:
msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);
```

### 3.29.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>tmo value not valid. | e0 = tmo value |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Process would swap but interrupts and/or scheduler are/is locked. | e0 = Lock counter value or<br>-1 if interrupt are locked. |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal flags. | e0 = flags |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>tmo-flag with wrong value. Likely system is corrupt. | e0 = tmo-flag |

## 3.30 sc_msgSizeGet

### 3.30.1 Description

This system call is used to get the requested size of a message. The requested size is the size of the message buffer when it was allocated. The actual kernel internal used fixed size might be larger.

### 3.30.2 Syntax

```
sc_bufsize_t sc_msgSizeGet (
   sc_msgptr_t       msgptr);
```

### 3.30.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 3.30.4 Return Value

Requested size of the message.

### 3.30.5 Example

```
/* Get the size of a message */

sc_msg_t msg:
sc_bufsize_t size;
msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

size = sc_msgSizeGet(&msg);
```

### 3.30.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.31 sc_msgSizeSet

### 3.31.1 Description

This system call is used to decrease the requested size of a message buffer.

The originally requested message buffer size is smaller (or equal) than the SCIOPTA internal used fixed buffer size. If the need of message data decreases with time it is sometimes favourable to decrease the requested message buffer size as well. Some internal operation are working on the requested buffer size.

The fixed buffer size for the message will not be modified. The system does not support increasing the buffer size.

### 3.31.2 Syntax

```
sc_bufsize_t sc_msgSizeSet (
   sc_msgptr_t       msgptr,
   sc_bufsize_t      newsz
);
```

### 3.31.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |
| **newsz** | New requested size of the message buffer. |

### 3.31.4 Return Value

New requested buffer size if call without error condition.

Old requested buffer size if it was a wrong request such as requesting a higher buffer size as the old one.

### 3.31.5 Example

```
/* Change size of a message */

sc_msg_t msg:

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

/* ... do something ... */

sc_msgSizeSet(&msg, sizeof(reply_msg_t)); /* reduce size before returning */
sc_msgTx(&msg, sc_msgSndGet(&msg), 0);    /* return to sender (ACK) */
```

### 3.31.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_BUFSIZE ¦ SC_ERR_MODULE_FATAL<br>Illegal buffer sizes. | e0 = Buffer sizes |
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_EENLARGE_MSG ¦ SC_ERR_MODULE_FATAL<br>Message would be enlarged. | e0 = Buffer size<br>e1 = Pointer to message |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.32 sc_msgSndGet

### 3.32.1 Description

This system call is used to get the process ID of the sender of a message.

The kernel will examine the message buffer to determine the process who has transmitted the message buffer.

### 3.32.2 Syntax

```
sc_pid_t sc_msgSndGet (
   sc_msgptr_t          msgptr
);
```

### 3.32.3 Parameter

**msgptr** Pointer to message pointer.

### 3.32.4 Return Value

Process ID of the sender of the message if the message was sent at least once.

Process ID of the owner of the message if the message was never sent.

### 3.32.5 Example

```
/* Get the sender of a message */

sc_msg_t msg;
sc_pid_t sndr;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);
sndr = sc_msgSndGet(&msg);
```

### 3.32.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 3.33 sc_msgTx

### 3.33.1 Description

This system call is used to transmit a SCIOPTA message to a process (the addressee process).

Each SCIOPTA process has one message queue for messages which have been sent to the process. The sc_msgTx system call will enter the message at the end of the receivers message queue.

The caller cannot access the message buffer any longer as it is not any more the owner. The receiving process will become the owner of the message. **NULL** is loaded into the caller´s message pointer **msgptr** to avoid unintentional message access by the caller after transmitting.

The receiving process will be swapped-in if it has a higher priority than the sending process.

If the addressee of the message resides not in the callers module and this module is not registered as a friend module the message will be copied before the transmit call will be executed. Messages which are transmitted across modules boundaries are always copied except if the modules are “friends”. To copy such a message the kernel will allocate a buffer from the pool of the module where the receiving process resides big enough to fit the message and copy the whole message. Message buffer copying depends on the friendship settings of the module where the buffer was originally allocated.

If the receiving process is not within the same target (CPU) as the caller the message will be sent to the connector process where the (distributed) receiving process is registered.

### 3.33.2 Syntax

```
void sc_msgTx (
   sc_msgptr_t     msgptr,
   sc_pid_t        addr,
   flags_t         flags
);
```

### 3.33.3 Parameter

| **msgptr** | Pointer to message pointer. |
|---|---|

| **addr** | The process ID of the addressee. |
|---|---|
| <pid> | Valid SCIOPTA PID |
| SC_CURRENT_PID | The caller himself. |

| **flags** | Transmitt flags. |
|---|---|
| 0 | Normal sending. |
| SC_MSGTX_RTN2SNDR | Return message if addressee does not exist or if there is no memory to copy it into the addressee's module. |

### 3.33.4 Return Value

None.

### 3.33.5 Example

```
/* Send TEST_MSG to "addr" */

sc_msg_t msg;
sc_pid_t addr;

/* ... */
msg = sc_msgAlloc(sizeof(test_msg_t),TEST_MSG, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
sc_msgTx( &msg, sndr, 0 );
```

### 3.33.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Process type not valid. | e0 = pid of addressee<br>e1 = Process type |

KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL

Addressee pid not valid (is init0. bigger than MODULE_MAXPROCESS). Illegal CONNECTOR pid. Illegal addressees module-index.

e0 = Addressee process ID

KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL

Flag is neither 0 nor SC_MSGTX_RTN2SND

e0 = Flag value
e1 = 2 (position)

KERNEL_EALREADY_DEFINED ¦ SC_ERR_PROCESS_FATAL

Caller tries to send a timeout message but message is already a timeout message.

e0 = Pointer to message header

## 3.34 sc_msgTxAlias

### 3.34.1 Description

This system call is used to transmit a SCIOPTA message to a process by setting a process ID as sender.

The usual sc_msgTx system call sets always the calling process as sender. If you need to set another process ID as sender you can use this sc_msgTxAlias call.

This call is mainly used in communication software such as SCIOPTA connector processes where processes on other CPU´s are addressed. CONNECTOR processes will use this system call to enter the original sender of the other CPU.

Otherwise sc_msgTxAlias works the same way as sc_msgTx.

### 3.34.2 Syntax

```
void sc_msgTxAlias (
   sc_msgptr_t       msgptr,
   sc_pid_t          addr,
   flags_t           flags,
   sc_pid_t          alias
);
```

### 3.34.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |
| **addr** | The process ID of the addressee. |
| <pid> | Valid SCIOPTA PID |
| SC_CURRENT_PID | The caller himself. |
| **flags** | Sending flags. |
| 0 | Normal sending. |
| SC_MSGTX_RTN2SNDR | Return message if addressee does not exist or if there is no memory to copy it into the addressee's module. |
| **alias** | The process ID specified as sender. |

### 3.34.4 Return Value

None.

### 3.34.5 Example

```
/* Send TEST_MSG to process "addr" as process "other" */

sc_msg_t msg;
sc_pid_t addr;
sc_pid_t other;


/* ... */

msg = sc_msgAlloc(sizeof(test_msg_t),TEST_MSG, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
sc_msgTxAlias( &msg, sndr, 0, other );
```

### 3.34.6 Errors

Same errors as in chapter 3.33.6 “Errors” on page 3-54.

## 3.35 sc_poolCreate

### 3.35.1 Description

This system call is used to create a new message pool inside the callers module.

### 3.35.2 Syntax

```
sc_poolid_t sc_poolCreate (
   char            *start,
   sc_plsize_t     size,
   unsigned int    nbufs,
   sc_bufsize_t    *bufsize,
   const char      *name
);
```

### 3.35.3 Parameter

| | |
|---|---|
| **start** | Start address of the pool. |
| <start> | Start address |
| 0 | The kernel will automatically take the next free address in the module |
| **size** | Size of the message pool. |
| | The minimum size must be the size of the maximum number of messages which ever are allocated at the same time plus the pool control block (pool_cb).<br>The size of the pool control block (pool_cb) can be calculated<br>according to the following formula:<br>pool_cb = 68 + n * 20 + stat * n * 20<br>n Maximum buffer sizes defined for the whole system<br>(and not the buffer sizes of the created pool). Value n can be 4, 8 or 16.<br>statProcess statistics or message statistics are used (1) or not used (0). |
| **nbufs** | The number of fixed buffer sizes. |
| | This can be 4, 8 or 16. It must always be lower or equal of the fixed buffer sizes which is defined for the whole system. |
| **bufsizes** | Pointer to an array of the fixed buffer sizes in ascending order. |
| **name** | Pointer to the name of the pool to create. |
| | The name is represented by a ASCII character string terminated by 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore. |

### 3.35.4 Return Value

Pool ID of the created message pool.

### 3.35.5 Example

```
static const sc_bufsize_t bufsizes[8]=
{
   4,
   8,
   16,
   32,
   64,
   128,
   256,
   700
};

myPool_plid = sc_poolCreate(
   /* start-address     */ 0,
   /* total size        */ 4000,
   /* number of buffers */ 8,
   /* buffersizes       */ bufsizes,
   /* name              */ "myPool"
);
```

### 3.35.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL | |
| Illegal module.<br>module >= SC_MAX_MODULE<br>module == SC_NIL | e0 = Module ID |
| KERNEL_EILL_NAME ¦ SC_ERR_MODULE_FATAL | |
| Illegal pool name requested. | e0 = Requested pool name |
| KERNEL_ENO_MORE_POOL ¦ SC_ERR_MODULE_FATAL | |
| Maximum number of pools for module reached. | e0 = Number of pools in module cb. |

KERNEL_EILL_NUM_SIZES ¦ SC_ERR_MODULE_FATAL

Illegal number of buffer sizes. e0 = Number of requested buffer sizes.

KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL

No more memory in module for pool. e0 = Requested pool size.

KERNEL_EILL_POOL_SIZE ¦ SC_ERR_MODULE_FATAL

Size is not 4-byte aligned e0 = Requested pool size.
Even the biggest buffer does not fit.

KERNEL_EILL_PARAMETER ¦ SC_ERR_MODULE_FATAL

Pool start address is not 4-byte aligned. e0 = Requested pool start address.

## 3.36 sc_poolDefault

### 3.36.1 Description

This system call sets a message pool as default pool.

The default pool will be used by the **sc_msgAlloc** system call if the parameter for the pool to allocate the message from is defined as **SC_DEFAULT_POOL**.

Each process can set its default message pool by sc_poolDefault. The defined default message pool is stored inside the process control block. The initial default message pool at process creation is 0.

The default pool is also used if a message sent from another module needs to be copied.

### 3.36.2 Syntax

```
sc_poolid_t sc_poolDefault (
   int          idx
);
```

### 3.36.3 Parameter

| **idx** | Pool ID. |
|---|---|
| Zero or positive | Pool ID. |
| -1 | Request to return the ID of the default pool. |

### 3.36.4 Return Value

Pool ID of the default pool.

### 3.36.5 Example

```
pl = sc_poolIdGet("fs_pool");
if ( pl != SC_ILLEGAL_POOLID ){
   sc_poolDefault(pl);
}
```

### 3.36.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_POOL_ID ¦ SC_ERR_PROCESS_WARNING | |
| Pool index not valid.<br>Pool index > 16<br>Pool index > MODULE_MAXPOOLS | e0 = Requested pool index. |

## 3.37 sc_poolHookRegister

### 3.37.1 Description

This system call will register a pool create or pool kill hook.

There can be one pool create and one pool kill hook per module.

If sc_poolHookRegister is called from within a module a module pool hook will be registered.

A global pool hook will be registered when sc_poolHookRegister is called from the start hook function which is called before SCIOPTA is initialized.

Each time a pool is created or killed (depending on the setting of parameter **type**) the pool hook of the caller will be called if such a hook exists.

### 3.37.2 Syntax

```
sc_poolHook_t *sc_poolHookRegister (
   int             type,
   sc_poolHook_t   *newhook
);
```

### 3.37.3 Parameter

| **type** | Defines the type of registered pool hook. |
|---|---|
| SC_SET_POOLCREATE_HOOK | Registers a pool create hook. Every time a pool is created, this hook will be called. |
| SC_SET_POOLKILL_HOOK | Registers a pool kill hook. Every time a pool is killed, this hook will be called. |

| **newhook** | Pool hook function pointer. |
|---|---|
| <funcptr> | Function pointer to the hook |
| NULL | The pool hook will be removed and unregistered. |

### 3.37.4 Return Value

Function pointer to the previous pool hook if the pool hook was registered.

NULL if no pool hook was registered.

### 3.37.5 Example

```
sc_poolHook_t oldPoolHook;

oldPoolHook = sc_poolHookRegister(SC_SET_POOLCREATE_HOOK,plHook);
```

### 3.37.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Pool hook type not defined. | e0 = Pool hook type.<br>e1 = 0 |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Pool hook function pointer not valid. | e0 = Pool hook type.<br>e1 = 1 |

## 3.38 sc_poolIdGet

### 3.38.1 Description

This system call is used to get the ID of a message pool.

In contrast to the call sc_poolIdGet, you can just give the name as parameter and not a path.

### 3.38.2 Syntax

```
sc_poolid_t sc_poolIdGet (
   const char        *name
);
```

### 3.38.3 Parameter

| **name** | Pool name. |
|---|---|
| <name> | Pointer to the 0 terminated name string. |
| NULL | Default pool. |
| SC_NIL | Default pool. |
| Empty string | Default pool. |

### 3.38.4 Return Value

Pool ID if pool was found.

SC_ILLEGAL_POOLID if pool was not found.

### 3.38.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet("fs_pool");
if ( pl != SC_ILLEGAL_POOLID ){
   sc_poolDefault(pl);
}
```

### 3.38.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_NAME ¦ SC_ERR_PROCESS_FATAL<br>Illegal pool name. | e0 = pointer to pool name. |

## 3.39 sc_poolInfo

### 3.39.1 Description

This system call is used to get a snap-shot of a pool control block.

SCIOPTA maintains a pool control block per pool which contains information about the pool. System level debugger or run-time debug code can use this system call to get a copy of the control block.

The caller supplies a pool control block structure in a local variable. The kernel will fill the structure with the control block data.

The structure content will reflect the pool control block data at a certain moment which will be determined by the kernel. It is therefore a data snap-shot of which the exact time cannot be retrieved. You cannot directly access the pool control blocks.

The structure of the pool control block is defined in the pool.h include file.

### 3.39.2 Syntax

```
int sc_poolInfo (
   sc_moduleid_t     mid,
   sc_poolid_t       plid,
   sc_pool_cb_t      *info
);
```

### 3.39.3 Parameter

| | |
|---|---|
| **mid** | Module ID where the pool resides of which the control block will be returned. |
| **plid** | ID of the pool of which the pool control block data will be returned. |
| **info** | Pointer to a local structure of a pool control block.<br>This structure will be filled with the pool control block data. |

### 3.39.4 Return Value

!= 0 if the pool control block data was successfully retrieved.

== 0 if the pool control block could not be retrieved.

### 3.39.5 Pool Info Structure

The pool info is a structure containing a snap-shot of the pool control block.

It is included in the header file pool.h.

```
struct sc_pool_cb_s{
  sc_save_poolid_t poolid;

  sc_save_voidptr_t start;
  sc_save_voidptr_t end;
  sc_save_voidptr_t cur;
  sc_save_uint_t lock;

  sc_save_uint_t nbufsizes;
  sc_save_plsize_t size;
  sc_save_pid_t creator;

  sc_save_bufsize_t bufsizes[SC_MAX_NUM_BUFFERSIZES];
  idbl_t freed[SC_MAX_NUM_BUFFERSIZES];
  idbl_t waiter[SC_MAX_NUM_BUFFERSIZES];

  char name[SC_POOL_NAME_SIZE+1];
#if SC_MSG_STAT == 1
  sc_pool_stat_t stat;
#endif
} sc_pool_cb_t;
```

### 3.39.6 Structure Members

| | |
|---|---|
| **poolid** | Pool ID. |
| **start** | Start of pool-data area. |
| **end** | End of pool (first byte not in pool). |
| **cur** | First free byte inside pool. |
| **lock** | Lock setting. Not locked if 0. |
| **nbufsizes** | Number of buffer sizes. |
| **size** | Complete pool size. |

| | |
|---|---|
| **creator** | Process which created the pool. |
| **bufsizes** | Array of buffers. |
| **freed** | List of free'd buffers. |
| **waiter** | List of processes waiting for a buffer. |
| **name** | Pointer to pool name. |
| **stat** | Statistics information. See chapter 3.39.7 "Pool Statistics Info Structure" on page 3-67. |

### 3.39.7 Pool Statistics Info Structure

The pool statistics info is a structure inside the pool control block containing containing pool statistics information.

It is included in the header file pool.h.

```
struct sc_pool_stat_s {
  __u32 cnt_req[SC_MAX_NUM_BUFFERSIZES];     /* No. requests for a spec. size */
  __u32 cnt_alloc[SC_MAX_NUM_BUFFERSIZES];   /* No. allocation of a spec. size */
  __u32 cnt_free[SC_MAX_NUM_BUFFERSIZES];    /* No. releases of a spec. size */
  __u32 cnt_wait[SC_MAX_NUM_BUFFERSIZES];    /* No. unfullfilled allocations */
  sc_bufsize_t maxalloc[SC_MAX_NUM_BUFFERSIZES]; /* largest wanted size */
}sc_pool_stat_t;
```

### 3.39.8 Structure Members

| | |
|---|---|
| **cnt_req** | Number of buffer requests for a specific size. |
| **cnt_alloc** | Number of buffer allocations of a specific size. |
| **cnt_free** | Number of buffer releases of a specific size. |
| **cnt_wait** | Number of unfullfilled buffer allocations of specific size. |
| **maxalloc** | Largest wanted size. |

### 3.39.9 Example

```
sc_poolid_t pl;
sc_pool_cb_t pool_info;
int check;

pl = sc_poolIdGet("my_pool");
check = sc_poolInfo (pl, &pool_info);
```

### 3.39.10 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL | |
| Illegal pointer to **info** structure. | e0 = 0. |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pool ID. | e0 = pool ID. |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL | |
| Illegal module. | e0 = module ID. |

## 3.40 sc_poolKill

### 3.40.1 Description

This system call is used to kill a message pool.

A message pool can only be killed if all messages in the pool are freed (returned).

The killed pool memory can be reused later by a new pool if the size of the new pool is not exceeding the size of the killed pool.

Every process inside a module can kill a pool.

### 3.40.2 Syntax

```
void sc_poolKill (
   sc_poolid_t     plid
);
```

### 3.40.3 Parameter

| | |
|---|---|
| **plid** | ID of the pool to be killed. |

### 3.40.4 Return Value

None.

### 3.40.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet("my_pool");
sc_poolKill (pl);
```

### 3.40.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPOOL_IN_USE ¦ SC_ERR_MODULE_FATAL | |
| Pool is in use and cannot be killed. | e0 = pool cb.<br>e1 = pool lock counter. |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pool ID. | e0 = pool ID. |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL | |
| Illegal module. | e0 = module ID. |

## 3.41 sc_poolReset

### 3.41.1 Description

This system call is used to reset a message pool in its original state.

All messages in the pool must be freed and returned before a sc_poolReset call can be used.

The structure of the pool will be re-initialized. The message buffers in free-lists will be transformed back into unused memory. This “fresh” memory can now be used by sc_msgAlloc to allocate new messages.

Each process in a module can reset a pool.

### 3.41.2 Syntax

```
void sc_poolReset (
   sc_poolid_t     plid
);
```

### 3.41.3 Parameter

| | |
|---|---|
| **plid** | ID of the pool to reset. |

### 3.41.4 Return Value

None.

### 3.41.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet("my_pool");
sc_poolReset (pl);
```

### 3.41.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPOOL_IN_USE ¦ SC_ERR_MODULE_FATAL | |
| Pool is in use and no reset can be performed. | e0 = pool cb.<br>e1 = pool lock counter. |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pool ID. | e0 = pool ID. |

## 3.42 sc_procAtExit

### 3.42.1 Description

This system call is used to register a function to be called if a prioritized process is killed.

This allows to do some cleaning work if a process is killed. The sc_procAtExit system call is also used to register an error process from a module's init process. See

### 3.42.2 Syntax

```
void (*)(void) sc_procAtExit(
              void (*func)(void)
);
```

### 3.42.3 Parameter

| **func** | Function to be called. |
|---|---|
| <funcptr> | Pointer to the function to be called. |
| NULL | No function to register |
| SC_NIL | No function to register |

### 3.42.4 Return Value

Parameter func gets the pointer to the old function or NULL if none was registerred.

### 3.42.5 Example

```
void errorProcess(sc_errcode_t err,const sc_errMsg_t *errMsg);

void HelloSciopta(void)
{
  sc_procAtExit((sc_atExitFunc_t *)errorProcess);
}
```

### 3.42.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Illegal function pointer. | e0 = function pointer. |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Can only be called within a prioritized process. | e0 = process type. |

## 3.43 sc_procAttrGet

### 3.43.1 Description

This system call is used to get specific attributes for a process.

### 3.43.2 Syntax

```
int sc_procAttrGet(
          sc_pid_t        pid,
          sc_procAttr_t   attribute,
          void            *value
)
```

### 3.43.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to get the attribute. |
| SC_CURRENT_PID | Current running (caller) process. |

| **attribute** | Process attribute. See also sc_procAttr_t in proc.h. |
|---|---|
| SC_PROCATTR_NOP | Just check process |
| SC_PROCATTR_STACKUSAGE | Stack usage in % |
| SC_PROCATTR_NALLOC | Number of messages allocated |
| SC_PROCATTR_NQUEUE | Number messages in the queue |
| SC_PROCATTR_NOBSERVE | Number of observations |
| SC_PROCATTR_TYPE | process type |
| SC_PROCATTR_STATE | Process state<br>SC_PROCATTR_STATE_WAIT_RX<br>Process waits on message receive.<br>SC_PROCATTR_STATE_WAIT_TRG<br>Process waits on trigger.<br>SC_PROCATTR_STATE_WAIT_ALLOC<br>Process waits for message to be allocated.<br>SC_PROCATTR_STATE_WAIT_TMO<br>Process waits on message receive with timeout. |
| SC_PROCATTR_IS_CONNECTOR | Process is a connector if value is TRUE. |
| SC_PROCATTR_STOPCNT | Stopcounter. Process is stopped if value is != 0 |
| SC_PROCATTR_NAME | Process name |
| SC_PROCATTR_LAST | End of list |

| **value** | Pointer to the read process attribute value. |
|---|---|

### 3.43.4 Return Value

0 if process is not found.

1 value successfully written.

### 3.43.5 Example

```
int msg_number

if (sc_procAttrGet(SC_CURRENT_PID,SC_PROCATTR_NQUEUE,&msg_number)) {
     // msg_number valid
   } else {
     // msg_number not valid
}
```

### 3.43.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal process attribute. | e0 = process attribute.<br>e1 = 1 |
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Pointer to value not valid (NULL). | |

## 3.44 sc_procCreate2

### 3.44.1 Description

This system call is used to request the kernel daemon to create a process. The standard kernel daemon (**sc_kerneld**) needs to be defined and started at system configuration.

The process will be created within the callers module.

The maximum number of processes for a specific module is defined at module creation.

### 3.44.2 Syntax

```
sc_pid_t sc_procPrioCreate (
   const sc_pdb_t     *pdb,
   int                state,
   sc_poolid_t        plid
);
```

### 3.44.3 Parameter

| | |
|---|---|
| **pdb** | Pointer to the process descriptor block (pdb) which defines the process to create. |
| **state** | Process state after creation. |
| SC_PDB_STATE_RUN | The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes. |
| SC_PDB_STATE_STP | The process is stopped. Use the sc_procStart system call to start the process. |
| **plid** | Pool ID. |
| | Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated. |

### 3.44.4 Return Value

ID of the created process.

### 3.44.5 Process Descriptor Block pdb

The process descriptor block is a structure which is defining a process to be created.

It is included in the header file pcb.h.

It is devided into a common part valid for all process types and a process type specific part.

```
#define PDB_COMMON(para)     \
  __uint type;               \
  const char *name;          \
  void (* entry)(para);      \
  sc_bufsize_t stacksize;    \
  sc_pcb_t * pcb;            \
  char *stack;               \
  __u8 super;                \
  __u8 fpu;                  \
  __u16 spare

typedef struct sc_pdbcmn_s {
  PDB_COMMON(void);
} sc_pdbcommon_t;

typedef struct sc_pdbtim_s {
  PDB_COMMON(int);
  sc_ticks_t period;
  sc_ticks_t initial_delay;
} sc_pdbtim_t;

typedef struct sc_pdbint_s {
  PDB_COMMON(int);

  __uint vector;
} sc_pdbint_t;

typedef struct sc_pdbprio_s {
  PDB_COMMON(void);

  sc_ticks_t slice;
  __uint prio;
} sc_pdbprio_t;

typedef union sc_pdb_u {
  sc_pdbcommon_t cmn;
  sc_pdbprio_t prio;
  sc_pdbint_t irq;
  sc_pdbtim_t tim;
} sc_pdb_t;
```

### 3.44.6 Structure Members Common for all Process Types

| **type** | Process type. |
|---|---|
| SC_PROCPRIOCREATE | Prioritized process. |
| SC_PROCTIMCREATE | Timer process. |
| SC_PROCINTCREATE | Interrupt process. |
| **name** | Pointer to process name. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore. |
| **entry** | Pointer to process function. |
| | This is the address where the created process will start execution. |
| **stacksize** | Process stack size. |
| | The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize (160 bytes) |
| **pcb** | PCB pointer. |
| <pcb_ptr> | Pointer to a PCB |
| 0 | PCB will be allocated by the kernel. |
| **stack** | Stack address. |
| <stack_addr> | Pointer to a stack. Shall be taken from a pool or static memory within the module. |
| 0 | Stack will be allocated by the kernel. |
| **super** | Mode. |
| SC_KRN_FLAG_TRUE | Supervisor mode. |
| SC_KRN_FLAG_FALSE | User mode. |
| **fpu** | Floating point unit. |
| SC_KRN_FLAG_TRUE | Process does use FPU. |
| SC_KRN_FLAG_FALSE | Process does not use FPU. |
| **spare** | Not used, write 0. |

### 3.44.7 Additional Structure Members for Prioritized Processes

**slice** Time slice of the prioritized process.

**prio** Process priority.

The priority of the process which can be from 0 to 31. 0 is the highest priority.

### 3.44.8 Additional Structure Members for Interrupt Processes

**vector** Interrupt vector.

Interrupt vector connected to the created interrupt process. This is CPU-dependent.

### 3.44.9 Additional Structure Members for Timer Processes

**period** Period of time between calls to the timer process in ticks.

**initdelay** Initial delay in ticks before the first time call to the timer process.

### 3.44.10 Example

```
static const sc_pdbprio_t pdb = {
  /* process-type    */ PCB_TYPE_STATIC_PRI,
  /* process-name    */ "proc_A",
  /* function-name   */ proc_A,
  /* stacksize       */ 1024,
  /* pcb             */ 0,
  /* stack           */ 0,
  /* supervisor-flag */ SC_KRN_FLAG_TRUE,
  /* FPU-flag        */ SC_KRN_FLAG_FALSE,
  /* spare           */ 0,
  /* timeslice      */ 0,
  /* priority        */ 16
};
proc_A_pid = sc_procCreate2( (const sc_pdb_t *)&pdb,SC_PDB_STATE_RUN,0x0);
```

### 3.44.11 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter **pdb** not valid (0 or SC_NIL). | |
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL<br>mcb->freesize <= SIZEOF_PCB:<br>mcb->freesize <= stacksize element of pdb: | <br>e0 = mcb->freesize<br>e0 = stacksize element of pdb<br>e1 = mcb->freesize |
| KERNEL_EILL_VECTOR ¦ SC_ERR_MODULE_FATAL<br>Parameter vector (interrupt process) of pdb not valid (>SC_MAX_INT_VECTOR). | <br>e0 = parameter: vector<br>e1 = pdb<br>e2 = 9 |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL<br>Parameter slice (prioritized process) of pdb not valid:<br>Parameter period (timer process) of pdb not valid: | <br>e0 = parameter: slice<br>e0 = parameter: period (timer process)<br>e1 = pdb<br>e2 = 9 |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL<br>Parameter initial_dealy (timer process) of pdb not valid. | <br>e0 = parameter: initial_delay<br>e1 = pdb<br>e2 = 10 |
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Parameter type of pdb not valid. | <br>e0 = parameter: type<br>e1 = pdb<br>e2 = 0 |
| KERNEL_ENO_MORE_PROC ¦ SC_ERR_MODULE_FATAL<br>Number of maximum processes reached. | <br>e0 = No of process element of mcb<br>e1 = pdb<br>e2 = mcb |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_MODULE_FATAL<br>Parameter name of pdb not valid. | e0 = pdb parameter: name<br>e1 = pdb<br>e2 = 1 |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Module cb is not valid (mcb == SC_NIL). | e0 = mid<br>e1 = 0<br>e2 = 1 |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Message which holds pcb or stack is<br>not within current module. | e0 = Pointer to pcb or stack.<br>e1 = Pointer to mcb of current module.<br>e2 = Pointer to mcb of pcb/stack<br>message buffer. |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_MODULE_FATAL<br>Module cb is not valid (mcb == SC_NIL). | e0 = pdb parameter: priority<br>e1 = pdb<br>e2 = 10 |
| KERNEL_EILL_STACKSIZE ¦ SC_ERR_MODULE_FATAL<br>Parameter stack of pdb not valid. | e0 = pdb parameter: stack<br>e1 = pdb<br>e2 = 3 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter **pdb** not valid (!pdb \|\| pdb == SC_NIL). | e0 = Parameter **pdb**<br>e1 = 0<br>e2 = 0 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter **state** not valid. | e0 = Parameter **state**<br>e1 = 0<br>e2 = 1 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal module ID.<br>mid low 24 bits != 0 or<br>midx >= SC_MAX_MODULES | e0 = mid<br>e1 = 0<br>e2 = 2 |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter pcb of pdb not valid (==0). | e0 = pdb parameter: pcb<br>e1 = 0<br>e2 = 4 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter stack of pdb not valid (==0). | e0 = pdb parameter: stack<br>e1 = 0<br>e2 = 5 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter entry of pdb not valid. | e0 = pdb parameter: entry<br>e1 = pdb<br>e2 = 2 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter super not valid. | e0 = pdb parameter: super<br>e1 = pdb<br>e2 = 6 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter fpu not valid. | e0 = pdb parameter: fpu<br>e1 = pdb<br>e2 = 7 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter spare not valid. | e0 = 0<br>e1 = pdb<br>e2 = 8 |
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Not enough space for pcb or stack in selected message buffer. | e0 = Pointer to pcb or stack.<br>e1 = Pointer to mcb of current module.<br>e2 = Size of PCB or stacksize. |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_MODULE_FATAL<br>Parameter vector (interrupt process) of pdb vector already defined. | e0 = pdb parameter: vector<br>e1 = pdb<br>e2 = 9 |

## 3.45 sc_procDaemonRegister

### 3.45.1 Description

This system call is used to register a process daemon.

The process daemon manages process names in a SCIOPTA system. If a process calls sc_procIdGet the kernel will send a sc_procIdGet message to the process daemon. The process daemon will search the process name list and return the corresponding process ID to the kernel if found.

There can only be one process daemon per SCIOPTA system.

The standard process daemon **sc_procd** is included in the SCIOPTA kernel. This process daemon needs to be defined and started at system configuration as a static process.

### 3.45.2 Syntax

```
int sc_procRegisterDaemon (void);
```

### 3.45.3 Parameter

None.

### 3.45.4 Return Value

0 if the process daemon was successfully installed.

!=0 if the process daemon could not be installed.

### 3.45.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Calling process is not a prioritized process. | |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Calling process is not in system module. | |

## 3.46 sc_procDaemonUnregister

### 3.46.1 Description

This call is used by a process daemon to unregister itself.

A statically installed process daemon cannot be unregistered.

### 3.46.2 Syntax

```
int sc_procUnregisterDaemon (void);
```

### 3.46.3 Parameter

None

### 3.46.4 Return Value

0 if the process daemon was not a process daemon.

!=0 if the process daemon was successfully unregistered.

### 3.46.5 Errors

None

## 3.47 sc_procHookRegister

### 3.47.1 Description

This system call will register a process hook of the type defined in parameter **type**. The type can be a create hook, kill hook or swap hook.

Each time a process will be created the create hook will be called if there is one installed.

Each time a process will be killed the kill hook will be called if there is one installed.

Each time a process swap is initiated by the kernel the swap hook will be called if there is one installed.

### 3.47.2 Syntax

```
sc_procHook_t *sc_procHookRegister (
   int             type,
   sc_procHook_t   *newhook
);
```

### 3.47.3 Parameter

| **type** | Type of process hook. |
|---|---|
| SC_SET_PROCCREATE_HOOK | Registers a process create hook. Every time a process is created, this hook will be called. |
| SC_SET_PROCKILL_HOOK | Registers a process kill hook. Every time a process is killed, this hook will be called. |
| SC_SET_PROCSWAP_HOOK | Registers a process swap hook. Every time a process swap is initiated by the kernel, this hook will be called. |

| **newhook** | Process hook function pointer. |
|---|---|
| <funcptr> | Function pointer to the process hook. |
| NULL | Removes and unregisters the process hook. |

### 3.47.4 Return Value

Function pointer to the previous process hook if process hook was registered.

0 if no process hook was registered.

### 3.47.5 Example

```
druidHook = sc_procHookRegister(SC_SET_PROCSWAP_HOOK,swapHook);
```

### 3.47.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Illegal process hook type. | e0 = type |

## 3.48 sc_procIdGet

### 3.48.1 Description

This call is used to get the process ID of a process by providing the name of the process.

In SCIOPTA processes are organized in systems (CPUs) and modules within systems. There is always at least one module called system module (module 0). Depending where the process resides (system, module) not only the process name needs to be supplied but also the including system and module name.

This call forwards the request to the process daemon. The standard process daemon (**sc_procd**) needs to be defined and started at system configuration.

### 3.48.2 Syntax

```
sc_pid_t sc_procIdGet (
   const char           *path,
   sc_ticks_t           tmo
);
```

### 3.48.3 Parameter

| **path** | Pointer to the path with the name of the process. |
|---|---|
| path::=process_name | Process resides within the caller's module. |
| path::='/'process_name | Process resides in the system module of the caller´s target. |
| path::='//'<system_name>'/'process_name | Process resides in the system module of an external target. |
| path::='/'<module_name>'/'process_name | Process resides in another than the system module of the caller's target. |
| path::='//'<system_name>'/'<module_name>'/'process_name | If the process resides in another than the system module of an external target. |

| **tmo** | Time to wait for a response in ticks. |
|---|---|
| | This parameter is not allowed if asynchronous timeout is disabled at system configuration (**SCONF**). |
| SC_NO_TMO | No timeout, returns immediately. |
| 0 < tmo < SC_TMO_MAX | Timeout value in system ticks. |

### 3.48.4 Return Value

Process ID of the found process if the process was found within the tmo time period.

Current process ID (process ID of the caller) if parameter path is NULL and parameter tmo is SC_NO_TMO.

SC_ILLEGAL_PID if process was not found within the tmo time period.

### 3.48.5 sc_procIdGet in Interrupt Processes

The sc_procIdGet system call can also be used in an interrupt process. The process daemon sends a reply message to the interrupt process (interrupt process src parameter == 1).

The reply message is defined as follows:

```
#define SC_PROCIDGETMSG_REPLY (SC_MSG_BASE+0x10d)

typedef struct sc_procIdGetMsgReply_s {
   sc_msgid_t           id;
   sc_pid_t             pid;
   sc_errorcode_t       error;
   int                  more;
)sc_procIdGetMsgReply_t;
```

### 3.48.6 Example

```
sc_pid_t slave_pid;

slave_pid = sc_procIdGet("slave",SC_NO_TMO);
```

### 3.48.7 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_PROCESS_FATAL<br>Illegal path. | e0 = pointer to path |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Process is PCB_TYPE_IDL. | e0 = process type |

## 3.49 sc_procKill

### 3.49.1 Description

This system call is used to request the kernel daemon to kill a process.

The standard kernel daemon (**sc_kerneld**) needs to be defined and started at system configuration.

Any process type (prioritized, interrupt, timer) can be killed. No external processes (on a remote CPU) can be killed.

If a cleaning-up is executed (depending on the **flag** parameter) all message buffers owned by the process will be returned to the message pool. If an observe is active on that process the observe messages will be sent to the observing processes. A significant time can elapse before a possible observe message is posted.

### 3.49.2 Syntax

```
void sc_procKill (
   sc_pid_t        pid,
   flags_t         flag
);
```

### 3.49.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to be killed. |
| SC_CURRENT_PID | Current running (caller) process. |

| **flag** | Process kill flag. |
|---|---|
| 0 | A cleaning up will be executed. |
| SC_PROCKILL_KILL | No cleaning up will be requested. |

### 3.49.4 Return Value

None

### 3.49.5 Example

```
sc_procKill(SC_CURRENT_PID,0);
```

### 3.49.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>pid == 0<br>pid == SC_ILLEGAL_PID<br>mid too large<br>process index too large. | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_WARNING<br>Process killed or pid not valid. | e0 = pid |

## 3.50 sc_procNameGet

### 3.50.1 Description

This call is used to get the full name of a process.

The name will be returned inside a SCIOPTA message buffer which is allocated by the kernel. The kernel sets the caller as owner of the message.

The user must free the message after use.

If the process (pid) does not exist (or does not exist any more) and the pid has a valid value (between min and max) the kernel calls the error hook (if it exists) and generates a warning. After returning from the error hook the system call sc_procNameGet returns zero. If the pid has no valid value (out of scope) the kernel calls the error hook with a fatal error. If there is no error hook, he system loops at the error label.

### 3.50.2 Syntax

```
sc_msg_t sc_procNameGet (
   sc_pid_t           pid
);
```

### 3.50.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process where the name is requested. |
| SC_CURRENT_PID | Current running (caller) process. |

### 3.50.4 Return Value

Message owned by the caller if the process exists.

If the return value is nonzero the returned message buffer is owned by the caller and the message is of type **sc_procNameGetMsgReply_t** and the message ID is **SC_PROCNAMEGETMSG_REPLY**. The message data contains the 0 terminated ASCII name string of the process. The message is defined in the **sciopta.msg** include file.

```
typedef struct sc_procNameGetMsgReply_s {
   sc_msgid_t      id;
   sc_errcode_t    error;
   char            target[SC_MODULE_NAME_SIZE+1];
   char            module[SC_MODULE_NAME_SIZE+1];
   char            process[SC_PROC_NAME_SIZE+1];
} sc_procNameGetMsgReply_t;
```

### 3.50.5 Example

```
sc_msg_t senderName;

senderName = sc_procNameGet(sender);
```

### 3.50.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>Illegal pid. | e0 = pid |

## 3.51 sc_procObserve

### 3.51.1 Description

This system call is used to supervise a process.

The sc_procObserve system call will request the message to be sent back if the given process dies (process supervision). If the supervised process disappears from the system (process ID) the kernel will send the requested and registered message to the supervisor process.

The process to supervise can be external (in another CPU).

### 3.51.2 Syntax

```
void sc_procObserve (
   sc_msgptr_t       msgptr,
   sc_pid_t          pid
);
```

### 3.51.3 Parameter

**msgptr** Pointer to the observe message pointer.

Pointer to the message which will be returned if the supervised process disappears. The message must be of the following type:

```
struct err_msg {
   sc_msgid_t      id;
   sc_errcode_t    error;
   /* user defined data */
};
```

**pid** Process ID of the process which will be supervised.

### 3.51.4 Return Value

None.

### 3.51.5 Example

```
struct dead_s {
  sc_msgid_t id;
  sc_errcode_t errcode;
};

union sc_msg{
  sc_msgid_t id;
  struct dead_s dead;
};

sc_msg_t msg;

msg = sc_msgAlloc(sizeof(struct dead_s),0xdead,0,SC_FATAL_IF_TMO);
sc_procObserve(&msg,slave_pid);
```

### 3.51.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>Illegal pid. | e0 = pid |

## 3.52 sc_procPathCheck

### 3.52.1 Description

This call is used to check if the construction of a path is correct. It checks the lengths of the system, module and process names and the number of slashes.

### 3.52.2 Syntax

```
sc_errcode_t sc_procPathCheck (
   char              *path
);
```

### 3.52.3 Parameter

| **path** | Pointer to the path with the name of the process. |
|---|---|
| path::=process_name | Process resides within the caller's module. |
| path::='/'process_name | Process resides in the system module of the caller´s target. |
| path::='//'<system_name>'/'process_name | Process resides in the system module of an external target. |
| path::='/'<module_name>'/'process_name | Process resides in another than the system module of the caller's target. |
| path::='//'<system_name>'/'<module_name>'/'process_name | If the process resides in another than the system module of an external target. |

### 3.52.4 Return Value

!= 0 if the path is correct.

== 0 if the path is wrong.

### 3.52.5 Example

```
if ( !sc_procPathCheck("//target0//target1/module/slave") ){
    sc_miscError(0x1002,0);
}
```

### 3.52.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_SYSTEM_FATAL<br>Illegal path (pointer to path == 0). | e0 = pointer to path name. |

## 3.53 sc_procPathGet

### 3.53.1 Description

This call is used to get the full path of a process.

The path will be returned inside a SCIOPTA message buffer which is allocated by the kernel. The kernel sets the caller as owner of the message.

The user must free the message after use.

If the process (pid) does not exist (or does not exist any more) and the pid has a valid value (between min and max) the kernel calls the error hook (if it exists) and generates a warning. After returning from the error hook the system call sc_procNameGet returns zero. If the pid has no valid value (out of scope) the kernel calls the error hook with a fatal error. If there is no error hook, he system loops at the error label.

### 3.53.2 Syntax

```
sc_msg_t sc_procPathGet (
   sc_pid_t          pid,
   flags_t           flags
);
```

### 3.53.3 Parameter

| **pid** | Process ID of the process where the path is requested. |
|---|---|

| **flags** | sc_procPathGet flags. |
|---|---|
| != 0 | The full path is returned:<br>'//'<**system_name**>'/'<**module_name**>'/'**process_name** |
| ==0 | The short path is returned:<br>'/'<**module_name**>'/'**process_name** |

### 3.53.4 Return Value

Message owned by the caller if the process exists.

If the return value is nonzero the returned message buffer is owned by the caller and the message is of type **sc_procPathGetMsgReply_t** and the message ID is **SC_PROCPATHGETMSG_REPLY**. The message data contains the 0 terminated ASCII name string of the process. The message is defined in the **sciopta.msg** include file.

```
typedef struct sc_procPathGetMsgReply_s {
   sc_msgid_t        id;
   sc_pid_t          pid;
   sc_errcode_t      error;
   char              path[1];
} sc_procPathGetMsgReply_t;
```

### 3.53.5 Example

```
sc_msg_t msg;

msg = sc_procPathGet(SC_CURRENT_PID,1);
if ( strstr(msg->path.path,"node1") ){
   remote = "//node2/node2/echo";
} else {
   remote = "//node1/node1/echo";
}
```

### 3.53.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Illegal pid. | e0 = pid |

## 3.54 sc_procPpidGet

### 3.54.1 Description

This call is used to get the process ID of the parent (creator) of a process.

### 3.54.2 Syntax

```
sc_pid_t sc_procPpidGet (
   sc_pid_t        pid
);
```

### 3.54.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to be killed. |
| SC_CURRENT_PID | Current running (caller) process. |

### 3.54.4 Return Value

Process ID of the parent process if the parent process exists

Process ID of the parent process of the caller if parameter pid was SC_CURRENT_PID.

SC_ILLEGAL_PID if the parent process does no longer exist.

### 3.54.5 Example

```
typedef struct key_s {
  __u8 scan;
  __u8 cntrl;
} sckey_t;

#define KEYB_MSG 0x30000001
typedef struct keyb_msg_s{
  sc_msgid_t id;
  sckey_t data;
} keyb_msg_t;
sc_msg_t msg;

sc_pid_t ttyd_pid = sc_procPpidGet(SC_CURRENT_PID);
msg = sc_msgAlloc(sizeof(keyb_msg_t),KEYB_MSG,0,SC_ENDLESS_TMO);
if ( msg ){
   msg->keyb.data.scan = key;
   msg->keyb.data.cntrl = control_keys;

(&msg,ttyd_pid,0);
}
```

### 3.54.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Illegal pid. | e0 = pid |

## 3.55 sc_procPrioGet

### 3.55.1 Description

This process is used to get the priority of a prioritized process.

In SCIOPTA the priority ranges from 0 to 31. 0 is the highest and 31 the lowest priority.

### 3.55.2 Syntax

```
sc_prio_t sc_procPrioGet (
   sc_pid_t     pid
);
```

### 3.55.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to get the priority. |
| SC_CURRENT_PID | Current running (caller) process. |

### 3.55.4 Return Value

Process priority.

32 if there was a warning of an invalid CONNECTOR process.

### 3.55.5 Example

```
// Create process "proc_A" with lower priority than caller

sc_pid_t proc_A_pid;
sc_prio_t prio = sc_procPrioGet (SC_CURRENT_PID) + 1;

    static const sc_pdbprio_t pdb = {
      /* process-type    */ PCB_TYPE_STATIC_PRI,
      /* process-name    */ "proc_A",
      /* function-name   */ proc_A,
      /* stacksize       */ 1024,
      /* pcb             */ 0,
      /* stack           */ 0,
      /* supervisor-flag */ SC_KRN_FLAG_TRUE,
      /* FPU-flag        */ SC_KRN_FLAG_FALSE,
      /* spare           */ 0,
      /* time-slice      */ 0,
      /* priority        */ prio
    };
    proc_A_pid = sc_ProcCreate2( (const sc_pdb_t *)&pdb,SC_PDB_STATE_RUN,0x0);
```

### 3.55.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pid. | e0 = pid |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = pid<br>e1 = process type |

## 3.56 sc_procPrioSet

### 3.56.1 Description

This call is used to set the priority of a process.

Only the priority of the caller's process can be set and modified.

If the new priority is lower to other ready processes the kernel will initiate a context switch and swap-in the process with the highest priority.

If there are already existing processes at the same priority, the process which has just moved to that priority will be put at the end of the list and swapped-out.

Init processes are treated specifically. An init process is the first process in a module and does always exist. An init process can set its priority on level 32 (this is the only process which can have a priority of 32). This will redefine the process and it becomes an idle process. The init process should always set its priority to 32 after it has accomplished its initialization work. The idle process will be called by the kernel if there are no processes ready for getting the CPU (all are in a waiting state).

### 3.56.2 Syntax

```
void sc_procPrioSet (
   sc_prio_t       prio
);
```

### 3.56.3 Parameter

| | |
|---|---|
| **prio** | Process priority. |
| | The new priority of the caller's process (0 .. 31). |

### 3.56.4 Return Value

None.

### 3.56.5 Example

```
// Switch caller to lowest-priority

sc_procPrioSet(31);
```

### 3.56.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = process type |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_PROCESS_FATAL | |
| Illegal priority. Priority == 32. | e0 = process type<br>e1 = module priority |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_PROCESS_FATAL | |
| Illegal priority.<br>Priority > 32.<br>priority > max_prio. | e0 = process type<br>e1 = -1 |

## 3.57 sc_procSchedLock

### 3.57.1 Description

This system call will lock the scheduler and return the number of times it has been locked before.

SCIOPTA maintains a scheduler lock counter. If the counter is 0 scheduling is activated. Each time a process calls sc_procSchedLock the counter will be incremented.

Interrupts are not blocked if the scheduler is blocked by sc_procSchedLock.

### 3.57.2 Syntax

```
int sc_procSchedLock (void);
```

### 3.57.3 Parameter

None

### 3.57.4 Return Value

Internal scheduler lock counter. Number of times the scheduler has been locked.

### 3.57.5 Example

```
// count instances

sc_procSchedLock();
++counter;
sc_procSchedUnlock();
```

### 3.57.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | |

## 3.58 sc_procSchedUnlock

### 3.58.1 Description

This system call will unlock the scheduler.

SCIOPTA maintains a scheduler lock counter. Each time a process calls sc_procSchedUnlock the counter will be decremented. If the counter reaches a value of 0 the SCIOPTA scheduler is called and activated. The ready process with the highest priority will be swapped in.

It is illegal to unlock a not blocked scheduler.

### 3.58.2 Syntax

```
void sc_procSchedUnlock (void);
```

### 3.58.3 Parameter

None.

### 3.58.4 Return Value

None

### 3.58.5 Example

```
// count instances

sc_procSchedLock();
++counter;
sc_procSchedUnlock();
```

### 3.58.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|

KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL
Caller is not a prioritized process.

KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL
Interrupts are locked.

KERNEL_EUNLOCK_WO_LOCK ¦ SC_ERR_MODULE_FATAL
Lockcounter == 0.

## 3.59 sc_procSliceGet

### 3.59.1 Description

This call is used to get the time slice of a prioritized or timer process.

The time slice is the period of time between calls to the timer process in ticks or the the slice of round-robin scheduled prioritized processes on the same priority.

### 3.59.2 Syntax

```
sc_ticks_t sc_procSliceGet (
   sc_pid_t        pid
);
```

### 3.59.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to get the time slice. |
| SC_CURRENT_PID | Current running (caller) process. |

### 3.59.4 Return Value

Period of time between calls to any given timer process in ticks

### 3.59.5 Example

```
sc_ticks_t new_ticks;
new_ticks = sc_procSliceGet(SC_CURRENT_PID);
```

### 3.59.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Process/Module disappeared | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pid. | e0 = pid |

## 3.60 sc_procSliceSet

### 3.60.1 Description

This call is used to set the time slice of a timer process. The modified time slice will become active after the current time slice expired or if the timer gets started. It can only be activated after the old time slice has elapsed.

### 3.60.2 Syntax

```
void sc_procSliceSet (
   sc_pid_t          pid,
   sc_ticks_t        slice
);
```

### 3.60.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to set the time slice. |
| SC_CURRENT_PID | Current running (caller) process. |
| **slice** | New period of time between calls to the timer process in ticks. |

### 3.60.4 Return Value

None.

### 3.60.5 Example

```
sc_procSliceSet(SC_CURRENT_PID,5);
```

### 3.60.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Process/Module disappeared | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pid. | e0 = pid |

## 3.61 sc_procStart

### 3.61.1 Description

This system call will start a prioritized or timer process.

SCIOPTA maintains a start/stop counter per process. If the counter is >0 the process is stopped. Each time a process calls sc_procStart the counter will be decremented. If the counter has reached the value of 0 the process will start.

If the started process is a prioritized process and its priority is higher than the priority of the currently running process, it will be swapped in and the current process swapped out.

If the started process is a timer process, it will be entered into the timer list with its time slice.

It is illegal to start a process which was not stopped before.

### 3.61.2 Syntax

```
void sc_procStart (
   sc_pid_t        pid
);
```

### 3.61.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to be started. |

### 3.61.4 Return Value

None.

### 3.61.5 Example

```
sc_procStart(proc_A_pid);
```

### 3.61.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized or timer process. | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller. | e0 = pid |
| Process is init process. | |
| KERNEL_ESTART_NOT_STOPPED ¦ SC_ERR_MODULE_FATAL | |
| Stop counter already 0 | e0 = pid |

## 3.62 sc_procStop

### 3.62.1 Description

This system call will stop a prioritized or timer process.

SCIOPTA maintains a start/stop counter per process. If the counter is >0 the process is stopped. Each time a process calls sc_procStop the counter will be incremented.

If the stopped process is the currently running prioritized process, it will be halted and the next ready process will be swapped in.

If a timer process will be stopped, it will immediately removed from the timer list and the system will not wait until the current time slice expires.

The sc_procStop call will clear a pending timeout.

### 3.62.2 Syntax

```
void sc_procStop (
   sc_pid_t        pid
);
```

### 3.62.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to be stopped. |
| SC_CURRENT_PID | Current running (caller) process will be stopped. |

### 3.62.4 Return Value

None.

### 3.62.5 Example

```
sc_procStop(SC_CURRENT_PID);
```

### 3.62.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROC ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized or timer process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller.<br>Process is init process. | e0 = pid |
| KERNEL_EILL_VALUE ¦ SC_ERR_MODULE_FATAL | |
| Stop counter overrun | |

## 3.63 sc_procUnobserve

### 3.63.1 Description

This system call is used to cancel an installed supervision of a process.

The message given by the sc_procObserve system call will be freed by the kernel.

### 3.63.2 Syntax

```
void sc_procUnobserve (
   sc_pid_t   pid,
   sc_pid_t   observer
);
```

### 3.63.3 Parameter

| **pid** | Supervised process ID. |
|---|---|
| <pid> | Process ID of the process which is supervised. |

| **observer** | Observer process ID. |
|---|---|
| <pid> | Process ID of the observer process. |
| SC_CURRENT_PID | Current process is observer. |

### 3.63.4 Return Value

None.

### 3.63.5 Example

```
sc_procUnobserve(slave,SC_CURRENT_PID);
```

### 3.63.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller. | e0 = pid |
| Process is init process. | |

## 3.64 sc_procVarDel

### 3.64.1 Description

This system call is used to remove a process variable from the process variable data area.

### 3.64.2 Syntax

```
int sc_procVarDel (
   sc_tag_t        tag
);
```

### 3.64.3 Parameter

| | |
|---|---|
| **tag** | Process variable tag. |
| | User defined tag of the process variable which was set by the sc_procVarSet call. |

### 3.64.4 Return Value

0 if the system call fails and the process variable could not be removed.

!=0 if the process variable was successfully removed.

### 3.64.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No process variable set | |

## 3.65 sc_procVarGet

### 3.65.1 Description

This system call is used to read a process variable.

### 3.65.2 Syntax

```
int sc_procVarGet (
   sc_tag_t        tag,
   __u32           *value
);
```

### 3.65.3 Parameter

| | |
|---|---|
| **tag** | Process variable tag. |
| | User defined tag of the process variable which was set by the sc_procVarSet call. |
| **value** | Process variable. |
| | Pointer to the variable where the process variable will be stored. |

### 3.65.4 Return Value

0 if the system call fails and the process variable could not be read.

!=0 if the process variable was successfully read.

### 3.65.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No procVar set or value == NULL | |

## 3.66 sc_procVarInit

### 3.66.1 Description

This system call is used to setup and initialize a process variable area.

The user needs to allocate a message with the size of

```
sizeof(sc_varpool_t) + ( n * sizeof(sc_local_t) )
```

which holds the process variables.

### 3.66.2 Syntax

```
void sc_procVarInit (
   sc_msgptr_t          varpool,
   unsigned int         n
);
```

### 3.66.3 Parameter

| | |
|---|---|
| **varpool** | Process variable buffer. |
| <ptr> | Pointer to the message buffer holding the process variables. |
| 0 or SC_NIL | The kernel will allocate the message buffer. |
| **n** | Maximum number of process variables. |

### 3.66.4 Return Value

None.

### 3.66.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No procVar set or value == NULL | |
| KERNEL_ENOT_OWNER ¦ SC_ERR_PROCESS_FATAL<br>Process does not own the buffer | e0 = owner |
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Size too small. | e0 = size |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_PROCESS_FATAL<br>Process variable already set. | e0 = pointer to message buffer |

## 3.67 sc_procVarRm

### 3.67.1 Description

This system call is used to remove a whole process variable area.

### 3.67.2 Syntax

```
sc_msg_t sc_procVarRm (void);
```

### 3.67.3 Parameter

None

### 3.67.4 Return Value

Pointer to the message buffer holding the process variables.

### 3.67.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No procVar set or value == NULL | |

## 3.68 sc_procVarSet

### 3.68.1 Description

This system call is used to define or modify a process variable.

### 3.68.2 Syntax

```
int sc_procVarSet (
   sc_tag_t        tag,
   __u32           value
);
```

### 3.68.3 Parameter

| | |
|---|---|
| **tag** | Process variable tag. |
| | User defined tag of the process variable. |
| **value** | Value of the process variable. |

### 3.68.4 Return Value

0 if the system call fails and the process variable could not be defined or modified.

!=0 if the process variable was successfully defined or modified.

### 3.68.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No procVar set | |

## 3.69 sc_procVectorGet

### 3.69.1 Description

This system call is used to get the interrupt vector of an interrupt process.

### 3.69.2 Syntax

```
int sc_procVectorGet (
        sc_pid_t        pid
);
```

### 3.69.3 Parameter

| | |
|---|---|
| **pid** | Process ID. |
| | Process ID of the interrupt process. |

### 3.69.4 Return Value

Interrupt vector of the interrupt process.

### 3.69.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not an interrupt process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller.<br>Process is init process. | e0 = pid |

## 3.70 sc_procWakeupEnable

### 3.70.1 Description

This system call is used to enable the wakeup of a timer or interrupt process.

### 3.70.2 Syntax

```
void sc_procWakeupEnable(void)
```

### 3.70.3 Parameter

None.

### 3.70.4 Return Value

None.

### 3.70.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Caller is not an interrupt or timer process. | e0 = process type |

## 3.71 sc_procWakeupDisable

### 3.71.1 Description

This system call is used to disable the wakeup of a timer or interrupt process.

### 3.71.2 Syntax

```
void sc_procWakeupDisable(void)
```

### 3.71.3 Parameter

None.

### 3.71.4 Return Value

None.

### 3.71.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Caller is not an interrupt or timer process. | e0 = process type |

## 3.72 sc_procYield

### 3.72.1 Description

This system call is used to yield the CPU to the next ready process within the current process's priority group.

### 3.72.2 Syntax

```
void sc_procYield (void);
```

### 3.72.3 Parameter

None.

### 3.72.4 Return Value

None.

### 3.72.5 Example

```
sc_procYield();
```

### 3.72.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Scheduling not possible as it is locked. | |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | e0 = process type |

## 3.73 sc_sleep

### 3.73.1 Description

This call is used to suspend the calling process for a defined time. The requested time must be given in number of system ticks.

The calling process will get into a waiting state and swapped out. After the timeout has elapsed the process will become ready again and will be swapped in if it has the highest priority of all ready processes.

The process will be waiting for at least the requested time minus one system tick.

### 3.73.2 Syntax

```
sc_time_t sc_sleep (
   sc_ticks_t      tmo
);
```

### 3.73.3 Parameter

| | |
|---|---|
| **tmo** | Timeout. |
| | Number of system ticks to wait. |

### 3.73.4 Return Value

Activation time. The absolute time (tick counter) value when the calling process became ready.

### 3.73.5 Example

```
void resetPHY() {
    // Setup some I/O pins
    sc_sleep(2);
    // Setup some other I/O pins
    sc_sleep(2);
    // Setup last I/O pins
    sc_sleep(2);
}
```

### 3.73.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Scheduler is locked. | |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL<br>Illegal timeout value. | e0 = tmo |

## 3.74 sc_tick

### 3.74.1 Description

This function calls directly the kernel tick function and advances the kernel tick counter by 1.

The kernel maintains a counter to control the timing functions. The timer needs to be incremented in regular intervals.

If the processor contains an on-chip timer the kernel uses it by default. The on-chip timer is set-up by the kernel automatically at start-up and all parameters are defined in the SCIOPTA configuration utility. In this case the user does not need to call sc_tick.

If the processor does not have an on-chip timer or the user does not want to use it, sc_tick must be called explicitly by the user from within an user interrupt process. The user is responsible to write the user interrupt process and to setup the timer chip to define the requested tick interval.

This system call is only allowed in hardware activated interrupt processes and is not allowed to be used in interrupt processes which have been activated by trigger, message sent, process creation and killing.

### 3.74.2 Syntax

```
void sc_tick (void);
```

### 3.74.3 Parameter

None.

### 3.74.4 Return Value

None.

### 3.74.5 Example

```
SC_INT_PROCESS(sysTick,src)
{
   if ( src == 0 ){
      sc_tick();
         /* Handle timer irq */
   }
}
```

### 3.74.6 Errors

None

## 3.75 sc_tickActivationGet

### 3.75.1 Description

This functions returns the tick time of last activation of the calling process.

### 3.75.2 Syntax

```
sc_time_t sc_tickActivationGet(void)
```

### 3.75.3 Parameter

None

### 3.75.4 Return Value

Activation time. The absolute time (tick counter) value when the calling process became ready.

### 3.75.5 Errors

None

## 3.76 sc_tickGet

### 3.76.1 Description

This call is used to get the actual kernel tick counter value. The number of system ticks from the system start are returned.

### 3.76.2 Syntax

```
sc_time_t sc_tickGet (void);
```

### 3.76.3 Parameter

None

### 3.76.4 Return Value

Current value of the tick timer.

### 3.76.5 Example

```
t = sc_tickGet();
for(i = 0; i < 100; ++i ){
   cache_flush_range((char *)0x3000000,0x8000);
   memcpy32B((char *)0x2000000,(char *)0x3000000,0x100000);
}
t = sc_tickGet()-t;
kprintf(0,"Copy in 100MB in %d ms\n",sc_tickTick2Ms(t));
```

### 3.76.6 Errors

None

## 3.77 sc_tickLength

### 3.77.1 Description

This system call is used to set the current system tick length in micro seconds.

### 3.77.2 Syntax

```
__u32 sc_tickLength (
   __u32           ticklength
);
```

### 3.77.3 Parameter

| **ticklength** | Tick length. |
|---|---|
| 0 | The current tick length will just be returned without modifying it. |
| <tick_length> | The tick length in micro seconds. |

### 3.77.4 Return Value

Tick length in microseconds.

### 3.77.5 Example

```
kprintf(0,"Setting up system-timer ...");
pit_init(200, 0);// 200Hz == 5ms
sc_tickLength(4999);
pic_irqEnable(PIC_SRC_PIT0);
kprintf(0,"done\n");
```

### 3.77.6 Errors

None

## 3.78 sc_tickMs2Tick

### 3.78.1 Description

This system call is used to convert a time from milliseconds into system ticks.

### 3.78.2 Syntax

```
sc_time_t sc_tickMs2Tick (
   __u32           ms
);
```

### 3.78.3 Parameter

| | |
|---|---|
| **ms** | Time in milliseconds. |

### 3.78.4 Return Value

Time in system ticks.

### 3.78.5 Example

```
int tmo = 1000;

while (tmo < 8000 && (dev = ips_devGetByName ("eth0")) == NULL) {
   sc_sleep(sc_tickMs2Tick (tmo));
   tmo *= 2;
}
```

### 3.78.6 Errors

None

## 3.79 sc_tickTick2Ms

### 3.79.1 Description

This system call is used to convert a time from system ticks into milliseconds.

The calculation is based on tick-length and limited to 32 bit.

### 3.79.2 Syntax

```
__u32 sc_tickTick2Ms (
   sc_ticks_t           t
);
```

### 3.79.3 Parameter

| | |
|---|---|
| **t** | Time in system ticks. |

### 3.79.4 Return Value

Time in milliseconds.

### 3.79.5 Example

```
t0 = sc_tickGet();
for(cnt = 0 ; cnt < 1000000; ++cnt){
   sc_procYield();
}
t1 = sc_tickGet();
t2 = sc_tickTick2Ms(t1-t0);
```

### 3.79.6 Errors

None

## 3.80 sc_tmoAdd

### 3.80.1 Description

This system call is used to request a timeout message from the kernel after a defined time.

The caller needs to allocate a message and include the pointer to this message in the call. The kernel will send this message back to the caller after the time has expired.

This is an asynchronous call, the caller will not be blocked.

The registered timeout can be cancelled by the sc_tmoRm call before the timeout has expired. This system call returns the timeout ID which could be used later to cancel the timeout.

### 3.80.2 Syntax

```
sc_tmoid_t sc_tmoAdd (
   sc_ticks_t         tmo,
   sc_msgptr_t        msgptr
);
```

### 3.80.3 Parameter

| | |
|---|---|
| **tmo** | Timeout. |
| | Number of system tick after which the message will be sent back by the kernel. |
| **msgptr** | Pointer to the timeout message pointer. |
| | Pointer to the message pointer of the message which will be sent back by the kernel after the elapsed time. |

### 3.80.4 Return Value

Timeout ID.

### 3.80.5 Example

```
sc_tmoid_t tmoid;

msg = sc_msgAlloc(sizeof(ctrl_poll_t),TCS_CTRL_POLL,0,SC_FATAL_IF_TMO);
tmoid = sc_tmoAdd((sc_ticks_t)sc_tickMs2Tick(1000),&msg);
```

### 3.80.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_PROCESS_FATAL<br>Caller is not a prioritized process. | e0 = process type |
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>Illegal timeout value. | e0 = tmo |
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_PROCESS_FATAL<br>Message is already a timeout message. | e0 = Pointer to the message |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |

## 3.81 sc_tmoRm

### 3.81.1 Description

This system call is used to remove a timeout before it is expired.

If the process has already received the timeout message and the user still tries to cancel the timeout with the sc_tmoRm call, the kernel will generate a fatal error.

After the call the value of the timeout id is zero.

### 3.81.2 Syntax

```
sc_msg_t sc_tmoRm (
   sc_tmoid_t         *id
);
```

### 3.81.3 Parameter

| | |
|---|---|
| **id** | Timeout ID. |
| | Pointer to timeout ID which was given when the timeout was registered by the sc_tmoAdd call. |

### 3.81.4 Return Value

Pointer to the timeout message which was defined at registering it by the sc_tmoAdd call.

### 3.81.5 Example

```
sc_msg_t tmomsg;
tmomsg = sc_tmoRm(&tmoid);
sc_msgFree(&tmomsg);
```

### 3.81.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Timeout expired, but not received (not in the queue) | |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Timeout ID is already cleared. | e0 = Pointer to timeout ID |
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = pid of the owner |

## 3.82 sc_trigger

### 3.82.1 Description

This system call is used to activate a process trigger.

The trigger value of the addressed process trigger will be incremented by 1. If the trigger value becomes greater than zero the process waiting at the trigger will become ready and swapped in if it has the highest priority of all ready processes.

### 3.82.2 Syntax

```
void sc_trigger (
   sc_pid_t        pid
);
```

### 3.82.3 Parameter

| | |
|---|---|
| **pid** | Process ID. |
| | ID of the process which trigger will be activated. |

### 3.82.4 Return Value

None.

### 3.82.5 Example

```
sc_pid_t slave_pid;

slave_pid = sc_procIdGet("slave",SC_NO_TMO);

sc_trigger(slave_pid);
```

### 3.82.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL | |
| Illegal process type. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Process disappeared | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL | |
| Process is an init or external process | e0 = pid |

## 3.83 sc_triggerValueGet

### 3.83.1 Description

This system call is used to get the value of a process trigger.

The caller can get the trigger value from any process in the system.

### 3.83.2 Syntax

```
sc_triggerval_t sc_triggerValueGet (
   sc_pid_t        pid
);
```

### 3.83.3 Parameter

| | |
|---|---|
| **pid** | Process ID. |
| | ID of the process which trigger is returned. |

### 3.83.4 Return Value

Trigger value.

INT_MAX if no valid process.

### 3.83.5 Example

```
sc_pid_t slave_pid;
sc_triggerval_t slavetrig;

slave_pid = sc_procIdGet("slave",SC_NO_TMO);

slave_trig = sc_triggerValueGet(slave_pid);
```

### 3.83.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL | |
| Process is an init or external process | e0 = pid |

## 3.84 sc_triggerValueSet

### 3.84.1 Description

This system call is used to set the value of a process trigger to any positive value.

The caller can only set the trigger value of its own trigger.

### 3.84.2 Syntax

```
void sc_triggerValueSet (
   sc_triggerval_t         value
);
```

### 3.84.3 Parameter

| | |
|---|---|
| **value** | Trigger value. |
| | The new trigger value to be stored. |

### 3.84.4 Return Value

None.

### 3.84.5 Example

```
sc_triggerValueSet (1);
sc_triggerWait (1, SC_ENDLESS_TMO);
```

### 3.84.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Illegal trigger value. | e0 = Trigger value |

## 3.85 sc_triggerWait

### 3.85.1 Description

This system call is used to wait on the process trigger.

The sc_triggerWait call will wait on the trigger of the callers process. The trigger value will be decremented by the value **dec** of the parameters.

If the trigger value becomes negative or equal zero, the calling process will be suspended and swapped out. The process will become ready again if the trigger value becomes positive. This occurs if another process has activated the trigger a sufficient number of times or with a sufficient trigger value.

The caller can also specify a timeout value **tmo**. The caller will not wait longer than the specified time for the trigger. If the timeout expires the process will be swapped in again.

The activation time is saved for **sc_triggerWait** in prioritized processes. The activation time is the absolute time (tick counter) value when the process became ready.

### 3.85.2 Syntax

```
int sc_triggerWait (
   sc_triggerval_t    dec,
   sc_ticks_t         tmo
);
```

### 3.85.3 Parameter

| **dec** | Decrease value. |
|---|---|
| | The number to decrease the process trigger value. |
| **tmo** | Timeout. |
| SC_ENDLESS_TMO | Timeout is not used. Blocks and waits endless until trigger. |
| SC_NO_TMO | Generates a system error. |
| SC_FATAL_IF_TMO | Generates a system error. |
| 0 < tmo =< SC_TMO_MAX | Timeout value in system ticks. Waiting on trigger with timeout. Blocks and waits the specified number of ticks for trigger. |

### 3.85.4 Return Value

SC_TRIGGER_TRIGGERED if the trigger occurred.

SC_TRIGGER_NO_WAIT if the process did not swap out

SC_TRIGGER_TMO if a timeout occurred

SC_TRIGGER_WAKEUP if the kernel will wakeup a timer or interrupt process.

### 3.85.5 Example

```
sc_triggerValueSet (1);
sc_triggerWait (1, SC_ENDLESS_TMO);
```

### 3.85.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Interrupts and/or scheduler are/is locked. | e0 = Lock counter value or<br>-1 if interrupt are locked. |
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Illegal trigger decrement value (<=0). | e0 = Decrement value |
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>tmo value not valid. | e0 = tmo value |

# 4 Kernel Error Codes

## 4.1 Introduction

SCIOPTA has many built-in error check functions.

Contrary to most conventional real-time operating systems, SCIOPTA uses a centralized mechanism for error reporting, called Error Hook. In traditional real-time operating systems, the user needs to check return values of system calls for a possible error condition. In SCIOPTA all error conditions will end up in the Error Hook. This guarantees that all errors are treated and that the error handling does not depend on individual error strategies which might vary from user to user.

Please consult the SCIOPTA - Kernel V2, User's Manual for detailed information about the SCIOPTA error handling.

When the error hook is called from the kernel, all information about the error are transferred in 32-bit error word parameter. There are also additional 32-bit extra words (parameters **e0, e1, e2, ...**) available to the user.

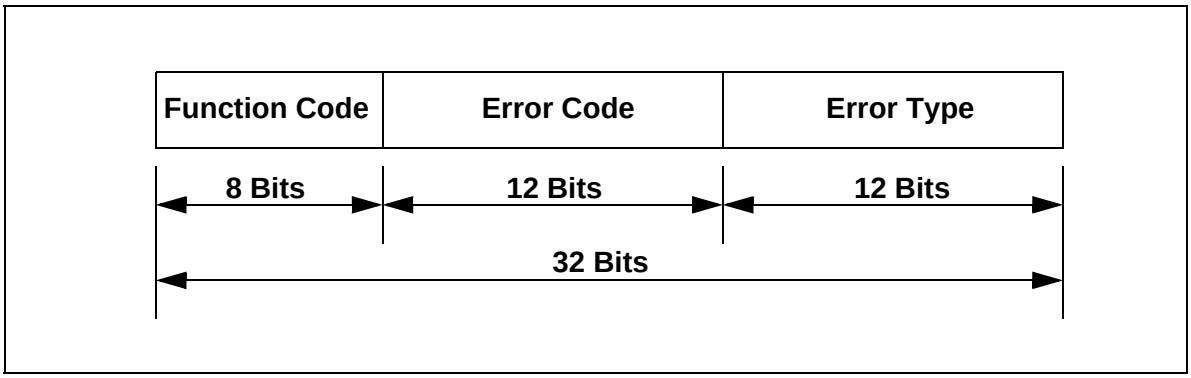


**Figure 4-1: 32-bit Error Word**

The **Function Code** defines from what SCIOPTA system call the error was initiated. The **Error Code** contains the specific error information and the **Error Type** informs about the severeness of the error.

## 4.2 Include Files

The error codes are defined in the **err.h** include file.
File location: <install_folder>\sciopta\<version>\include\kernel\

The error descriptions are defined in the **errtxt.h** include file.
File location: <install_folder>\sciopta\<version>\include\ossys\

## 4.3 Function Codes

| Name | Number | Error Source |
|---|---|---|
| SC_MSGALLOC | 0x01 | sc_msgAlloc |
| SC_MSGFREE | 0x02 | sc_msgFree |
| SC_MSGADDRGET | 0x03 | sc_msgAddrGet |
| SC_MSGSNDGET | 0x04 | sc_msgSndGet |
| SC_MSGSIZEGET | 0x05 | sc_msgOwnerGet |
| SC_MSGSIZESET | 0x06 | sc_msgSizeGet |
| SC_MSGOWNERGET | 0x07 | sc_msgSizeSet |
| SC_MSGTX | 0x08 | sc_msgTx |
| SC_MSGTXALIAS | 0x09 | sc_msgTxAlias |
| SC_MSGRX | 0x0A | sc_msgRx |
| SC_MSGPOOLIDGET | 0x0B | sc_poolIdGet |
| SC_MSGACQUIRE | 0x0C | sc_msgAcquire |
| SC_MSGALLOCCLR | 0x0D | sc_msgAllocClr |
| SC_MSGHOOKREGISTER | 0x0E | sc_msgHookRegister |
| SC_MSGHDCHECK | 0x0F | sc_msgHdCheck |
| SC_TMOADD | 0x10 | sc_tmoAdd |
| SC_TMORM | 0x11 | sc_tmoRm |
| SC_MSGDATACRCSET | 0x14 | sc_msgDataCrcSet, see Safety Manual |
| SC_MSGDATACRCGET | 0x15 | sc_msgDataCrcGet, see Safety Manual |
| SC_MSGDATACRCDIS | 0x16 | sc_msgDataCrcDis, see Safety Manual |
| SC_MSGFLOWSIGNATUREUPDATE | 0x17 | sc_msgFlowSignatureUpdate, see Safety Manual |
| SC_POOLCREATE | 0x18 | sc_poolCreate |
| SC_POOLRESET | 0x19 | sc_poolReset |
| SC_POOLKILL | 0x1A | sc_poolKill |
| SC_POOLINFO | 0x1B | sc_poolInfo |
| SC_POOLDEFAULT | 0x1C | sc_poolInfo |
| SC_POOLIDGET | 0x1D | sc_procIdGet |
| SC_POOLHOOKREGISTER | 0x1E | sc_poolHookRegister |
| SC_POOLCBCHK | 0x1F | sc_poolCbChk, see Safety Manual |
| SC_MISCERRORHOOKREGISTER | 0x20 | sc_miscErrorHookRegister |
| SC_MISCKERNELDREGISTER | 0x21 | sc_miscKerneldRegister, internel function |
| SC_MISCCRCCONTD | 0x22 | sc_miscCrcContd |
| SC_MISCCRC | 0x23 | sc_miscCrc |
| SC_MISCCRC32CONTD | 0x24 | sc_miscCrc32Contd |
| SC_MISCCRC32 | 0x25 | sc_miscCrc32 |
| SC_MISCERRNOSET | 0x26 | sc_miscErrnoSet |
| SC_MISCERRNOGET | 0x27 | sc_miscErrnoGet |
| SC_MISCERROR | 0x28 | sc_miscError |

| Name | Number | Error Source |
|---|---|---|
| SC_MISCFLOWSIGNATUREINIT | 0x2D | sc_miscFlowSignatureInit, see Safety Manual |
| SC_MISCFLOWSIGNATUREUPDATE | 0x2E | sc_miscFlowSignatureUpdate, see Safety Manual |
| SC_MISCFLOWSIGNATUREGET | 0x2F | sc_miscFlowSignatureGet, see Safety Manual |
| SC_PROCWAKEUPENABLE | 0x30 | sc_procWakeupEnable |
| SC_PROCWAKEUPDISABLE | 0x31 | sc_procWakeupDisable |
| SC_PROCPRIOGET | 0x32 | sc_procPrioGet |
| SC_PROCPRIOSET | 0x33 | sc_procPrioSet |
| SC_PROCSLICEGET | 0x34 | sc_procSliceGet |
| SC_PROCSLICESET | 0x35 | sc_procSliceSet |
| SC_PROCIDGET | 0x36 | sc_procIdGet |
| SC_PROCPPIDGET | 0x37 | sc_procPpidGet |
| SC_PROCNAMEGET | 0x38 | sc_procNameGet |
| SC_PROCPATHGET | 0x39 | sc_procPathGet |
| SC_PROCATTRGET | 0x3A | sc_procAttrGet |
| SC_PROCVECTORGET | 0x3B | sc_procVectorGet |
| SC_PROCPATHCHECK | 0x3C | sc_procPathGet |
| SC_PROCSTART | 0x40 | sc_procStart |
| SC_PROCSTOP | 0x41 | sc_procStop |
| SC_PROCSCHEDLOCK | 0x42 | sc_procSchedLock |
| SC_PROCSCHEDUNLOCK | 0x43 | sc_procSchedUnlock |
| SC_PROCYIELD | 0x44 | sc_procYield |
| SC_PROCCREATE2 | 0x45 | sc_procCreate2 |
| SC_PROCKILL | 0x46 | sc_procKill |
| SC_PROCOBSERVE | 0x47 | sc_procObserve |
| SC_PROCUNOBSERVE | 0x48 | sc_procUnobserve |
| SC_PROCVARINIT | 0x49 | sc_procVarInit |
| SC_PROCVARGET | 0x4A | sc_procVarGet |
| SC_PROCVARSET | 0x4B | sc_procVarSet |
| SC_PROCVARDEL | 0x4C | sc_procVarDel |
| SC_PROCVARRM | 0x4D | sc_procVarRm |
| SC_PROCATEXIT | 0x4E | sc_procAtExit |
| SC_PROCHOOKREGISTER | 0x4F | sc_procHookRegister |
| SC_PROCDAEMONREGISTER | 0x50 | sc_procDaemonRegister |
| SC_PROCDAEMONUNREGISTER | 0x51 | sc_procDaemonUnregister |
| SC_PROCCBCHK | 0x5C | sc_procCbChk, see Safety Manual |
| SC_PROCFLOWSIGNATUREINIT | 0x5D | sc_procFlowSignatureInit, see Safety Manual |
| SC_PROCFLOWSIGNATUREUPDATE | 0x5E | sc_procFlowSignatureUpdate, see Safety Manual |
| SC_PROCFLOWSIGNATUREGET | 0x5F | sc_procFlowSignatureGet, see Safety Manual |
| SC_MODULECREATE2 | 0x60 | sc_moduleCreate2 |
| SC_MODULEKILL | 0x61 | sc_moduleKill |

| Name | Number | Error Source |
|---|---|---|
| SC_MODULENAMEGET | 0x62 | sc_moduleNameGet |
| SC_MODULEIDGET | 0x63 | sc_moduleIdGet |
| SC_MODULEINFO | 0x64 | sc_moduleInfo |
| SC_MODULEPRIOGET | 0x65 | sc_modulePrioGet |
| SC_MODULESTOP | 0x6C | sc_moduleStop |
| SC_MODULECBCHK | 0x6F | sc_moduleCbChk, see Safety Manual |
| SC_TRIGGERVALUESET | 0x70 | sc_triggerValueSet |
| SC_TRIGGERVALUEGET | 0x71 | sc_triggerValueGet |
| SC_TRIGGER | 0x72 | sc_trigger |
| SC_TRIGGERWAIT | 0x73 | sc_triggerWait |
| SC_TICKACTIVATIONGET | 0x78 | sc_tickActivationGet |
| SC_TICK | 0x79 | sc_tick |
| SC_TICKGET | 0x7A | sc_tickGet |
| SC_TICKLENGTH | 0x7B | sc_tickLength |
| SC_TICKMS2TICK | 0x7C | sc_tickMs2Tick |
| SC_TICKTICK2MS | 0x7D | sc_tickTick2Ms |
| SC_SLEEP | 0x7E | sc_sleep |
| SC_CONNECTORREGISTER | 0x80 | sc_connectorRegister |
| SC_CONNECTORUNREGISTER | 0x81 | sc_connectorUnregister |
| SC_DISPATCHER | 0x88 | sc_sysProcSwapper |
| SC_IRQDISPATCHER | 0x89 | sc_sysIrqDispatcher |
| SC_SYSERROR | 0x8B | many |
| SC_SYSLDATACORRUPT | 0x8D | many |

## 4.4 Error Codes

| Name | Number | Description |
|---|---|---|
| KERNEL_EILL_POOL_ID | 0x001 | Illegal pool ID. |
| KERNEL_ENO_MOORE_POOL | 0x002 | No more pool. |
| KERNEL_EILL_POOL_SIZE | 0x003 | Illegal pool size. |
| KERNEL_EPOOL_IN_USE | 0x004 | Pool still in use. |
| KERNEL_EILL_NUM_SIZES | 0x005 | Illegal number of buffer sizes. |
| KERNEL_EILL_BUF_SIZES | 0x006 | Illegal buffersizes. |
| KERNEL_EILL_BUFSIZE | 0x007 | Illegal buffersize. |
| KERNEL_EOUTSIDE_POOL | 0x008 | Message outside pool. |
| KERNEL_EMSG_HD_CORRUPT | 0x009 | Message header corrupted. |
| KERNEL_ENIL_PTR | 0x00A | NIL pointer. |
| KERNEL_EENLARGE_MSG | 0x00B | Message enlarged. |
| KERNEL_ENOT_OWNER | 0x00C | Not owner of the message. |
| KERNEL_EOUT_OF_MEMORY | 0x00D | Out of memory. |
| KERNEL_EILL_VECTOR | 0x00E | Illegal interrupt vector. |
| KERNEL_EILL_SLICE | 0x00F | Illegal time slice. |
| KERNEL_ENO_KERNELD | 0x010 | No kernel daemon started. |
| KERNEL_EMSG_ENDMARK_CORRUPT | 0x011 | Message endmark corrupted. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT | 0x012 | Previous message's endmark corrupted. |
| KERNEL_EILL_DEFPOOL_ID | 0x013 | Illegal default pool ID. |
| KERNEL_ELOCKED | 0x014 | Illegal system call while scheduler locked. |
| KERNEL_EILL_PROCTYPE | 0x015 | Illegal process type. |
| KERNEL_EILL_INTERRUPT | 0x016 | Illegal interrupt. |
| KERNEL_EILL_EXCEPTION | 0x017 | Illegal unhandled exception. |
| KERNEL_EILL_SYSCALL | 0x018 | Illegal syscall number. |
| KERNEL_EILL_NESTING | 0x019 | Illegal interrupt nesting. |
| KERNEL_EUNLOCK_WO_LOCK | 0x01F | Unlock without lock. |
| KERNEL_EILL_PID | 0x020 | Illegal process ID. |
| KERNEL_ENO_MORE_PROC | 0x021 | No more processes. |
| KERNEL_EMODULE_TOO_SMALL | 0x022 | Module size too small. |
| KERNEL_ESTART_NOT_STOPPED | 0x023 | Starting of a not stopped process. |
| KERNEL_EILL_PROC | 0x024 | Illegal process. |
| KERNEL_EILL_NAME | 0x025 | Illegal name. |
| KERNEL_EILL_TARGET_NAME | 0x025 | Illegal target name. |
| KERNEL_EILL_MODULE_NAME | 0x025 | Illegal module name. |
| KERNEL_EILL_MODULE | 0x027 | Illegal module ID. |
| KERNEL_EILL_PRIORITY | 0x028 | Illegal priority. |
| KERNEL_EILL_STACKSIZE | 0x029 | Illegal stacksize. |
| KERNEL_ENO_MORE_MODULE | 0x02A | No more modules available. |

| Name | Number | Description |
|---|---|---|
| KERNEL_EILL_PARAMETER | 0x02B | Illegal parameter. |
| KERNEL_EILL_PROC_NAME | 0x02C | Illegal process name. |
| KERNEL_EPROC_NOT_PRIO | 0x02D | Not a prioritized process. |
| KERNEL_ESTACK_OVERFLOW | 0x02E | Stack overflow. |
| KERNEL_ESTACK_UNDERFLOW | 0x02F | Stack underflow. |
| KERNEL_EILL_VALUE | 0x030 | Illegal value. |
| KERNEL_EALREADY_DEFINED | 0x031 | Already defined. |
| KERNEL_ENO_MORE_CONNECTOR | 0x032 | No more connectors available. |
| KERNEL_EMODULE_OVERLAP | 0x033 | Module memory overlaps. |
| KERNEL_EPROC_TERMINATE | 0xFFF | Process terminated. |

## 4.5 Error Types

| Name | Bit | Description |
|---|---|---|
| SC_ERR_SYSTEM_FATAL | 0x01 | This type of error will stop the whole target. |
| SC_ERR_MODULE_FATAL | 0x02 | This type of error results in killing the module if an error hook returns a value of !=0. |
| SC_ERR_PROCESS_FATAL | 0x04 | This type of error results in killing the process if an error hook returns a value of !=0. |
| SC_ERR_SYSTEM_WARNING | 0x10 | Warning on target level.The system continues if an error hook is installed. |
| SC_ERR_MODULE_WARNING | 0x20 | Warning on module level. The system continues if an error hook is installed. |
| SC_ERR_PROC_WARNING | 0x40 | Warning on process level. The system continues if an error hook is installed. |

# 5 Manual Versions

## 5.1 Manual Version 4.2

- Chapter 1.3.2 CPU Families, Renesas RX600 added.

## 5.2 Manual Version 4.1

- Chapter 1 The SCIOPTA System, Rewritten.
- Chapter 3.19 sc_msgAcquire, parameter description corrected.
- Chapter 3.29 sc_msgRx, last paragraph in 3.29.1 Description, added.
- Chapter 3.33.3 Parameter, flags parameter description corrected.
- Chapter 3.44.3 Parameter, pdb parameter description corrected.
- Chapter 3.46 sc_procDaemonUnregister, return value added.
- Chapter 3.52 sc_procPathCheck, system call now documented.
- Chapter 3.53.3 Parameter, description <pid> corrected.
- Chapter 3.55.4 Return Value, last line added.
- Chapter 3.66.3 Parameter, varpool parameter values 0 and SC_NIL added. Parameter n description corrected.
- Chapter 3.69 sc_procVectorGet, can now be called from any process to get interrupt vector of an interrupt process defined by parameter pid.
- Chapter 3.73 sc_sleep, return value (activation time) added.
- Chapter 3.75 sc_tickActivationGet, return value better described.
- Chapter 3.80.6 Errors, KERNEL_ENOT_OWNER error: extra parameters corrected.
- Chapter 3.81.6 Errors, KERNEL_ENOT_OWNER error: extra parameter e0 corrected.
- Chapter 3.85 sc_triggerWait, last paragraph in 3.84.1 Description, added.

## 5.3 Manual Version 4.0

- Support for SCIOPTA Version 2 Kernel.
- SCIOPTA Manual system restructured and rewritten.
- Splitted into a Kernel Reference Manual and User's Manual.
- No more target specific manuals. Target specific information are included in the Kernel Reference Manual and User's Manual.

## 5.4 Manual Version 3.2

- Chapter 2.4.9 GNU Tool Chain Installation, rewritten due to CodeSourcery GCC tool chain support.
- Chapter 3.2.1 Class Diagram, replaces old block diagram.
- Chapter 3.2.3.1 Equipment, CodeSourcery GCC tool chain now used.
- Chapter 6.6.4 sc_procKill, "The sc_procKill system calls returns before the cleaning work begins." removed from the last paragraph.

- Chapter 13.2.1 Tools, CodeSourcery GCC tool chain now used.
- Chapter 13.3.2 Tools, CodeSourcery GCC tool chain now used.
- Chapter 13.4.2 Tools, CodeSourcery GCC tool chain now used.

## 5.5 Manual Version 3.1

- Chapter 2.4.10 Eclipse C/C++ Development Tooling - CDT, rewritten.
- Chapter 3 Getting Started, rewritten.
- Chapter 4.15.2 Eclipse, rewritten.
- Chapter 5.2.5 Priorities, upper limit effective priority of 31 described.
- Chapter 6, Application Programming Interface,
- Chapter 6.3.6 sc_msgRx and chapter 6.3.1 sc_msgAlloc, parameter tmo, value modified.
- Chapter 6.4.1 sc_poolCreate, parameter size, pool calculation value n better defined. Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.4.3 sc_poolIdGet, return value “poolID of default pool” added.
- Chapter 6.5.6 sc_procPrioSet, paragraph: “If there are already existing processes at the same priority, the process which has just moved to that priority will be put at the end of the list and swapped-out.” added.
- Chapter 6.6.1 sc_procPrioCreate, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.6.2 sc_procIntCreate, “User” interrupt process type removed, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.6.3 sc_procTimCreate, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.10.1 sc_moduleIdGet, parameter name text: “...or zero for current module” appended at description and return value for parameter name=NULL added. Return value SC_NOSUCH_MODULE modified to SC_ILLEGAL_MID if Module name was not found.
- Chapter 6.11.1 sc_moduleCreate, parameter textsize renamed into initsize. Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.12.3 sc_moduleFriendGet, return value “1” modified to “!=”.
- Chapter 6.13.2 sc_tick, last paragraph added.
- Chapter 6.15.2 sc_triggerWait, parameter tmo better described.
- Chapter 6.19.1 sc_miscErrorHookRegister, function and parameter newhook are of type pointer. Global and Module error hook difference described.
- Chapter 6.19.2 sc_msgHookRegister, function and parameter newhook are of type pointer.
- Chapter 6.19.3 sc_poolHookRegister, function and parameter newhook are of type pointer. Global and Module pool hook difference described.
- Chapter 6.19.4 sc_procHookRegister, function and parameter newhook are of type pointer. Global and Module process hook difference described.
- Chapter 7.7.2.7 Example, system call sc_procIdGet parameter corrected.
- Chapter 7.11.4 Error Hook Declaration Syntax, return value “!= 0” modified.
- Chapter 7.11.6 Error Hooks Return Behaviour, added.

- Chapter 9.10 i.MX27 System Functions and Drivers, added.
- Chapter 9.22 LOGIC i.MX27 LITEKIT, added.
- Chapter 13.3 Eclipse IDE and GNU GCC, rewritten.
- Chapter 15.3 Function Codes, code 0x0E, 0x0D, 0x3E, 0x3F, 0x57, 0x5C, 0x5D, 0x5E and 0x5F added.
- Chapter 15.4 Error Codes, code 0x016, 0x017 and 0x018 added.
- Chapter 16.9.3 Debug Configuration, Stack Check added to the list.

## 5.6 Manual Version 3.0

- Manual restructured and rewritten.

## 5.7 Manual Version 2.1

- Chapter 2.4 Installation Procedure Windows Hosts, now modified for customer specific deliveries.
- Chapter 2.4.5 SCIOPTA_HOME Environment Variable, UNIX Shell versions removed.

## 5.8 Manual Version 2.0

- The following manuals have been combinded in this new SCIOPTA ARM - Kernel, User's Guide:
    - SCIOPTA - Kernel, User's Guide Version 1.8
    - SCIOPTA - Kernel, Reference Manual Version 1.7
    - SCIOPTA - ARM Target Manual

## 5.9 Former SCIOPTA - Kernel, User's Guide Versions

### 5.9.1 Manual Version 1.8

- Back front page, Litronic AG became SCIOPTA Systems AG.
- Chapter 2.3.4.1 Prioritized Process, icon now correct.
- Chapter 2.3.4.2 Interrupt Process, last paragrpah added.
- Chapter 2.3.4.5 Supervisor Process, rewritten.
- Chapter 2.5.2 System Module, rewritten.
- Chapter 4.11.1 Start Sequence, added.
- Chapter 4.11.3 C Startup, rewritten.
- Chapter 4.11.5 INIT Process, added.
- Chapter 4.11.6 Module Start Function, added.
- Chapter 4.7.2.6 Example, in system call msg_alloc SC_ENDLESS_TMO replaced by SC_DEFAULT_POOL.

### 5.9.2 Manual Version 1.7

- Chapter 3.9.1 Configuring ARM Target Systems, Inter-Module settings added.
- Chapter 3.9.2 Configuring Coldfire Target Systems, Inter-Module settings added.
- Chapter 3.9.3 Configuring PowerPC Target Systems, Inter-Module settings added.

### 5.9.3 Manual Version 1.6

- Configuration chapter added (moved from the target manuals).

### 5.9.4 Manual Version 1.5

- All **union sc_msg *** changed to **sc_msg_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **union sc_msg **** changed to **sc_msgptr_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- Manual now splitted into a User's Guide and Reference Manual.

### 5.9.5 Manual Version 1.4

- Chapter 4.7.3.2 Example, OS_INT_PROCESS changed into correct SC_INT_PROCESS.
- Chapter 2.3.4.4 Init Process, rewritten.
- Chapter 4.5 Processes, former chapters **4.5.6 Idle Proces**s and **4.5.7 Supervisor Process** removed.
- Chapter 4.5.1 Introduction, last paragraph about supervisor processes added.
- Chapter 4.5.5 Init Process, rewritten.
- Chapter 6.8 sc_miscErrorHookRegister, syntax corrected.
- Chapter 6.21 sc_mscAlloc, timeout parameter **tmo** better specified.
- Chapter 6.27 sc_msgRx, timeout parameter **tmo** better specified.
- Chapter 4.10.4 Error Hook Declaration Syntax, user !=**0** user error.
- Chapter 4.9 SCIOPTA Daemons, moved from chapter 2.9 and rewritten.
- Chapter 6.41 sc_procDaemonRegister, last paragraph of the description rewritten.
- Chapters 6.45 sc_procIntCreate, 6.46 sc_procKill, 6.51 sc_procPrioCreate, 6.60 sc_procTimCreate and 6.62 sc_procUsrIntCreate, information about **sc_kerneld** are given.
- Chapter 4.10.5 Example, added.

### 5.9.6 Manual Version 1.3

- Chapter 6.26 sc_msgPoolIdGet, return value **SC_DEFAULT_POOL** defined.
- Chapter 6.33 sc_poolCreate, pool size formula added.
- Chapter 2.4.4 Message Pool, maximum number of pools for compact kernel added.
- Chapter 4.8 SCIOPTA Memory Manager - Message Pools, added.
- Chapter 6.9 sc_moduleCreate, modul size calculation modified.

- Chapter 6.40 sc_procCreate, 6.45 sc_procIntCreate, 6.51 sc_procPrioCreate and 6.60 sc_procTim Create, stacksize calculation modified.

### 5.9.7 Manual Version 1.2

- Top cover back side: Address of SCIOPTA France added.
- Chapter 2.6 Trigger, second paragraph: At process creation the value of the trigger is initialized to **one**.
- Chapter 2.6 Trigger, third paragraph: The **sc_triggerWait**() call decrements the value of the trigger and the calling process will be blocked and swapped out if the value gets negative **or equal zero**.
- Chapter 2.7 Process Variables, second paragraph: The tag and the process variable have a fixed size **large enough to hold a pointer**.
- Chapter 2.7 Process Variables, third paragraph: Last sentence rewritten.
- Chapter 4.5.3.1 Interrupt Process Declaration Syntax, irg_src is of type **int** added.
- Chapter 4.5.6 Idle Process, added.
- Chapter 4.10.4 Error Hook Declaration Syntax, Parameter **user** : user != 0 (User error).
- System call **sc_procRegisterDaemon** changed to **sc_DaemonRegister** and **sc_procUnregisterDaemon** changed to **sc_procDaemonUnregister**.
- System call sc_miscErrorHookRegister, return values better specified.
- System call sc_moduleCreate, parameter **size** value “code” added in Formula.
- System call sc_moduleNameGet, return value **NULL** added.
- System call sc_msgAcquire, condition modified.
- System Call sc_msgAlloc, **SC_DEFAULT_POOL** better specified.
- Systme Call sc_msgHookRegister, description modified and return value better specified.
- System call sc_msgRx, parameters better specified.
- System call sc_poolHookRegister, return value better specified.
- System call sc_procHookRegister, return value better specified.
- System call sc_procIdGet, last paragraph in **Description** added.
- System calls sc_procVarDel, sc_procVarGet and procVarSet, return value **!=0** introduced.
- Chapter 7.3 Function Codes, errors **0x38** to **0x3d** added.
- System call **sc_procUnobserve** added.
- Chapters 2.5.2 System Module and 4.3 Modules, the following sentence was removed: The system module runs always on supervisor level and has all access rights.
- Chapter 2.5.3 Messages and Modules, third paragraph rewritten.
- Chapter 6.31 sc_msgTx, fifth paragraph rewritten.

### 5.9.8 Manual Version 1.1

- System call **sc_moduleInfo** has now a return parameter.
- New system call **sc_procPathGet**.

- System call **sc_moduleCreate** formula to calculate the size of the module (parameter **size**) added.
- Chapter 4.12 SCIOPTA Design Rules, moved at the end of chapter **"System Design"**.
- New chapter 4.6 Addressing Processes.
- Chapter 7 Kernel Error Codes, new sequence of sub chapters. Smaller font used.
- Chapter 4.10 Error Hook, completely rewritten.
- New chapter 4.11 System Start.

### 5.9.9 Manual Version 1.0

Initial version.

## 5.10 Former SCIOPTA - Kernel, Reference Manual Versions

### 5.10.1 Manual Version 1.7

- Back front page, Litronic AG became SCIOPTA Systems AG.
- Chapter 3.24 sc_msgHookRegister, text: There can be one module message hook per module replaced by: There can be one module message hook of each type (transmitt/receive) per module.
- Chapter 3.27 sc_msgRx, **flag** parameter **SC_MSGRX_NOT** : text: An array of messages is given which will be excluded from receive replaced by: An array of message ID's is given which will be excluded from receive.
- Chapter 3.47 sc_procNameGet and chapter 3.49 sc_procPathGet, chapter "**Return Value**" rewritten.

### 5.10.2 Manual Version 1.6

- Chapter 3.27 sc_msgRx, **tmo** parameter, SC_TMO_NONE replaced by SC_NO_TMO. Parameter better specified.
- Chapter 3.27 sc_msgRx, **wanted** parameter, NULL replaced by SC_MSGRX_ALL.
- Chapter 3.7 sc_miscError, **err** parameter, bits 0, 1 and 2 documented.
- Chapter 3.44 sc_procIdGet, if paramter **path** is NULL and parameter **tmo** is SC_NO_TMO this system call returns the callers process ID.

### 5.10.3 Manual Version 1.5

- All **union sc_msg *** changed to **sc_msg_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **union sc_msg **** changed to **sc_msgptr_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- Chapter 6, System Call Reference, page layout for all system calls modified.
- Chapter 6.81 sc_triggerWait, third paragraph rewritten.
- Chapters 6.9 sc_moduleCreate and 6.17 sc_moduleKill, information about **sc_kerneld** are given.
- Chapter 6.44 sc_procIdGet, added text: this parameter is not allowed if **asynchronous timeout** is disabled at system configuration (**sconf**).
- Manual split into a User's Guide and a Reference Manual.

### 5.10.4 Manual Version 1.4

- Chapter 4.7.3.2 Example, OS_INT_PROCESS changed into correct SC_INT_PROCESS.
- Chapter 2.3.4.4 Init Process, rewritten.
- Chapter 4.5 Processes, former chapters **4.5.6 Idle Process** and **4.5.7 Supervisor Process** removed.
- Chapter 4.5.1 Introduction, last paragraph about supervisor processes added.
- Chapter 4.5.5 Init Process, rewritten.
- Chapter 6.8 sc_miscErrorHookRegister, syntax corrected.
- Chapter 6.21 sc_mscAlloc, timeout parameter **tmo** better specified.
- Chapter 6.27 sc_msgRx, timeout parameter **tmo** better specified.
- Chapter 4.10.4 Error Hook Declaration Syntax, user !=**0** user error.
- Chapter 4.9 SCIOPTA Daemons, moved from chapter 2.9 and rewritten.
- Chapter 6.41 sc_procDaemonRegister, last paragraph of the description rewritten.
- Chapters 6.45 sc_procIntCreate, 6.46 sc_procKill, 6.51 sc_procPrioCreate, 6.60 sc_procTimCreate and 6.62 sc_procUsrIntCreate, information about **sc_kerneld** are given.
- Chapter 4.10.5 Example, added.

### 5.10.5 Manual Version 1.3

- Chapter 6.26 sc_msgPoolIdGet, return value **SC_DEFAULT_POOL** defined.
- Chapter 6.33 sc_poolCreate, pool size formula added.
- Chapter 2.4.4 Message Pool, maximum number of pools for compact kernel added.
- Chapter 4.8 SCIOPTA Memory Manager - Message Pools, added.
- Chapter 6.9 sc_moduleCreate, modul size calculation modified.
- Chapter 6.40 sc_procCreate, 6.45 sc_procIntCreate, 6.51 sc_procPrioCreate and 6.60 sc_procTim Create, stacksize calculation modified.

### 5.10.6 Manual Version 1.2

- Top cover back side: Address of SCIOPTA France added.
- Chapter 2.6 Trigger, second paragraph: At process creation the value of the trigger is initialized to **one**.
- Chapter 2.6 Trigger, third paragraph: The **sc_triggerWait**() call decrements the value of the trigger and the calling process will be blocked and swapped out if the value gets negative **or equal zero**.
- Chapter 2.7 Process Variables, second paragraph: The tag and the process variable have a fixed size **large enough to hold a pointer**.
- Chapter 2.7 Process Variables, third paragraph: Last sentence rewritten.
- Chapter 4.5.3.1 Interrupt Process Declaration Syntax, irg_src is of type **int** added.
- Chapter 4.5.6 Idle Process, added.
- Chapter 4.10.4 Error Hook Declaration Syntax, Parameter **user** : user != 0 (User error).

- System call **sc_procRegisterDaemon** changed to **sc_DaemonRegister** and **sc_procUnregisterDaemon** changed to **sc_procDaemonUnregister**.
- System call sc_miscErrorHookRegister, return values better specified.
- System call sc_moduleCreate, parameter **size** value “code” added in Formula.
- System call sc_moduleNameGet, return value **NULL** added.
- System call sc_msgAcquire, condition modified.
- System Call sc_msgAlloc, **SC_DEFAULT_POOL** better specified.
- Systme Call sc_msgHookRegister, description modified and return value better specified.
- System call sc_msgRx, parameters better specified.
- System call sc_poolHookRegister, return value better specified.
- System call sc_procHookRegister, return value better specified.
- System call sc_procIdGet, last paragraph in **Description** added.
- System calls sc_procVarDel, sc_procVarGet and procVarSet, return value **!=0** introduced.
- Chapter 7.3 Function Codes, errors **0x38** to **0x3d** added.
- System call **sc_procUnobserve** added.
- Chapters 2.5.2 System Module and 4.3 Modules, the following sentence was removed: The system module runs always on supervisor level and has all access rights.
- Chapter 2.5.3 Messages and Modules, third paragraph rewritten.
- Chapter 6.31 sc_msgTx, fifth paragraph rewritten.

### 5.10.7 Manual Version 1.1

- System call **sc_moduleInfo** has now a return parameter.
- New system call **sc_procPathGet**.
- System call **sc_moduleCreate** formula to calculate the size of the module (parameter **size**) added.
- Chapter 4.12 SCIOPTA Design Rules, moved at the end of chapter **“System Design”**.
- New chapter 4.6 Addressing Processes.
- Chapter 7 Kernel Error Codes, new sequence of sub chapters. Smaller font used.
- Chapter 4.10 Error Hook, completely rewritten.
- New chapter 4.11 System Start.

### 5.10.8 Manual Version 1.0

Initial version.

## 5.11 Former SCIOPTA ARM - Target Manual Versions

### 5.11.1 Manual Version 2.2

- Back front page, Litronic AG became SCIOPTA Systems AG.
- Chapter 2.2 The SCIOPTA ARM Delivery and chapter 2.4.1 Main Installation Window, tiny kernel added.
- Chapter 3 Getting Started, in the example folder, additional directories for boards have been introduced.
- Chapter 3 Getting Started, the Eclipse project files and the file **copy_files.bat** are now stored in the "\phyCore2294" board sub-directory of the example folder.
- Chapter 3 Getting Started, the SCIOPTA SCONF configuration file is now called **hello.xml** (was hello_phyCore2294.xml before).
- Chapter 5.8.3 Assembling with IAR Systems Embedded Workbench, added.
- Chapter 5.10.3 Compiling with IAR Systems Embedded Workbench, added.
- Chapter 5.12.3 Linking with IAR Systems Embedded Workbench, added.
- Chapter 5.13.1.1 Memory Regions, last paragraph added.
- Chapter 5.13.1.2 Module Sizes, name is now **<module_name>_size** (was <module_name>_free before).
- Chapter 5.13.3 IAR Systems Embedded Workbench Linker Script, added.
- Chapter 5.14 Data Memory Map, redesigned and now one memory map for all environments.
- Chapter 5.14.4 IAR Systems Embedded Workbench©, added.
- Chapter 6 Board Support Packages, file lists modified for SCIOPTA ARM version 1.7.2.5
- Chapter 6.3 ATMEL AT96SAM7S-EK Board, added.
- Chapter 6.4 ATMEL AT96SAM7X-EK Board, added.
- Chapter 6.5 IAR Systems STR711-SK Board, added.

### 5.11.2 Manual Version 2.1

- Chapter 1.1 About this Manual, SCIOPTA product list updated.
- Chapter 2.4.1 Main Installation Window, Third Party Products, new version for GNU Tool Chain (version 1.4) and MSys Build Shell (version 1.0.10).
- Chapter 2.4.7 GNU Tool Chain Installation, new GCC Installation version 1.4 including new gcc version 3.4.4, new binutils version 2.16.1 and new newlib version 1.13.1. The installer creates now two directories (and not three).
- Chapter 2.4.8 MSYS Build Shell, new version 1.0.10.
- Chapter 3, Getting Started: Equipment, new versions for GNU GCC and MSys.
- Chapter 3, Getting Started: List of copied files (after executed copy_files.bat) removed.
- Chapter 3.5.1 Description (Web Server), paragraph rewritten.
- Chapter 3.13.2.1 Equipment, serial cable connection correctly described.
- Chapter 3.13.2.2 Step-By-Step Tutorial, DRUID and DRUID server setup rewritten.
- Chapter 5.16 Integrated Development Environments, new chapter.

### 5.11.3 Manual Version 2.0

- Manual rewritten.
- Own manual version, moved to version 2.0

### 5.11.4 Manual Version 1.7.2

- Installation: all IPS Applications such as Web Server, TFTP etc. in one product.
- Getting started now for all products.
- Chapter 4, Configuration now moved into Kernel User's Guide.
- New BSP added: Phytec phyCORE-LPC2294.
- Uninstallation now separately for every SCIOPTA product.
- Eclipse included in the SCIOPTA delivery.
- New process SCP_proxy introduced in Getting Started - DHCP Client Example.
- IPS libraries now in three verisons (standard, small and full).

### 5.11.5 Manual Version 1.7.0

- All **union sc_msg *** changed to **sc_msg_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **union sc_msg **** changed to **sc_msgptr_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- 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.
- All **sdd_objInfo_t *** changed to **sdd_objInfo_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **ips_dev_t *** changed to **ips_dev_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **ipv4_arp_t *** changed to **ipv4_arp_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **ipv4_route_t *** changed to **ipv4_route_t NEARPTR** to support SCIOPTA 16 Bit systems.
- IAR support added in the kernel.
- Web server modifiied.
- TFTP server added (in addition to client).
- DHCP server added (in addition to client).
- DRUID System Level Debugger added.

# 6 Index

**F**



**G**

**R**



**S**

**T**



**U**



**W**



**Y**