SCIOPTA Architecture Manual

v1.1

Contents

Abstract

This document describes the **SCIOPTA Architecture Manual** for the SCIOPTA Kernels.

Copyright

Disclaimer

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

Trademark

SCIOPTA is a registered trademark of SCIOPTA Systems GmbH.

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 (TCP/IP) including IPS Applications (Web Server, TFTP, FTP, DNS, DHCP, Telnet and SMTP)

- FAT File System

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

1.5 SCIOPTA Reference Manual

The SCIOPTA Reference Manual contains the reference of all system calls in alphabetical order and the SCIOPTA error handling .

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 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.
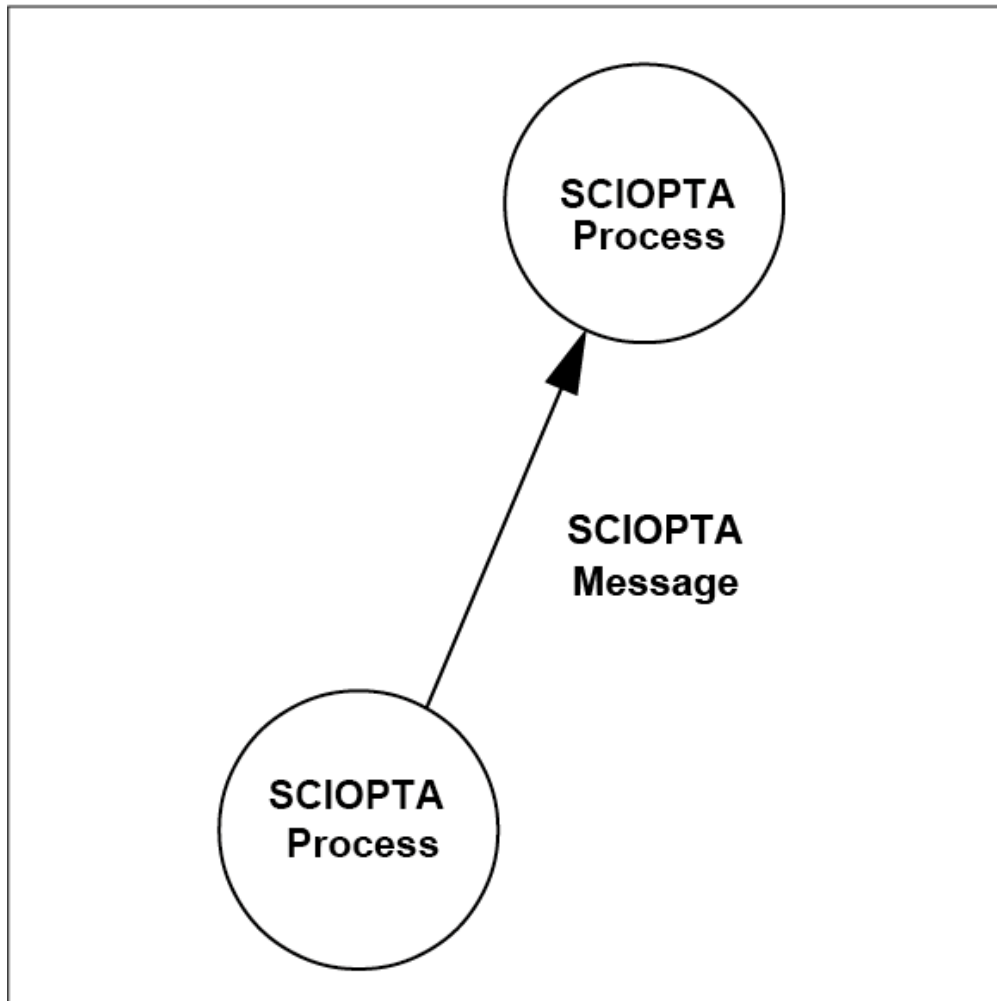
Figure 1. Messag Based RTOS

A typical system controlled by **SCIOPTA** consists of a number of more or less independent tasks called processes. Each process can be seen as if it has 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 is used by processes to recognize the different message and also allows to select which messages to receive from the message queue FIFO. All other messages are kept back in the message queue of the process.

Messages are dynamically allocated from a message pool.
There are three Kernels within **SCIOPTA**, please see chapter "SCIOPTA Kernels" in the Kernel Reference Manual for more information.

2.1 SCIOPTA Kernel V2

SCIOPTA Real-Time Kernel V2 (Version 2) is the successor to the **SCIOPTA** standard kernel V1 (Version 1).

Differences between **SCIOPTA** Kernel V2 and V1:

- Module priority: No process inside a module is allowed to have a higher priority than modules's maximum priority.

- Module friendship concept has been removed.

- System calls `sc_procPathGet` and `sc_procNameGet` return `NULL` if PID valid, but does not exist anymore.

- Time slice for prioritized processes is now fully supported and documented.

- The system call `sc_procSliceSet` is only allowed within same module.

- Parameter n in system call `sc_procVarInit` is now real maximum number of process variables. It was n+1 before.

- The activation time is set when a process becomes ready.

- The activation time is saved for `sc_msgRx` in prioritized processes.

- System call `sc_sleep` returns now the activation time.

- The error handling has been modified.

- Error hook API changed.

- Error process and error proxy introduced.

- New system calls:

  - `sc_moduleCreate2` call replaces the call sc_moduleCreate. It gets the module parameters now from a module descriptor block (mdb).

  - `sc_modulePrioGet` Returns the priority of a module.

  - `sc_moduleStop` Stops a whole module.

  - `sc_procAtExit` Register a function to be called if a prioritized process is killed.

  - `sc_procAttrGet` Returns specific process attributes.

  - `sc_procCreate2` call replaces the calls `sc_procPrioCreate`, `sc_procIntCreate` and `sc_procTimCreate`. The process parameters now defined in a process descriptor block (pdb).

  - `sc_msgHdChk` Integrity check of message header.

  - `sc_msgFind` Finds messages which have been allocated or already received.

  - `sc_tickActivationGet` Returns the tick time of last activation of the calling process.

  - `sc_miscCrc32` Calculates a 32 bit CRC over a specified memory range.

  - `sc_miscCrc32Contd` Calculates a 32 bit CRC over an additional memory range.

3 Modules

3.1 Introduction

SCIOPTA allows you to group processes into functional units called modules. Very often you want to decompose a complex application into smaller units which you can realize in **SCIOPTA** by using modules. This will improve system structure. A **SCIOPTA** process can only be created from within a module.

A typical example would be to encapsulate a whole communication stack into one module and to protect it against other function modules in a system.

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

3.2 System Module

There is always one static module in a **SCIOPTA** system.

This module is called system module (sometimes also named module 0) and is automatically created by the kernel at system start.

3.3 Module Priority

SCIOPTA modules contain a (module) priority.

3.3.1 Kernel V1

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

The effective priority has an upper limit of 31 which will never be exceeded even if the addition of module priority and process priority is higher. This technique assures that the process with highest process priority (0) cannot disturb processes in modules with lower module priority (module protection).

3.3.2 Kernels V2 and V2INT

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.

This guarantees that modules with low priority tasks do not interrupt high priority modules.

3.4 System Protection

In larger systems it is often necessary to protect certain areas to be accessed by others. In **SCIOPTA** the user can achieve this 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 laid down at module boundaries.

System protection and MMU support is optional in **SCIOPTA**.

3.5 SCIOPTA Module Friend Concept

3.5.1 Kernel V1

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.

A module can be declared as friend by the <u>sc_moduleFriendAdd</u> 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 <u>sc_moduleFriendAdd</u> 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.

3.5.2 Kernels V2 and V2INT

Not supported!

3.6 Module Creation

3.6.1 Static Module Creation

Static modules are modules which are automatically created when the systems boots up. They are defined in the SCONF configuration tool. Please consult the CPU-Specific **SCIOPTA** System Manuals for more information.
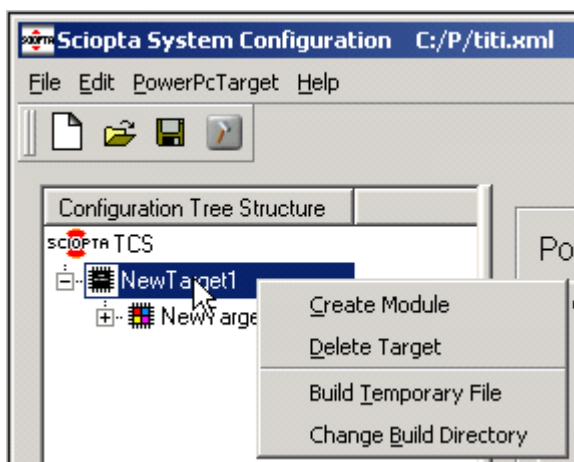
Figure 2. Static Module Creation with SCONF Tool

3.6.2 Dynamic Module Creation

Modules can also be created dynamically by the <u>sc_moduleCreate</u> (Kernel V1) or the <u>sc_moduleCreate2</u> (Kernels V2 and V2INT) system calls.

Dynamic Module Creation Kernel V1

```
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
);
```

Dynamic Module Creation Kernels V2 and V2INT

```
sc_moduleid_t sc_moduleCreate2(
    sc_mdb_t *mdb        // Pointer to the module descriptor block
);
```

4 Processes

4.1 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 process priority 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).

All **SCIOPTA** processes have system wide unique process identities.

A **SCIOPTA** process is always part of a **SCIOPTA** module. Please consult chapter Modules for more information about **SCIOPTA** modules.

4.2 Process States

A process running under **SCIOPTA** is always in the **RUNNING**, **READY** or **WAITING** state.
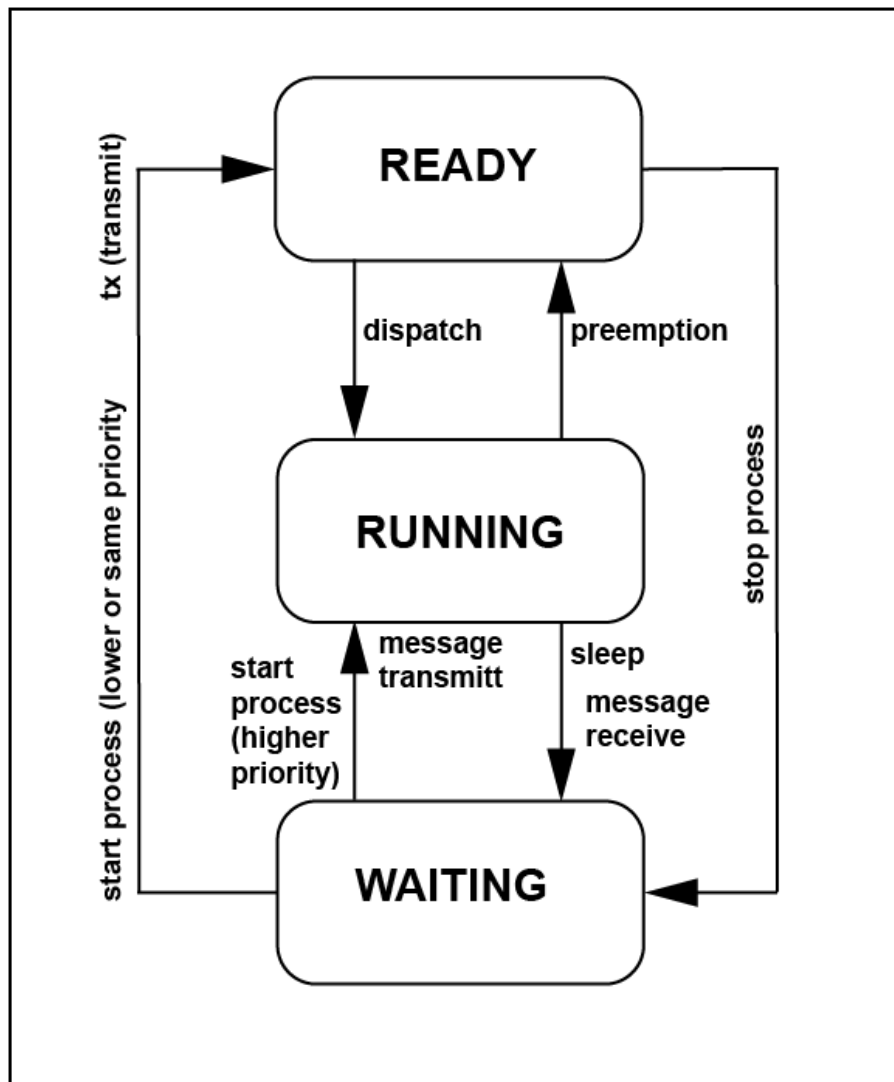
Figure 3. State Diagram of SCIOPTA Kernel

4.2.1 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.

4.2.2 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.

4.2.3 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 waits for a delay to expire.

- The process waits on a SCIOPTA trigger.

- The Process waits to be started.

4.3 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. At start-up the kernel puts all static created processes into READY or WAITING (stopped) state.

Static processes 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.
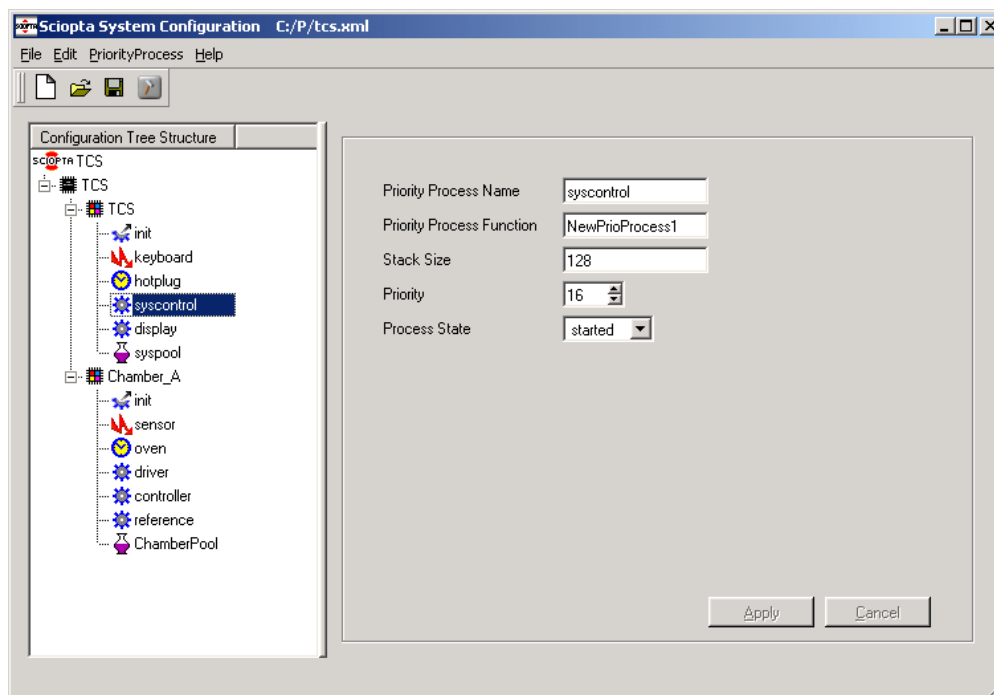
Figure 4. Process Configuration Window for Static Processes

4.4 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.

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.

Create Process System Call (prioritzed process for Kernel V1)

```
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
);
```

Create Process System Call (Kernel V2)

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

4.5 Process Identity

Each process has a unique process identity (process ID) which is used in **SCIOPTA** system calls when processes need to be addressed.

The process ID will be allocated by the operating system for all processes which you have entered during **SCIOPTA** configuration (static processes) or will be returned when you are creating processes dynamically. The kernel maintains a list with all process names and their process IDs.

The user can get Process IDs by using the sc_procIdGet system call including the process name.

4.6 Prioritized Processes ⚙

In **SCIOPTA** a process can be seen as independent tasks. **SCIOPTA** guarantees that always the most important process at a certain moment is executing. Each prioritized process has a priority and the **SCIOPTA** scheduler is giving CPU time to processes according to these priorities. The process with higher priority runs (gets the CPU) 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.

Most of the time in a **SCIOPTA** real-time system is spent in prioritized processes. It is where collected data is analysed and complicated control structures are executed.

Prioritized processes respond much slower than interrupt processes, but they can spend a relatively long time to work with data.

4.6.1 Creating and Declaring Prioritized Processes

Static prioritized processes are defined in the **SCIOPTA** configuration utility (SCONF) and created by the kernel automatically at system startup. Please consult the CPU-Specific **SCIOPTA** System Manuals for more information.

Dynamic prioritized processes are created by using the sc_procPrioCreate (Kernel V1) or the sc_procCreate2 (Kernels V2 and V2INT) system call and killed dynamically with the sc_procKill system call.

4.6.2 Process Priorities

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

By assigning a priority 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. Subsystems 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.

Kernel V1:

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

Kernels V2 and V2INT:

The process priority cannot be higher than the module priority of the module where the process resides.

See also chapter "[Module Priority](#)";

4.6.3 Writing Prioritized Processes

4.6.3.1 Process Declaration Syntax

All prioritized processes in **SCIOPTA** *must contain the following declaration:*

```
SC_PROCESS(<proc_name>)
{
  for (;;)
  {
    /* Code for process <proc_name> */
  }
}
```

4.6.3.2 Process Template

```
#include <sciopta.h>   /* SCIOPTA standard prototypes and definitions */

SC_PROCESS(proc_name)  /* Declaration for prioritized process proc_name */
{
  /* Local variables */

  /* Process initialization code */

  for (;;)  /* "for-ever"-loop declaration. */
  {         /* A SCIOPTA prioritized process may never return */

            /* It is an error to terminate a prioritized process */
            /* If a prioritized process terminates and returns   */
            /* the SCIOPTA kernel will produce an error condition */
            /* and call the SCIOPTA error hook */


            /* Code for process proc_name */

  }
}
```

4.7 Interrupt Processes ⚡

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

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

Interrupt process is the fastest process type in **SCIOPTA** and will respond almost immediately to events. As the system is blocked during interrupt handling interrupt processes must perform their task in the shortest time

possible. By default, **SCIOPTA** does not allow interrupt nesting. If the user application needs it, the interrupts must be released inside the interrupt process. Consult the **SCIOPTA** Systems Manuals on how this can be done.

A typical example is the control of a serial line. Receiving incoming characters might be handled by an interrupt process by storing the incoming arrived characters in a local buffer returning after each storage of a character. If this takes too long characters will be lost. If a defined number of characters of a message have been received the whole message will be transferred to a prioritized process which has more time to analyse the data.

4.7.1 Creating and Declaring Interrupt Processes

Static interrupt processes are defined in the **SCIOPTA** configuration utility (SCONF) and created by the kernel automatically at system startup.

> **Kernel V1:**
> Dynamic interrupt process are created by using the `sc_procIntCreate` system call and killed dynamically with the `sc_procKill` system call.

> **Kernels V2 and V2INT:**
> Dynamic interrupt process are created by using the `sc_procCreate2` system call and killed dynamically with the `sc_procKill` system call.

4.7.2 Interrupt Process Priorities

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

4.7.3 Writing Interrupt Processes

4.7.3.1 Interrupt Process Declaration Syntax

```
SC_INT_PROCESS(<proc_name>, <irq_src>)
{
  /* Code for interrupt process <proc_name> */
}

SC_INT_PROCESS_EX(<proc_name>, <irq_src>, <vector>)
{
  /* Code for interrupt process <proc_name> */
}
```

4.7.3.2 Interrupt Source Parameter irq_src

This parameter is set by the kernel depending on the interrupt source.

4.7.3.3 Interrupt Source Parameter Values

SC_PROC_WAKEUP_HARDWARE	The interrupt process is activated by a real hardware interrupt.
SC_PROC_WAKEUP_MESSAGE	The interrupt process is activated by a message sent to the interrupt process.
SC_PROC_WAKEUP_TRIGGER	The interrupt process is activated by a trigger event.
SC_PROC_WAKEUP_START	The interrupt process is activated when the process is started. (Only for Kernels V2 and V2INT)

SC_PROC_WAKEUP_CREATE	The interrupt process is activated when the process is created. This allows the interrupt process to execute some initialization code.
SC_PROC_WAKEUP_KILL	The interrupt process is activated when the process is killed. This allows the interrupt process to execute some exit code.
SC_PROC_WAKEUP_STOP	The interrupt process is activated when the process is stopped. (Only for Kernels V2 and V2INT).
SC_PROC_WAKEUP_CALLBACK	The callback is called. (Only SCIOPTA Simulator)

4.7.3.4 Interrupt Vector Parameter vector

This is the hardware interrupt vector which did activate the interrupt process.

See sc_procIrqRegister for more information.

4.7.3.5 Interrupt Process Template for Kernel V1

```c
#include <sciopta.h> /* SCIOPTA standard prototypes and definitions */

SC_INT_PROCESS(proc_name, irq_src)/* Declaration for interrupt process proc_name */
{
  /* Local variables */

  switch (irq_src) {

  case SC_PROC_WAKEUP_HARDWARE: /* Generated by hardware */
    /* Code for hardware interrupt handling */
  break;

  case SC_PROC_WAKEUP_CREATE:   /* Generated when process created */
    /* Initialization code */
  break;

  case SC_PROC_WAKEUP_KILL:     /* Generated when process killed */
    /* Exit code */
  break;

  case SC_PROC_WAKEUP_MESSAGE:  /* Generated by a message sent to */
                                /* this interrupt process */
    /* Code for receiving a message */
  break;

  case SC_PROC_WAKEUP_TRIGGER:  /* Generated by a SCIOPTA trigger event */
    /* Code for trigger event handling */
  break;

  default:
    /* Error handling sc_miscError() */
  break;
  }
}
```

4.7.3.6 Interrupt Process Template for Kernels V2 and V2INT

```c
#include <sciopta.h> /* SCIOPTA standard prototypes and definitions */

SC_INT_PROCESS(proc_name, irq_src)/* Declaration for interrupt process proc_name */
{
  /* Local variables */

  switch (irq_src) {

  case SC_PROC_WAKEUP_HARDWARE: /* Generated by hardware */
    /* Code for hardware interrupt handling */
  break;

  case SC_PROC_WAKEUP_CREATE:   /* Generated when process created */
    /* Initialization code */
  break;

  case SC_PROC_WAKEUP_KILL:     /* Generated when process killed */
    /* Exit code */
  break;
```

```
    case SC_PROC_WAKEUP_MESSAGE:  /* Generated by a message sent to */
                                  /* this interrupt process */
        /* Code for receiving a message */
    break;

    case SC_PROC_WAKEUP_TRIGGER:  /* Generated by a SCIOPTA trigger event */
        /* Code for trigger event handling */
    break;

    case SC_PROC_WAKEUP_STOP:     /* Generated when process is stopped */
        /* Stop code */
    break;

    case SC_PROC_WAKEUP_START:    /* Generated when process is started */
        /* Start code */
    break;

    default:
        /* Error handling sc_miscError() */
    break;
    }
}
```

4.8 Timer Processes ⏰

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.

When configuring or creating a timer process, the user defines the period in numer of ticks.

Timer processes will be used for tasks which need to be executed at precise periodic intervals.

As the timer process runs on interrupt level it is as important as for normal interrupt processes to return as fast as possible.

4.8.1 Creating and Declaring Timer Processes

Static timer processes are defined in the **SCIOPTA** configuration utility (SCONF) and created by the kernel automatically at system startup.

> **Kernel V1:**
> Dynamic interrupt process are created by using the sc_procTimCreate system call and killed dynamically with the sc_procKill system call.

> **Kernels V2 and V2INT:**
> Dynamic interrupt process are created by using the sc_procCreate2 system call and killed dynamically with the sc_procKill system call.

4.8.2 Timer Process Priorities

The timer processes are bound to the system tick. Therefore they run with the same hardware priority as the tick interrupt.

4.8.3 Writing Timer Processes

Timer processes are written exactly the same way as interrupt processes. Please consult chapter "Writing Interrupt Processes" for information how to write interrupt processes.

Unlike interrupts, timer processes can be stopped and started Kernel V1 and Kernel V2/V2INT.

4.9 Init Processes 🛩

The init process is the first process in a module. Each module has at least one process and this is the init process.

At module start the init process gets automatically the highest priority (0). After the init process has done some work, the user can change the process priority. If the user does not change the priority, **SCIOPTA** it will set the priority to a specific lowest level (32) and enter an endless loop.

The init process acts therefore also as idle process which will run when all other processes of a module are in the waiting state.

4.9.1 Creating and Declaring Init Processes

In static modules the init process is written, created and started automatically. Static modules are defined and configured in the SCONF configuration utility. The code of the init process is generated automatically by the SCONF configuration tool and included in the file sconf.c. The init process function name will be set automatically by the kernel in sconf.c to: <module_name>_init. The init process of the system module will create all static **SCIOPTA** objects such as other modules, processes and pools.

In dynamic modules the init process is also created and started automatically. But the code of the init process must be written by the user. The entry point of the init process is given as parameter of the dynamic module create system calls. Please see below for more information how to write init processes for dynamic modules.

4.9.2 Init Process Priorities

At start-up the init process gets the highest priority (0).

After the init process has done its work it will change its priority to a specific lowest level (32) and enter an endless loop.

Priority 32 is only allowed for the init process. All other processes are using priority 0 - 31.

4.9.3 Writing Init Processes

Statically created init processes call a user function (module hook) which is named like the module. This user function can be empty or does not have to return.

Example: Module "dev", module hook:

```
void dev()
{
  sc_procPrioSet(31);
  for(;;){
    heartBeat();
  }
}
```

Only init processes of dynamic modules must be written by the user. The entry point of the init process is given as parameter of the dynamic module create system calls. At start-up the init process gets the highest priority (0). The user must set the priority to 32 at the end of the init process code.

Template of a minimal init process of a dynamic module:

```
SC_PROCESS(dynamicmodule_init)
{
    /* Important init work on priority level 0 can be included here */
  sc_procPrioSet(32);
  for(;;)  ASM_NOP;  /* init is now the idle process */
}
```

4.10 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.

4.10.1 Process Daemons

The process daemon (sc_procd) is identifying processes by name and supervises created and killed processes.

Whenever you are using the sc_procIdGet system call you need to start the process daemon.
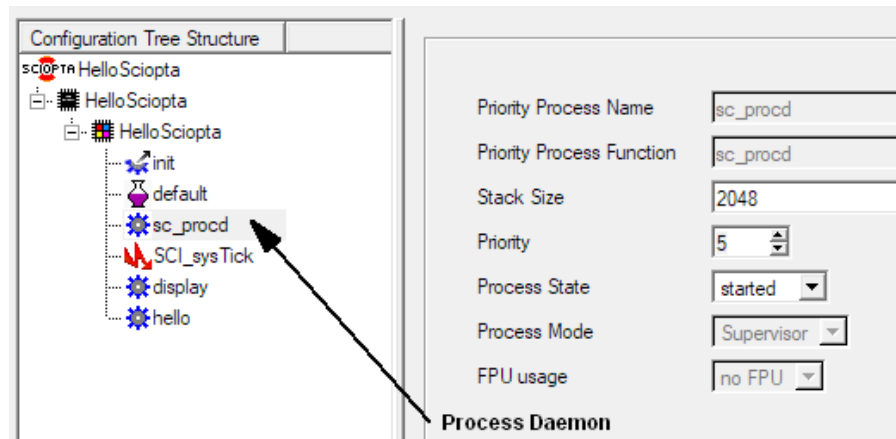
Figure 5. Process Daemon Declaration in SCONF

The process daemon is part of the kernel. But to use it you need to define and declare it in the SCONF configuration utility.

The process daemon can only be created and placed in the system module.

4.10.2 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.

Whenever you are using process or module create or kill system calls you need to start the kernel daemon.
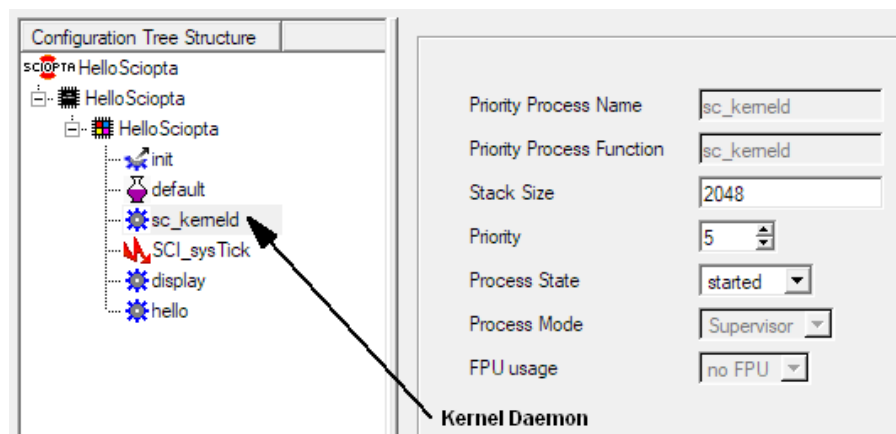
Figure 6. Kernel Daemon Declaration in SCONF

The kernel daemon is part of the kernel. But to use it you need to define and declare it in the SCONF configuration utility.

The kernel daemon can only be cleared and placed in the system module.

4.11 Process Stacks

When creating processes either statically in the SCONF configuration tool or dynamically with the sc_procPrioCreate, sc_procIntCreate, sc_procTimCreate in Kernel V1 or sc_procCreate2 in Kernel V2 and V2INT system calls you always need to give a stack size.

All process types (init, interrupt, timer, prioritized and daemon) need a stack.

The stack size given must be big enough to hold the call stack and the maximum used local data in the process.

When you start designing a system it is good design practice to define a the stack as large as possible. In a later stage you can measure the used stack with the **SCIOPTA** DRUID system level debugger or the sc_procAttrGet system call (Kernel V2 and V2INT only) and reduce the stacks if needed.

4.11.1 Unified Interrupt Stack for ARM Architecture

Only for Kernel V1:
For the ARM architecture a unified interrupt stack can be used in interrupt and timer processes. In this case all interrupt and timer processes share the same stack.

The "unified IRQ stack" checkbox must be selected in the system configuration window of the SCONF utility to enable this feature.

The stack size given must be big enough to hold the stacks of the interrupt processes with the biggest stack needs taken in account the interrupt nesting.

4.11.2 Interrupt Nesting for ARM Architecture

Only for Kernel V1:
If interrupt process nesting is used in the ARM architecture, the maximum nesting level of interrupt processes must be declared in the system configuration (SCONF).

4.12 Stack Protector

Only for Kernel V2 and V2INT on PowerPC:
In a **SCIOPTA** system you can enable a stack protection called stack protector.

This is a function supplied by the compiler manufacturer which checks if the stack frame of a called function still fits in the stack. The compiler will add a function prologue to each function which will be stack protected.

For the Windriver PowerPC compiler a function using this feature must be compiled with the -Xstack-probe switch.

SCIOPTA needs to know if stack protector is used as it allocates a global variable for this purpose. You can enable stack protector in the SCONF configuration tool.

4.13 Addressing Processes

4.13.1 Introduction

In a typical **SCIOPTA** design you need to address processes. For example you want to

- send **SCIOPTA** messages to a process
- kill a process
- get a stored name of a process
- observe a process
- get or set the priority of a process
- start and stop processes

In **SCIOPTA** you are addressing processes by using their process ID (pid). There are two methods to get process IDs depending if you have to do with static or dynamic processes.

4.13.2 Get Process IDs of Static Processes

Static processes are created by the kernel at start-up. They are designed with the **SCIOPTA** SCONF configuration utility by defining the name and all other process parameters such as priority and process stack sizes.

You can address static process by appending

_pid

to the process name if the process resides in the system module. If the static process resides inside another module than the system module, you need to precede the process name with the module name and an underscore in between.

For instance if you have a static process defined in the system module with the name controller you can address it by giving controller_pid. To send a message to that process you can use:

```
sc_msgTx(mymsg, controller_pid, SC_MSGTX_NO_FLAG);
```

If you have a static process in the module tcs (which is not the system module) with the name display you can address it by giving tcs_display_pid. To send a message to that process you can use:

```
sc_msgTx(mymsg, tcs_display_pid, SC_MSGTX_NO_FLAG);
```

4.13.3 Get Process IDs of 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 process IDs of dynamic processes can be retrieved by using the system call sc_procIdGet.

The process creation system calls sc_procPrioCreate, sc_procIntCreate and sc_procTimCreate in **Kernel V1** and the sc_procCreate2 in **Kernel V2 and V2INT** will also return the process IDs which can be used for further addressing.

4.14 Process Variables

Each process can store local variables inside a protected data area. Process variables are variables which can only be accesses by functions wthin the context of the process.

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.
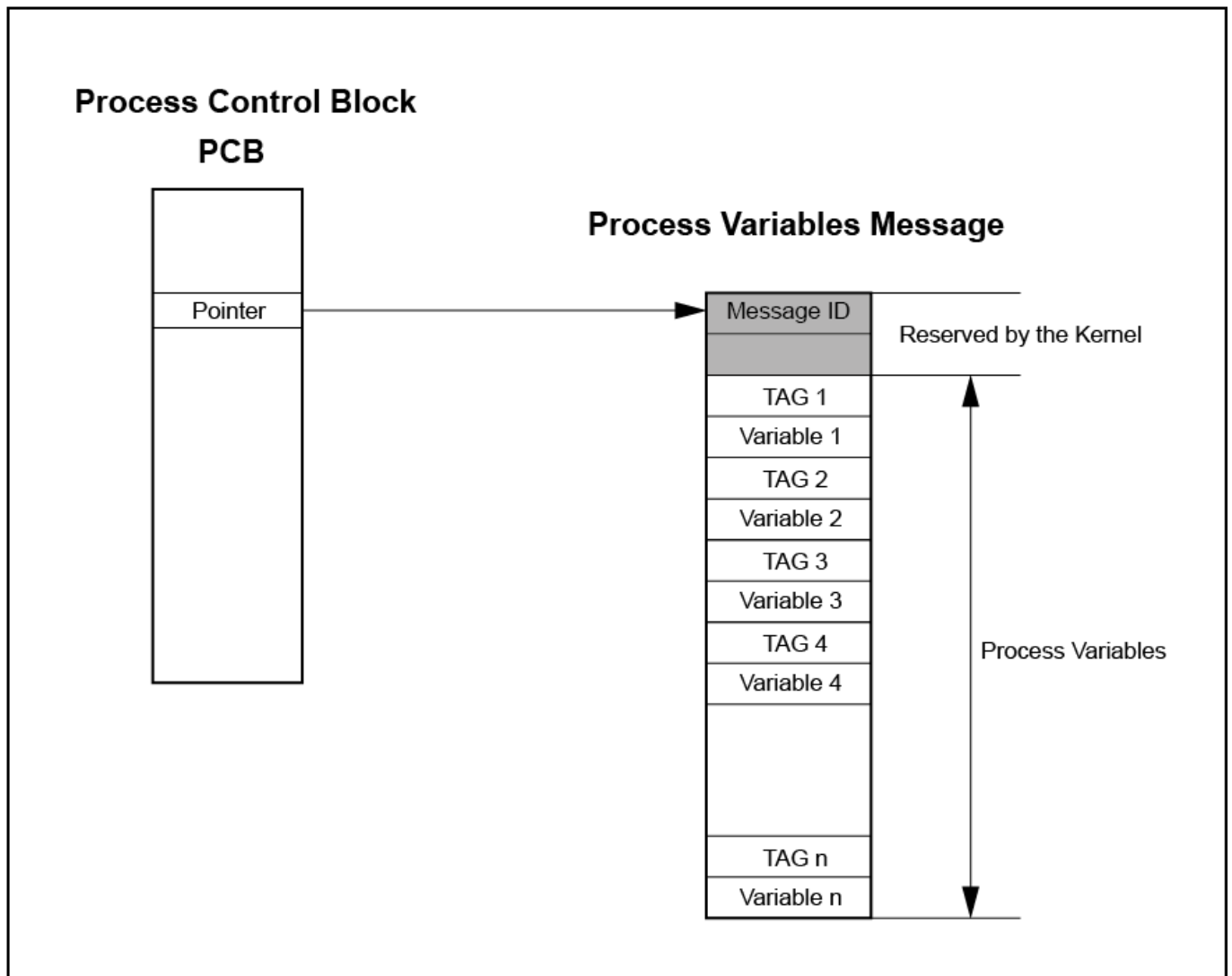
Figure 7. SCIOPTA Process Variables

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.

4.15 Process 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.
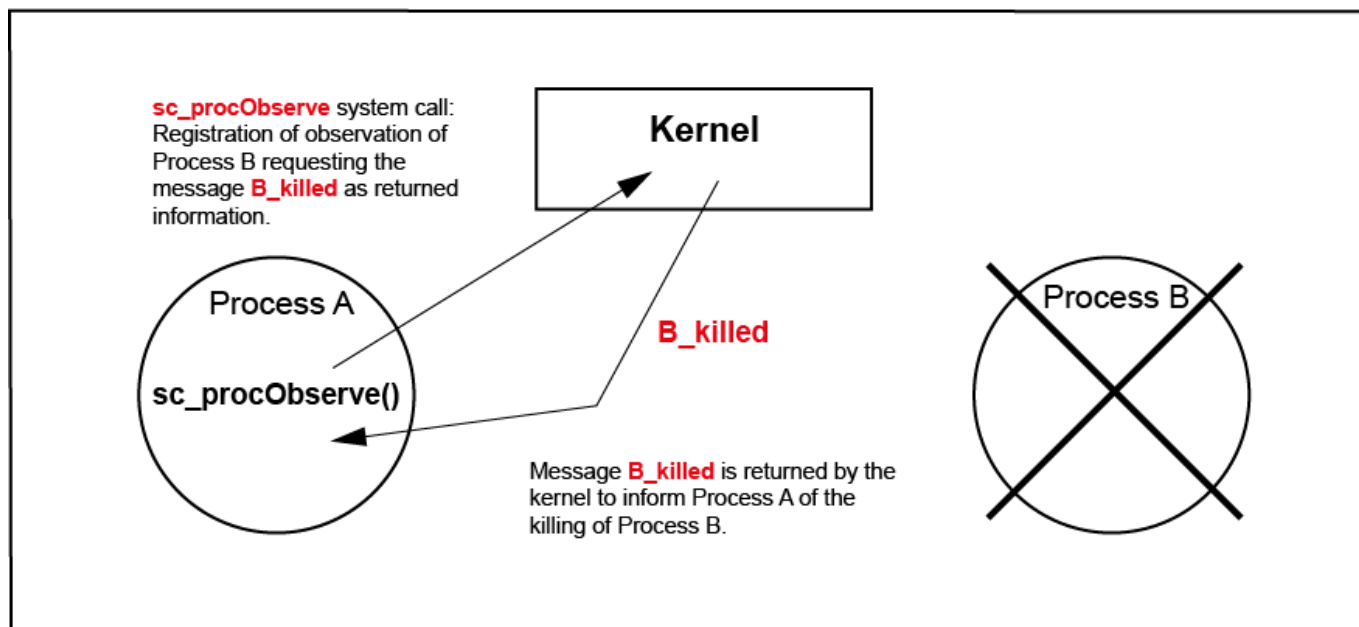
Figure 8. SCIOPTA Process Observation

5 Messages

5.1 Introduction

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

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

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

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

5.2 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 is a hidden data structure which will be used by the kernel. The user can access this information by specific **SCIOPTA** system calls. The following information are stored in the message header:

- Process ID of message owner
- Message size
- Process ID of transmitting process
- Process ID of addressed process
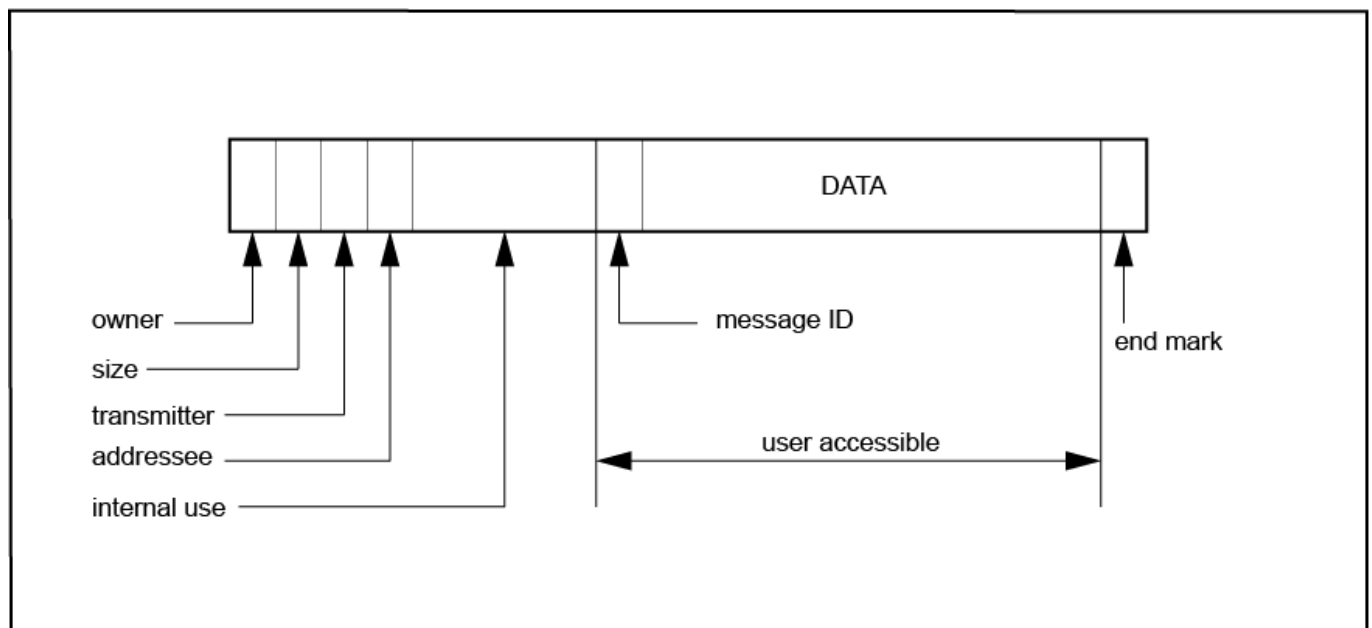
Figure 9. SCIOPTA Message Structure

When a process allocates a message it becomes the owner of the message. If the process transmits the message to another process, then this one becomes the 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.

Each process has a message queue for incoming and one for owned messages. Messages are not moved into these queues but rather linked to it.

5.3 Message Size

The user may allocate messages with any arbirtrary number of bytes. The returned message may be larger as the kernel chooses the best fitting message from a list of 4,8 or 16 different buffer sizes. These are defined when the pool is created.

The difference of requested bytes and returned bytes can not be accessed by the user and will be unused at this moment. 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 optimise the buffer sizes.

5.3.1 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.

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.

5.4 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 free the message and return it to the pool. After this the process may no longer access the message.

Please consult "Pools" for more information about message pools.

5.5 Message Passing

Message passing is the favourite method for interprocess communication in **SCIOPTA**. Contrary to mailbox interprocess 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.

The sc_msgAlloc or the sc_msgRx calls 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 kernel and puts the message in the queue of the receiver process. 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. After reception, the process becomes owner of the process and gets the pointer to access the message contents. The `sc_msgRx` 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 shall be returned to the system by `sc_msgFree`.
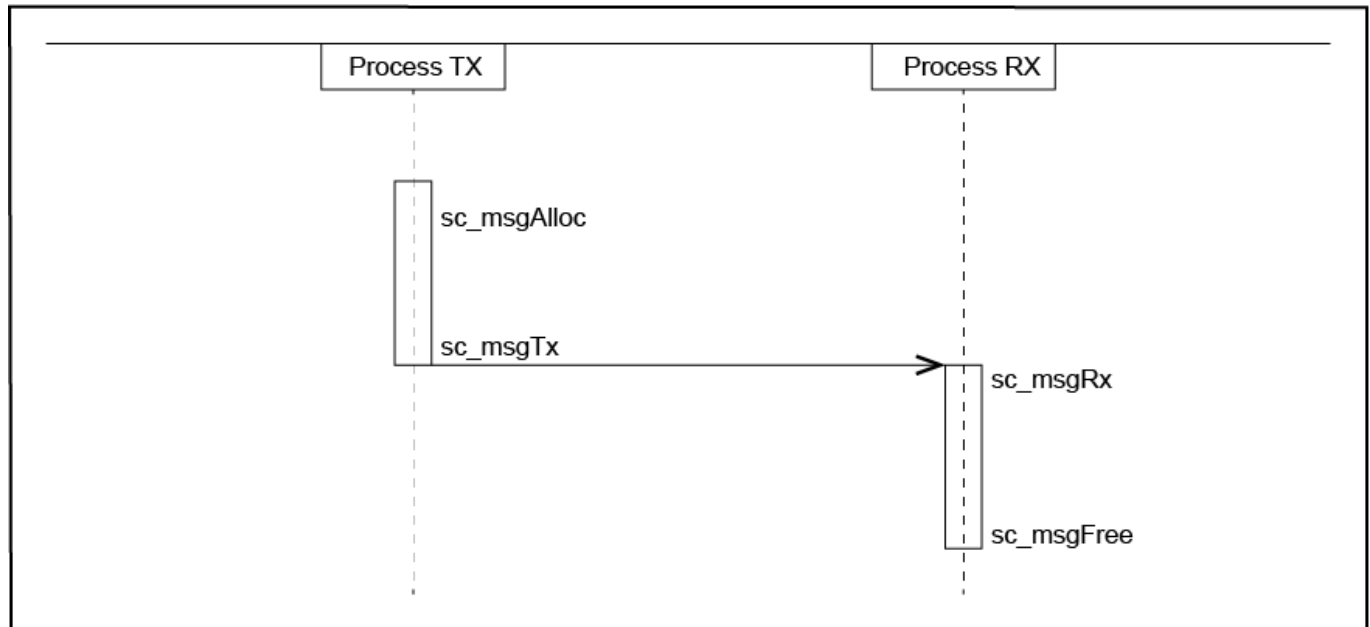
Figure 10. SCIOPTA Message Passing

5.6 Message Declaration

The following method for declaring, accessing and writing message buffers minimizes the risk for bad message accesses and provides standardized code which is easy to read and to reuse.

Very often designers of message passing real-time systems are using for each message type a separate message file as include file. Every process can use specific messages by just using a simple include statement for this message.

The **SCIOPTA** message declaration syntax can be divided into three parts:

- Message identifier definition
- Message structure definition
- Message union declaration

5.6.1 Message Identifier

5.6.1.1 Description

The declaration of the message identifier is usually the first line in a message declaration file. The message number can also be described as message class. Each message class should have a unique message number for identification purposes.

We recommend to write the message name in upper case letters.

5.6.1.2 Syntax

```
#define MESSAGE_NAME (<msg_nr>)
```

5.6.1.3 Parameter

msg_nr	Message Identifier (ID).

5.6.2 Message Structure

5.6.2.1 Description

Immediately after the message number declaration usually the message structure declaration follows. We recommend to write the message structure name in lower case letters in order to avoid mixing up with message number declaration.

The message ID (or message number) id must be the first declaration in the message structure. It is used by the **SCIOPTA** kernel to identify **SCIOPTA** messages. After the message ID all structure members are declared. There is no limit in structure complexity for **SCIOPTA** messages. It is only limited by the message size which you are selecting at message allocation.

5.6.2.2 Syntax

```
struct <message_name>
{
  sc_msgid_t   id;
  <member_type> <member>;
      .
      .
};
```

5.6.2.3 Parameter

message_name	Name of the message.
id	This the place where the message number (or message ID) will be stored.
member	Message data member.

5.6.3 Message Union

5.6.3.1 Description

All processes which are using **SCIOPTA** messages should include the following message union declaration.
The union `sc_msg` is used to standardize a message declaration for files using **SCIOPTA** messages.

5.6.3.2 Syntax

```
union sc_msg
{
  sc_msgid_t       id;
  <message_type_1> <message_name_1>
  <message_type_2> <message_name_2>
  <message_type_3> <message_name_3>
      .
      .
};
```

SCIOPTA

5.6.3.3 Parameter

id	Message ID
	Must be included in this union declaration. It is used by the **SCIOPTA** kernel to identify **SCIOPTA** messages.
message_name_n	Messages, the process will use.

5.7 Message Number (ID) Organization

Message numbers (also called message IDs) should be well organized in a **SCIOPTA** project.

5.7.1 Global Message Number Defines File

All message IDs greater than `0x80000000` are reserved for **SCIOPTA** internal messages and should not be used by the application. These messages are defined in the file `defines.h` and `sciopta.msg`. Please consult these files for managing and organizing the message IDs of your application.

defines.h	System wide constant definitions.
	File location: `<installation_folder>\sciopta\<version>\include\ossys\`
sciopta.msg	System wide constant definitions.
	File location: `<installation_folder>\sciopta\<version>\include\`

5.8 Example

```
#define CHAR_MSG (5)

typedef struct char_msg_s
{
  sc_msgid_t id;
  char       character;
} char_msg_t;

union sc_msg
{
  sc_msgid_t id;
  char_msg_t char_msg;
};

SC_PROCESS(keyboard)
{
  sc_msg_t msg; /* Process message pointer */
  sc_pid_t to;  /* Receiving process ID    */

  to = sc_procIdGet ("display", SC_NO_TMO); /* Get process ID       */
                                            /* for process display */

  for (;;)
  {
    msg = sc_msgAlloc(sizeof (char_msg_t),
                 CHAR_MSG,
                 SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);      /* Allocates the message */
    msg->char_msg.character = 0x40;     /* Loads 0x40 */
    sc_msgTx (&msg, to, SC_MSGTX_NO_FLAG); /* Sends message to process display */

    sc_sleep (1000); /* Waits 1000 ticks */
  }
}
```

5.9 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 if the Inter-Module setting in the system configuration utility is set to "always copy". If it set to never copy, messages between modules are not copied.

> **Kernel V1**
>
> A module can be declared as friend of another module. A message sent to a process in a friend module will not be copied. But the returned message will, as the friend ship unilateral. To avoid this the receiver needs to declare the sender also as friend.

To copy such a message the kernel will allocate a buffer from the default pool of the module where the receiving process resides. It must be guaranteed that there is a big enough buffer in the receiving module available to fit the message.
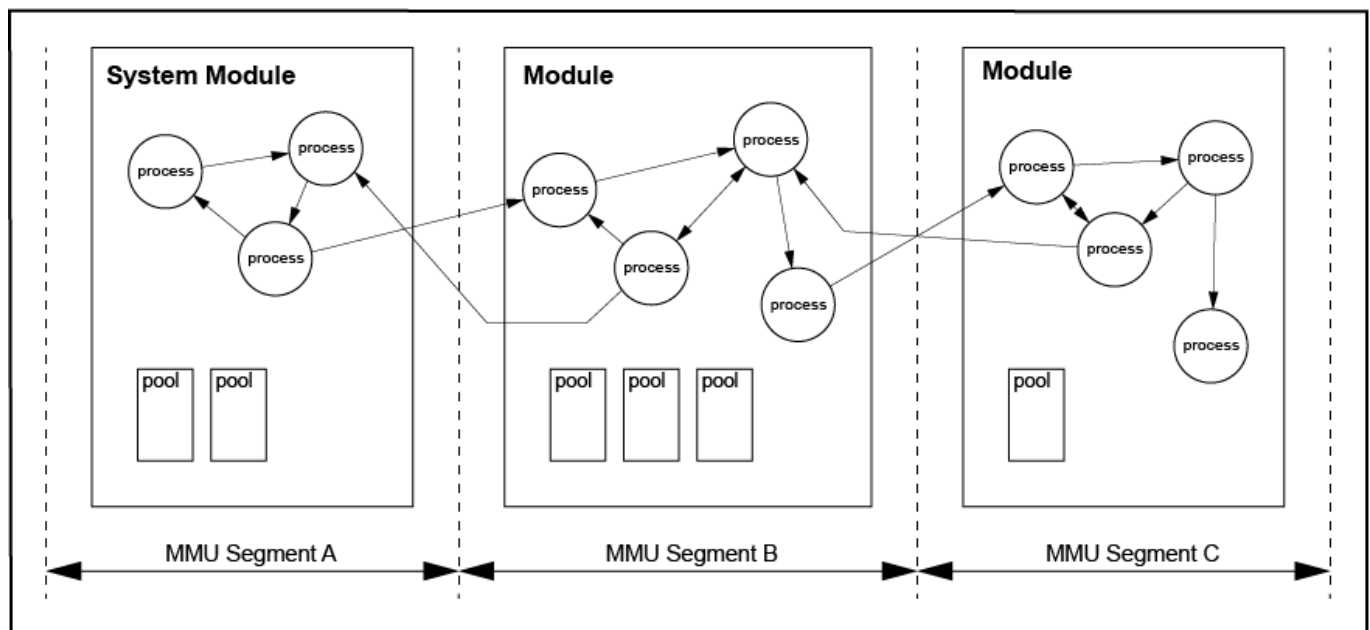
Figure 11. SCIOPTA Messages and Modules

5.10 Returning Sent Messages

There is a specific flag (SC_MSGTX_RTN2SNDR) in the sc_msgTx system call available to get messages back if it was not possible to deliver it.

If this flag is set in sc_msgTx the message will be returned if the addressed process does not exist or there is not enough space in the receiving message pool when sent to another module.

In this case the sender process must receive the message after it has been sent with the sc_msgRx system call.

5.11 Message Passing and Scheduling

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

The process with the highest priority is running (owning the CPU). **SCIOPTA** maintains 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 continue after the interrupt has finished unless a process with a higher priority became ready (e.g. by a message fromt the interrupt process). In this case the CPU is given to the now highest priority process. The pre-emted process will continue where interrupted when it became highest priority process again.

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. There is no forced re-scheduling based on a tick unless requested by the process by setting a time-slice.
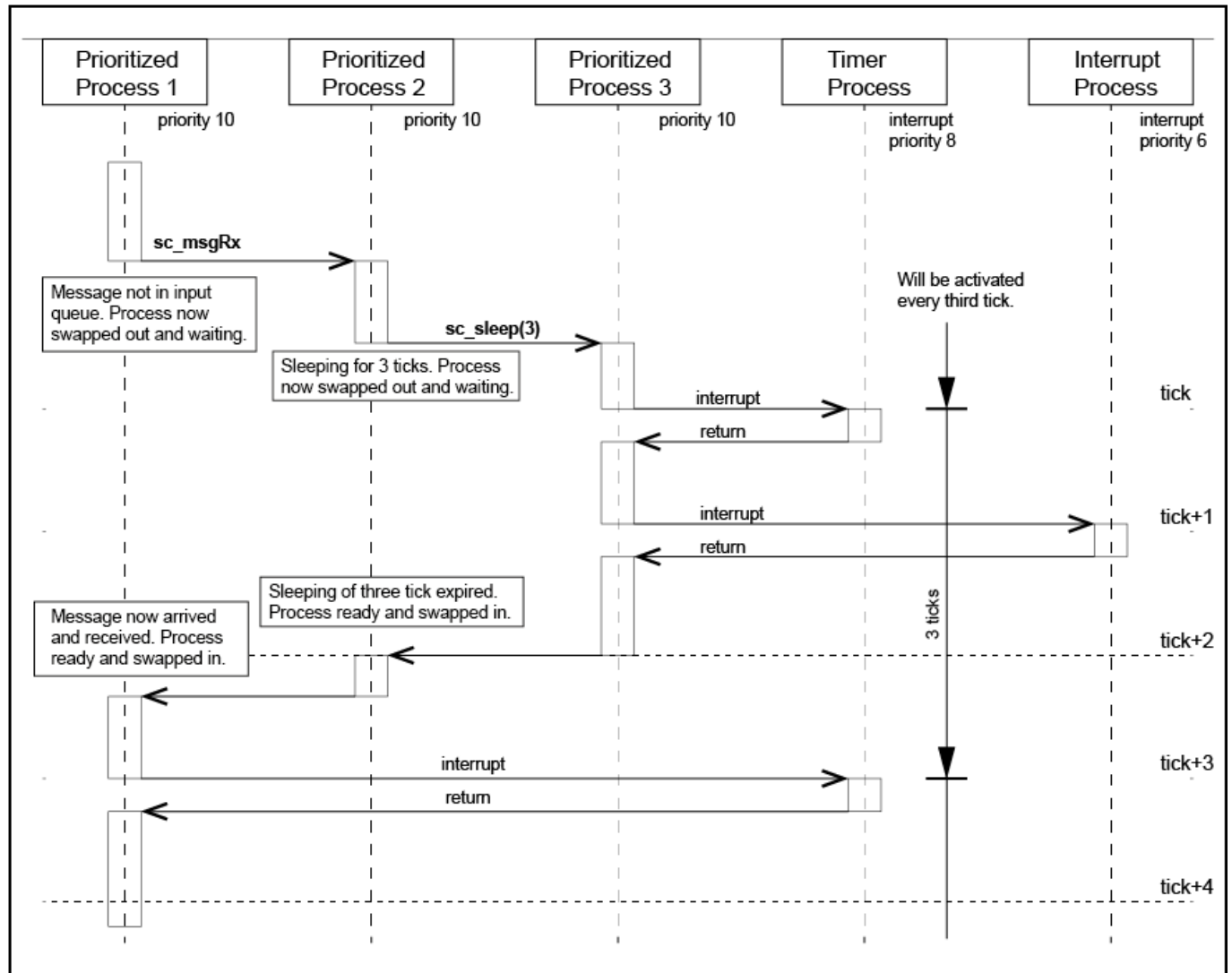
Figure 12. SCIOPTA Scheduling Sequence Example

5.12 Message Sent to Unknown Process

If the kernel receives a message which was sent to an undefined process, this message is transferred by the kernel to the default connector process. In the process where the default connector is registered, an error handling can then take place.

This means that each application should contain a default connector process.

If the message is sent with the flag **SC_MSGTX_RTN2SND**, then the message will stay with the sender.

5.12.1 Example

```
SC_PROCESS(sweepbus)
{
  sc_msg_t msg;
  (void)sc_connectorRegister(1);   /* Register as default connector */
  for (;;){
    msg = sc_msgRx(SC_ENDLESS_TMO, SC_MSGRX_ALL, MSGRX_MSGID);

    /* Error handling */

    sc_miscError(SC_ERR_SYSTEM_FATAL | 0x10000, (sc_extra_t)msg);

  }
}
```

As soon a message does not have a correct receiver, the kernel is forwarding this message to process sweepbus. In this process the error_hook is called.

6 Pools

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 free it.

There can be up to 127 pools per module for a standard kernel. Please consult chapter "Modules" 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 where it was created.

The size of a pool will be defined when the pool is created. By killing a module, all pools 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.

6.1 Message Pool Size

The minimum message pool size is the size of the maximum number of messages which ever are allocated at the same time plus the pool control block (pool_cb).

The pool control block (pool_cb) can be calculated according to the following formula:

pool_cb = 68 + n * 20 + stat * n * 20

where:

n	Number of buffer sizes	
	value	4, 8, 16

stat	Process statistics or message statistics	
	1	used
	0	unused

6.2 Creating Pools

6.2.1 Static Pool Creation

Static pools are pools which are automatically created when the systems boots up. They are defined in the SCONF configuration tool.
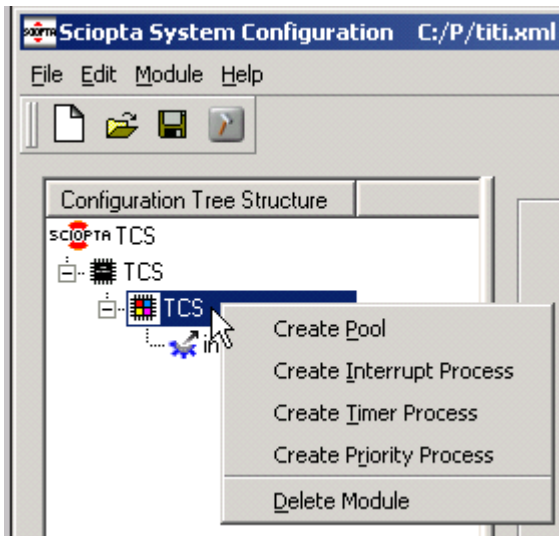
Figure 13. Pool Creation by SCONF

6.2.2 Dynamic Pool Creation

Another way is to create modules dynamically by the `sc_poolCreate` system call.

Dynamic Pool Creation

```
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"
);
```

7 Hooks

7.1 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.

Hooks need to be declared in the **SCIOPTA** kernel configuration SCONF. Please consult the **SCIOPTA** System Manuals for more information.

Additionally you also need to declare hooks by using specific system calls.

7.2 Error Hook

The Error Hook is the most important user hook function and should 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.

The error hook is described in "Error Handling".

7.3 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.

Message hooks must be registered by using the `sc_msgHookRegister` system call.

7.4 Process Hooks

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

SCIOPTA allows to configure a Process Swap Hook. It 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.

> **Kernel V1:**
>
> Select MMU hook checkbox in the SCIOPTA kernel configuration SCONF to enable MMU/MPU support.

> **Kernel V2:**
>
> If the IRQ Swap hook is activated, then the swap hook is also called before an interrupt process is entered.
>
> Process hooks must be registered by using the `sc_procHookRegister` system call.

7.5 Pool Hooks

Each time a pool is created or killed, the kernel is calling the **Pool Create Hooks** and **Pool Kill Hooks** if thes hooks have been registerred by the `sc_poolHookRegister` system call.

SCIOPTA

8 System Start and Setup

8.1 Start Sequence

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

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

1. The kernel calls the function reset_hook.

2. The kernel performs some internal initialization.

3. The kernel calls cstartup to initialize the C system.

4. The kernel calls the function start_hook.

5. The kernel calls the function TargetSetup. The code of this function is automatically generated by the SCONF configuration utility and included in the file sconf.c. TargetSetup creates the system module.

6. The kernel calls the dispatcher.

7. The first process (init process of the system module) is swapped in.

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

8. The init process of the system module creates all static modules, processes and pools.

9. The init process of the system module calls the system module start function. The name of the function corresponds to the name of the system module.

10. The process priority of the init process of the system module is set to 32 and loops for ever.

11. The init process of each created static module calls the user module start function of each module. The name of the function corresponds to the name of the respective module.

12. The process priority of the init process of each created static module is set to 32 and loops for ever.

13. The process with the highest system priority will be swapped-in and executed.

8.2 Reset Hook

In **SCIOPTA** a reset hook must always be present and must have the name reset_hook.

The reset hook must be written by the user.

After system reset the **SCIOPTA** kernel initializes a small stack and jumps directly into the reset hook.

The reset hook is mainly used to do some basic chip and board settings. The C environment is not yet initialized when the reset hook executes. Therefore the reset hook should be written in assembler. For some C environments it might be written in C.

There is no reset hook in the **SCIOPTA** Simulator.

8.2.1 Syntax

```
int reset_hook (void);
```

8.2.2 Parameter

None.

8.2.3 Return Value

!=0	The kernel will immediately call the dispatcher. This will initiate a warm start.
==0	The kernel will jump to the C startup function. This will initiate a cold start.

8.2.4 Location

Reset hooks are compiler manufacturer and board specific. Reset hook examples can be found in the **SCIOPTA** Board Support Package deliveries.

resethook.S	Very early hardware initialization code written in assembler.
	The extension .S is used in GCC for assembler source files. For other compiler packages the extensions for assembler source files might be different.
	File location: <installation_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\src

8.3 C Startup

After a cold start the kernel will call the C startup function. It initializes the C system and replaces the library C startup function. C startup functions are compiler specific.

There is no C startup function needed in the **SCIOPTA Simulator**.

8.4 Starting the SCIOPTA Simulator

Only for the SCIOPTA SCSIM Simulator:

You need to write the WinMain method and include the "sciopta_start" system call to implement a **SCIOPTA** WIN32 simulator application.

In the delivered **SCIOPTA** examples the WinMain method and the whole startup code is usually included in the file system.c.

system.c	SCIOPTA SCSIM Simulator setup including the WinMain method.
	File location: <installation_folder>\sciopta\<version>\exp\krn\win32\hello\

8.4.1 Module Data RAM

In **SCIOPTA** system running in a real target CPU the module RAM memory map is defined in the linker scripts.

In the **SCIOPTA** SCSIM Simulator you need to declare the module RAM by a character array of the size of the module.

8.5 Start Hook

The start hook must always be present and must have the name start_hook. The start hook must be written by the user. If a start hook is declared the kernel will jump into it after the C environment is initialized.

The start hook is mainly used to do chip, board and system initialization. As the C environment is initialized it can

be written in C. The start hook would also be the right place to include the registration of the system error hook (see "Error Hook Registering") and other kernel hooks.

8.5.1 Syntax

```
void start_hook (void);
```

8.5.2 Parameter

None.

8.5.3 Return Value

None.

8.5.4 Location

In the delivered **SCIOPTA** examples the start hook is usually included in the file system.c

system.c	System configuration file including hooks (e.g. start_hook) and other setup code.
	File location: <installation_folder>\sciopta\<version>\exp\<product>\<arch>\<example>\<board>\

8.6 Init Process

The init process is the first process in a module. Each module has at least one process and this is the init process. At module start the init process gets automatically the highest priority (0). After the init process has done some important work it will change its priority to the lowest level (32) and enter an endless loop.

Priority 32 is only allowed for the init process. All other processes are using priority 0 - 31. The INIT process acts therefore also as idle process which will run when all other processes of a module are in the waiting state.

The init process of the system module will first be swapped-in followed by the init processes of all other modules.

The code of the module init Processes are automatically generated by the SCONF configuration utility and placed in the file sconf.c. The module init Processes will automatically be named to <module_name>_init and created.

8.7 Module Start Functions

Please consult "Modules" for general information about **SCIOPTA** modules.

8.7.1 System Module Start Function

After all static modules, pools and processes have been created by the init Process of the system module the kernel will call a system module start function. This is function with the same name as the system module and must be written by the user. Blocking system calls are not allowed in the system module start function. All other system calls may be used.

In the delivered **SCIOPTA** examples the system module start function is usually included in the file system.c:

system.c	System configuration file including hooks (e.g. start_hook) and other setup code.
	File location: <installation_folder>\sciopta\<version>\exp\<product>\<arch>\<example>\<board>\

8.8 User Module Start Function

All other user modules have also own individual module start functions. These are functions with the same name of the respective defined and configured modules which will be called by the init process of each respective module.

After returning from the module start functions the init processes of these modules will change its priority to 32 and go into sleep. These user module start functions can use all **SCIOPTA** system calls.

The user module start function does not have to be left. It does not have to become an idle process (set priority to 32).

SCIOPTA

9 SCIOPTA Trigger

9.1 Description

The trigger in **SCIOPTA** is a method which allows to synchronize processes even faster as with messages. With a trigger, a process will be notified and woken-up by another process. Triggers are used only for process coordination and synchronization and cannot carry data. Triggers should only be used if the designer has severe timing problems and are intended for these rare cases where message passing would be too slow.

Each process has one trigger available. A trigger is an integer variable owned by the process. At process creation the value of the trigger is initialized to one.
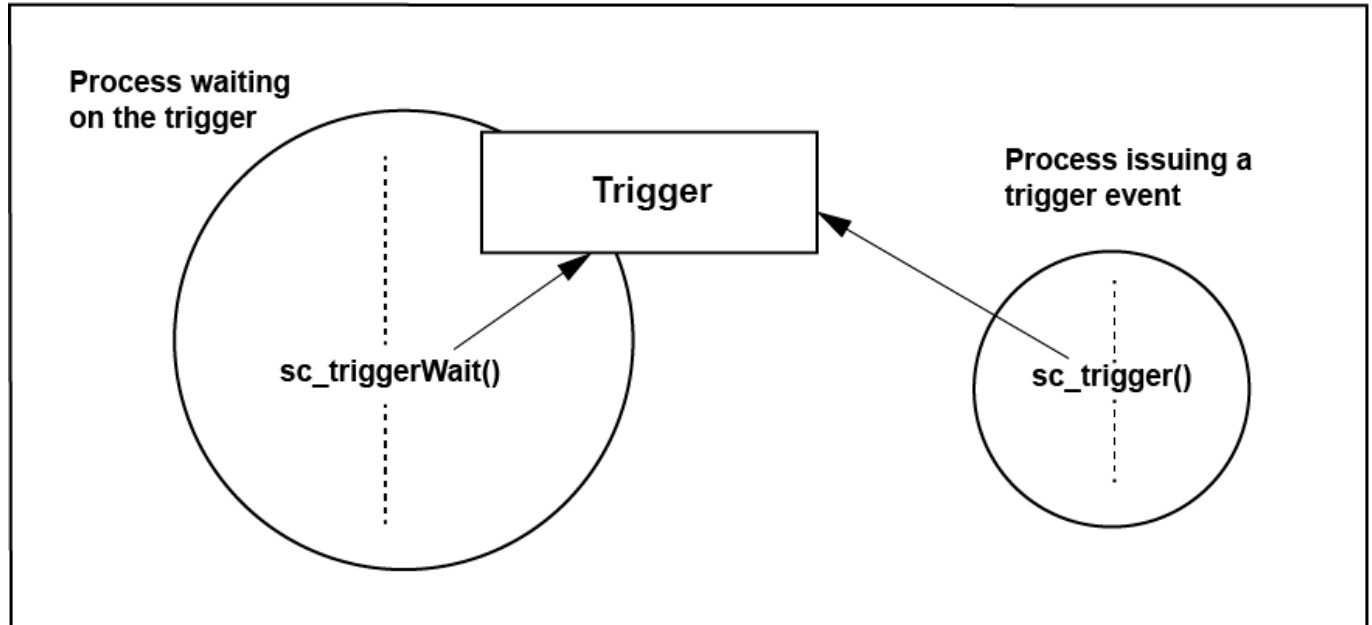
Figure 14. SCIOPTA Trigger

9.2 Using SCIOPTA Trigger

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 to zero. Only the owner process of the trigger can wait for it. The process gets ready again when either the optional timeout expires or the trigger variable become greater than zero by other processes calling sc_trigger. In case of an timeout, the trigger is incremented by the same amount as it has been decremented before.

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

The sc_triggerValueSet system call allows to sets the value of a trigger. Only the owner of the trigger can set the value. Processes can also read their own or other's value of the trigger variable by the sc_triggerValueGet 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.

The trigger variable is bound to an upper limit of 0x7fffffff.

Trigger Example

```
/* This is the interrupt process activating the trigger of process trigproc */
```

```
extern sc_pid_t trigproc_pid

SC_INT_PROCESS(myint, src)
{
  if (src == SC_PROC_WAKEUP_HARDWARE){
    sc_trigger(trigproc_pid);
  }
}

/* This is the prioritized process trigproc which waits on its trigger */

SC_PROCESS(trigproc)
{

  /* At process creation the value of the trigger is initialized      */
  /* to zero. If this is not the case you have to initialize it with  */
  /* the sc_triggerValueSet() system call                             */

  for (;;)
  {

    sc_triggerWait(1,SC_ENDLESS_TMO); /* Process waits on the trigger */
            .
            .
        /* Trigger was activated by process myint */
            .
            .
  }
}
```

10 Time Management

10.1 Introduction

Time management is one of the most important tasks of a real-time operating system. There are many functions in **SCIOPTA** which depend on time. A process can for example wait a specific time for a message to arrive or can be suspended for a specific time or timer processes can be defined which are activated at periodic time intervals.

10.2 System Tick

Time is managed by **SCIOPTA** by a tick timer which can be selected and configured by the user.

Typical time values between two ticks range between one and 10 milliseconds.

10.3 Configuring the System Tick

The system tick is configured by the sciopta configuration utility SCONF (please consult the **SCIOPTA** Systems Manuals for your specific CPU family).

10.4 External Tick Interrupt Process

An external tick interrupt process is usually included in the board support package.

systick.<ext>	System tick interrupt process.
	File location: <installation_folder>\sciopta\<version>\bsp\<arch>\<cpu>\

10.5 Tickless System

SCIOPTA scheduling does not depend on the system tick, therefore it is possible to run a **SCIOPTA** system completely tickless if there is no need for timeouts.

10.6 Timeout Server

10.6.1 Introduction

SCIOPTA has a built-in message based time-out server. Processes can register a time-out job with the time-out server. This done by the sc_tmoAdd system call which requests a time-out message from the kernel after a defined time.

10.6.2 Using the Timeout Server

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.

A time-out is requested by the sc_tmoAdd system call.

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

The registered time-out can be cancelled by the sc_tmoRm call before the time-out has expired.

11 Error Handling

11.1 Introduction

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

1. 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.

2. Process identifiers are verified in different kernel functions.

3. Ownership of messages are checked.

4. 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 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.

11.2 Error Sequence Kernel V1

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 are normal error handling functions and must be written by the user. Depending on the type of error (fatal or nonfatal) 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 (at label SC_ERROR) and all interrupts are disabled.

Note: The use of module error hooks is deprecated

11.3 Error Sequence Kernel V2 and V2INT

For system wide fatal errors (error type: **SC_ERR_SYSTEM_FATAL**) the kernel will directly call the error hook.

For all other types of errors and warnings the error hook will be called if no error process is registered.

If there was a fatal module or process error (error types: **SC_ERR_MODULE_FATAL** or **SC_ERR_PROCESS_FATAL**) and if an error process exists, the module or process will be stopped and then the error process will be activated.

If there was a module or process warning (error types: **SC_ERR_MODULE_WARNING** or **SC_ERR_PROCESS_WARNING**) and if an error process exists it will be activated.

For double faults (e.g. a fault during error handling) the kernel jumps to the label **sc_fatal** and loops for ever with interrupts disabled.

11.4 Error Hook Kernel V1

In **SCIOPTA** all error conditions will end up in the error hook. As already stated there are two error hooks available: the Module Error Hook and the Global Error Hook.

An error hook can only use the following system calls:

`sc_miscCrc`	Calculates a 16 bit CRC over a specified memory range.
`sc_miscCrcContd`	Calculates a 16 bit CRC over an additional memory range.
`sc_miscErrnoGet`	Returns the process error number (errno) variable.
`sc_moduleIdGet`	Returns the ID of a module.
`sc_moduleInfo`	Returns a snap-shot of a module control block (mcb).
`sc_moduleNameGet`	Returns the name of a module.
`sc_poolIdGet`	Returns the ID of a message pool.
`sc_poolInfo`	Returns a snap-shot of a pool control block.
`sc_procPpidGet`	Returns the process ID of the parent (creator) of a process.
`sc_procPrioGet`	Returns the priority of a prioritized process.
`sc_procSliceGet`	Returns the time slice of a timer process.
`sc_procVarDel`	Removes a process variable from the process variable data area.
`sc_procVarGet`	Returns a process variable.
`sc_procVarSet`	Defines or modifies a process variable.
`sc_tickGet`	Returns the actual kernel tick counter value.
`sc_tickLength`	Sets the current system tick length in micro seconds.
`sc_tickMs2Tick`	Converts a time from milliseconds into system ticks.
`sc_tickTick2Ms`	Converts a time from system ticks into milliseconds.
`sc_triggerValueGet`	Returns the value of a process trigger.

11.5 Error Hook Kernel V2 and V2INT

In **SCIOPTA** all error conditions will end up in the error hook.

Only one error hook can be registered in the whole system.

The error hook can only use the following system calls:

`sc_miscCrc`	Calculates a 16 bit CRC over a specified memory range.
`sc_miscCrc32`	Calculates a 32 bit CRC over a specified memory range.
`sc_miscCrcContd`	Calculates a 16 bit CRC over an additional memory range.
`sc_miscCrcContd32`	Calculates a 32 bit CRC over an additional memory range.
`sc_miscErrnoGet`	Returns the process error number (errno) variable.

sc_moduleIdGet	Returns the ID of a module.
sc_moduleInfo	Returns a snap-shot of a module control block (mcb).
sc_moduleNameGet	Returns the name of a module.
sc_poolIdGet	Returns the ID of a message pool.
sc_poolInfo	Returns a snap-shot of a pool control block.
sc_procAttrGet	Returns process attributes.
sc_procPpidGet	Returns the process ID of the parent (creator) of a process.
sc_procPrioGet	Returns the priority of a prioritized process.
sc_procSliceGet	Returns the time slice of a timer process.
sc_procVarDel	Removes a process variable from the process variable data area.
sc_procVarGet	Returns a process variable.
sc_procVarSet	Defines or modifies a process variable.
sc_tickGet	Returns the actual kernel tick counter value.
sc_tickLength	Sets the current system tick length in micro seconds.
sc_tickMs2Tick	Converts a time from milliseconds into system ticks.
sc_tickTick2Ms	Converts a time from system ticks into milliseconds.
sc_triggerValueGet	Returns the value of a process trigger.

11.6 Error Information

When the error hook is called from the kernel, all information about the error are transferred in 32-bit error word (parameter errcode). Please consult "Kernel Error Reference" for detailed description of the **SCIOPTA** error word. There are also up to four additional 32-bit extra words (parameters extra1 … extra3) available to the user.

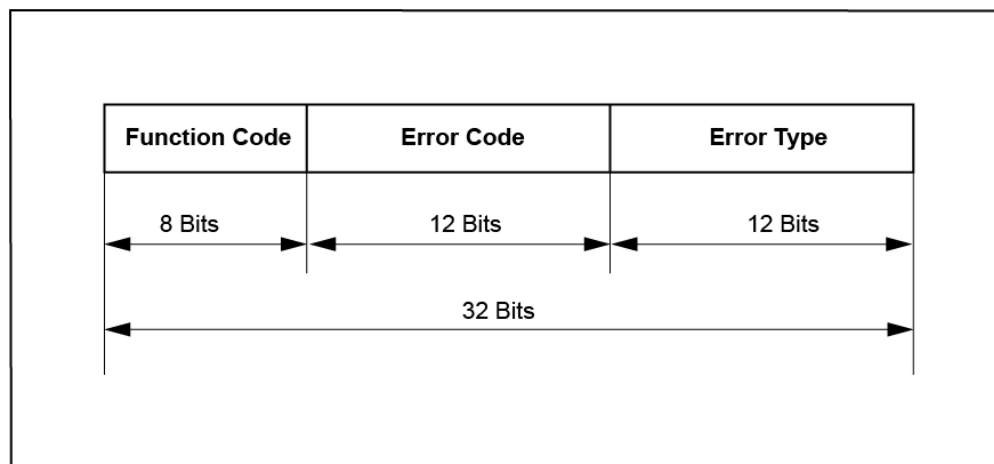| Function Code	Error Code	Error Type
8 Bits	12 Bits	12 Bits
	32 Bits	

Figure 15. 32-bit Error Word (Parameter: errcode)

The Function Code defines from which **SCIOPTA** system call the error was initiated.

The Error Code contains the specific error information.

The Error Type informs about the source and type of error.

There are three error types in a **SCIOPTA** kernel.

SC_ERR_SYSTEM_FATAL	System wide fatal error.
SC_ERR_MODULE_FATAL	Module wide fatal error.
SC_ERR_PROCESS_FATAL	Process wide fatal error.

There are three error warnings in a **SCIOPTA** kernel.

SC_ERR_SYSTEM_WARNING	System wide warning.
SC_ERR_MODULE_WARNING	Module wide warning.
SC_ERR_PROCESS_WARNING	Process wide warning.

11.7 Error Hook Registering

An error hook is registered by using the <u>sc_miscErrorHookRegister</u> system call.

> **Kernel V1:**
> If the error hook is registered from within the system module it is registered as a global error hook. In this case the error hook registering will be done in the start hook.If the error hook is registered from within a module which is not the system module it will be registered as a module error hook.
>
> **Note:** Module error hook is deprecated.

> **Kernel V2 and V2INT:**
> The error hook can only be registered in the start hook.

11.8 Error Hook Declaration Syntax Kernel V1

11.8.1 Description

For each registered error hook there must be a declared error hook function.

11.8.2 Syntax

```
int <err_hook_name> (sc_errcode_t errcode, sc_extra_t extra, int user, sc_pcb_t *pcb)
{

  ... error hook code

};
```

11.8.3 Parameter

errcode	Error word.
	Error word containing the function code which defines from which **SCIOPTA** system call the error was initiated, the error code which contains the specific error information and the error type which informs about the source and type of error.

extra	Error extra word.
	Gives additional information depending on the error code.

user	User/system error flag	
	!=0	User error
	==0	System Error

pcb	Process control block.
	Pointer to process control block of the process where the error occurred. Please consult **pcb.h** for more information about the module control block structure.

11.8.4 Error Hook Example

```c
#include "sconf.h"
#include <sciopta.h>
#include <ossys/errtxt.h>

#if SC_ERR_HOOK == 1
int error_hook(sc_errcode_t err,void *ptr,int user,sc_pcb_t *pcb)
{
  kprintf(9,"Error\n %08lx(%s,line %d in %s) %08lx %8lx %08lx %08lx\n",
    (int)pcb>1 ? pcb->pid:0,
    (int)pcb>1 ? pcb->name:"xx",
    (int)pcb>1 ?pcb->cline:0,
    (int)pcb>1 ?pcb->cfile:"xx",
    pcb,
    err,
    ptr,
    user);
  if ( user != 1 && ((err>>12)&0xfff) <= SC_MAXERR && (err>>24) <= SC_MAXFUNC )
  {
    kprintf(0,"Function: %s\nError: %s\n",
    func_txt[err>>24],
    err_txt[(err>>12)&0xfff]);
  }
  return 0;
}
#endif
```

11.9 Error Hook Declaration Syntax Kernel V2 and V2INT

11.9.1 Description

For each registered error hook there must be a declared error hook function.

11.9.2 Syntax

```c
void <err_hook_name> (sc_errcode_t err, const sc_errMsg_t *errMsg)
{
    ... error hook code

};
```

11.9.3 Parameter

err	Error word.
	Error word containing the function code which defines from which **SCIOPTA** system call the error was initiated, the error code which contains the specific error information and the error type which informs about the source and type of error.

errMsg	Pointer to the error message pointer.
	Message is filled by the kernel error handling.

11.9.4 Kernel Error Message Structure

```
typedef struct sc_errMsg_s {
  uint32_t      user;
  sc_errcode_t  error;
  sc_extra_t    extra0;
  sc_extra_t    extra1;
  sc_extra_t    extra2;
  sc_extra_t    extra3;
  sc_module_cb_t *cmcb;    // Current active module
  sc_pcb_t       *cpcb;
  sc_module_cb_t *emcb;    // Module where error occured
  sc_pcb_t       *epcb;
} sc_errMsg_t;
```

11.9.4.1 Structure Members

user	User/system error flag	
	!= 0	User error
	== 0	System Error

error	Error word	
	Error word containing the function code which defines from which **SCIOPTA** system call the error was initiated, the error code which contains the specific error information and the error type which informs about the source and type of error.	

extra	System specific extra error words	
	extra0	Please consult "Kernel Error Reference" for description.
	extra1	Please consult "Kernel Error Reference" for description.
	extra2	Please consult "Kernel Error Reference" for description.
	extra3	Please consult "Kernel Error Reference" for description.

cmcb	Module control block.
	Pointer to the current module control block. Please consult **module.h** for more information about the module control block structure.

cpcb	Process control block.
	Pointer to the current process control block. Please consult **pcb.h** for more information about the module control block structure.

emcb	Module control block.
	Pointer to module control block of the module where the error occurred. Please consult **module.h** for more information about the module control block structure.

epcb	Process control block.

	Pointer to process control block of the process where the error occurred. Please consult **pcb.h** for more information about the module control block structure.

11.9.5 Header Files

misc.h	Miscellaneous defines.
module.h	Module defines including module control block (mcb).
pcb.h	Process defines including process control block (pcb). File location: <installation_folder>\sciopta\<version>\include\kernel2\

11.9.6 Error Hook Example

```c
#include "sconf.h"
#include <sciopta.h>
#include <sciopta.msg>
#include <ossys/errtxt.h>

void error_hook(sc_errcode_t err,const sc_errMsg_t *errMsg)
{
  sc_pcb_t *pcb = errMsg->pcb;

  kprintf(9,"%s-Error\n"
       " %08lx(%s,line %d in %s)\n"
       " pcb = %08lx err = %08lx extra = %08lx,%08lx,%08lx,%08lx\n",
       errMsg->user ? "User" : "System",
       pcb ? ERR_PCB_PID:0,
       pcb ? ERR_PCB_NAME:"xx",
       pcb ? pcb->cline:0,
       pcb ? pcb->cfile:"xx",
       pcb,
       err,
       errMsg->extra0,errMsg->extra1,errMsg->extra2,errMsg->extra3);

  if ( errMsg->user != 1 && ((err>>12)&0xfff) <= SC_MAXERR && (err>>24) <= SC_MAXFUNC )
  {
    kprintf(0,"Function: %s\nError: %s\n",
       func_txt[err>>24],
       err_txt[(err>>12)&0xfff]);
  }
  kprintf(0,"<stopped>\n");
}
```

11.10 Error Hooks Return Behaviour Kernel V1

The actions of the kernel after returning from the module or global error hook depend on the error hook return values and the error types as described in the following table.

Global Error Hook		Module Error Hook		Error Type	Action
exists	return value	exists	return value	Module Error Fatal	

No	-	No	-	X	Endless Loop.
Yes	0	No	-	Yes	Endless Loop.
Yes	0	No	-	No	Endless Loop.
Yes	1	No	-	Yes	Kill module and swap out.
Yes	1	No	-	No	Return & continue.

11.11 Error Process Kernel V2 and V2INT

Contrary to the error hook the error process can use all non-blocking **SCIOPTA** system calls. The error process runs in the context of the init process.

Only one error process can be registered for the whole system.

11.11.1 Error Process Registering

The error process will be registered by calling sc_procAtExit in the init process of the system module. It behaves like an interrupt process and should be as short as possible.

It should delegate the error-recovery and error-handling to an error proxy.

11.11.2 Example of an Error Process

```
#include "sconf.h"
#include <sciopta.h>
#include <sciopta.msg>
#include <ossys/errtxt.h>

union sc_msg {
  sc_msgid_t id;
  sc_moduleKillMsg_t mKill;
  sc_procKillMsg_t   pKill;
};

void errorProcess(sc_errcode_t err,const sc_errMsg_t *errMsg)
{
#ifdef __DCC__
#pragma weak errorProxy_pid
#endif
  extern sc_pid_t errorProxy_pid;
  sc_msg_t msg;
  error_hook(err,errMsg);
  if ( err & SC_ERR_MODULE_FATAL ){
    kprintf(1,
      "Modul fatal error\n"
      "Request killing\n");
    msg = sc_msgAlloc(sizeof(sc_moduleKillMsg_t),
                      SC_MODULEKILLMSG,
                      0,
                      SC_NO_TMO);
  if ( !msg ){
    kprintf(0,"Could not get a message\n");
    return;
  }
  msg->mKill.mid = save_midGet(&errMsg->mcb->id);
  msg->mKill.flags = 0;
    } else if (  err & SC_ERR_PROCESS_FATAL ){
    kprintf(1,
    "Process fatal error\n"
    "Request killing\n");
    msg = sc_msgAlloc(sizeof(sc_procKillMsg_t),
                      SC_PROCKILLMSG,
                      0,
                      SC_NO_TMO);
    if ( !msg ){
      kprintf(1,"Could not get a message\n");
      return;
    }
      msg->pKill.pid = save_pidGet(&errMsg->pcb->pid);
      msg->pKill.flag = 0;
    } else {
      return;
  }
  if ( errorProxy_pid && errorProxy_pid != SC_ILLEGAL_PID ){
    sc_msgTx(&msg, errorProxy_pid, SC_MSGTX_RTN2SND);
  }
  if ( msg ){
    sc_msgFree(&msg);
    kprintf(0,"No errorProxy\n");
  }
}
```

11.12 The Error Proxy Kernel V2 and V2INT

The error proxy is a normal prioritized process which works on behalf of the error process. The error proxy is optional and can use all **SCIOPTA** system calls.

Communication between error process and error proxy is done by normal **SCIOPTA** messages.

For example an error proxy can kill and create modules and processes.

11.12.1 Example

```c
#include "sconf.h"
#include <sciopta.h>
#include <sciopta.msg>
#include <ossys/errtxt.h>

union sc_msg {
  sc_msgid_t id;
  sc_moduleKillMsg_t mKill;
  sc_procKillMsg_t   pKill;
};

SC_PROCESS(errorProxy)
{
  static const sc_msgid_t sel[3] = {
    SC_MODULEKILLMSG,
    SC_PROCKILLMSG,
    0
  };
  sc_msg_t msg;

  for(;;){
    msg = sc_msgRx(SC_ENDLESS_TMO, NULL, SC_MSGRX_MSGID);
    switch(msg->id){
    case SC_MODULEKILLMSG:
      sc_moduleKill(msg->mKill.mid,msg->mKill.flags);
      sc_msgFree(&msg);
      break;
    case SC_PROCKILLMSG:
      sc_procKill(msg->pKill.pid,msg->pKill.flag);
      sc_msgFree(&msg);
      break;
    default:
      sc_miscError(SC_ERR_SYSTEM_FATAL,msg->id);
    }
  }
}
```

11.13 The errno Variable

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

12 Distributed Systems

12.1 Introduction

SCIOPTA is a message based systems and therefore very well suited to support distributed multi-CPU systems. For an application programmer it does not matter if he is transmitting a message to a process on the same CPU or on a remote CPU. He will use exactly the same system calls. The **SCIOPTA** kernels and the **SCIOPTA** CONNECTOR processes have knowledge of the whole distributed environment and they take care of all details when messages need to be sent beyond CPU boundaries.

12.2 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 represents the name of the remote system.
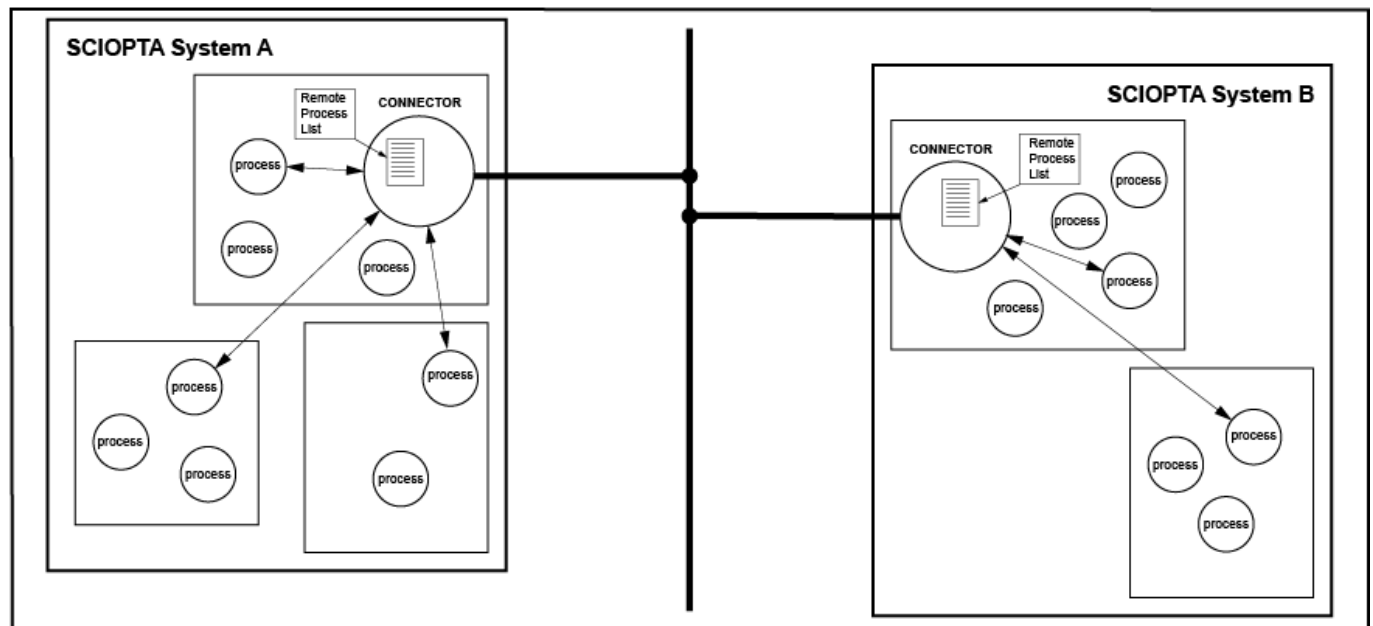There must be one connector per remote system.

Figure 16. SCIOPTA Distributed Systems

12.3 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:

//remote-system/module/procname

If the process does not reside on the same CPU as the caller, the kernel transmits a message to the CONNECTOR process including the inquiry. If the path and process is found in the remote process list, the CONNECTOR will assign a free PID for the system and send a reply 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) process using the returned 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 on the remote system can work with remote systems the same way as if these processes where local.

12.4 Unknown Process

If the kernel receives a message which was sent to an undefined process, this message is transferred by the kernel to the default connector process.

Please consult "Message Sent to Unknown Process" for a detailed description.

13 Manual Versions

13.1 Initial

- Initial version.

13.2 Typos

- Fix links

13.3 Chapter folding

- Initial chapters are folded.

- Some clarifcations.

- Layout fixes.