SCIOPTA

**High Performance
Real-Time Operating Systems**

Technical Description

Copyright

Disclaimer

Trademark

SCIOPTA is a registered trademark of Litronic AG.

Table of Contents

SCIOPTA - Real-Time Operating System

SCIOPTA - Real-Time Operating System

1   Introduction

1.1   About this Technical Description

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

- Pre-emptive Multi-Tasking **Real-Time Kernel**.

- **IPS** - Internet Protocols.

- **SFS** - File System.

- **CONNECTOR** - support for distributed multi-CPU systems.

- **SMMS** - support for Memory Management Units and system protection.

- **DRUID** - system level debug suite.

The purpose of this **Technical Description** is to give an introduction into the technologies and methods used in **SCIOPTA**.

Please consult the **SCIOPTA** User's Guides for more information.

1.2   Real-Time Operating System Overview

A real-time operating system (RTOS) is the core control software in a real-time system. In a real-time system it must be guaranteed that specific tasks respond to external events within a limited and specified time.
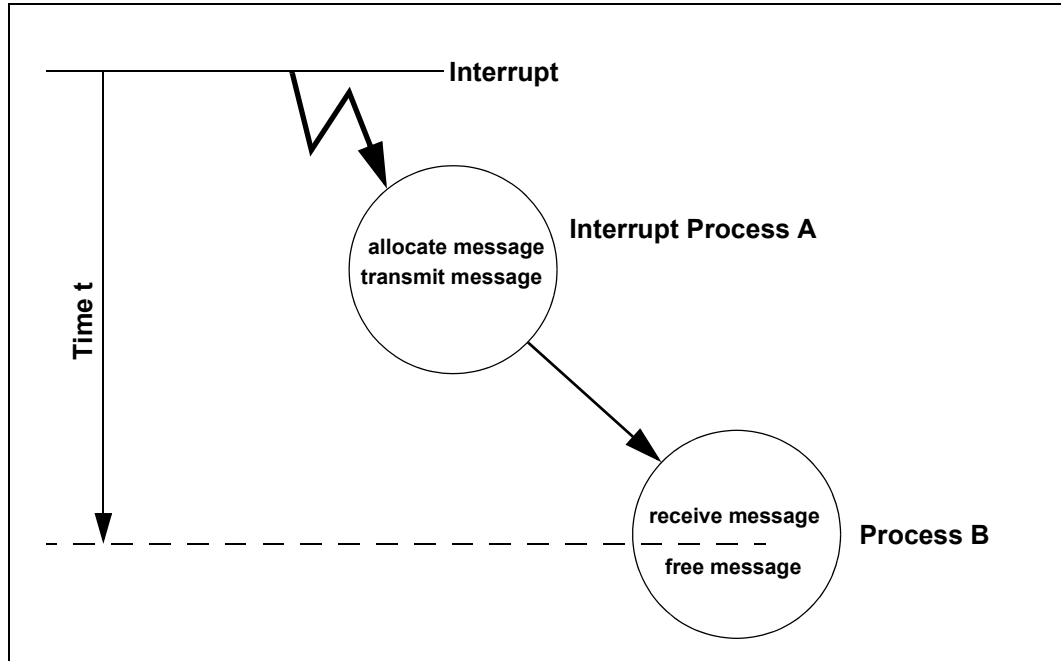
Figure 1-1: Real-Time System Definition

Figure 1-1 shows a typical part of a real-time system. An external interrupt is activating an interrupt process which allocates a message and transmits the message to a prioritized process. The time t between the occurrence of the interrupt and the processing of the interrupt in process B must not exceed a specified maximum time under any circumstances. This maximum time must not depend on system resources such as number of processes or number of messages.

1.2.1   Management Duties

A real-time operating system fulfils many different tasks such as:

• resource management (CPU, Memory and I/O)

• time management

• interprocess communication management

1.2.1.1   CPU Management

As a user of a real-time operating system you will divide your program into a number of small program parts. These parts are called processes and will normally operate independently of each other and be connected through some interprocess communication connections.

It is obvious that only one process can use the CPU at a time. One important task of the real-time operating system is to activate the various processes according to their importance. The user can control this by assigning priorities to the processes.

The real-time operating system guarantees the execution of the most important part of a program at any particular moment.

1.2.1.2   Memory Management

The real-time operating system will control the memory needs and accesses of a system and always guarantee the real-time behaviour of the system. Specific functions and techniques are offered by a real-time operating system to protect memory from writing by processes that should have no access to them.

Thus, allocating, freeing and protecting of memory buffers used by processes are one of the main duties of the memory management part of a real-time operating system.

1.2.1.3   Input/Output Management

Another important task of a real-time operating system is to support the user in designing the interfaces for various hardware such as input/output ports, displays, communication equipment, storage devices etc.

1.2.1.4   Time Management

In a real-time system it is very important to manage time-dependent applications and functions appropriately. There are many timing demands in a real-time system such as notifying the user after a certain time, activating particular tasks cyclically or running a function for a specified time. A real-time operating system must be able to manage these timing requirements by scheduling activities at, or after a certain specified time.

1.2.1.5   Interprocess Communication

The designer of a real-time system will divide the whole system into processes. One design goal of a real-time system is to keep the processes as isolated as possible. Even so, it is often necessary to exchange data between processes.

Interprocess relations can occur in many different forms:

- global variables

- function calls

- timing interactions

- priority relationships

- interrupt enabling/disabling

- semaphores

- message passing

- etc.

One of the duties of a real-time operating system is to manage interprocess communication and to control exchange of data between processes.

2   Kernel

2.1   Introduction

SCIOPTA is a pre-emptive multi-tasking high performance real-time operating system (rtos) for using in embedded systems. SCIOPTA is a so-called message based rtos that is, interprocess communication and coordination are realized by messages.

A typical system controlled by SCIOPTA consists of a number of more or less independent programs called processes. Each process can be seen as if it had the whole CPU for its own use. SCIOPTA controls the system by activating the correct processes according to their priority assigned by the user. Occurred events trigger SCIOPTA to immediately switch to a process with higher priority. This ensures a fast response time and guarantees the compliance with the real-time specifications of the system.

In SCIOPTA processes communicate and cooperate by exchanging messages. Messages can have a content to move data from one process to the other or can be empty just to coordinate processes. Often, process switches can occur as a result of a message transfer.

Besides data and some control structures messages contain also an identity (number).

This can be used by a process for selecting specific messages to receive at a certain moment. All other messages are kept back in the message queue of the receiving process.

Messages are dynamically allocated from a message pool. Messages in SCIOPTA include also ownership. Only messages owned by a process can be accessed by the process. Therefore only one process at a time may access a message (the owner). This automatically excludes access conflicts by a simple and elegant method.

Timing jobs registered by processes are managed by SCIOPTA. Processes which want to suspend execution for a specified time or processes which want to receive messages and declaring specified time-out can all use the timing support of the SCIOPTA system calls.
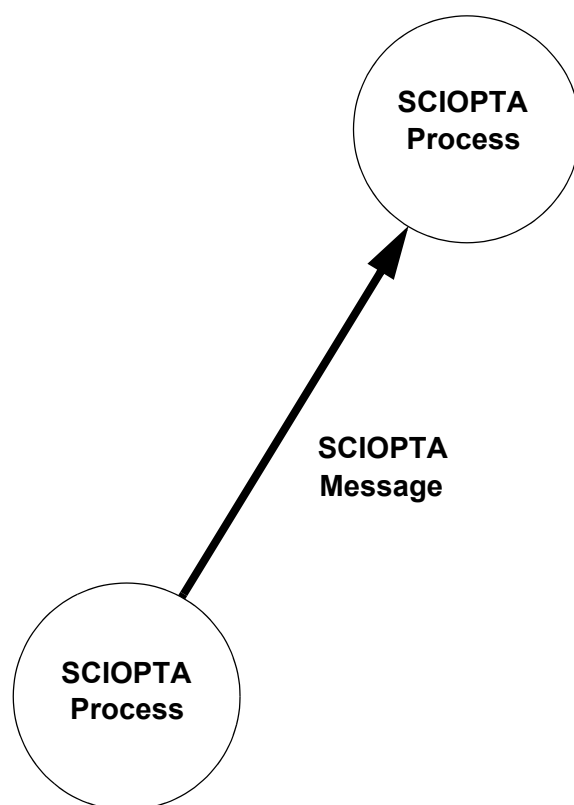
2.2   SCIOPTA Compact

For applications which require a smaller footprint the **SCIOPTA Compact Kernel** is available.

The SCIOPTA Compact kernel does not support the module concept (see chapter **2.5 "Modules" on page 2-11**) and observation (see chapter **2.11 "Observation" on page 2-18**). Modules and observation are features which are not needed in all systems, mainly in 16 bit embedded systems. This is the reason why SCIOPTA Compact is available for 16 bit target processors. The full featured SCIOPTA kernel is aimed at 32 bit processor applications.

By removing the module and observation support, it was possible to reduce the size and complexity of the kernel considerably. The typical size of a SCIOPTA Compact kernel is around 6 kbytes whereas the full featured SCIOPTA kernel has a size of about 25 kbytes. The precise sizes are processor dependent.

Please note that most of the SCIOPTA kernels (including the SCIOPTA Compact kernel) are completely written in assembler language. That's why the SCIOPTA kernels distinguish themselves by a very high performance.

The features and system calls which are not supported by SCIOPTA Compact are market with a frame similar to the following example:

> **Please Note:** The system call **sc_moduleCreate** is not supported in the **SCIOPTA Compact Kernel**.

2.3   Processes

2.3.1   Introduction

An independent instance of a program running under the control of SCIOPTA is called process. SCIOPTA is assigning CPU time by the use of processes and guarantees that at every instant of time, the most important process ready to run is executing. The system interrupts processes if other processes with higher priority must execute (become ready).

2.3.2   Process States

A process running under SCIOPTA is always in the **RUNNING**, **READY** or **WAITING** state.

2.3.2.1   Running

If the process is in the running state it executes on the CPU. Only one process can be in running state in a single CPU system.

2.3.2.2   Ready

If a process is in the ready state it is ready to run meaning the process needs the CPU, but another process with higher priority is running.

2.3.2.3   Waiting

If a process is in the waiting state it is waiting for events to happen and does not need the CPU meanwhile. The reasons to be in the waiting state can be:

- The process tried to receive a message which has (not yet) arrived.

- The process called the sleep system call and waits for the delay to expire.

- The process waits on a SCIOPTA trigger.

- The Process waits on a start system call if it was previously stopped.
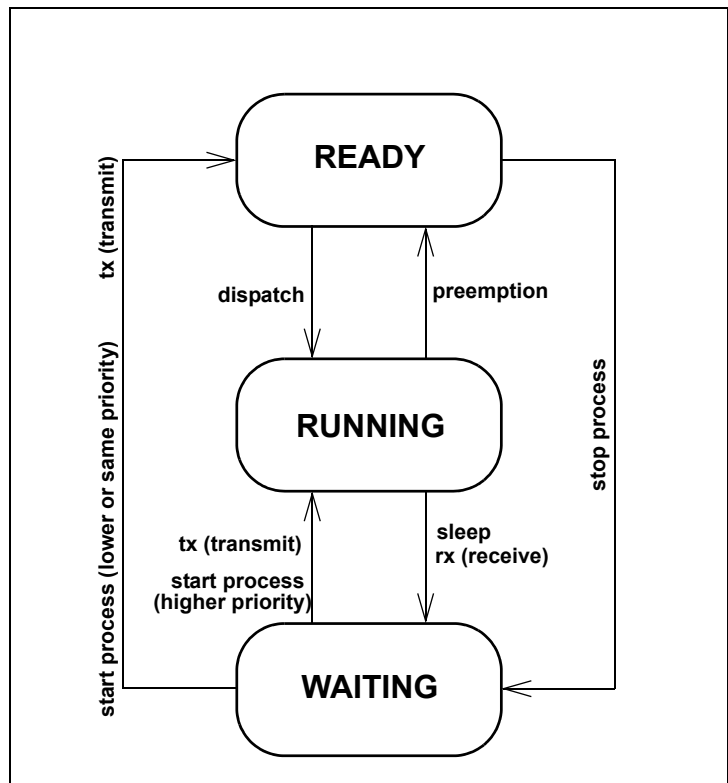
Figure 2-1: State Diagram of SCIOPTA Kernel

2.3.3   Process Categories

In SCIOPTA processes are divided into two main groups, namely static and dynamic processes. They mainly differ in the way they are created and in their dynamic behaviour during run-time.

All SCIOPTA processes have system wide unique process identities.

A SCIOPTA process is always part of a SCIOPTA module. Please consult chapter **2.5 "Modules" on page 2-11** for more information about the SCIOPTA module concept.

2.3.3.1   Static Processes

Static processes are created by the kernel at start-up. They are designed inside a configuration utility by defining the name and all other process parameters such as priority and process stack sizes. At start-up the kernel puts all static created processes into READY or WAITING (stopped) state.

Static process are supposed to stay alive as long as the whole system is alive. But nevertheless in SCIOPTA static processes can be killed at run-time but they will not return their used memory.
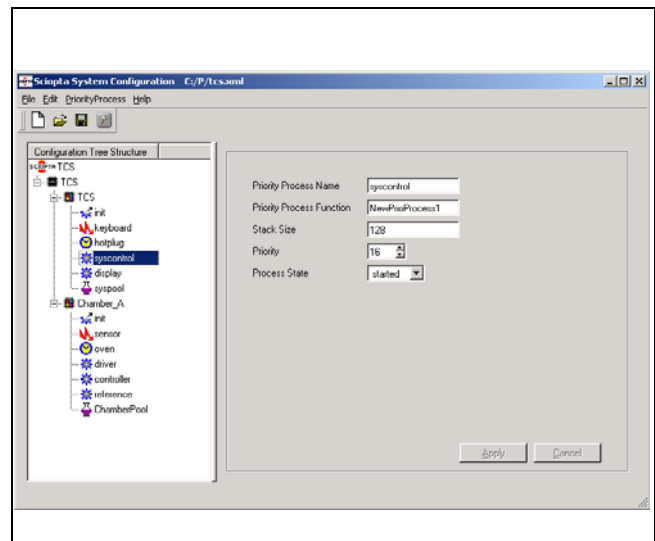
Figure 2-2: Process Configuration Window for Static Processes

2.3.3.2 Dynamic Processes

Dynamic processes can be created and killed during run-time. Often dynamic processes are used to run multiple instances of common code. The number of instances is only limited by system resources and does not to be known before running the system.

Another advantage of dynamic processes is that the resources such as stack space will be given back to the system after a dynamic process is killed.

```
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);
```

Figure 2-3: Create Process System Call

2.3.4   Process Types

2.3.4.1   Prioritized Process

In a typical SCIOPTA system prioritized processes are the most common used process types. Each prioritized process has a priority and the SCIOPTA scheduler is running ready processes according to these priorities. The process with higher priority before the process with lower priority.

If a process has terminated its job for the moment by for example waiting on a message which has not yet been sent or by calling the kernel sleep function, the process is put into the waiting state and is not any longer ready.

2.3.4.2   Interrupt Process

An interrupt is a system event generated by a hardware device. The CPU will suspend the actually running program and activate an interrupt service routine assigned to that interrupt.

The programs which handle interrupts are called interrupt processes in SCIOPTA. SCIOPTA is channelling interrupts internally and calls the appropriate interrupt process.

The priority of an interrupt process is assigned by hardware of the interrupt source. Whenever an interrupt occurs the assigned interrupt process is called, assuming that no other interrupt of higher priority is running. If the interrupt process with higher priority has completed his work, the interrupt process of lower priority can continue.

2.3.4.3   Timer Process

A timer process in SCIOPTA is a specific interrupt process connected to the tick timer of the operating system. SCIOPTA is calling each timer process periodically derived from the operating system tick counter. When configuring or creating a timer process, the user defines the number of system ticks to expire from one call to the other individually for each process.

2.3.4.4   Init Process

The init process is the first process in a module (please consult chapter **2.5 "Modules" on page 2-11** for an introduction in SCIOPTA modules). Each module has at least one process and this is the init process. The init process acts also as idle process which will run when all other processes of a module are in the waiting state.

2.3.4.5 Supervisor Process

SCIOPTA allows you to group processes together into modules. Modules can be created and killed dynamically during run-time. But there is one static module in each SCIOPTA system. This module is called system module (please consult chapter **2.5 "Modules" on page 2-11** for more information about the SCIOPTA module concept).

Processes placed in the system module are called supervisor processes. Supervisor processes have full access rights to system resources. Typical supervisor processes are found in device drivers.

2.3.4.6 Daemons

Daemons are internal processes in a SCIOPTA system. They are running on kernel level and are taking over specific tasks which are better done in a process rather than in a pure kernel function.

2.3.5 Priorities

Each SCIOPTA process and module has a specific priority. The user defines the priorities at system configuration or when creating the module or the process. Process and module priorities can be modified during run-time.

For process scheduling SCIOPTA uses a combination of the module priority and process priority called **effective priority**. The kernel determines the effective priority as follows:

**Effective Priority =
Module Priority + Process Priority**

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

2.3.5.1 Prioritized Processes

By assigning a priority to prioritized processes (including init and supervisor processes as well as daemons) the user designs groups of processes or parts of systems according to response time requirements. Ready processes with high priority are always interrupting processes with lower priority. Systems and modules with high priority processes have therefore faster response time.

Priority values for prioritized processes in SCIOPTA can be from 0 to 31. 0 is the highest and 31 the lowest priority level.

2.3.5.2 Interrupt Processes

The priority of an interrupt process is assigned by hardware of the interrupt source.

2.3.5.3 Timer Processes

Timer processes are specific interrupt processes which all are running on the same interrupt priority level of the timer hardware which generates the SCIOPTA tick.
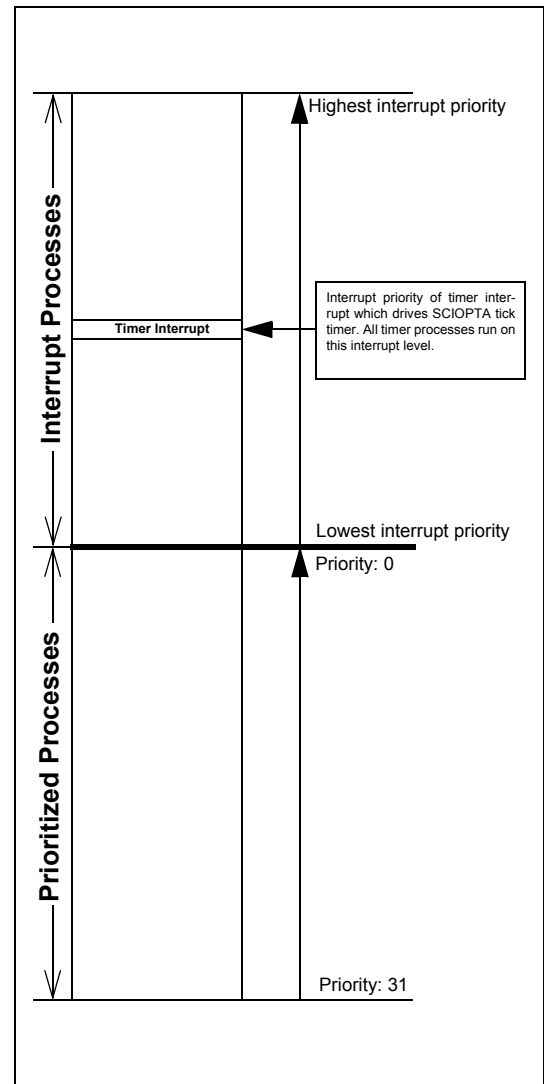
Figure 2-4: SCIOPTA Priority Diagram

2.4 Messages

2.4.1 Introduction

SCIOPTA is a so called Message Based Real-Time Operating System. Interprocess communication and co-ordination is done by messages. Message passing is a very fast, secure, easy to use and good to debug method.

2.4.2 Message Structure

Every SCIOPTA message has a message identity and a range reserved for message data which can be freely accessed by the user. Additionally there are some hidden data structure which will be used by the kernel. The user can access these message information by specific SCCIOPTA system calls. The following message system information are stored in the message header:

- Process ID of message owner

- Message size

- Process ID of transmitting process

- Process ID of addressed process
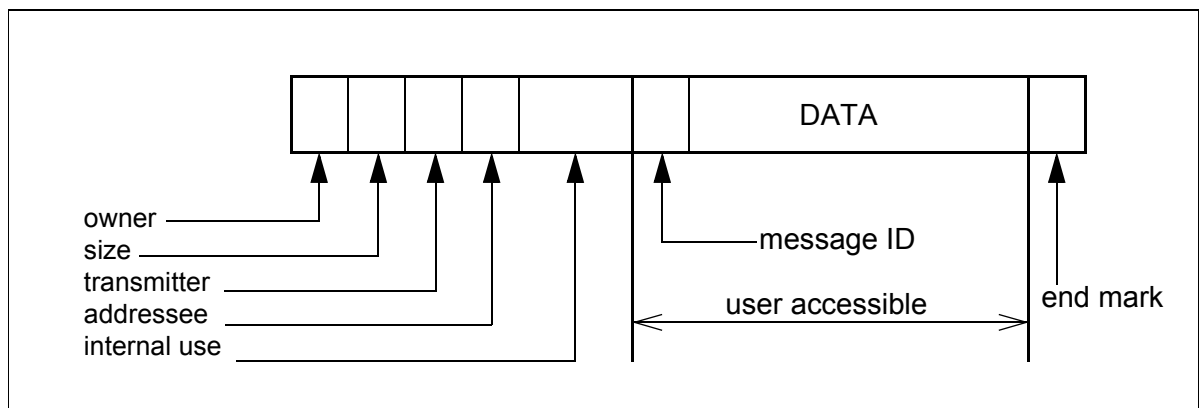
Figure 2-5: SCIOPTA Message Structure

When a process is allocating a message it will be the owner of the message. If the process is transmitting the message to another process, the other process will become owner. After transmitting, the sending process cannot access the message any more. This message ownership feature eliminates access conflicts in a clean and efficient way.

Every process has a message queue where all owned (allocated or received) messages are stored. This message queue is not a own physically separate allocated memory area. It consists rather of a double linked list inside message pools.

2.4.3   Message Sizes

If a process allocates a message there is also the size to be given. The user just gives the number of bytes needed. SCIOPTA is not returning the exact amount of bytes requested but will select one of a list of buffer sizes which is large enough to contain the requested number. This list can contain 4, 8 or 16 sizes which will be defined when a message pool is created.

The difference of requested bytes and returned bytes can not be accessed by the user and will be unused. It is therefore very important to select the buffer sizes to match as close as possible those needed by your application to waste as little memory as possible.

This pool buffer manager used by SCIOPTA is a very well known technique in message based systems. The SCIOPTA memory manager is very fast and deterministic. Memory fragmentation is completely avoided. But the user has to select the buffer sizes very carefully otherwise there can be unused memory in the system.

As you can have more than one message pool in a SCIOPTA system and you can create and kill pools at every moment the user can adapt message sizes very well to system requirements at different system states because each pool can have a different set of buffer sizes.

By analysing a pool after a system run you can find out unused memory and optimize the buffer sizes.

2.4.3.1   Example

A message pool is created with 8 buffer sizes with the following sizes: 4, 10, 20, 80, 200, 1000, 4048, 16000.

If a message is allocated from that pool which requests 300 bytes, the system will return a buffer with 1000 bytes. The difference of 700 bytes is not accessible by the user and is wasted memory.

If 300 bytes buffer are used more often, it would be good design to modify the buffer sizes for this pool by changing the size 200 to 300.

2.4.4   Message Pool

Messages are the main data object in SCIOPTA. Messages are allocated by processes from message pools. If a process does not need the messages any longer it will be given back (freed) by the owner process.

There can be up to 127 pools per module (please consult chapter **2.5 "Modules" on page 2-11** for more information about the SCIOPTA module concept). The maximum number of pools will be defined at module creation. A message pool always belongs to the module from where it was created.

The size of a pool will be defined when the pool will be created. By killing a module the corresponding pool will also be deleted.

Pools can be created, killed and reset freely and at any time.

The SCIOPTA kernel is managing all existing pools in a system. Messages are maintained by double linked list in the pool and SCIOPTA controls all message lists in a very efficient way therefore minimizing system latency.

2.4.5   Message Passing

Message passing is the favourite method for interprocess communication in SCIOPTA. Contrary to mailbox inter-process communication in traditional real-time operating systems SCIOPTA is passing messages directly from process to process.

Only messages owned by the process can be transmitted. A process will become owner if the message is allocated from the message pool or if the process has received the message. When allocating a message by the **sc_msgAlloc()** system call the user has to define the message ID and the size.

The size is given in bytes and the **sc_msgAlloc()** function of SCIOPTA chooses an internal size out of a number of 4, 8 or 16 fixed sizes (see also chapter **2.4.3 "Message Sizes" on page 2-9**).

The **sc_msgAlloc()** or the **sc_msgRx()** call returns a pointer to the allocated message. The pointer allows the user to access the message data to initialize or modify it.

The sending process transmits the message by calling the **sc_msgTx()** system call. SCIOPTA changes the owner of the message to the receiving process and puts the message in the queue of the receiver process. In reality it is a linked list of all messages in the pool transmitted to this process.

If the receiving process is blocked at the **sc_msgRx()** system call and is waiting on the transmitted message the kernel is performing a process swap and activates the receiving process. As owner of the message the receiving process can now get the message data by pointer access. The **sc_msgRx()** call in SCIOPTA supports selective receiving as every message includes a message ID and sender.

If the received message is not needed any longer or will not be forwarded to another process it can be returned to the system by the **sc_msgFree()** and the message will be available for other allocations.
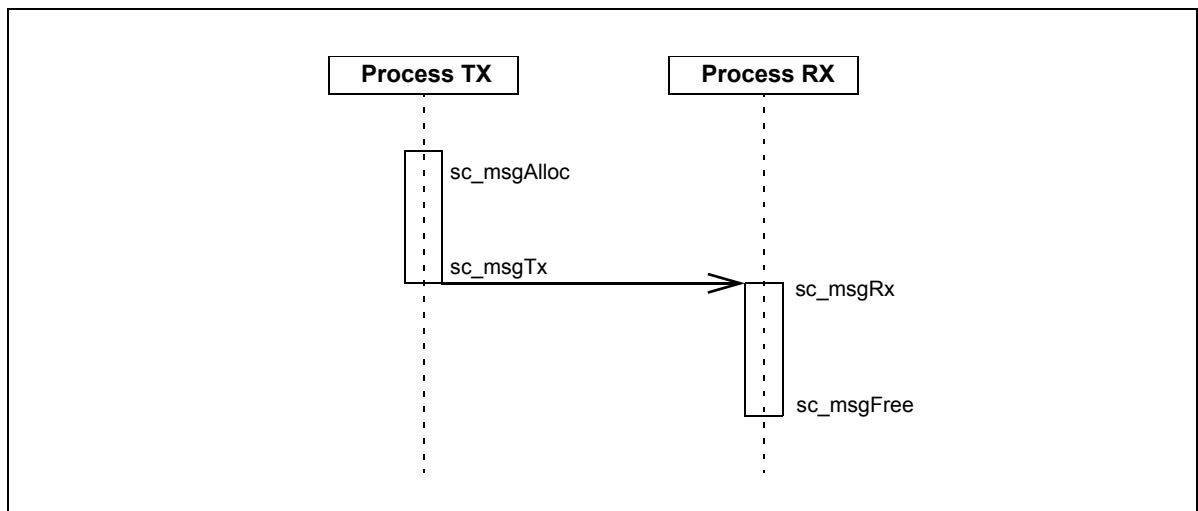

Figure 2-6: Message Sequence Chart of a SCIOPTA Message Passing

SCIOPTA - Real-Time Operating System

2.5   Modules

Processes can be grouped into modules to improve system structure. A process can only be created from within a module.

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. Modules can be created and killed at system start or dynamically during run-time. If a module is killed all processes in the module will be killed and therefore all messages freed and afterwards all pools deleted.

Please Note:

The module concept is not supported in the **SCIOPTA Compact Kernel**. All features described in this chapter 2.5 "Modules" are not available in the **SCIOPTA Compact Kernel**. The **SCIOPTA Compact Kernel** has only one module (the system module) and MMU is not supported.

2.5.1   SCIOPTA Module Friend Concept

SCIOPTA supports also the "friend" concept. Modules can be "friends" of other modules. This has mainly consequences on whether message will be copied or not at message passing. Please consult chapter **2.5.3 "Messages and Modules" on page 2-12** for more information.

A module can be declared as friend by the **sc_moduleFriendAdd** () system call. The friendship is only in one direction. If module A declares module B as a friend, module A is not automatically also friend of Module B. Module B would also need to declare Module A as friend by the **sc_moduleFriendAdd** () system call.

Each module maintains a 128 bit wide bit field for the declared friends. For each friend a bit is set which corresponds to its module ID.

2.5.2   System Module

There is always one static system module in a SCIOPTA system. This module is called system module (sometimes also named module 0) and is the only static module in a system.

SCIOPTA

2.5.3   Messages and Modules

A process can only allocate a message from a pool inside the same module.

Messages transmitted and received within a module are not copied, only the pointer to the message is transferred.

Messages which are transmitted across modules boundaries are always copied except if the modules are "friends". To copy such a message the kernel will allocate a buffer from the pool of the module where the receiving process resides big enough to fit the message and copy the whole message. Message buffer copying depends on the friendship settings of the module where the buffer was originally allocated.

A module can be declared as friend of another module. The message which was transmitted from the module to its declared friend will not be copied. But in return if the friend sends back a message it will be copied. To avoid this the receiver needs to declare the sender also as friend.
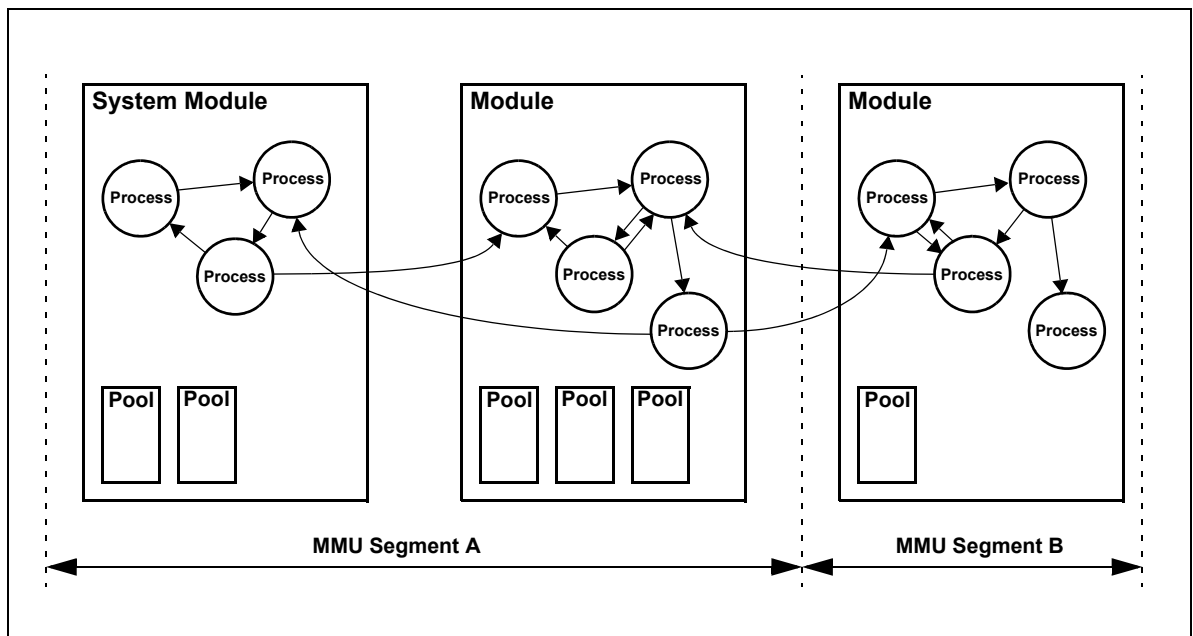
Figure 2-7: SCIOPTA Module Structure

2.5.4   System Protection

In bigger systems it is often necessary to protect certain system areas to be accesses by others. In SCIOPTA the user can achieve such protection by grouping processes into modules creating sub-systems which can be protected.

Full protection is achieved if memory segments are isolated by a hardware Memory Management Unit (MMU). In SCIOPTA such protected memory segments would be layed down at module boundaries.

System protection and MMU support is optional in SCIOPTA and should only be used and configured if you need this feature.

SCIOPTA - Real-Time Operating System

2.6 Trigger

The trigger in SCIOPTA is a method which allows to synchronise processes even faster as it would be possible with messages. With a trigger a process will be notified and woken-up by another process. Trigger are used only for process co-ordination and synchronisation and cannot carry data.

Each process has one trigger available. A trigger is basically a integer variable owned by the process. At process creation the value of the trigger is initialized to one.

There are four system calls available to work with triggers. The **sc_triggerWait**() call decrements the value of the trigger and the calling process will be blocked and swapped out if the value gets negative or equal zero. Only the owner process of the trigger can wait for it. An interrupt process cannot wait on its trigger. The process waiting on the trigger will become ready when another process triggers it by issuing a **sc_trigger**() call which will make the value of the trigger non-negative.

The process which is waiting on a trigger can define a time-out value. If the time-out has elapsed it will be triggered (become non-negative) by the operating system (actually: The previous state of the trigger is restored).

If the now ready process has a higher priority than the actual running process the operating system will pre-empt the running process and execute the triggered process.

The **sc_triggerSet**() system calls allows to sets the value of a trigger. Only the owner of the trigger can set the value. Processes can also read the values of trigger by the **sc_triggerGet**() call.

Also interrupt processes have a trigger but they cannot wait on it. If a process is triggering an interrupt process, the interrupt process gets a software event. This is the same as if an interrupt occurs. The user can investigate a flag which informs if the interrupt process was activated by a real interrupt or woken-up by such a trigger event.
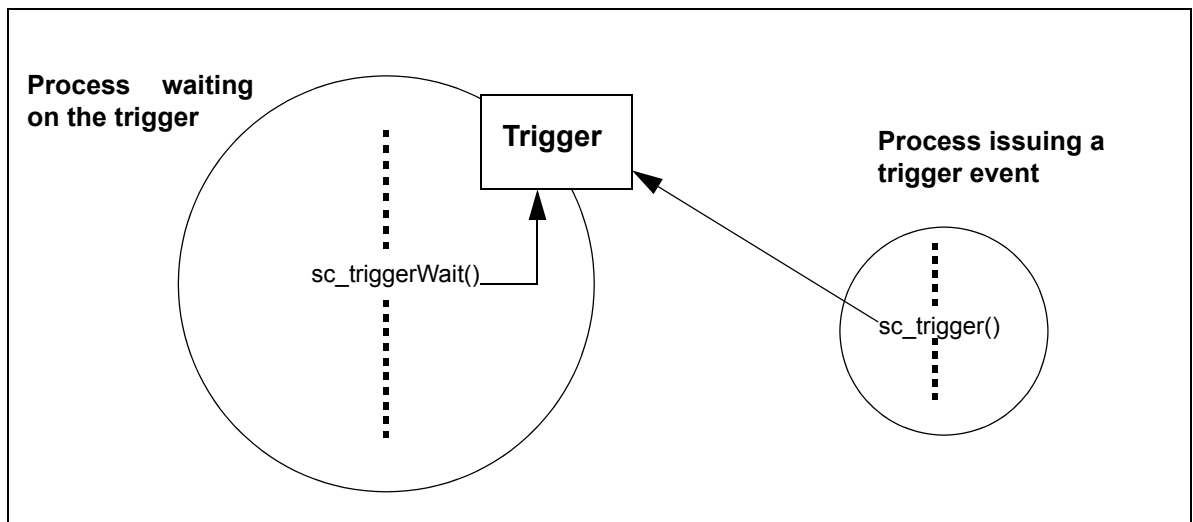
Figure 2-8: SCIOPTA Trigger

2.7   Process Variables

Each process can store local variables inside a protected data area. The process variable are usually maintained inside a SCIOPTA message and managed by the kernel. The user can access the process variable by specific system calls.

Process Control Block
PCB

Pointer

Process Variable Message

Message ID
Reserved by the kernel

TAG 1
Variable 1
TAG 2
Variable 2
TAG 3
Variable 3
TAG 4
Variable 4
TAG n
Variable n
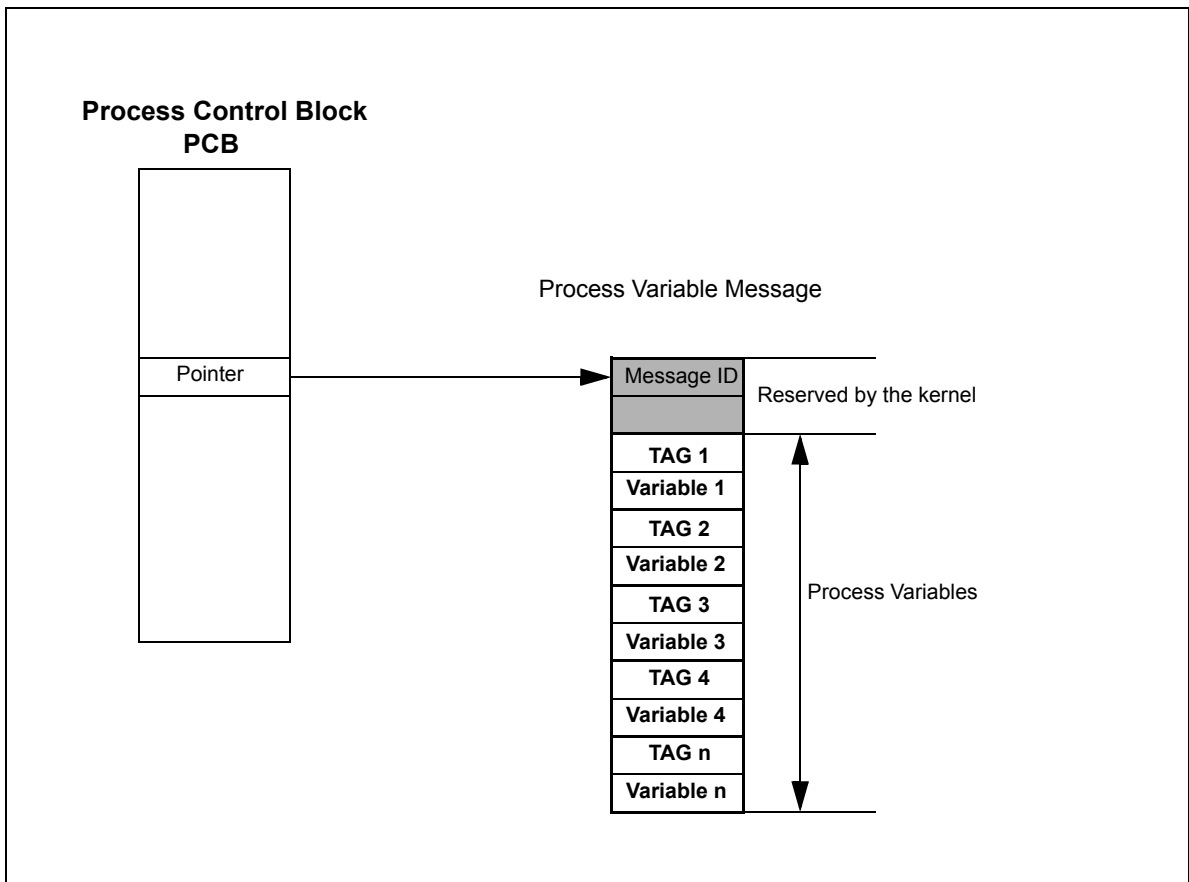
Process Variables

Figure 2-9: SCIOPTA Process Variables

There can be one process variable data area per process. The user needs to allocate a message to hold the process variables. Each variable is preceded by a user defined tag which is used to access the variable. The tag and the process variable have a fixed size large enough to hold a pointer.

It is the user´s responsibility to allocate a big enough message buffer to hold the maximum needed number of process variables. The message buffer holding the variable array will be removed from the process. The process may no longer access this buffer directly. But it can retrieve the buffer if for instance the number of variables must be changed.

2.8   Error Handling

2.8.1   General

SCIOPTA has many built-in error check functions. The following list shows some examples.

- When allocating a message it is checked if the requested buffer size is available and if there is still enough memory in the message pool.

- Process identities are verified in different kernel functions.

- Ownership of messages are checked.

- Parameters and sources of system calls are validated.

- The kernel will detect if messages and stacks have been over written beyond its length.

Contrary to most conventional real-time operating systems, SCIOPTA uses a centralized mechanism for error reporting, called Error Hooks. 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 an 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.

There are two error hooks available:

A) Module Error Hook

B) Global Error Hook

If the kernel detect an error condition it will first call the module error hook and if it is not available call the global error hook. Error hooks must be written by the user. Depending on the type of error (fatal or non-fatal) it will not be possible to return from an error hook. If there are no error hooks present the kernel will enter an infinite loop.

2.8.2   The errno Variable

Each SCIOPTA process has an errno variable. This variable is used mainly by library functions to set the errno variable. The errno variable can only be accessed by some specific SCIOPTA system calls.

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

SCIOPTA - Real-Time Operating System

2.9 SCIOPTA Daemons

Daemons are internal processes in SCIOPTA and are structured the same way as ordinary processes. They have a process control block (pcb), a process stack and a priority. Not all SCIOPTA daemons are part of the standard SCIOPTA delivery.

2.9.1 Process Daemon

The **Process Daemon** (**sc_procd**) is identifying processes by name and supervises created and killed processes.

2.9.2 Kernel Daemon

The **Kernel Daemon** (**sc_kerneld**) is creating and killing modules and processes. Some time consuming system work of the kernel (such as module and process killing) returns to the caller without having finished all related work. The **Kernel Daemon** is doing such work at appropriate level.

2.10 SCIOPTA Scheduling

SCIOPTA uses the pre-emptive prioritized scheduling for all prioritized process types. Timer process are scheduled on a cyclic base at well defined time intervals.

The prioritized process with the highest priority is running (owning the CPU). SCIOPTA is maintaining a list of all prioritized processes which are ready. If the running process becomes not ready (i.e. waiting on at a message receive which has not yet arrived) SCIOPTA will activate the next prioritized process with the highest priority. If there are more than one processes on the same priority ready SCIOPTA will activate the process which became ready in a first-in-first-out methodology.

Interrupt and timer process will always pre-empt prioritized processes. The intercepted prioritized process will be swapped in again when the interrupting system on the higher priority has terminated.

Timer processes run on the tick-level of the operating system.

The SCIOPTA kernel will do a re-scheduling at every, receive call, transmit call, process yield call, trigger wait call, sleep call and all system time-out which have elapsed.
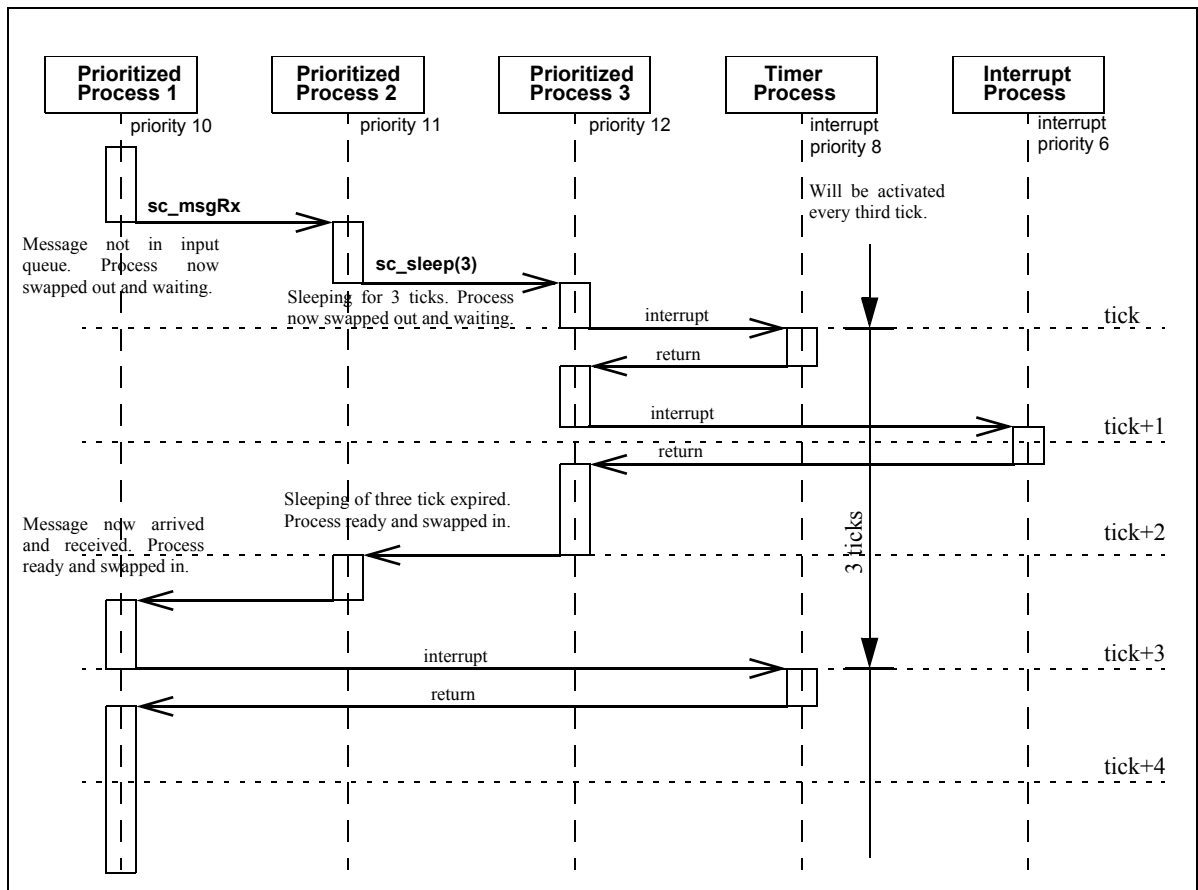

Figure 2-10: Scheduling Sequence Example

SCIOPTA - Real-Time Operating System *(vertical, left margin)*

2.11 Observation

Communication channels between processes in SCIOPTA can be observed no matter if the processes are local or distributed over remote systems. The process calls **sc_procObserve()** which includes the pointer to a return message and the process ID of the process which should be observed.

If the observed process dies the kernel will send the defined message back to the requesting process to inform it. This observation works also with remote process lists in connectors. This means that not only remote processes can be observed but also connection problems in communication links if the connectors includes the necessary functionality.

~~Please Note:~~

~~Observation is not supported in the **SCIOPTA Compact Kernel**.~~

Registration of observation of process B requesting the message B_killed as returned information.

Kernel

process A

sc_procObserve()

B_killed

Message B_killed is returned by the kernel to inform process A of the killing of process B.

process B

Observed link
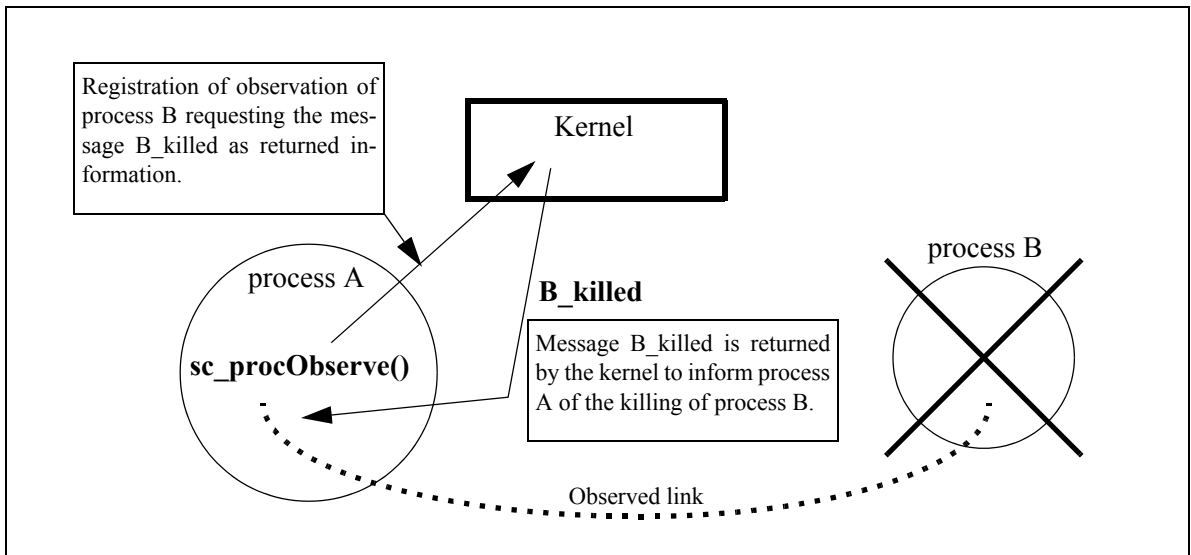
Figure 2-11: SCIOPTA Observation

SCIOPTA - Real-Time Operating System

2.12   Hooks

2.12.1   Introduction

Hooks are user written functions which are called by the kernel at different location. They are only called if the user defined them at configuration. User hooks are used for a number of different purposes and are target system dependent.

2.12.2   Error Hook

The error hook is the most important user hook function and will normally be included in most of the systems. An error hook can be used to log the error and additional data on a logging device if the kernel has detected an error condition. Please consult also chapter **2.8 "Error Handling" on page 2-15** for additional information.

2.12.3   Message Hooks

In SCIOPTA you can configure **Message Transmit Hooks** and **Message Receive Hooks**. These hooks are called each time a message is transmitted to any process or received by any process. Transmit and Receive Hooks are mainly used by user written debugger to trace messages.

2.12.4   Pool Hooks

Pool Create Hooks and **Pool Kill Hooks** are available in SCIOPTA mainly for debugging purposes. Each time a pool is created or killed the kernel is calling these hooks provided that the user has configured the system accordingly.

2.12.5   Process Hooks

If the user has configured **Process Create Hooks** and **Process Kill Hooks** into the kernel these hooks will be called each time if the kernel creates or kills a process.

SCIOPTA allows to configure a **Process Swap Hook**. The Process Swap Hook is called by the kernel each time a new process is about to be swapped in. This hook is also called if the kernel is entering idle mode.

SCIOPTA - Real-Time Operating System

2.13   System Calls

2.13.1   Introduction

This chapter contains a list and a short description of all SCIOPTA kernel system calls. The system calls are listed in functional groups.

2.13.2   Message System Calls

sc_msgAcquire	Changes the owner of the message. The caller becomes the owner of the message.
sc_msgAddrGet	Returns the process ID of the addressee of the message.
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 initializes the message data to 0.
sc_msgFree	Returns a message to the message pool.
sc_msgHookRegister	Registers a message hook.
sc_msgOwnerGet	Returns the process ID of the owner of the message.
sc_msgPoolIdGet	Returns the pool ID of a message.
sc_msgRx	Receives one ore more defined messages.
sc_msgSndGet	Returns the process ID of the sender of the message.
sc_msgSizeGet	Returns the size of the message buffer.
sc_msgSizeSet	Modifies the size of a message buffer.
sc_msgTx	Sends a message to a process.
sc_msgTxAlias	Sends a message to a process by setting any process ID.

2.13.3   Process System Calls

sc_procCreate	Requests the kernel daemon to create process. ~~This system call is only available by the SCIOPTA Compact kernel.~~
sc_procDaemonRegister	Registers a process daemon which is responsible for pidGet request. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procDaemonUnregister	Unregisters a process daemon. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procHookRegister	Registers a process hook.
sc_procIdGet	Returns the process ID of a process.

SCIOPTA - Real-Time Operating System

sc_procIntCreate	Requests the kernel daemon to create an interrupt process. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procKill	Requests the kernel daemon to kill a process.
sc_procNameGet	Returns the full name of a process.
sc_procObserve	Request a message to be sent if the given process pid dies (process supervision). ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procPathGet	Returns the path of a process.
sc_procPpidGet	Returns the process ID of the parent of a process. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procPrioCreate	Requests the kernel daemon to create a prioritized process. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procPrioGet	Returns the priority of a process.
sc_procPrioSet	Sets the priority of a process.
sc_procSchedLock	Locks the scheduler and returns the number of times it has been locked before.
sc_procSchedUnLock	Unlocks the scheduler by decrementing the lock counter by one.
sc_procSliceGet	Returns the time slice of a timer process.
sc_procSliceSet	Sets the time slice of a timer process.
sc_procStart	Starts a process.
sc_procStop	Stops a process.
sc_procTimCreate	Requests the kernel daemon to create a timer process. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procUnobserve	Cancels the observation of a process. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procUsrIntCreate	Requests the kernel daemon to create a user interrupt process. ~~This system call is not supported by the SCIOPTA Compact kernel.~~
sc_procVectorGet	Returns the interrupt vector of an interrupt process.
sc_procYield	Yields the CPU to the next ready process within the current process's priority group.

2.13.4   Module System Calls

> **Please Note:**
>
> The module concept is not supported in the **SCIOPTA Compact Kernel**. All module system calls are not available in the **SCIOPTA Compact Kernel**. The **SCIOPTA Compact Kernel** has only one module (the system module).

sc_moduleCreate	Creates a module.
sc_moduleFriendAdd	Adds a module to the module friend set.
sc_moduleFriendAll	Sets a modules as friend.
sc_moduleFriendGet	Returns if module is member of a module friend set.
sc_moduleFriendNone	Removes all modules from a module friend set.
sc_moduleFriendRm	Removes a module from the module friend set.
sc_moduleIdGet	Returns the ID of a module.
sc_moduleInfo	Returns a snap-shot of a module control block.
sc_moduleKill	Kills a module.
sc_moduleNameGet	Returns the full name of a module.

2.13.5   Message Pool Calls

sc_poolCreate	Creates a message pool.
sc_poolDefault	Sets a message pool as default pool.
sc_poolHookRegister	Registers a pool hook.
sc_poolIdGet	Returns the ID of a message pool.
sc_poolInfo	Returns a snap-shot of a pool control block.
sc_poolKill	Kills a message pool.
sc_poolReset	Resets a message pool in its original state.

2.13.6   Timing Calls

sc_sleep	Suspends a process for a defined time.
sc_tmoAdd	Request a time-out message after a defined time.
sc_tmoRm	Remove a time-out job.

2.13.7   System Tick Calls

sc_tick	Calls the kernel tick function. Advances the kernel tick counter by 1.
sc_tickGet	Returns the actual kernel tick counter value.
sc_tickLength	Returns/sets the current system tick-length.
sc_tickMs2Tick	Converts a time from milliseconds into ticks.
sc_tickTick2Ms	Converts a time from ticks into milliseconds.

2.13.8   Process Trigger Calls

sc_trigger	Signals a process trigger.
sc_triggerValueGet	Returns the value of a process trigger.
sc_triggerValueSet	Sets the value of a process trigger.
sc_triggerWait	Waits on its process trigger.

2.13.9   Process Variable Calls

sc_procVarDel	Deletes a process variable.
sc_procVarGet	Returns a process variable.
sc_procVarInit	Initializes a process variable area.
sc_procVarRm	Removes a process variable area.
sc_procVarSet	Sets a process variable.

2.13.10   Connector Process Calls

sc_connectorRegister	Registers a connector process.
sc_connectorUnregister	Removes a registered connector process.

2.13.11 Miscellaneous and Error Calls

sc_miscCrc   Calculates a CRC over a specified memory range. ~~This system call is not supported by the SCIOPTA Compact kernel.~~

sc_miscCrcContd   Calculates a CRC over an additional memory range. ~~This system call is not supported by the SCIOPTA Compact kernel.~~

sc_miscErrnoSet   Sets the error code.

sc_miscErrnoGet   Returns the error code.

sc_miscError   Error call.

sc_miscErrorHookRegister   Registers an Error Hook.

SCIOPTA - Real-Time Operating System

3   SCIOPTA Device Driver Concept

3.1   Diagram
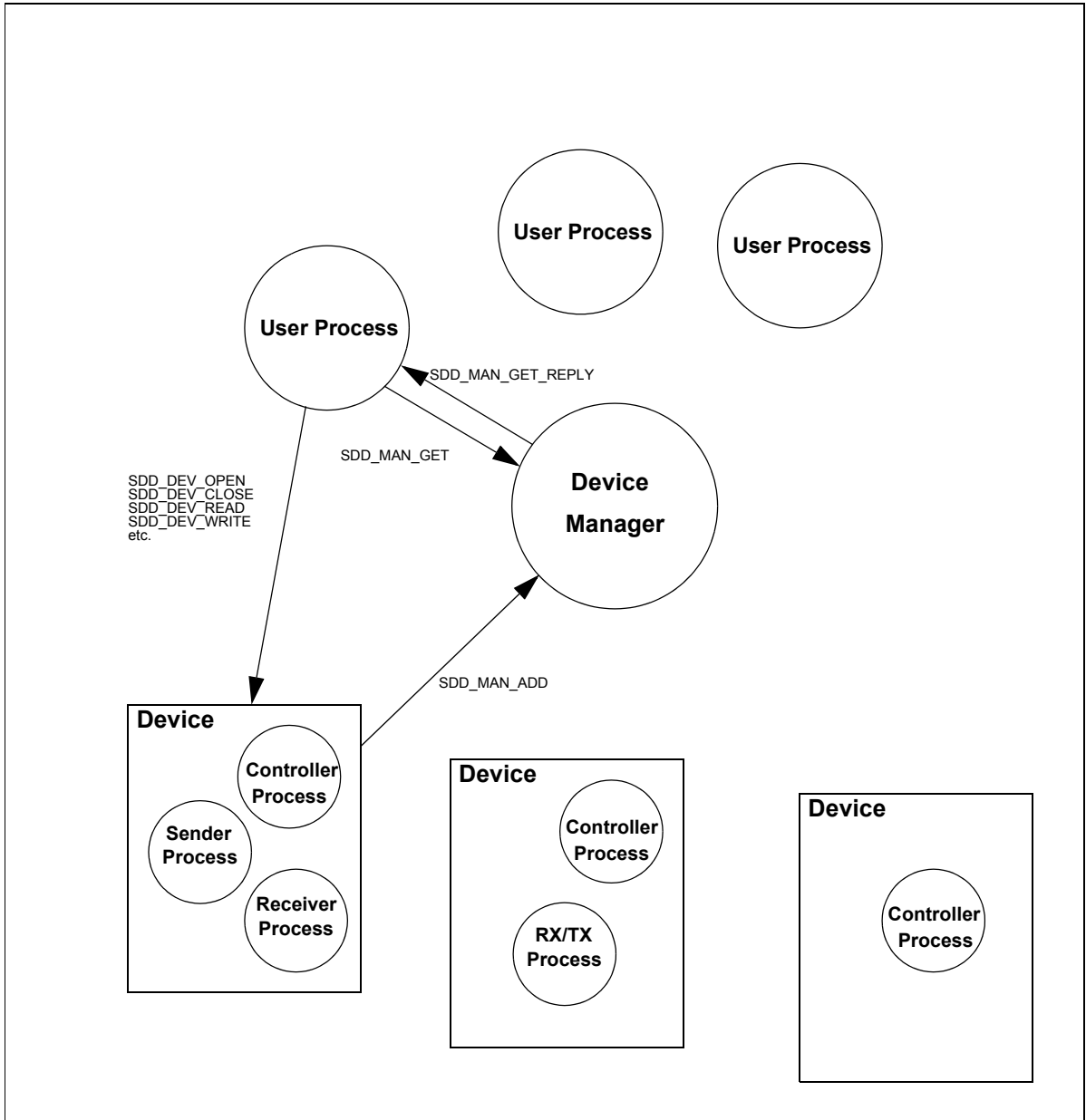
Figure 3-1: SCIOPTA Device Driver Concept

3.2   Overview

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

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

The user process is getting information about the device from the device manager and communicates directly with the device processes. As SCIOPTA is a message based system all communication is done by SCIOPTA messages. But there is also a function interface and file descriptor interface (posix) available.

3.3   SDD Descriptors

SDD Descriptors are data structures in SCIOPTA containing information about specific drivers or objects. Drivers are not only programs managing and controlling devices (device drivers) they can also represent objects which are managing protocols, file system files or other system resources.

SDD descriptors are stored as standard SCIOPTA messages inside message pools. That is why SDD descriptors contain a message ID structure element.

3.3.1   General SDD Descriptor Definition

An **SDD Object Descriptor** contains information about a **SCIOPTA Object**.

3.3.2   Specific SDD Descriptors

- **SDD Device Descriptors** contain information about **Device Drivers**.
- **SDD Protocol Descriptors** contain information about **IPS Protocol Drivers**.
- **SDD File Descriptors** contain information about **SFS File Drivers**.

3.3.3   Standard SDD Descriptor Structure

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

base.id   Standard SCIOPTA message ID.

base.error   Error code.

base.handle	Opaque handle of the driver. Handles are used to manage multiple instances of drivers. The handle points to a data structure of the driver which contains specific information about a driver instance.
manager	Contains the manager access handle.
type	Type of the driver or object.
name	Contains the name of the driver. The name must be unique within a domain. A manager corresponds to a domain.
controller	The controller process ID of a driver.
sender	The sender process ID of a driver.
receiver	The receiver process ID of a driver.

3.4   Registering Devices

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

3.5   Using Devices

User processes will communicate directly with device drivers. Before a user process can communicate with a device it must get the device descriptor from the device manager. Using the message interface the user process can request a device by sending a **SDD_MAN_GET** message to the device manager which responds with a **SDD_MAN_GET_REPLY** message. This reply message contains the device descriptor with full information about the device including:

• Process ID(s) of all processes (controller, sender and receiver)

• Device handle (pointer to a data structure of the device which holds additional device information such as unit numbers etc.)

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

3.6   Message Sequence Chart Register and Use of a Device

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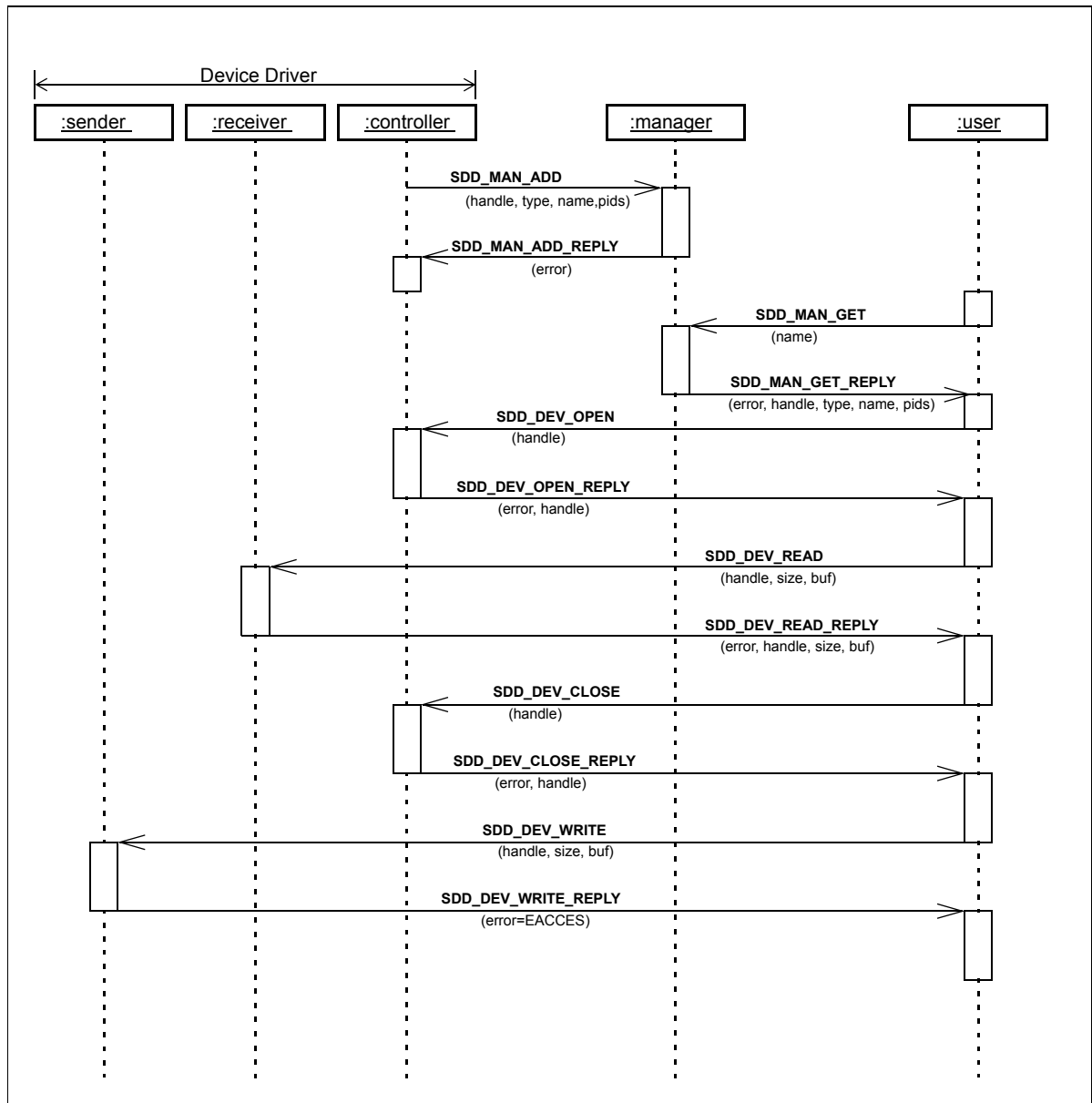

Figure 3-2: MSC Adding a Device and Use It

SCIOPTA

3.7   Device Driver Application Programmers Interface

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

The SCIOPTA device driver system is based on the SCIOPTA message passing technology. You can access the device driver functionality by exchanging messages. This results in a very efficient, fast and direct way of working with SCIOPTA.

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

Another convenient way is to use the Posix File Descriptor Interface as it is a standardized API.
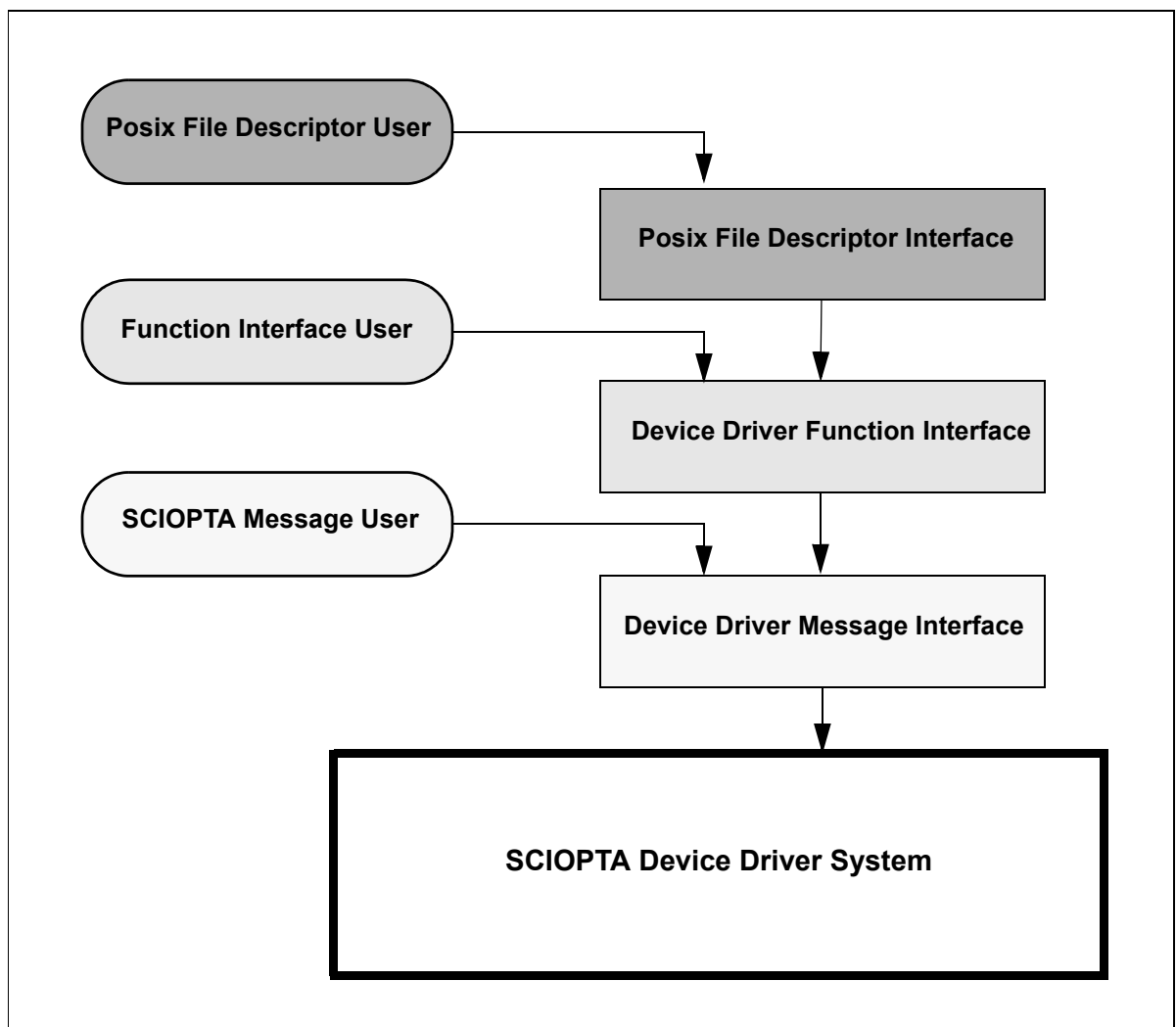
Figure 3-3: SCIOPTA Device Driver System API

SCIOPTA - Real-Time Operating System

3.7.1   Device Driver Messages

SDD_DEV_CLOSE	Closes an open device.
SDD_DEV_IOCTL	Sets or gets specific parameters to/from device drivers on the hardware layer.
SDD_DEV_OPEN	Opens a device for read, write or read/write.
SDD_DEV_READ	Reads data from a device driver.
SDD_DEV_WRITE	Writes data to a device driver.
SDD_ERROR	Mainly used by device driver and other processes which do not use reply messages as answer of request messages for returning error codes.
SDD_FILE_RESIZE	Increases or decreases the size of an existing file.
SDD_FILE_SEEK	Positions, writes and reads pointers in a file.
SDD_MAN_ADD	Adds a new device in the device driver system.
SDD_MAN_GET	Gets the device descriptor of a registered device.
SDD_MAN_GET_FIRST	Gets the device descriptor of the first registered device from the manager's device list.
SDD_MAN_GET_NEXT	Get the device descriptor of the next registered device from the manager's device list.
SDD_MAN_INFO	Collects information from a manager.
SDD_MAN_RM	Removes a device from the device driver system.
SDD_OBJ_DUP	Creates a copy of a device with identical data structures.
SDD_OBJ_INFO	Collects information from a device or SDD object.
SDD_OBJ_RELEASE	Releases an on-the-fly object.

3.7.2   Device Driver Functions

sdd_devAread()	Reads data in an asynchronous mode from a device driver.
sdd_devClose()	Closes an open device.
sdd_devIoctl()	Gets and sets specific parameters in device drivers on the hardware layer.
sdd_devOpen()	Opens device for read, write or read/write, or to create a device (file).
sdd_devRead()	Reads data from a device driver.
sdd_devWrite()	Writes data to a device driver.
sdd_fileResize()	Increases or decreases the size of an existing file.
sdd_fileSeek()	Positions writes and reads pointers in a file.
sdd_manAdd()	Adds a new device in the device driver system.

SCIOPTA - Real-Time Operating System

sdd_manGetByName()	Gets the SDD device descriptor of a registered device from the manager's device list by giving the name as parameter.
sdd_manGetFirst()	Gets the device descriptor of the first registered device from the manager's device list.
sdd_manGeNext()	Get the device descriptor of the next registered device from the manager's device list.
sdd_manNoOfItems()	Get the number of registered devices of the manager device list.
sdd_manGetRoot()	Creates an SDD object descriptor of a root manager process.
sdd_manRm()	Removes a device from the device driver system.
sdd_netbufAlloc()	Requests a new network buffer.
sdd_netbufCopy()	Copies one network buffer into another.
sdd_netbufCur()	Moves the cur index of a network buffer.
sdd_netbufCurReset()	Resets the cur index of a network buffer.
sdd_netbufHeadRoom()	Gets the size of the head of a network buffer.
sdd_netbufPull()	Increases the head.
sdd_netbufPush()	Decrease the head.
sdd_netbufPut()	Increases the data section of a network buffer toward end.
sdd_netbufTailRoom()	Gets the size of the tail of a network buffer.
sdd_netbufTrimm()	Cuts down the data section to the given value.
sdd_objDup()	Creates a copy of a device with identical data structures.
sdd_objInfo()	Collects information from a device or SDD object.
sdd_objRelease()	Releases an on-the-fly object.
sdd_objResolve()	Returns the last struct manager in a given path for hierarchical organized managers.

3.8   Hierarchical Structured Managers

In a SCIOPTA system there can be more than one manager and managers can be organized in a hierarchical structure. This can already be seen as the base of a file system. In a hierarchical manager organization, managers reside below the root managers and have a nested organization. Hierarchical organized manager systems are mainly used in file systems such as the SCIOPTA SFS.
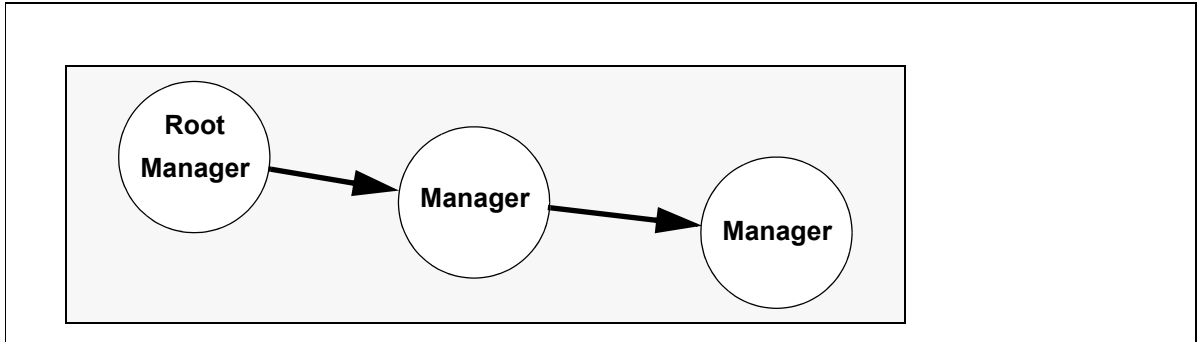
Figure 3-4: SCIOPTA Hierarchical Structured Managers

3.9   Board Support Packages

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

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

List and description of all

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

4   IPS Internet Protocols TCP/IP

4.1   Introduction

The IPS Internet Protocol Suite allows embedded systems running SCIOPTA to communicate with other embedded systems or computers running SCIOPTA or different operating systems.

SCIOPTA IPS is a high performance TCP/IP communication stack for embedded systems. IPS supports TCP, UDP, ICMP, IGMP, IP, IGMP, Fragmentation, PPP and Ethernet. This product has been specially designed to meet the requirement of modern internet protocol network applications in embedded systems. This gives IPS the advantages over traditional internet stacks, of having higher performance and a lower memory footprint.

SCIOPTA IPS is a scalable stack. Memory footprint and RAM needs depend on the IPS configuration and can be adapted to specific applications needs.

4.2   IPS Protocol Layers

The SCIOPTA IPS internet protocols are developed in layers the same way as all standard TCP/IP protocols. Each layer is responsible for a different task of the communications. The IPS protocol suite is the combination of specific protocols at different layers. IPS can be considered to be a four-layer system.
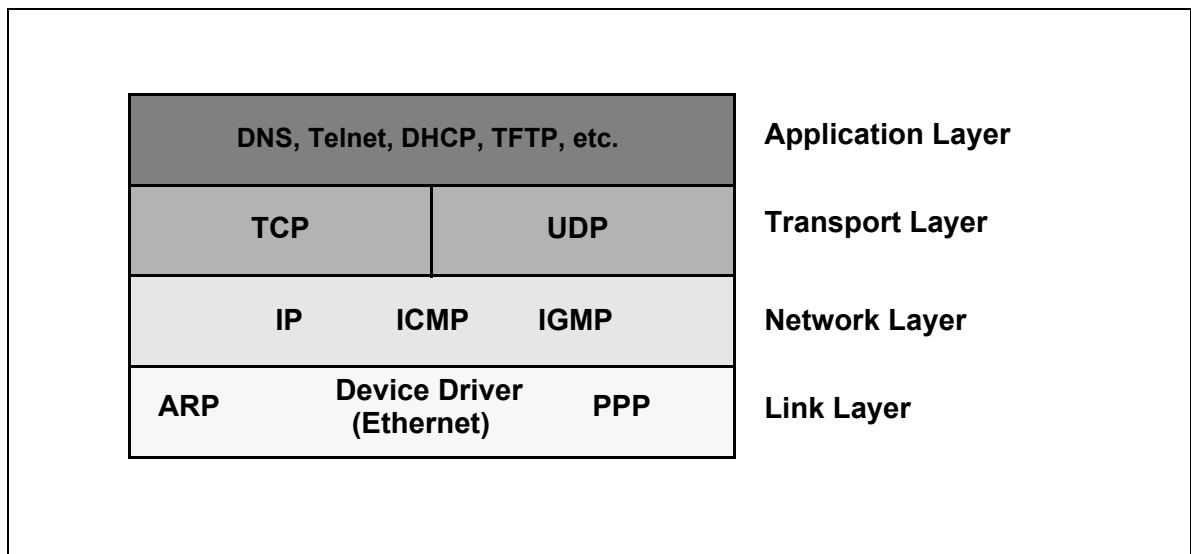
Figure 4-1: IPS Protocol Layers

4.2.1   Application Layer

The application layer controls and manages a specific application. There are many standard applications available for SCIOPTA IPS such as DNS, Telnet, DHCP, TFTP etc. For each application there is a specific SCIOPTA IPS manual available.

SCIOPTA - Real-Time Operating System

4.2.2   Transport Layer

The transport layer is responsible to provide a flow of data between two systems for the application on the upper layer. As in all TCP/IP stack, there are two different transport protocols available in IPS. The User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

4.2.2.1   User Datagram Protocol

UDP provides a quite simple service to the application layer by just sending packets of data called datagrams from one system to the other. UDP provides no reliability, there is no guarantee that the datagrams reach the other end. The application layer must provide any desired reliability.

4.2.2.2   Transmission Control Protocol

TCP on the other hand provides a reliable flow of data between two systems. It handles functions such as

*   setting timeouts to make sure the other system acknowledges packages that are sent,
*   acknowledging received packages,
*   dividing the data passed to it from the application into data sizes appropriate for the layer below,
*   and many other things.

As TCP provides reliable data package flow, the application layer does not need to add reliability.

4.2.3   Network Layer

The network layer manages the sending and transmitting of packages around the network. For example, the routing is handled by the network layer. Internet Protocol (IP), Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP) are provided in the network layer.

4.2.4   Link Layer

The link layer includes the device driver and the network device (such as Ethernet for example). The link layer handles all the hardware details of physically interfacing with the network media (such as the cable).

The Address Resolution Protocol (ARP) is included in the link layer and is used to convert between the IP address and network interface address.

There is also a Point-to-Point Protocol (PPP) available for SCIOPTA IPS which is placed in the link layer. PPP provides a way to encapsulate IP datagrams on a serial link. Please consult the SCIOPTA - IPS PPP, User´s Guide and Reference Manual for more information about PPP.

4.3 IPS Processes

All parts of the SCIOPTA IPS stack are encapsulated in SCIOPTA static or dynamic processes which can be started and stopped individually. This gives a highly modular design which can be scaled for specific applications.
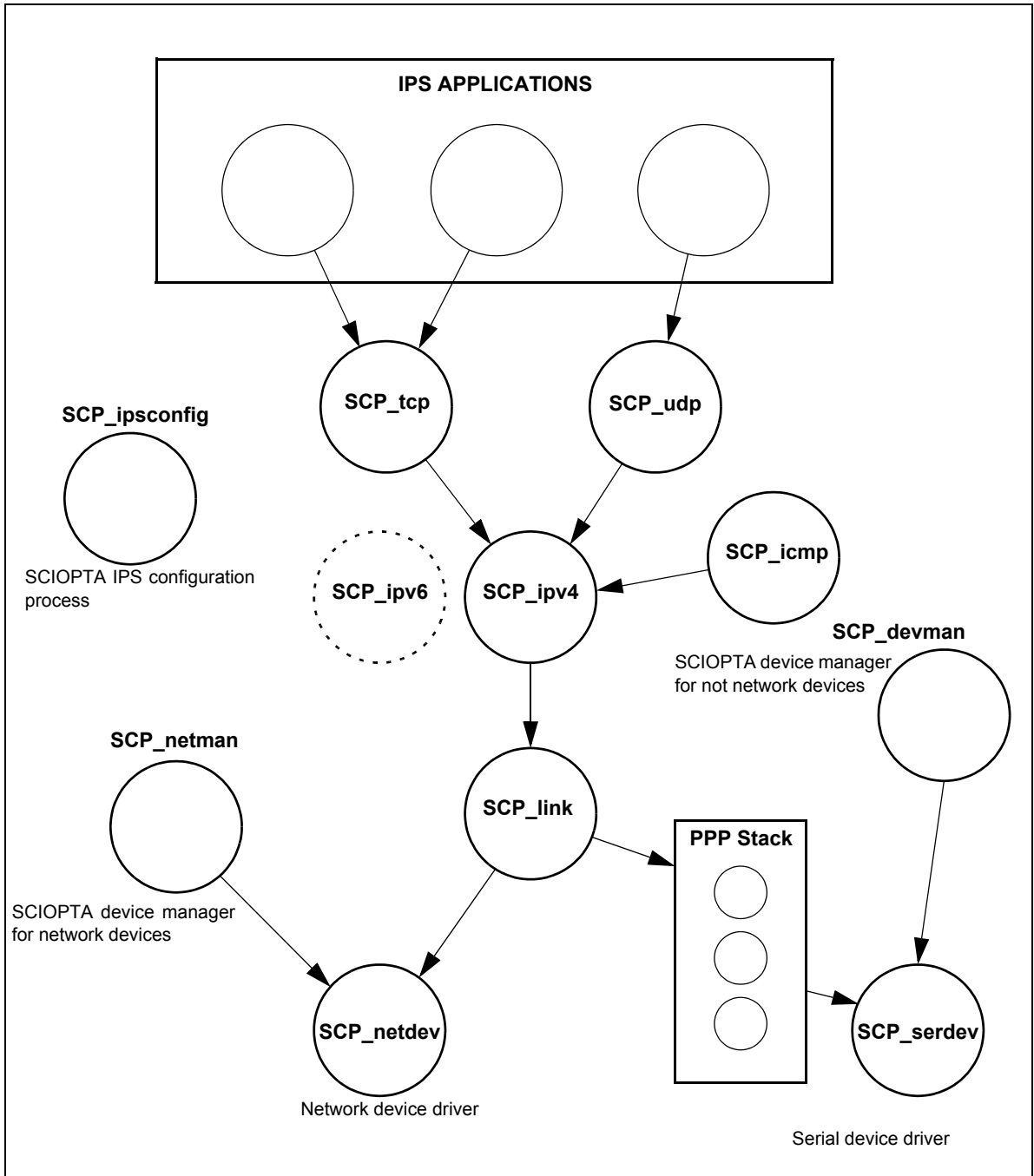
Figure 4-2: IPS Processes

4.4   Using IPS

The SCIOPTA IPS processes manage the protocols, add and maintain the protocol headers and control the interfaces. During the protocol management and while the data transfer occurs the user can still perform some concurrent work. The IPS function interface library must be linked to the process which need to access the IPS stack.
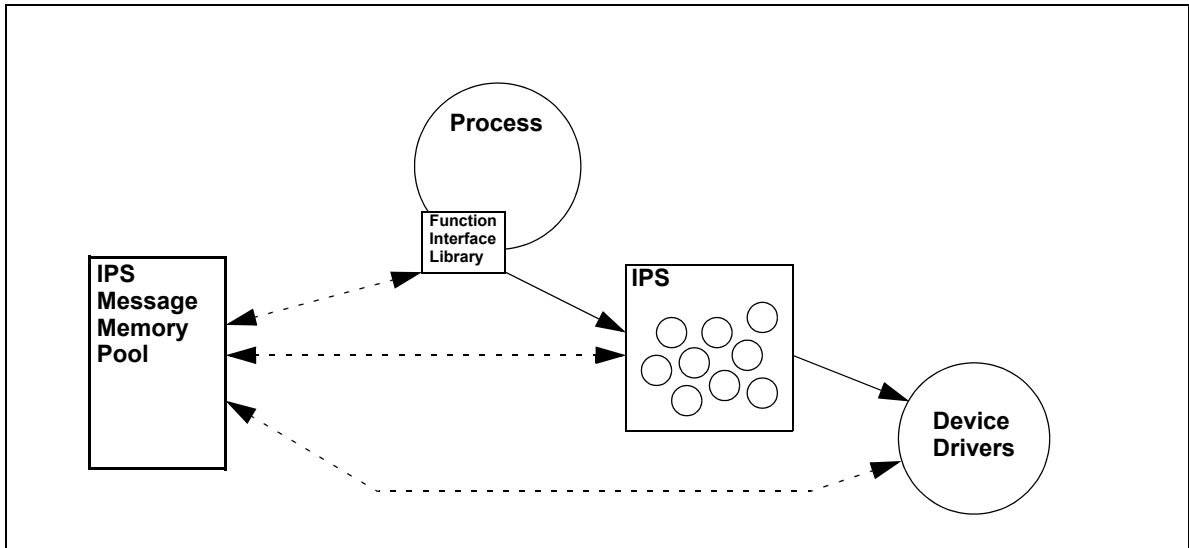
Figure 4-3: IPS Processes

For programming communication applications the programmer can use the well known standard socket function calls such as socket(), accept(), connect(), read().

In addition some specific interface methods are provided in SCIOPTA IPS which helps to increase the communication performance:

- A socket with the socket option **SO_SC_ASYNC** causes IPS to use the SCIOPTA message queue (message pool) for receiving data independent of how the data was sent. This method allows the user to concurrently work with ordinary SCIOPTA messages while waiting and handling received TCP or UDP data packages.

- SCIOPTA IPS includes an efficient flow-control.

- SCIOPTA IPS features a zero copy throughput.

4.5   IPS Application Programmers Interface

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

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

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

Another convenient way is to use the BSD Socket Interface as it is a standardized API for most Internet Protocols applications. There are also IPS specific system calls included for supporting asynchronous mode.
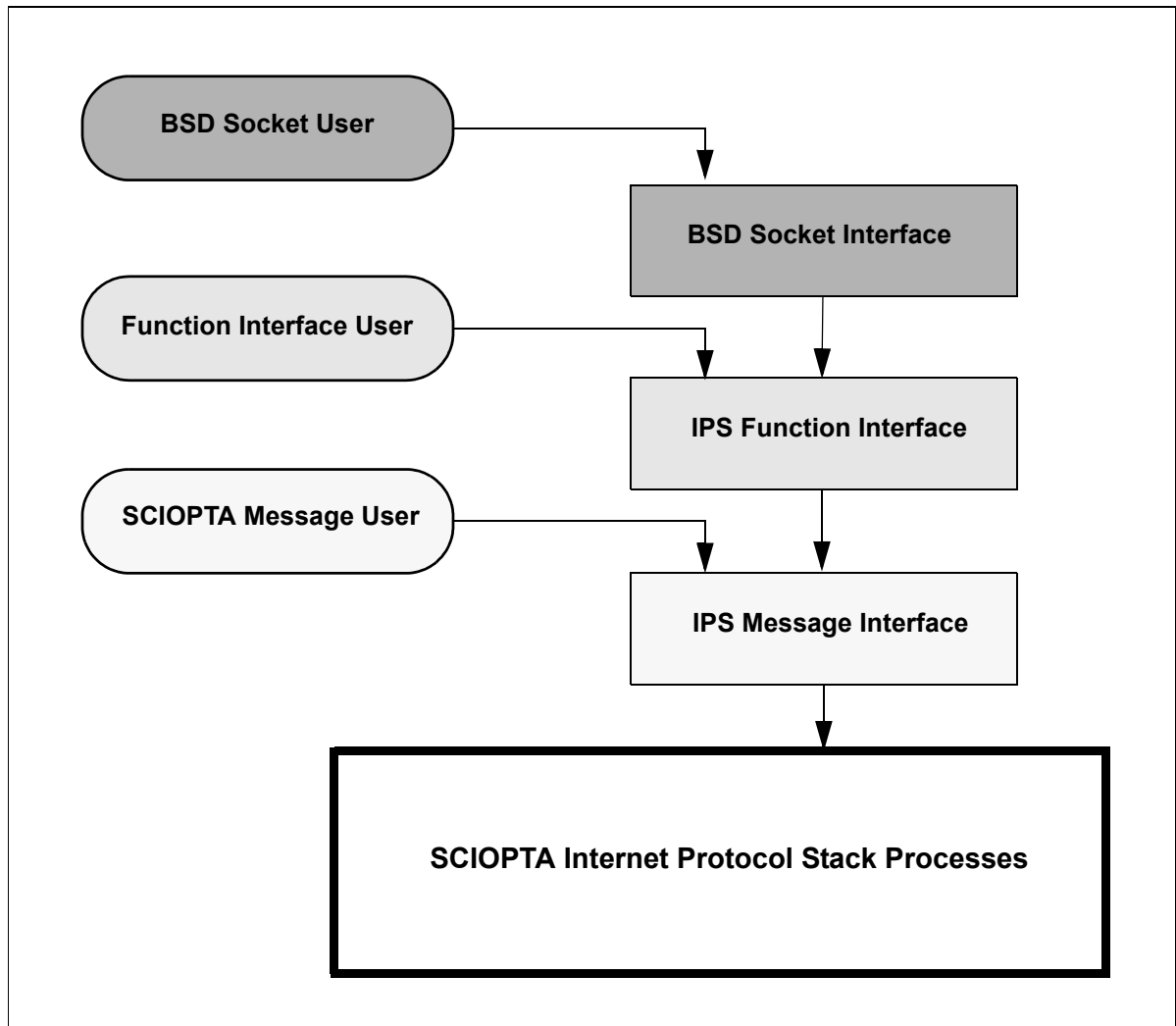
Figure 4-4: SCIOPTA IPS API

4.5.1 IPS Messages

IPS_ACCEPT	Establishes a TCP client-server connection on the server side.
IPS_ACK	Acknowledges a network buffer.
IPS_BIND	Defines a specific IP address and port number where the UDP or TCP protocol driver receives network packages.
IPS_CONNECT	Specifies the remote side of an address connection.
IPS_LISTEN	Activate a listen which is waiting on connection requests of destination peers on the binded source port and IP address.
IPS_ROUTER_ADD	Adds a route in IPS.
IPS_ROUTER_RM	Removes a route in IPS.
IPS_SET_OPTION	Sets options associated with a protocol driver.
SDD_MAN_ADD	Adds a new network device in the device driver system.
SDD_MAN_GET	Get the SDD network device descriptor of a registered network device.
SDD_MAN_RM	Remove a registered network device in the device driver system.
SDD_NET_RECEIVE	Receives network data from a network device driver.
SDD_NET_RECEIVE_URGENT	Receives very urgent network data from a network device driver.
SDD_NET_SEND	Sends network data to a network device driver.
SDD_OBJ_INFO	Gets information from a network device driver.

4.5.2 IPS Functions

ips_accept()	Establishes a TCP client-server connection on the server side.
ips_bind()	Defines a specific IP address and port number where the UDP or TCP protocol driver receives network packages.
ips_connect()	Initiates a connection on a access handle.
ips_devGetByHwAddr()	Gets an SDD network device descriptor for the given hardware MAC address.
ips_devGetByName()	Get an SDD network device descriptor for the given name.
ips_devRegister()	Register an SDD network device descriptor at the network device manager process SCP_netman.
ips_devUnregister()	Removes an SDD network device descriptor from the network device manager process SCP_netman.
ips_getOption()	Gets the options associated with a access handle of a SDD protocol descriptor.
ips_ipv4ArpAdd()	Adds a static ARP entry.
ips_ipv4ArpRM()	Removes a static ARP entry.

ips_ipv4GetProtocol()	Gets an SDD protocol descriptor.
ips_ipv4RouteAdd()	Adds a route in IPS.
ips_ipv4RouteRm()	Removes a route in IPS.
ips_listen()	Activates a listen which is waiting on connection requests of destination peers on the binded source port and IP address.
ips_netbufAck()	Acknowledges a netbuffer.
ips_setOption()	Sets the options associated with a access handle of an SDD protocol descriptor.

4.5.3   BSD Socket Functions

accept()	Accepts a connection on a socket.
bind()	Binds a name to the socket.
close()	Close a BSD descriptor.
connect()	Initiates a connection on a socket.
dup()	Duplicates a BSD descriptor.
gethostbyname()	Returns a structure of type hostent for the given host name.
gethostbyaddr()	Returns the host names in struct hostent for the given host address.
getpeername()	Returns the name of a connected peer.
getsockname()	Returns the socket name.
getsockopt()	Gets the options associated with a socket.
inet_aton()	Converts the Internet host address from the standard numbers-and-dots notation into binary data in net byte order and stores it.
inet_ntoa()	Converts the Internet host address given in network byte order to a string in standard numbers-and-dots notation.
ips_ack()	Acknowledges a network buffer and allows a flow control in the IPS stack.
ips_alloc()	Allocates a network buffer message.
ips_getpeername()	Retrieves the sender (peer) of a netbuffer.
ips_send()	Sends a netbuffer to the specified BSD socket descriptor.
listen()	Listen for connections on a socket.
setsockopt()	Sets the options associated with a socket.
socket()	Creates an endpoint for communication.

5   Distributed Systems

5.1   Introduction

SCIOPTA is a message based real-time operating system and therefore very well adapted for designing distributed multi-CPU systems. Message based operating systems where initially designed to fulfil the requirements of distributed systems.

In this manual we will use the term **SCIOPTA System** to designate a whole target environment on one CPU.

5.2   CONNECTORS

CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA Systems. There may be more than one CONNECTOR process in a system or module. CONNECTOR processes can be seen globally inside a SCIOPTA system by other processes. The name of a CONNECTOR process must be identical to the name of the remote target system.
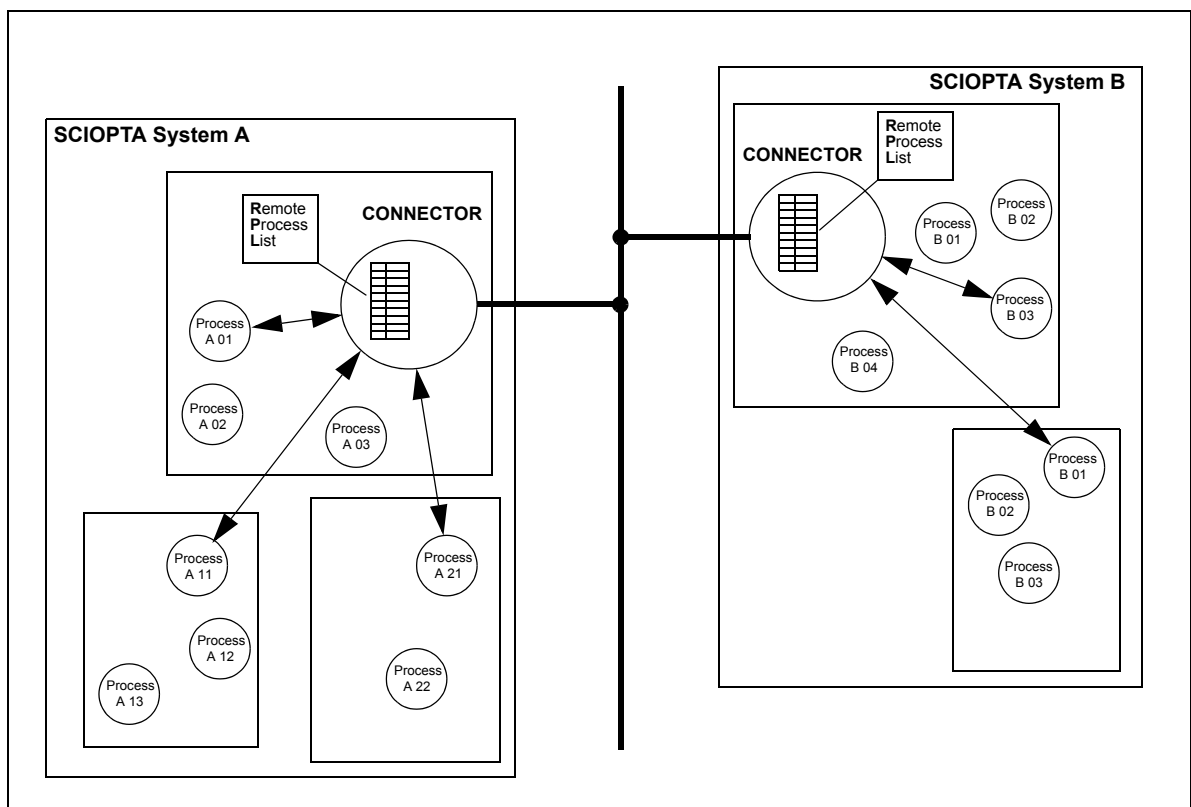
Figure 5-1: SCIOPTA Distributed System

A connector process can be defined as default connector process. There can be only one default connector process in a system and it can have any name.

SCIOPTA

5.3   Transparent Communication

If a process in one system (CPU) wants to communicate with a process in another system (CPU) it first will search for the remote process by using the **sc_procIdGet()** system call. The parameter of this call includes the process name and the path to where to find it in the form: system/module/procname.The kernel transmits a message to the connector including the inquiry.

All connectors start communicating to search for the process. If the process is found in the remote system the connector will assign a free pid for the system, add it in a remote process list and transmits a message back to the kernel including the assigned pid. The kernel returns the pid to the caller process.

The process can now transmit and receive messages to the (remote) pid as if the process is local. A similar remote process list is created in the connector of the remote system. Therefore the receiving process in the remote system can work with remote systems the same way as if these processes where local.

If a message is sent to a process on a target system which does not exist (any more), the message will be forwarded to the default connector process.

5.4   Observation

Communication channels between processes in SCIOPTA can be observed no matter if the processes are local or distributed over remote systems. The process calls **sc_procObserve()** which includes the pointer to a return message and the pid of the process which should be observed.

If the observed process dies the kernel will send the defined message back to the requesting process to inform it. This observation works also with remote process lists in connectors. This means that not only remote processes can be observed but also connection problems in communication links if the connectors includes the necessary functionality.

Figure 5-2: SCIOPTA Observation

SCIOPTA - Real-Time Operating System

6   Index

SCIOPTA - Real-Time Operating System

SCIOPTA - Real-Time Operating System

SCIOPTA - Real-Time Operating System

SCIOPTA - Real-Time Operating System

SCIOPTA - Real-Time Operating System

SCIOPTA

SCIOPTA - Real-Time Operating System