High Performance
Real-Time Operating Systems

Kernel

User's Guide

Copyright

Disclaimer

Trademark

Headquarters

Litronic AG
Gartenstrasse 76
CH-4052 Basel
Switzerland
Tel. +41 61 276 90 90
Fax +41 61 276 90 99
email: sales@sciopta.com
www.sciopta.com

Germany

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

France

Sciopta Systems France
3, boulevard de l'Europe
F-68100 Mulhouse
France
Tel. +33 3 89 36 33 91
Fax +33 3 89 45 57 10
email: sales@sciopta.com
www.sciopta.com

Document No. S05014RL1

SCIOPTA

Table of Contents

1 Introduction

1.1 SCIOPTA Real-Time Operating System

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
- BSP - Board support packages
- 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

1.2 About This Manual

The purpose of this **SCIOPTA - Kernel, User´s Guide** is to give all needed information how to use the **SCIOPTA** real-time kernel in an embedded project.

Please consult the **SCIOPTA - Kernel, Reference Manual** for a complete description of all system calls and error messages.

After a short introduction into real-time operating systems, detailed information about the technologies and methods used in the **SCIOPTA** kernels are given. Furthermore you will find useful information about system design and configuration.

Other target specific information can be found in the **SCIOPTA - Target Manual** which is different for each **SCIOPTA** supported processor family and includes:

- Installation Information
- Getting started examples
- Information about the system building procedures
- Description of the board support packages (BSP)
- List of distributed files
- Release notes and version history

1.3 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.

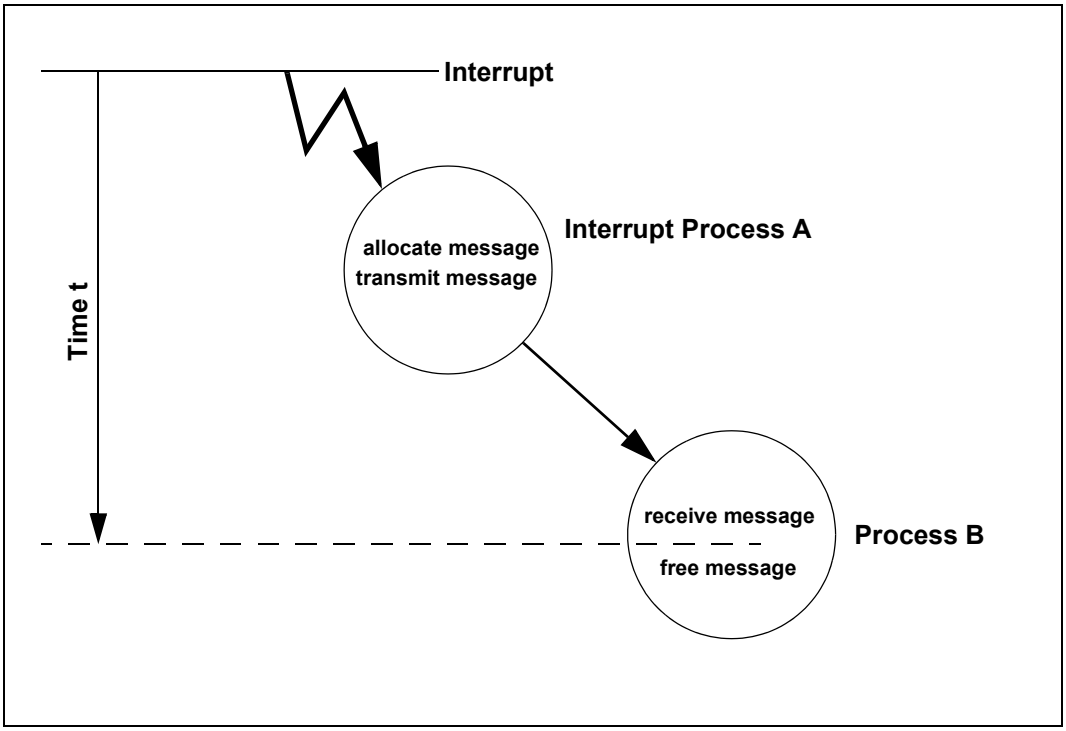


Figure 1-1: Real-Time System Definition

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.

1.3.1 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

1.3.1.1 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.

1.3.1.2 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.

1.3.1.3 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.

1.3.1.4 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.

1.3.1.5 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 such as global variables, function calls, timing interactions, priority relationships, interrupt enabling/disabling, semaphore, message passing.

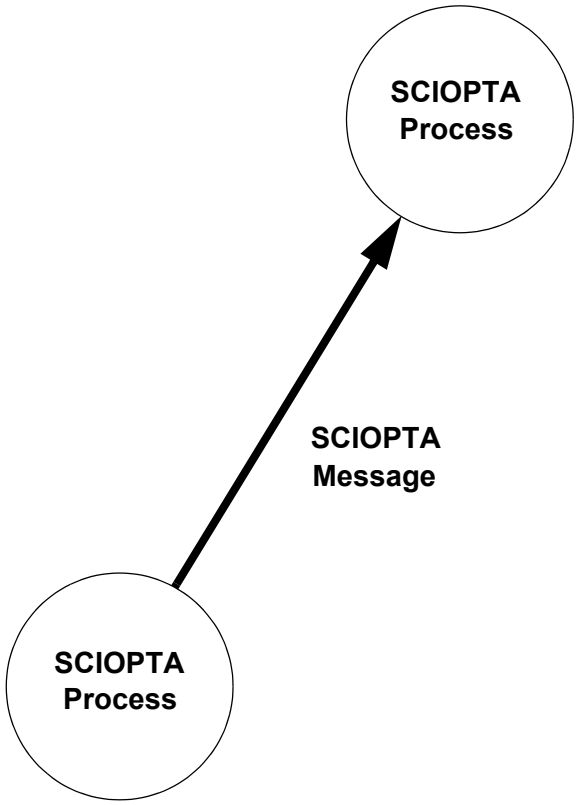
One of the duties of a real-time operating system is to manage interprocess communication and to control exchange of data between processes.

2 SCIOPTA Technology and Methods

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 2-19**). Modules and observation are features which are not needed in all systems, mainly in 8 bit and 16 bit embedded systems. This is the reason why SCIOPTA Compact is mainly available for 8/16 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.

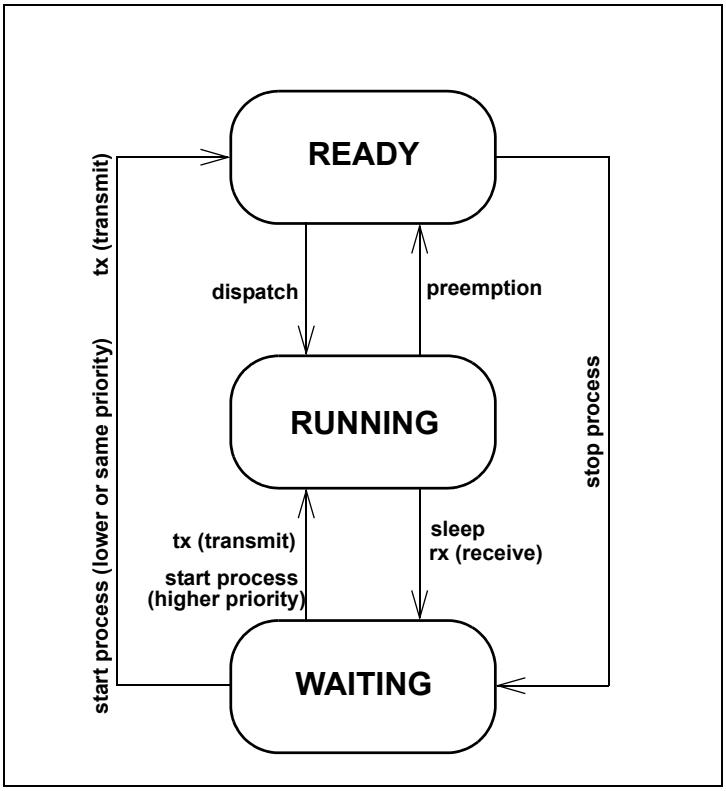


Figure 2-1: State Diagram of SCIOPTA Kernel

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.

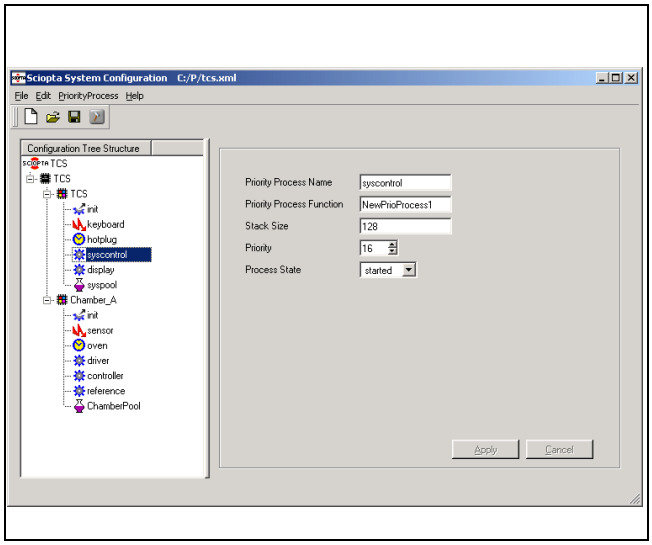


Figure 2-2: Process Configuration Window for Static Processes

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. 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.

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.

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

The **Kernel Daemon** (**sc_kerneld**) is creating and killing modules and processes.

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.

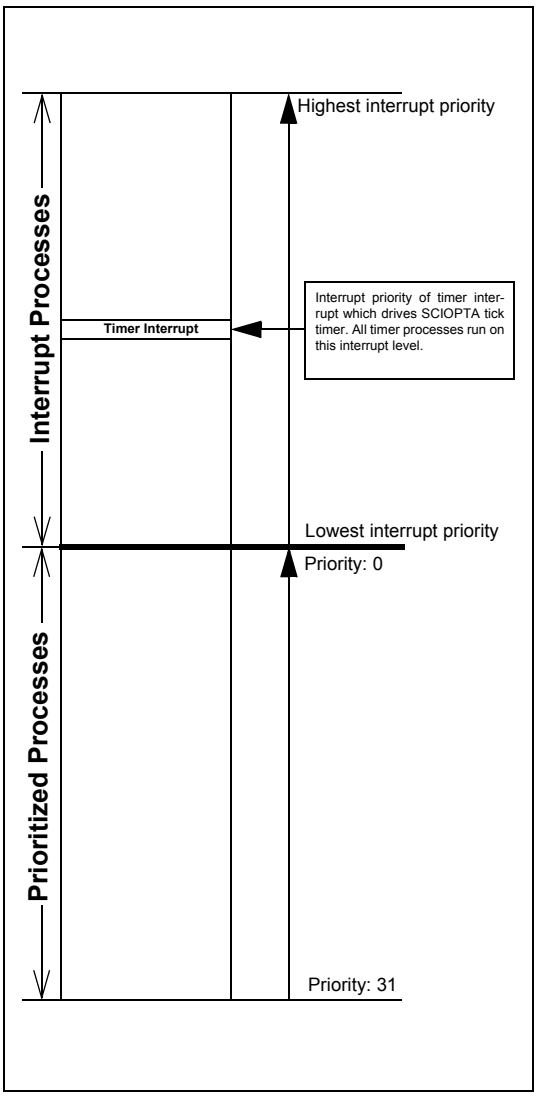


Figure 2-4: SCIOPTA Priority Diagram

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 SCIOPTA 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

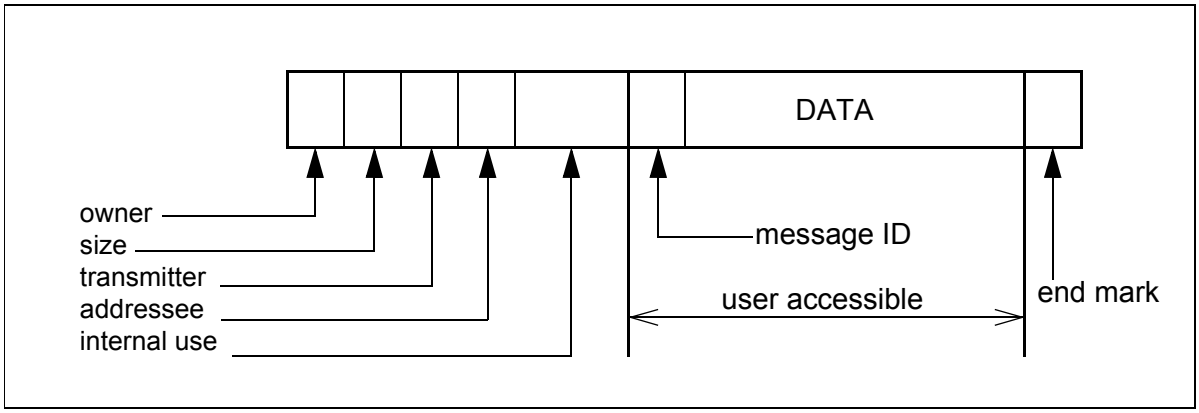


Figure 2-5: SCIOPTA Message Structure

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 for a standard kernel (32-bit) and up to 15 pools for a compact kernel (16-bit). 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 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 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.

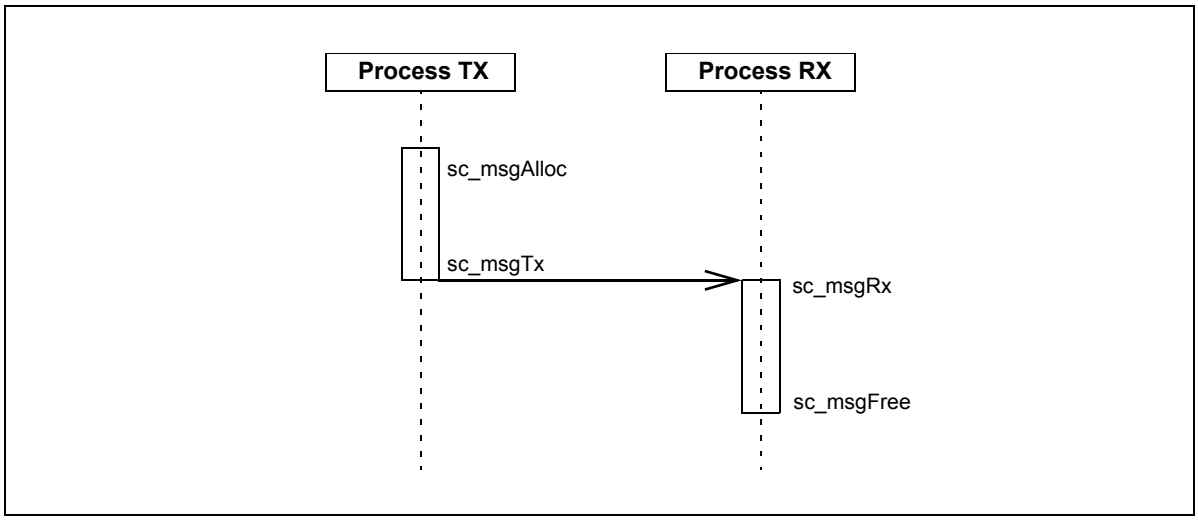


Figure 2-6: Message Sequence Chart of a SCIOPTA Message Passing

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.

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.

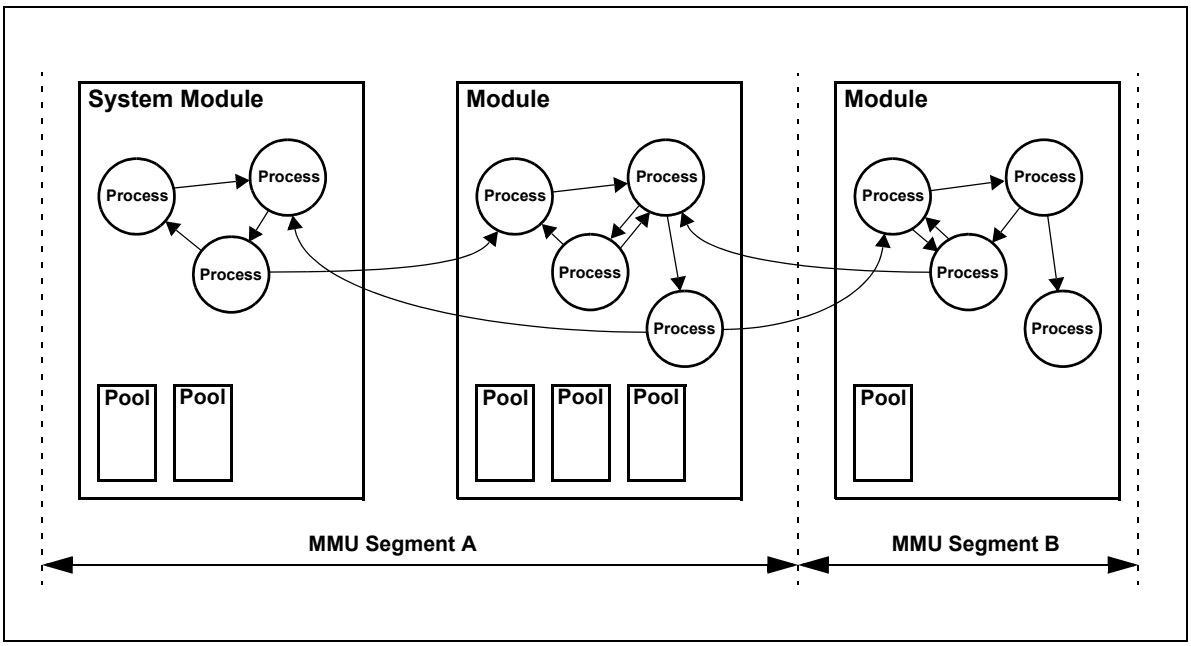


Figure 2-7: SCIOPTA Module Structure

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.

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.

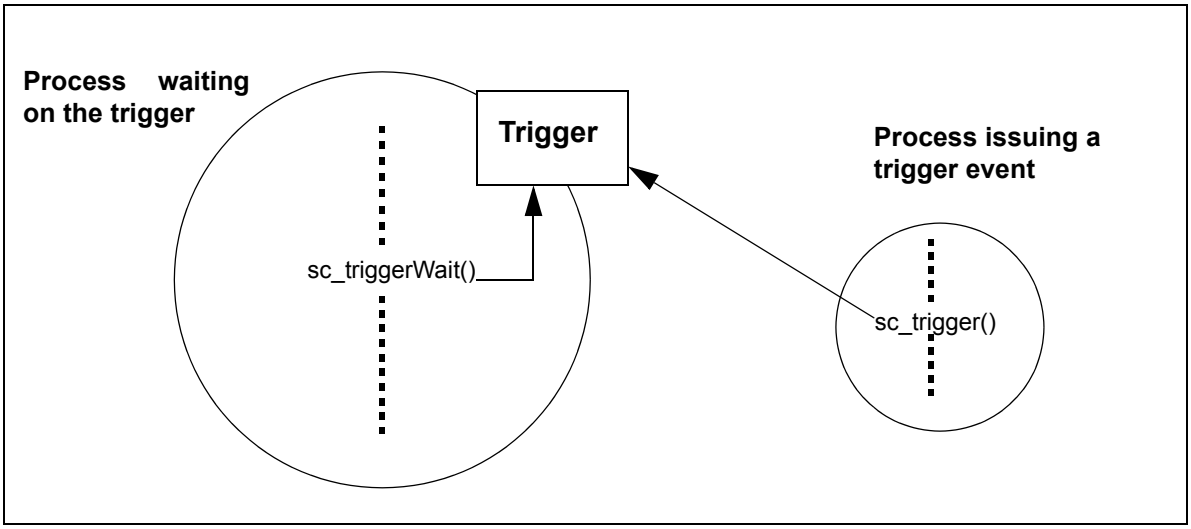


Figure 2-8: SCIOPTA Trigger

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.

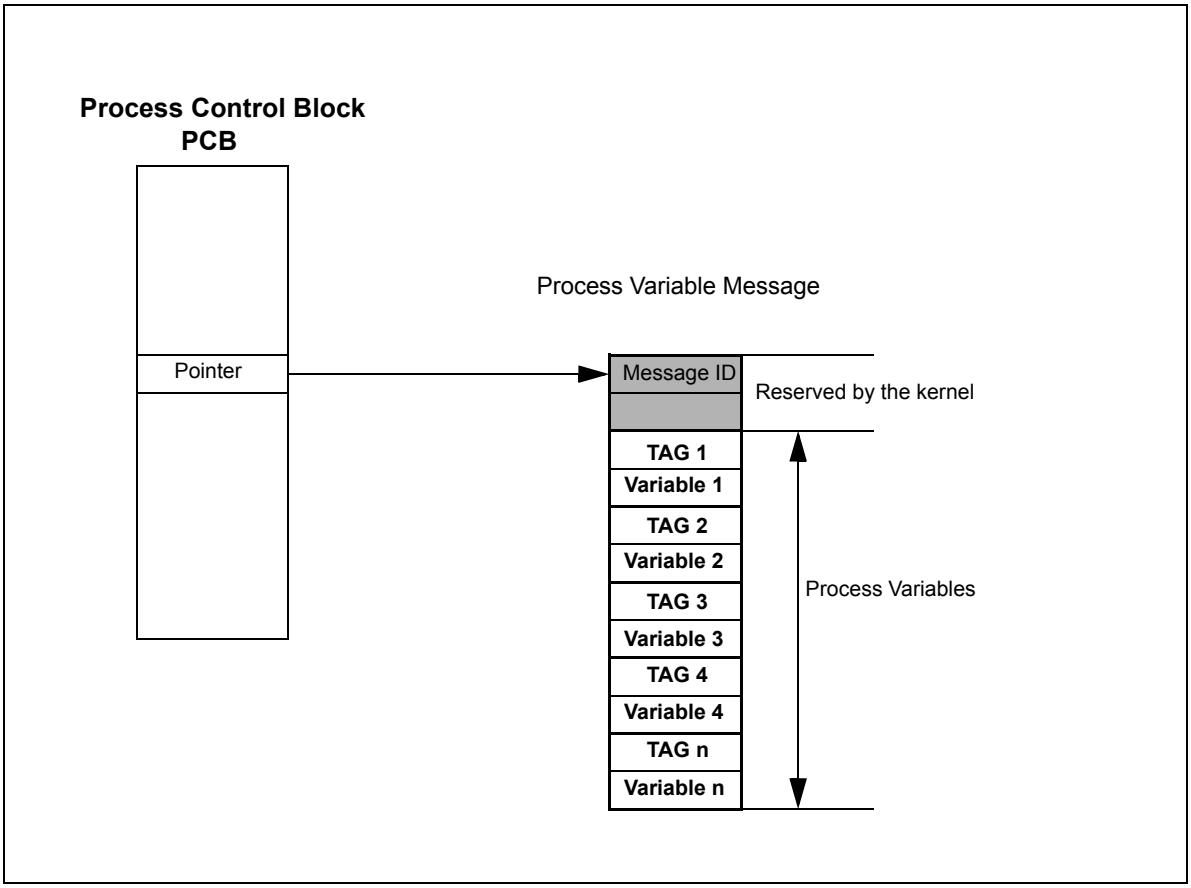


Figure 2-9: SCIOPTA Process Variables

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.

2.9 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.

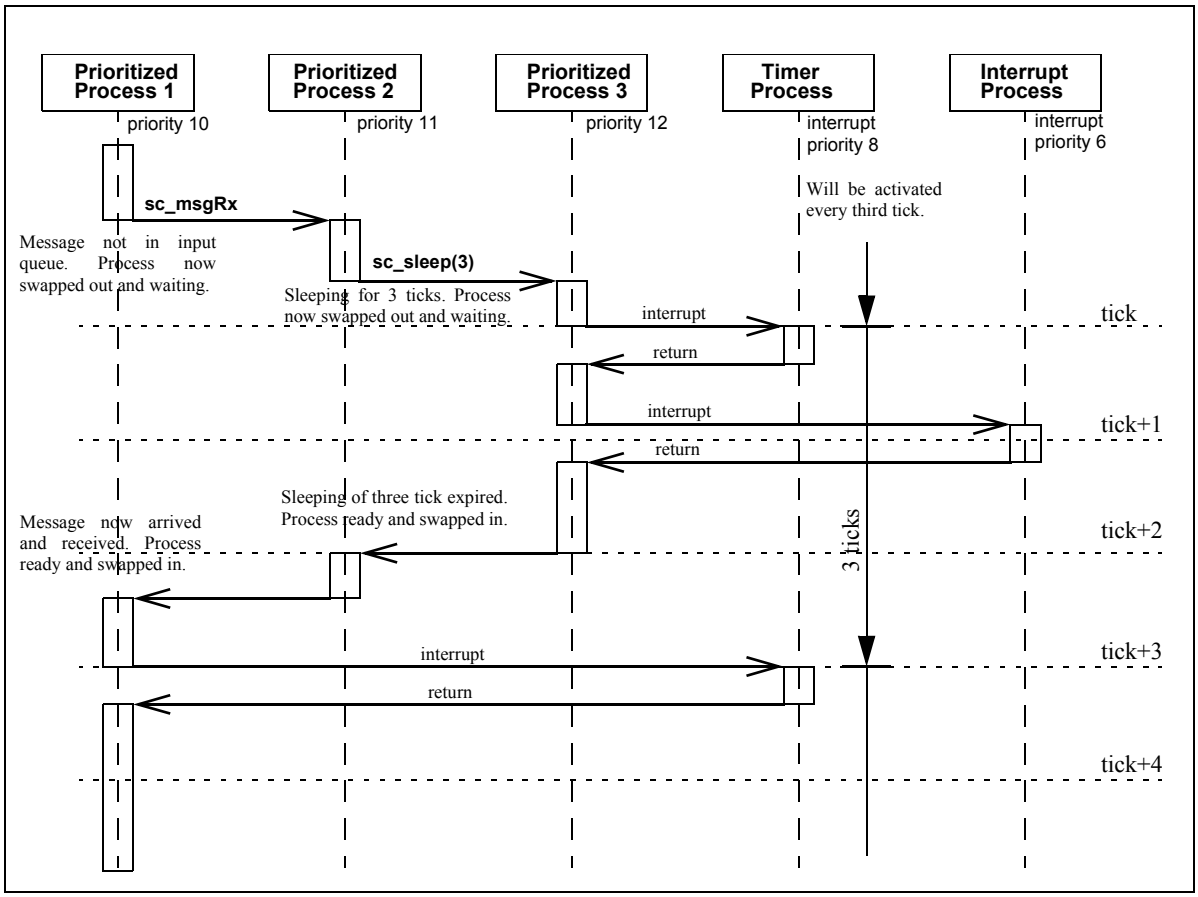


Figure 2-10: Scheduling Sequence Example

2.10 Distributed Systems

2.10.1 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.

2.10.2 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.

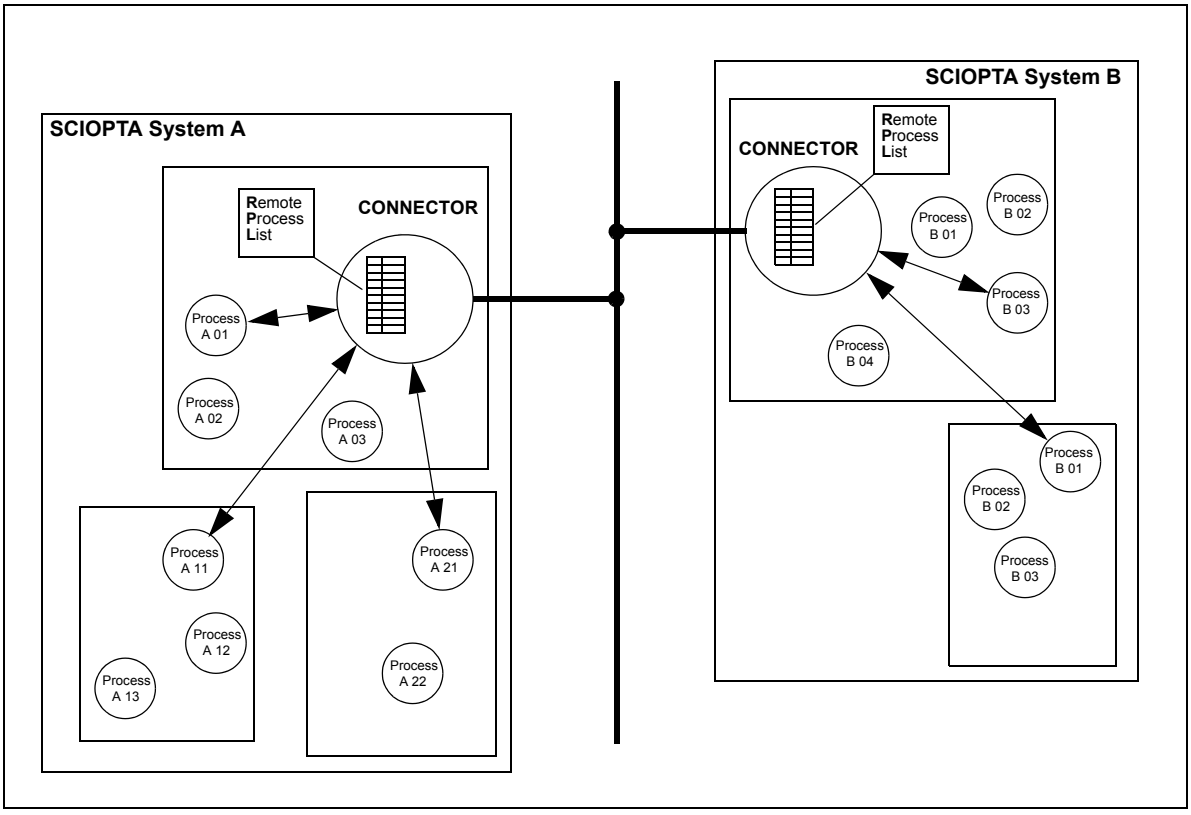


Figure 2-11: SCIOPTA Distributed System

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.

2.10.3 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 process ID for the system, add it in a remote process list and transmits a message back to the kernel including the assigned process ID. The kernel returns the process ID to the caller process.

The process can now transmit and receive messages to the (remote) process ID 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.

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**.

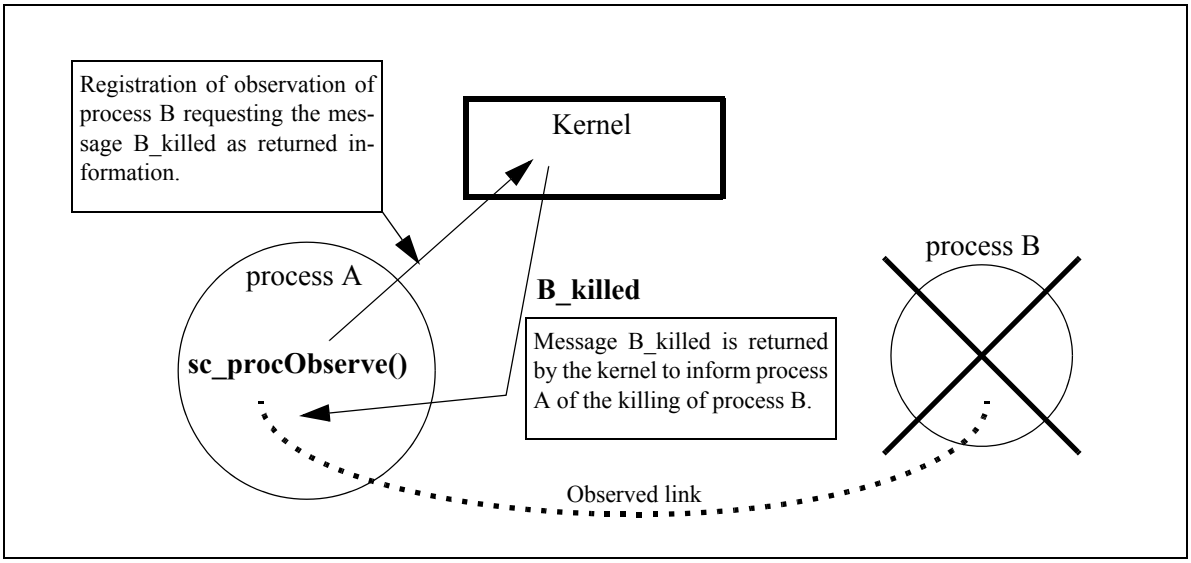


Figure 2-12: SCIOPTA Observation

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.

3 Configuration

3.1 Introduction

The kernel of a SCIOPTA system needs to be configured before you can generated the whole system. In the SCIOPTA configuration utility **SCONF** (sconf.exe) you will define the parameters for SCIOPTA systems such as name of systems, static modules, processes and pools etc.

The **SCONF** program is a graphical tool which will save all settings in an external XML file. If the setting are satisfactory for your system **SCONF** will generate three source files containing the configurable part of the kernel. These files must be included when the SCIOPTA system is generated.

A SCIOPTA project can contain different SCIOPTA Systems which can also be in different CPUs. For each SCIOPTA System defined in **SCONF** a set of source files will be generated.

3.2 Starting SCONF

The SCIOPTA configuration utility **SCONF** (config.exe) can be launched from the **SCONF** short cut of the Windows Start menu or the windows workspace. After starting the welcome screen will appear. The latest saved project will be loaded or an empty screen if the tool was launched for the first time.

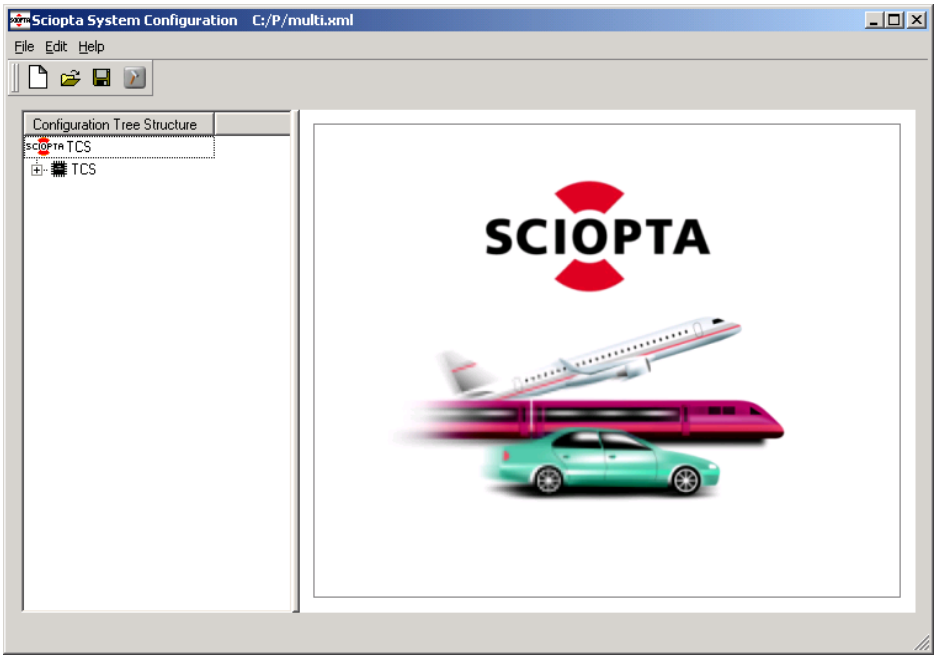


Figure 3-1: SCIOPTA Configuration Utility Start Screen

3.3 Preference File sc_config.cfg

The SCIOPTA Configuration Utility **SCONF** is storing some preference setting in the file **sc_config.cfg**.

Actually there are only three settings which are stored and maintained in this file:

1. Project name of the last saved project.
2. Location of the last saved project file.
3. Warning state (enabled/disabled).

The **sc_config.cfg** file is located in the home directory of the user. The location cannot be modified.

Every time **SCONF** is started the file **sc_config.cfg** is scanned and the latest saved project is entered.

At every project save the file **sc_config.cfg** is updated.

3.4 Project File

The project can be saved in an external XML file **<project_name>.xml**. All configuration settings of the project are stored in this file.

3.5 SCONF Windows

To configure a SCIOPTA system with **SCONF** you will work mainly in two windows.

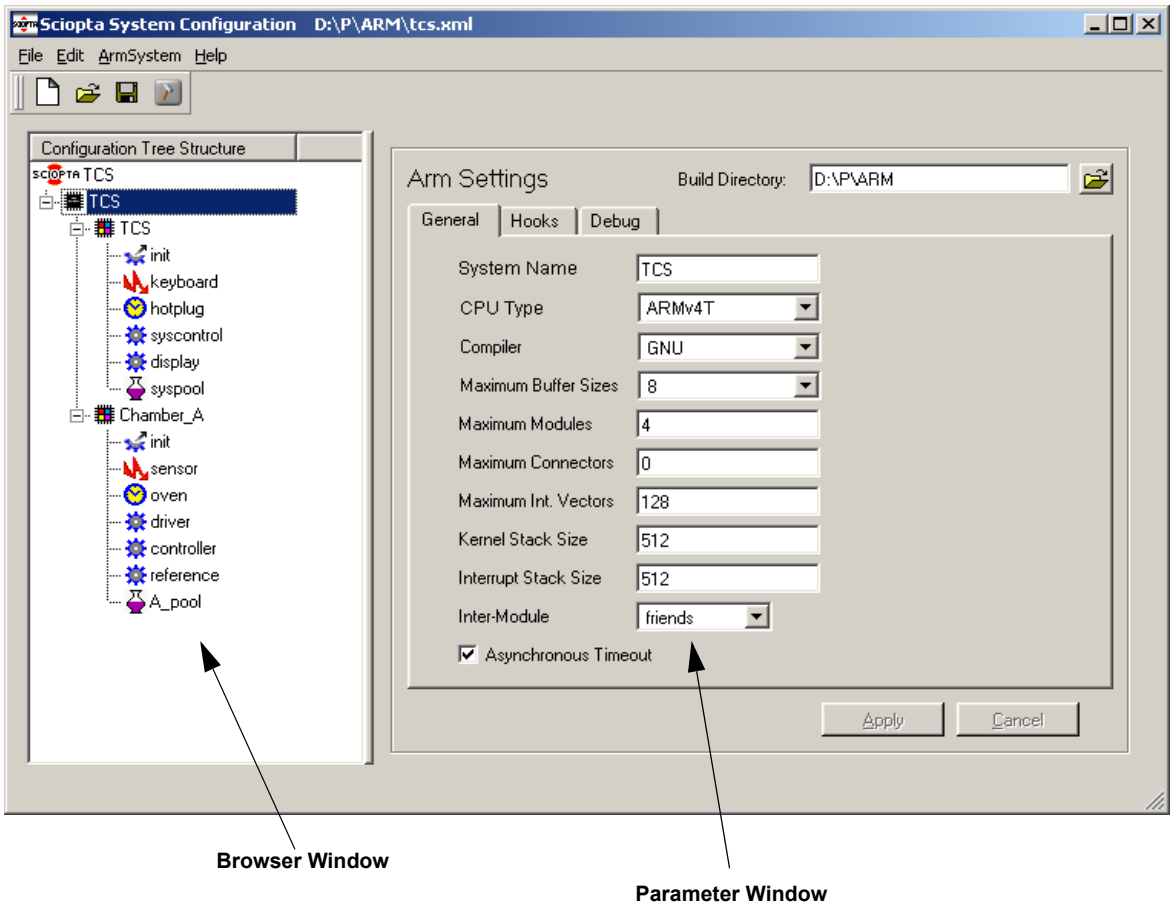


Figure 3-2: SCONF Windows

3.5.1 Parameter Window

For every level in the browser window (process level, module level, system level and project level) the layout of the parameter window change and you can enter the configuration parameter for the specific item of that level (e.g. parameters for a specific process). To open a specific parameter window just click on the item in the browser window.

3.5.2 Browser Window

The browser window allows you to browse through a whole SCIOPTA project and select specific items to configure.

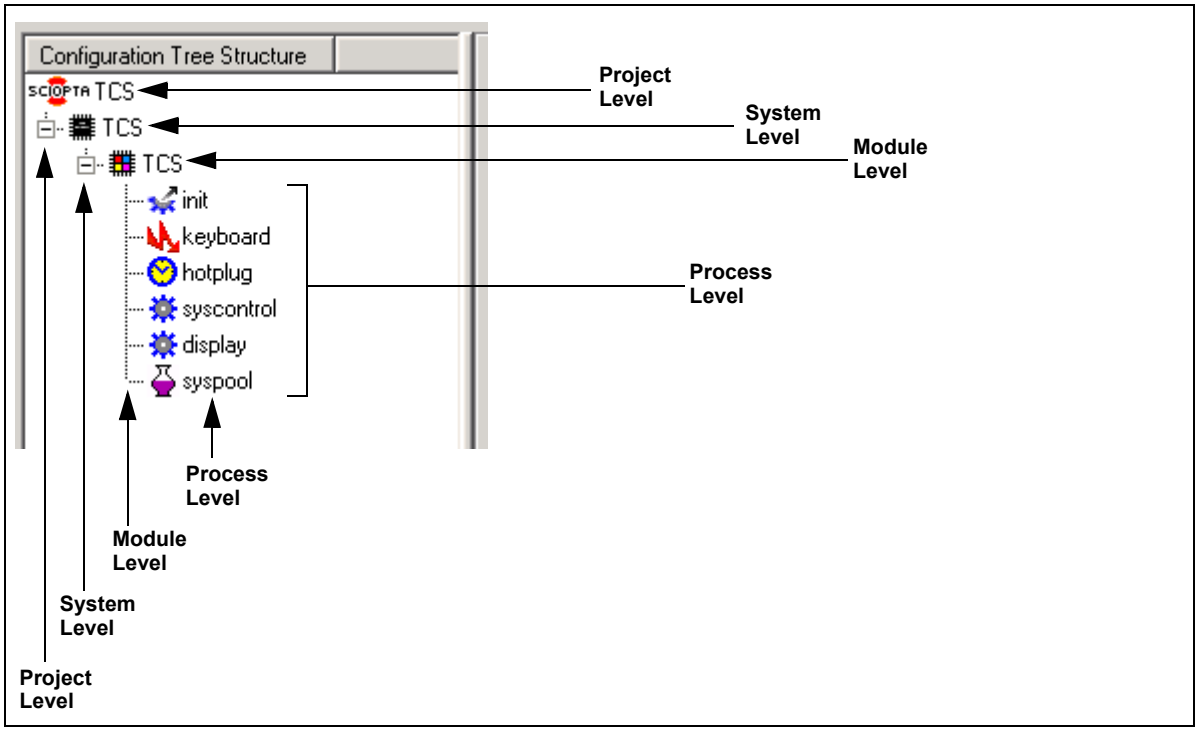


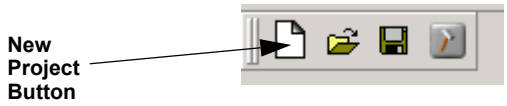
Figure 3-3: Browser Windows

The browser shows four configuration levels and every level can expand into a next lower level. To activate a level you just need to point and click on it. On the right parameter window the configuration settings for this level can be viewed and modified.

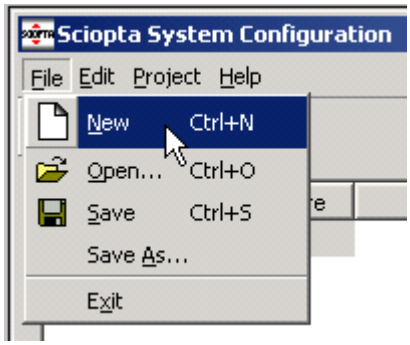
- The uppermost level is the **Project Level** where all project configurations will be done. The project name can be defined and you can create new systems for the project.
- In the **System Level** you are configuring the system for one CPU. You can create the static modules for the system and configure system specific settings.
- In SCIOPTA you can group processes into modules. On the **Module Level** you can configure the module parameters and create static processes and message pools.
- The parameters of processes and message pools can be configured in the **Process Level**.

3.6 Creating a New Project

To create a new project select the **New** button in the tool bar:

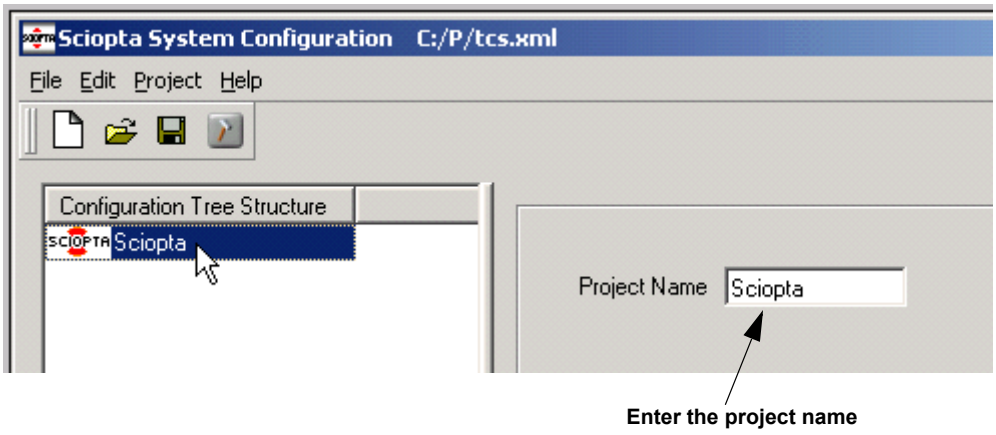


You also can create a new project from the file menu or by the Ctrl+N keystroke:



3.7 Configure the Project

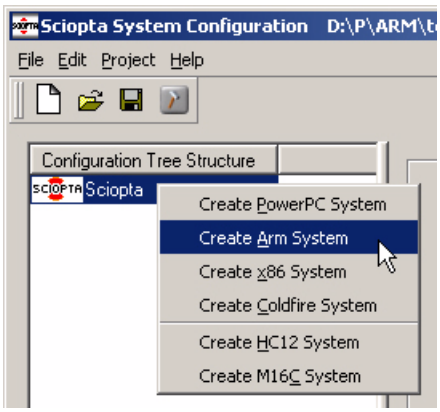
You can define and modify the project name. Click on the project name on the right side of the SCIOPTA logo and enter the project name in the parameter window.



Click on the Apply button to accept the name of the project.

3.8 Creating Systems

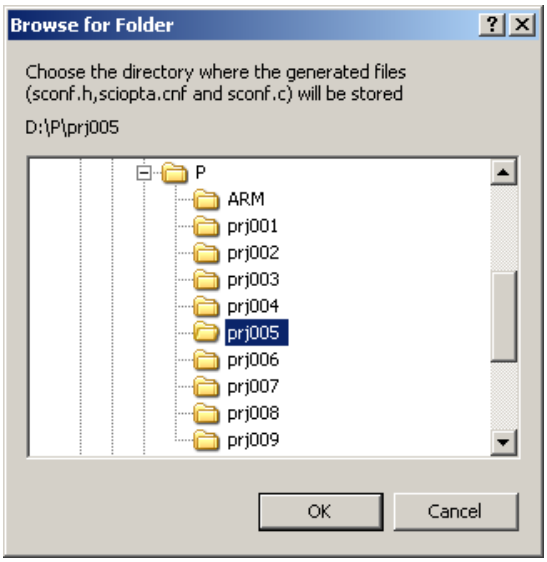
From the project level you can create new systems. Move the mouse pointer over the project and right-click the mouse.



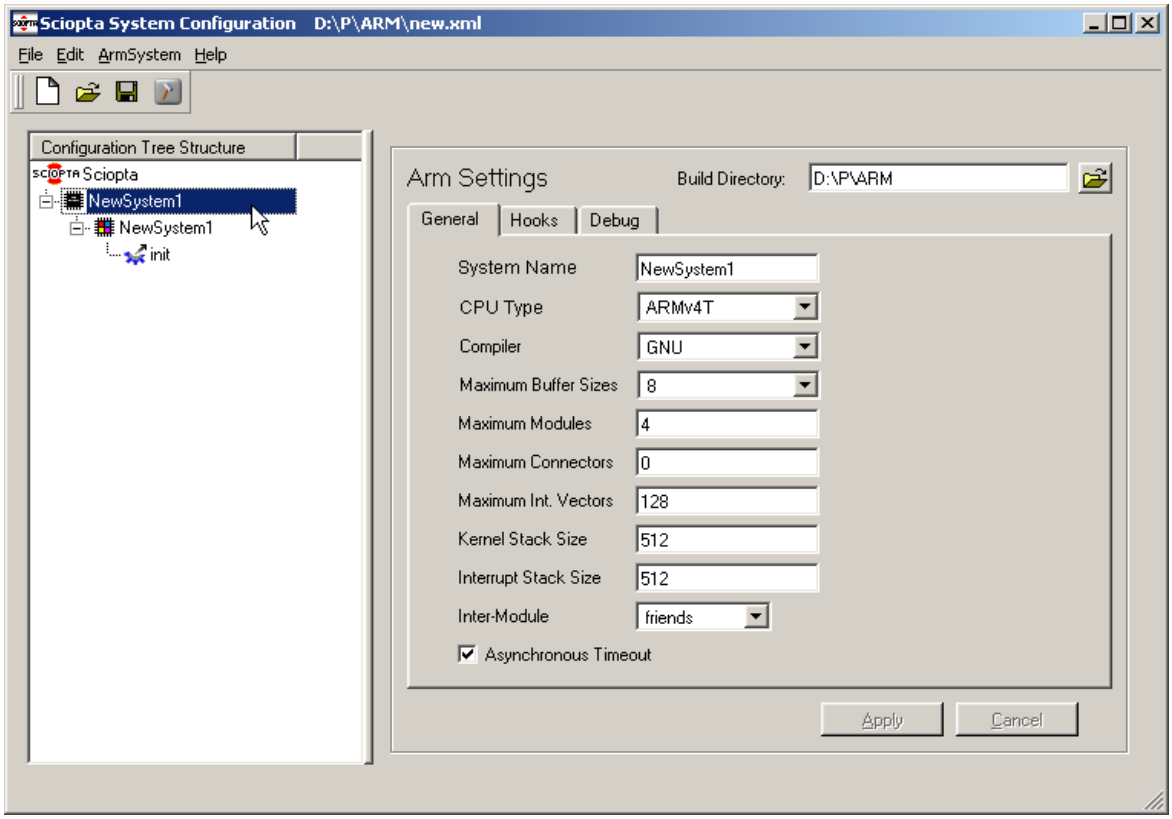
A pop-up menu appears and allows you to select a system out of all SCIOPTA supported target CPUs.

The same selection can be made by selecting the **Project** menu from the menu bar.

SCONF asks you to enter a directory where the generated files (sconf.h, sciopta.cnf and sconf.c) will be stored:



A new system for your selected CPU with the default name **New System 1**, the system module (module id 0) with the same name as the new target and a **init process** will be created.



You can create up to 128 systems inside a SCIOPTA project. The targets do not need to be of the same processor (CPU) type. You can mix any types or use the same types to configure a distributed system within the same SCIOPTA project.

You are now ready to configure the individual targets.

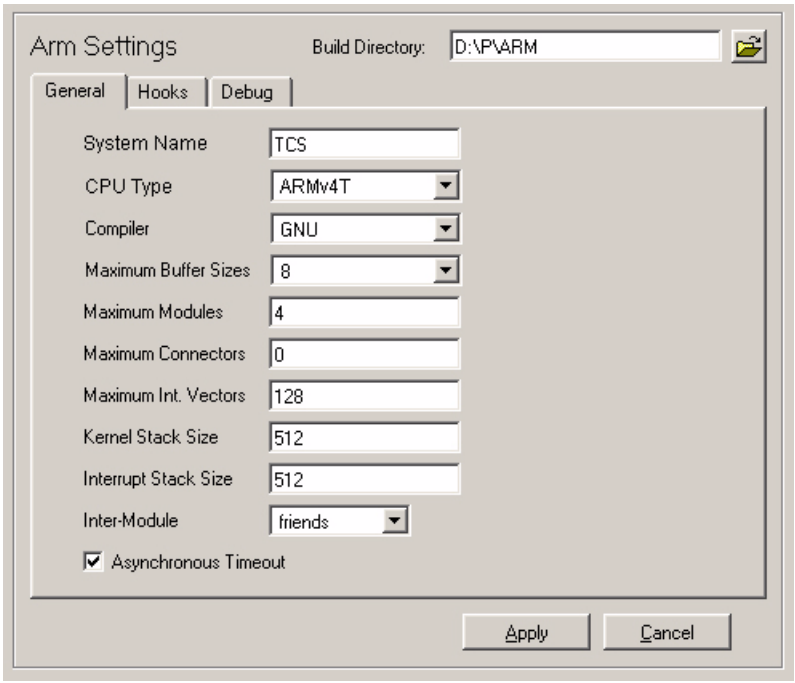
3.9 Configuring Target Systems

After selecting a system with your mouse, the corresponding parameter window on the right side will show the parameters for the selected target CPU system.

3.9.1 Configuring ARM Target Systems

The system configuration for ARM target systems is divided into 3 tabs: General, Hooks and Debug.

3.9.1.1 General Configuration



System Name

Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.

CPU Type

Give the name of you specific target processor derivative.

Compiler

Select the C/C++ Cross compiler which you are using for this SCIOPTA system.

Maximum Buffer 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 is configured here in the maximum buffer sizes entry.

Maximum Modules

Here you can define a maximum number of modules which can be created in this system. The maximum value is 127 modules. It is important that you give here a realistic value of maximum number of modules for your system as SCIOPTA is initializing some memory statically at system start for the number of modules given here.

Maximum Connectors

CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA systems. The maximum number of connectors in a system may not exceed 127 which correspond to the maximum number of systems.

Maximum Int. Vectors

Enter here the maximum number of interrupt vector in you ARM system. The default value is 128.

Kernel Stack Size

Currently not used. Entered values are not considered.

Interrupt Stack Size

Currently not used. Entered values are not considered.

Inter-Module

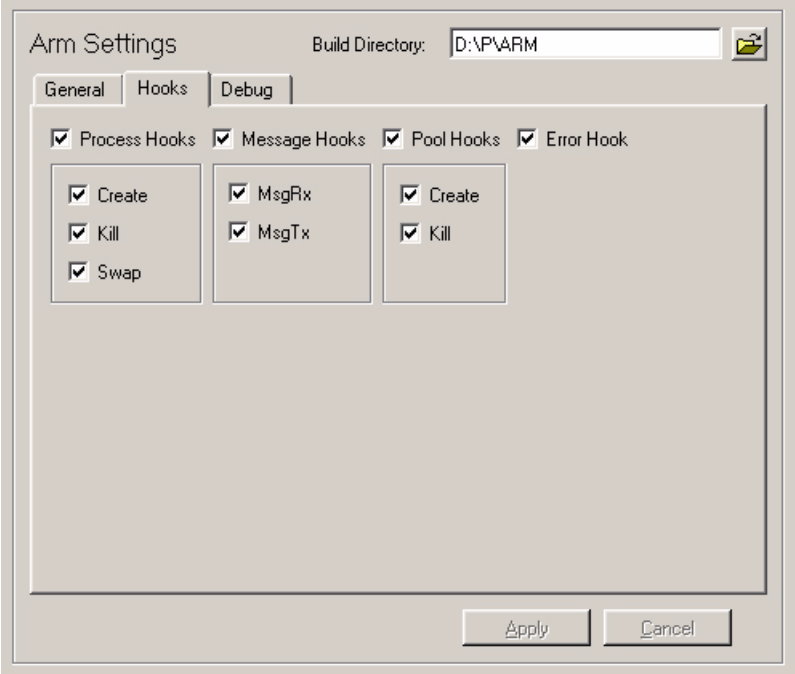
Selects the module relations and defines the message copy behaviour.

never copy	Messages between modules are never copied.
always copy	Messages between modules are always copied.
friends	The message copy behaviour is defined by the friendship setting between the modules. Please consult chapter **2.5.1 “SCIOPTA Module Friend Concept” on page 2-11** for more information.

Asynchronous Timeout

When checked the SCIOPTA ARM kernel includes timeout-server functionality.

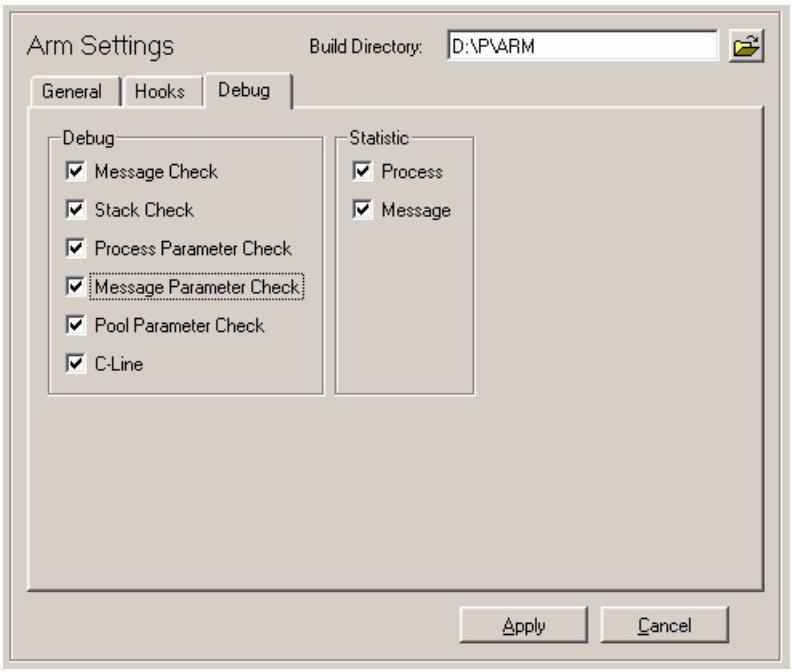
3.9.1.2 Configuring Hooks



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

You can enable all existing hooks separately by selecting the corresponding check box.

3.9.1.3 Debug Configuration



Message Check

If you are selecting this check box some test functions on messages will be included in the kernel.

Stack Check

If you are selecting this check box some test functions on process stacks will be included in the kernel.

Process Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the process system calls.

Message Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the message system calls.

Pool Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the pool system calls.

C-Line

If you are selecting this check box the kernel will include line number information which can be used by the SCIOPTA DRUID Debug System or an error hook. Line number and file of the last system call is recorded in the per process data.

Process Statistics

If you are selecting this check box the kernel will maintain a process statistics data field where information such as number of process swaps can be read.

Message Statistics

If you are selecting this check box the kernel will maintain a message statistics data field in the pool control block where information such as number of message allocation can be read.

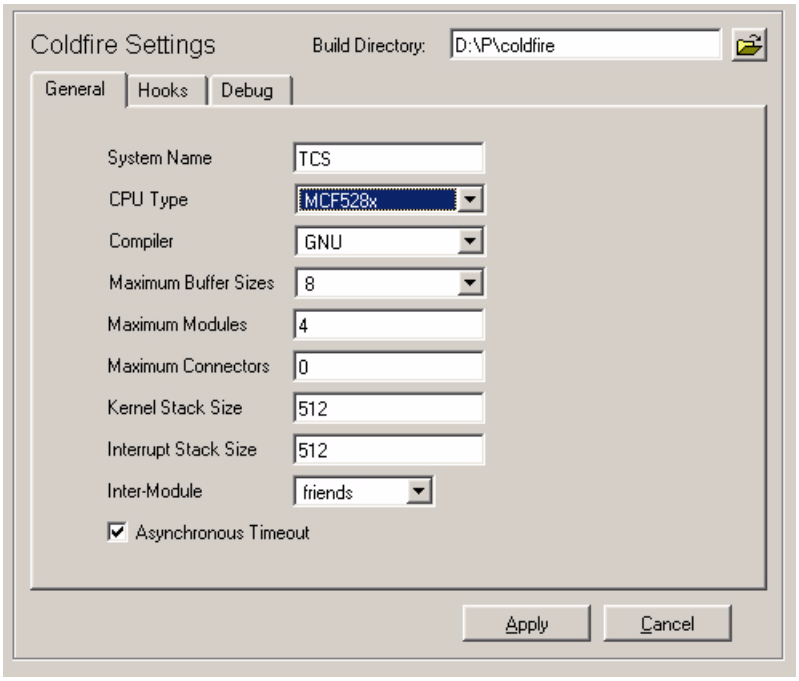
Applying Target Configuration

Click on the Apply button to accept the target configuration settings.

3.9.2 Configuring Coldfire Target Systems

The system configuration for Coldfire target systems is divided into 3 tabs: General, Hooks and Debug.

3.9.2.1 General Configuration



System Name

Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.

CPU Type

Give the name of you specific target processor derivative.

Compiler

Select the C/C++ Cross compiler which you are using for this SCIOPTA system.

Maximum Buffer 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 is configured here in the maximum buffer sizes entry.

Maximum Modules

Here you can define a maximum number of modules which can be created in this system. The maximum value is 127 modules. It is important that you give here a realistic value of maximum number of modules for your system as SCIOPTA is initializing some memory statically at system start for the number of modules given here.

Maximum Connectors

CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA systems. The maximum number of connectors in a system may not exceed 127 which correspond to the maximum number of systems.

Kernel Stack Size

Currently not used. Entered values are not considered.

Interrupt Stack Size

Currently not used. Entered values are not considered.

Inter-Module

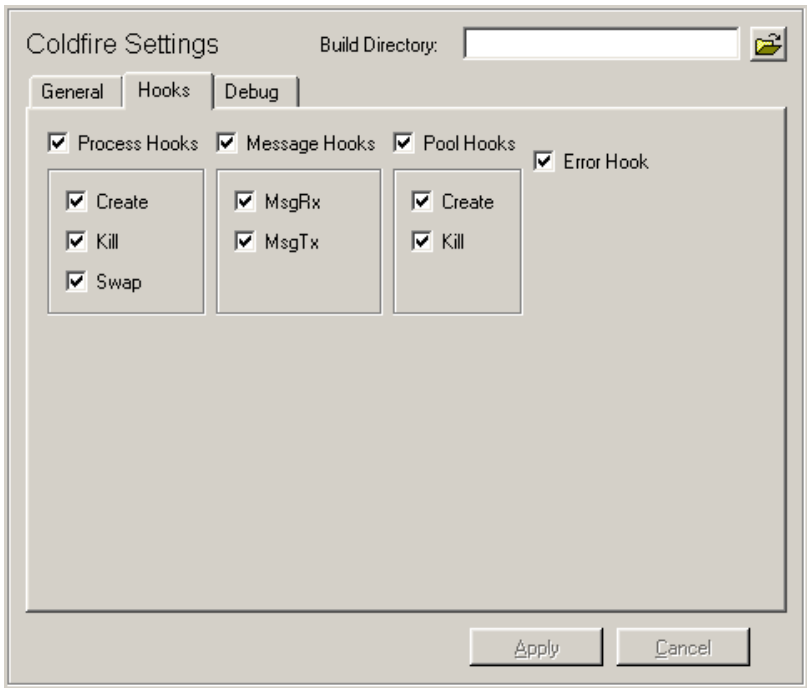
Selects the module relations and defines the message copy behaviour.

never copy	Messages between modules are never copied.
always copy	Messages between modules are always copied.
friends	The message copy behaviour is defined by the friendship setting between the modules. Please consult chapter **2.5.1 "SCIOPTA Module Friend Concept" on page 2-11** for more information.

Asynchronous Timeout

When checked the SCIOPTA Coldfire kernel includes timeout-server functionality.

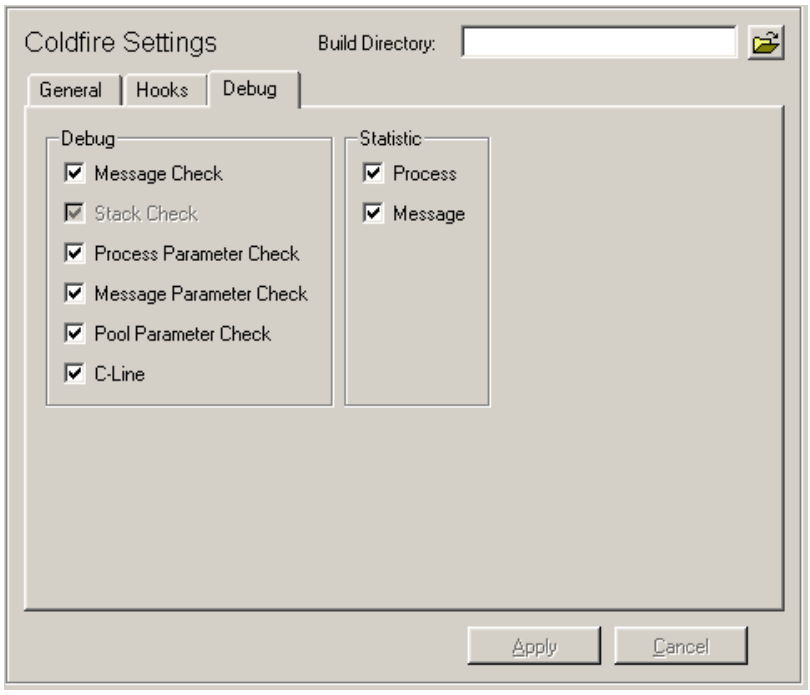
3.9.2.2 Configuring Hooks



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

You can enable all existing hooks separately by selecting the corresponding check box.

3.9.2.3 Debug Configuration



Message Check

If you are selecting this check box some test functions on messages will be included in the kernel.

Process Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the process system calls.

Message Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the message system calls.

Pool Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the pool system calls.

C-Line

If you are selecting this check box the kernel will include line number information which can be used by the SCIOPTA DRUID Debug System or an error hook. Line number and file of the last system call is recorded in the per process data.

Process Statistics

If you are selecting this check box the kernel will maintain a process statistics data field where information such as number of process swaps can be read.

Message Statistics

If you are selecting this check box the kernel will maintain a message statistics data field in the pool control block where information such as number of message allocation can be read.

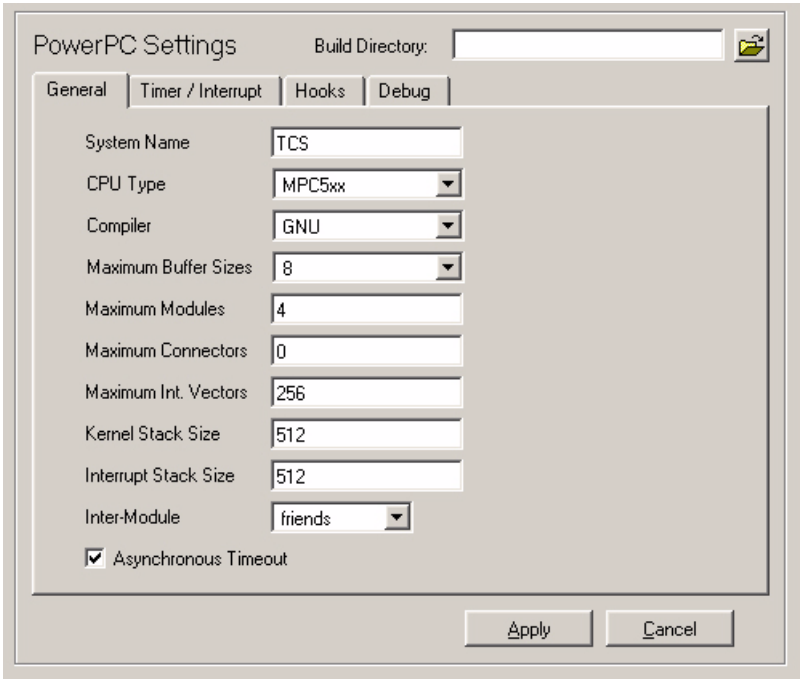
Applying Target Configuration

Click on the Apply button to accept the target configuration settings.

3.9.3 Configuring PowerPC Target Systems

The system configuration for PowerPC target systems is divided into 4 tabs: General, Timer/Interrupt, Hooks and Debug.

3.9.3.1 General Configuration



System Name

Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.

CPU Type

Give the name of you specific target processor derivative.

Compiler

Select the C/C++ Cross compiler which you are using for this SCIOPTA system.

Maximum Buffer 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 is configured here in the maximum buffer sizes entry.

Maximum Modules

Here you can define a maximum number of modules which can be created in this system. The maximum value is 127 modules. It is important that you give here a realistic value of maximum number of modules for your system as SCIOPTA is initializing some memory statically at system start for the number of modules given here.

Maximum Connectors

CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA systems. The maximum number of connectors in a system may not exceed 127 which correspond to the maximum number of systems.

Kernel Stack Size

Currently not used. Entered values are not considered.

Interrupt Stack Size

Currently not used. Entered values are not considered.

Inter-Module

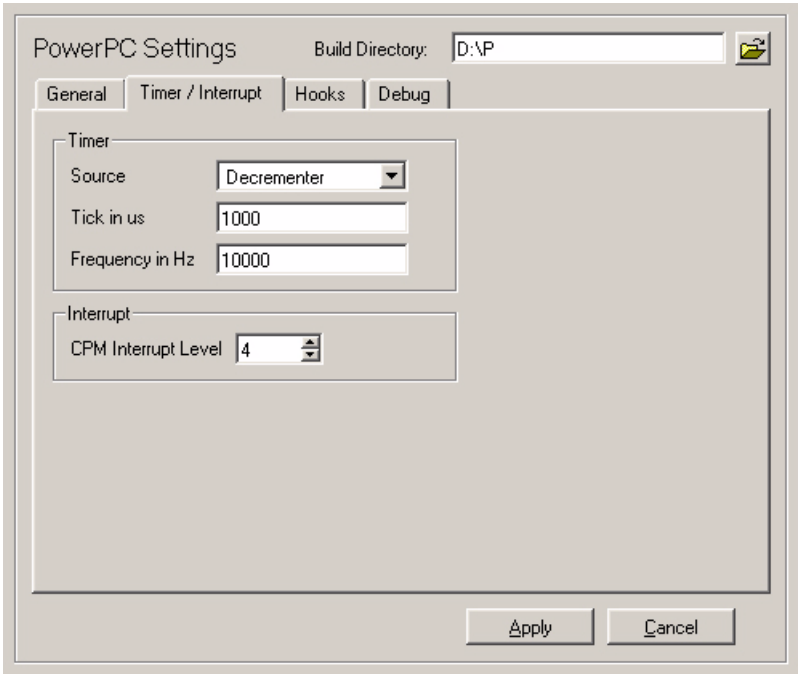
Selects the module relations and defines the message copy behaviour.

never copy	Messages between modules are never copied.
always copy	Messages between modules are always copied.
friends	The message copy behaviour is defined by the friendship setting between the modules. Please consult chapter **2.5.1 "SCIOPTA Module Friend Concept" on page 2-11** for more information.

Asynchronous Timeout

When checked the SCIOPTA PowerPC kernel includes timeout-server functionality.

3.9.4 Timer and Interrupt Configuration



Timer Source

The SCIOPTA real-time kernel uses an internal tick timer to manage and control all timing tasks. Here you can select which timer you want to use to generate the tick. If you are selecting <**none**> you need to write your own tick function (usually a user interrupt service routine) which will call **sc_tick()** explicitly.

Timer Tick

Enter here the tick interval in micro seconds. The value can only be entered if you have not selected **none** as the timer source.

Timer Frequency

You need to enter the clock frequency of the processor which is used by the timer source. The value can only be entered if you have not selected **none** as the timer source.

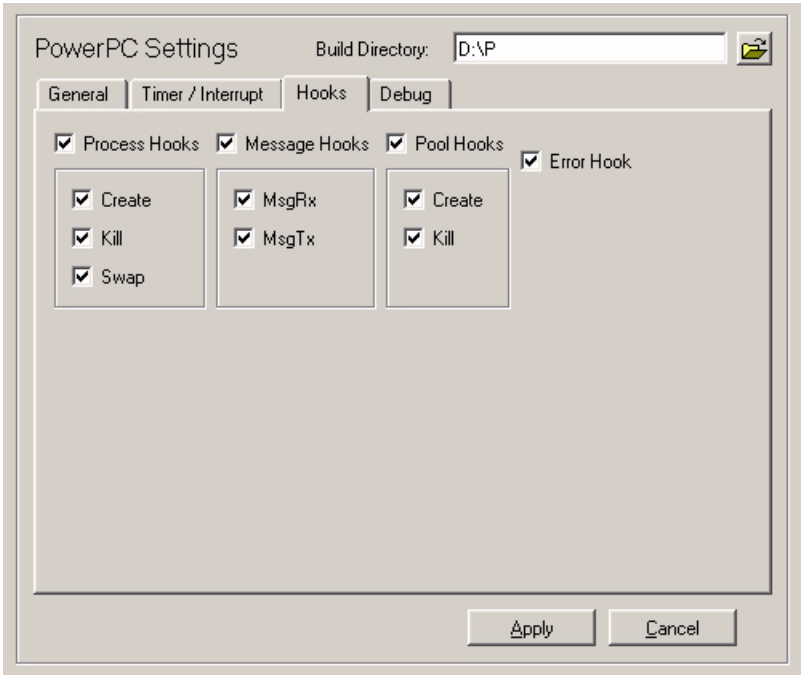
Interrupt

Enter your selected interrupt level for the PowerPC CPM. Please consult the PowerPC User´s Manual for more information about the CPM.

Please Note:

This setting does not exist for the PowerPC MPC500 processors.

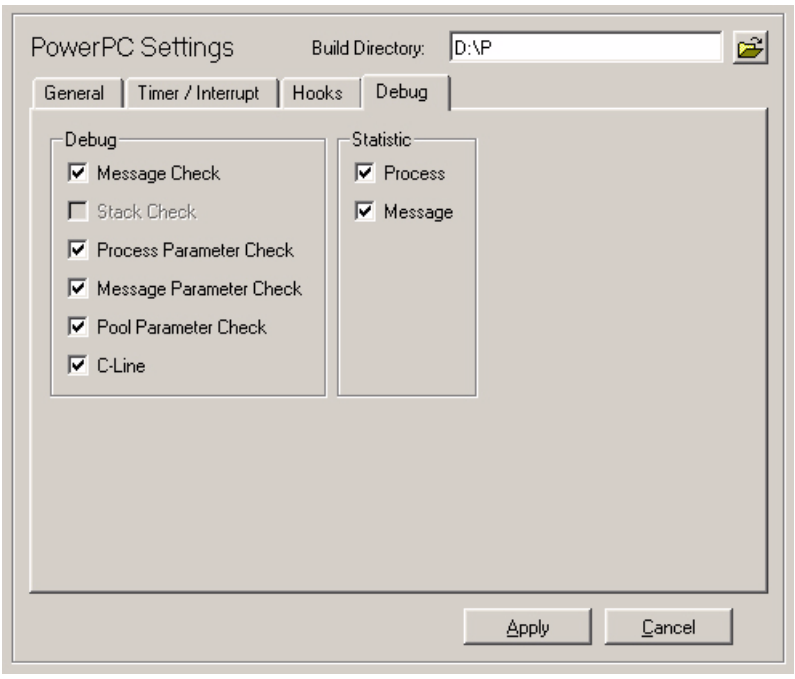
3.9.4.1 Configuring Hooks



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

You can enable all existing hooks separately by selecting the corresponding check box.

3.9.4.2 Debug Configuration



Message Check

If you are selecting this check box some test functions on messages will be included in the kernel.

Process Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the process system calls.

Message Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the message system calls.

Pool Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the pool system calls.

C-Line

If you are selecting this check box the kernel will include line number information which can be used by the SCIOPTA DRUID Debug System or an error hook. Line number and file of the last system call is recorded in the per process data.

Process Statistics

If you are selecting this check box the kernel will maintain a process statistics data field where information such as number of process swaps can be read.

Message Statistics

If you are selecting this check box the kernel will maintain a message statistics data field in the pool control block where information such as number of message allocation can be read.

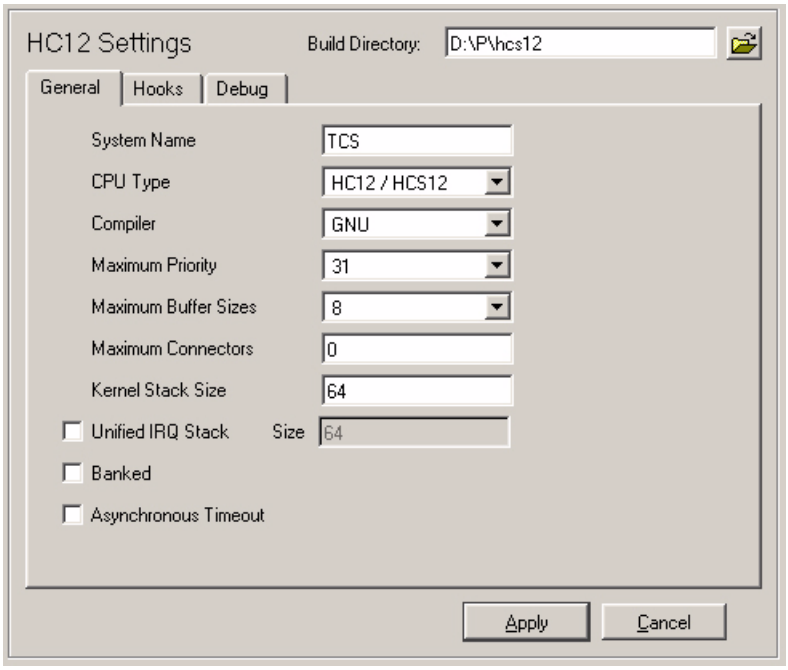
Applying Target Configuration

Click on the Apply button to accept the target configuration settings.

3.9.5 Configuring HCS12 Target Systems

The system configuration for HCS12 target systems is divided into 3 tabs: General, Hooks and Debug.

3.9.5.1 General Configuration



System Name

Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.

CPU Type

Give the name of you specific target processor derivative.

Compiler

Select the C/C++ Cross compiler which you are using for this SCIOPTA system.

Maximum Priority

Select the maximum priority level in the HCS12 system. The process priorities can then have priority levels between 0 (highest) to the values set inside this field (lowest).

Maximum Buffer 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 or 8 sizes which is configured here in the maximum buffer sizes entry.

Maximum Connectors

CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA systems. You can configure a maximum number of 15 CONNECTORS.

Kernel Stack Size

Stack size for kernel usage.

Unified IRQ Stack Checkbox

When this checkbox is selected all interrupt processes share the same stack. The size of the stack can be defined.

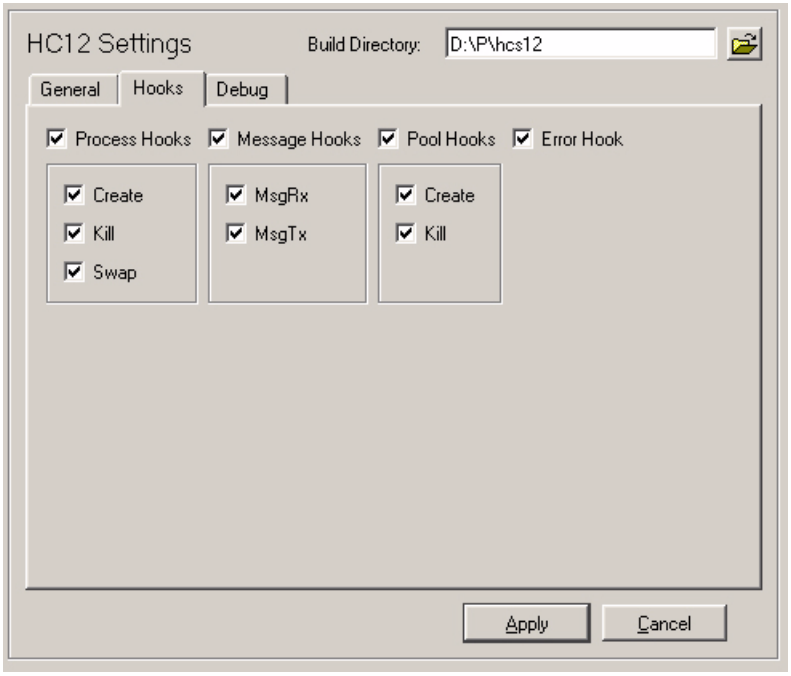
Banked Checkbox

Enables bankswitching support in the kernel.

Asynchronous Timeout

When checked the SCIOPTA HCS12 kernel includes timeout-server functionality.

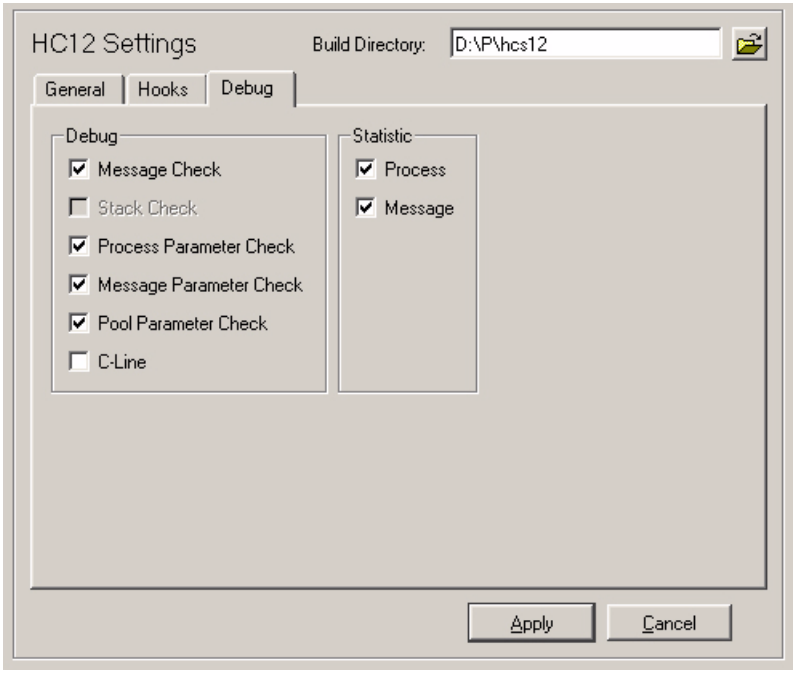
3.9.5.2 Configuring Hooks



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

You can enable all existing hooks separately by selecting the corresponding check box.

3.9.5.3 Debug Configuration



Message Check

If you are selecting this check box some test functions on messages will be included in the kernel.

Process Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the process system calls.

Message Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the message system calls.

Pool Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the pool system calls.

C-Line

If you are selecting this check box the kernel will include line number information which can be used by the SCIOPTA DRUID Debug System or an error hook. Line number and file of the last system call is recorded in the per process data.

Process Statistics

If you are selecting this check box the kernel will maintain a process statistics data field where information such as number of process swaps can be read.

Message Statistics

If you are selecting this check box the kernel will maintain a message statistics data field in the pool control block where information such as number of message allocation can be read.

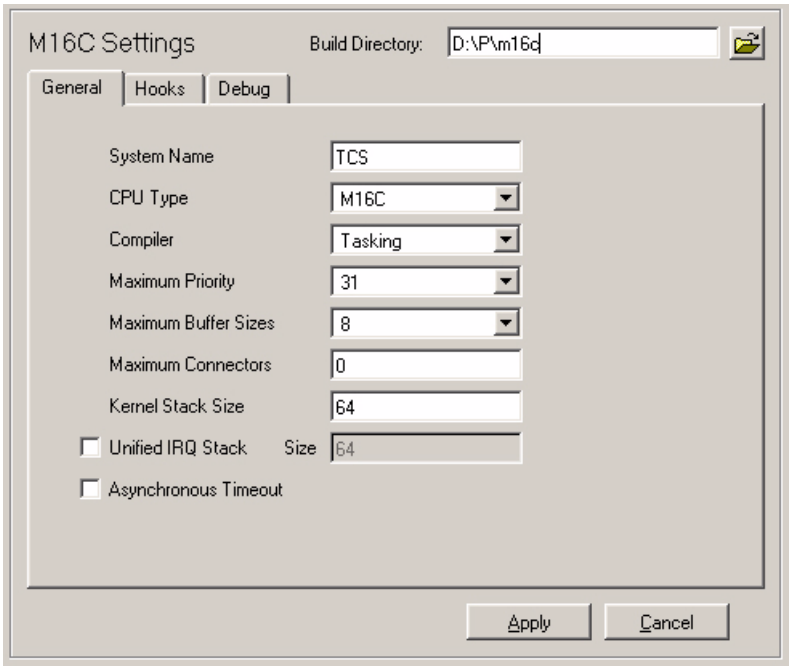
Applying Target Configuration

Click on the Apply button to accept the target configuration settings.

3.9.6 Configuring M16C Target Systems

The system configuration for M16C target systems is divided into 3 tabs: General, Hooks and Debug.

3.9.6.1 General Configuration



System Name

Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.

CPU Type

Give the name of you specific target processor derivative.

Compiler

Select the C/C++ Cross compiler which you are using for this SCIOPTA system.

Maximum Priority

Select the maximum priority level in the M16C system. The process priorities can then have priority levels between 0 (highest) to the values set inside this field (lowest).

Maximum Buffer 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 or 8 sizes which is configured here in the maximum buffer sizes entry.

Maximum Connectors

CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA systems. You can configure a maximum number of 15 CONNECTORS.

Kernel Stack Size

Stack size for kernel usage.

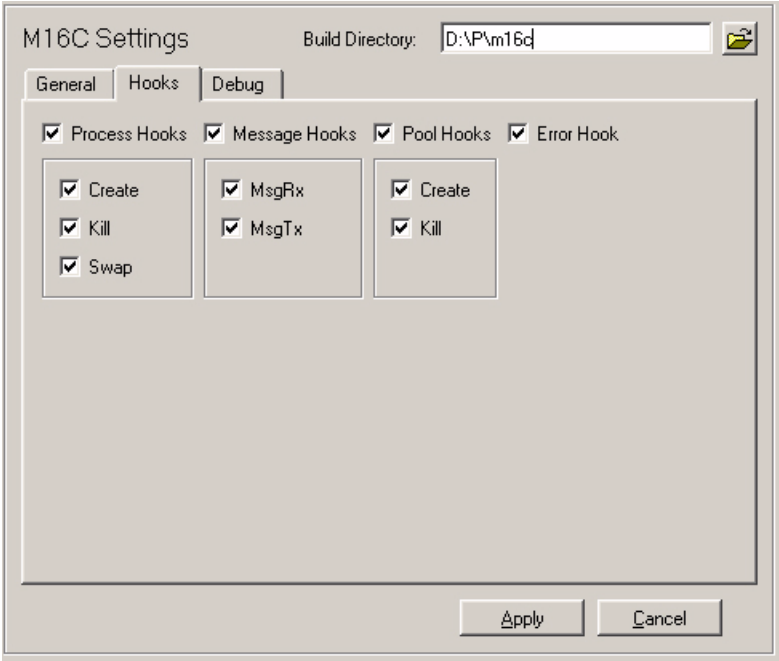
Unified IRQ Stack Checkbox

When this checkbox is selected all interrupt processes share the same stack. The size of the stack can be defined.

Asynchronous Timeout

When checked the SCIOPTA M16C kernel includes timeout-server functionality.

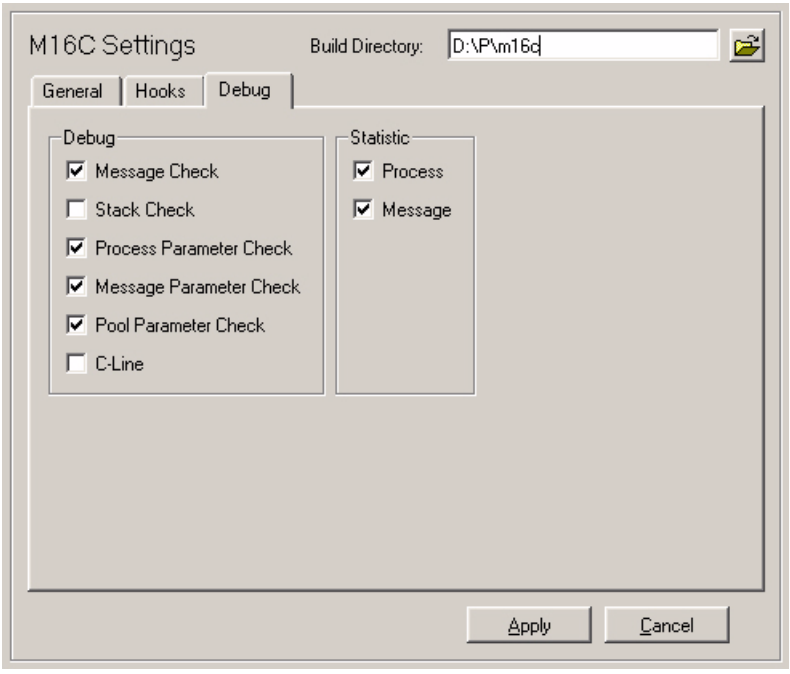
3.9.6.2 Configuring Hooks



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

You can enable all existing hooks separately by selecting the corresponding check box.

3.9.6.3 Debug Configuration



Message Check

If you are selecting this check box some test functions on messages will be included in the kernel.

Stack Check

Stack check is currently not available in SCIOPTA M16C.

Process Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the process system calls.

Message Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the message system calls.

Pool Parameter Check

If you are selecting this check box the kernel will do some testing on the parameters of the pool system calls.

C-Line

C-Line check is currently not available in SCIOPTA M16C.

Process Statistics

If you are selecting this check box the kernel will maintain a process statistics data field where information such as number of process swaps can be read.

Message Statistics

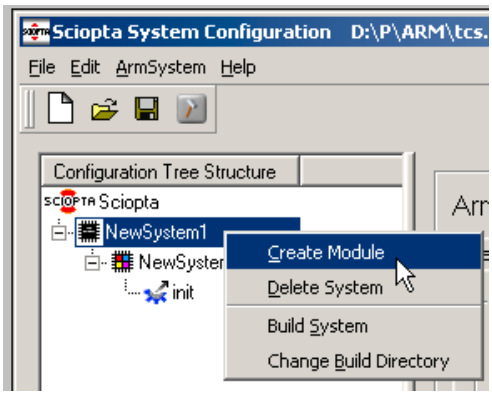
If you are selecting this check box the kernel will maintain a message statistics data field in the pool control block where information such as number of message allocation can be read.

Applying Target Configuration

Click on the Apply button to accept the target configuration settings.

3.10 Creating Modules

From the system level you can create new modules. Move the mouse pointer over the system and right-click the mouse.

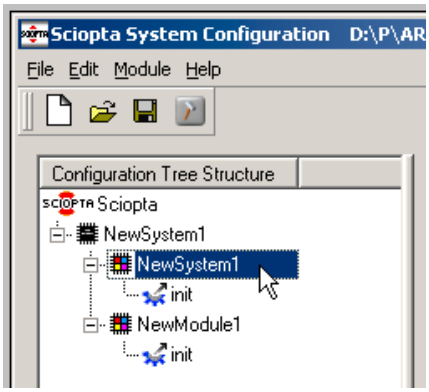


Modules can only be created for SCIOPTA 32-Bit systems. In SCIOPTA 16-Bit systems (SCIOPTA Compact) there is only one module (the system module).

A pop-up menu appears and allows you to create a new module.

The same selection can be made by selecting the Target System from the menu bar.

A new module for your selected target with a default name and an **init process** in the module will be created.

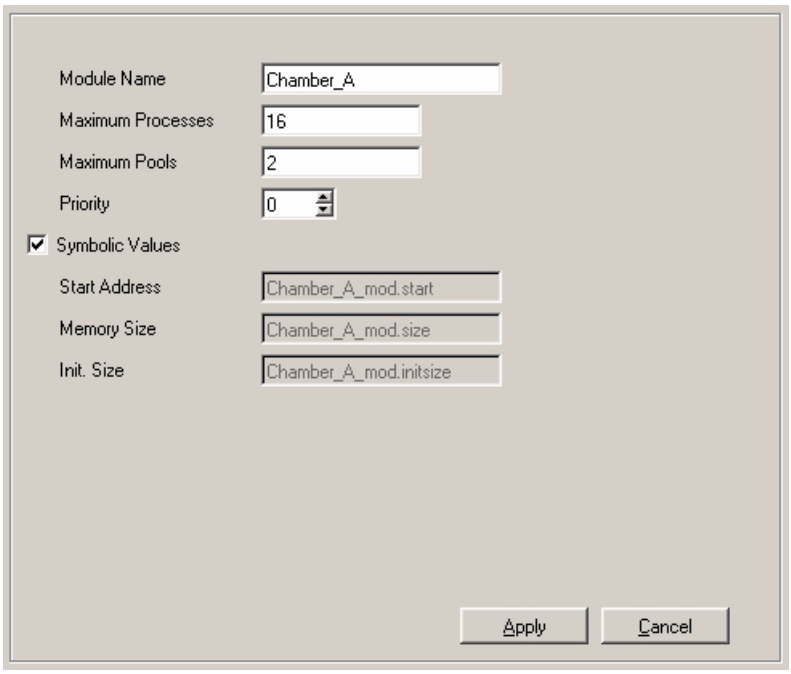


You can create up to 127 modules inside a SCIOPTA 32-Bit system.

You are now ready to configure the individual modules.

3.11 Configuring Modules

After selecting a module with your mouse, the corresponding parameter window on the right side will show the module parameters.



Module Name

Enter the name of the module. If you have selected the system module (the first module or the module with the id 0) you cannot give or modify the name as it will have the same name as the target system.

For SCIOPTA 16-Bit systems (SCIOPTA Compact) there is only one module, the module name gets the name from the system and cannot be modified here on the module level.

Maximum Processes

Maximum number of processes in the module. The kernel will not allow to create more processes inside the module than stated here.

For SCIOPTA 32-Bit systems the maximum value is 16383.

For SCIOPTA 16-Bit systems (SCIOPTA Compact) the maximum value is 255.

Maximum Pools

Maximum number of pools in the module. The kernel will not allow to create more pools inside the module than stated here.

For SCIOPTA 32-Bit systems the maximum value is 128.

For SCIOPTA 16-Bit systems (SCIOPTA Compact) the maximum value is 15. A SCIOPTA 16-Bit system must have at least two pools.

Priority

This entry is only available for SCIOPTA 32-Bit systems.

Enter the priority of the module.

Each module has also a priority which can range between 0 (highest) to 31 (lowest) priority. For process scheduling SCIOPTA uses a combination of the module priority and process priority called **effective priority**.

Please consult the SCIOPTA Kernel, User's Guide and Reference Manual for more information about module and process priority.

Start Address

This entry is only available for SCIOPTA 32-Bit systems.

This is the start address of the module in RAM.

Best is to specify a label which will be resolved at link time (e.g. <module_name>_mod). The specified label will be used in the linker script. Therefore all memory allocation for all modules is controlled by the linker script.

You may specify an absolute address, but you need to be very carefully check with the linker script to avoid overlapping.

Memory Size

This entry is only available for SCIOPTA 32-Bit systems.

Size of the module in bytes (RAM).

Best is to specify a label which will be resolved at link time (e.g. <module_name>_size). The specified label will be used in the linker script. Therefore all memory allocation for all modules is controlled by the linker script.

You may specify an absolute address, but you need to be very carefully check with the linker script to avoid overlapping.

The minimum module size can be calculated according to the following formula (bytes):

size_mod = p * 128 + stack + pools + mcb + textsize

where:

p	Number of static processes
stack	Sum of stack sizes of all static processes
pools	Sum of sizes of all message pools
mcb	module control block (see below)
textsize	Init size (see below)

The size of the module control block (mcb) can be calculated according to the following formula (bytes):

size_mcb = 96 + friends + hooks * 4 + c

where:

friends	if friends are not used: 0 if friends are used 16 bytes
hooks	number of hooks configured
c	For ARM and PowerPC: c = 8 For Coldfire c = 0

Please consult chapter **3.9.1.1 “General Configuration” on page 3-8** and **3.9.1.2 “Configuring Hooks” on page 3-10** for information about friend and hook settings.

Init Size

This entry is only available for SCIOPTA 32-Bit systems.

Size of the memory which is initialized by the C-startup function (cstartup.S).

Best is to specify a label which will be resolved at link time (e.g. <module_name>_initsize). The specified label will be used in the linker script. Therefore all memory allocation for all modules is controlled by the linker script.

You may specify an absolute address, but you need to be very carefully check with the linker script to avoid overlapping.

Symbolic Values

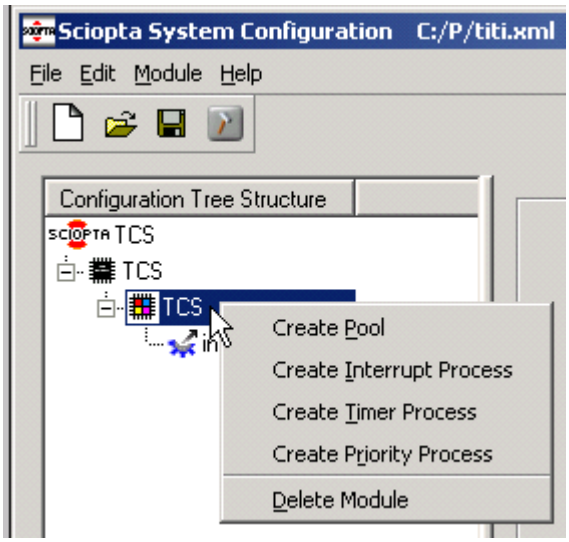
You need to select this checkbox if you want to specify labels instead of absolute values for the module addresses and module size (start address, memory size and init size). The labels (<module_name>_mod, <module_name>_size and <module_name>_initsize) will be used by the linker script and resolved at link time. Therefore all memory allocation for all modules is controlled by the linker script.

Applying Module Configuration

Click on the Apply button to accept the module configuration settings.

3.12 Creating Processes and Pools

From the module level you can create new processes and pools. Move the mouse pointer over the module and right-click the mouse.

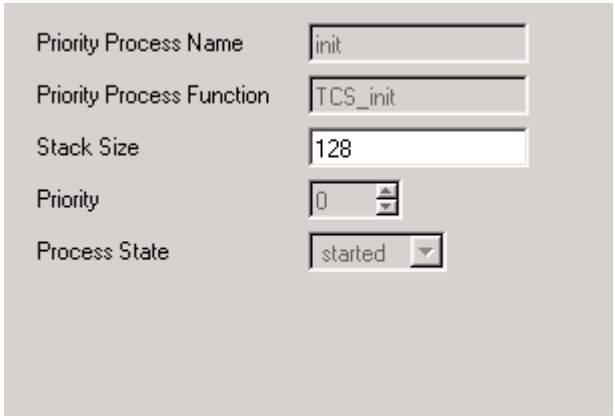


A pop-up menu appears and allows you to create pools, interrupt processes, timer processes and prioritized processes.

The same selection can be made by selecting the **Module** menu from the menu bar.

3.13 Configuring the Init Process

After selecting the init process with your mouse the parameter window on the right side will show the configuration parameters for the init process. There is always one init process per module and this process has the highest priority. Only the stack size of the init process can be configured.



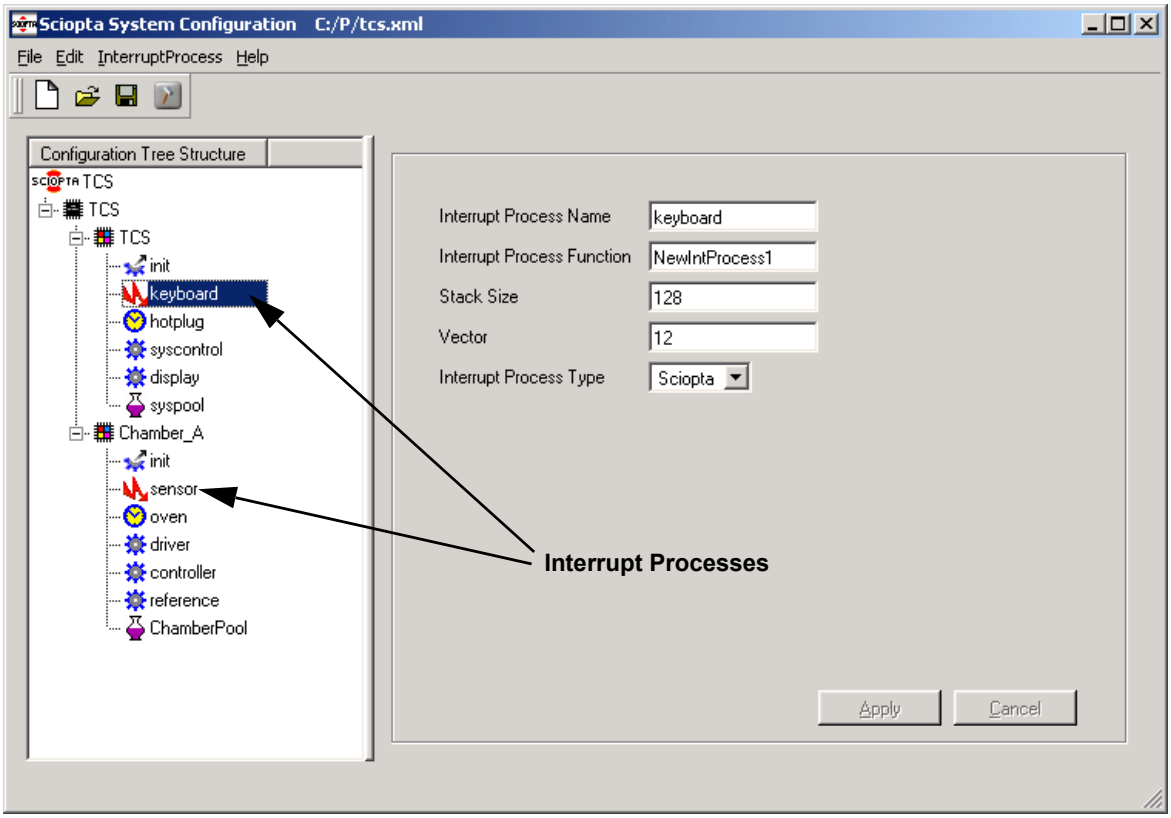
Stack Size

Enter the stack size for the init process.

Applying Init Process Configuration

Click on the Apply button to accept the init process configuration settings.

3.14 Interrupt Process Configuration



Interrupt Process Name

Enter the name of the interrupt process.

Interrupt Process Function

Function name of the interrupt process function. This is the address where the created process will start execution.

Stack Size

Stacksize of the created process in bytes.

Vector

Enter the interrupt vector connected to the interrupt process. Please consult the target manuals for more information about interrupt handling and vectors.

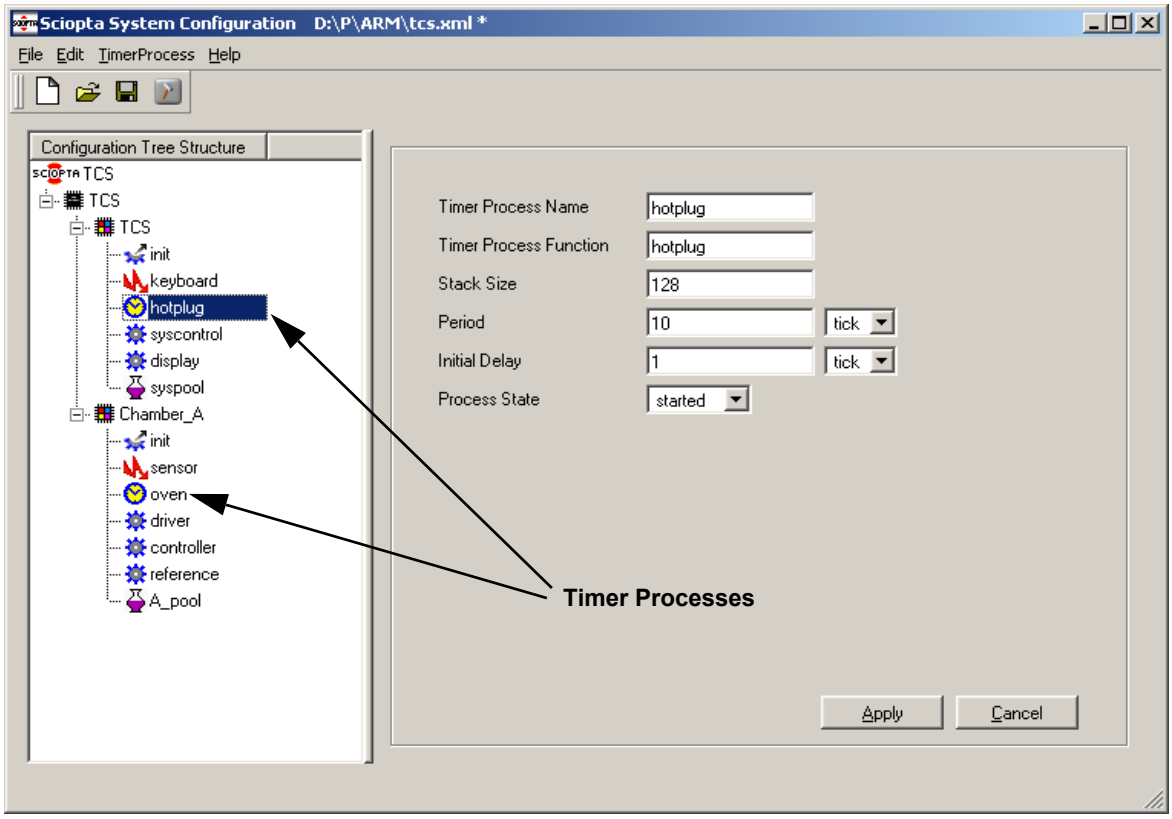
Interrupt Process Type

Select the type of the interrupt process. Interrupt processes of type **Sciopta** are handled by the kernel and may use (not blocking) system calls while interrupt processes of type **User** are handled outside the kernel and may not use system calls.

Applying the Interrupt Process Configuration

Click on the Apply button to accept the interrupt process configuration settings.

3.15 Timer Process Configuration



Timer Process Name

Enter the name of the timer process.

Timer Process Function

Enter the function name of the timer process function. This is the address where the created process will start execution.

Stack Size

Stacksize of the created process in bytes.

Period

Period of time between calls to the timer process in ticks or in milliseconds.

Initial Delay

Initial delay in ticks before the first time call to the timer process in ticks or milliseconds.

Process State

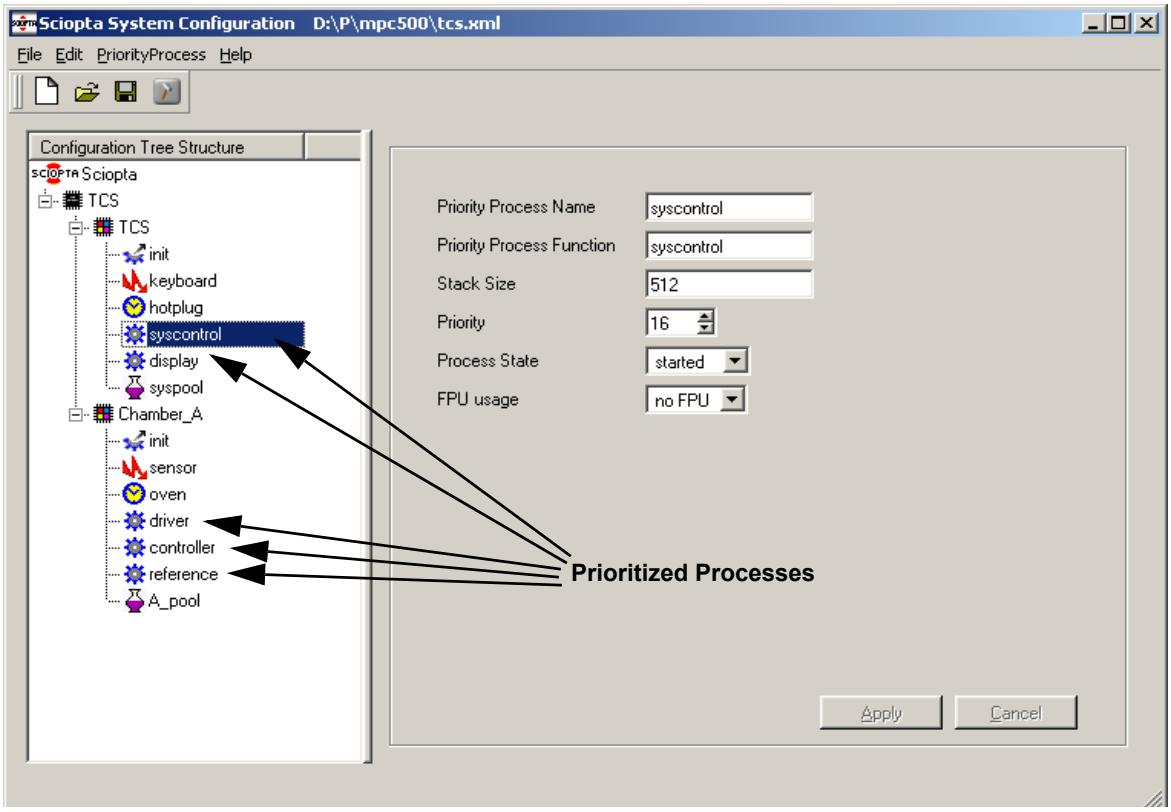
Enter the state which the process should have after creation.

started	The process will be on **READY** state. It is ready to run and will be swapped-in if it has the highest priority of all **READY** processes.
stopped	The process is stopped. Use the **sc_procStart** system call to start the process.

Applying the Timer Process Configuration

Click on the Apply button to accept the timer process configuration settings.

3.16 Prioritized Process Configuration



Priority Process Name

Enter the name of the prioritized process.

Priority Process Function

Enter the function name of the prioritized process function. This is the address where the created process will start execution.

Stack Size

Stacksize of the created process in bytes.

Priority

Enter the process priority.

For process scheduling SCIOPTA uses a combination of the module priority and process priority called **effective priority**.

Process State

Enter the state which the process should have after creation.

started	The process will be on **READY** state. It is ready to run and will be swapped-in if it has the highest priority of all **READY** processes.
stopped	The process is stopped. Use the **sc_procStart** system call to start the process.

FPU Usage

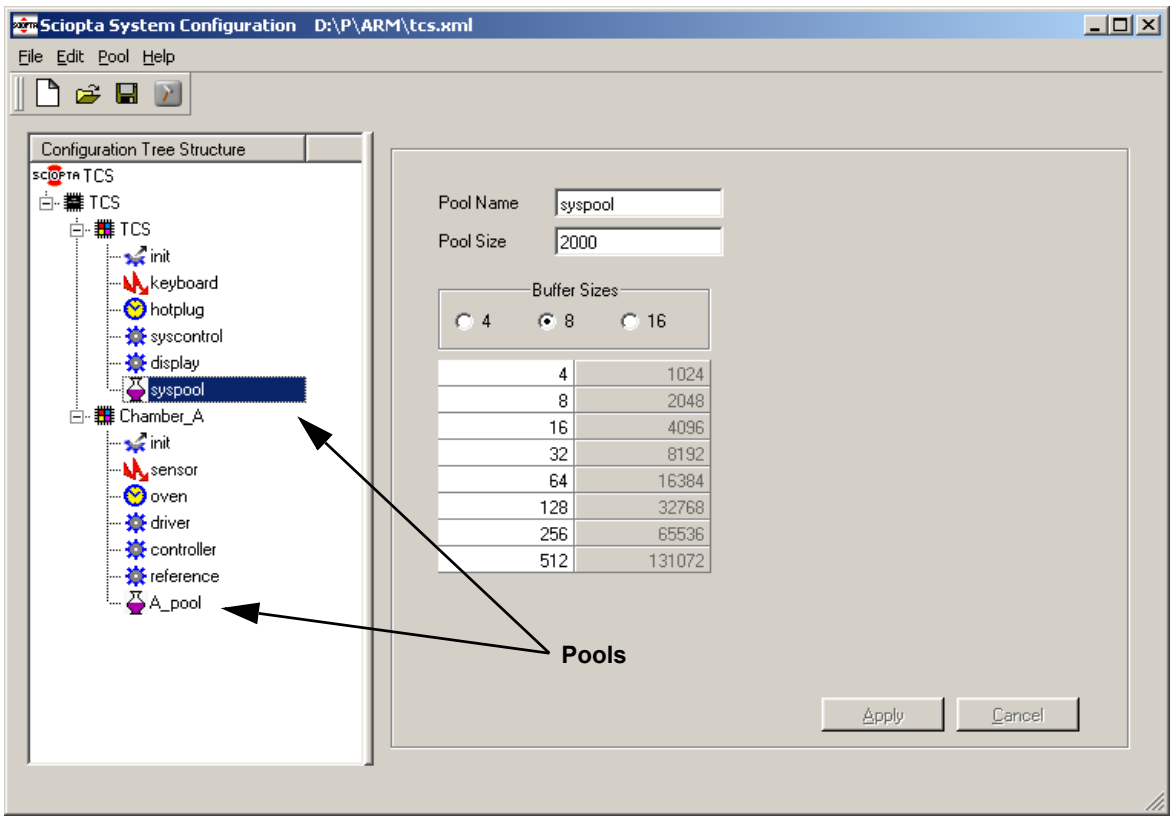
This checkbox exists only for CPU's which have a Floating Point Unit.

Enter your selection if you want to use the floating point unit (FPU) of the processor.

Applying the Priority Process Configuration

Click on the Apply button to accept the priority process configuration settings.

3.17 Pool Configuration



Pool Name

Enter the name of the pool.

Pool Size

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

For **SCIOPTA 32-Bit** systems the size of the pool control block (pool_cb) can be calculated according to the following formula:

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

where:

n	buffer sizes (4, 8 or 16)
stat	process statistics (see chapter **"Process Statistics" on page 3-12**) or message statistics (see chapter **"Message Statistics" on page 3-12**) are used (1) or not used (0).

For **SCIOPTA 16-Bit** systems (SCIOPTA Compact) the size of the pool control block (pool_cb) can be calculated according to the following formula:

pool_cb = 15 + n *6 + stat * n * 10

where:

n	buffer sizes (4 or 8)
stat	process statistics (see chapter **"Process Statistics" on page 3-12**) or message statistics (see chapter **"Message Statistics" on page 3-12**) are used (1) or not used (0).

Buffer Sizes

For SCIOPTA 32-Bit systems select 4,8 or 16 fixed buffer sizes for the pool. Define the different buffer sizes for your selection.

For SCIOPTA 16-Bit systems (SCIOPTA Compact) select 4 or 8 fixed buffer sizes for the pool. Define the different buffer sizes for your selection.

Applying the Pool Configuration

Click on the Apply button to accept the pool configuration settings.

3.18 Build

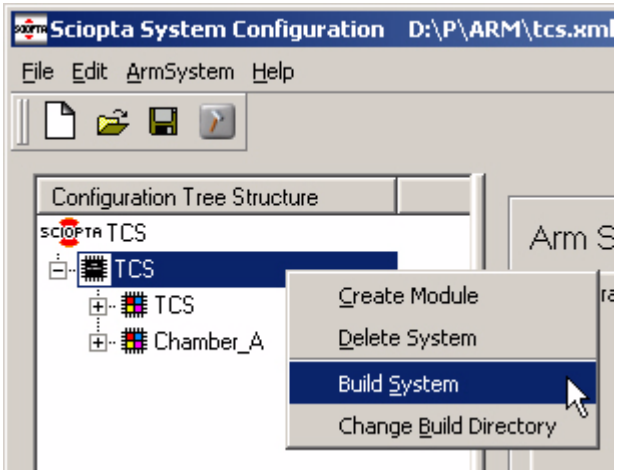
The **SCONF** will generate three files which need to be included into your SCIOPTA project.

sciopta.cnf	This is the configured part of the kernel which will be included when the SCIOPTA kernel (sciopta.s) is assembled.
sconf.h	This is a header file which contains some configuration settings.
sconf.c	This is a C source file which contains the system initialization code.

Please consult system building chapter of the target manuals for more information about system building.

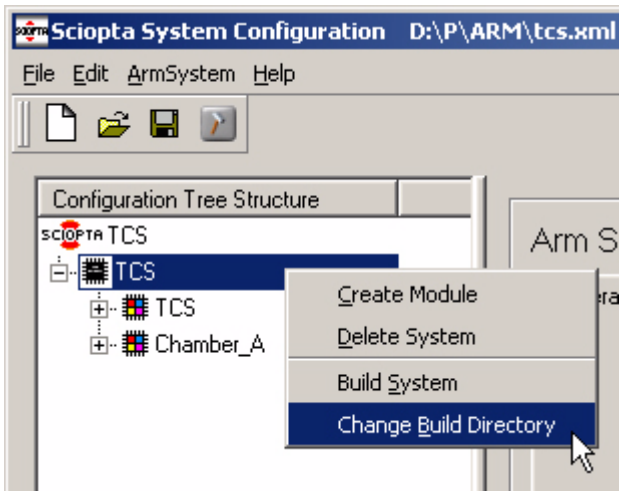
3.18.1 Build System

To build the three files click on the system and right click the mouse. Select the menu **Build System**. The files sciopta.cnf, sconf.h and sconf.c will be generated into your defined build directory.



3.18.2 Change Build Directory

When you are creating a new system, **SCONF** ask you to give the directory where the three generated files will be stored. You can modify this build directory for each system individually by clicking to the system which you want to build and right click the mouse.



Select the last item in the menu for changing the build directory.

The actual Build Directory is shown in the System Settings Window:

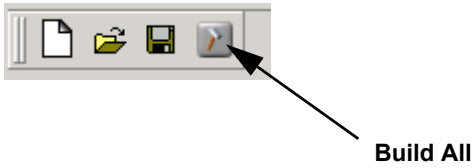
You can change the Build Directory also from the System Settings Window. Click on the Browse Button and select the new directory.

You can change the Build Directory also from the System Settings Window by entering directly the Build Directory Path.

3.18.3 Build All

If you have more than one system in your project, you can build all systems at once by clicking on the Build All button.

Select the **Build All** button from the button bar to generate the set of three files for each system.



The files sciopta.cnf, sconf.h and sconf.c will be generated for every target into the defined build directories of each target which exists in the project.

SCONF will prompt for generating the files for each system.

Please note:

You need to have different build directories for each system as the names of the three generated files are the same for each system.

3.19 Command Line Version

3.19.1 Introduction

The **SCONF** configuration utility can also be used from a command line.

This is useful if you want to modify or create the XML configuration file manually or if the XML configuration file will be generated by a tool automatically and you want to integrate the configuration process in a makefile. The best way to become familiar with the structure of the XML file is to use the graphic **SCONF** tool once and save the XML file.

3.19.2 Syntax

By calling the **SCONF** utility with a **-c** switch, the command line version will be used automatically.

```
<install dir>\bin\win32\sconf.exe  -c  <XML File>
```

You need to give also the extension of the XML file.

4 System Design

4.1 Introduction

System design includes all phase from system analysis, through specification to system design, coding and testing. In this chapter we will give you some useful information of what methods, techniques and structures are available in SCIOPTA to fulfil your real-time requirements for your embedded system.

4.2 System Partition

When you are analysing a real-time system you need to partition a large and complex real-time system into smaller components and you need to design system structures and interfaces carefully. This will not only help in the analysing and design phase, it will also help you maintaining and upgrading the system.

In a SCIOPTA controlled real-time system you have different structure elements available to decompose the whole system into smaller parts. The following chapters describe the possibilities which is offered by SCIOPTA.

4.3 Modules

SCIOPTA allows you to group tasks 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.

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. Modules can be moved and copied between CPU´s and systems

You can define modules from the SCIOPTA configuration tool **SCONF**.These modules are created automatically by the operating system at startup.

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.

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.

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.

Please Note:

The module concept is not supported in the **SCIOPTA Compact Kernel**. The **SCIOPTA Compact Kernel** has only one module (the system module).

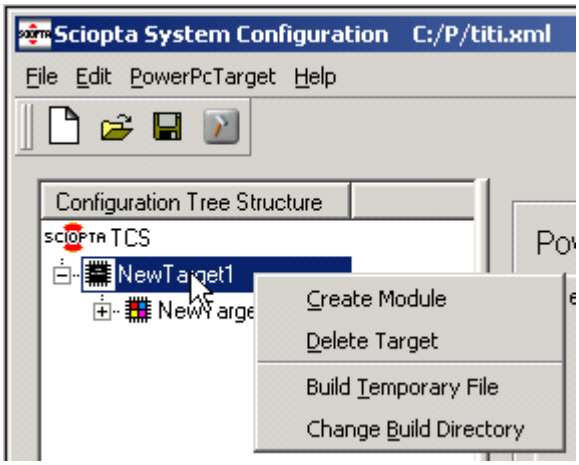


Figure 4-1: Module Creation by SCONF

Please consult the SCIOPTA - Target Manual of your processor for more information about module creation by the SCONF tool.

Another way is to create modules dynamically by the **sc_moduleCreate()** system call.

```
sc_moduleid_t sc_moduleCreate(const char *name,           /* name of module */
                              void (*init)(void),        /* init-process */
                              sc_bufsize_t stacksize,    /* stack-size of init process */
                              sc_prio_t prio,            /* module priority */
                              char *start,
                              sc_modulesize_t size,
                              sc_modulesize_t textsize,
                              unsigned int max_pools,
                              unsigned int max_procs);
```

Figure 4-2: Module Creation by sc_moduleCreate() System Call

4.4 Resource Management

Typical resources in a real-time operating system such as peripheral devices or memory must usually be shared between different clients. The system must be designed in a way to ensure mutual exclusion while accessing the resource for writing and reading.

A good practice in SCIOPTA to realize such a mutual exclusion is to encapsulate the shared resources inside a SCIOPTA process. User processes can communicate with the shared resource by sending and receiving SCIOPTA message to and from the encapsulating process.

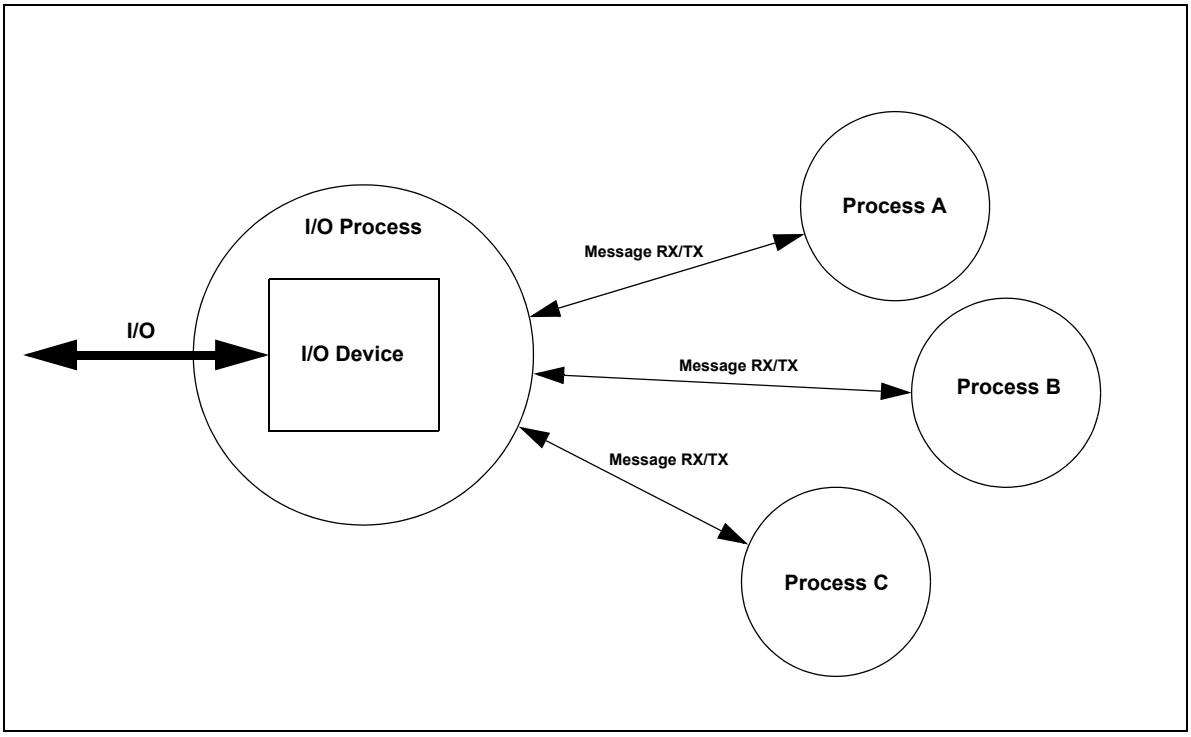


Figure 4-3: Resource Encapsulation

Please consult also the manual: SCIOPTA Board Support Package, User´s Guide where detailed information about the SCIOPTA device driver concept can be found.

4.5 Processes

4.5.1 Introduction

In SCIOPTA a process can be seen as an independent program which executes as if it has the whole CPU available. The operating systems guarantees that always the most important process at a certain moment is executing. If a more important process (with a higher priority) wants to run, SCIOPTA will swap-out the actual running process and swap-in the new process and allow the execution. This is called a process switch.

There are different type of processes available in SCIOPTA which differ mainly in the way they are activated and running

- prioritized process
- interrupt process
- timer process
- init process

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 a **procIdGet()** system call including the process name.

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. 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.

4.5.2 Prioritized Processes

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 completed its work it becomes not ready and will be swapped out by the operating system. The “not-ready” state is entered if a process is waiting for a message (**sc_msgRx()** receive system call), requesting a delay (**sc_sleep()** sleep system call) or call other blocking SCIOPTA system calls. A process can also be swapped out by SCIOPTA when another process with higher priority becomes ready and wants the CPU.

SCIOPTA is a pre-emptive kernel. Therefore processes can be interrupted any time even inside any C instruction or between almost any two assembler instructions.

Prioritized processes can use all SCIOPTA system calls. They must never end and should be written in such a way to loop back and the beginning and waiting for another system event.

Prioritized process can be created and killed dynamically with the **sc_procPrioCreate()** and **sc_procKill()** system calls.

4.5.2.1 Process Declaration Syntax

Description

All prioritized processes in SCIOPTA must contain the following declaration:

Syntax

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

Parameter

proc_name	Name of the prioritized process.

4.5.2.2 Process Template

In this chapter a template for a prioritized process in SCIOPTA is provided.

```
#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 */


   }

}
```

4.5.3 Interrupt Processes

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 program handling 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.

In traditional real-time operating systems, an interrupt process is a function which executes exclusively by a hardware event. In SCIOPTA there is an additional parameter introduced which allows the execution of the interrupt process by other events such as:

- init (defined when the process is created)
- exit (defined when process is killed)
- hardware (defined if a hardware event occurs)
- msg (defined if a message is sent to the interrupt process)
- trigger (defined if a trigger event should activate the interrupt process)

SCIOPTA interrupt processes may never suspend itself and become not ready. It must run from the beginning to the end each time it is called. Therefore SCIOPTA interrupt processes may not use blocking system calls such as waiting for a message with time-out or sleeping.

4.5.3.1 Interrupt Process Declaration Syntax

Description

All interrupt processes in SCIOPTA must contain the following declaration:

Syntax

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

Parameter

proc_name Name of the interrupt process.

irq_src Interrupt source. Depending of this value the interrupt process can execute different code for different interrupt sources. The following values are defined:

0	The interrupt process is activated by a real hardware interrupt.
1	The interrupt process is activated by a message sent to the interrupt process.
2	The interrupt process is activated by a trigger event.
-1	The interrupt process is activated when the process is created. This allows the interrupt process to execute some initialization code.
-2	The interrupt process is activated when the process is killed. This allows the interrupt process to execute some exit code.

irg_src is of type **int**.

4.5.3.2 Interrupt Process Template

In this chapter a template for an interrupt process in SCIOPTA is provided.

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

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

   if (irq_src == 0)     /* Generated by hardware */
  {

      /* Code for hardware interrupt handling */

   }
   else if (irq_src == -1)    /* Generated when process created */
   {

      /* Initialization code */

   }
   else if (irq_src == -2) /* Generated when process killed */
   {

      /* Exit code */

   }
   else if (irq_src == 1)     /* Generated by a message sent to this */
   {                          /* interrupt process */

      /* Code for receiving a message */

   }
   else if (irq_src == 2)     /* Generated by a SCIOPTA */
                              /* trigger event */
   {

      /* Code for trigger event handling */

   }
}
```

4.5.4 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.

Timer processes behaves and are designed the same way as interrupt processes. All information given in chapter **4.5.3 "Interrupt Processes" on page 4-6** are also valid for timer processes.

4.5.4.1 Timer Process Declaration Syntax

Description

All timer processes in SCIOPTA must contain the following declaration:

Syntax

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

Parameter

Same as for interrupt processes. Please consult chapter **4.5.3.1 "Interrupt Process Declaration Syntax" on page 4-7**.

4.5.4.2 Timer Process Template

You can use the same template as for interrupt processes (see chapter **4.5.3.2 "Interrupt Process Template" on page 4-8**).

4.5.5 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.

4.5.5.1 Init Process in Static Modules

Static modules are defined and configured in the **SCONF** configuration utility. In static modules the init process is created and started automatically. The code of the init process is generated automatically by the SCONF configuration tool and included in the file sconf.c. At start-up the init process gets the highest priority (0).

The user can write a function (with the name of the module) which will be called by the init process. This function can include late start-up code for a SCIOPTA system. The function executes on the (highest) priority level of the init process. The init process will create all static SCIOPTA objects such as processes and pools.

At the end the init process sets its priority level to 32.

This is a code fragment of a typical sconf.c file which was automatically generated (module name = HelloSciopta):

```
SC_PROCESS(HelloSciopta_init)
{
 extern void HelloSciopta(void);
 sc_procSchedLock();
{
 static const sc_bufsize_t bufsizes[4]=
 {
  8,
  12,
  16,
  32
 };
 default_plid = sc_sysPoolCreate(0,0x1000,4,(sc_bufsize_t *)bufsizes,"default",0);
}
SCI_sysTick_pid = sc_sysProcCreate("SCI_sysTick",(void (*)
                (void))SCI_sysTick,256,127,0,1,0,0,0,SC_PROCUSRINTCREATESTATIC);

hello_pid = sc_sysProcCreate("hello",hello,512,0,16,1,0,0,0,
                                                    SC_PROCPRIOCREATESTATIC);

display_pid = sc_sysProcCreate("display",display,512,0,17,1,0,0,0,
                                                    SC_PROCPRIOCREATESTATIC);

 sc_procSchedUnlock();
 HelloSciopta();
 sc_procPrioSet(32);
 for(;;) ASM_NOP;
}
```

4.5.5.2 Init Process in Dynamic Modules

Dynamic modules are created and configured by the **sc_moduleCreate()** system call during run-time. In dynamic modules the init process is created and started automatically. 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 sc_moduleCreate() system call. At start-up the init process gets the highest priority (0).

Template of a minimal init process of a dynamic module:

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

4.5.6 Selecting Process Type

SCIOPTA provides different types of processes to help the system designer to realize real-time applications more efficiently. Every process type is designed to perform specific duties in a real-time system.

4.5.6.1 Prioritized Process

Prioritized process are the most used process types in a system. 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.

Prioritized process have a specific priority. The real-time system designer assigns priority to processes or group of processes in order to guarantee the real-time behaviour of the system.

4.5.6.2 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.

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.

4.5.6.3 Timer Process

Timer processes will be used for tasks which need to be executed at precise cyclic intervals. For instance checking a status bit or byte at well defined moments in time can be performed by timer processes.

Another example is to measure a voltage at regular intervals. As timer processes execute on the interrupt level of the timer interrupt it is assured that no voltage measurement samples are lost.

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

4.6 Addressing Processes

4.6.1 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.

4.6.2 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 the string

_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, myflags);
```

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, myflags);
```

4.6.3 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 such as **sc_procCreate**, **sc_procIntCreate, sc_procPrioCreate** and **sc_procTimCreate** will also return the process IDs which can be used for further addressing.

4.7 Interprocess Communication

4.7.1 Introduction

Interprocess communication needs to be designed carefully in a real-time system.

In SCIOPTA there are a view different ways to design interprocess communication which are presented in the following chapters.

4.7.2 SCIOPTA Messages

4.7.2.1 Description

Please consult chapter **2.4 “Messages” on page 2-8** for a introduction into SCIOPTA messages.

Messages are the preferred tool 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**.

4.7.2.2 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. You could use the extension .msg for such include files.

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

- Message number definition
- Message structure definition
- Message union declaration

4.7.2.3 Message Number

Description

The declaration of the message number 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.

Syntax

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

Parameter

msg_nr	Message number which should be unique for each message class.

4.7.2.4 Message Structure

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 **id** item must be the first declaration in the message structure. It is used by the SCIOPTA kernel to identify SCIOPTA messages. After the message ID (or message number) all structure members can be 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.

Syntax

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

Parameter

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

4.7.2.5 Message Union

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.

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>
      .
      .
      .
};
```

Parameter

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

4.7.2.6 Example

This is a very small example showing how to handle messages in a SCIOPTA process. The process “keyboard” just allocates a messages fills it with a character and sends it to a process “display”.

```
#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_DEFAULT_POOL);   /* Get process ID */
                                                     /* for process display */

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

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

4.7.3 SCIOPTA Trigger

4.7.3.1 Description

Please consult chapter 2.6 “Trigger” on page 2-13 for a introduction into SCIOPTA Trigger.

SCIOPTA triggers are sometimes used to synchronize two processes and can be used in place of SCIOPTA messages. Triggers should only be used if the designer has severe timing problems and are intended for these rare cases where message passing would be to slow.

Please note that SCIOPTA triggers can only be used for process synchronisation as they cannot carry data.

4.7.3.2 Example

This is a very small example how triggers can be used in SCIOPTA processes. A prioritized process is waiting on its trigger and will be executed when another process (in this case an interrupt process) is activating the trigger.

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

extern sc_pid_t    trigproc_pid

OS_INT_PROCESS (myint, 0)
{
      .
      .
      .
   sc_trigger (trigproc_pid); /* This call makes process trigproc ready */
}


/* 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 */
            .
            .
   }
}
```

4.8 SCIOPTA Memory Manager - Message Pools

4.8.1 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 for a standard kernel (32-bit) and up to 15 pools for a compact kernel (16-bit). 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.

4.8.2 Message Pool size

The minumum 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:

Standard (32-Bit) kernel:

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

where:

n	buffer sizes (4, 8 or 16)
stat	process statistics or message statistics are used (1) or not used (0).

Compact (16-Bit) kernel:

pool_cb = 15 + n * 6 + stat * n * 10 + P

where:

n	buffer sizes (4 or 8)
stat	process statistics or message statistics are used (1) or not used (0).
P	processor dependent. For MSP430 = 1, for all others = 0.

Please consult the configuration chapter of the target manual for more information about statistics.

4.8.3 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.

4.8.4 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.

4.8.5 Message Administration Block

Each SCIOPTA message contains a hidden data structure which will be used by the kernel. The user can access these message information only by specific SCIOPTA system calls. Information such as the process ID of the message owner, the message size, the process ID of the transmitting process and the process ID of the addressed process are included in the message header adminstration block. Please consult chapter **2.4 “Messages” on page 2-8** for more information about SCIOPTA messages.

For the standard kernels (32-bit) the size of the message header is 32 bytes.

For the compact kernels (16-bit) the size of the message header is 10 bytes.

Each SCIOPTA message can contain an end-mark. This end-mark is used for the kernel message check if the message check option is enabled at kernel configuration. Please consult the configuration chapter of the SCIOPTA target manual for more information about message check.

For the standard kernels (32-bit) the size of the end-mark is 4 bytes.

For the compact kernels (16-bit) the size of the end-mark is 1 byte.

4.9 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.

4.9.1 Process Daemon

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.

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 should be placed in the system module.

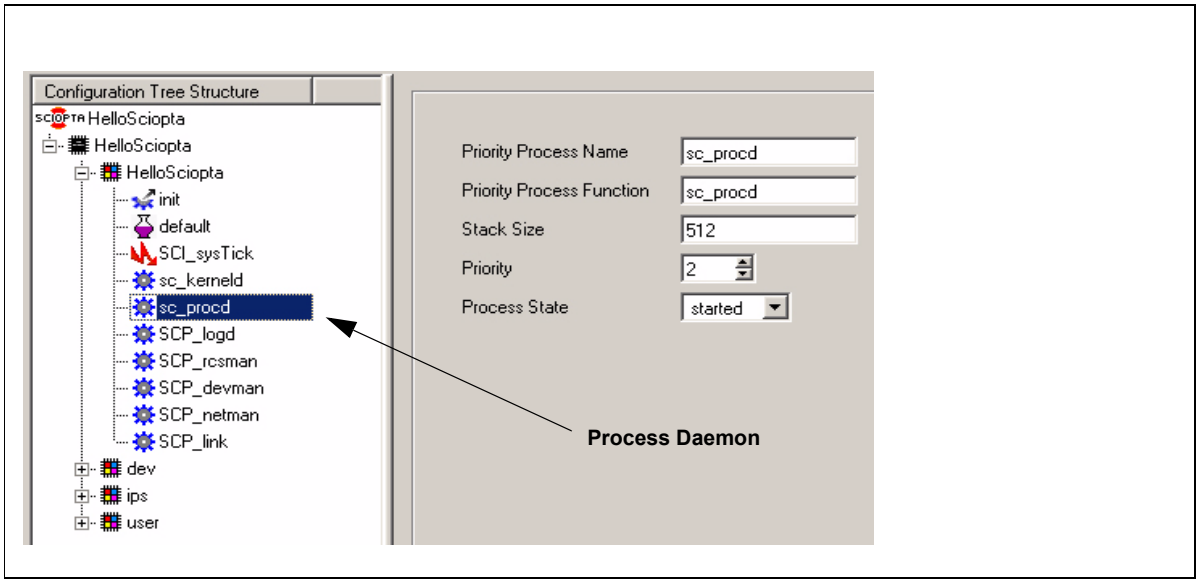


Figure 4-4: Process Daemon Declaration in SCONF

Please Note:

There is no process daemon to declare and to define in the **SCIOPTA Compact Kernel**. This function is defined and started automatically by the kernel.

4.9.2 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.

Whenever you are using process or module create or kill system call you need to start the kernel daemon.

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 should be placed in the system module.

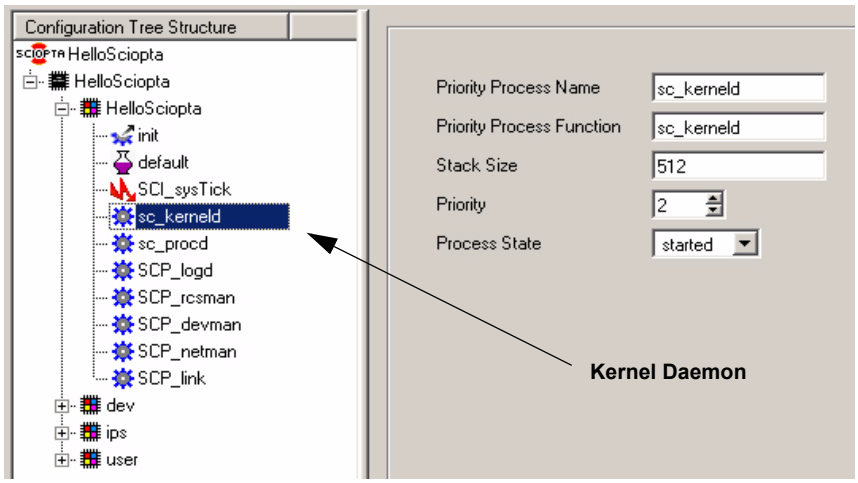


Figure 4-5: Kernel Daemon Declaration in SCONF

Please Note:

There is no kernel daemon to declare and to define in the **SCIOPTA Compact Kernel**. This function is defined and started automatically by the kernel.

4.10 Error Hook

4.10.1 Introduction

Contrary to most conventional real-time operating systems, SCIOPTA uses a centralized mechanism for error reporting called Error Hooks. In traditional real-time operating system, 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 are normal error handling functions and 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 (at label **SC_ERROR**) and all interrupts are disabled.

4.10.2 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**).

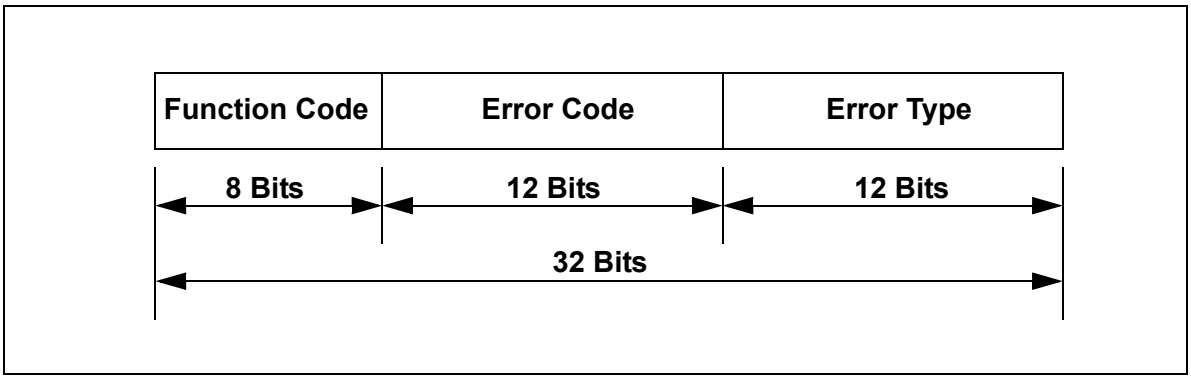


Figure 4-6: 32-bit Error Word (Parameter: errcode)

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

There is also an additional 32-bit extra word (parameter **extra**) available to the user.

Please consult the SCIOPTA - Kernel, Reference Manual the for a detailed description of the error codes.

4.10.3 Error Hook Registering

An error hook is registered by using the **sc_miscErrorHookRegister** system call by giving the error hook's name as parameter.

If the error hook is registered from within the system module it is registered as a global error 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.

4.10.4 Error Hook Declaration Syntax

Description

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

Syntax

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

   ... error hook code

};
```

Parameter

errcode	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. See chapter **4.10.2 "Error Information" on page 4-23**.	
extra	Gives additional information depending on the error code.	
user	user != 0	User error.
	user == 0	System error.
pcb	Pointer to the Process Control Block PCB of the process which generated the error.	

Return Value

!= 0	Continue/resume if error was not fatal.
== 0	Jumps to infinite loop at label **SC_ERRROR**.

4.10.5 Example

```
#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
```

4.11 System Start

4.11.1 Reset Hook

Description

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 must be written in assembler.

Syntax

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

Parameter

none

Return Value

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

4.11.2 C Startup

After a cold start the kernel will call the C startup function to initialize the C environment. This is target and compiler specific.

4.11.3 Start Hook

Description

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.

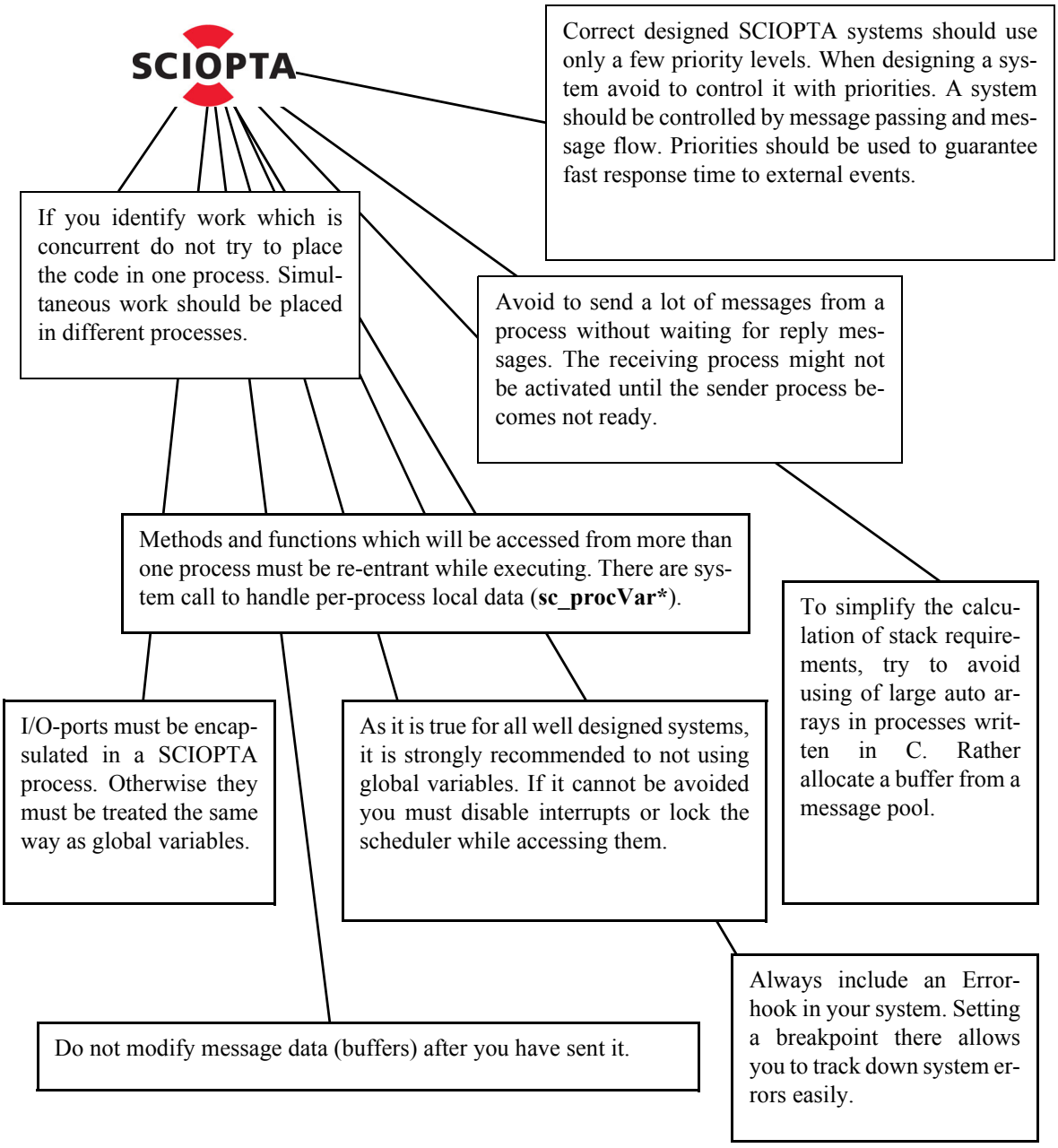
After the start hook has executed the kernel will call the dispatcher and the system will start.

Syntax

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

4.12 SCIOPTA Design Rules

As already stated in this document, SCIOPTA is a message based real-time operating system. Interprocess communication and synchronization is done by way of message passing. This is a very performant and strong design technology. Nevertheless the SCIOPTA user has to follow some rules to design message based systems efficiently and easy to debug.



5 Manual Revision

5.1 Manual Version 1.7

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

5.2 Manual Version 1.6

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

5.3 Manual Version 1.5

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

5.4 Manual Version 1.4

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

5.5 Manual Version 1.3

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

5.6 Manual Version 1.2

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

5.7 Manual Version 1.1

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

5.8 Manual Version 1.0

Initial version.

6 Index

Symbols

A

B

C

N

O

P

R

S

T

U

W

X