SCIOPTA Kernel Reference Manual

v5.3

Contents

Abstract

This document is the **SCIOPTA Reference Manual** for the SCIOPTA Kernels.

Copyright

Disclaimer

Trademark

Contact

Corporate Headquarters
SCIOPTA Systems GmbH
Hauptstrasse 293
79576 Weil am Rhein
Germany
Tel. +49 7621 940 919 0
Fax +49 7621 940 919 19
email: sales@sciopta.com
www.sciopta.com

1 SCIOPTA Real-Time Operating System

1.1 Introduction

SCIOPTA is a high performance fully pre emptive real-time operating system for hard real-time application available for many target platforms.

Available modules:

- Pre-emptive Multitasking Real-Time Kernel
- SCIOPTA Memory Management System, Support for MMU/MPU
- Board Support Packages
- IPS Internet Protocols v4/v6 (TCP/IP) including IPS Applications (Web Server, TFTP, FTP, DNS, DHCP, Telnet and SMTP)
- FAT File System
- (fail) SAFE FAT File System
- Flash File System, NOR and NAND
- Universal Serial Bus, USB Device
- Universal Serial Bus, USB Host
- DRUID System Level Debugger including kernel awareness packages for source debuggers
- SCIOPTA PEG Embedded GUI
- CONNECTOR support for distributed multi CPU systems
- SCAPI SCIOPTA API for Windows or LINUX hosts or other OS
- SCSIM SCIOPTA Simulator

SCIOPTA Real-Time Operating System contains 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 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.

The SCIOPTA Real-Time Kernel has a very high performance. 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.

1.2 CPU Families

SCIOPTA is delivered for many CPU architectures such as the various Arm Ltd. families, RX (Renesas), Power architecture (NXP, STM), Blackfin (Analog Devices) and Aurix (Infineon).
Please consult the latest version of the SCIOPTA Price List for the complete list or ask our sales team if you are missing a specific architecture.

Initially mainly used in the automation and process control industry, IEC 61508 is more and more accepted for applications in other industries including automotive and medical where safety and reliability are paramount.

1.3 SCIOPTA Kernels

There are three Kernels (Technologies) within SCIOPTA: V1, V2 and V2INT. The V1 Kernels are written in 100% Assembler and are specifically tuned for the ARM Architectures. V2 Kernels are mostly written in "C" and available for many CPUs and Architectures. V2INT kernels have built-in integrity of RTOS data to be used in safety certified systems.

All three Kernels certified by TUeV Sued Munich to IEC61508 SIL 3, EN50128 SIL 3/4 and ISO26262 ASIL-D.

1.4 About this Manual

This SCIOPTA Reference Manual contains the reference of all system calls in alphabetical order.

1.5 SCIOPTA Architecture Manual

The SCIOPTA Architecture Manual contains a detailed description and introduction into SCIOPTA working concepts, structures and elements.

1.6 SCIOPTA Getting Start Manuals

The SCIOPTA Getting Start Manuals gives all needed information how to use SCIOPTA Real-Time Kernel in an embedded project for specific CPU Families.

1.7 SCIOPTA Kernel Configuration SCONF Manuals

The SCIOPTA kernel system needs to be configured before you can generate the whole system. The SCIOPTA configuration utility SCONF Manual gives all needed information and the parameters to be defined such as name of systems, static modules, processes, and pools, etc.

2 System Calls Overview

2.1 Introduction

This chapter list all SCIOPTA system calls in functional groups. Please consult chapter System Calls Reference for an alphabetical list.

Please Note:
There are three Kernel Technologies within SCIOPTA: V1, V2 and V2INT. The V1 Kernels are written in 100% Assembler and are specifically tuned for the ARM Architectures. V2 Kernels are mostly written in "C" and available for many CPUs and Architectures. V2INT kernels have built-in integrity of RTOS data to be used in safety certified systems.
If nothing is noted in the following lists below, the system call is valid for all three Kernel Technologies.

2.2 Prerequisites

The CPU must be in a privileged mode before calling the kernel and have access to all SCIOPTA relevant memory areas.

On ARMv4T,ARMv5T*,ARMv7-A/R or ARMv8-R this means *SYS* mode!

2.3 Message System Calls

2.3.1 Message Passing

- sc_msgAlloc(): Allocates a memory buffer of selectable size from a message pool
- sc_msgAllocClr(): Allocates a memory buffer of selectable size from a message pool and will initialize the data area of the message to 0.
- sc_msgAllocTx(): Allocates a message and sends it to the addressee.
- sc_msgTx(): Transmits a message to a process.
- sc_msgTxAlias(): Transmit a message to a process by setting a process ID as sender.
- sc_msgRx(): Receives messages.
- sc_msgFree(): Returns an allocated message to the message pool.

2.3.2 Message Information

- sc_msgFind(): Finds a message which has been allocated or already received.
- sc_msgAddrGet(): Gets the process ID of the addressee of a message.
- sc_msgSndGet(): Gets the process ID of the sender of a message.
- sc_msgOwnerGet(): Gets the process ID of the owner of a message.
- sc_msgPoolIdGet(): Gets the pool ID of a message.
- sc_msgSizeGet(): Gets the requested size of a message buffer.
- sc_msgHdCheck(): Checks the message header.

2.3.3 Message Modification

- sc_msgAcquire(): Changes the owner of a message.
- sc_msgSizeSet(): Decrease the requested size of a message buffer.

2.3.4 Message Safety

- sc_msgDataCrcDis(): Disables the message data CRC check for a message.
- sc_msgDataCrcGet(): Checks the message data checksum.
- sc_msgDataCrcSet(): Sets the message data checksum.
- sc_msgFlowSignatureUpdate(): Updates a global message flow signature.

2.3.5 Message Debugging

- sc_msgHookRegister(): Registers a message hook.

2.4 Process System Calls

2.4.1 Process Creation and Killing

- sc_procPrioCreate(): Creates a prioritized process in a V1 Kernel.
- sc_procIntCreate(): Creates an interrupt process in a V1 Kernel.
- sc_procTimCreate(): Creates a timer process in a V1 Kernel.
- sc_procCreate2(): Creates a process in a V2 and V2INT Kernel.
- sc_procKill(): Kills a process.

2.4.2 Process Controlling

- sc_procYield(): Yields the CPU to the next ready process.
- sc_procStart(): Starts a prioritized or timer process.
- sc_procStop(): Stopps a prioritized or timer process.
- sc_procSchedLock(): Locks the scheduler.
- sc_procSchedUnlock(): Unlocks the scheduler.
- sc_procIrqRegister(): Register an existing interrupt process for interrupt vectors.
- sc_procIrqUnregister(): Unregisters previously registered interrupts.
- sc_procSliceGet(): Gets the time slice of a prioritized or timer process.
- sc_procSliceSet(): Sets the time slice of a prioritized or timer process.
- sc_procWakeupEnable(): Enables the wakeup of a timer or interrupt process.
- sc_procWakeupDisable(): Disables the wakeup of a timer or interrupt process.
- sc_procPrioGet(): Gets the priority of a prioritized process.
- sc_procPrioSet(): Sets the priority of a process.
- sc_procDaemonRegister(): Registers a process daemon.
- sc_procDaemonUnregister(): Unregisters a process daemon.

2.4.3 Process Information

- sc_procIdGet(): Gets the process ID of a process by providing the name of the process.
- sc_procNameGet(): Gets the full name of a process.
- sc_procPpidGet(): Gets the process ID of the parent (creator) of a process.

- sc_procAttrGet(): Gets the specific attributes of a process.
- sc_procVectorGet(): Gets the interrupt vector of an interrupt process.
- sc_procCBChk(): Does a diagnostic test for all elements of the process control block of specific process.

2.4.4 Process Variables

- sc_procVarSet(): Sets or modifies a process variable.
- sc_procVarInit(): Does a setup and initializes a process variable area.
- sc_procVarGet(): Reads a process variable.
- sc_procVarDel(): Deletes a process variable from the process variable data area.
- sc_procVarRm(): Remove a whole process variable area.

2.4.5 Process Supervision

- sc_procObserve(): Supervices a process.
- sc_procUnobserve(): Cancels an installed supervision of a process.
- sc_procAtExit(): Registers a function to be called if a prioritized process is killed.
- sc_procHookRegister(): Supervices a process.

2.4.6 Process Safety

- sc_procFlowSignatureInit(): Initializes the caller's process program flow signature.
- sc_procFlowSignatureGet(): Gets the caller's process program flow signature.
- sc_procFlowSignatureUpdate(): Updates the caller's process program flow signature.

2.5 Module System Calls

2.5.1 Module Creation and Killing

- sc_moduleCreate(): Creates a module in a V1 Kernel.
- sc_moduleCreate2(): Creates a module in a V2/V2INT Kernel.
- sc_moduleKill(): Kills a module.

2.5.2 Module Controlling

- sc_moduleStop(): Stops a module.

2.5.3 Module Information

- sc_moduleCBChk(): Does a diagnostic test for all elements of the module control block of specific module.
- sc_moduleIdGet(): Gets the ID of a module.
- sc_moduleNameGet(): Gets the name of a module.
- sc_modulePrioGet(): Gets the priority of a module.
- sc_moduleInfo(): Gets a snap-shot of a module control block.

2.5.4 Module Friendship

- sc_moduleFriendAdd(): Adds a module to the friendlist in a V1 kernel.
- sc_moduleFriendAll(): Defines all existing modules in a system as friend in a V1 kernel.
- sc_moduleFriendGet(): Checks if a module is a friend in a V1 kernel.
- sc_moduleFriendNone(): Remove all modules from the friendlist in a V1 kernel.
- sc_moduleFriendRm(): Remove a module from the friendlist in a V1 kernel.

2.6 Message Pool System Calls

2.6.1 Pool Creation and Killing

- sc_poolCreate(): Creates a new message pool inside the callers module.
- sc_poolKill(): Kills a message pool.

2.6.2 Pool Controlling

- sc_poolDefault(): Sets a message pool as default pool.
- sc_poolReset(): Resets a message pool in its original state.
- sc_poolHookRegister(): Registers a pool create or pool kill hook.

2.6.3 Pool Information

- sc_poolCBChk(): Does a diagnostic test for all elements of the pool control block.
- sc_poolIdGet(): Gets the ID of a message pool by its name.
- sc_poolInfo(): Gets a snap-shot of a pool control block.

2.7 Safe Data Type System Calls

- sc_safe_charSet(): Sets safe data of specific char types at a given address in memory.
- sc_safe_charGet(): Gets safe data of specific char types.
- sc_safe_shortSet(): Sets safe data of specific short types at a given address in memory.
- sc_safe_shortGet(): Gets safe data of specific short types.
- sc_safe_<type>Set(): Set safe data of specific types at a given address in memory.
- sc_safe_<type>Get(): Gets safe data of specific types.

Kernels: Only V2INT

2.8 Timing System Calls

- sc_sleep(): Suspends the calling process for a defined time.

2.9 Timeout server System Calls

- sc_tmoAdd(): Requests a timeout message from the kernel after a defined time.
- sc_tmoRm(): Remove a timeout before it is expired.

2.10 System Tick System Calls

- sc_tick(): Calls directly the kernel tick function and advances the kernel tick counter by 1.

- sc_tickLength(): Sets or gets the current system tick length in microseconds.
- sc_tickGet(): Gets the actual kernel tick counter value.
- sc_tickGet64(): Returns the current system tick (V2 only).
- sc_tickActivationGet(): Returns the tick time of last activation of the calling process.
- sc_tickTick2Ms(): Converts a time from system ticks into milliseconds.
- sc_tickMs2Tick(): Convert a time from milliseconds into system ticks.

2.11 Process Trigger System Calls

- sc_trigger(): Activates a process trigger.
- sc_triggerValueSet(): Sets the value of a process trigger to any positive value.
- sc_triggerValueGet(): Gets the value of a process trigger.
- sc_triggerWait(): Waits on a process trigger.

2.12 CONNECTOR System Calls

- sc_connectorRegister(): Registers a connector process.
- sc_connectorUnregister(): Removes a registered connector process.
- sc_connectorRemote2Local(): Translate a remote PID to a local one.
- sc_connectorLocal2Remote(): Translate a local one to a remote PID.

2.13 CRC System Calls

- sc_miscCrc(): Calculates a 16 bit cyclic redundancy check value (CRC-16-CCITT) over a specified memory range.
- sc_miscCrcContd(): Calculates a 16 bit cyclic redundancy check value (CRC-16-CCITT) over an additional memory range.
- sc_miscCrc32(): Calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3) over a specified memory range.
- sc_miscCrc32Contd(): Calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3) over an additional memory range.
- sc_miscCrcString(): calculates a cyclic redundancy check value of a zero terminated string.
- sc_miscKerneldRegister(): Register caller as kerneld.

2.14 Error System Calls

- sc_miscError(): Calls the error hooks with a user error.
- sc_miscError2():Calls the error hooks with an user error (V2, V2INT).
- sc_miscErrnoSet(): Sets the process error number (errno) variable.
- sc_miscErrnoGet(): Gets the process error number (errno) variable.

2.15 Global Flow Control System Calls

- sc_miscFlowSignatureInit(): Initializes a global program flow signature.
- sc_miscFlowSignatureGet(): Gets a global program flow signature.

- sc_miscFlowSignatureUpdate(): Updates a global program flow signature.

2.16 Simulator System Calls

- sciopta_start(): Starts a SCIOPTA V1 Kernel Simulator application.
- sciopta_end(): Ends a SCIOPTA V1 Kernel Simulator application.

2.17 BSP System Calls

- sc_sysIrqDispatcher(): Handle hardware interrupt.
- sc_sysIrqEpilogue(): Finalize interrupt handling.
- sc_sysSWI(): Handle Software Interrupts
- sc_sysSVC(): Handle Software Interrupts

3 System Calls Reference

3.1 Introduction

This chapter contains a detailed description of all SCIOPTA kernel system calls in alphabetical order.

3.2 sc_connectorRegister

3.2.1 Description

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 by the kernel to the local connector processes.

Kernels: V1, V2 and V2INT

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	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

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_SYSTEM_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process type (see **pcb.h**).

Code | Type	**KERNEL_EALREADY_DEFINED | SC_ERR_SYSTEM_FATAL**
Description 1	Default CONNECTOR is already defined.
Extra Value 1	e0 = pid

Description 2	Process is already a CONNECTOR.
Extra Value 2	e0 = 0

Code | Type	**KERNEL_ENO_MORE_CONNECTOR**
Description	The maximum number of CONNECTORS is reached.

3.3 sc_connectorRemote2Local

3.3.1 Description

Translate a PID of a remote process to a local "remote" PID.

Kernels: V1, V2 and V2INT

3.3.2 Syntax

```
sc_pid_t sc_connectorRemote2Local( sc_pid_t connPid, sc_pid_t remotePid );
```

3.3.3 Parameter

connPid	PID of the connector for the remote system.
remotePid	remote PID.

3.3.4 Return Value

Remote PID.

3.3.5 Example

```
sc_connectorRemote2Local(connPid, remotePid);
```

3.3.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	Caller is not a connector process.
Extra Value	e0 = 0

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	connPid is not a (valid) connector PID.
Extra Value	e0 = connPid

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	remotePid is already a remote PID.
Extra Value	e0 = remotePid

3.4 sc_connectorLocal2Remote

3.4.1 Description

Translates a local remote PID to the actual PID in the remote system.

Kernels: V1, V2 and V2INT

3.4.2 Syntax

```
sc_pid_t sc_connectorLocal2Remote( sc_pid_t remotePid );
```

3.4.3 Parameter

remotePid	remote PID.

3.4.4 Return Value

pid or SC_ILLEGAL_PID if called from interrupt/timer

3.4.5 Example

```
sc_pid_t remotePid;
sc_pid_t realPid;
remotePid = sc_msgSndGet(&msg);
realPid = sc_connectorLocal2Remote(remotePid);
```

3.4.6 Errors

None.

3.5 sc_connectorUnregister

3.5.1 Description

Remove a registered connector process. The caller becomes a normal prioritized process.

Kernels: V1, V2 and V2INT

3.5.2 Syntax

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

3.5.3 Parameter

None.

3.5.4 Return Value

None.

3.5.5 Example

```
sc_connectorUnregister();
```

3.5.6 Errors

Code | Type	**KERNEL_ENO_CONNECTOR | SC_ERR_SYSTEM_FATAL**
Description	Caller is not a connector process.
Extra Value	e0 = 0

3.6 sc_miscCrc

3.6.1 Description

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

The start value of the CRC is 0xFFFF.

Kernels: V1, V2 and V2INT

3.6.2 Syntax

```
uint16_t sc_miscCrc( const uint8_t  *data, unsigned int length );
```

3.6.3 Parameter

data	Pointer to the memory range.
length	Number of bytes.

3.6.4 Return Value

The 16 bit CRC value.

3.6.5 Example

```
typedef struct ips_socket_s {
  sc_msgid_t id;
  uint16_t   srcPort;
  uint16_t   dstPort;
  ips_addr_t srcIp;
  ips_addr_t dstIp;
  dbl_t      list;
}ips_socket_t;

uint16_t     crc;
ips_socket_t *ref;

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

3.6.6 Errors

None.

3.7 sc_miscCrcContd

3.7.1 Description

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

The variable start is the CRC start value.

Kernels: V1, V2 and V2INT

3.7.2 Syntax

```
uint16_t sc_miscCrcContd( const uint8_t *data, unsigned int length, uint16_t startHash );
```

3.7.3 Parameter

data	Pointer to the memory range.
length	Number of bytes.
startHash	CRC start value.

3.7.4 Return Value

The 16 bit CRC value.

3.7.5 Example

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

3.7.6 Errors

None.

3.8 sc_miscCrc32

3.8.1 Description

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.

Kernels: V2 and V2INT

3.8.2 Syntax

```
uint32_t sc_miscCrc32( const uint8_t *data, unsigned int length );
```

3.8.3 Parameter

data	Pointer to the memory range.
length	Number of bytes.

3.8.4 Return Value

The inverted 32 bit CRC value.

3.8.5 Example

```
uint32_t bcrc;
uint32_t burst[4];

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

3.8.6 Errors

None.

3.9 sc_miscCrc32Contd

3.9.1 Description

Calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3, polynominal: 0x04C11DB7) over an additional memory range.

Kernels: V2 and V2INT

3.9.2 Syntax

```
uint32_t sc_miscCrc32Contd( const uint8_t *data, unsigned int length, uint32_t startHash );
```

3.9.3 Parameter

data	Pointer to the memory range.
length	Number of bytes.
startHash	CRC32 start value.

3.9.4 Return Value

The inverted 32 bit CRC value.

3.9.5 Example

```
uint32_t b2crc;
const uint8_t * burst2;

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

3.9.6 Errors

None.

3.10 sc_miscErrnoGet

3.10.1 Description

Get the process error number (errno) variable.

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

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

Kernels: V1, V2 and V2INT

3.10.2 Syntax

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

3.10.3 Parameter

None.

3.10.4 Return Value

Process error code.

3.10.5 Example

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

3.10.6 Errors

None.

3.11 sc_miscErrnoSet

3.11.1 Description

Set the process error number (errno) variable.

Each SCIOPTA process has an errno variable.

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

Kernels: V1, V2 and V2INT

3.11.2 Syntax

```
void sc_miscErrnoSet( sc_errcode_t err );
```

3.11.3 Parameter

err	User defined error code.

3.11.4 Return Value

None.

3.11.5 Example

```
sc_miscErrnoSet(ENODEV);
```

3.11.6 Errors

None.

3.12 sc_miscCrcString

3.12.1 Description

Calculates a 16bit CRC (CRC-16-CCITT) value of a zero terminated string.

Kernels: V1, V2 and V2INT

3.12.2 Syntax

```
uint16_t sc_miscCrcString( const char *data );
```

3.12.3 Parameter

data	Pointer to the memory range.

3.12.4 Return Value

The 16 bit CRC value.

3.12.5 Example

```
const char *process;

hash = sc_miscCrcString(process);
```

3.12.6 Errors

None.

3.13 sc_miscKerneldRegister

3.13.1 Description

Register caller as kernel daemon.
The kernel daemon is used by the kernel to create and kill processes and modules.

There can only be one kernel daemon per SCIOPTA system.

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

Kernels: V1, V2 and V2INT

3.13.2 Syntax

```
int sc_miscKerneldRegister(void)
```

3.13.3 Parameter

None.

3.13.4 Return Value

!= 0	registration successfull.

3.13.5 Example

None.

3.13.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	Caller not in system-module.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Caller not prio.
Extra Value	e0 = proctype.

3.14 sc_miscError

3.14.1 Description

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.

Kernels: V1, V2 and V2INT

3.14.2 Syntax

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

3.14.3 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.	

3.14.4 Return Value

None.

3.14.5 Example

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

3.14.6 Errors

None.

3.15 sc_miscError2

3.15.1 Description

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.

Kernels: V2 and V2INT

3.15.2 Syntax

```
void sc_miscError2( sc_errcode_t  err, sc_extra_t extra0, sc_extra_t extra1, sc_extra_t extra2, sc_extra_t extra3);
```

3.15.3 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>
extra0…3	Additional data to pass to the error hook.	

3.15.4 Return Value

None.

3.15.5 Example

```
sc_miscError2( MY_ERR_BASE + MY_ER001, 0, 1, 2, 3);
```

3.15.6 Errors

None.

3.16 sc_miscErrorHookRegister

3.16.1 Description

Register an error hook.

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

Kernels: V1, V2 and V2INT

> **Kernel V1:**
> If the error hook is registered in the start_hook it is a global error hook. If registered from a process it is a module error hook.

3.16.2 Syntax

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

3.16.3 Parameter

newhook	Function pointer to the hook.	
	<fptr>	Function pointer.
	NULL	Will unregister the error hook.

3.16.4 Return Value

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

NULL if no error hook was registered.

3.16.5 Example

```
sc_errHook_t error_hook;

sc_miscErrorHookRegister( error_hook );
```

3.16.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_SYSTEM_FATAL**
Description	Process ID of caller is not valid (SC_ILLEGAL_PID).
Extra Value	e0 = pid.

3.17 sc_miscFlowSignatureGet

3.17.1 Description

Get a global program flow signature at index **id**.

Kernels: V2 and V2INT

3.17.2 Syntax

```
uint32_t sc_miscFlowSignatureGet( unsigned int id );
```

3.17.3 Parameter

id	Global flow signature ID.
	Identity of the global flow signature which is the index into the sc_globalFlowSignatures array.

3.17.4 Return Value

Signature value.

3.17.5 Example

```
uint32_t currentSig;
currentSig = sc_miscFlowSignatureGet(10);
```

3.17.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Flow signature ID not valid (id >= SC_MAX_GFS_IDS).
Extra Value	e0 = Flow signature ID (id).

3.18 sc_miscFlowSignatureInit

3.18.1 Description

Initialize a global program flow signature at index id.

Kernels: V2 and V2INT

3.18.2 Syntax

```
void sc_miscFlowSignatureInit(unsigned int id, uint32_t signature );
```

3.18.3 Parameter

id	Global flow signature ID.
	Index of the global flow signature which is the index into the sc_globalFlowSignatures array.
signature	Initial signature.
	Value to be stored in the sc_globalFlowSignatures array.

3.18.4 Return Value

None.

3.18.5 Example

```
sc_miscFlowSignatureInit(10, 0xDEADBEEF);
```

3.18.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Flow signature ID not valid (id >= SC_MAX_GFS_IDS).
Extra Value	e0 = Flow signature ID (id).

3.19 sc_miscFlowSignatureUpdate

3.19.1 Description

Update a global program flow signature with a 32bit token.

Kernels: V2 and V2INT

3.19.2 Syntax

```
uint32_t sc_miscFlowSignatureUpdate( unsigned int id, uint32_t token );
```

3.19.3 Parameter

id	Global flow signature ID.
	Index of the global flow signature.
token	Token value.
	Token value to calculate new signature.

3.19.4 Return Value

Signature value.

3.19.5 Example

```
uint32_t currentSig;
currentSig = sc_miscFlowSignatureUpdate( 10, 0xCAFECAFE);
```

3.19.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Flow signature ID not valid (id >= SC_MAX_GFS_IDS).
Extra Value	e0 = Flow signature ID (id).

3.20 sc_moduleCBChk

3.20.1 Description

Do diagnostic test for all elements of the module control block of specific module.

Kernels: V2INT

3.20.2 Syntax

```
int sc_moduleCBChk( sc_moduleid_t mid, uint32_t *addr, unsigned int *size );
```

3.20.3 Parameter

mid	Module ID or SC_CURRENT_MID when module is current.
addr	Pointer to the address of corrupted data.
	Will be stored if mcb is corrupted.
size	Pointer to the size of corrupted data.
	Will be stored if mcb is corrupted.

3.20.4 Return Value

== 0	If the mid is wrong.
== 1	If the module control block is correct and therefore not corrupted.
== -1	If the module control block is corrupted.

3.20.5 Example

```
sc_mid_t mid = sc_moduleIdGet("ips");
if ( sc_moduleCBChk(mid, NULL, NULL) != 1 ){
  kprintf(0,"IPS corrupted!");
}
```

3.20.6 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_PROCESS_FATAL**
Description	Parameter mid not valid (>= SC_MAX_MODULE).
Extra Value	e0 = mid.

3.21 sc_moduleCreate

3.21.1 Description

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 a priority which can range between 0 (highest) to 31 (lowest) priority. For process scheduling SCIOPTA uses a combination of the module priority and process priority called effective priority. The kernel determines this effective priority as follows:

peff = min (pmodule + pprocess, 31)

This technique assures that process with highest process priority (0) cannot disturb processes in modules with lower module priority (module protection).

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

If the module priority of the created module is higher than the effective priority of the caller the init process of the created module will be swapped in.

Kernels: V1

3.21.2 Syntax

```
sc_moduleid_t sc_moduleCreate(
  const char      *name,
  void (*init)    (void),
  sc_bufsize_t    stacksize,
  sc_prio_t       moduleprio,
  char            *start,
  sc_modulesize_t size,
  sc_modulesize_t initsize,
  unsigned int    max_pools,
  unsigned int    max_procs
);
```

3.21.3 Parameter

name	Pointer to the module name.
	The name is an ASCII character string terminated be 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore.
init	Function pointer to the init process function.
	This is the address where the init process of the module will start execution.
stacksize	Stacksize of the INIT process in bytes.
moduleprio	Module priority.
	The priority of the module which range from 0 to 31.

size	Size of the module in bytes.
	The minimum module size can be estimated according to the following formula (bytes):
	size = p * 256 + stack + pools + mcb + initsize
	p Number of static processes.
	stack Sum of stack sizes of all static processes.
	pools Sum of sizes of all message pools.
	mcb Size of module control block.
	Size of module control block can be calculated as follows:
	mcb = 96 + friends + hooks * 4
	friends 0 if friends are not used, 16 if friends are used.
	hooks Number of hooks configured.
initsize	Size of the initialized data.
max_pools	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.
max_procs	Maximum number of processes in the module.
	The kernel will not allow to create more processes inside the module than stated here. Maximum value is 16384.

3.21.4 Return Value

Module ID.

3.21.5 Example

```
extern sc_module_addr_t m2mod;

my_mid = sc_moduleCreate(
    /* name           */ "m2mod",
    /* init function  */ m2mod_init,
    /* init stacksize */ 512,
    /* module prio    */ 2,
    /* module start   */ (char *)m2mod.start,
    /* module size    */ (uint32_t)m2mod.size,
    /* init size      */ (uint32_t)m2mod.initsize,
    /* max. pools     */ 4,
    /* max. process   */ 32);
  )
```

3.21.6 Errors

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EMODULE_TOO_SMALL | SC_ERR_SYSTEM_FATAL**
Description	Process control blocks and pool control blocks do not fit in module size.

Extra Value	e0 = Module size.

Code | Type	**KERNEL_EILL_NAME | SC_ERR_SYSTEM_FATAL**
Description	Requested name does not comply with SCIOPTA naming rules or does already exist.

Code | Type	**KERNEL_ENO_MORE_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Maximum number of modules reached.
Extra Value	e0 = Number of modules.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Module addresses or sizes not valid.
	Start address, size or initsize unaligned.
	initsize > size.

3.22 sc_moduleCreate2

3.22.1 Description

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.

Kernels: V2 and V2INT

3.22.2 Syntax

```
sc_moduleid_t sc_moduleCreate2( sc_mdb_t  *mdb );
```

3.22.3 Parameter

mdb	Pointer to the module descriptor block (mdb) which defines the module to create.
	See Module Descriptor Block

3.22.4 Return Value

Module ID.

3.22.5 Module Descriptor Block

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

The definition of the structure can be found in **module.h**"

Structure is the same for all architecturs (see **module.h**) !

For ARM:

```
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;
  uint8_t          safetyFlag;
  uint8_t          spare_b;
  uint16_t         spare_h;
  uint32_t         *pt;
};
typedef struct sc_mdb_s sc_mdb_t;
```

For AURIX:

```
typedef 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;
  uint8_t safetyFlag;
  uint8_t nCSA;
  uint8_t core;
  uint8_t spare_b;
  uint32_t *pt;
} sc_mdb_t;
```

For PowerPC:

```
typedef 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;
  uint8_t safetyFlag;
  uint8_t core;
  uint16_t spare_h;
  uint32_t *pt;
} sc_mdb_t;
```

3.22.5.1 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. Allowed characters are A-Z, a-z, 0-9 and underscore.

maddr	Maximum module priority.
	See Module Address and Size

maxPrio	Pointer to a structure containing the module addresses and size.
	The priority of the module which range 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 processes inside the module than stated here. Maximum value is 512.

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.	
	SC_KRN_FLAG_FALSE	Non-Safety module.
	SC_KRN_FLAG_TRUE	Safety module.

nCSA	max. CSA, AURIX only, else 0.

core	Future use, write 0.

spare_b	Spare.
	Write 0.

spare_h	Spare.
	Write 0.

pt	Pointer to the page table for MMU/MPU.	
	<pageptr>	Pointer to the MMU/MPU page table.

	NULL	No MMU/MPU in the system.

3.22.6 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**.

For ARM:

```
typedef struct sc_module_addr_s {
  char     *start;
  uint32_t size;
  uint32_t initsize;
} sc_module_addr_t;
```

For PowerPC:

```
typedef struct sc_module_addr_ppc_s {
  uint8_t *start;
  uint32_t size;
  uint32_t initsize;
  uint32_t sdata;                          /* sdata pointer (r2)             */
  uint32_t sdata2;                         /* sdata2 pointer (r13)           */
} sc_module_addr_ppc_t;
```

For AURIX:

```
typedef struct sc_module_addr_aurix_s {
  uint8_t *start;
  uint32_t size;
  uint32_t initsize;
  uint32_t csa_start;                    /* start of CSA space             */
  uint32_t csa_max;                      /* maximum number of contexts     */
} sc_module_addr_aurix_t;
```

3.22.6.1 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 as follows: size = (sizeof(sc_pcb_t)sizeof(sc_pcb_t *))*n(proc) + sum(stacks) size= (sizeof(sc_pool_cb_t *)*n(pool) + sum(pool) size+= initsize + sizeof(sc_module_cb_t) sc_pcb_t : Process Control Block sc_pool_cb_t : Pool Control Block (size depends on configuration!) sc_module_cb_t : Module Control Block (size depends on configuration!) initsize : Module local data like initialized variables or code. Could be zero.

initsize	Size of the initialized data.

sdata	sdata pointer (r2) (only **PPC**).

sdata2	sdata2 pointer (r13) (only **PPC**).

csa_start	Start of CSA space (only **AURIX**).

sdata2	Maximum number of contexts (only **AURIX**).

3.22.7 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);
```

3.22.8 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter **mdb** not valid. 0 or SC_NIL.

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EMODULE_TOO_SMALL | SC_ERR_SYSTEM_FATAL**
Description	Process control blocks and pool control blocks do not fit in module size.
Extra Value	e0 = Module size.

Code | Type	**KERNEL_EILL_NAME | SC_ERR_SYSTEM_FATAL**
Description	Requested name does not comply with SCIOPTA naming rules or does already exist.

Code | Type	**KERNEL_ENO_MORE_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Maximum number of modules reached.
Extra Value	e0 = Number of modules.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_SYSTEM_FATAL**

Description	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

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Module addresses or sizes not valid. Start address, size or initsize unaligned. initsize > size.

Code | Type	**KERNEL_EMODULE_OVERLAP | SC_ERR_SYSTEM_FATAL**
Description	Modules do overlap.
Extra Value	e0 = Requested start address e1 = Module start address

3.23 sc_moduleFriendAdd

3.23.1 Description

Add a module to the friendlist. The caller defines the module mid as friend. The module is entered in the friend set of the caller.

Messages sent to a process in module which is "friend" will not be copied.

Kernels: V1

3.23.2 Syntax

```
void sc_moduleFriendAdd( sc_moduleid_t mid );
```

3.23.3 Parameter

mid	Module ID.
	The ID of the module to add.

3.23.4 Return Value

None.

3.23.5 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = Module ID.

3.24 sc_moduleFriendAll

3.24.1 Description

Define all existing modules in a system as friend.

Kernels: V1

3.24.2 Syntax

```
void sc_moduleFriendAll(void);
```

3.24.3 Parameter

None.

3.24.4 Return Value

None.

3.24.5 Errors

None.

3.25 sc_moduleFriendGet

3.25.1 Description

Check if a module is a friend. The caller will be informed if the module in parameter mid is a friend.

Kernels: V1

3.25.2 Syntax

```
int sc_moduleFriendGet( sc_moduleid_t mid );
```

3.25.3 Parameter

mid	Module ID.
	The ID of the module which will be checked if it is a friend or not.

3.25.4 Return Value

== 0	If the module is not a friend (not included in the friend set)
!== 0	If the mModule is a friend (included in the friend set)

3.25.5 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = Module ID.

3.26 sc_moduleFriendNone

3.26.1 Description

Remove all modules from the friendlist.

Kernels: V1

3.26.2 Syntax

```
void sc_moduleFriendNone(void);
```

3.26.3 Parameter

None.

3.26.4 Return Value

None.

3.26.5 Errors

None.

3.27 sc_moduleFriendRm

3.27.1 Description

Remove a module from the friendlist. The caller removes the module in parameter mid as friend.

Kernels: V1

3.27.2 Syntax

```
void sc_moduleFriendRm( sc_moduleid_t mid );
```

3.27.3 Parameter

mid	Module ID.
	The ID of the module of the old friend to remove.

3.27.4 Return Value

None.

3.27.5 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = Module ID.

3.28 sc_moduleIdGet

3.28.1 Description

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.

Kernels: V1, V2 and V2INT

3.28.2 Syntax

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

3.28.3 Parameter

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

3.28.4 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.

3.28.5 Example

```
sc_moduleid_t mid;

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

3.28.6 Errors

Code | Type	**KERNEL_EILL_MODULE_NAME | SC_ERR_PROCESS_WARNING**
Description	String pointed to by name too long.
Extra Value	e0 = Name.

3.29 sc_moduleInfo

3.29.1 Description

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. A 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 this structure with the module control block data.

You cannot directly access the module control blocks.

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

Kernels: V1, V2 and V2INT

3.29.2 Syntax

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

3.29.3 Parameter

mid	Module ID.	
	<mid>	ID of the module.
	SC_CURRENT_MID	Current module ID (module ID of the caller).
info	Pointer to a local structure of a module control block.	
	See Module Info Structure	

3.29.4 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.

3.29.5 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**.

For Kernels V1:

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

For Kernels V2 and V2INT:

```
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;
```

3.29.5.1 Structure Members

mid	Module ID.
ppid	Process which created the module (**V2/INT only**).
name	Name of the module.
text	Current address into module text segment.
textsize	Size of module text segment (initialized data as it does also contain static variables).
data	Start address of the module's data area.
datasize	Total size of the module's data area.
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..

3.29.6 Example

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

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

3.29.7 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter info not valid (info == 0).

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_PROCESS_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = mid.

3.30 sc_moduleKill

3.30.1 Description

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.

Kernels: V1, V2 and V2INT

3.30.2 Syntax

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

3.30.3 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.

3.30.4 Return Value

None.

3.30.5 Example

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

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

3.30.6 Errors

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Module to be killed is the system module.
	MID is not valid (mid >= SC_MAX_MODULE).
	MID is not valid (mcb ==SC_NIL).

Extra Value	e0 = mid.

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_SYSTEM_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = pcb.

3.31 sc_moduleNameGet

3.31.1 Description

Get the name of a module.

The name will be returned as a 0 terminated string.

Kernels: V1, V2 and V2INT

3.31.2 Syntax

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

3.31.3 Parameter

mid	Module ID.	
	<mid>	Module ID of the module to get the name.
	SC_CURRENT_MID	Current module ID (module ID of the caller).

3.31.4 Return Value

Name string	If the module was found.
NULL	If the Module was not found.

3.31.5 Example

```
sc_moduleid_t mid;

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

3.31.6 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_SYSTEM_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = mid.

3.32 sc_modulePrioGet

3.32.1 Description

Get the priority of a module.

Kernels: V1, V2 and V2INT

3.32.2 Syntax

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

3.32.3 Parameter

mid	Module ID.	
	<mid>	Module ID of the module to get the priority.
	SC_CURRENT_MID	Current module ID (module ID of the caller).

3.32.4 Return Value

Module priority	If the module was found and was valid.
SC_ILLEGAL_PRIO	If the module was not valid (mcb == SC_NIL).

3.32.5 Example

```
sc_moduleid_t mid;

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

3.32.6 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_PROCESS_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = mid.

3.33 sc_moduleStop

3.33.1 Description

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.

Kernels: V2 and V2INT

3.33.2 Syntax

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

3.33.3 Parameter

mid	Module ID.	
	<mid>	Module ID of the module to stop.
	SC_CURRENT_MID	Current module ID (module ID of the caller).

3.33.4 Return Value

None.

3.33.5 Example

```
sc_moduleid_t mid;

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

3.33.6 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_PROCESS_FATAL**
Description	Module ID not valid (mid >= SC_MAX_MODULE).
Extra Value	e0 = mid.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Stopcounter overrun.
Extra Value	e0 = pcb.

3.34 sc_msgAcquire

3.34.1 Description

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.

Kernels: V1, V2 and V2INT

3.34.2 Syntax

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

3.34.3 Parameter

msgptr	Pointer to the message buffer pointer.

3.34.4 Return Value

None.

3.34.5 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 */
```

3.34.6 Errors

Code | Type	**KERNEL_EMSG_HD_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message header is corrupt. Kernel is the message owner.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.35 sc_msgAddrGet

3.35.1 Description

Get the process ID of the addressee of a message.

This system call is used in communication software of distributed multi-CPU systems (using connector processes). It allows to retrieve the original addressee if the kernel has forwarded a message to a connector which was sent to a remote process.

Kernels: V1, V2 and V2INT

3.35.2 Syntax

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

3.35.3 Parameter

msgptr	Pointer to the message buffer pointer.

3.35.4 Return Value

Process ID of the addressee of the message.

3.35.5 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);
```

3.35.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
	e1 = Pointer to message.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**

Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.36 sc_msgAlloc

3.36.1 Description

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.

SCIOPTA is not returning a buffer with the exact amount of bytes requested but will select the best fit from a list of fixed buffer sizes. This list can contain 4, 8 or 16 different 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 memory shortage in message pools (tmo).

Kernels: V1, V2 and V2INT

3.36.2 Syntax

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

3.36.3 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.	
	<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. Note: This parameter is not recommended. Use with caution.
	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. The function will not return.
	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.

3.36.4 Return Value

Pointer to the allocated buffer or pointer to the message or NULL if the timeout has expired.

3.36.5 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 */
);
```

3.36.6 Errors

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_PROCESS_FATAL**
Description	Pool index is not available.
Extra Value	e0 = Pool inde.

Code | Type	**KERNEL_EILL_BUFSIZE | SC_ERR_PROCESS_FATAL**
Description	Illegal message size was requested.
Extra Value	e0 = Requested size.
	e1 = Pool CB (or -1).

Code | Type	**KERNEL_EOUT_OF_MEMORY | SC_ERR_MODULE_FATAL**
Description	Request for number of bytes could not be fulfilled.
Extra Value	e0 = size.
	e1 = Pool CB.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_PROCESS_FATAL**
Description	tmo value not valid.
Extra Value	e0 = tmo value.

Code | Type	**KERNEL_EILL_DEFPOOL_ID | SC_ERR_PROCESS_WARNING**
Description	Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook.
Extra Value	e0 = Pool index.

Code | Type	**KERNEL_ELOCKED | SC_ERR_MODULE_FATAL**
Description	Process would swap but interrupts and/or scheduler are/is locked.
Extra Value	e0 = Lock counter value or -1 if interrupt are locked.

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Illegal process type.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**

Description	tmo-flag with wrong value. Likely system is corrupt.
Extra Value	e0 = tmo-flag.

3.37 sc_msgAllocClr

3.37.1 Description

This system call works exactly the same as sc_msgAlloc but will clear the data area to zero.

Kernels: V1, V2 and V2INT

3.37.2 Syntax

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

3.37.3 Parameter

Parameter values are the same as in sc_msgAlloc.

3.37.4 Return Value

Return values are the same as in sc_msgAlloc.

3.37.5 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 */
);
```

3.37.6 Errors

Errors are the same as in sc_msgAlloc.

3.38 sc_msgAllocTx

3.38.1 Description

Allocates a message of 12 (32 bit systems) or 20 (64 bit systems) bytes from the default pool of the addressee and stores id, data1 and data2 in this message. Then the message is transmitted to the addressee.

This call combines sc_msgAlloc with sc_msgTx and no copying is involved if the message is sent across module boundaries.

Kernels: V1, V2 and V2INT

3.38.2 Syntax

```
void sc_msgAllocTx(sc_msgid_t id,int data1,int data2, sc_pid_t addressee );
```

3.38.3 Parameter

id	Message ID.	
	The message ID which will be placed at the beginning of the data buffer of the message.	
data1	Data 1 to send.	
data2	Data 2 to send.	
addressee	The process ID of the addressee.	
	<pid>	Valid SCIOPTA Process ID. Addressee must be a prioritized process.
	SC_CURRENT_PID	The caller himself.

3.38.4 Return Value

None.

3.38.5 Example

```
sc_msgAllocTx(ACK_MSG, 10, 20, dest_pid);
```

3.38.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Addressee pid not valid (is init0). Bigger than MODULE_MAXPROCESS). Illegal CONNECTOR pid. Illegal addressees module-index.
Extra Value	e0 = Addressee process ID.

Code | Type	**KERNEL_EILL_BUFSIZE | SC_ERR_PROCESS_FATAL**
Description	Illegal message size was requested.
Extra Value	e0 = Requested size.
	e1 = Pool CB (or -1).

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_EILL_DEFPOOL_ID | SC_ERR_PROCESS_WARNING**
Description	Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook.
Extra Value	e0 = Pool index.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_MODULE_FATAL**
Description	Pool index in message header has an illegal value. Thrown in the addressee's module.
Extra Value	e0 = Pool index.

Code | Type	**KERNEL_EOUTSIDE_POOL | SC_ERR_MODULE_FATAL**
Description	The pointer is outside the pool. Possible pool id corruption. Thrown in the addressee's module.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EOUT_OF_MEMORY | SC_ERR_MODULE_FATAL**
Description	Request for number of bytes could not be fulfilled. Thrown in the addressee's module.
Extra Value	e0 = size.
	e1 = Pool CB.

3.39 sc_msgDataCrcDis

3.39.1 Description

Disable the message data CRC check for the message in the parameter.

Kernels: V2 and V2INT

3.39.2 Syntax

```
void sc_msgDataCrcDis( sc_msgptr_t msg );
```

3.39.3 Parameter

msg	Pointer to the message pointer.

3.39.4 Return Value

None.

3.39.5 Example

TBD

3.39.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_EILL_BUFSIZE | SC_ERR_PROCESS_FATAL**
Description	Illegal message size was requested.
Extra Value	e0 = Requested size.
	e1 = Pool CB (or -1).

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = PID of owner.
	e1 = Pointer to message header.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**

Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.40 sc_msgDataCrcGet

3.40.1 Description

Check the message data checksum.

In the SCIOPTA Safety Kernel INT the message header may contain a CRC32 hash of the message data. If the message data is corrupt or no check is performed a 0 is returned. If the message data is valid the checksum is returned.

Kernels: V2 and V2INT

3.40.2 Syntax

```
uint32_t sc_msgDataCrcGet( sc_msgptr_t msg );
```

3.40.3 Parameter

msg	Pointer to the message pointer.

3.40.4 Return Value

CRC	If the CRC was set and valid.
0	If CRC was not set or invalid.

3.40.5 Example

TBD

3.40.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = PID of owner.
	e1 = Pointer to message header.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.41 sc_msgDataCrcSet

3.41.1 Description

In the SCIOPTA Safety Kernel INT the message header may contain a CRC32 value of the message.

Calculate a CRC32 over the message data and store it in the message header.

Kernels: V2INT

3.41.2 Syntax

```
uint32_t sc_msgDataCrcSet( sc_msgptr_t msg );
```

3.41.3 Parameter

msg	Pointer to the message pointer.

3.41.4 Return Value

CRC of the message data.

3.41.5 Example

TBD

3.41.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = PID of owner.
	e1 = Pointer to message header.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.42 sc_msgFind

3.42.1 Description

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 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 caller.

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.

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.

Kernels: V2 and V2INT

3.42.2 Syntax

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

3.42.3 Parameter

mp	Pointer to a message in the allocated-messages queue.	
	<ptr>	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_FIND_ALL	All messages will be searched.

flag	Select array definition.	
	SC_FIND_MSGID	An array of wanted message IDs is given.
	SC_FIND_PID	An array of process IDs from where sent messages are received is given.
	SC_FIND_NOT	An array of message IDs is given which will be excluded from the search.
	SC_FIND_BOTH	An array of pairs of message IDs and process IDs are given to search for specific messages from specific transmitting processes.

3.42.4 Return Value

Pointer to the found message.

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

3.42.5 Example

TBD

3.42.6 Errors

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Illegal flags.
Extra Value	e0 = Flags.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	tmo-flag with wrong value. Likely system is corrupt.
Extra Value	e0 = tmo-flag.

3.43 sc_msgFlowSignatureUpdate

3.43.1 Description

Update a global message flow signature with message header elements: message ID, sender process ID and addressee process ID. The result is returned and also stored back to the global flow signature.

Kernels: V2 and V2INT

3.43.2 Syntax

```
uint32_t sc_msgFlowSignatureUpdate( unsigned int id, sc_msgptr_t msg );
```

3.43.3 Parameter

id	Global flow signature ID.
	Index of the global flow signature.
msg	Pointer to message.

3.43.4 Return Value

Signature value.

3.43.5 Example

```
msg = sc_msgRx(SC_ENDLESS_TMO, SC_MSGRX_ALL, SC_MSGRX_MSGID);
currentSig = sc_msgFlowSignatureUpdate(10, &msg);
```

3.43.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_PROCESS_FATAL**
Description	Flow signature ID not valid (id >= SC_MAX_GFS_IDS).
Extra Value	e0 = Flow signature ID (id).

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.44 sc_msgFree

3.44.1 Description

Return an allocated message to the message pool. Message buffers which have been returned can be used again.

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.

Another process actually waiting to allocate a message of a full pool will become ready and therefore the caller process can be 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 to.

Kernels: V1, V2 and V2INT

3.44.2 Syntax

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

3.44.3 Parameter

msgptr	Pointer to message pointer.

3.44.4 Return Value

None.

3.44.5 Example

```
/* Free a message */

sc_msg_t msg;

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

3.44.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = PID of owner.
	e1 = Pointer to message.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Process ID is wrong.
Extra Value	e0 = Process ID.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Module in message header has an illegal value.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_MODULE_FATAL**
Description	Pool index in message header has an illegal value.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EMSG_HD_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Either pool ID or buffersize index are corrupted.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EOUTSIDE_POOL | SC_ERR_MODULE_FATAL**
Description	The pointer is outside the pool. Possible pool id corruption.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Message is a timout message.

3.45 sc_msgHdCheck

3.45.1 Description

Check the message header. The header will be checked for plausibility.

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

Kernels: V2 and V2INT

3.45.2 Syntax

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

3.45.3 Parameter

msgptr	Pointer to message pointer.

3.45.4 Return Value

!=0	If the message header is ok.
==0	If the message header is corrupted.

3.45.5 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!");
};
```

3.45.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

3.46 sc_msgHookRegister

3.46.1 Description

Register a message hook.

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

Kernels V1 only: 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.

Kernels V1 only: First the module and then the global message hook will be called.

The transmit hook is called before the message is entered in the addressee's queue.
The receive hook is called directly before returning to the receiver.

Kernels: V1, V2 and V2INT

3.46.2 Syntax

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

3.46.3 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.

3.46.4 Return Value

<funcptr>	Function pointer to the previous message hook. if the message hook was registered.
NULL	If no message hook was registered.

3.46.5 Example

```
sc_msgHook_t *oldTx;
sc_msgHook_t *oldRx;

/* stop trace if running */
oldTx = sc_msgHookRegister (SC_SET_MSGTX_HOOK, 0);
oldRx = sc_msgHookRegister (SC_SET_MSGRX_HOOK, 0);
```

3.46.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Wrong type (unknown or not active).
Extra Value	e0 = Requested message hook type.

3.47 sc_msgOwnerGet

3.47.1 Description

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.

Kernels: V1, V2 and V2INT

3.47.2 Syntax

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

3.47.3 Parameter

msgptr	Pointer to message pointer.

3.47.4 Return Value

Process ID of the owner of the message.

3.47.5 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 );
```

3.47.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.48 sc_msgPoolIdGet

3.48.1 Description

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.

Kernels: V1, V2 and V2INT

3.48.2 Syntax

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

3.48.3 Parameter

msgptr	Pointer to message pointer.

3.48.4 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.

3.48.5 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 );
```

3.48.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
	e1 = Pointer to message

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.49 sc_msgRx

3.49.1 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.

Kernel V2 only: 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.

Kernerl V2 only: 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.

Kernels: V1, V2 and V2INT

3.49.2 Syntax

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

3.49.3 Parameter

tmo	Timeout.	
	SC_ENDLESS_TMO	Blocks and waits endless until the message is received.
	SC_NO_TMO	No time out, 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 IDs from where sent messages are received is given.
	SC_MSGRX_BOTH	An array of pairs of message IDs and process IDs are given to receive specific messages from specific transmitting processes.
	SC_MSGRX_NOT	An array of message IDs is given which will be excluded from receive.

3.49.4 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.

3.49.5 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_msg_rx_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);
```

3.49.6 Errors

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_PROCESS_FATAL**
Description	tmo value not valid.
Extra Value	e0 = tmo value.

Code | Type	**KERNEL_ELOCKED | SC_ERR_MODULE_FATAL**
Description	Process would swap but interrupts and/or scheduler are/is locked.
Extra Value	e0 = Lock counter value or -1 if interrupt are locked.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Illegal flags.
Extra Value	e0 = flags.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	tmo-flag with wrong value. Likely system is corrupt.
Extra Value	e0 = tmo-flag.

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	The calling process uses a timeout and is not a prioritized process.
Extra Value	e0 = tmo-flag.

3.50 sc_msgSizeGet

3.50.1 Description

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.

Kernels: V1, V2 and V2INT

3.50.2 Syntax

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

3.50.3 Parameter

msgptr	Pointer to message pointer.

3.50.4 Return Value

Requested size of the message.

3.50.5 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 );
```

3.50.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
Extra Value	e1 = Pointer to message.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.

Extra Value	e0 = Pointer to previous message.

3.51 sc_msgSizeSet

3.51.1 Description

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.

Kernels: V1, V2 and V2INT

3.51.2 Syntax

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

3.51.3 Parameter

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

3.51.4 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.

3.51.5 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) */
```

3.51.6 Errors

Code | Type	**KERNEL_EILL_BUFSIZE | SC_ERR_MODULE_FATAL**
Description	Illegal buffer sizes.
Extra Value	e0 = Buffer sizes.

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_MODULE_FATAL**
Description	Parameter size is smaller than the size of a message id.
Extra Value	e0 = Buffer sizes.

Code | Type	**KERNEL_EENLARGE_MSG | SC_ERR_MODULE_FATAL**
Description	Message would be enlarged.
Extra Value	e0 = Buffer size.
Extra Value	e1 = Pointer to message.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
Extra Value	e1 = Pointer to message.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.52 sc_msgSndGet

3.52.1 Description

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.

Kernels: V1, V2 and V2INT

3.52.2 Syntax

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

3.52.3 Parameter

msgptr	Pointer to message pointer.

3.52.4 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.

3.52.5 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 );
```

3.52.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
Extra Value	e1 = Pointer to message.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.

Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

3.53 sc_msgTx

3.53.1 Description

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 kernel 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 caller's module and this module is not registered as a friend module then 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 default pool of the addressee 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.

Kernels: V1, V2 and V2INT

3.53.2 Syntax

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

3.53.3 Parameter

msgptr	Pointer to message pointer.	
addr	The process ID of the addressee.	
	<pid>	Valid SCIOPTA Process ID.
	SC_CURRENT_PID	The caller himself.
flags	Transmitt flags.	
	More than one value can be defined and must be separated by OR instructions.	
	SC_MSGTX_NO_FLAG	Normal sending.
	SC_MSGTX_RTN2SNDR	Return message if addressee does not exist or if there is no memory to copy it into the addressee module.

3.53.4 Return Value

None.

3.53.5 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 );
```

3.53.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_MODULE_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.
Extra Value	e0 = Pointer to message pointer.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
Extra Value	e1 = Pointer to message.

Code | Type	**KERNEL_EMSG_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Message endmark is corrupt.
Extra Value	e0 = Pointer to message.

Code | Type	**KERNEL_EMSG_PREV_ENDMARK_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Endmark of previous message is corrupt.
Extra Value	e0 = Pointer to previous message.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Process type not valid.
Extra Value	e0 = pid of addressee.
Extra Value	e1 = Process type.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Flag is neither 0 nor SC_MSGTX_RTN2SND.
Extra Value	e0 = Flag value.
Extra Value	e1 = 2 (position).

Code | Type	**KERNEL_EALREADY_DEFINED | SC_ERR_PROCESS_FATAL**
Description	Caller tries to send a timeout message but message is already a timeout message.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**

Description	Module in message header has an illegal value.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_MODULE_FATAL**
Description	Pool index in message header has an illegal value.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EMSG_HD_CORRUPT | SC_ERR_MODULE_FATAL**
Description	Either pool ID or buffersize index are corrupted.
Extra Value	e0 = Pointer to message header.

Code | Type	**KERNEL_EOUTSIDE_POOL | SC_ERR_MODULE_FATAL**
Description	The pointer is outside the pool. Possible pool id corruption.
Extra Value	e0 = Pointer to message header.

3.54 sc_msgTxAlias

3.54.1 Description

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 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.

Kernels: V1, V2 and V2INT

3.54.2 Syntax

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

3.54.3 Parameter

msgptr	Pointer to message pointer.	
addr	The process ID of the addressee.	
	<pid>	Valid SCIOPTA Process ID.
	SC_CURRENT_PID	The caller himself.
flags	Transmitt flags.	
	More than one value can be defined and must be separated by OR instructions.	
	SC_MSGTX_NO_FLAG	Normal sending.
	SC_MSGTX_RTN2SNDR	Return message if addressee does not exist or if there is no memory to copy it into the addressee module.

3.54.4 Return Value

None.

3.54.5 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 );
```

3.54.6 Errors

Same errors as in previous chapter sc_msgTx.

3.55 sc_poolCBChk

3.55.1 Description

Do diagnostic test for all elements of the pool control block of specific message pool.

Kernels: V2INT

3.55.2 Syntax

```
int sc_poolCBChk(sc_modulid_t mid, sc_plid_t idx, uint32_t *addr, unsigned int *size );
```

3.55.3 Parameter

mid	Module ID or SC_CURRENT_MID when module is current.
idx	Pool index.
addr	Pointer to the address of corrupted data.
	Will be stored if pool cb is corrupted.
size	Pointer to the size of corrupted data.
	Will be stored if pool cb is corrupted.

3.55.4 Return Value

== 0	If the mid or idx is wrong.
== 1	If the pool control block is correct and therefore not corrupted.
== -1	If the pool control block is corrupted.

3.55.5 Example

```
mid = sc_moduleIdGet("ips");
if ( sc_poolCBChk(mid, 0, NULL, NULL) != 1 ){
  kprintf(0,"IPS Pool 0 corrupt!\n");
}
```

3.55.6 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_PROCESS_FATAL**
Description	Parameter mid not valid (>= SC_MAX_MODULE).
Extra Value	e0 = mid.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_PROCESS_FATAL**
Description	Pool index too large.
Extra Value	e0 = Pool Index.

3.56 sc_poolCreate

3.56.1 Description

Create a new message pool inside the callers module.

Kernels: V1, V2 and V2INT

3.56.2 Syntax

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

3.56.3 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). The size of the pool control block (pool_cb) can be calculated according to the following formula	
	pool_cb = 68 + n * 20 + stat * n * 20	
	n	Maximum buffer sizes defined for the whole system (and not the buffer sizes of the created pool). Value n can be 4, 8 or 16..
	stat	Process 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. Allowed characters are A-Z, a-z, 0-9 and underscore.	

3.56.4 Return Value

Pool ID of the created message pool.

3.56.5 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"
);
```

3.56.6 Errors

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Illegal module. module >= SC_MAX_MODULE. module == SC_NIL
Extra Value	e0 = Module ID.

Code | Type	**KERNEL_EILL_BUF_SIZES | SC_ERR_MODULE_FATAL**
Description	Illegal buffer sizes.
Extra Value	e0 = Requested buffer sizes.

Code | Type	**KERNEL_EILL_NAME | SC_ERR_MODULE_FATAL**
Description	Illegal pool name requested.
Extra Value	e0 = Requested pool name.

Code | Type	**KERNEL_ENO_MORE_POOL | SC_ERR_MODULE_FATAL**
Description	Maximum number of pools for module reached.
Extra Value	e0 = Number of pools in module cb.

Code | Type	**KERNEL_EILL_NUM_SIZES | SC_ERR_MODULE_FATAL**
Description	Illegal number of buffer sizes.
Extra Value	e0 = s.

Code | Type	**KERNEL_EOUT_OF_MEMORY | SC_ERR_MODULE_FATAL**
Description	No more memory in module for pool.
Extra Value	e0 = Requested pool size.

Code | Type	**KERNEL_EILL_POOL_SIZE | SC_ERR_MODULE_FATAL**
Description	Size is not 4-byte aligned. Even the biggest buffer does not fit.
Extra Value	e0 = Requested pool size.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_MODULE_FATAL**
Description	ool start address is not 4-byte aligned.

Extra Value	e0 = Requested pool start address.

3.57 sc_poolDefault

3.57.1 Description

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.

Kernels: V1, V2 and V2INT

3.57.2 Syntax

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

3.57.3 Parameter

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

3.57.4 Return Value

Pool ID of the default pool.

3.57.5 Example

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

3.57.6 Errors

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_PROCESS_WARNING**
Description	Pool index not valid.
	Pool index > 16.
	Pool index > MODULE_MAXPOOLS.
Extra Value	e0 = Requested pool index.

3.58 sc_poolHookRegister

3.58.1 Description

Register a pool create or pool kill hook.

V1 only: 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.

Kernels: V1, V2 and V2INT

3.58.2 Syntax

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

3.58.3 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.

3.58.4 Return Value

Function pointer to the previous pool hook	If the pool hook was registered.
NULL	If no pool hook was registered.

3.58.5 Example

```
sc_poolHook_t oldPoolHook;

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

3.58.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Pool hook type not defined.

Extra Value	e0 = Pool hook type.
	e1 = 0.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Pool hook function pointer not valid.
Extra Value	e0 = Pool hook type.
	e1 = 1.

3.59 sc_poolIdGet

3.59.1 Description

Get the ID of a message pool by its name.

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

Kernels: V1, V2 and V2INT

3.59.2 Syntax

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

3.59.3 Parameter

name	Pointer to the 0 terminated name string.	
	NULL	Default pool.
	SC_NIL	Default pool.
	Empty string	Default pool.

3.59.4 Return Value

Pool ID	If pool was found.
SC_ILLEGAL_POOLID	If pool was not found.

3.59.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet( "fs_pool" );

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

3.59.6 Errors

Code | Type	**KERNEL_EILL_NAME | SC_ERR_PROCESS_FATAL**
Description	Illegal pool name.
Extra Value	e0 = pointer to pool name.

3.60 sc_poolInfo

3.60.1 Description

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

Kernels: V1, V2 and V2INT

3.60.2 Syntax

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

3.60.3 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. See
	This structure will be filled with the pool control block data. It is included in the header file **pool.h**

3.60.4 Return Value

!=0	If pool control block data was successfully retrieved.
==0	If the pool control block could not be retrieved.

3.60.5 Pool Control Block 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;
```

3.60.5.1 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.
	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 freed buffers.
waiter	List of processes waiting for a buffer.
name	Pointer to pool name.
stat	Statistics information. See chapter Pool Statistics Info Structure

3.60.6 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{
  uint32_t cnt_req[SC_MAX_NUM_BUFFERSIZES];      /* No. requests for a spec. size */
  uint32_t cnt_alloc[SC_MAX_NUM_BUFFERSIZES];    /* No. allocation of a spec. size */
  uint32_t cnt_free[SC_MAX_NUM_BUFFERSIZES];     /* No. releases of a spec. size */
  uint32_t cnt_wait[SC_MAX_NUM_BUFFERSIZES];     /* No. unfullfilled allocations */
  sc_bufsize_t maxalloc[SC_MAX_NUM_BUFFERSIZES]; /* largest wanted size */
} sc_pool_stat_t;
```

3.60.6.1 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.

3.60.7 Example

```
sc_moduleid_t modid;
sc_poolid_t pl;
sc_pool_cb_t pool_info;
```

```
int check;

modid = sc_moduleIdGet("my_module")
pl = sc_poolIdGet("my_pool");

check = sc_poolInfo( modid, pl, &pool_info );
```

3.60.8 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Illegal pointer to info structure.
Extra Value	e0 = 0.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_MODULE_FATAL**
Description	Illegal pool ID.
Extra Value	e0 = pool ID.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Illegal module.
Extra Value	e0 = module ID.

3.61 sc_poolKill

3.61.1 Description

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.

Kernels: V1, V2 and V2INT

3.61.2 Syntax

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

3.61.3 Parameter

plid	ID of the pool to be killed.

3.61.4 Return Value

None.

3.61.5 Example

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

3.61.6 Errors

Code | Type	**KERNEL_EPOOL_IN_USE | SC_ERR_MODULE_FATAL**
Description	Pool is in use and cannot be killed.
Extra Value	e0 = pool cb.
	e1 = pool lock counter.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_MODULE_FATAL**
Description	Illegal pool ID.
Extra Value	e0 = pool ID.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Illegal module.
Extra Value	e0 = module ID.

3.62 sc_poolReset

3.62.1 Description

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.

Kernels: V1, V2 and V2INT

3.62.2 Syntax

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

3.62.3 Parameter

plid	ID of the pool to reset.

3.62.4 Return Value

None.

3.62.5 Example

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

3.62.6 Errors

Code | Type	**KERNEL_EPOOL_IN_USE | SC_ERR_MODULE_FATAL**
Description	Pool is in use and no reset can be performed.
Extra Value	e0 = pool cb.
	e1 = pool lock counter.

Code | Type	**KERNEL_EILL_POOL_ID | SC_ERR_MODULE_FATAL**
Description	Illegal pool ID.
Extra Value	e0 = pool ID.

3.63 sc_procAtExit

3.63.1 Description

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.

The function runs in the context of the caller but has no access to variables on the stack (stack is rewound)!

Kernels: V2 and V2INT

3.63.2 Syntax

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

3.63.3 Parameter

func	Function to be called.	
	<funcptr>	Pointer to the function to be called.
	NULL	Remove previous registered function.
	SC_NIL	No function to register.

3.63.4 Return Value

Pointer to the old function	If none was registerred.
NULL	If none was registerred.

3.63.5 Example

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

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

3.63.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_PROCESS_FATAL**
Description	Illegal function pointer.
Extra Value	e0 = Function pointer.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Wrong process type. Can only be called within a prioritized process.
Extra Value	e0 = Process type.

3.64 sc_procAttrGet

3.64.1 Description

Get specific attributes for a process.

Kernels: V2 and V2INT

3.64.2 Syntax

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

3.64.3 Parameter

pid	Pocess 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 checks if the process exists.
	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. It is a bit field which is zero or a combination of the following values:
		SC_PROCATTR_STATE_WAIT_RX Process waits on message receive.
		SC_PROCATTR_STATE_WAIT_TRG Process waits on trigger.
		SC_PROCATTR_STATE_WAIT_ALLOC Process waits for message to be allocated.
		SC_PROCATTR_STATE_WAIT_TMO Process waits with timeout.
		SC_PROCATTR_STATE_READY process is ready.
	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.

value	Pointer where to store the attribute.
	Could be NULL for SC_PROCATTR_NOP.
	Should point to a 32bit variable or in case of SC_PROCATTR_NAME to an array of at least SC_PROC_NAME_SIZE+1 bytes.

3.64.4 Return Value

== 0	If process is not found.
== 1	If value was successfully written.

3.64.5 Example

```
int msg_number

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

3.64.6 Errors

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Illegal process attribute.
Extra Value	e0 = Process attribute.
	e1 = 1

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Pointer to value not valid (NULL)

3.65 sc_procCBChk

3.65.1 Description

Do diagnostic test for all elements of the process control block of specific process.

Kernels: V2INT

3.65.2 Syntax

```
int sc_procCBChk(sc_pid_t pid, uint32_t  *addr, unsigned int *size );
```

3.65.3 Parameter

pid	Process ID.	
	<pid>	Process ID of the process to get the pcb.
	SC_CURRENT_PID	Current running (caller) process.
addr	Address of corrupted data.	
size	Size of corrupted data.	

3.65.4 Return Value

== 0	If the pid is wrong.
== 1	If the process control block is correct and therefore not corrupted.
== -1	If the process control block is corrupted.

3.65.5 Example

```
if ( sc_procCBChk(SC_CURRENT_PID, NULL, NULL) != 1 ){
  kprintf(0,"PCB corrupted\n");
}
```

3.65.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Parameter pid not valid (== SC_ILLEGAL_PID).
Extra Value	e0 = pid.

3.66 sc_procCreate2

3.66.1 Description

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.

Kernels: V2 and V2INT

3.66.2 Syntax

```
sc_pid_t sc_procCreate2(const sc_pdb_t *pdb, int state, sc_poolid_t plid );
```

3.66.3 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.	

3.66.4 Return Value

ID of the created process.

3.66.5 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.

For all CPUs except Aurix:

```
#define PDB_COMMON(para)  \
  uint32_t type;          \
  const char *name;       \
  void (* entry)(para);   \
  sc_bufsize_t stacksize; \
  sc_pcb_t * pcb;         \
  char *stack;            \
  uint8_t super;          \
  uint8_t fpu;            \
  uint16_t spare
```

For Aurix: CPUs

```
#define PDB_COMMON(para)  \
  uint32_t type;          \
  const char *name;       \
  void (* entry)(para);   \
  sc_bufsize_t stacksize; \
  sc_pcb_t * pcb;         \
  char *stack;            \
  uint8_t super;          \
  uint8_t fpu;            \
  uint8_t spare_b         \
  uint8_t nCSA            \
  uint8_t *csabtm
```

```
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);
  unsigned int ivector;
} sc_pdbint_t;
```

```
typedef struct sc_pdbprio_s {
  PDB_COMMON(void);
  sc_ticks_t   slice;
  unsigned int 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;
```

3.66.6 Structure Members Common for all Process Types

type	Process type.	
	PCB_TYPE_PRI	Prioritized process.
	PCB_TYPE_TIM	Timer process.
	PCB_TYPE_INT	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. Allowed 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. Processor typic alignment restriction apply.	
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.	

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.	
spare_b	Not used, write 0 (for Aurix only).	
nCSA	Number of CSAs (for Aurix only).	
csabtm	Start of the CSA memory (for Aurix only).	
	Must be in DSPR and aligned on 64 bytes.	

3.66.7 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.

3.66.8 Additional Structure Members for Interrupt Processes

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

3.66.9 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.

3.66.10 Example

```
static const sc_pdbprio_t pdb = {
  /* process-type    */ PCB_TYPE_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) ;
```

3.66.11 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter pdb not valid (0 or SC_NIL).

Code | Type	**KERNEL_EOUT_OF_MEMORY | SC_ERR_MODULE_FATAL**
Description	mcb - > freesize < = SIZEOF_PCB.
Extra Value	e0 = mcb¬freesize.
Description	mcb - > freesize < = stacksize element of pdb.
Extra Value	e0 = stacksize element of pdb.
	e1 = mcb¬freesize.

Code | Type	**KERNEL_EILL_VECTOR | SC_ERR_MODULE_FATAL**
Description	Parameter vector (interrupt process) of pdb not valid (>SC_MAX_INT_VECTOR).
Extra Value	e0 = parameter: vector.
	e1 = pdb.
	e2 = 9.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_MODULE_FATAL**
Description	Parameter slice (prioritized process) of pdb not valid.
Extra Value	e0 = parameter: slice.
Description	Parameter period (timer process) of pdb not valid.
Extra Value	e0 = parameter: period (timer process).
	e1 = pdb.
	e2 = 9.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_MODULE_FATAL**
Description	Parameter initial_dealy (timer process) of pdb not valid.
Extra Value	e0 = parameter: initial_delay.
	e1 = pdb.
	e2 = 10.

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Parameter type of pdb not valid.
Extra Value	e0 = parameter: type.
	e1 = pdb.
	e2 = 0.

Code | Type	**KERNEL_ENO_MORE_PROC | SC_ERR_MODULE_FATAL**
Description	Number of maximum processes reached.
Extra Value	e0 = No of process element of mcb.
	e1 = pdb.
	e2 = mcb.

Code | Type	**KERNEL_EILL_PROC_NAME | SC_ERR_MODULE_FATAL**
Description	Parameter name of pdb not valid.
Extra Value	e0 = pdb parameter: name.
	e1 = pdb.
	e2 = 1.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Module cb is not valid (mcb == SC_NIL).
Extra Value	e0 = mid.
	e1 = 0.
	e2 = 1.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Message which holds pcb or stack is not within current module.
Extra Value	e0 = Pointer to pcb or stack.
	e1 = Pointer to mcb of current module.
	e2 = Pointer to mcb of pcb/stack message buffer.

Code | Type	**KERNEL_EILL_PRIORITY | SC_ERR_MODULE_FATAL**
Description	Module cb is not valid (mcb == SC_NIL).
Extra Value	e0 = pdb parameter: priority.
	e1 = pdb.

	e2 = 10.

Code | Type	**KERNEL_EILL_STACKSIZE | SC_ERR_MODULE_FATAL**
Description	Parameter stack of pdb not valid.
Extra Value	e0 = pdb parameter: stack.
	e1 = pdb.
	e2 = 3.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter pdb not valid (!pdb
	pdb == SC_NIL).
Extra Value	e0 = pdb parameter: pdb.
	e1 = 0.
	e2 = 0.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter state not valid.
Extra Value	e0 = pdb parameter: state.
	e1 = 0.
	e2 = 1.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Illegal module ID. mid low 24 bits != 0 or e1 = 0 midx >= SC_MAX_MODULES.
Extra Value	e0 = mid.
	e1 = 0.
	e2 = 2.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter pcb of pdb not valid (==0).
Extra Value	e0 = pdb parameter: pcb.
	e1 = 0.
	e2 = 4.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter stack of pdb not valid (==0).
Extra Value	e0 = pdb parameter: stack.
	e1 = 0.

	e2 = 5.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter entry of pdb not valid.
Extra Value	e0 = pdb parameter: entry.
	e1 = pdb.
	e2 = 2.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter super not valid.
Extra Value	e0 = pdb parameter: super.
	e1 = pdb.
	e2 = 6.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter fpu not valid.
Extra Value	e0 = pdb parameter: fpu.
	e1 = pdb.
	e2 = 7.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Parameter spare not valid.
Extra Value	e0 = 0.
	e1 = pdb.
	e2 = 8.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Not enough space for pcb or stack in elected message buffer.
Extra Value	e0 = Pointer to pcb or stack.
	e1 = Pointer to mcb of current module.
	e2 = Size of PCB or stacksize.

Code | Type	**KERNEL_EALREADY_DEFINED | SC_ERR_MODULE_FATAL**
Description	Parameter vector (interrupt process) of pdb vector already defined.
Extra Value	e0 = pdb parameter: vector.
	e1 = pdb.
	e2 = 9.

3.67 sc_procDaemonRegister

3.67.1 Description

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.

Kernels: V1, V2 and V2INT

3.67.2 Syntax

```
int sc_procDaemonRegister(void);
```

3.67.3 Parameter

None.

3.67.4 Return Value

== 0	Process daemon could not be registered.
!= 0	If the process daemon was successfully registered.

3.67.5 Example

```
if ( sc_procDaemonRegister() == 0 ){
  kprintf(0,"Another procd running\n");
}
```

3.67.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Calling process is not a prioritized process.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Calling process is not in system module.

3.68 sc_procDaemonUnregister

3.68.1 Description

Unregister the current process as process-daemon.

Kernels: V1, V2 and V2INT

3.68.2 Syntax

```
int sc_procDaemonUnregister(void);
```

3.68.3 Parameter

None.

3.68.4 Return Value

== 0	If if the process daemon was not a process daemon.
!= 1	If the process daemon was successfully unregistered.

3.68.5 Example

```
TBD
```

3.68.6 Errors

None.

3.69 sc_procFlowSignatureGet

3.69.1 Description

Get the caller's process program flow signature.

Kernels: V2 and V2INT

3.69.2 Syntax

```
uint16_t sc_procFlowSignatureGet(void);
```

3.69.3 Parameter

None.

3.69.4 Return Value

Signature value.

3.69.5 Example

```
TBD
```

3.69.6 Errors

None.

3.70 sc_procFlowSignatureInit

3.70.1 Description

Initialize the caller's process program flow signature.

The process flow signature calculates a 16 bit CRC over given tokens. It uses the same polynomial as sc_miscCrc / sc_miscCrcContd.

Kernels: V2 and V2INT

3.70.2 Syntax

```
void sc_procFlowSignatureInit(
  uint16_t signature
);
```

3.70.3 Parameter

signature	Signature value.
	Initial value to be stored in the process control block of the callers process.

3.70.4 Return Value

None.

3.70.5 Example

```
TBD
```

3.70.6 Errors

None.

3.71 sc_procFlowSignatureUpdate

3.71.1 Description

Update the caller's program flow signature with the token given as parameter.

The result is returned.

Kernels: V2 and V2INT

3.71.2 Syntax

```
uint16_t sc_procFlowSignatureUpdate(
  uint32_t token
);
```

3.71.3 Parameter

token	Token value.
	Token value to calculate new CRC16.

3.71.4 Return Value

Signature value.

3.71.5 Example

```
TBD
```

3.71.6 Errors

None.

3.72 sc_procHookRegister

3.72.1 Description

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.

V2 Only: If enabled, the swap hook is also called when an interrupt is activated by hardware event.

Kernels: V2 and V2INT

3.72.2 Syntax

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

3.72.3 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.

3.72.4 Return Value

Function pointer to the previous process hook	If process hook was registered.
NULL	If no process hook was registered.

3.72.5 Example

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

3.72.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_SYSTEM_FATAL**
Description	Illegal process hook type.
Extra Value	e0 = type.

3.73 sc_procIdGet

3.73.1 Description

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. In case of an external process, the request is forwarded to the respective Connector process.

Kernels: V1, V2 and V2INT

3.73.2 Syntax

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

3.73.3 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.
	SC_ENDLESS_TMO	Waits for ever.

3.73.4 Return Value

Process ID of the found process	If the process was found within the tmo time period or empty.
Current process ID (process ID of the caller)	If parameter path is NULL.
SC_ILLEGAL_PID	If process was not found within the tmo time period.

3.73.5 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;
```

3.73.6 Example

```
sc_pid_t slave_pid;

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

3.73.7 Errors

Code | Type	**KERNEL_EILL_PROC_NAME | SC_ERR_PROCESS_FATAL**
Description	Illegal path.
Extra Value	e0 = pointer to path.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Process is PCB_TYPE_IDL.
Extra Value	e0 = process type.

3.74 sc_procIntCreate

3.74.1 Description

Request the kernel daemon to create an interrupt process. The standard kernel daemon (sc_kerneld) needs to be defined and started at system configuration. The interrupt process will be of type Sciopta. Interrupt processes of type Sciopta are handled by the kernel and may use (not blocking) system calls.

The process will be created within the callers module.

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

Kernels: V1

3.74.2 Syntax

```
sc_pid_t sc_procIntCreate(
  const char    *name,
  void (*entry) (int),
  sc_bufsize_t  stacksize,
  int           vector,
  sc_prio_t     prio,
  int           state,
  sc_poolid_t   plid
);
```

3.74.3 Parameter

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. Allowed 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).
vector	Interrupt Vector.
	Interrupt vector connected to the created interrupt process. This is CPU-dependent.
prio	N/A.
	Must be set to 0 (reserved for later use).
state	N/A.
	Must be set to 0 (reserved for later use).
plid	Pool ID.
	Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated.

3.74.4 Return Value

ID of the created process.

3.74.5 Example

```
hello_pid = sc_procIntCreate(
  /* process name */ "SCI_tick",
  /* process func */ (void (*) (void))SCI_tick,
  /* stacksize    */ 256,
  /* vector       */ 25,
  /* priority     */ 0,
  /* state        */ 0,
  /* pool-id      */ SC_DEFAULT_POOL
);
```

3.74.6 Errors

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EILL_PROC_NAME | SC_ERR_MODULE_FATAL**
Description	Parameter name not valid.
Extra Value	e0 = Pointer to process name.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Illegal process type.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_MODULE_FATAL**
Description	Illegal interrupt vector.
Extra Value	e0 = Interrupt vector.

3.75 sc_procIrqRegister

3.75.1 Description

Register an existing interrupt process for one more more other interrupt vectors.

Called from an interrupt process, registers the caller to be activated also if interrupt "vector" fires.

Intention of this system call is to allow handling of interrupts in the non-safe part of an application. To do so, a (safe) interrupt process may register for multiple interrupts and notify a (non-safe) priority process.

To minimize the overhead, the sc_msgAllocTx() system call can be used, which can carry enough information to handle the interrupt. Since allocation is done in the receivers pool, no copying is needed if the message is sent across module boundaries.

It is a fatal error, if "vector" is already used.

Kernels: V1, V2 and V2INT

3.75.2 Syntax

```
void sc_procIrqRegister(uint32_t vector );
```

3.75.3 Parameter

vector	Interrupt Vector.
	Must be in the range 0..SC_MAX_INT_VECTOR.

3.75.4 Return Value

None.

3.75.5 Example

```
SC_INT_PROCESS(SCI_canMBX0, src)
{
  if ( src == SC_PROC_WAKEUP_CREATE ){
    sc_procIrqRegister( CAN_MBX1_IRQ_VECTOR );
    sc_procIrqRegister( CAN_MBX2_IRQ_VECTOR );
  ...
}
```

3.75.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Not an interrupt process.
Extra Value	e0 = Process ID.

Code | Type	**KERNEL_EILL_VECTOR | SC_ERR_PROCESS_FATAL**
Description	Illegal interrupt vector.
Extra Value	e0 = Interrupt vector.

3.76 sc_procIrqUnregister

3.76.1 Description

Unregister previously registered interrupts.

Called from an interrupt process, removes it from being activated thru interrupt "vector".

It is a fatal error, if "vector" is not registered on the caller or if <vector> is the vector the interrupt was created for.

Kernels: V1, V2 and V2INT

3.76.2 Syntax

```
void sc_procIrqUnregister( uint32_t vector );
```

3.76.3 Parameter

vector	Interrupt Vector.
	Must be in the range 0..SC_MAX_INT_VECTOR.

3.76.4 Return Value

None.

3.76.5 Example

```
SC_INT_PROCESS( SCI_canMBX0, src )
{
  if ( src == SC_PROC_WAKEUP_KILL ){
    sc_procIrqUnregister( CAN_MBX1_IRQ_VECTOR );
    sc_procIrqUnregister( CAN_MBX2_IRQ_VECTOR );
  ...
}
```

3.76.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Not an interrupt process.
Extra Value	e0 = Process ID.

Code | Type	**KERNEL_EILL_VECTOR | SC_ERR_PROCESS_FATAL**
Description	Illegal interrupt vector.
Extra Value	e0 = Interrupt vector.

3.77 sc_procKill

3.77.1 Description

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.

Kernels: V1, V2 and V2INT

3.77.2 Syntax

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

3.77.3 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.

3.77.4 Return Value

None.

3.77.5 Example

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

3.77.6 Errors

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	pid == 0 or pid == SC_ILLEGAL_PID or mid too large or process index too large.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_WARNING**
Description	Process killed or pid not valid.
Extra Value	e0 = pid.

3.78 sc_procNameGet

3.78.1 Description

Get the full name of a process.

The name will be returned inside a SCIOPTA message 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 NULL. 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.

Kernels: V1, V2 and V2INT

3.78.2 Syntax

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

3.78.3 Parameter

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

3.78.4 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;
```

3.78.5 Example

```
sc_msg_t senderName;

senderName = sc_procNameGet( sender );
```

3.78.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	Illegal pid.

Extra Value	e0 = pid.

3.79 sc_procObserve

3.79.1 Description

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 supervising process.

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

Kernels: V1, V2 and V2INT

3.79.2 Syntax

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

3.79.3 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.

<pid>	Process ID of the process to be supervised.

3.79.4 Return Value

None.

3.79.5 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 );
```

3.79.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**

Description	Either pointer to message or pointer to message pointer are zero.

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	Illegal pid.
Extra Value	e0 = pid.

3.80 sc_procPathCheck

3.80.1 Description

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 and if it contains only valid character (A-Z,a-z,0-9 and underscore).

Kernels: V1, V2 and V2INT

3.80.2 Syntax

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

3.80.3 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.

3.80.4 Return Value

!=0	If the path is correct.
==0	If the path is wrong.

3.80.5 Example

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

3.80.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_SYSTEM_FATAL**
Description	Illegal path (pointer to path == 0).
Extra Value	e0 = pointer to path name.

3.81 sc_procPathGet

3.81.1 Description

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.

Kernels: V1, V2 and V2INT

3.81.2 Syntax

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

3.81.3 Parameter

pid	Process ID.	
	<pid>	Process ID of the process where the path is requested.
flags	**sc_procPathGet** flags.	
	!=0	The full path is returned: '//'<system_name>'/'<module_name>'/'process_name
	==0	The short path is returned: '/'<module_name>'/'process_name

3.81.4 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;
```

3.81.5 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";
```

```
}
```

3.81.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Illegal pid.
Extra Value	e0 = pid.

3.82 sc_procPpidGet

3.82.1 Description

Get the process ID of the parent (creator) of a process.

Kernels: V1, V2 and V2INT

3.82.2 Syntax

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

3.82.3 Parameter

pid	Process ID.	
	<pid>	Process ID of the process where its parent is requested.
	SC_CURRENT_PID	Current running (caller) process.

3.82.4 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.

3.82.5 Example

```
typedef struct key_s {
  uint8_t scan;
  uint8_t 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);
}
```

3.82.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Illegal pid.
Extra Value	e0 = pid.

3.83 sc_procPrioCreate

3.83.1 Description

Request the kernel daemon to create a prioritized 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.

Only supervisor-processes will be created (and no FPU).

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

Kernels: V1

3.83.2 Syntax

```
sc_pid_t sc_procPrioCreate(
  const char    *name,
  void (*entry) (void),
  sc_bufsize_t  stacksize,
  sc_ticks_t    slice,
  sc_prio_t     prio,
  int           state,
  sc_poolid_t   plid
);
```

3.83.3 Parameter

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. Allowed 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).	
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.	
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.	

3.83.4 Return Value

ID of the created process.

3.83.5 Example

```
hello_pid = sc_procPrioCreate(
  /* process name   */ "hello",
  /* process func   */ (void (*) (void))hello,
  /* stacksize      */ 512,
  /* slice          */ 0,
  /* priority       */ 16,
  /* run-state      */ SC_PDB_STATE_RUN,
  /* pool-id        */ SC_DEFAULT_POOL
);
```

3.83.6 Errors

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EILL_PROC_NAME | SC_ERR_MODULE_FATAL**
Description	Parameter name not valid.
Extra Value	e0 = Pointer to process name.

Code | Type	**KERNEL_EILL_PRIORITY | SC_ERR_MODULE_FATAL**
Description	Illegal priority (>=31).
Extra Value	e0 = Requested priority.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Illegal process type.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_MODULE_FATAL**
Description	Illegal slice value.
Extra Value	e0 = Slice.

Code | Type	**KERNEL_EILL_STACKSIZE | SC_ERR_MODULE_FATAL**
Description	Stack not valid.
Extra Value	e0 = Requested stacksize.

Code | Type	**KERNEL_EILL_MODULE | SC_ERR_MODULE_FATAL**
Description	Module cb is not valid.
Extra Value	e0 = Module ID.

Code | Type	**KERNEL_ENO_MORE_PROC | SC_ERR_MODULE_FATAL**

Description	Number of maximum processes reached.
Extra Value	e0 = No of processes.

Code | Type	**KERNEL_EOUT_OF_MEMORY | SC_ERR_MODULE_FATAL**
Description	Size does not fit into module memory..

3.84 sc_procPrioGet

3.84.1 Description

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.

Kernels: V1, V2 and V2INT

3.84.2 Syntax

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

3.84.3 Parameter

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

3.84.4 Return Value

Priority of any given process	If parameter pid was any process.
Priority of the callers process	If parameter pid was SC_CURRENT_PID.
32	If there was a warning of an invalid CONNECTOR process.

3.84.5 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);
```

3.84.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Illegal pid.

Extra Value	e0 = pid.

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = pid.
	e1 = Process type.

3.85 sc_procPrioSet

3.85.1 Description

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 will redefine the process and it becomes an idle process. The idle process will be called by the kernel if there are no processes ready.

Kernels: V1, V2 and V2INT

3.85.2 Syntax

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

3.85.3 Parameter

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

3.85.4 Return Value

None.

3.85.5 Example

```
// Switch caller to lowest-priority
sc_procPrioSet(31);
```

3.85.6 Errors

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_EILL_PRIORITY | SC_ERR_PROCESS_FATAL**
Description	Illegal priority. Priority == 32.
Extra Value	e0 = Process type.
	e1 = Module Priority.

Code | Type	**KERNEL_EILL_PRIORITY | SC_ERR_PROCESS_FATAL**
Description	Illegal priority. Priority > 32.
Extra Value	e0 = Process type.
	e1 = -1.

3.86 sc_procSchedLock

3.86.1 Description

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 locked.

Kernels: V1, V2 and V2INT

3.86.2 Syntax

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

3.86.3 Parameter

None

3.86.4 Return Value

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

3.86.5 Example

```
// count instances
sc_procSchedLock();
++counter;
sc_procSchedUnlock();
```

3.86.6 Errors

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process type.

3.87 sc_procSchedUnlock

3.87.1 Description

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 locked scheduler.

Kernels: V1, V2 and V2INT

3.87.2 Syntax

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

3.87.3 Parameter

None

3.87.4 Return Value

None.

3.87.5 Example

```
// count instances

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

3.87.6 Errors

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_ELOCKED | SC_ERR_MODULE_FATAL**
Description	Interrupts are locked.

Code | Type	**KERNEL_EUNLOCK_WO_LOCK | SC_ERR_MODULE_FATAL**
Description	Lockcounter == 0.

3.88 sc_procSliceGet

3.88.1 Description

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.

Kernels: V1, V2 and V2INT

3.88.2 Syntax

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

3.88.3 Parameter

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

3.88.4 Return Value

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

3.88.5 Example

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

3.88.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = pid.
	e1 = Process type.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Process/Module disappeared.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Illegal pid.
Extra Value	e0 = pid.

3.89 sc_procSliceSet

3.89.1 Description

Set the time slice of a prioritized or 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.

Kernels: V1, V2 and V2INT

3.89.2 Syntax

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

3.89.3 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.	
	!=0	0 is only allowed for priortized processes and disables the time-slice.

3.89.4 Return Value

None.

3.89.5 Example

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

3.89.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = pid.
	e1 = Process type.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Process/Module disappeared.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Illegal pid.
Extra Value	e0 = pid.

3.90 sc_procStart

3.90.1 Description

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.

Kernels: V1, V2 and V2INT

3.90.2 Syntax

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

3.90.3 Parameter

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

3.90.4 Return Value

None.

3.90.5 Example

```
sc_procStart(proc_A_pid);
```

3.90.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process or timer process.
Extra Value	e0 = pid.
	e1 = Process type.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Illegal pcb.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**

Description	Process is caller. Process is init process.
Extra Value	e0 = pid.

Code | Type	**KERNEL_ESTART_NOT_STOPPED | SC_ERR_MODULE_FATAL**
Description	Stop counter already 0.
Extra Value	e0 = pid.

3.91 sc_procStop

3.91.1 Description

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.

Kernels: V2 only: An interrupt will be invoked (wakeup) with SC_PROC_WAKEUP_STOP.

Kernels: V1: It is illegal to stop an interrupt process.

Kernels: V1, V2 and V2INT

3.91.2 Syntax

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

3.91.3 Parameter

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

3.91.4 Return Value

None.

3.91.5 Example

```
sc_procStop(SC_CURRENT_PID);
```

3.91.6 Errors

Code | Type	**KERNEL_EILL_PROC | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized or timer process.
Extra Value	e0 = pid.
	e1 = Process type.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Illegal pcb.

Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Process is caller. Process is init process.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_MODULE_FATAL**
Description	Stop counter already 0

3.92 sc_procTimCreate

3.92.1 Description

Request the kernel daemon to create a timer 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.

Kernels: V1

3.92.2 Syntax

```
sc_pid_t sc_procTimCreate(
  const char    *name,
  void (*entry) (int),
  sc_bufsize_t  stacksize,
  sc_ticks_t    period,
  sc_ticks_t    initdelay,
  int           state,
  sc_poolid_t   plid
);
```

3.92.3 Parameter

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. Allowed 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).	
period	Time period.	
	Period of time between calls to the timer process in ticks.	
initdelay	Initial time delay.	
	Initial delay in ticks before the first time call to the timer process.	
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.	

3.92.4 Return Value

ID of the created process.

3.92.5 Example

```
hello_pid = sc_procTimCreate(
  /* process name */ "SCI_tick",
  /* process func */ (void (*) (void))SCI_tick,
  /* stacksize    */ 256,
  /* period       */ 10,
  /* initdelay    */ 0,
  /* state        */ SC_PDB_STATE_RUN,
  /* pool-id      */ SC_DEFAULT_POOL
);
```

3.92.6 Errors

Code | Type	**KERNEL_ENO_KERNELD | SC_ERR_PROCESS_FATAL**
Description	There is no kernel daemon defined in the system.

Code | Type	**KERNEL_EILL_PROC_NAME | SC_ERR_MODULE_FATAL**
Description	Parameter name not valid.
Extra Value	e0 = Pointer to process name.

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Illegal process type.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_MODULE_FATAL**
Description	Illegal slice valueSlice.
Extra Value	e0 = Slice.

3.93 sc_procUnobserve

3.93.1 Description

Cancel an installed supervision of a process.

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

Kernels: V1, V2 and V2INT

3.93.2 Syntax

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

3.93.3 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.

3.93.4 Return Value

None.

3.93.5 Example

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

3.93.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Illegal pcb.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Process is caller. Process is init process.
Extra Value	e0 = pid.

3.94 sc_procVarDel

3.94.1 Description

Remove a process variable from the process variable data area.

Kernels: V1, V2 and V2INT

3.94.2 Syntax

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

3.94.3 Parameter

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

3.94.4 Return Value

==0	If the system call fails and the process variable could not be removed.
!=0	If the process variable was successfully removed.

3.94.5 Example

```
(void) sc_procVarDel(0xCAFE0001);
```

3.94.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	No process variable set.

3.95 sc_procVarGet

3.95.1 Description

Read a process variable.

Kernels: V1, V2 and V2INT

3.95.2 Syntax

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

3.95.3 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.

3.95.4 Return Value

==0	If the system call fails and the process variable could not be found and read.
!=0	If the process variable was successfully read.

3.95.5 Example

```
sc_var_t color;
if ( sc_procVarGet(0xC01103, &color) == 0 ){
  color = 10;
}
```

3.95.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	No procVar set or value == NULL.

3.96 sc_procVarInit

3.96.1 Description

Retup and initialize a process variable area.

Kernels: V1: The user should allocate a message that can hold (n+1) variable:

size = sizeof(sc_local_t)*(n+1);

Kernels: V2: The user should allocate a message for n variables plus controll block:

size = sizeof(sc_varpool_t)+sizeof(sc_local_t)*n;

Kernels: V1, V2 and V2INT

3.96.2 Syntax

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

3.96.3 Parameter

varpool	Process variable buffer.	
	<ptr>	Pointer to the message buffer holding the process variables.
	NULL or SC_NIL	Kernels V2 only: The kernel will allocate the message buffer.

3.96.4 Return Value

None.

3.96.5 Example

```
// Create var pool of 10 entries, let kernel allocate message
sc_procVarInit(NULL, 10);
```

3.96.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	No procVar set or value == NULL.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_PROCESS_FATAL**
Description	Process does not own the buffer.
Extra Value	e0 = owner.

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_PROCESS_FATAL**
Description	Size too small.
Extra Value	e0 = size.

Code | Type	**KERNEL_EALREADY_DEFINED | SC_ERR_PROCESS_FATAL**
Description	rocess variable already set.
Extra Value	e0 = Pointer to message buffer.

3.97 sc_procVarRm

3.97.1 Description

Remove a whole process variable area.

Kernels: V1, V2 and V2INT

3.97.2 Syntax

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

3.97.3 Parameter

None.

3.97.4 Return Value

Pointer to the message buffer holding the process variables.

3.97.5 Example

```
msg = sc_procVarRm();
sc_msgFree(&msg);
```

3.97.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	No procVar set or value == NULL.

3.98 sc_procVarSet

3.98.1 Description

Set or modify a process variable.

Kernels: V1, V2 and V2INT

3.98.2 Syntax

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

3.98.3 Parameter

tag	Process variable tag.
	User defined tag of the process variable. Valid values: All except 0.
value	Value of the process variable.

3.98.4 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.

3.98.5 Example

```
if ( sc_procVarSet(0xC01103, 12) == 0 ){
  kprintf(0,"Could not set color variable\n");
}
```

3.98.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	No procVar set.

3.99 sc_procVectorGet

3.99.1 Description

Get the interrupt vector of an interrupt process.

Kernels: V1, V2 and V2INT

3.99.2 Syntax

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

3.99.3 Parameter

pid	Process ID.
	Process ID of the interrupt process.

3.99.4 Return Value

Interrupt vector of the interrupt process

3.99.5 Example

```
kprintf(0,"Current vector %d\n", sc_procVectorGet(SC_CURRENT_PID));
```

3.99.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Caller is not an interrupt process.
Extra Value	e0 = pid.
	e1 = Process type.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Illegal pcb.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_FATAL**
Description	Process is caller. Process is init process.
Extra Value	e0 = pid.

3.100 sc_procWakeupEnable

3.100.1 Description

Enable the wakeup of a timer or interrupt process.

Kernels: V1, V2 and V2INT

Please Note: In V1 wakeup is active by default. In V2 and V2INT sc_procWakeupEnable must be called explicitely

3.100.2 Syntax

```
void sc_procWakeupEnable(void)
```

3.100.3 Parameter

None.

3.100.4 Return Value

None.

3.100.5 Example

```
---
```

3.100.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Caller is not an interrupt process or timer process.
Extra Value	e0 = Process type.

3.101 sc_procWakeupDisable

3.101.1 Description

Disable the wakeup of a timer or interrupt process.

Kernels: V1, V2 and V2INT

3.101.2 Syntax

```
void sc_procWakeupDisable(void)
```

3.101.3 Parameter

None.

3.101.4 Return Value

None.

3.101.5 Example

```
---
```

3.101.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_MODULE_FATAL**
Description	Caller is not an interrupt process or timer process.
Extra Value	e0 = Process type.

3.102 sc_procYield

3.102.1 Description

Yield the CPU to the next ready process within the current process's priority group.

Kernels: V1, V2 and V2INT

3.102.2 Syntax

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

3.102.3 Parameter

None.

3.102.4 Return Value

None.

3.102.5 Example

```
sc_procYield();
```

3.102.6 Errors

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process type.

3.103 sc_safe_charGet

3.103.1 Description

Get safe data of specific char types. The data is stored once in normal and in inverted format.
These functions are **not** part of the kernel.

Kernels: V2 and V2INT

3.103.2 Syntax

```
char sc_safe_charGet( sc_safe_char_t *si )
unsigned char sc_safe_ucharGet( sc_safe_uchar_t *si )
int8_t sc_safe_s8Get( sc_safe_s8_t *si )
uint8_t sc_safe_u8Get( sc_safe_u8_t *si )
```

3.103.3 Parameter

si	Address of safe data storage.
	Start address where value and its inverted version are stored.

3.103.4 Return Value

None.

3.103.5 Example

```
---
```

3.103.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter si not valid (== 0).

3.104 sc_safe_charSet

3.104.1 Description

Set safe data of specific char types at a given address in memory. The data is stored once in normal and in inverted format.
These functions are **not** part of the kernel.

Kernels: V2 and V2INT

3.104.2 Syntax

```
void sc_safe_charSet( sc_safe_char_t *si, char v)
void sc_safe_ucharSet( sc_safe_uchar_t *si, unsigned char v)
void sc_safe_s8Set( sc_safe_s8_t *si, int8_t v)
void sc_safe_u8Set( sc_safe_u8_t *si, uint8_t v)
```

3.104.3 Parameter

si	Address of safe data storage.
	Start address where value and its inverted version are stored.
v	Safe data.
	Safe data of specific char types to be stored.

3.104.4 Return Value

None.

3.104.5 Example

```
---
```

3.104.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter si not valid (== 0).

3.105 sc_safe_<type>Get

3.105.1 Description

Get safe data of specific types. The data is stored once in normal and in inverted format.
These functions are **not** part of the kernel.

Kernels: V2 and V2INT

3.105.2 Syntax

```
int sc_safe_intGet( sc_safe_int_t *si );
unsigned int sc_safe_intGet( sc_safe_uint_t *si );
long sc_safe_intGet( sc_safe_long_t *si );
unsigned long sc_safe_intGet( sc_safe_ulong_t *si );
int32_t sc_safe_intGet( sc_safe_s32_t *si );
uint32_t sc_safe_intGet( sc_safe_u32_t *si );
sc_pool_cb_t *sc_safe_poolcb_ptrGet( sc_safe_poolcb_ptr_t *si );
sc_pcb_t *sc_safe_pcbptrGet( sc_safe_pcbptr_t *si );
sc_module_cb_t *sc_safe_mcbptrGet( sc_safe_mcbptr_t *si );
sc_modulesize_t sc_safe_modulesizeGet( sc_safe_modulesize_t*si );
sc_ticks_t sc_safe_ticksGet( sc_safe_ticks_t *si );
sc_time_t sc_safe_timeGet( sc_safe_time_t *si );
sc_pid_t sc_safe_pidGet( sc_safe_pid_t *si );
sc_mid_t sc_safe_midGet( sc_safe_mid_t *si );
sc_errcode_t sc_safe_errcodeGet( sc_safe_errcode_t *si );
void *sc_safe_voidptrGet( sc_safe_voidptr_t *si );
sc_triggerval_t sc_safe_triggervalGet( sc_safe_triggerval_t *si );
sc_plsize_t sc_safe_plsizeGet( sc_safe_plsize_t *si );
sc_poolid_t sc_safe_poolidGet( sc_safe_poolid_t *si );
sc_bufsize_t sc_safe_bufsizeGet( sc_safe_bufsize_t *si );
sc_prio_t sc_safe_prioGet( sc_safe_prio_t *si );
```

3.105.3 Parameter

si	Address of safe data storage.
	Start address where value and its inverted version are stored.

3.105.4 Return Value

None.

3.105.5 Example

```
---
```

3.105.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter si not valid (== 0).

3.106 sc_safe_<type>Set

3.106.1 Description

Set safe data of specific types at a given address in memory. The data is stored once in normal and in inverted format.
These functions are **not** part of the kernel.

Kernels: V2 and V2INT

3.106.2 Syntax

```
void sc_safe_intSet( sc_safe_int_t *si, int v );
void sc_safe_uintSet( sc_safe_uint_t *si, unsigned int v );
void sc_safe_longSet( sc_safe_long_t *si, long v );
void sc_safe_ulongSet( sc_safe_ulong_t *si, unsigned long v );
void sc_safe_s32Set( sc_safe_s32_t *si, int32_t v );
void sc_safe_u32Set( sc_safe_u32_t *si, uint32_t v );
void sc_safe_poolcb_ptrSet( sc_safe_poolcb_ptr_t *si, sc_pool_cb_t *v );
void sc_safe_pcbptrSet( sc_safe_pcbptr_t *si, sc_pcb_t *v );
void sc_safe_mcbptrSet( sc_safe_mcbptr_t *si, sc_module_cb_t *v );
void sc_safe_modulesizeSet( sc_safe_modulesize_t *si, sc_modulesize_t v );
void sc_safe_ticksSet( sc_safe_ticks_t *si, sc_ticks_t v );
void sc_safe_timeSet( sc_safe_time_t *si, sc_time_t v );
void sc_safe_pidSet( sc_safe_pid_t *si, sc_pid_t v );
void sc_safe_midSet( sc_safe_mid_t *si, sc_mid_t v );
void sc_safe_errcodeSet( sc_safe_errcode_t *si, sc_errcode_t v );
void sc_safe_voidptrSet( sc_safe_voidptr_t *si, void *v );
void sc_safe_triggervalSet( sc_safe_triggerval_t *si, sc_triggerval_t v );
void sc_safe_plsizeSet( sc_safe_plsize_t *si, sc_plsize_t v );
void sc_safe_poolidSet( sc_safe_poolid_t *si, sc_poolid_t v );
void sc_safe_bufsizeSet( sc_safe_bufsize_t *si, sc_bufsize_t v );
void sc_safe_prioSet( sc_safe_prio_t *si, sc_prio_t v );
```

3.106.3 Parameter

si	Address of safe data storage.
	Start address where value and its inverted version are stored.
v	Safe data.
	Safe data of specific types to be stored.

3.106.4 Return Value

None.

3.106.5 Example

```
---
```

3.106.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter si not valid (== 0).

3.107 sc_safe_shortGet

3.107.1 Description

Get safe data of specific short types. The data is stored once in normal and in inverted format.
These functions are **not** part of the kernel.

Kernels: V2 and V2INT

3.107.2 Syntax

```
short sc_safe_shortGet( sc_safe_short_t *si )
unsigned short sc_safe_ushortGet( sc_safe_ushort_t *si )
int16_t sc_safe_s16Get( sc_safe_s16_t *si )
uint16_t sc_safe_u16Get( sc_safe_u16_t *si )
```

3.107.3 Parameter

si	Address of safe data storage.
	Start address where value and its inverted version are stored.

3.107.4 Return Value

Retrieved safe data of specific short types.

3.107.5 Example

```
---
```

3.107.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter si not valid (== 0).

3.108 sc_safe_shortSet

3.108.1 Description

Set safe data of specific short types at a given address in memory. The data is stored once in normal and in inverted format.
These functions are **not** part of the kernel.

Kernels: V2 and V2INT

3.108.2 Syntax

```
void sc_safe_shortSet( sc_safe_short_t *si, short v )
void sc_safe_ushortSet( sc_safe_ushort_t *si, unsigned short v )
void sc_safe_s16Set( sc_safe_s16_t *si, int16_t v )
void sc_safe_u16Set( sc_safe_u16_t *si, uint16_t v )
```

3.108.3 Parameter

si	Address of safe data storage.
	Start address where value and its inverted version are stored.
v	Safe data.

3.108.4 Return Value

None.

3.108.5 Example

```
---
```

3.108.6 Errors

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Parameter si not valid (== 0).

3.109 sc_sleep

3.109.1 Description

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.

Kernels: V1, V2 and V2INT

3.109.2 Syntax

```
Kernels V1:
void sc_sleep( sc_ticks_t tmo );

Kernels V2:
sc_time_t sc_sleep( sc_ticks_t tmo );
```

3.109.3 Parameter

tmo	Timeout.
	Number of system ticks to wait.

3.109.4 Return Value

Kernels: V2 only: Activation time. The absolute time (tick counter) value when the calling process became ready.

3.109.5 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 );
}
```

3.109.6 Errors

Code | Type	**KERNEL_ELOCKED | SC_ERR_MODULE_FATAL**
Description	Scheduler is locked.

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_MODULE_FATAL**
Description	Caller is not a prioritized process.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_MODULE_FATAL**
Description	Illegal timeout value.
Extra Value	e0 = tmo.

3.110 sc_tick

3.110.1 Description

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.

The user shall setup an periodic interrupt process and call sc_tick. The lenght of the period shall be published by the sc_tickLength system call. sc_tick must be called explicitly.

This system call is only allowed in hardware activated interrupt processes.

Note: A SCIOPTA system can be used tickless. In this case, sc_tick will not be called. Consequently, no timeouts are allowed.

Kernels: V1, V2 and V2INT

3.110.2 Syntax

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

3.110.3 Parameter

None.

3.110.4 Return Value

None.

3.110.5 Example

```
SC_INT_PROCESS(sysTick, src)
{
  if ( src == SC_PROC_WAKEUP_HARDWARE ){
    sc_tick();

       /* Handle timer irq */
  }
}
```

3.110.6 Errors

None.

3.111 sc_tickActivationGet

3.111.1 Description

Returns the tick time of last activation of the calling process.

Kernels: V2 and V2INT

3.111.2 Syntax

```
sc_time_t sc_tickActivationGet(void);
```

3.111.3 Parameter

None.

3.111.4 Return Value

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

3.111.5 Example

```
for(;;){
  msg = sc_msgRx(SC_ENDLESS_TMO, SC_MSGRX_ALL, SC_MSGRX_MSGID);
  if ( (sc_tickGet() - sc_tickActivationGet()) > 10 ){
    kprintf(0,"Time exceeded before I got the CPU\n");
  }
  ...
}
```

3.111.6 Errors

None.

3.112 sc_tickGet

3.112.1 Description

Get the actual kernel tick counter value. The number of system ticks from the system start are returned.

Kernels: V1, V2 and V2INT

3.112.2 Syntax

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

3.112.3 Parameter

None.

3.112.4 Return Value

Current value of the tick timer.

3.112.5 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));
```

3.112.6 Errors

None.

3.113 sc_tickGet64

3.113.1 Description

Return current tick-value

Kernels: V2

3.113.2 Syntax

```
sc_time64_t sc_tickGet64(void);
```

3.113.3 Parameter

None.

3.113.4 Return Value

current tick.

3.113.5 Example

```
sc_time64_t t64;
  t64 = sc_tickGet64(void);
```

3.113.6 Errors

None.

3.114 sc_tickLength

3.114.1 Description

Set or get the current system tick length in microseconds.

Note: This value is informational only and has no impact on the kernel behaviour like scheduling. But the function sc_tickMs2Tick and sc_tickTick2Ms rely on it.

Kernels: V1, V2 and V2INT

3.114.2 Syntax

```
uint32_t sc_tickLength( uint32_t ticklength );
```

3.114.3 Parameter

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

3.114.4 Return Value

Tick length in microseconds.

3.114.5 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" );
```

3.114.6 Errors

None.

3.115 sc_tickMs2Tick

3.115.1 Description

Convert a time from milliseconds into system ticks.

Note: This function may round input values larger then UINT32_MAX/1000.

Kernels: V1, V2 and V2INT

3.115.2 Syntax

```
sc_time_t sc_tickMs2Tick( uint32_t ms );
```

3.115.3 Parameter

ms	Time in milliseconds.

3.115.4 Return Value

Time in system ticks.

3.115.5 Example

```
int tmo = 1000;

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

3.115.6 Errors

None.

3.116 sc_tickTick2Ms

3.116.1 Description

Convert a time from system ticks into milliseconds.

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

Note: This function may round input values larger then UINT32_MAX/1000.

Kernels: V1, V2 and V2INT

3.116.2 Syntax

```
uint32_t sc_tickTick2Ms( sc_ticks_t t );
```

3.116.3 Parameter

t	Time in system ticks.

3.116.4 Return Value

Time in milliseconds.

3.116.5 Example

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

t2 = sc_tickTick2Ms( t1-t0 );
```

3.116.6 Errors

None.

3.117 sc_tmoAdd

3.117.1 Description

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.

Kernels: V1, V2 and V2INT

3.117.2 Syntax

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

3.117.3 Parameter

tmo	Time Out.
	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.

3.117.4 Return Value

Timeout ID.

3.117.5 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 );
```

3.117.6 Errors

Code | Type	**KERNEL_EPROC_NOT_PRIO | SC_ERR_PROCESS_FATAL**
Description	Caller is not a prioritized process.
Extra Value	e0 = Process Type.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_PROCESS_FATAL**
Description	Illegal timeout value.
Extra Value	e0 = tmo.

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.

Code | Type	**KERNEL_EALREADY_DEFINED | SC_ERR_PROCESS_FATAL**
Description	Message is already a timeout message.
Extra Value	e0 = Pointer to the message.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = Owner.
	e1 = Pointer to message.

3.118 sc_tmoRm

3.118.1 Description

Remove a timeout before it is expired.

The user can cancel the timeout even if it has expired but the timeout-message was not yet reveived.

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

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

Note: It is recommend to set the timeout ID variable to zero if the timeout message was received.

Kernels: V1, V2 and V2INT

3.118.2 Syntax

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

3.118.3 Parameter

tid	Timeout ID.
	Pointer to timeout ID which was given when the timeout was registered by the sc_tmoAdd call.

3.118.4 Return Value

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

3.118.5 Example

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

3.118.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_PROCESS_FATAL**
Description	Timeout expired, but not received (not in the queue)

Code | Type	**KERNEL_EILL_PARAMETER | SC_ERR_PROCESS_FATAL**
Description	Timeout ID is already cleared.
Extra Value	e0 = Pointer to timeout ID.

Code | Type	**KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL**
Description	Either pointer to message or pointer to message pointer are zero.

Code | Type	**KERNEL_ENOT_OWNER | SC_ERR_MODULE_FATAL**
Description	Process does not own the message.
Extra Value	e0 = pid of the owner.

3.119 sc_trigger

3.119.1 Description

Activate a process trigger.

The trigger value of the addressed process's 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.

Kernels: V1, V2 and V2INT

3.119.2 Syntax

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

3.119.3 Parameter

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

3.119.4 Return Value

None.

3.119.5 Example

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

3.119.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Illegal process type.
Extra Value	e0 = pid.
	e1 = Process type.

Code | Type	**KERNEL_EILL_PID | SC_ERR_MODULE_WARNING**
Description	Process disappeared.
Extra Value	e0 = pid.

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	Process is an init or external process.
Extra Value	e0 = pid.

3.120 sc_triggerValueGet

3.120.1 Description

Get the value of a process trigger.

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

Kernels: V1, V2 and V2INT

3.120.2 Syntax

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

3.120.3 Parameter

pid	Process ID.
	ID of the process which trigger is returned.

3.120.4 Return Value

Trigger value.

INT_MAX if no valid process.

3.120.5 Example

```
sc_pid_t slave_pid;
sc_triggerval_t slavetrig;

slave_pid = sc_procIdGet("slave", SC_NO_TMO);

slave_trig = sc_triggerValueGet(slave_pid);
```

3.120.6 Errors

Code | Type	**KERNEL_EILL_PID | SC_ERR_PROCESS_FATAL**
Description	Process is an init or external process.
Extra Value	e0 = pid.

3.121 sc_triggerValueSet

3.121.1 Description

Set the value of a process trigger to any positive value.

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

Kernels: V1, V2 and V2INT

3.121.2 Syntax

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

3.121.3 Parameter

value	Trigger value.
	The new trigger value to be stored.

3.121.4 Return Value

None.

3.121.5 Example

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

3.121.6 Errors

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_PROCESS_FATAL**
Description	Illegal trigger value.
Extra Value	e0 = Trigger value.

3.122 sc_triggerWait

3.122.1 Description

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.

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 ready in again and the trigger value will be incremented by the amount it has been decrement before.

Kernels: V2 only: 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.

Kernels: V1, V2 and V2INT

3.122.2 Syntax

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

3.122.3 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.

3.122.4 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.

3.122.5 Example

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

3.122.6 Errors

Code | Type	**KERNEL_EILL_PROCTYPE | SC_ERR_PROCESS_FATAL**
Description	Illegal process type.
Extra Value	e0 = Process type.

Code | Type	**KERNEL_ELOCKED | SC_ERR_MODULE_FATAL**
Description	Interrupts and/or scheduler are/is locked.
Extra Value	e0 = Lock counter value or -1 if interrupt are locked..

Code | Type	**KERNEL_EILL_VALUE | SC_ERR_PROCESS_FATAL**
Description	Illegal trigger decrement value (¬0).
Extra Value	e0 = Decrement value.

Code | Type	**KERNEL_EILL_SLICE | SC_ERR_PROCESS_FATAL**
Description	tmo value not valid.
Extra Value	e0 = tmo value.

3.123 sciopta_end

3.123.1 Description

Ends a SCIOPTA Simulator application.

The control is returned to the Windows operating system.

This system call is only available in the SCIOPTA V1 Simulator.

Kernels: V1

3.123.2 Syntax

```
void sciopta_end(void);
```

3.123.3 Parameter

None.

3.123.4 Return Value

None.

3.123.5 Example

```
/* timeout expired */
if ( (t - ts) > 10 || result == WAIT_TIMEOUT ){
  sciopta_end();
  WaitForSingleObject(sciopta, INFINITE);
  break;
}
```

3.123.6 Errors

None.

3.124 sciopta_start

3.124.1 Description

Starts a SCIOPTA Kernel Simulator application. It must be placed in the startup code of your Windows application.

This system call is only available in the SCIOPTA V1 Simulator.

Kernels: V1

3.124.2 Syntax

```
int sciopta_start(
  char           *cmdline,
  sciopta_t      *psciopta,
  sc_pcb_t       **connectors,
  sc_pcb_t       **pirq_vectors,
  sc_module_cb_t **modules,
  void (*start_hook)(void),
  void (*TargetSetup)(void),
  void (*sysPutchar)(int ),
  void (*idle_hook)(void)
);
```

3.124.3 Parameter

cmdline	Command line of the application. This parameter is not used so far.
psciopta	Pointer to the SCIOPTA kernel control block.
connectors	Pointer to the connector PCB pointer array.
pirq_vectors	Pointer to the interrupt PCB pointer array.
modules	Pointer to the module CB pointer array.
start_hook	Function pointer to the start_hook.
TargetSetup	Function pointer to the target (system) setup function.
sysPutchar	Function pointer to a put-character function. This is used by the kernel internal debug functions.
idle_hook	Function pointer to the idle_hook.

3.124.4 Return Value

None.

3.124.5 Example

```
/* from sconf.c */
extern sciopta_t sciopta;
extern void TargetSetup(void);
extern sc_module_cb_t * sc_modules[SC_MAX_MODULE];
extern sc_pcb_t * sc_connectors[1]; /* dummy */
extern uint32_t sc_globalFlowSignatures[1]; /* dummy */
extern sc_pcb_t * sc_irq_vectors[SC_MAX_INT_VECTORS];
extern dbl_t sc_tmoLists[1];     /* dummy */
extern sc_errMsg_t sc_errMsg; /* Last error information */
extern const uint32_t scioptaConfig[5];

/* prototypes */
void start_hook(void);
void sys_putchar(int c);


char * cmdline = (char *)para;
```

```
err = sciopta_start(cmdline,
                    &sciopta,
                    sc_connectors,
                    sc_irq_vectors,
                    sc_modules,
                    sc_tmoLists,
                    sc_globalFlowSignatures,
                    scioptaConfig,
                    start_hook,
                    TargetSetup,
                    sys_putchar,
                    &sc_errMsg);
```

3.124.6 Errors

None.

3.125 sc_sysIrqDispatcher

3.125.1 Description

The hardware-level interrupt handling is not part of the kernel. The BSP shall call this kernel function to activate the respective interrupt process.
SCIOPTA interrupt vector numbers correspondent to the parameter **<vector>** if passed.

3.125.2 Syntax

Kernels: V1,V2, INT

Cortex-M

The kernel reads the NVIC register **<IPSR>** to determine the vector. If this is not wanted, call **sc_sysVirtualIrqDispatcher()** instead.

```
void sc_sysIrqDispatcher(void);
void sc_sysVirtualIrqDispatcher(uint32_t vector);
```

ARMv4T, ARMv5TE, ARMv7-R/A

The user interrupt handler shall determine the interrupt vector and call the kernel function with the SCIOPTA vector. If the SCIOPTA vector is different from the hardware vector, this vector may be passed as second parameter. This parameter is provided to the interrupt process as <org_vector>.

The CPU must be in SYS mode and MPU or MMU must be disabled when calling.

```
void sc_sysIrqDispatcher(uint32_t vector);
void sc_sysIrqDispatcher(uint32_t vector, uint32_t org_vector);
```

An interrupt process may be defined either:

```
SC_INT_PROCESS(<function>, src);
```

or

```
SC_INT_PROCESS_EX(<function>, src, org_vector);
```

Kernels: V2, INT

ARM64

One of these functions depending on the **GIC** used (see SoC manual) shall be called directly from IRQ EL0 and IRQ ELx vectors.
The kernel will retrieve the vector directly from the **GIC** registers.
The *_wsh functions call the IRQ-swap hook if registered.
The function will not return to the caller!

The CPU must be in EL1 when calling.

```
__noreturn void sc_sysIrqDispatcher_gic500_wsh(void);
__noreturn void sc_sysIrqDispatcher_gic500(void);
__noreturn void sc_sysIrqDispatcher_gic400_wsh(void);
__noreturn void sc_sysIrqDispatcher_gic400(void);
```

AURIX

Place a call to either function on entry 510 in the INTTAB (see SoC manual).
The kernel will determine the actual vector by reading **ICR** and call the respective interrupt process.
The *_wsh function calls the IRQ-swap hook if registered.
The function will not return.

```
void sc_sysIrqDispatcher_wsh(void)
void sc_sysIrqDispatcher(void)
```

Blackfin

Call either function from the hardware interrupt handler.
The *_wsh function calls the IRQ-swap hook if registered.
The function will not return.

```
__noreturn void sc_sysIrqDispatcher_wsh(uint32_t vector)
__noreturn void sc_sysIrqDispatcher(uint32_t vector)
```

RX

Call this function from the hardware interrupt handler.
The parameter shall be the vector number **multiplied by 4**.
The function will not return.

```
__noreturn void sc_sysIrqDispatcher(uint32_t vectorBy4)
```

PowerPC

Kernels: INT

Call this function from the hardware exception handler.
The function will not return.

```
__noreturn void sc_sysIrqDispatcher(uint32_t vector);
```

3.126 sc_sysIrqEpilogue

3.126.1 Description

ARMv4T, ARMv5TE, ARMv7-R/A

This function shall be called from BSP code after all pending interrupts were handled.
The function will not return.

The CPU must be in SYS mode and MPU or MMU must be disabled when calling.

```
void sc_sysIrqEpilogue(void);
```

Kernels: V1, V2

3.127 sc_sysSWI

3.127.1 Description

ARMv4T, ARMv5TE, ARMv7-R/A

This is the SWI (software interrupt) function to handle trap interface.
It must be called from vector 2.
The function will not return.

Kernels: V1, V2

```
void sc_sysSWI(void);
```

3.128 sc_sysSVC

3.128.1 Description

Cortex-M

This is the SVC (supervisor call) function to handle trap interface.
Place in the vector table on vector 11.
The function will not return.

Kernels: V1, V2

```
void sc_sysSVC(void);
```

4 Kernel Error Reference

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 SCIOPTA Architecture Manual chapter “Error Handling” for more 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.

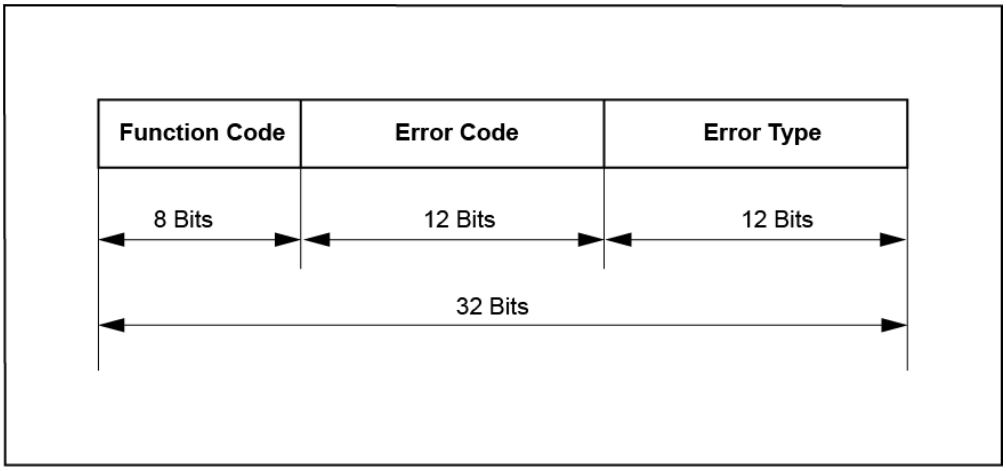

Figure 1. 32-bit Error Word

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.
- **Kernel V1** `<install_folder>\sciopta\<version>\include\kernel\`
- **Kernel V2** `<install_folder>\sciopta\<version>\include\kernel2\`
- **Kernel INT** `<install_folder>\sciopta\<version>\include\ikernel\`

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

4.3 Function Codes (Kernels V1)

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_msgSizeGet
SC_MSGSIZESET	0x06	sc_msgSizeSet
SC_MSGOWNERGET	0x07	sc_msgOwnerGet
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_POOLCREATE	0x10	sc_poolCreate
SC_POOLRESET	0x11	sc_poolReset
SC_POOLKILL	0x12	sc_poolKill
SC_POOLINFO	0x13	sc_poolInfo
SC_POOLDEFAULT	0x14	sc_poolDefault
SC_POOLIDGET	0x15	sc_poolIdGet
SC_SYSPOOLKILL	0x16	sc_sysPoolKill (Internal)
SC_POOLHOOKREGISTER	0x17	sc_poolHookRegister
SC_MISCERRORHOOKREGISTER	0x18	sc_miscErrorHookRegister
SC_MISCKERNELDREGISTER	0x19	sc_miscKerneldRegister
SC_MISCCRCCONTD	0x1A	sc_miscCrcContd
SC_MISCCRC	0x1B	sc_miscCrc
SC_MISCERRNOSET	0x1C	sc_miscErrnoSet
SC_MISCERRNOGET	0x1D	sc_miscErrnoGet
SC_PROCWAKEUPENABLE	0x1E	sc_procWakeupEnable
SC_PROCWAKEUPDISABLE	0x1F	sc_procWakeupDisable
SC_PROCPRIOGET	0x20	sc_procPrioGet

Name	Number	Error Source
SC_PROCPRIOSET	0x21	sc_procPrioSet
SC_PROCSLICEGET	0x22	sc_procSliceGet
SC_PROCSLICESET	0x23	sc_procSliceSet
SC_PROCIDGET	0x24	sc_procIdGet
SC_PROCPPIDGET	0x25	sc_procPpidGet
SC_PROCNAMEGET	0x26	sc_procNameGet
SC_PROCSTART	0x27	sc_procStart
SC_PROCSTOP	0x28	sc_procStop
SC_PROCVARINIT	0x29	sc_procVarInit
SC_PROCSCHEDUNLOCK	0x2A	sc_procSchedUnlock
SC_PROCPRIOCREATESTATIC	0x2B	sc_procPrioCreateStatic (Internal)
SC_PROCINTCREATESTATIC	0x2C	sc_procIntCreateStatic (Internal)
SC_PROCTIMCREATESTATIC	0x2D	sc_procTimCreateStatic (Internal)
SC_PROCUSRINTCREATESTATIC	0x2E	sc_procUsrIntCreateStatic (Internal)
SC_PROCPRIOCREATE	0x2F	sc_procPrioCreate
SC_PROCINTCREATE	0x30	sc_procIntCreate
SC_PROCTIMCREATE	0x31	sc_procTimCreate
SC_PROCUSRINTCREATE	0x32	sc_procUsrIntCreate
SC_PROCKILL	0x33	sc_procKill
SC_PROCYIELD	0x34	sc_procYield
SC_PROCOBSERVE	0x35	sc_procObserve
SC_SYSPROCCREATE	0x36	sc_sysProcCreate (Internal)
SC_PROCSCHEDLOCK	0x37	sc_procSchedLock
SC_PROCVARGET	0x38	sc_procVarGet
SC_PROCVARSET	0x39	sc_procVarSet
SC_PROCVARDEL	0x3A	sc_procVarDel
SC_PROCVARRM	0x3B	sc_procVarRm
SC_PROCUNOBSERVE	0x3C	sc_procUnobserve
SC_PROCPATHGET	0x3D	sc_procPathGet
SC_PROCPATHCHECK	0x3E	sc_procPathCheck
SC_PROCHOOKREGISTER	0x3F	sc_procHookRegister
SC_MODULECREATE	0x40	sc_moduleCreate
SC_MODULEKILL	0x41	sc_moduleKill

Name	Number	Error Source
SC_MODULENAMEGET	0x42	sc_moduleNameGet
SC_MODULEIDGET	0x43	sc_moduleIdGet
SC_MODULEINFO	0x44	sc_moduleInfo
SC_MODULEPRIOSET	0x45	sc_modulePrioSet (Internal)
SC_MODULEPRIOGET	0x46	sc_modulePrioGet
SC_MODULEFRIENDADD	0x47	sc_moduleFriendAdd
SC_MODULEFRIENDRM	0x48	sc_moduleFriendRm
SC_MODULEFRIENDGET	0x49	sc_moduleFriendGet
SC_MODULEFRIENDNONE	0x4A	sc_moduleFriendNone
SC_MODULEFRIENDALL	0x4B	sc_moduleFriendAll
SC_PROCIRQREGISTER	0x4C	sc_procIrqRegister
SC_PROCIRQUNREGISTER	0x4D	sc_procIrqUnregister
SC_PROCDAEMONUNREGISTER	0x4E	sc_procDaemonUnregister
SC_PROCDAEMONREGISTER	0x4F	sc_procDaemonRegister
SC_TRIGGERVALUESET	0x50	sc_triggerValueSet
SC_TRIGGERVALUEGET	0x51	sc_triggerValueGet
SC_TRIGGER	0x52	sc_trigger
SC_TRIGGERWAIT	0x53	sc_triggerWait
SC_SYSERROR	0x54	sc_sysError (Internal)
SC_MISCERROR	0x55	sc_miscError
SC_MODULECREATE2	0x56	sc_moduleCreate2
SC_TICK	0x57	sc_tick
SC_TMOADD	0x58	sc_tmoAdd
SC_TMO	0x59	sc_tmo (Internal)
SC_SLEEP	0x5A	sc_sleep
SC_TMORM	0x5B	sc_tmoRm
SC_TICKGET	0x5C	sc_tickGet
SC_TICKLENGTH	0x5D	sc_tickLength
SC_TICKMS2TICK	0x5E	sc_tickMs2Tick
SC_TICKTICK2MS	0x5F	sc_tickTick2Ms
SC_CONNECTORREGISTER	0x60	sc_connectorRegister
SC_CONNECTORUNREGISTER	0x61	sc_connectorUnregister
	0x62	<dispatcher>

Name	Number	Error Source
	0x63	reserved
	0x64	reserved
SC_MSGALLOCTX	0x65	sc_msgAllocTx
SC_CONNECTORREMOTE2LOCAL	0x66	sc_connectorRemote2local (Internal)

4.4 Function Codes (Kernels V2 and V2INT)

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_msgSizeGet
SC_MSGSIZESET	0x06	sc_msgSizeSet
SC_MSGOWNERGET	0x07	sc_msgOwnerGet
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_MSGFIND	0x12	sc_msgFind
SC_MSGALLOCTX	0x13	sc_msgAllocTx
SC_MSGDATACRCSET	0x14	sc_msgDataCrcSet
SC_MSGDATACRCGET	0x15	sc_msgDataCrcGet
SC_MSGDATACRCDIS	0x16	sc_msgDataCrcDis
SC_MSGFLOWSIGNATUREUPDATE	0x17	sc_msgFlowSignatureUpdate
SC_POOLCREATE	0x18	sc_poolCreate
SC_POOLRESET	0x19	sc_poolReset
SC_POOLKILL	0x1A	sc_poolKill
SC_POOLINFO	0x1B	sc_poolInfo
SC_POOLDEFAULT	0x1C	sc_poolDefault
SC_POOLIDGET	0x1D	sc_poolIdGet
SC_POOLHOOKREGISTER	0x1E	sc_poolHookRegister
SC_POOLCBCHK	0x1F	sc_poolCBChk
SC_MISCERRORHOOKREGISTER	0x20	sc_miscErrorHookRegister

Name	Number	Error Source
SC_MISCKERNELDREGISTER	0x21	sc_miscKerneldRegister
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
SC_MISCCRCSTRING	0x29	sc_miscCrcString
	0x2A	reserved
	0x2B	reserved
	0x2C	reserved
SC_MISCFLOWSIGNATUREINIT	0x2D	sc_miscFlowSignatureInit
SC_MISCFLOWSIGNATUREUPDATE	0x2E	sc_miscFlowSignatureUpdate
SC_MISCFLOWSIGNATUREGET	0x2F	sc_miscFlowSignatureGet
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_procPathCheck
SC_PROCIRQREGISTER	0x3D	sc_procIrqRegister
SC_PROCIRQUNREGISTER	0x3E	sc_procIrqUnregister
	0x3F	reserved
SC_PROCSTART	0x40	sc_procStart
SC_PROCSTOP	0x41	sc_procStop

Name	Number	Error Source
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_PROCDAEMONUNREGISTER	0x50	sc_procDaemonUnregister
SC_PROCDAEMONREGISTER	0x51	sc_procDaemonRegister
	0x52	
	0x53	reserved
	0x54	
	0x55	
	0x56	
	0x57	
	0x58	
	0x59	
	0x5A	
	0x5B	
SC_PROCCBCHK	0x5C	sc_procCBChk
SC_PROCFLOWSIGNATUREINIT	0x5D	sc_procFlowSignatureInit
SC_PROCFLOWSIGNATUREUPDATE	0x5E	sc_procFlowSignatureUpdate
SC_PROCFLOWSIGNATUREGET	0x5F	sc_procFlowSignatureGet
SC_MODULECREATE2	0x60	sc_moduleCreate2
SC_MODULEKILL	0x61	sc_moduleKill
SC_MODULENAMEGET	0x62	sc_moduleNameGet

Name	Number	Error Source
SC_MODULEIDGET	0x63	sc_moduleIdGet
SC_MODULEINFO	0x64	sc_moduleInfo
SC_MODULEPRIOGET	0x65	sc_modulePrioGet
SC_MODULEFRIENDADD	0x66	sc_moduleFriendAdd
SC_MODULEFRIENDRM	0x67	sc_moduleFriendRm
SC_MODULEFRIENDGET	0x68	sc_moduleFriendGet
SC_MODULEFRIENDNONE	0x69	sc_moduleFriendNone
SC_MODULEFRIENDALL	0x6A	sc_moduleFriendAll
SC_MODULESTART	0x6B	sc_moduleStart (Internal)
SC_MODULESTOP	0x6C	sc_moduleStop
	0x6D	reserved
	0x6E	reserved
SC_MODULECBCHK	0x6F	sc_moduleCBChk
SC_TRIGGERVALUESET	0x70	sc_triggerValueSet
SC_TRIGGERVALUEGET	0x71	sc_triggerValueGet
SC_TRIGGER	0x72	sc_trigger
SC_TRIGGERWAIT	0x73	sc_triggerWait
	0x74	reserved
	0x75	reserved
	0x76	reserved
	0x77	reserved
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
	0x7F	reserved
SC_CONNECTORREGISTER	0x80	sc_connectorRegister
SC_CONNECTORUNREGISTER	0x81	sc_connectorUnregister
	0x82	reserved
	0x83	reserved

Name	Number	Error Source
	0x84	reserved
	0x85	reserved
	0x86	reserved
	0x87	reserved
	0x88	dispatcher (Internal)
	0x89	irq_dispatcher (Internal)
	0x8A	kernel_private (Internal)
SC_SYSERROR	0x8B	sc_sysError (many)
	0x8C	sc_safe_* (many)
SC_SYSLDATACORRUPT	0x8D	various
	0x8E	reserved
	0x8F	reserved

4.5 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.

Name	Number	Description
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.
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.

4.6 Error Types

Name	Number	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 Initial

- Whole manual restructured and rewritten.

5.2 System calls added

- `sc_sysIrqDispatcher` extended
- `sc_sysIrqEpilogue`, `sc_sysSWI` and `sc_sysSVC` added
- Layout reworked

5.3 Typos/Design change

- various places
- `sc_procYield`: {V2_INT} Remove scheduler/interrupt test. Now like **[V1]**

5.4 CPU mode clarification

- CPU mode advide before calling any function added.
- Add initial chapter folding for PDF.