



**High Performance  
Real-Time Operating Systems**

---

**ARM Cortex  
Kernel**

**User's Guide**

# Copyright

Copyright (C) 2008 by SCIOPTA Systems AG. All rights reserved. No part of this publication may be reproduced, transmitted, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, optical, chemical or otherwise, without the prior written permission of SCIOPTA Systems AG. The Software described in this document is licensed under a software license agreement and maybe used only in accordance with the terms of this agreement.

# Disclaimer

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

# Trademark

**SCIOPTA** is a registered trademark of SCIOPTA Systems AG.

## Headquarters

SCIOPTA Systems AG  
Fiechthagstrasse 19  
4103 Bottmingen  
Switzerland  
Tel. +41 61 423 10 62  
Fax +41 61 423 10 63  
email: [sales@sciopta.com](mailto:sales@sciopta.com)  
[www.sciopta.com](http://www.sciopta.com)

## Table of Contents

<b>1.</b>	<b>Introduction .....</b>	<b>1-1</b>
1.1	SCIOPTA ARM Cortex Real-Time Operating System .....	1-1
1.2	About This Manual .....	1-2
1.3	Real-Time Operating System Overview .....	1-3
1.3.1	Management Duties .....	1-3
1.3.1.1	CPU Management .....	1-4
1.3.1.2	Memory Management .....	1-4
1.3.1.3	Input/Output Management .....	1-4
1.3.1.4	Time Management .....	1-4
1.3.1.5	Interprocess Communication .....	1-4
<b>2.</b>	<b>Installation.....</b>	<b>2-1</b>
2.1	Introduction .....	2-1
2.2	The SCIOPTA ARM Cortex Delivery .....	2-1
2.3	System Requirements.....	2-1
2.3.1	Windows .....	2-1
2.3.2	Linux .....	2-1
2.4	Installation Procedure Windows Hosts .....	2-2
2.4.1	Main Installation Window.....	2-2
2.4.2	Product Versions .....	2-3
2.4.3	Installation Location.....	2-3
2.4.4	Release Notes .....	2-3
2.4.5	Short Cuts.....	2-3
2.4.6	SCIOPTA_HOME Environment Variable.....	2-4
2.4.7	Setting SCIOPTA Path Environment Variable .....	2-4
2.4.8	Uninstalling SCIOPTA ARM Cortex .....	2-4
2.4.9	GNU Tool Chain Installation.....	2-5
2.4.10	Luminary Stellaris Cortex-M3 Family Driver Library .....	2-6
2.4.11	STMicroelectronics STM32 Firmware Library .....	2-6
2.4.12	Eclipse C/C++ Development Tooling - CDT .....	2-7
<b>3.</b>	<b>Getting Started.....</b>	<b>3-1</b>
3.1	Introduction .....	3-1
3.2	Getting Started - Hello Example .....	3-1
3.2.1	Description .....	3-1
3.2.2	Eclipse IDE and GNU GCC.....	3-2
3.2.2.1	Equipment .....	3-2
3.2.2.2	Step-By-Step Tutorial .....	3-3
3.2.3	Other Environments .....	3-5
3.2.3.1	Makefile and GNU GCC.....	3-5
3.2.3.2	iSYSTEM winIDEA and GNU GCC.....	3-5
3.2.3.3	IAR Embedded Workbench .....	3-5
<b>4.</b>	<b>SCIOPTA Kernel Project Overview.....</b>	<b>4-1</b>
4.1	Introduction .....	4-1
4.2	Files .....	4-1
4.3	SCIOPTA Techniques and Concepts.....	4-2
4.4	SCIOPTA System Calls .....	4-3
4.5	Writing your Application .....	4-3

4.6	Error Handling .....	4-4
4.7	System Configuration and Initialization .....	4-4
4.8	General System Functions and Drivers .....	4-5
4.9	ARM Cortex Family System Functions .....	4-5
4.10	ARM Cortex CPU System Functions and Drivers .....	4-6
4.11	Board Functions and Drivers .....	4-6
4.12	Kernel Configuration .....	4-8
4.13	Kernel .....	4-8
4.14	Libraries and Include Files .....	4-8
4.15	Building the Project .....	4-9
4.15.1	Makefiles .....	4-9
4.15.2	Eclipse .....	4-9
4.15.3	iSYSTEM winIDEA .....	4-9
4.15.4	IAR Embedded Workbench for ARM .....	4-10
<b>5.</b>	<b>SCIOPTA Technology and Methods .....</b>	<b>5-1</b>
5.1	Introduction .....	5-1
5.2	Processes .....	5-2
5.2.1	Introduction .....	5-2
5.2.2	Process States .....	5-2
5.2.2.1	Running .....	5-2
5.2.2.2	Ready .....	5-2
5.2.2.3	Waiting .....	5-2
5.2.3	Process Categories .....	5-3
5.2.3.1	Static Processes .....	5-3
5.2.3.2	Dynamic Processes .....	5-3
5.2.4	Process Types .....	5-4
5.2.4.1	Prioritized Process .....	5-4
5.2.4.2	Interrupt Process .....	5-4
5.2.4.3	Timer Process .....	5-4
5.2.4.4	Init Process .....	5-5
5.2.4.5	Supervisor Process .....	5-5
5.2.4.6	Daemons .....	5-5
5.2.5	Priorities .....	5-6
5.2.5.1	Prioritized Processes .....	5-6
5.2.5.2	Interrupt Processes .....	5-6
5.2.5.3	Timer Processes .....	5-6
5.3	Messages .....	5-7
5.3.1	Introduction .....	5-7
5.3.2	Message Structure .....	5-7
5.3.3	Message Sizes .....	5-8
5.3.3.1	Example .....	5-8
5.3.4	Message Pool .....	5-8
5.3.5	Message Passing .....	5-9
5.4	Modules .....	5-10
5.4.1	SCIOPTA Module Friend Concept .....	5-10
5.4.2	System Module .....	5-10
5.4.3	Messages and Modules .....	5-11
5.4.4	System Protection .....	5-11
5.5	Trigger .....	5-12
5.6	Process Variables .....	5-13
5.7	Time Management .....	5-14

5.7.1	System Tick.....	5-14
5.7.2	Time-Out Server.....	5-14
5.8	SCIOPTA Scheduling.....	5-15
5.9	Distributed Systems .....	5-16
5.9.1	Introduction .....	5-16
5.9.2	CONNECTORS .....	5-16
5.9.3	Transparent Communication .....	5-17
5.10	Observation .....	5-18
5.11	Error Handling .....	5-19
5.11.1	General .....	5-19
5.11.2	The errno Variable .....	5-19
5.12	Hooks .....	5-20
5.12.1	Introduction .....	5-20
5.12.2	Error Hook .....	5-20
5.12.3	Message Hooks .....	5-20
5.12.4	Pool Hooks .....	5-20
5.12.5	Process Hooks .....	5-20
<b>6.</b>	<b>Application Programming Interface.....</b>	<b>6-1</b>
6.1	Introduction .....	6-1
6.2	System Calls List.....	6-1
6.3	Message System Calls.....	6-4
6.3.1	sc_msgAlloc.....	6-4
6.3.2	sc_msgAllocClr.....	6-7
6.3.3	sc_msgFree.....	6-8
6.3.4	sc_msgTx .....	6-10
6.3.5	sc_msgTxAlias .....	6-12
6.3.6	sc_msgRx .....	6-14
6.3.7	sc_msgAcquire.....	6-17
6.3.8	sc_msgAddrGet.....	6-18
6.3.9	sc_msgOwnerGet .....	6-19
6.3.10	sc_msgPoolIdGet .....	6-20
6.3.11	sc_msgSndGet.....	6-21
6.3.12	sc_msgSizeGet .....	6-22
6.3.13	sc_msgSizeSet.....	6-23
6.4	Pool System Calls .....	6-24
6.4.1	sc_poolCreate.....	6-24
6.4.2	sc_poolKill .....	6-25
6.4.3	sc_poolIdGet .....	6-26
6.4.4	sc_poolDefault .....	6-27
6.4.5	sc_poolInfo.....	6-28
6.4.6	sc_poolReset .....	6-29
6.5	Process Status System Calls.....	6-30
6.5.1	sc_procIdGet .....	6-30
6.5.1.1	sc_procIdGet in Interrupt Processes.....	6-31
6.5.2	sc_procPpidGet .....	6-32
6.5.3	sc_procNameGet.....	6-33
6.5.3.1	Returned Message .....	6-33
6.5.4	sc_procPathGet .....	6-34
6.5.4.1	Returned Message .....	6-34
6.5.5	sc_procPrioGet.....	6-35
6.5.6	sc_procPrioSet.....	6-36

6.5.7	sc_procSchedLock .....	6-37
6.5.8	sc_procSchedUnlock .....	6-38
6.5.9	sc_procSliceGet .....	6-39
6.5.10	sc_procSliceSet .....	6-40
6.5.11	sc_procStart .....	6-41
6.5.12	sc_procStop .....	6-42
6.5.13	sc_procVectorGet .....	6-43
6.5.14	sc_procYield .....	6-44
6.6	Process Create/Kill System Calls .....	6-45
6.6.1	sc_procPrioCreate .....	6-45
6.6.2	sc_procIntCreate .....	6-46
6.6.3	sc_procTimCreate .....	6-47
6.6.4	sc_procKill .....	6-48
6.7	Process Daemon System Calls .....	6-49
6.7.1	sc_procDaemonRegister .....	6-49
6.7.2	sc_procDaemonUnregister .....	6-50
6.8	Process Observation System Calls .....	6-51
6.8.1	sc_procObserve .....	6-51
6.8.2	sc_procUnobserve .....	6-52
6.9	Process Variables System Calls .....	6-53
6.9.1	sc_procVarInit .....	6-53
6.9.2	sc_procVarSet .....	6-54
6.9.3	sc_procVarGet .....	6-55
6.9.4	sc_procVarDel .....	6-56
6.9.5	sc_procVarRm .....	6-57
6.10	Module Status System Calls .....	6-58
6.10.1	sc_moduleIdGet .....	6-58
6.10.2	sc_moduleNameGet .....	6-59
6.10.3	sc_moduleInfo .....	6-60
6.11	Module Create/Kill System Calls .....	6-61
6.11.1	sc_moduleCreate .....	6-61
6.11.2	sc_moduleKill .....	6-63
6.12	Module Friendship System Calls .....	6-64
6.12.1	sc_moduleFriendAdd .....	6-64
6.12.2	sc_moduleFriendAll .....	6-65
6.12.3	sc_moduleFriendGet .....	6-66
6.12.4	sc_moduleFriendNone .....	6-67
6.12.5	sc_moduleFriendRm .....	6-68
6.13	Timing System Calls .....	6-69
6.13.1	sc_sleep .....	6-69
6.13.2	sc_tick .....	6-70
6.13.3	sc_tickGet .....	6-71
6.13.4	sc_tickLength .....	6-72
6.13.5	sc_tickMs2Tick .....	6-73
6.13.6	sc_tickTick2Ms .....	6-74
6.14	Time-out Server System Calls .....	6-75
6.14.1	sc_tmoAdd .....	6-75
6.14.2	sc_tmoRm .....	6-76
6.15	Trigger System Calls .....	6-77
6.15.1	sc_trigger .....	6-77
6.15.2	sc_triggerWait .....	6-78
6.15.3	sc_triggerValueGet .....	6-79

6.15.4	sc_triggerValueSet .....	6-80
6.16	CONNECTOR System Calls .....	6-81
6.16.1	sc_connectorRegister .....	6-81
6.16.2	sc_connectorUnregister .....	6-82
6.17	CRC System Calls .....	6-83
6.17.1	sc_miscCrc .....	6-83
6.17.2	sc_miscCrcContd .....	6-84
6.18	Error System Calls .....	6-85
6.18.1	sc_miscError .....	6-85
6.18.2	sc_miscErrnoGet .....	6-86
6.18.3	sc_miscErrnoSet .....	6-87
6.19	Hook Registering System Calls .....	6-88
6.19.1	sc_miscErrorHookRegister .....	6-88
6.19.2	sc_msgHookRegister .....	6-89
6.19.3	sc_poolHookRegister .....	6-90
6.19.4	sc_procHookRegister .....	6-91
<b>7.</b>	<b>Application Programming .....</b>	<b>7-1</b>
7.1	Introduction .....	7-1
7.2	System Partition .....	7-1
7.3	Modules .....	7-1
7.4	Resource Management .....	7-3
7.5	Processes .....	7-4
7.5.1	Introduction .....	7-4
7.5.2	Prioritized Processes .....	7-4
7.5.2.1	Process Declaration Syntax .....	7-5
7.5.2.2	Process Template .....	7-5
7.5.3	Interrupt Processes .....	7-6
7.5.3.1	Interrupt Process Declaration Syntax .....	7-7
7.5.3.2	Interrupt Process Template .....	7-8
7.5.4	Timer Process .....	7-9
7.5.4.1	Timer Process Declaration Syntax .....	7-9
7.5.4.2	Timer Process Template .....	7-9
7.5.5	Init Process .....	7-10
7.5.5.1	Init Process in Static Modules .....	7-10
7.5.5.2	Init Process in Dynamic Modules .....	7-11
7.5.6	Selecting Process Type .....	7-12
7.5.6.1	Prioritized Process .....	7-12
7.5.6.2	Interrupt Process .....	7-12
7.5.6.3	Timer Process .....	7-12
7.6	Addressing Processes .....	7-13
7.6.1	Introduction .....	7-13
7.6.2	Get Process IDs of Static Processes .....	7-13
7.6.3	Get Process IDs of Dynamic Processes .....	7-13
7.7	Interprocess Communication .....	7-14
7.7.1	Introduction .....	7-14
7.7.2	SCIOPTA Messages .....	7-14
7.7.2.1	Description .....	7-14
7.7.2.2	Message Declaration .....	7-14
7.7.2.3	Message Number .....	7-15
7.7.2.4	Message Structure .....	7-15
7.7.2.5	Message Union .....	7-16

7.7.2.6	Message Number (ID) organization .....	7-16
7.7.2.7	Example.....	7-17
7.7.3	SCIOPTA Trigger .....	7-18
7.7.3.1	Description .....	7-18
7.7.3.2	Example.....	7-18
7.8	SCIOPTA Memory Manager - Message Pools .....	7-19
7.8.1	Message Pool .....	7-19
7.8.2	Message Pool size .....	7-19
7.8.3	Message Sizes .....	7-20
7.8.4	Example.....	7-20
7.8.5	Message Administration Block .....	7-20
7.9	SCIOPTA Daemons .....	7-21
7.9.1	Process Daemon .....	7-21
7.9.2	Kernel Daemon .....	7-22
7.10	Trap Interface .....	7-23
7.11	Error Hook .....	7-24
7.11.1	Introduction .....	7-24
7.11.2	Error Information .....	7-24
7.11.3	Error Hook Registering .....	7-25
7.11.4	Error Hook Declaration Syntax.....	7-25
7.11.5	Example.....	7-26
7.11.6	Error Hooks Return Behaviour .....	7-27
7.12	SCIOPTA ARM Cortex Exception Handling .....	7-28
7.12.1	Introduction .....	7-28
7.12.2	Interrupt Handler.....	7-28
7.12.3	SCIOPTA Interrupt Process .....	7-28
7.13	SCIOPTA Design Rules.....	7-29
<b>8.</b>	<b>System and Application Configuration .....</b>	<b>8-1</b>
8.1	Introduction .....	8-1
8.2	System Start .....	8-1
8.2.1	Start Sequence.....	8-1
8.2.2	Reset Hook .....	8-2
8.2.3	C Startup.....	8-3
8.2.4	Start Hook .....	8-3
8.2.5	INIT Process.....	8-3
8.2.6	Module Hook .....	8-4
8.2.6.1	System Module Hook.....	8-4
8.2.6.2	User Modules Hooks.....	8-4
<b>9.</b>	<b>Board Support Packages.....</b>	<b>9-1</b>
9.1	Introduction .....	9-1
9.2	General System Functions.....	9-1
9.3	ARM Cortex Family System Functions .....	9-1
9.4	STM32 System Functions and Drivers .....	9-3
9.4.1	General STM32 Functions and Definitions .....	9-3
9.4.2	STM32 System Tick Driver .....	9-4
9.4.2.1	STM32 System Tick Interrupt Process .....	9-4
9.4.3	STM32 UART Functions .....	9-4
9.5	Stellaris System Functions and Drivers .....	9-5
9.5.1	General Stellaris Functions and Definitions .....	9-5



9.5.2	Stellaris System Tick Driver .....	9-6
9.5.2.1	Stellaris System Tick Interrupt Process .....	9-6
9.6	Olimex STM32-P103 Board .....	9-7
9.6.1	Description .....	9-7
9.6.2	STM32-P103 General Board Functions and Definitions .....	9-9
9.6.3	STM32-P103 LED Driver .....	9-9
9.7	STMicroelectronics STM3210E-EVAL Evaluation Board .....	9-10
9.7.1	Description .....	9-10
9.7.2	STM3210E-EVAL General Board Functions and Definitions .....	9-12
9.7.3	STM3210E-EVAL LED Driver .....	9-12
9.8	Luminary LM3S6965 Board .....	9-13
9.8.1	Description .....	9-13
9.8.2	LM3S6965 General Board Functions and Definitions .....	9-15
9.8.3	LM3S6965 LED Driver .....	9-15
9.9	Luminary® Stellaris Family Driver Library .....	9-16
<b>10.</b>	<b>Kernel Configuration .....</b>	<b>10-1</b>
10.1	Introduction .....	10-1
10.2	System .....	10-1
10.3	Modules .....	10-2
10.3.1	Small Systems .....	10-2
10.3.2	Multi-Module Systems .....	10-3
10.4	Processes .....	10-4
10.4.1	INIT Process .....	10-4
10.4.2	Interrupt Processes .....	10-5
10.4.3	Timer Processes .....	10-6
10.4.4	Prioritized Processes .....	10-7
10.4.5	Message Pools .....	10-8
10.5	Generating the Configuration Files .....	10-9
10.5.1	Generated Kernel Configuration Files .....	10-9
10.5.2	sciopta.cnf .....	10-10
10.5.3	sconf.h .....	10-10
10.5.4	sconf.c .....	10-10
<b>11.</b>	<b>Libraries and Include Files .....</b>	<b>11-1</b>
11.1	Kernel .....	11-1
11.2	Kernel Libraries .....	11-1
11.2.1	GNU GCC Kernel Libraries .....	11-1
11.2.1.1	Optimization “X” .....	11-1
11.2.1.2	“t” .....	11-1
11.2.1.3	Included SCIOPTA Kernel Libraries .....	11-2
11.2.1.4	Building Kernel Libraries for GCC .....	11-2
11.2.2	IAR Kernel Libraries .....	11-3
11.2.2.1	Optimization “X” .....	11-3
11.2.2.2	“t” .....	11-3
11.2.2.3	Included SCIOPTA Kernel Libraries .....	11-3
11.2.2.4	Building Kernel Libraries for IAR .....	11-4
11.2.3	ARM RealView Kernel Libraries .....	11-5
11.2.3.1	Optimization “X” .....	11-5
11.2.3.2	“t” .....	11-5
11.2.3.3	Included SCIOPTA Kernel Libraries .....	11-5

11.2.3.4	Building Kernel Libraries for ARM RealView.....	11-6
11.3	Luminary® Stellaris Family Driver Libraries.....	11-7
11.3.1	GNU GCC Stellaris Family Driver Libraries.....	11-7
11.3.2	IAR Stellaris Family Driver Libraries.....	11-7
11.3.3	ARM RealView Stellaris Family Driver Libraries .....	11-7
11.4	Include Files .....	11-8
11.4.1	Include Files Search Directories .....	11-8
11.4.2	Main Include File sciopta.h.....	11-8
11.4.3	Configuration Definitions sconf.h.....	11-8
11.4.4	Main Data Types types.h.....	11-9
11.4.5	ARM Data Types types.h.....	11-9
11.4.6	Global System Definitions defines.h.....	11-9
<b>12.</b>	<b>Linker Scripts and Memory Map .....</b>	<b>12-1</b>
12.1	Introduction .....	12-1
12.2	GCC Linker Script .....	12-1
12.2.1	Memory Regions.....	12-1
12.2.2	Module Sizes.....	12-2
12.2.3	Specific Module Values .....	12-3
12.3	IAR Embedded Workbench Linker Script.....	12-4
12.4	ARM RealView Linker Script .....	12-4
12.5	Data Memory Map .....	12-5
<b>13.</b>	<b>System Building .....</b>	<b>13-1</b>
13.1	Introduction .....	13-1
13.2	Makefile and GNU GCC.....	13-1
13.2.1	Tools.....	13-1
13.2.2	Environment Variables.....	13-1
13.2.3	Makefile Location .....	13-2
13.2.4	Project Settings.....	13-2
13.3	Eclipse IDE and GNU GCC.....	13-3
13.3.1	Introduction .....	13-3
13.3.2	Tools.....	13-3
13.3.3	Environment Variables.....	13-3
13.3.4	Project Settings and Building.....	13-4
13.4	iSYSTEM© winIDEA .....	13-5
13.4.1	Introduction .....	13-5
13.4.2	Tools.....	13-5
13.4.3	Environment Variables.....	13-5
13.4.4	winIDEA Project Files Location.....	13-6
13.4.5	Project Settings.....	13-6
13.5	IAR Embedded Workbench for ARM .....	13-7
13.5.1	Introduction .....	13-7
13.5.2	Tools.....	13-7
13.5.3	Environment Variables.....	13-7
13.5.4	IAR EW Project Files Location .....	13-8
13.5.5	Project Settings.....	13-8
<b>14.</b>	<b>SCIOPTA Debugging .....</b>	<b>14-1</b>
14.1	Introduction .....	14-1
14.2	Kernel Awareness for Third-Party Debuggers.....	14-2

14.2.1	Kernel Awareness for Lauterbach Trace32.....	14-2
<b>15.</b>	<b>Kernel Error Codes .....</b>	<b>15-1</b>
15.1	Introduction .....	15-1
15.2	Include Files .....	15-1
15.3	Function Codes.....	15-2
15.4	Error Codes .....	15-4
15.5	Error Types.....	15-5
<b>16.</b>	<b>SCONF Kernel Configuration Utility.....</b>	<b>16-1</b>
16.1	Introduction .....	16-1
16.2	Starting SCONF .....	16-1
16.3	Preference File sc_config.cfg.....	16-2
16.4	Project File .....	16-2
16.5	SCONF Windows.....	16-3
16.5.1	Parameter Window .....	16-3
16.5.2	Browser Window .....	16-4
16.6	Creating a New Project .....	16-5
16.7	Configure the Project .....	16-5
16.8	Creating Systems.....	16-6
16.9	Configuring ARM Cortex Target Systems .....	16-8
16.9.1	General Configuration.....	16-8
16.9.2	Configuring Hooks.....	16-10
16.9.3	Debug Configuration.....	16-11
16.10	Creating Modules .....	16-12
16.11	Configuring Modules .....	16-13
16.12	Creating Processes and Pools.....	16-16
16.13	Configuring the Init Process.....	16-17
16.14	Interrupt Process Configuration .....	16-18
16.15	Timer Process Configuration .....	16-20
16.16	Prioritized Process Configuration .....	16-22
16.17	Pool Configuration .....	16-24
16.18	Build.....	16-26
16.18.1	Build System .....	16-26
16.18.2	Change Build Directory .....	16-27
16.18.3	Build All.....	16-28
16.19	Command Line Version .....	16-29
16.19.1	Introduction .....	16-29
16.19.2	Syntax.....	16-29
<b>17.</b>	<b>Manual Versions .....</b>	<b>17-1</b>
17.1	Manual Version 1.1 .....	17-1
17.2	Manual Version 1.0.....	17-2
<b>18.</b>	<b>Index .....</b>	<b>18-1</b>

## 1 Introduction

### 1.1 SCIOPTA ARM Cortex Real-Time Operating System

**SCIOPTA ARM Cortex** is a high performance real-time operating system for microcontrollers using the ARM Cortex cores ARM Cortex-A8, ARM Cortex-A9 MPCore, ARM Cortex-A9 Single Core Processor, ARM Cortex-M1, ARM Cortex-M3 and ARM Cortex-R4(F) and other derivatives of the ARM Cortex family. Including:

**Luminary Micro Stellaris®**

LM3Sxxxx and all other ARM Cortex based microcontrollers.

**STMicroelectronics STM32**

STM32F101, STM32F102, STM32F103 and all other ARM Cortex based microcontrollers.

Including all microcontrollers from other suppliers which have ARM Cortex cores.

The full featured operating system environment includes:

- KRN - Pre-emptive Multi-Tasking Real-Time Kernel
- BSP - Board Support Packages
- IPS - Internet Protocols (TCP/IP)
- IPS Applications - Internet Protocols Applications (Web Server, TFTP, DNS, DHCP, Telnet, SMTP etc.)
- SFATFS - FAT File System
- SFFS - FLASH File System, NOR
- SFFSN - FLASH File system, NAND support
- USBD - Universal Serial Bus, Device
- USBH - Universal Serial Bus, Host
- DRUID - System Level Debugger
- SCIOPTA PEG - Portable Embedded GUI
- CONNECTOR - Support for Distributed Multi-CPU Systems
- SMMS - Support for MMU
- SCAPI - SCIOPTA API on Windows or LINUX host
- SCSIM - SCIOPTA Simulator

# 1 Introduction

---

## 1.2 About This Manual

The purpose of this **SCIOPTA ARM Cortex - Kernel, User's Guide** is to give all needed information how to use the SCIOPTA ARM Cortex real-time kernel in an embedded project and includes a complete description of all system calls and error messages.

After a short introduction into real-time operating systems and a description of the installation procedure, detailed information about the technologies and methods used in the SCIOPTA ARM Cortex Kernel are given. Detailed descriptions of Getting Started examples will allow a fast and smooth introduction into SCIOPTA.

Furthermore you will find useful information about system design and configuration. Also target specific information such as an overview of the system building procedures and a description of the board support packages (BSP) can be found in the manual.

### 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.

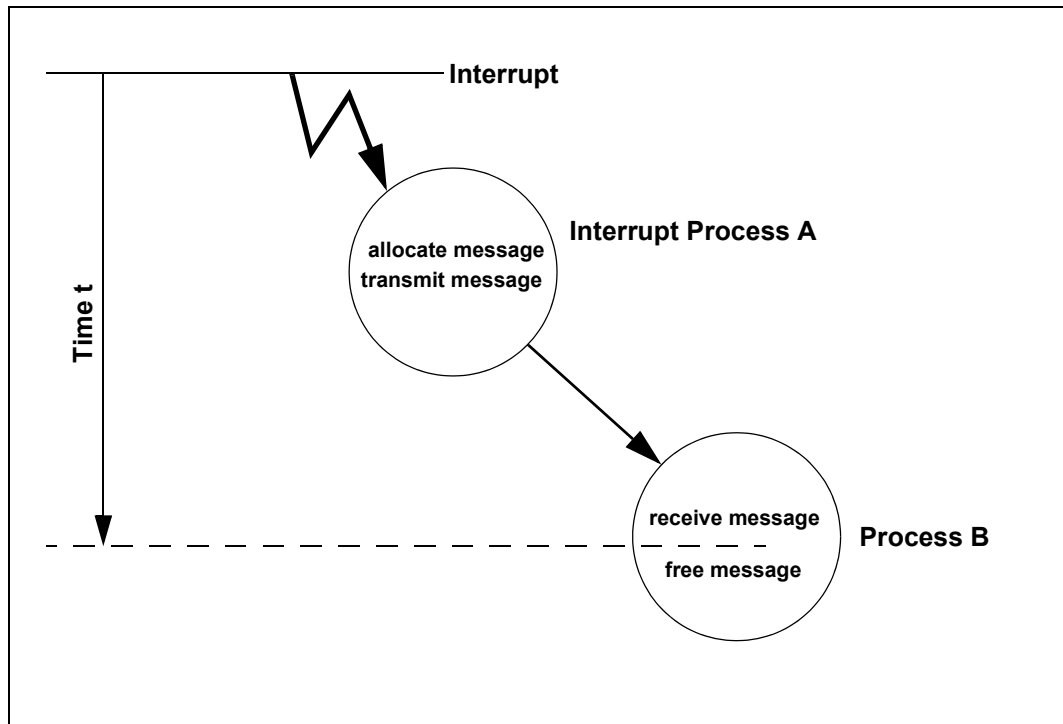


Figure 1-1: Real-Time System Definition

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 Introduction

---

## 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 Installation

### 2.1 Introduction

This chapter describes how to install SCIOPTA ARM Cortex. Topics such as system requirements, installation procedure and uninstallation are covered herein.

### 2.2 The SCIOPTA ARM Cortex Delivery

Before you start the installation please check the SCIOPTA ARM Cortex delivery. The following items should be included:

- CD-ROM containing SCIOPTA ARM Cortex.
- Installation password.
- Manuals of your installed products:
  - SCIOPTA ARM Cortex - Kernel, User's Guide (this document).
  - SCIOPTA ARM Cortex - DRUID, User's Guide.
  - SCIOPTA ARM Cortex - IPS Internet Protocols, User's Guide.
  - SCIOPTA ARM Cortex - IPS Internet Protocols Applications, User's Guide.
  - SCIOPTA ARM Cortex - FAT File System, User's Guide.
  - SCIOPTA ARM Cortex - FLASH Safe File System, User's Guide.
  - SCIOPTA ARM Cortex - USB Device, User's Guide.
  - SCIOPTA ARM Cortex - USB Host, User's Guide.
  - SCIOPTA ARM Cortex - PEG+, User's Guide.
  - SCIOPTA ARM Cortex - SMMS Memory Protection, User's Guide.
  - SCIOPTA ARM Cortex - CONNECTOR, User's Guide.

### 2.3 System Requirements

#### 2.3.1 Windows

Personal Computer or Workstation with:

- Intel® Pentium® processor
- Microsoft® Windows XP Professional
- 64 MB of RAM
- 20 MB of available hard disk space

#### 2.3.2 Linux

- Linux® 2.2 kernel on X86 computer
- 64 MB of RAM
- 20 MB of available hard disk space



### 2.4 Installation Procedure Windows Hosts

#### 2.4.1 Main Installation Window

SCIOPTA is using a sophisticated software product delivery system which allows to supply you with a customized and customer specific delivery. You will find a customer number and the name of the licensee on the CD and the installation window.

Insert the CD-ROM into an available CD drive. This should autostart the SCIOPTA installation. If autostart does not execute you can manually start the installation by double clicking the file setup.exe on the CD.

The following main installation window will appear on your screen:

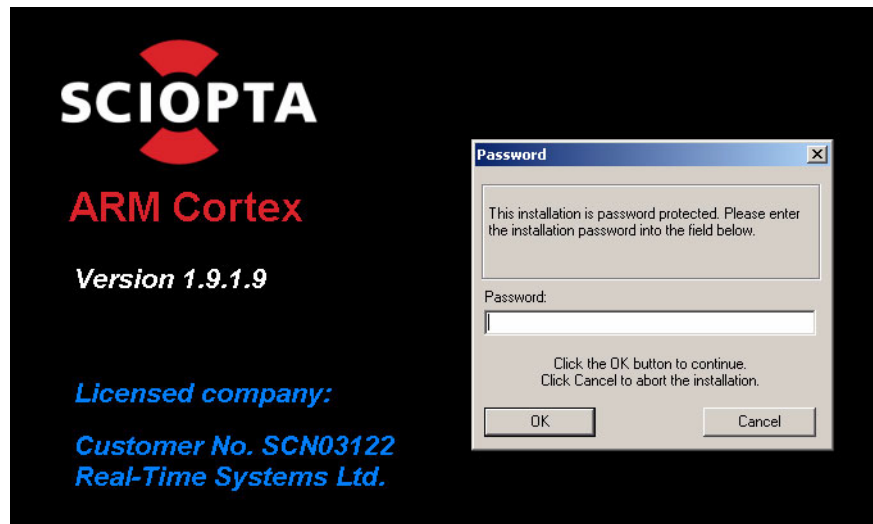


Figure 2-1: Main Installation Window

To install SCIOPTA ARM Cortex you must enter a password which was delivered by email or on paper.

The program will guide you through the installation process.

### 2.4.2 Product Versions

Each SCIOPTA product release is identified by a version number consisting of a four field compound number of the format:

**“X.Y.Z.F”**

The first digit, **X**, is used for the major release number which will identify a major increase in the product functionality and involves usually a total rewrite or redesigning of the product including changes in the SCIOPTA kernel API. This number starts at 1.

The second digit, **Y**, is used for a release number which is used to identify important enhancements. This number is incremented to indicate new functionality in the product and may include changes in function calls without modifications in the SCIOPTA kernel API. This number starts at 0.

The third digit, **Z**, stands for feature release number. The feature release number is iterated to identify when functionality have been increased and new files, board support packages or CPUs have been added. This requires also changes in the documentation. This number starts at 0.

The fourth digit, **F**, is called the build number and changes if modifications on the examples or board support packages have been made. This number starts at 0.

### 2.4.3 Installation Location

The SCIOPTA products will be installed at the following location:

**<Destination Folder>\sciopta\<version>\** (in this manual also referred as **<install\_folder>\sciopta\<version>\**)

The expression **<version>** stands for the SCIOPTA four digit version number (e.g. 1.7.2.1)

If you are not modifying the Destination Folder SCIOPTA ARM Cortex will be installed at: **c:\sciopta\<version>\**

Please make sure that all SCIOPTA ARM Cortex products of one version are installed in the same destination folder.

### 2.4.4 Release Notes

This SCIOPTA ARM Cortex – Kernel delivery includes a text file named **RN\_ARM CX\_KRN.txt** which contains a description of the changes of the actual version compared to the last delivered version. It allows you to decide if you want to install and use the new version.

You will also find a file **revisions.txt** which contains a list of all installed files including the following information for each file: file name, document number, file version and file description.

### 2.4.5 Short Cuts

The program will install the **sconf** short-cut (to run the SCIOPTA configuration program **sconf.exe**) in the folder **Sciopta** under the Windows Programs Menu.

If you are also installing the **DRUID System Debugger** the **druid** short-cut (to run **druid.exe**) and the **druid server** short-cut (to run **druids.exe**) will be installed in the folder **Sciopta** under the Windows Programs Menu and on the desktop.

## 2 Installation

### 2.4.6 SCIOPTA\_HOME Environment Variable

The SCIOPTA system building process needs the SCIOPTA\_HOME environment variable to be defined.

Please define the SCIOPTA\_HOME environment variable and set it to the following value:

`<install_folder>\sciopta\<version>`

The expression <version> stands for the SCIOPTA four digit version number.

**Example:** c:\sciopta\1.7.2.1

### 2.4.7 Setting SCIOPTA Path Environment Variable

If you are using makefiles to build your system, the SCIOPTA delivery includes the GNU Make utility. The following file are installed in the SCIOPTA bin\win32 directory:

- gnu-make.exe
- rm.exe
- rm.exe
- libiconv2.dll
- libintl3.dll

Please include

`<install_folder>\sciopta\<version>\bin\win32`

in your **path** environment variable.

### 2.4.8 Uninstalling SCIOPTA ARM Cortex

Each SCIOPTA ARM Cortex product is listed separately on the “currently installed programs” list in the “Add or Remove Programs” window of the Windows® “Control Panel”.

For each SCIOPTA product use the following procedure:

1. From the Windows **start** menu select **settings -> control panel -> add/remove programs**.
2. Choose the SCIOPTA product which you want to remove from the list of programs.
3. Click on the **Change/Remove** button.
4. Select the **Automatic** button and click **Next>**.
5. Click on the **Finish** button in the next windows.
6. Windows will now automatically uninstall the selected SCIOPTA product.

#### Please Note:

There might be (empty) directories which are not removed from the system. If you want you can remove it manually.

### 2.4.9 GNU Tool Chain Installation

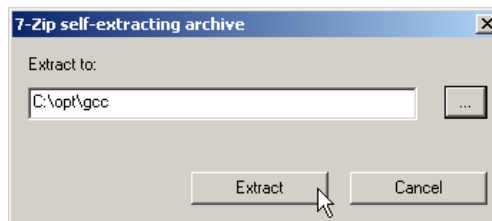
SCIOPTA ARM Cortex is supporting the GNU Tool Chain. The Sourcery G++ Lite Edition from CodeSourcery is directly supported. The Lite Edition contains only command-line tools and is available at no cost. A ready to install version is available from SCIOPTA.

The Sourcery G++ Lite Edition GNU Tool Chain Package for SCIOPTA delivery consists of:

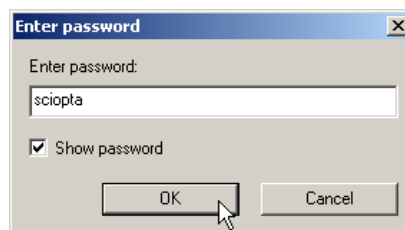
- GNU C & C++ Compilers for ARM
- GNU Assembler and Linker
- GNU C & C++ Runtime Libraries

Run the installer file **arm-2008q1\_cs\_sciopta.exe** which can be found on the SCIOPTA ARM CD.

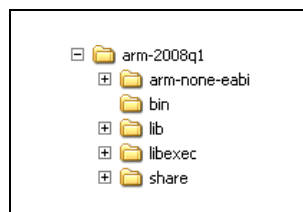
The installer does just unpack the compiler into your selected folder. No settings into Windows Registry will be done. Enter a suitable folder into the “Extract to:” line. You can also use the browse button.



The installer asks for password. Please enter “sciopta”.



The compiler structure will be installed in a **arm-2008q1** folder under your above selected unpacking folder. You can also copy the compiler structure at any suitable place.



**Define above compiler “bin” directory for compiler call in your project IDE and/or include this directory in the path environment variable.**

## 2 Installation

### 2.4.10 Luminary Stellaris Cortex-M3 Family Driver Library

For the Luminary Stellaris Cortex-M3 Family we are using the family driver library which is available from the Luminary WebSite.

Please download the driver package from here: <http://www.luminarymicro.com>

Install the driver library at a suitable place.

The **SCIOPTA** system building process needs a **LMI\_DRIVER** environment variable to be defined.

Please define the **LMI\_DRIVER** environment variable and set it to the folder where the driver library was installed (e.g. C:\DriverLib).

### 2.4.11 STMicroelectronics STM32 Firmware Library

For the STMicroelectronics STM32 Family we are using the family driver library which is available from the STMicroelectronics WebSite.

Please download the driver package **STM32F10xFWLib (version 2.0.3)** from here: <http://www.st.com>

Install the driver library at a suitable place.

The **SCIOPTA** system building process needs a **STM32\_FWLIB** environment variable to be defined.

Please define the **STM32\_FWLIB** environment variable and set it to the folder where the **inc** and **src** of the STM32 Firmware Library reside (e.g. <STM32LIB\_Install\_folder>\FWLib\library).

To avoid warnings during the build process you should uncomment the **U8\_MAX**, **U16\_MAX** and **U32\_MAX** defines in the file **\inc\stm32f10x\_type.h**

Example:

```
#ifndef U8_MAX /* SCIOPTA defines these */
#define U8_MAX ((u8)255)
#endif
#define S8_MAX ((s8)127)
#define S8_MIN ((s8)-128)
#ifndef U16_MAX /* SCIOPTA defines these */
#define U16_MAX ((u16)65535u)
#endif
#define S16_MAX ((s16)32767)
#define S16_MIN ((s16)-32768)
#ifndef U32_MAX /* SCIOPTA defines these */
#define U32_MAX ((u32)4294967295uL)
#endif
#define S32_MAX ((s32)2147483647)
#define S32_MIN ((s32)-2147483648)
```

### 2.4.12 Eclipse C/C++ Development Tooling - CDT

The CDT (C/C++ Development Tools) Project provides a fully functional C and C++ Integrated Development Environment (IDE) for the Eclipse platform.

Please consult <http://www.eclipse.org/> for more information about Eclipse.

Please consult <http://www.eclipse.org/cdt> for more information about Eclipse CDT (C/C++ Development Tools) and to download the latest version.

Actually SCIOPTA supports Eclipse Platform Version 3.4.1 (Ganymede) with Eclipse C/C++ Development Tools (CDT) Version 5.0.1. This combined platform can be downloaded from above eclipse/cdt website or installed from the SCIOPTA Cortex CD (eclipse-cpp-ganymede-SR1-win32.zip). To install it, just unzip this file in a folder of your choice and run eclipse.exe. Later versions might also work.

There are two Eclipse CDT Plug-Ins available for SCIOPTA Cortex:

- com.sciopta.ui\_x.x.x.jar
- com.sciopta.gnu.cortex\_x.x.x.jar

The “x.x.x” represent the actual version. Both files can be found on the SCIOPTA Cortex CD. To install the plug-ins, exit Eclipse and copy both files into the “plugins” folder of the Eclipse CDT installation. The plug-ins are now available for the next eclipse session. To check the correct installation open the “About Eclipse Platform Plug-Ins” window (menu: **Help** -> **About Eclipse Platform** and click on the **Plug-In Details** button).

The SCIOPTA getting started examples are based on the Eclipse CDT with SCIOPTA Plug-Ins.

The Eclipse IDE requires that a Java Run-Time Environment (JRE) be installed on your machine to run. Please consult the Eclipse Web Site to check if your JRE supports your Eclipse environment. JRE can be downloaded from the SUN or IBM Web Sites.

## 3 Getting Started

### 3.1 Introduction

These are small tutorial examples which gives you a good introduction into typical SCIOPTA systems and products. They can be used as a starting point for more complex applications and your real projects.

#### Please Note:

The getting-started examples are using specific integrated development environments, build utilities, compilers, cpus and boards. If you are using another environment you may need to include other files from the delivery or modify the used files. Usually the following files are concerned: project file (system.c), linker script (<board\_name.ld for GNU), BSP assembler files (led.S, resethook.S), BSP C files (druid\_uart.c, fec.c, serial.c, simple\_uart.c, systick.c) and C startup file (cstartup.S).

### 3.2 Getting Started - Hello Example

#### 3.2.1 Description

The Getting Started System for the SCIOPTA Kernel consists of two SCIOPTA processes exchanging standard SCIOPTA messages.

Process **hello** sends four messages (**STRING\_MSG\_ID**) containing a character string to process **display**. For each transmitted message, process **hello** waits for an acknowledge message (**ACK\_MSG\_ID**) from process **display** before the next string message is sent.

After all four messages have been sent process **hello** sleeps for a while and restarts the whole cycle again for ever.

Each message is received, displayed and freed by process **display**. Process **display** sends back an acknowledge message (**ACK\_MSG\_ID**) for every received message.

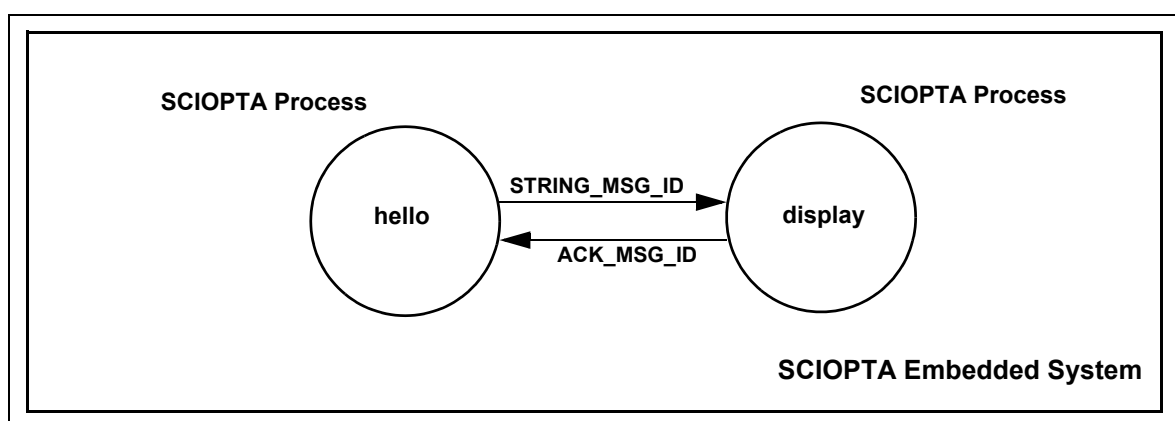


Figure 3-1: Getting Started Kernel Example

### 3.2.2 Eclipse IDE and GNU GCC

#### 3.2.2.1 Equipment

The following equipment is used to run this getting started example:

- Personal Computer or Workstation with: Intel® Pentium® processor, Microsoft® Windows XP, 256 MB of RAM and 200 MB of available hard disk space.
- CodeSourcery GNU C & C++ Sourcery G++ Lite Edition for ARM Version Q1 2008. This package can be found on the SCIOPTA Cortex CD.
- A debugger/emulator for Cortex which supports GNU GCC such as the iSYSTEM winIDEA Emulator/Debugger for Cortex or the Lauterbach TRACE32 debugger for Cortex.
- Target board which is supported by SCIOPTA examples. For each supported board there is a directory in the example folder: <install\_folder>\sciopta\<version>\exp\krm\arm\hello\.
- SCIOPTA Cortex - Kernel.
- C/C++ Eclipse Platform Version 3.4.1 (Ganymede) with Eclipse C/C++ Development Tools (CDT) Version 5.0.1.
- SCIOPTA Cortex Eclipse CDT Plug-Ins (com.sciopta.ui\_1.0.0.jar and com.sciopta.gnu.cortex\_1.0.0.jar).
- In order to run the Eclipse Platform you also need the Sun Java 2 SDK, Standard Edition for Microsoft Windows.
- If you are using a Luminary Stellaris Cortex-M3 Family CPU you will need the Luminary Stellaris Cortex-M3 Family Driver Library.
- If you are using a STMicroelectronics STM32 Family CPU you will need the STMicroelectronics STM32 Firmware Library.
- This getting started example is sending some specific example messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board.
  - For the **STM32-P103** board RS232\_2 is used.
  - For the **STM3210E-EVAL** board USART\_2 is used.
  - For the **LM3S6965** evaluation board USB/Serial is used.

Please consult files system.c, simple\_uart.c, simple\_uart.h and config.h for selecting another UART or for another board.



### 3.2.2.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA\_HOME is defined as described in chapter [2.4.6 “SCIOPTA\\_HOME Environment Variable” on page 2-4](#).
2. For the Luminary Stellaris Cortex-M3 Family, check that the environment variable LMI\_DRIVER is defined as described in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#).
3. For the STMicroelectronics STM32 Family, check that the environment variable STM32\_FWLIB is defined as described in chapter [2.4.11 “STMicroelectronics STM32 Firmware Library” on page 2-6](#).
4. Be sure that the CodeSourcery GNU C & C++ compiler bin directory is included in the PATH environment variable as described in chapter [2.4.9 “GNU Tool Chain Installation” on page 2-5](#).
5. Be sure that the SCIOPTA \win32\bin directory is included in the PATH environment variable as described in chapter [2.4.7 “Setting SCIOPTA Path Environment Variable” on page 2-4](#).
6. Create a project folder to hold all project files (e.g. d:\myprojects\sciopta) if you have not already done it for other getting-started projects.
7. Launch Eclipse. The Workspace Launcher window opens.
8. Select your created project folder (e.g. c:\myproject\sciopta) as your workspace (by using the Browse button).
9. Click the **OK** button. The workbench windows opens.
10. Deselect **“Build Automatically”** in the **Project** menu.
11. Click on the **Workbench - Go to the Workbench** button (on the right side).
12. Maximize the workbench.
13. Open the **New Project** window (menu: **File -> New -> C Project**).
14. The C Project window opens. Enter the project name: **krn\_hello**.
15. Open the folder for your board in the project type window and select **Empty Project**.
16. Click on the **Finish** button.
17. The next steps we will executed outside Eclipse.
18. Copy the script **copy\_files.bat** from the example directory for your selected target board:  
`<install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\`  
 to your project folder.
19. Open a command window (windows cmd.exe) and go to your project folder.
20. Type copy\_files to execute the **copy\_files.bat** batch file. All needed project files will be copied from the SCIOPTA delivery to your project folder.
21. Close the command window and return to Eclipse.
22. Run the SCIOPTA configuration utility (menu: **SCIOPTA -> SCONF** or click the SCIOPTA button in the upper button bar).
23. Ignore a possible error message by clicking the **OK** button in the **critical error** window.
24. The SCIOPTA configuration tool (**SCONF**) opens. Execute the following steps inside **SCONF**.
25. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF** (menu: **File > Open**).
26. Click on the **Build All** button or press **Ctrl-B** to build the kernel configuration files.  
 The files sciopta.cnf, sconf.c and sconf.h. will be created in your project folder.

### 3 Getting Started

---

27. Close the **SCONF** configuration utility.
28. Swap back to the Eclipse workbench. Make sure that the kernel hello project (**krn\_hello**) is highlighted and open the project (menu: **Project > Open Project**).
29. Expand the project by selecting the **[+]** button and make sure that the **krn\_hello** project is highlighted.
30. Type the F5 key (or menu: **File > Refresh**) to refresh the project.
31. Now you can see all files in the Eclipse Navigator window.
32. Select the Console window at the bottom of the Eclipse workbench to see the project building output.
33. Be sure that the project (**krn\_hello**) is high-lighted and build the project (menu: **Project > Build Project** or **Build** button).
34. The executable (**krn\_hello.elf**) will be created in the debug folder of the project.
35. Launch your Cortex source-level emulator/debugger and load the resulting sciopta.elf.
36. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. The speed must be set to 115200 baud.
37. For some emulators/debuggers specific project and board initialization files can be found in the created project folder or in other example directories.
38. Now you can start the system and check the log messages on your host terminal window.
39. You can also set breakpoints anywhere in the example system and watch the behaviour.

## 3 Getting Started

---

### 3.2.3 Other Environments

#### 3.2.3.1 Makefile and GNU GCC

The GNU make utility (gnu-make.exe) is included in the SCIOPTA delivery:

```
<install_folder>\sciopta\<version>\bin\win32\
```

You can use the same **copy\_files.bat** file as for the eclipse environment from the example directory for your selected target board:

```
<install_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\
```

This copies all files into your example folder including the makefile. The makefile is written for all supported boards. You need to give the board as a parameter when calling the makefile as follows:

**gnu-make BOARD\_SEL=xx** (where xx defines the board)

Please consult the makefile for more information.

#### 3.2.3.2 iSYSTEM winIDEA and GNU GCC

For some boards ready to use project files for iSYSTEM winIDEA are included.

You can use the same **copy\_files.bat** file as for the eclipse environment from the example directory for your selected target board:

```
<install_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\
```

This copies all files into your example folder including the needed iSYSTEM winIDEA project files (iC3000.xqrf, iC3000.xjrf, <board>.ini and winIDEA\_gnu.ind).

#### 3.2.3.3 IAR Embedded Workbench

For some boards ready to use project files for IAR Systems Embedded Workbench are included.

There is a specific **copy\_files\_iar.bat** file from the example directory for your selected target board available:

```
<install_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\
```

This copies all files into your example folder including the needed IAR Embedded Workbench project files (<board>.mac, <board>.xcl, <board>.eww and <board>.ewd).

## 4 SCIOPTA Kernel Project Overview

### 4.1 Introduction

This is an introduction into a typical SCIOPTA ARM Cortex kernel application. It follows a chronological order and for each step lists and describes the files needed from the SCIOPTA ARM Cortex delivery.

Only kernel applications are covered. If you are using other SCIOPTA real-time products such as networking software, file systems, USB software, CONNECTOR packages for supporting distributed systems, memory management unit support software etc., please read the user manuals which are included in the delivery of these products for information how to use it.

### 4.2 Files

List of typical files needed from the SCIOPTA distribution for a small example (e.g. kernel hello).

File	Description	Compiler	Link
display.c	Example	All	<a href="#">4.5 “Writing your Application” on page 4-3</a>
hello.c	Example	All	
hello.msg	Example message definition	All	
error.c	Error hook	All	<a href="#">4.6 “Error Handling” on page 4-4</a>
system.c	System module functions	All	<a href="#">4.7 “System Configuration and Initialization” on page 4-4</a>
map.c	Module mapping	IAR EW	
winIDEA_gnu.ind	winIDEA indirection file	GNU	<a href="#">4.8 “General System Functions and Drivers” on page 4-5</a>  <a href="#">4.9 “ARM Cortex Family System Functions” on page 4-5</a>
module.ld	Modules linker script	GNU	
cortexm3_cstartup.S	C Startup file	GNU	
cortexm3_cstartup.s	C Startup file	ARM RealView	
cortexm3_exception.S	Exception handler	GNU	
cortexm3_exception.s	Exception handler	ARM RealView	
cortexm3_exception.s	Exception handler	IAR EW V4	
cortexm3_exception.s79	Exception handler	IAR EW V5	
cortexm3_vector.S	Vector table	GNU	
cortexm3_vector.s	Vector table	ARM RealView	
cortexm3_vector.s	Vector table	IAR EW V4	
cortexm3_vector.s79	Vector table	IAR EW V5	
systick.c	System tick interrupt process	All	<a href="#">4.10 “ARM Cortex CPU System Functions and Drivers” on page 4-6</a>
simple_uart.c	UART routines	All	

## 4 SCIOPTA Kernel Project Overview



File	Description	Compiler	Link
led.c	Board led function	All	<a href="#">4.11 “Board Functions and Drivers” on page 4-6</a>
boardSetup.c	Board setup	All	
<board>.ld	Main linker script	GNU	
<board>.xcl	Linker script	IAR EW	
<board>.sct	Linker script	ARM RealView	
config.h	Board settings	All	
<board>.ini	winIDEA board initialization	GNU	
<board>.mac	IAR CSpy Board initialization	IAR EW	
<board>.cmm	Trace32 board initialization	All	
hello.xml	SCIOPTA configuration file	All	<a href="#">4.12 “Kernel Configuration” on page 4-8</a>
sciopta.S	Kernel source	GNU	<a href="#">4.13 “Kernel” on page 4-8</a>
sciopta.s	Kernel source	ARM RealView	
sciopta.s	Kernel source	IAR EW	
Makefile	Example makefile	GNU	<a href="#">4.15.1 “Makefiles” on page 4-9</a>
board.mk	Board makefile	GNU	
iC3000.xjrf	winIDEA project file	GNU	<a href="#">4.15.3 “iSYSTEM winIDEA” on page 4-9</a>
iC3000.xqrf	winIDEA project file	GNU	
<board>.ewd	Project file	IAR EW	<a href="#">4.15.4 “IAR Embedded Workbench for ARM” on page 4-10</a>
<board>.eww	Project file	IAR EW	

### 4.3 SCIOPTA Techniques and Concepts

Before you can start writing any embedded applications using SCIOPTA you need become familiar with the concepts and techniques used by SCIOPTA.

Please read chapter [5 “SCIOPTA Technology and Methods” on page 5-1](#) to get an introduction into the SCIOPTA technology.

In a new project you have first to determine the specification of the system. As you are designing a real-time system, speed requirements needs to be considered carefully including worst case scenarios. Defining function blocks, environment and interface modules will be another important part for system specification.

Systems design includes defining the modules, processes and messages. SCIOPTA is a message based real-time operating system therefore specific care needs to be taken to follow the design rules for such systems. Data should always be maintained in SCIOPTA messages and shared resources should be encapsulated within SCIOPTA processes.

### 4.4 SCIOPTA System Calls

To design SCIOPTA systems, modules and processes, to handle interprocess communication and to understand the included software of the SCIOPTA delivery you need to have detailed knowledge of the SCIOPTA application programming interface (API). The SCIOPTA API consist of a number of system calls to the SCIOPTA kernel to let the SCIOPTA kernel execute the needed functions.

The SCIOPTA kernel has over 80 system calls. Some of these calls are very specific and are only used in particular situations. Thus many system calls are only needed if you are designing dynamic applications for creating and killing SCIOPTA objects. Other calls are exclusively foreseen to be used in CONNECTOR processes which are needed in distributed applications.

One of the strength of SCIOPTA is that it is easy-to-use. A large part of a typical SCIOPTA application can be written by using the system calls which are handling the interprocess communication: [sc\\_msgAlloc](#), [sc\\_msgTx](#), [sc\\_msgRx](#) and [sc\\_msgFree](#). These four system calls together with [sc\\_msgOwnerGet](#) which returns the owner of a message and [sc\\_sleep](#) which is used to suspend a process for a defined time, are often sufficient to write whole SCIOPTA applications.

Please consult chapter [6 “Application Programming Interface” on page 6-1](#) for a detailed description of all SCIOPTA system calls.

### 4.5 Writing your Application

Now you are ready to write your first SCIOPTA application. You need to design your modules, processes and the interprocess communication and coordination.

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 based real-time system you have different structure elements available to decompose the whole system into smaller parts such as modules and processes.

In chapter [7 “Application Programming” on page 7-1](#) you will find information how to design and use modules, processes, messages and other SCIOPTA objects.

For the "hello" getting started example the following files contain the application processes and messages:

#### Files

hello.c	Example process “hello”
display.c	Example process “display”
hello.msg	Example message definitions.

File location<installation\_folder>\sciopta\<version>\exp\krm\common\hello\

### 4.6 Error Handling

SCIOPTA uses a centralized mechanism for error reporting called Error Hooks. Error Hooks must be registered and written by the user.

Please consult chapter [5.11 “Error Handling” on page 5-19](#), chapter [7.11 “Error Hook” on page 7-24](#) and chapter [15 “Kernel Error Codes” on page 15-1](#) for more information about SCIOPTA error handling.

#### File

error.c	Error hook example.
---------	---------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\common\

The error hook can for example be registered in the SCIOPTA start hook which contains early startup code. The start hook could for instance be placed in an example configuration file. In the SCIOPTA getting started examples the file **system.c** contains some system initialization code including the start hook and the error hook registration. See the next chapter for more information about the system initialization file.

### 4.7 System Configuration and Initialization

System and application configuration consists basically to write the start hook (see chapter [8.2.4 “Start Hook” on page 8-3](#)), the system module hook (see chapter [8.2.6.1 “System Module Hook” on page 8-4](#)) and other system initialization functions. This is usually done in a specific file called system.c which can be found in the SCIOPTA examples deliveries.

For the simple "hello" getting started example the file system.c contains the system and application configuration for the example system module:

#### Source File

system.c	System configuration file including hooks and other setup code.
----------	---

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\

System and application configuration functions for other modules (user module hooks) is usually done in specific files having the same name as the module (**dev.c**, **ips.c** etc.) which can also be found in the SCIOPTA examples deliveries. Please consult also chapter [8.2.6.2 “User Modules Hooks” on page 8-4](#).

For IAR you need to define the free RAM of the modules in a separate file. In this area there are no initialized data. Module Control Block (ModuleCB), Process Control Blocks (PCBs), Stacks and Message Pools are placed in this free RAM. Please consult also chapter [12.3 “IAR Embedded Workbench Linker Script” on page 12-4](#).

map.c	Module mapping for IAR
-------	------------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\

## 4.8 General System Functions and Drivers

System functions which do not depend on a specific board and a specific CPU and are common for SCIOPTA systems. Files for external chips and controllers. Please consult also chapter [9.2 “General System Functions” on page 9-1.](#)

### Project Files

winIDEA_gnu.ind	iSYSTEM winIDEA indirection file for GNU GCC
-----------------	--

File location: <install\_folder>\sciopta\<version>\bsp\common\include\

## 4.9 ARM Cortex Family System Functions

Setup and driver descriptions which do not depend on a specific board and are common for all ARM based processors and controllers. Please consult also chapter [9.3 “ARM Cortex Family System Functions” on page 9-1.](#)

### Project Files

module.ld	Linker script: Module sections (common to all boards) for GNU GCC
-----------	---

File location: <install\_folder>\sciopta\<version>\bsp\arm\include\

### Source Files

cortexm3_cstartup.S	Cortex M3 C startup assembler source for GNU GCC.
cortexm3_exception.S	Cortex M3 exception handler for GNU GCC.
cortexm3_vector.S	Cortex M3 vector table for GNU GCC.

File location: <install\_folder>\sciopta\<version>\bsp\arm\src\gnu\

cortexm3_cstartup.s	Cortex M3 C startup assembler source for ARM RealView.
cortexm3_exception.s	Cortex M3 exception handler for ARM RealView.
cortexm3_vector.s	Cortex M3 vector table for ARM RealView.

File location: <install\_folder>\sciopta\<version>\bsp\arm\src\arm\

cortexm3_exception.s	Cortex M3 exception handler for IAR Embedded Workbench Version 5.
cortexm3_exception.s79	Cortex M3 exception handler for IAR Embedded Workbench Version 4.
cortexm3_vector.s	Cortex M3 vector table for IAR Embedded Workbench Version 5.
cortexm3_vector.s79	Cortex M3 vector table for IAR Embedded Workbench Version 4.

File location: <install\_folder>\sciopta\<version>\bsp\arm\src\iar\



### 4.10 ARM Cortex CPU System Functions and Drivers

For a basic system some general CPU ARM Derivative CPU Family functions and drivers which are not board dependent are needed.

Please consult chapters:

[9.4 “STM32 System Functions and Drivers” on page 9-3](#)

[9.5 “Stellaris System Functions and Drivers” on page 9-5](#)

#### Files

simple_uart.c	Simple polling uart function for printf debugging or logging.
systick.c	System timer setup.

File location: <install\_folder>\sciopta\<version>\bsp\arm\<cpu>\src\

### 4.11 Board Functions and Drivers

For your board you need to implement board functions and drivers. Usually at least a board setup file (boardSetup.c) containing early board setup code is needed. Board setup files are board specific and examples can be found in the SCIOPTA Board Support Package deliveries.

Please consult also chapters:

[9.6 “Olimex STM32-P103 Board” on page 9-7](#)

[9.7 “STMicroelectronics STM3210E-EVAL Evaluation Board” on page 9-10](#)

[9.8 “Luminary LM3S6965 Board” on page 9-13](#)

#### Files

boardSetup.c	Board setup.
led.c	Low-level routines to access the board LEDs

File location: <install\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\src\

## 4 SCIOPTA Kernel Project Overview



### Linker Scripts

A link is usually controlled by a linker script or linker control file. Linker scripts are compiler and board specific. Linker script examples can be found in the SCIOPTA Board Support Package deliveries.

Please consult chapter [12 “Linker Scripts and Memory Map” on page 12-1](#) for more information about linker scripts.

<board>.ld	Linker script for GNU GCC.
<board>.xcl	Linker script for IAR EW.
<board>.sct	Linker script for ARM RealView

File location: <install\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

### Board Configuration

It is good design practice to include specific board configurations, defines and settings in a file. In the SCIOPTA board support package deliveries such an example file is available.

config.h	Board configuration defines.
stm32f10x_conf.h	Driver library configuration definitions.

File location: <install\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

### Debugger Board Setup

<board>.ini	winIDEA board initialization.
<board>.mac	IAR EW board initialization.
<board>.cmm	Lauterbach Trace32 board initialization.

File location: <install\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

## 4 SCIOPTA Kernel Project Overview

### 4.12 Kernel Configuration

To build your system properly you need to configure the target system, modules, processes and pools. This is done with the SCIOPTA configuration utility **SCONF**.

Please consult chapter [10 “Kernel Configuration” on page 10-1](#) for more information about kernel configuration.

Example SCIOPTA configuration files hello.xml which can be loaded in the **SCONF** configuration program are included in the SCIOPTA delivery.

#### SCONF Configuration File

hello.xml	SCIOPTA kernel configuration file.
-----------	------------------------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\

After your configuration is correct the **SCONF** utility will generate three files which need to be included into your SCIOPTA project.

#### Generated Kernel Configuration Files

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

### 4.13 Kernel

The SCIOPTA kernel is provided in assembler source file and therefore compiler manufacturer specific. The kernels can be found in the library directory of the SCIOPTA delivery.

See also chapter [11.1 “Kernel” on page 11-1](#) for more information about the SCIOPTA kernel delivery.

#### Source File

sciopta.S	Kernel source file for GNU GCC
sciopta.s	Kernel source file for IAR Embedded Workbench Version 5.
sciopta.s79	Kernel source file for IAR Embedded Workbench Version 4.
sciopta_ads.s	Kernel source file for ARM RealView.

File location: <install\_folder>\sciopta\<version>\lib\arm\krm\

### 4.14 Libraries and Include Files

Make sure that you have included the kernel libraries as described in chapter [11.2 “Kernel Libraries” on page 11-1](#) and the Luminary® Stellaris Family Driver Libraries as described in chapter [11.3 “Luminary® Stellaris Family Driver Libraries” on page 11-7](#).

Please make sure that the include search directories are defined as described in chapter [11.4 “Include Files” on page 11-8](#). Please consult this chapter for more information about SCIOPTA includes.



4.15 Building the Project

4.15.1 Makefiles

The most flexible and direct way to build a SCIOPTA system is to use makefiles. Makefiles for the getting started projects are included in the delivery.

For the "hello" getting started example the makefiles are delivered in the example projects delivery:

Makefiles

Makefile	Example makefile.
----------	-------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\

board.mk	Board dependent makefiles.
----------	----------------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\

4.15.2 Eclipse

Please consult the getting started chapter [3 “Getting Started” on page 3-1](#) if you want to use Eclipse as an IDE for editing and building SCIOPTA applications.

4.15.3 iSYSTEM winIDEA

If winIDEA and the iSYSTEM emulator/debugger is your preferred build and debug environment you will find winIDEA project files in the SCIOPTA examples deliveries.

For the "hello" getting started example the iSYSTEM winIDEA project file can be found in the example projects delivery:

Project file

iC3000.xjrf	iSYSTEM winIDEA project file
iC3000.xqrf	iSYSTEM winIDEA project file

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>



4.15.4 IAR Embedded Workbench for ARM

If you are using the IAR Embedded Workbench environment you will find IAR EW project files in the SCIOPTA examples deliveries.

For the "hello" getting started example the IAR EW project file can be found in the example projects delivery:

Project file

<board>.ewd	IAR EW project file
<board>.eww	IAR EW project file

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>

## 5 SCIOPTA Technology and Methods

### 5.1 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.

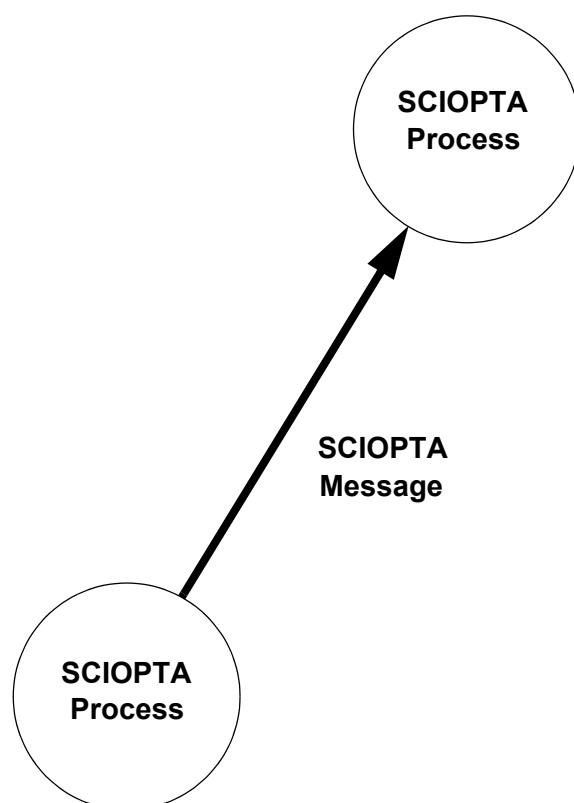
In SCIOPTA processes communicate and cooperate by exchanging messages. Messages can have a content to move data from one process to the other or can be empty just to coordinate processes. Often, process switches can occur as a result of a message transfer.

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.



## 5.2 Processes

### 5.2.1 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).

### 5.2.2 Process States

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

#### 5.2.2.1 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.

#### 5.2.2.2 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.

#### 5.2.2.3 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.

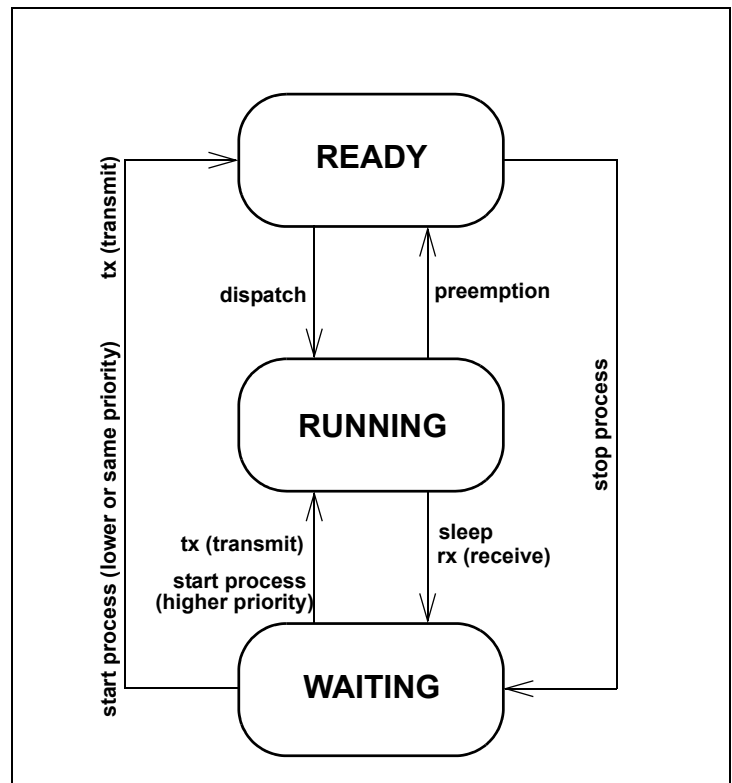


Figure 5-1: State Diagram of SCIOPTA Kernel

### 5.2.3 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 [5.4 “Modules” on page 5-10](#) for more information about the SCIOPTA module concept.

#### 5.2.3.1 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 processes are supposed to stay alive as long as the whole system is alive. But nevertheless in SCIOPTA static processes can be killed at run-time but they will not return their used memory.

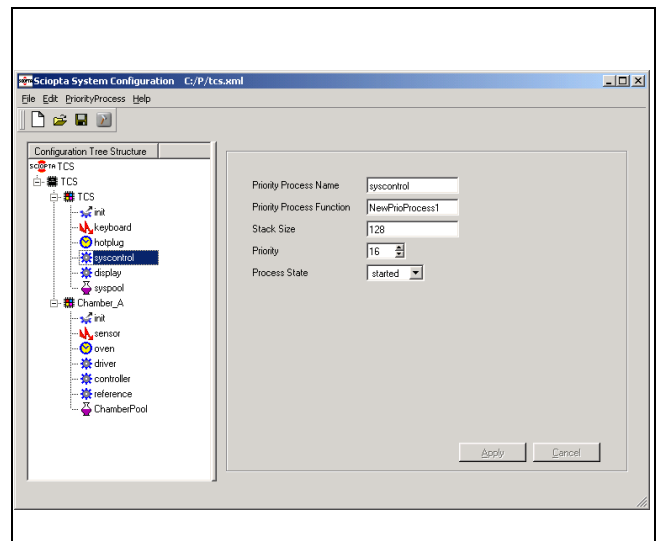


Figure 5-2: Process Configuration Window for Static Processes

#### 5.2.3.2 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 5-3: Create Process System Call



### 5.2.4 Process Types

#### 5.2.4.1 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.

#### 5.2.4.2 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.

In some SCIOPTA systems there might be two type of interrupt processes. 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.

#### 5.2.4.3 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.

### 5.2.4.4 Init Process



The init process is the first process in a module (please consult chapter [5.4 “Modules” on page 5-10](#) 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.

### 5.2.4.5 Supervisor Process



In SCIOPTA systems which include MMU protection prioritized processes can be defined to be user or supervisor processes. Supervisor processes have full access rights to system resources. Supervisor processes are often used in device drivers.

### 5.2.4.6 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. Please consult chapter [7.9.1 “Process Daemon” on page 7-21](#) for more information about the process daemon.

The **Kernel Daemon** (`sc_kerneld`) is creating and killing modules and processes. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about the kernel daemon.

### 5.2.5 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**

The effective priority has an upper limit of 31 which will never be exceeded even if the addition of module priority and process priority is higher.

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

#### 5.2.5.1 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.

#### 5.2.5.2 Interrupt Processes

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

#### 5.2.5.3 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.

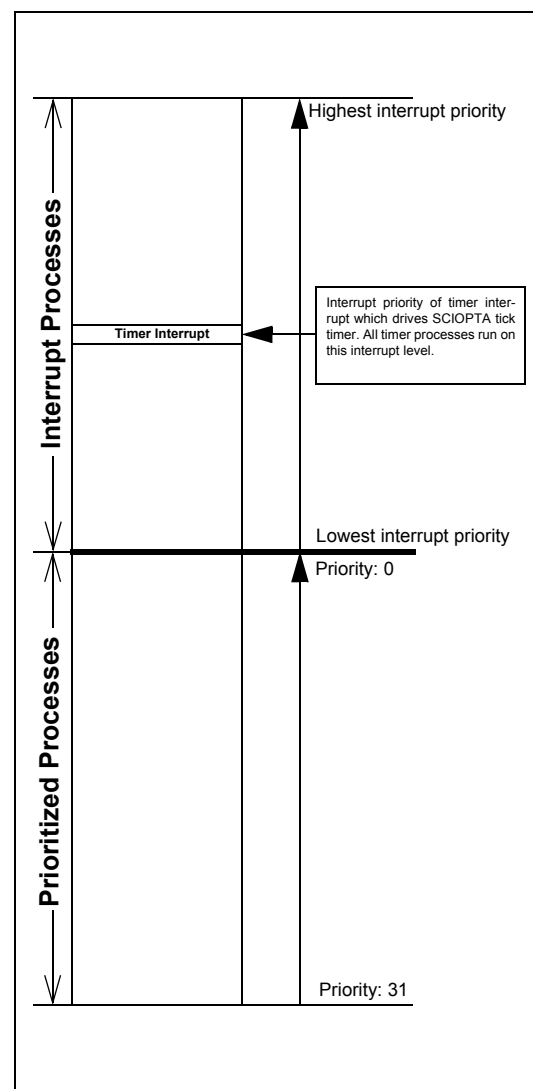


Figure 5-4: SCIOPTA Priority Diagram

### 5.3 Messages

#### 5.3.1 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.

#### 5.3.2 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

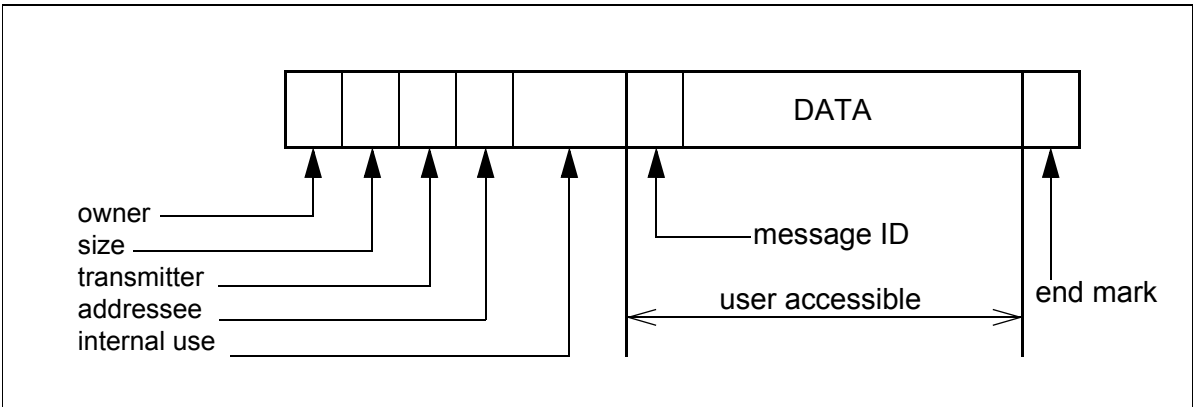


Figure 5-5: SCIOPTA Message Structure

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.

### 5.3.3 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.

#### 5.3.3.1 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.

### 5.3.4 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 [5.4 “Modules” on page 5-10](#) 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.

### 5.3.5 Message Passing

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

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

The size is given in bytes and the [sc\\_msgAlloc](#) function of SCIOPTA chooses an internal size out of a number of 4, 8 or 16 fixed sizes (see also chapter [5.3.3 “Message Sizes” on page 5-8](#)).

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.

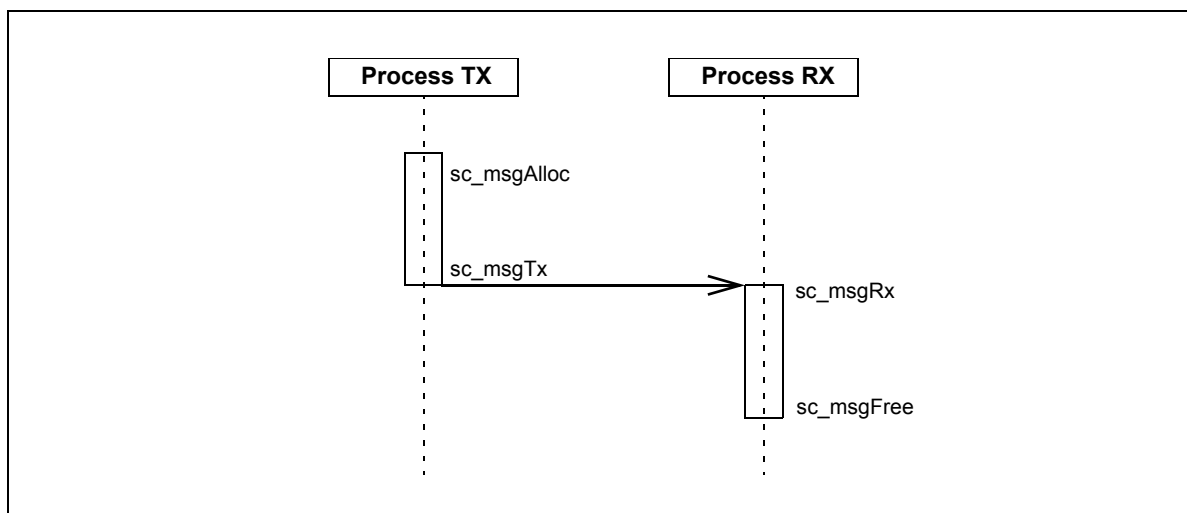


Figure 5-6: Message Sequence Chart of a SCIOPTA Message Passing

### 5.4 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.

#### 5.4.1 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 [5.4.3 “Messages and Modules” on page 5-11](#) 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.

#### 5.4.2 System Module

There is always at least one system module in a SCIOPTA system. This module is called system module (sometimes also named module 0) is a static module which is configured in the **SCONF** configuration utility.

### 5.4.3 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.

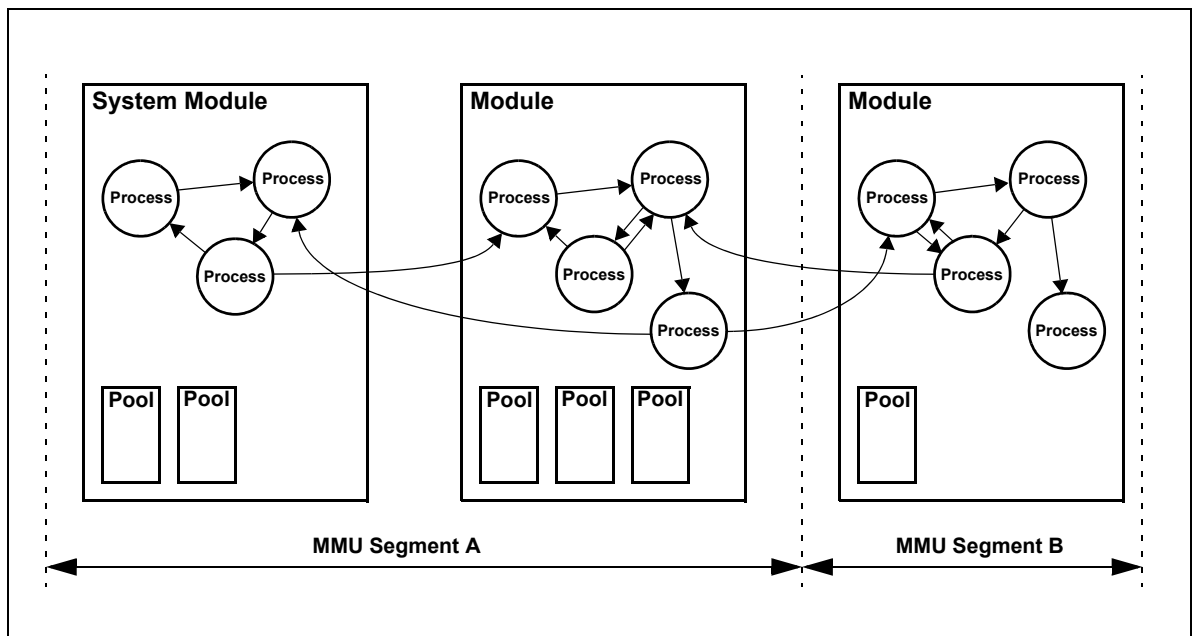


Figure 5-7: SCIOPTA Module Structure

### 5.4.4 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 laid 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.



## 5.5 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\\_triggerValueSet](#) 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\\_triggerValueGet](#) call.

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

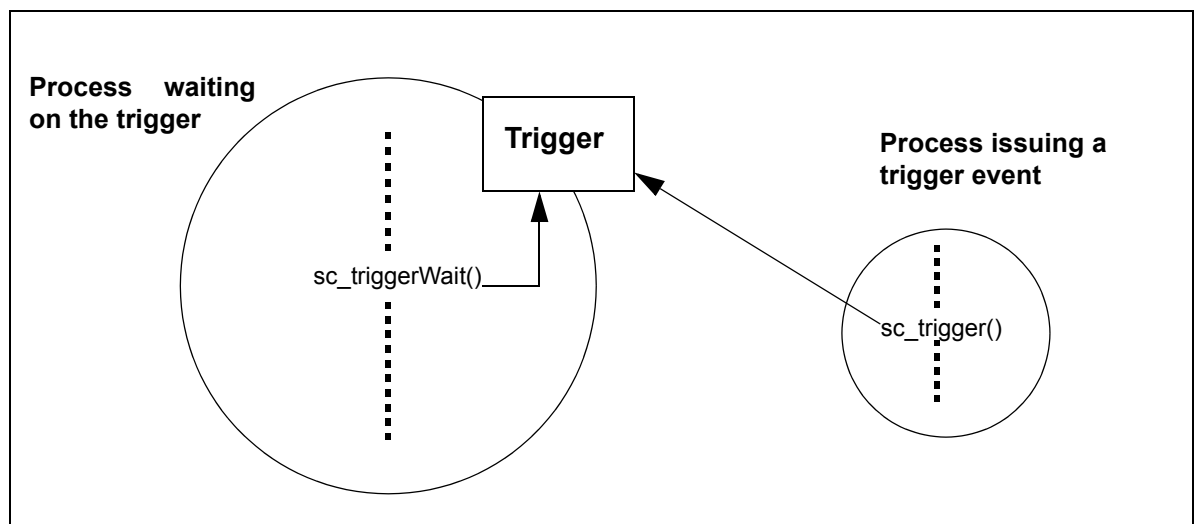


Figure 5-8: SCIOPTA Trigger

5.6 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.

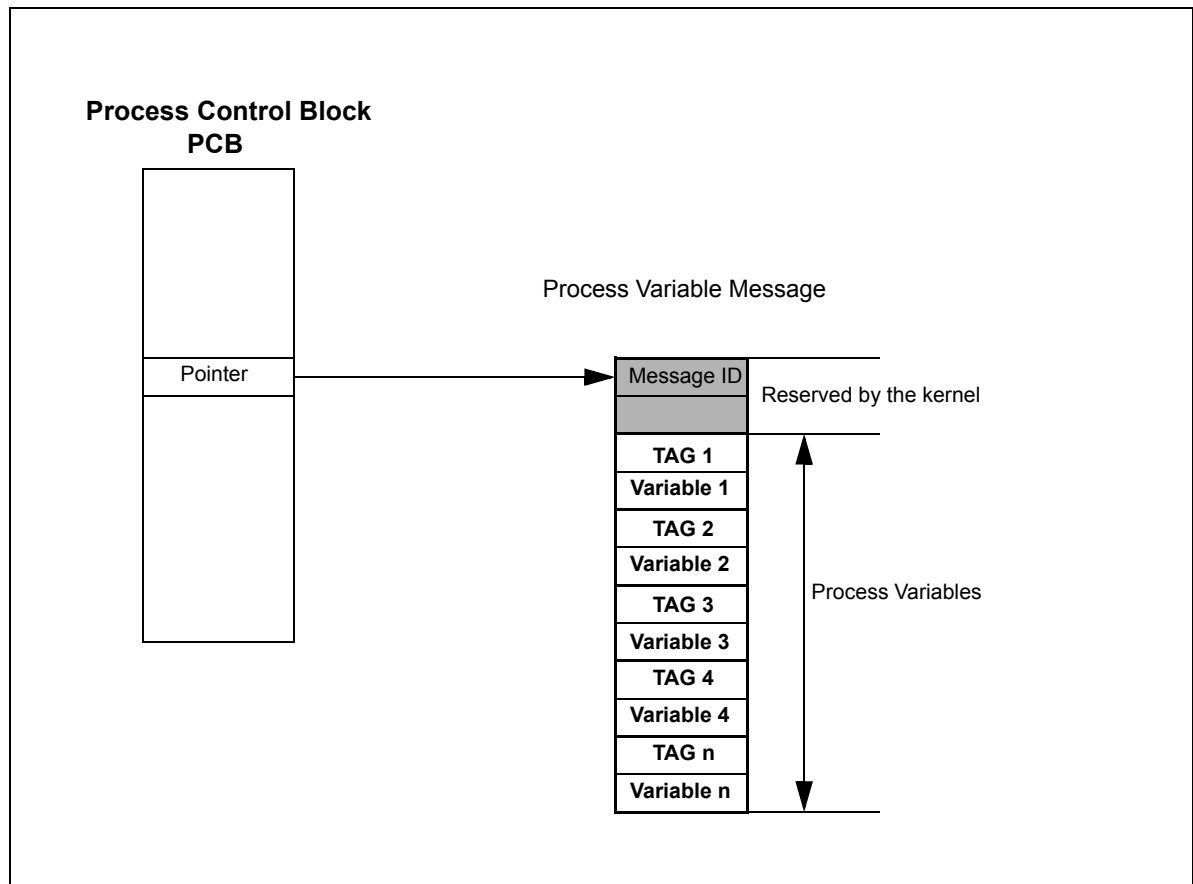


Figure 5-9: SCIOPTA Process Variables

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.

### 5.7 Time Management

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

#### 5.7.1 System Tick

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

Typical time values between two ticks range between one and then milliseconds. It is important to define the tick value to small as at every tick the kernel has some system work to perform, such as checking time-out and scheduling timer processes.

System tick should only be used for time-out and timing functions higher than one system tick. For very precise timing tasks it is better to use SCIOPTA interrupt processes connected to a CPU hardware timer.

The system tick is configured by the sciopta configuration utility a tick interrupt process must be specified which will call [sc\\_tick](#) at regular intervals. The tick interrupt process is usually included in the board support package.

In chapter [6.13 “Timing System Calls” on page 6-69](#) more information about SCIOPTA timing system calls are given.

#### 5.7.2 Time-Out Server

SCIOPTA has a built-in message based time-out server. Processes can register a time-out job at the time-out server. This done by the [sc\\_tmoAdd](#) system call which requests a time-out message from the kernel after a defined time.

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

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

The registered time-out can be cancelled by the [sc\\_tmoRm](#) call before the time-out has expired.

### 5.8 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.

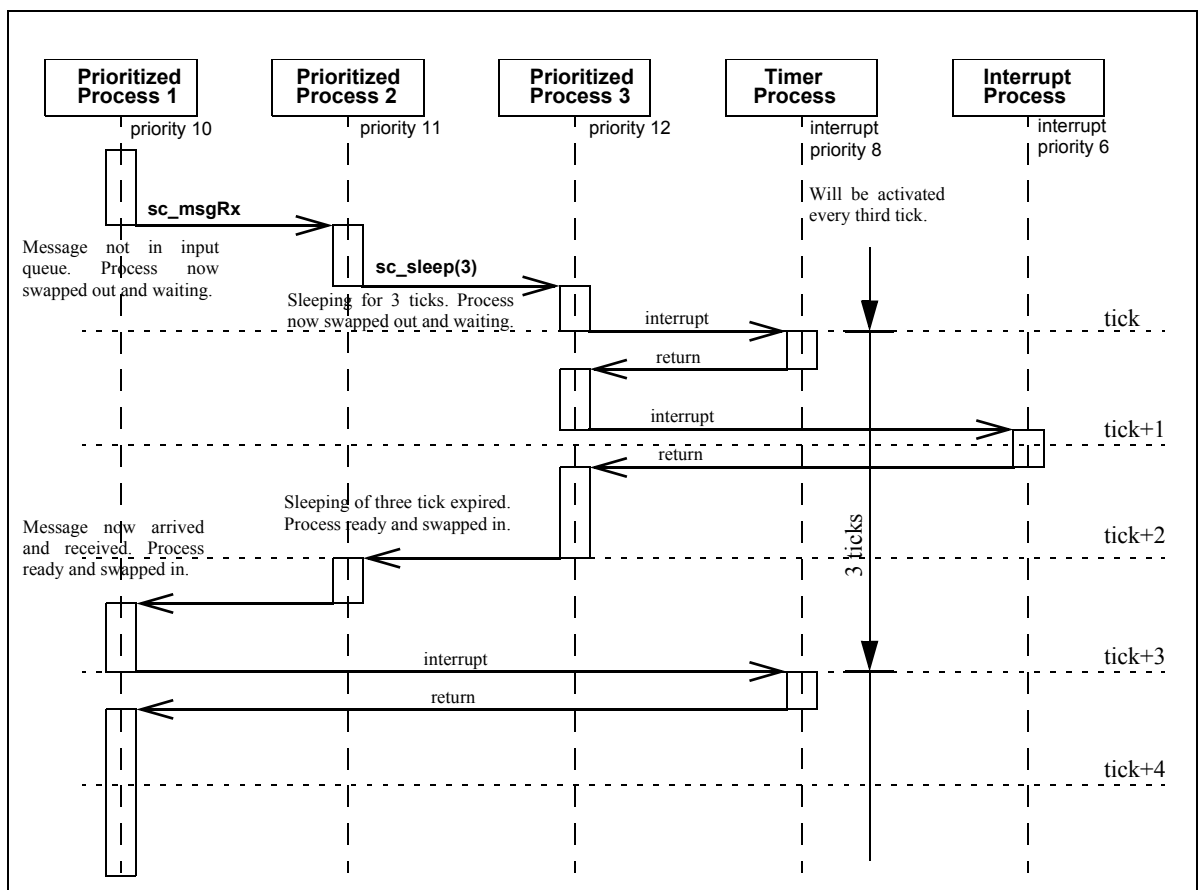


Figure 5-10: Scheduling Sequence Example

### 5.9 Distributed Systems

#### 5.9.1 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 were initially designed to fulfil the requirements of distributed systems.

#### 5.9.2 CONNECTORS

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

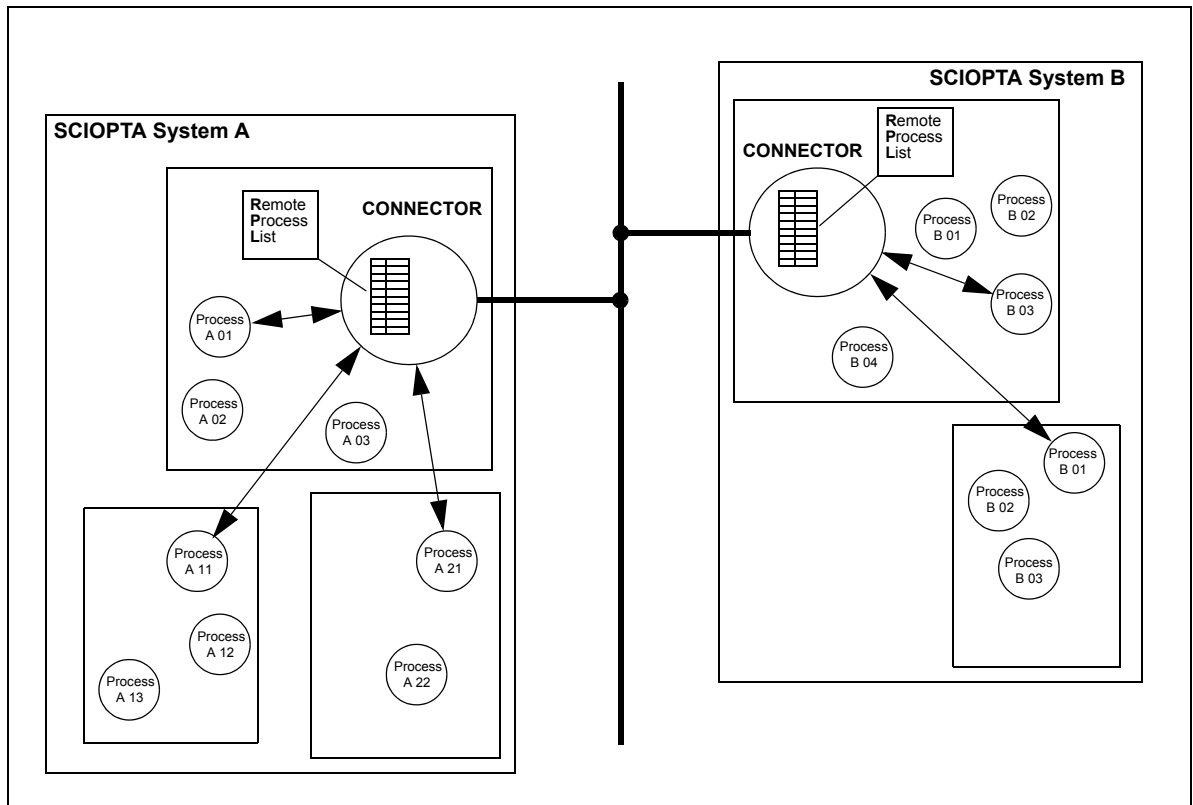


Figure 5-11: SCIOPTA Distributed System

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.

### 5.9.3 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 were local.

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

### 5.10 Observation

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

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

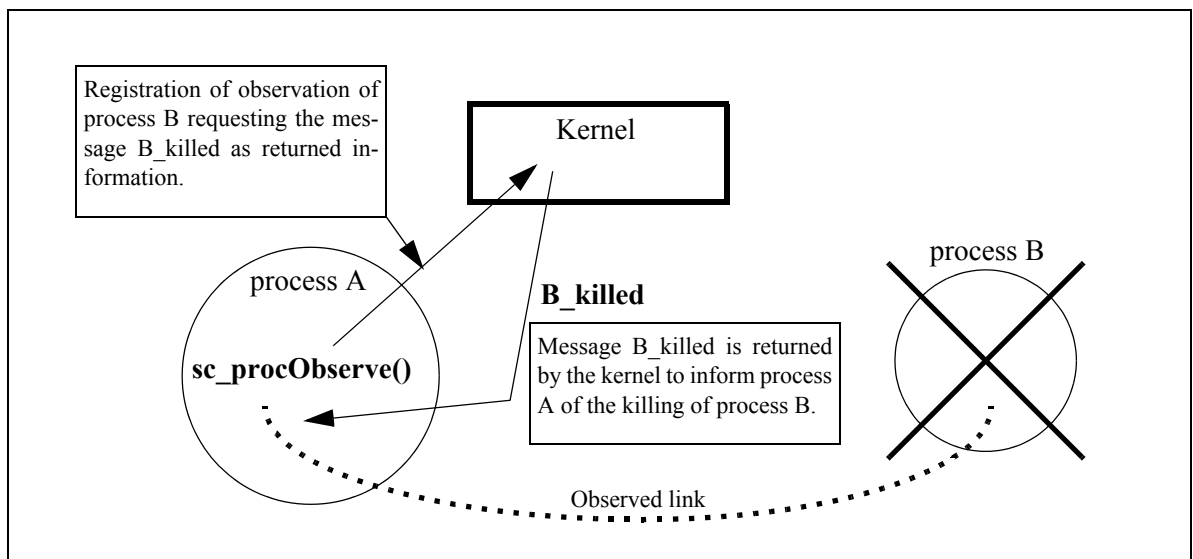


Figure 5-12: SCIOPTA Observation

### 5.11 Error Handling

#### 5.11.1 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.

#### 5.11.2 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.



### 5.12 Hooks

#### 5.12.1 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. Hooks need to be declared in the SCIOPTA kernel configuration (**SCONF**). Please consult chapter [16.9.2 “Configuring Hooks” on page 16-10](#) for more information. Additionally you need also need to declare hooks by using specific system calls. Please consult chapter [6.19 “Hook Registering System Calls” on page 6-88](#) for details.

#### 5.12.2 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 [7.11 “Error Hook” on page 7-24](#) and chapter [5.11 “Error Handling” on page 5-19](#) for additional information.

#### 5.12.3 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. Message hooks need to be registered as explained in chapter [6.19.2 “sc\\_msgHookRegister” on page 6-89](#).

#### 5.12.4 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. Pool hooks need to be registered as explained in chapter [6.19.3 “sc\\_poolHookRegister” on page 6-90](#).

#### 5.12.5 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.

Process hooks need to be registered as explained in chapter [6.19.4 “sc\\_procHookRegister” on page 6-91](#).

## 6 Application Programming Interface

### 6.1 Introduction

In chapter [6.2 “System Calls List” on page 6-1](#) the system calls are listed in alphabetical order while in the remaining chapters the system calls are listed in functional groups as follows:

System Call Group	Page
<a href="#">Message System Calls</a>	<a href="#">6-4</a>
<a href="#">Pool System Calls</a>	<a href="#">6-24</a>
<a href="#">Process Status System Calls</a>	<a href="#">6-30</a>
<a href="#">Process Create/Kill System Calls</a>	<a href="#">6-45</a>
<a href="#">Process Daemon System Calls</a>	<a href="#">6-49</a>
<a href="#">Process Observation System Calls</a>	<a href="#">6-51</a>
<a href="#">Process Variables System Calls</a>	<a href="#">6-53</a>
<a href="#">Module Status System Calls</a>	<a href="#">6-58</a>
<a href="#">Module Create/Kill System Calls</a>	<a href="#">6-61</a>
<a href="#">Module Friendship System Calls</a>	<a href="#">6-64</a>
<a href="#">Timing System Calls</a>	<a href="#">6-69</a>
<a href="#">Time-out Server System Calls</a>	<a href="#">6-75</a>
<a href="#">Trigger System Calls</a>	<a href="#">6-77</a>
<a href="#">CONNECTOR System Calls</a>	<a href="#">6-81</a>
<a href="#">CRC System Calls</a>	<a href="#">6-83</a>
<a href="#">Error System Calls</a>	<a href="#">6-85</a>
<a href="#">Hook Registering System Calls</a>	<a href="#">6-88</a>

### 6.2 System Calls List

System Call	Short Description	Page
<a href="#">sc_connectorRegister</a>	Registers a CONNECTOR process. The caller becomes a CONNECTOR process.	<a href="#">6-81</a>
<a href="#">sc_connectorUnregister</a>	Removes a process from the CONNECTOR process list.	<a href="#">6-82</a>
<a href="#">sc_miscCrc</a>	Calculates a 16 bit CRC over a specified memory range.	<a href="#">6-83</a>
<a href="#">sc_miscCrcContd</a>	Calculates a 16 bit CRC over an additional memory range.	<a href="#">6-84</a>
<a href="#">sc_miscErrnoGet</a>	Returns the process error number (errno) variable.	<a href="#">6-86</a>
<a href="#">sc_miscErrnoSet</a>	Sets the process error number (errno) variable.	<a href="#">6-87</a>
<a href="#">sc_miscError</a>	Calls the error hooks with an user error.	<a href="#">6-85</a>
<a href="#">sc_miscErrorHookRegister</a>	Registers an error hook.	<a href="#">6-88</a>
<a href="#">sc_moduleCreate</a>	Requests the kernel daemon to create a module.	<a href="#">6-61</a>
<a href="#">sc_moduleFriendAdd</a>	Adds a module to the friends of the caller.	<a href="#">6-64</a>

System Call	Short Description	Page
<a href="#">sc_moduleFriendAll</a>	Defines all existing modules in a system as friend.	<a href="#">6-65</a>
<a href="#">sc_moduleFriendGet</a>	Informs the caller if a module is a friend.	<a href="#">6-66</a>
<a href="#">sc_moduleFriendNone</a>	Removes all modules as friends of the caller.	<a href="#">6-67</a>
<a href="#">sc_moduleFriendRm</a>	Removes a module of the friends of the caller.	<a href="#">6-68</a>
<a href="#">sc_moduleIdGet</a>	Returns the ID of a module.	<a href="#">6-58</a>
<a href="#">sc_moduleInfo</a>	Returns a snap-shot of a module control block (mcb).	<a href="#">6-60</a>
<a href="#">sc_moduleKill</a>	Dynamically kills a whole module.	<a href="#">6-63</a>
<a href="#">sc_moduleNameGet</a>	Returns the name of a module.	<a href="#">6-59</a>
<a href="#">sc_msgAcquire</a>	Changes the owner of a message.	<a href="#">6-17</a>
<a href="#">sc_msgAddrGet</a>	Returns the process ID of the addressee of a message.	<a href="#">6-18</a>
<a href="#">sc_msgAlloc</a>	Allocates a memory buffer of selectable size from a message pool.	<a href="#">6-4</a>
<a href="#">sc_msgAllocClr</a>	Allocates a memory buffer of selectable size from a message pool and initializes the data area of the message to 0.	<a href="#">6-7</a>
<a href="#">sc_msgFree</a>	Returns a message to the message pool.	<a href="#">6-8</a>
<a href="#">sc_msgHookRegister</a>	Registers a message hook.	<a href="#">6-89</a>
<a href="#">sc_msgOwnerGet</a>	Returns the process ID of the owner of a message.	<a href="#">6-19</a>
<a href="#">sc_msgPoolIdGet</a>	Returns the pool ID of a message.	<a href="#">6-20</a>
<a href="#">sc_msgRx</a>	Receives a message.	<a href="#">6-14</a>
<a href="#">sc_msgSizeGet</a>	Returns the requested size of a message.	<a href="#">6-22</a>
<a href="#">sc_msgSizeSet</a>	Decreases the requested size of a message buffer.	<a href="#">6-23</a>
<a href="#">sc_msgSndGet</a>	Returns the process ID of the sender of a message.	<a href="#">6-21</a>
<a href="#">sc_msgTx</a>	Transmits a SCIOPTA message to a process (the addressee process).	<a href="#">6-10</a>
<a href="#">sc_msgTxAlias</a>	Transmits a SCIOPTA message to a process by setting a process ID as sender.	<a href="#">6-12</a>
<a href="#">sc_poolCreate</a>	Creates a new message pool inside the callers module.	<a href="#">6-24</a>
<a href="#">sc_poolDefault</a>	Sets a message pool as default pool.	<a href="#">6-27</a>
<a href="#">sc_poolHookRegister</a>	Registers a pool create or pool kill hook.	<a href="#">6-90</a>
<a href="#">sc_poolIdGet</a>	Returns the ID of a message pool.	<a href="#">6-26</a>
<a href="#">sc_poolInfo</a>	Returns a snap-shot of a pool control block.	<a href="#">6-28</a>
<a href="#">sc_poolKill</a>	Kills a message pool.	<a href="#">6-25</a>
<a href="#">sc_poolReset</a>	Resets a message pool in its original state.	<a href="#">6-29</a>
<a href="#">sc_procDaemonRegister</a>	Registers a process daemon.	<a href="#">6-49</a>
<a href="#">sc_procDaemonUnregister</a>	Removes a process from the process daemon list.	<a href="#">6-50</a>
<a href="#">sc_procHookRegister</a>	Registers a process hook.	<a href="#">6-91</a>
<a href="#">sc_procIdGet</a>	Returns the process ID of a process.	<a href="#">6-30</a>
<a href="#">sc_procIntCreate</a>	Requests the kernel daemon to create an Interrupt Process.	<a href="#">6-46</a>
<a href="#">sc_procKill</a>	Requests the kernel daemon to kill a process.	<a href="#">6-48</a>
<a href="#">sc_procNameGet</a>	Returns the full name of a process.	<a href="#">6-33</a>

System Call	Short Description	Page
<a href="#">sc_procObserve</a>	Supervises a process.	<a href="#">6-51</a>
<a href="#">sc_procPathGet</a>	Returns the full path of a process.	<a href="#">6-34</a>
<a href="#">sc_procPpidGet</a>	Returns the process ID of the parent (creator) of a process.	<a href="#">6-32</a>
<a href="#">sc_procPrioCreate</a>	Requests the kernel daemon to create a prioritized process.	<a href="#">6-45</a>
<a href="#">sc_procPrioGet</a>	Returns the priority of a prioritized process.	<a href="#">6-35</a>
<a href="#">sc_procPrioSet</a>	Sets the priority of a process.	<a href="#">6-36</a>
<a href="#">sc_procSchedLock</a>	Locks the scheduler and returns the number of times it has been locked before.	<a href="#">6-37</a>
<a href="#">sc_procSchedUnlock</a>	Unlocks the scheduler.	<a href="#">6-38</a>
<a href="#">sc_procSliceGet</a>	Returns the time slice of a timer process.	<a href="#">6-39</a>
<a href="#">sc_procSliceSet</a>	Sets the time slice of a timer process.	<a href="#">6-40</a>
<a href="#">sc_procStart</a>	Starts a prioritized or timer process.	<a href="#">6-41</a>
<a href="#">sc_procStop</a>	Stops a prioritized or timer process.	<a href="#">6-42</a>
<a href="#">sc_procTimCreate</a>	Requests the kernel daemon to create a timer process.	<a href="#">6-47</a>
<a href="#">sc_procUnobserve</a>	Cancels an installed supervision of a process.	<a href="#">6-52</a>
<a href="#">sc_procVarDel</a>	Removes a process variable from the process variable data area.	<a href="#">6-56</a>
<a href="#">sc_procVarGet</a>	Returns a process variable.	<a href="#">6-55</a>
<a href="#">sc_procVarInit</a>	Setups and initializes a process variable area.	<a href="#">6-53</a>
<a href="#">sc_procVarRm</a>	Removes a whole process variable area.	<a href="#">6-57</a>
<a href="#">sc_procVarSet</a>	Defines or modifies a process variable.	<a href="#">6-54</a>
<a href="#">sc_procVectorGet</a>	Returns the interrupt vector of the caller.	<a href="#">6-43</a>
<a href="#">sc_procYield</a>	Yields the CPU to the next ready process within the current process's priority group.	<a href="#">6-44</a>
<a href="#">sc_sleep</a>	Suspends the calling process for a defined time.	<a href="#">6-69</a>
<a href="#">sc_tick</a>	Calls directly the kernel tick function and advances the kernel tick counter by 1.	<a href="#">6-70</a>
<a href="#">sc_tickGet</a>	Returns the actual kernel tick counter value.	<a href="#">6-71</a>
<a href="#">sc_tickLength</a>	Sets the current system tick length in micro seconds.	<a href="#">6-72</a>
<a href="#">sc_tickMs2Tick</a>	Converts a time from milliseconds into system ticks.	<a href="#">6-73</a>
<a href="#">sc_tickTick2Ms</a>	Converts a time from system ticks into milliseconds.	<a href="#">6-74</a>
<a href="#">sc_tmoAdd</a>	Requests a time-out message from the kernel after a defined time.	<a href="#">6-75</a>
<a href="#">sc_tmoRm</a>	Removes a time-out before it is expired.	<a href="#">6-76</a>
<a href="#">sc_trigger</a>	Activates a process trigger.	<a href="#">6-77</a>
<a href="#">sc_triggerValueGet</a>	Returns the value of a process trigger.	<a href="#">6-79</a>
<a href="#">sc_triggerValueSet</a>	Set the value of a process trigger to any positive value.	<a href="#">6-80</a>
<a href="#">sc_triggerWait</a>	Waits on the process trigger.	<a href="#">6-78</a>

6.3 Message System Calls

6.3.1 sc\_msgAlloc

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

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

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

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

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

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

Parameter	Description	
size	The requested size of the message buffer.	
id	Message ID which will be placed at the beginning of the data buffer of the message.	
plid	Pool ID from where the message will be allocated.	
	Value	Description
	<pool id>	Pool ID from where the message will be allocated.
	SC_DEFAULT_POOL	Message will be allocated from the default pool. The default pool can be set by the system call <a href="#">sc_poolDefault</a> .

Parameter	Description	
<b>tmo</b>	Allocation timing parameter	
	Value	Description
	SC_ENDLESS_TMO	Time-out is not used. Blocks and waits endless until a buffer is available from the message pool.
	SC_NO_TMO	A NIL pointer will be returned if there is memory shortage in the message pool.
	SC_FATAL_IF_TMO	A (fatal) kernel error will be generated if a message buffer of the requested size is not available.
	$0 < \text{tmo} \leq \text{SC\_TMO\_MAX}$	Time-out value in system ticks. Alloc with time-out. Blocks and waits the specified number of ticks to get a message buffer.

Return Value	Condition
Pointer to the allocated buffer	Parameter <b>tmo</b> = SC_ENDLESS_TMO. Parameter <b>tmo</b> = SC_NO_TMO and if a buffer of the requested size is available. Parameter <b>tmo</b> > 0 and the system responds within the time-out period.
NIL pointer	Parameter <b>tmo</b> = SC_NO_TMO and if a buffer of the requested size is not available. Parameter <b>tmo</b> > 0 and the system does not respond within the time-out period.

Example

```

/* Allocate TEST_MSG from default pool */

sc_msg_t msg;

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

```

### Errors

Error Code	Extra	Level	Comment
0x01030002	tmo	module	Timeout parameter out of range.
0x0102d002	0	module	Calling process uses tmo but is not an prioritized process.
0x01027001	Pointer to module CB	system	Illegal module CB pointer, possible corruption of module table.
0x01013000	Pool index	none	Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook.
0x01007002	Requested size	module	Illegal message size requested.
0x01001002	Pool index	module	Pool <index> is not available.
0x01014002	0	module	Process would swap but interrupts and/or scheduler are/is locked.
0x0100d002	Requested size	module	Request for <size> bytes could not be fulfilled.

## 6.3.2 sc\_msgAllocClr

This system call works exactly the same as [sc\\_msgAlloc](#) but it will initialize the data area of the message to 0. Please consult chapter [6.3.1 “sc\\_msgAlloc” on page 6-4](#).

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

### Parameter

Same as in chapter [6.3.1 “sc\\_msgAlloc” on page 6-4](#).

### Return Value

Same as in chapter [6.3.1 “sc\\_msgAlloc” on page 6-4](#).

### Example

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

sc_msg_t msg;

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

### Errors

Error Code	Extra	Level	Comment
0x01030002	tmo	module	Timeout parameter out of range.
0x0102d002	0	module	Calling process uses tmo but is not an prioritized process.
0x01027001	Pointer to module CB	system	Illegal module CB pointer, possible corruption of module table.
0x01013000	Pool index	none	Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook.
0x01007002	Requested size	module	Illegal message size requested.
0x01001002	Pool index	module	Pool <index> is not available.
0x01014002	0	module	Process would swap but interrupts and/or scheduler are/is locked.
0x0100d002	Requested size	module	Request for <size> bytes could not be fulfilled.



## 6 Application Programming Interface

### 6.3.3 `sc_msgFree`

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

Only the owner of a message is allowed to free it by calling [sc\\_msgFree](#). It is a fatal error to free a message owned by another process. If you have, for example transmitted a message to another process it is the responsibility of the receiving process to free the message.

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

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

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

Parameter	Description
<b>msgptr</b>	Pointer to the message.

Return Value	
<b>none</b>	

#### Example

```
/* Free a message */
```

```
sc_msg_t msg;
```

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

### Errors

Error Code	Extra	Level	Comment
0x0200a002	0 or msg pointer	module	Either msg or pointer to msg are zero.
0x02020002	Process ID	module	Process ID wrong.
0x0200c002	sc_msg_hd_t *	module	Calling process does not own the message.
0x02027002	sc_msg_hd_t *	module	Module in message header has an illegal value.
0x02001002	sc_msg_hd_t *	module	Pool index in message header has an illegal value.
0x02009002	sc_msg_hd_t *	module	Either pool ID or buffersize index are corrupted.
0x02008002	sc_msg_hd_t *	module	The pointer is outside the pool. Possible pool id corruption.
0x02011002	sc_msg_hd_t *	module	Endmark of the buffer is destroyed.
0x02012002	sc_msg_hd_t *	module	Endmark of the previous buffer is destroyed. The pointer is the start of this buffer.

## 6 Application Programming Interface

### 6.3.4 sc\_msgTx

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

Each SCIOPTA process has one message queue for messages which have been sent to the process. The [sc\\_msgTx](#) system call will enter the message at the end of the receivers message queue.

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

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

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

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

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

Parameter	Description
<b>msgptr</b>	Pointer to the message.
<b>addr</b>	The process ID of the addressee.
<b>flags</b>	Must be set to 0 (reserved for later use).

Return Value	
none	

#### Example

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

```
sc_msg_t msg;
sc_pid_t addr;
```

```
/* ... */
```

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

## Errors

Error Code	Extra	Level	Comment
0x0800a002	0 or msg pointer	module	Either msg or pointer to msg are zero.
0x08020002	Process ID	module	Process ID wrong.
0x0800c002	sc_msg_hd_t *	module	Calling process does not own the message.
0x08027002	sc_msg_hd_t *	module	Module in message header has an illegal value.
0x08001002	sc_msg_hd_t *	module	Pool index in message header has an illegal value.
0x08009002	sc_msg_hd_t *	module	Either pool ID or buffersize index are corrupted.
0x08008002	sc_msg_hd_t *	module	The pointer is outside the pool. Possible pool id corruption.
0x08011002	sc_msg_hd_t *	module	Endmark of the buffer is destroyed.
0x08012002	sc_msg_hd_t *	module	Endmark of the previous buffer is destroyed. The pointer is the start of this buffer.

## 6.3.5 sc\_msgTxAlias

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

The usual [sc\\_msgTx](#) system call sets always the calling process as sender. If you need to set another process ID as sender you can use this [sc\\_msgTxAlias](#) call.

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

Otherwise [sc\\_msgTxAlias](#) works the same way as [sc\\_msgTx](#).

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

Parameter	Description
<b>msgptr</b>	Pointer to the message.
<b>addr</b>	The process ID of the addressee.
<b>flags</b>	Must be set to 0 (reserved for later use).
<b>alias</b>	The process ID specified as sender.

Return Value	
none	

### Example

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

sc_msg_t msg;
sc_pid_t addr;
sc_pid_t other;

/* ... */

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

### Errors

Error Code	Extra	Level	Comment
0x0900a002	0 or msg pointer	module	Either msg or pointer to msg are zero.
0x09020002	Process ID	module	Process ID wrong.
0x0900c002	sc_msg_hd_t *	module	Calling process does not own the message.
0x09027002	sc_msg_hd_t *	module	Module in message header has an illegal value.
0x09001002	sc_msg_hd_t *	module	Pool index in message header has an illegal value.
0x09009002	sc_msg_hd_t *	module	Either pool ID or buffersize index are corrupted.
0x09008002	sc_msg_hd_t *	module	The pointer is outside the pool. Possible pool id corruption.
0x09011002	sc_msg_hd_t *	module	Endmark of the buffer is destroyed.
0x09012002	sc_msg_hd_t *	module	Endmark of the previous buffer is destroyed. The pointer is the start of this buffer.

## 6 Application Programming Interface

### 6.3.6 sc\_msgRx

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

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

A pointer to an array (**wanted**) containing the messages (and/or process IDs) which will be scanned by [sc\\_msgRx](#). The array must be terminated by 0.

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

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

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

The caller can also specify a time-out value **tmo**. The caller will not wait (swapped out) longer than the specified time. If the time-out expires the process will be made ready again and [sc\\_msgRx](#) will return with **NULL**.

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

Parameter	Description	
<b>tmo</b>	Time-out parameter.	
	Value	Description
	SC_ENDLESS_TMO	Blocks and waits endless until the message is received.
	SC_NO_TMO	No time-out, returns immediately. Must be set for interrupt and timer processes.
	$0 < \text{tmo} \leq \text{SC\_TMO\_MAX}$	Time-out value in system ticks. Receive with time-out. Blocks and waits a specified maximum number of ticks to receive the message. If the time-out expires the process will be made ready again and <a href="#">sc_msgRx</a> will return with <b>NULL</b> .
<b>wanted</b>	Pointer to the message (or pid) array.	
	Value	Description
	<ptr>	Pointer to the message (or process ID) array.
	SC_MSGRX_ALL	All messages will be received.

Parameter	Description	
<b>flag</b>	More than one value can be defined and must be separated by OR instructions.	
	Value	Description
	SC_MSGRX_MSGID	An array of wanted message IDs is given.
	SC_MSGRX_PID	An array of process ID's from where sent messages are received is given.
	SC_MSGRX_NOT	An array of message ID's is given which will be excluded from receive.
	SC_MSGRX_BOTH	An array of pairs of message ID's and process ID's are given to receive specific messages from specific transmitting processes.

Return Value	Condition
Pointer to the received message	Message has been received. The caller becomes owner of the received message.
<b>0</b>	Time-out expired. The process will be made ready again.

### Examples

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

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

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

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



## 6 Application Programming Interface

```

/* wait for message from a certain process */

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

/* Wait for any message */

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

```

### Errors

Error Code	Extra	Level	Comment
0x0a02d002	0	module	The calling process uses a timeout and is not a prioritized process.
0x0a030002	tmo	module	The timeout parameter is out of range.
0x0a02b000	0	none	The flag parameter is wrong. This is a warning. Upon return the kernel chooses SC_MSGRX_MSGID.
0x0a014002	0	module	Process would swap but interrupts and/or scheduler is/are locked.

## 6 Application Programming Interface

### 6.3.7 sc\_msgAcquire

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

The kernel will copy the message into a new message buffer allocated from the default pool on the following condition:

The message resides not in a pool of the callers module and the callers module is not friend to the module where the message resides.

In this case the message pointer (**msgptr**) will be modified.

Please use [sc\\_msgAcquire](#) with care. Transferring message buffers without proper ownership control by using [sc\\_msgAcquire](#) instead of transmitting and receiving messages with [sc\\_msgTx](#) and [sc\\_msgRx](#) will cause problems if you are killing processes.

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

Parameter	Description
msgptr	Pointer to the message buffer.

Return Value	
none	

#### Example

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

sc_msg_t msg;
sc_msg_t msg2;

msg = sc_msgRx(SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

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

#### Errors

Error Code	Extra	Level	Comment
0x0c00a002	0 or msg pointer	module	Either msg or pointer to msg are zero.
0x0c009002	sc_msg_hd_t *	module	Header corrupted or caller tried to acquire an already freed message.

### 6.3.8 sc\_msgAddrGet

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

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

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

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

Parameter	Description
msgptr	Pointer to the message.

Return Value	Condition
process ID of the addressee of the message	None

#### Example

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

sc_msg_t msg;
sc_pid_t addr;

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

#### Errors

Error Code	Extra	Level	Comment
0x0300a002	0 or msg pointer	module	Either msg or pointer to msg are zero.

## 6 Application Programming Interface

### 6.3.9 sc\_msgOwnerGet

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

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

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

Parameter	Description
msgptr	Pointer to the message.

Return Value	Condition
process ID of the owner of the message	None

#### Example

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

```
sc_msg_t msg;
sc_pid_t owner;
```

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

#### Errors

Error Code	Extra	Level	Comment
0x0700a002	0 or msg pointer	module	Either msg or pointer to msg are zero.

## 6.3.10 sc\_msgPoolIdGet

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

When you are allocating a message with [sc\\_msgAlloc](#) you need to give the ID of a pool from where the message will be allocated. During run-time you sometimes need to this information from received messages.

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

Parameter	Description
msgptr	Pointer to the message.

Return Value	Condition
Pool ID where the message resides	The message is in the same module than the caller.
SC_DEFAULT_POOL	The message is <b>not</b> in the same module than the caller.

### Example

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

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

idx = sc_msgPoolIdGet (&msg);
```

### Errors

Error Code	Extra	Level	Comment
0x0b00a002	0 or msg pointer	module	Either msg or pointer to msg are zero.

### 6.3.11 sc\_msgSndGet

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

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

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

Parameter	Description
msgptr	Pointer to the message.

Return Value	Condition
Process ID of the sender of the message	Message was sent at least once.
Process ID of the owner of the message	Message was never sent.

#### Example

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

sc_msg_t msg;
sc_pid_t sndr;

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

#### Errors

Error Code	Extra	Level	Comment
0x0400a002	0 or msg pointer	module	Either msg or pointer to msg are zero.

## 6.3.12 sc\_msgSizeGet

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

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

Parameter	Description
msgptr	Pointer to the message.

Return Value	Condition
Requested size of the message	None

### Example

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

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

size = sc_msgSizeGet(&msg);
```

### Errors

Error Code	Extra	Level	Comment
0x0500a002	0 or msg pointer	module	Either msg or pointer to msg are zero.

### 6.3.13 sc\_msgSizeSet

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

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

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

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

Parameter	Description
<b>msgptr</b>	Pointer to the message.
<b>newsz</b>	New requested size of the message buffer.

Return Value	Condition
New requested buffer size	Call without error condition.
Old requested buffer size	Wrong request such as requesting a higher buffer size as the old one.

#### Example

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

sc_msg_t msg;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

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

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

#### Errors

Error Code	Extra	Level	Comment
0x0700a002	0 or msg pointer	module	Either msg or pointer to msg are zero.
0x06030002	size	module	Parameter size is smaller than the size of a message id.
0x0600b002	size	module	Message would be enlarged.



## 6 Application Programming Interface

### 6.4 Pool System Calls

#### 6.4.1 sc\_poolCreate

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

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

Parameter	Description						
<b>start</b>	Start address of the pool. If a 0 is given the kernel will automatically take the next free address in the module						
<b>size</b>	<p>Size of the message pool.</p> <p>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).</p> <p>The size of the pool control block (pool_cb) can be calculated according to the following formula:</p> $\text{pool\_cb} = 68 + n * 20 + \text{stat} * n * 20$ <p>where:</p> <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>n</td><td>Maximum buffer sizes defined for the whole system (and not the buffer sizes of the created pool). Value n can be 4, 8 or 16.</td></tr> <tr> <td>stat</td><td>Process statistics or message statistics are used (1) or not used (0).</td></tr> </table> <p>Please consult the configuration chapter <a href="#">16.9.3 “Debug Configuration” on page 16-11</a> for more information about statistics.</p>	Value	Description	n	Maximum buffer sizes defined for the whole system (and not the buffer sizes of the created pool). Value n can be 4, 8 or 16.	stat	Process statistics or message statistics are used (1) or not used (0).
Value	Description						
n	Maximum buffer sizes defined for the whole system (and not the buffer sizes of the created pool). Value n can be 4, 8 or 16.						
stat	Process statistics or message statistics are used (1) or not used (0).						
<b>nbufs</b>	The number of fixed buffer sizes. This can be 4, 8 or 16. It must always be lower or equal of the fixed buffer sizes which is defined for the whole system						
<b>bufsizes</b>	Pointer to an array of the fixed buffer sizes in ascending order.						
<b>name</b>	<p>Pointer to the name of the pool to create.</p> <p>The name is represented by a ASCII character string terminated by 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore.</p>						

Return Value	Condition
pool ID of the created message pool	

## 6.4.2 sc\_poolKill

This system call is used to kill a message pool.

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

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

Every process inside a module can kill a pool.

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

Parameter	Description
plid	Pool ID of the pool to be killed.

Return Value	
none	

## 6.4.3 sc\_poolIdGet

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

In contrast to the call [sc\\_procIdGet](#), you can just give the name as parameter and not a path.

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

Parameter	Description
name	Pointer to the 0 terminated name string of the pool.

Return Value	Condition
pool ID	Pool name was found.
poolID of default pool	Parameter name: zero or SC_DEFAULT_POOL
SC_ILLEGAL_POOLID	Pool name was not found.

## 6 Application Programming Interface

### 6.4.4 sc\_poolDefault

This system call sets a message pool as default pool.

The default pool will be used by the [sc\\_msgAlloc](#) system call if the parameter for the pool to allocate the message from is defined as `SC_DEFAULT_POOL`.

Each process can set its default message pool by [sc\\_poolDefault](#). The defined default message pool is stored inside the process control block. The initial default message pool at process creation is 0.

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

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

Parameter	Description	
idx	Pool ID	
	<b>Value</b>	<b>Description</b>
	Zero or positive	Pool ID
	-1	Request to return the ID of the default pool.

Return Value	Condition
pool ID of the default pool	

## 6 Application Programming Interface

### 6.4.5 sc\_poolInfo

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

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

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

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

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

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

Parameter	Description
<b>mid</b>	Module ID where the pool resides of which the control block will be returned.
<b>plid</b>	ID of the pool of which the pool control block data will be returned.
<b>info</b>	Pointer to a local structure of a pool control block. This structure will be filled with the pool control block data.

Return Value	Condition
<b>!=0</b>	The pool control block data was successfully retrieved.
<b>0</b>	System call fails and the pool control block data could not be retrieved.

## 6.4.6 sc\_poolReset

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

All messages in the pool must be freed and returned before a [sc\\_poolReset](#) call can be used.

The structure of the pool will be re-initialized. The message buffers in free-lists will be transformed back into unused memory. This “fresh” memory can now be used by [sc\\_msgAlloc](#) to allocate new messages.

Each process in a module can reset a pool.

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

Parameter	Description
plid	Pool ID of the pool to reset.

Return Value	
none	

6.5 Process Status System Calls

6.5.1 sc\_procIdGet

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

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

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

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

Parameter	Description						
<b>path</b>	Pointer to the path including the name of the process.  If the process resides within the caller's module: path::= <b>process_name</b>  If the process resides in the system module of the caller's target: path::=/" <b>process_name</b>  If the process resides in the system module of an external target: path::=/"< <b>system_name</b> >/" <b>process_name</b>  If the process resides in another than the system module of the caller's target: path::=/"< <b>module_name</b> >/" <b>process_name</b>  If the process resides in another than the system module of an external target: path::=/"< <b>system_name</b> >/"< <b>module_name</b> >/" <b>process_name</b>						
<b>tmo</b>	Time to wait for a response in ticks. This parameter is not allowed if asynchronous timeout is disabled at system configuration ( <b>SCONF</b> ).						
	<table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>SC_NO_TMO</td><td>No time-out, returns immediately.</td></tr> <tr> <td>0 &lt; tmo &lt; SC_TMO_MAX</td><td>Time-out value in system ticks.</td></tr> </table>	Value	Description	SC_NO_TMO	No time-out, returns immediately.	0 < tmo < SC_TMO_MAX	Time-out value in system ticks.
Value	Description						
SC_NO_TMO	No time-out, returns immediately.						
0 < tmo < SC_TMO_MAX	Time-out value in system ticks.						

Return Value	Condition
Process ID of the found process	Process was found within the tmo time period.
Current process ID (process ID of the caller)	Parameter <b>path</b> is NULL and parameter <b>tmo</b> is SC_NO_TMO.
SC_ILLEGAL_PID	Process was not found within the tmo time period.

### 6.5.1.1 `sc_procIdGet` in Interrupt Processes

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

The reply message is defined as follows:

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

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



## 6.5.2 sc\_procPidGet

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

```
sc_pid_t sc_procPidGet (
    sc_pid_t    pid
);
```

Parameter	Description	
pid		
	Value	Description
	<pid>	Process ID.
	SC_CURRENT_PID	Current process ID (process ID of the caller).

Return Value	Condition
Process ID of the parentprocess	Parent process exists
Process ID of the parentprocess of the caller	Parameter <b>pid</b> was SC_CURRENT_PID.
SC_ILLEGAL_PID	Parent process does no longer exist.

## 6 Application Programming Interface

### 6.5.3 sc\_procNameGet

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

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

The user must free the message after use.

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

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

Parameter	Description
pid	Process ID of the process where the name is requested.

Return Value	Condition
Message owned by the caller	Process exists

#### 6.5.3.1 Returned Message

If the return value is nonzero the returned message buffer is owned by the caller and the message is of type **sc\_procNameGetMsgReply\_t** and the message ID is **SC\_PROCNAMEGETMSG\_REPLY**. The message data contains the 0 terminated ASCII name string of the process. The message is defined in the **sciopta.msg** include file.

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

## 6 Application Programming Interface

### 6.5.4 sc\_procPathGet

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

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

The user must free the message after use.

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

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

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

Return Value	Condition
Message owned by the caller	Process exists

#### 6.5.4.1 Returned Message

If the return value is nonzero the returned message buffer is owned by the caller and the message is of type **sc\_procPathGetMsgReply\_t** and the message ID is **SC\_PROCPATHGETMSG\_REPLY**. The message data contains the 0 terminated ASCII name string of the process. The message is defined in the **sciopta.msg** include file.

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

## 6 Application Programming Interface

### 6.5.5 sc\_procPrioGet

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

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

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

Parameter	Description	
pid		
	Value	Description
	<pid>	Process ID.
	SC_CURRENT_PID	Current process ID (process ID of the caller).

Return Value	Condition
Priority of any given process	Parameter <b>pid</b> was any process.
Priority of the callers process	Parameter <b>pid</b> was SC_CURRENT_PID.

## 6 Application Programming Interface

### 6.5.6 sc\_procPrioSet

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

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

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

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

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

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

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

Return Value	
none	

## 6.5.7 sc\_procSchedLock

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

SCIOPTA maintains a scheduler lock counter. If the counter is 0 scheduling is activated. Each time a process calls [sc\\_procSchedLock](#) the counter will be incremented.

Interrupts are not blocked if the scheduler is blocked by [sc\\_procSchedLock](#).

### Syntax

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

Parameter	Description
none	

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

## 6.5.8 sc\_procSchedUnlock

This system call will unlock the scheduler.

SCIOPTA maintains a scheduler lock counter. Each time a process calls [sc\\_procSchedUnlock](#) the counter will be decremented. If the counter reaches a value of 0 the SCIOPTA scheduler is called and activated. The ready process with the highest priority will be swapped in.

It is illegal to unlock a not blocked scheduler.

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

Parameter	Description
none	

Return Value	
none	

## 6.5.9 sc\_procSliceGet

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

The time slice is the period of time between calls to the timer process in ticks.

```

sc_ticks_t sc_procSliceGet (
    sc_pid_t    pid
);
    
```

Parameter	Description	
pid		
	Value	Description
	<pid>	Any timer process ID.
	SC_CURRENT_PID	Current timer process ID (timer process ID of the caller).

Return Value	Condition
Period of time between calls to any given timer process in ticks	Parameter <b>pid</b> was any process.
Period of time between calls to the callers timer process in ticks	Parameter <b>pid</b> was SC_CURRENT_PID.



## 6.5.10 sc\_procSliceSet

This call is used to set the time slice of a timer process.

The modified time slice will become active after the current time slice expired or if the timer gets started.

It can only be activated after the old time slice has elapsed.

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

Parameter	Description	
pid		
	Value	Description
	<pid>	Any timer process ID.
	SC_CURRENT_PID	Current timer process ID (timer process ID of the caller).
slice	New period of time between calls to the timer process in ticks.	

Return Value	
none	

## 6 Application Programming Interface

### 6.5.11 sc\_procStart

This system call will start a prioritized or timer process.

SCIOPTA maintains a start/stop counter per process. If the counter is >0 the process is stopped. Each time a process calls [sc\\_procStart](#) the counter will be decremented. If the counter has reached the value of 0 the process will start.

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

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

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

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

Parameter	Description
pid	Process ID of the process to start.

Return Value	
none	

## 6 Application Programming Interface

### 6.5.12 sc\_procStop

This system call will stop a prioritized or timer process.

SCIOPTA maintains a start/stop counter per process. If the counter is >0 the process is stopped. Each time a process calls [sc\\_procStop](#) the counter will be incremented.

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

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

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

Parameter	Description
pid	Process ID of the process to stop.

Return Value	
none	

## 6.5.13 sc\_procVectorGet

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

The interrupt vector will only be returned if [sc\\_procVectorGet](#) is called from an interrupt process.

It is not an error to call [sc\\_procVectorGet](#) from other process types but the return value will be meaningless.

```
int sc_procVectorGet (void);
```

Parameter	Description
none	

Return Value	Condition
Interrupt vector	If called within an interrupt process.

## 6.5.14 sc\_procYield

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

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

Parameter	Description
none	

Return Value	
none	

## 6.6 Process Create/Kill System Calls

### 6.6.1 sc\_procPrioCreate

This system call is used to request the kernel daemon to create a prioritized process. The standard kernel daemon (**sc\_kerneld**) needs to be defined and started at system configuration. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about kernel daemons.

The process will be created within the callers module.

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

```
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
);
```

Parameter	Description	
<b>name</b>	Pointer to the name of the prioritized process to create.  The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore.	
<b>entry</b>	Function pointer to the process function. This is the address where the created process will start execution.	
<b>stacksize</b>	Stacksize of the created process in bytes. The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize.	
<b>slice</b>	Must be set to 0 (reserved for later use).	
<b>prio</b>	The priority of the process which can be from 0 to 31. 0 is the highest priority.	
<b>state</b>	Process state after creation.	
	<b>Value</b>	<b>Description</b>
	SC_PDB_STATE_RUN	The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes.
	SC_PDB_STATE_STP	The process is stopped. Use the <a href="#">sc_procStart</a> system call to start the process.
<b>plid</b>	Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated.	

Return Value	
Process ID of the created prioritized process	

6.6.2 sc\_procIntCreate

This system call is used to request the kernel daemon to create an interrupt process. The standard kernel daemon (**sc\_kerneld**) needs to be defined and started at system configuration. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about kernel daemons. The interrupt process will be of type **Sciopta**. Interrupt processes of type **Sciopta** are handled by the kernel and may use (not blocking) system calls.

The process will be created within the callers module.

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

```

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

Parameter	Description
<b>name</b>	Pointer to the name of the interrupt process to create. The name is represented by a ASCII character string terminated by 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore.
<b>entry</b>	Function pointer to the process function. This is the address where the created process will start execution.
<b>stacksize</b>	Stacksize of the created process in bytes. The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize.
<b>vector</b>	Interrupt vector connected to the created interrupt process. This is CPU-dependent.
<b>prio</b>	Must be set to 0 (reserved for later use).
<b>state</b>	Must be set to 1 (reserved for later use).
<b>plid</b>	Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated.

Return Value	
Process ID of the created interrupt process	

## 6.6.3 sc\_procTimCreate

This system call is used to request the kernel daemon to create a timer process. The standard kernel daemon (**sc\_kerneld**) needs to be defined and started at system configuration. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about kernel daemons.

The process will be created within the callers module.

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

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

Parameter	Description	
<b>name</b>	Pointer to the name of the timer process to create.  The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore.	
<b>entry</b>	Function pointer to the process function. This is the address where the created process will start execution.	
<b>stacksize</b>	Stacksize of the created process in bytes. The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize.	
<b>period</b>	Period of time between calls to the timer process in ticks.	
<b>initdelay</b>	Initial delay in ticks before the first time call to the timer process.	
<b>state</b>	Process state after creation.	
	<b>Value</b>	<b>Description</b>
	SC_PDB_STATE_RUN	The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes.
	SC_PDB_STATE_STP	The process is stopped. Use the <a href="#">sc_procStart</a> system call to start the process.
<b>plid</b>	Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated.	

Return Value	
Process ID of the created timer process	



## 6 Application Programming Interface

### 6.6.4 sc\_procKill

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

The standard kernel daemon (**sc\_kerneld**) needs to be defined and started at system configuration. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about kernel daemons.

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

If a cleaning-up is executed (depending on the **flag** parameter) all message buffers owned by the process will be returned to the message pool. If an observe is active on that process the observe messages will be sent to the observing processes. The [sc\\_procKill](#) system calls returns before the cleaning work begins. A significant time can elapse before a possible observe message is posted.

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

Parameter	Description	
<b>pid</b>	Process ID of the process to be killed.	
<b>flag</b>		
	Value	Description
	0	A cleaning up will be executed.
	SC_PROCKILL_KILL	No cleaning up will be requested.

Return Value	
<b>none</b>	

6.7 Process Daemon System Calls

6.7.1 `sc_procDaemonRegister`

This system call is used to register a process daemon.

The process daemon manages process names in a SCIOPTA system. If a process calls [sc\\_procIdGet](#) the kernel will send a `sc_procIdGet` message to the process daemon. The process daemon will search the process name list and return the corresponding process ID to the kernel if found.

There can only be one process daemon per SCIOPTA system.

The standard process daemon `sc_procd` is included in the SCIOPTA kernel. This process daemon needs to be defined and started at system configuration as a static process. Please consult chapter [7.9.1 “Process Daemon” on page 7-21](#) for more information about process daemon.

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

Parameter	Description
none	

Return Value	Condition
0	Process daemon was successfully installed.
!=0	System call fails and the process daemon could not be installed.

## 6.7.2 sc\_procDaemonUnregister

This call is used by a process daemon to unregister.

The name list of the daemon will be removed and messages still owned by the daemon will be freed.

A statically installed process daemon cannot be unregistered. Please consult chapter [7.9.1 “Process Daemon” on page 7-21](#) for more information about process daemon.

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

Parameter	Description
none	

Return Value	
none	

## 6.8 Process Observation System Calls

### 6.8.1 sc\_procObserve

This system call is used to supervise a process.

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

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

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

Parameter	Description
msgptr	Pointer to the message which will be returned if the supervised process disappears. The message must be of the following type: <pre>struct err_msg {     sc_msgid_t      id;     sc_errcode_t    error;     /* user defined data */ };</pre>
pid	Process ID of the process which will be supervised.

Return Value	
none	

## 6.8.2 sc\_procUnobserve

This system call is used to cancel an installed supervision of a process.  
 The message given by the [sc\\_procObserve](#) system call will be freed by the kernel.

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

Parameter	Description
pid	Process ID of the process which is supervised.
observer	Process ID of the observer process.

Return Value	
none	

## 6 Application Programming Interface

### 6.9 Process Variables System Calls

#### 6.9.1 sc\_procVarInit

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

The user needs to allocate a message with the size of

```
sizeof (sc_local_t) *size
```

The pointer to the allocated message needs to be included as parameter in [sc\\_procVarInit](#).

Please consult chapter [5.6 “Process Variables” on page 5-13](#) for more information about process variables.

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

Parameter	Description
<b>varpool</b>	Pointer to the message buffer holding the process variables.
<b>size</b>	Maximum number of process variables + 1.

Return Value	
<b>none</b>	

6.9.2 sc\_procVarSet

This system call is used to define or modify a process variable.  
 Please consult chapter [5.6 “Process Variables” on page 5-13](#) for more information about process variables.

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

Parameter	Description
tag	User defined tag for the process variable.
value	Value of the process variable.

Return Value	Condition
0	System call fails and the process variable could not be defined or modified.
!=0	Process variable was successfully defined or modified.

## 6.9.3 sc\_procVarGet

This system call is used to read a process variable.

Please consult chapter [5.6 “Process Variables” on page 5-13](#) for more information about process variables.

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

Parameter	Description
tag	User defined tag of the process variable which was set by the <a href="#">sc_procVarSet</a> call.
value	Pointer to the variable where the process variable will be stored.

Return Value	Condition
0	System call fails and the process variable could not be read.
!=0	Process variable was successfully read.



## 6.9.4 sc\_procVarDel

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

Please consult chapter [5.6 “Process Variables” on page 5-13](#) for more information about process variables.

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

Parameter	Description
tag	User defined tag of the process variable which was set by the <a href="#">sc_procVarSet</a> call.

Return Value	Condition
0	System call fails and the process variable could not be removed.
!=0	Process variable was successfully removed.

## 6.9.5 sc\_procVarRm

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

Please consult chapter [5.6 “Process Variables” on page 5-13](#) for more information about process variables.

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

Parameter	Description
none	

Return Value	
Pointer to the message buffer holding the process variables	

## 6 Application Programming Interface

### 6.10 Module Status System Calls

#### 6.10.1 `sc_moduleIdGet`

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

In contrast to the call [sc\\_procIdGet](#), you can just give the name as parameter and not a path.

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

Parameter	Description
<b>name</b>	Pointer the 0 terminated name string of the module or zero for current module.

Return Value	Condition
Module ID	Module name was found.
Current module ID (module ID of the caller)Parameter name is NULL.	Parameter <b>name</b> is NULL.
SC_ILLEGAL_MID	Module name was not found.

## 6.10.2 sc\_moduleNameGet

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

The name will be returned as a 0 terminated string.

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

Parameter	Description
mid	Module ID of which the name is requested.

Return Value	Condition
Name string of the module	Module was found.
0	Module was not found.

## 6.10.3 sc\_moduleInfo

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

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

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

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

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

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

Parameter	Description
<b>mid</b>	Module ID of which the control block data will be returned.
<b>info</b>	Pointer to a local structure of a module control block.

Return Value	Condition
<b>1</b> The <b>info</b> structure (see module.h) is filled with valid data	Module was found.
<b>0</b>	Module was not found.

### 6.11 Module Create/Kill System Calls

#### 6.11.1 sc\_moduleCreate

This system call is used to request the kernel daemon to create a module. The standard kernel daemon (**sc\_kerneld**) needs to be defined and started at system configuration. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about kernel daemons.

SCIOPTA processes can be grouped into modules to improve system structure.

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

Each module has also a priority which can range between 0 (highest) to 31 (lowest) priority. 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:

$$\text{Effective Priority} = \text{Module Priority} + \text{Process Priority}$$

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

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

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

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

Please consult chapter [5.4 “Modules” on page 5-10](#) for more information about SCIOPTA modules.

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

Parameter	Description
<b>name</b>	Pointer to the name of the module to create.  The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore.
<b>init</b>	Function pointer to the init process function. This is the address where the init process of the module will start execution.
<b>stacksize</b>	Stacksize of the INIT process in bytes.

Parameter	Description																
<b>moduleprio</b>	The priority of the module which can be from 0 to 31. 0 is the highest priority.																
<b>start</b>	Start address of the module in RAM.																
<b>size</b>	<p>Size of the module in bytes (RAM). The minimum module size can be calculated according to the following formula (bytes):</p> $\text{size\_mod} = p * 128 + \text{stack} + \text{pools} + \text{mcb} + \text{initsize}$ <p>where:</p> <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td><b>p</b></td><td>Number of static processes.</td></tr> <tr> <td><b>stack</b></td><td>Sum of stack sizes of all static processes.</td></tr> <tr> <td><b>pools</b></td><td>Sum of sizes of all message pools.</td></tr> <tr> <td><b>mcb</b></td><td>Module control block (see below)</td></tr> <tr> <td><b>initsize</b></td><td>Code (see parameter <b>initsize</b>) below</td></tr> </table> <p>The size of the module control block (mcb) can be calculated according to the following formula (bytes):</p> $\text{size\_mcb} = 96 + \text{friends} + \text{hooks} * 4$ <p>where:</p> <table> <tr> <td><b>friends</b></td><td>if friends are not used: 0 if friends are used: 16 bytes</td></tr> <tr> <td><b>hooks</b></td><td>Number of hooks configured</td></tr> </table> <p>Please consult chapter <a href="#">16.9.2 “Configuring Hooks” on page 16-10</a> for information about hook settings and chapter <a href="#">5.4.1 “SCIOPTA Module Friend Concept” on page 5-10</a> for information about friend settings.</p>	Value	Description	<b>p</b>	Number of static processes.	<b>stack</b>	Sum of stack sizes of all static processes.	<b>pools</b>	Sum of sizes of all message pools.	<b>mcb</b>	Module control block (see below)	<b>initsize</b>	Code (see parameter <b>initsize</b> ) below	<b>friends</b>	if friends are not used: 0 if friends are used: 16 bytes	<b>hooks</b>	Number of hooks configured
Value	Description																
<b>p</b>	Number of static processes.																
<b>stack</b>	Sum of stack sizes of all static processes.																
<b>pools</b>	Sum of sizes of all message pools.																
<b>mcb</b>	Module control block (see below)																
<b>initsize</b>	Code (see parameter <b>initsize</b> ) below																
<b>friends</b>	if friends are not used: 0 if friends are used: 16 bytes																
<b>hooks</b>	Number of hooks configured																
<b>size</b>	Size of the initialized data.																
<b>max_pools</b>	Maximum number of pools in the module. The kernel will not allow to create more pools inside the module than stated here. Maximum value is 128.																
<b>max_procs</b>	Maximum number of processes in the module. The kernel will not allow to create more pools inside the module than stated here. Maximum value is 16384.																

Return Value	
Module ID	

## 6 Application Programming Interface

### 6.11.2 sc\_moduleKill

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

The standard kernel daemon (**sc\_kerneld**) needs to be defined and started at system configuration. Please consult chapter [7.9.2 “Kernel Daemon” on page 7-22](#) for more information about kernel daemons.

All processes and pools in the module will be killed and removed.

The system call will return when the whole kill process is done.

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

Parameter	Description	
<b>mid</b>	Module ID of the module to be killed and removed.	
<b>flag</b>		
	Value	Description
	0	A cleaning up will be executed.
	SC_MODULEKILL_KILL	No cleaning up will be requested.

Return Value	
<b>none</b>	



## 6 Application Programming Interface

### 6.12 Module Friendship System Calls

#### 6.12.1 `sc_moduleFriendAdd`

This system call is used to add a module to the friends of the caller. The caller accept the module in parameter **mid** as friend. The module is entered in the friend set of the caller.

In SCIOPTA each module can select friend modules. This has mainly consequences on whether messages will be copied or not.

Please consult chapter [5.4.1 “SCIOPTA Module Friend Concept” on page 5-10](#) for more information about the SCIOPTA module friend concept.

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

Parameter	Description
<b>mid</b>	Module ID of the new friend to add.

Return Value	
<b>none</b>	

## 6.12.2 sc\_moduleFriendAll

This system call is used to define all existing modules in a system as friend.

In SCIOPTA each module can select friend modules. This has mainly consequences on whether messages will be copied or not.

Please consult chapter [5.4.1 “SCIOPTA Module Friend Concept” on page 5-10](#) for more information about the SCIOPTA module friend concept.

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

Parameter	Description
none	

Return Value	
none	

## 6 Application Programming Interface

### 6.12.3 sc\_moduleFriendGet

This system call is used to inform the caller if a module is a friend. The caller will be informed if the module in parameter **mid** is a friend. It returns if the module is member of the friend set of the caller.

In SCIOPTA each module can select friend modules. This has mainly consequences on whether messages will be copied or not.

Please consult chapter [5.4.1 “SCIOPTA Module Friend Concept” on page 5-10](#) for more information about the SCIOPTA module friend concept.

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

Parameter	Description
<b>mid</b>	The ID of the module which will be checked if it is a friend or not.

Return Value	Condition
<b>0</b>	Module is not a friend (not included in the friend set)
<b>!=0</b>	Module is a friend (included in the friend set)

## 6 Application Programming Interface

### 6.12.4 `sc_moduleFriendNone`

This system call is used to remove all modules as friends of the caller. All modules are removed in the friend set of the caller.

In SCIOPTA each module can select friend modules. This has mainly consequences on whether messages will be copied or not.

Please consult chapter [5.4.1 “SCIOPTA Module Friend Concept” on page 5-10](#) for more information about the SCIOPTA module friend concept.

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

Parameter	Description
none	

Return Value	
none	

## 6.12.5 sc\_moduleFriendRm

This system call is used to remove a module of the friends of the caller. The caller removes the module in parameter **mid** as friend. The module is removed in the friend set of the caller.

In SCIOPTA each module can select friend modules. This has mainly consequences on whether messages will be copied or not.

Please consult chapter [5.4.1 “SCIOPTA Module Friend Concept” on page 5-10](#) for more information about the SCIOPTA module friend concept.

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

Parameter	Description
mid	The module ID of the old friend to remove.

Return Value	
none	

## 6.13 Timing System Calls

### 6.13.1 sc\_sleep

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

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

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

```
void sc_sleep (
    sc_ticks_t    tmo
);
```

Parameter	Description
tmo	Number of system ticks to wait.

Return Value	
none	

## 6 Application Programming Interface

### 6.13.2 sc\_tick

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

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

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

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

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

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

Parameter	Description
none	

Return Value	
none	

6.13.3 sc\_tickGet

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

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

Parameter	Description
none	

Return Value	
Number of system ticks of the kernel tick counter	



## 6 Application Programming Interface

### 6.13.4 sc\_tickLength

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

```
__u32 sc_tickLength (
    __u32 t1
);
```

Parameter	Description	
t1		
	Value	Description
	0	The current tick length will just be returned without modifying it.
	<tick_length>	The tick length in micro seconds.

Return Value	
Tick length in microseconds	

## 6.13.5 sc\_tickMs2Tick

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

```

sc_time_t sc_tickMs2Tick (
    __u32      ms
);
    
```

Parameter	Description
ms	Time in milliseconds.

Return Value	
Time in system ticks.	

## 6.13.6 sc\_tickTick2Ms

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

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

Parameter	Description
t	Time in system ticks.

Return Value	
Time in milliseconds.	

## 6.14 Time-out Server System Calls

### 6.14.1 sc\_tmoAdd

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

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

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

The registered time-out can be cancelled by the [sc\\_tmoRm](#) call before the time-out has expired. This system call returns the time-out ID which could be used later to cancel the time-out.

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

Parameter	Description
tmo	Number of system tick after which the message will be sent back by the kernel.
msgptr	Pointer to the message which will be sent back by the kernel after the elapsed time.

Return Value	
Time-out ID	

## 6 Application Programming Interface

### 6.14.2 `sc_tmoRm`

This system call is used to remove a time-out before it is expired.

If the process has already received the time-out message and the user still tries to cancel the time-out with the [sc\\_tmoRm](#) call, the kernel will generate a fatal error.

After the call the value of the time-out id is zero.

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

Parameter	Description
<b>id</b>	Time-out ID which was given when the time-out was registered by the <a href="#">sc_tmoAdd</a> call.

Return Value	
Pointer to the time-out message which was defined at registering it by the <a href="#">sc_tmoAdd</a> call.	

## 6.15 Trigger System Calls

### 6.15.1 sc\_trigger

This system call is used to activate a process trigger.

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

Please consult chapter [5.5 “Trigger” on page 5-12](#) for more information about SCIOPTA trigger.

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

Parameter	Description
pid	ID of the process which trigger will be activated.

Return Value	
Incremented trigger value	

## 6.15.2 sc\_triggerWait

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

The [sc\\_triggerWait](#) call will wait on the trigger of the callers process. The trigger value will be decremented by the value **dec** of the parameters.

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

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

Please consult chapter [5.5 “Trigger” on page 5-12](#) for more information about SCIOPTA trigger.

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

Parameter	Description
<b>dec</b>	The number to decrease the process trigger value.
<b>tmo</b>	Maximum time-out value in system ticks which the process will wait on its trigger. A value of tmo == 0 or SC_NO_TMO will generate a system error.

Return Value	Condition
<b>0</b>	The trigger was decremented but the calling process was not suspended as the trigger value is still positive or zero.
<b>1</b>	Trigger occurred and the trigger value became negative
<b>-1</b>	Time-out expired
The return values are defined in the file trigger.h located at: <install_folder>\sciopta\<version>\exp\krm\include\	

## 6.15.3 sc\_triggerValueGet

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

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

Please consult chapter [5.5 “Trigger” on page 5-12](#) for more information about SCIOPTA trigger.

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

Parameter	Description
pid	ID of the process which trigger is returned.

Return Value	
Trigger value	



## 6.15.4 sc\_triggerValueSet

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

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

Please consult chapter [5.5 “Trigger” on page 5-12](#) for more information about SCIOPTA trigger.

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

Parameter	Description
value	The new trigger value which will be stored.

Return Value	
none	

6.16 CONNECTOR System Calls

6.16.1 sc\_connectorRegister

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

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

Please consult chapter [5.9 “Distributed Systems” on page 5-16](#) for more information about connector processes.

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

Parameter	Description	
defaultConn		
	Value	Description
	0	The caller becomes a connector process. The name of the process correspond to the name of the target.
	!=0	The caller becomes the default connector for the system.

Return Value	
Specific connector ID which is used to define the process ID for distributed processes.	

## 6.16.2 sc\_connectorUnregister

This system call is used to remove a registered connector process.

The caller becomes a normal prioritized process.

Please consult chapter [5.9 “Distributed Systems” on page 5-16](#) for more information about connector processes.

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

Parameter	Description
none	

Return Value	
none	

6.17 CRC System Calls

6.17.1 sc\_miscCrc

This function calculates a 16 bit CRC over a specified memory range.  
 The start value of the CRC is 0xFFFF.

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

Parameter	Description
data	Pointer to the memory range.
len	Number of bytes.

Return Value	
The 16 bit CRC value.	

## 6.17.2 sc\_miscCrcContd

This function calculates a 16 bit CRC over an additional memory range.

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

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

Parameter	Description
<b>data</b>	Pointer to the memory range.
<b>len</b>	Number of bytes.
<b>start</b>	CRC start value.

Return Value	
The 16 bit CRC value.	

6.18 Error System Calls

6.18.1 sc\_miscError

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

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

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

Please consult chapter [7.11 “Error Hook” on page 7-24](#) and [5.11 “Error Handling” on page 5-19](#) for more information about error handling and error hooks.

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

Parameter	Description	
err	User defined error code.	
	Bits 0, 1 and 2 of err defines if the error is fatal or not and if it is generated by the system, a module or a process.	
	Bit 0	
	Value	Description
	1	Fatal error on system level (SC_ERR_TARGET_FATAL).
	0	No fatal error on system level.
	Bit 1	
	Value	Description
	1	Fatal error on module level (SC_ERR_MODULE_FATAL).
	0	No fatal error on module level.
	Bit 2	
	Value	Description
	1	Fatal error on process level (SC_ERR_PROCESS_FATAL).
	0	No fatal error on process level.
misc	Additional data to pass to the error hook.	

Return Value	
none	

## 6.18.2 sc\_miscErrnoGet

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

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

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

Please consult chapter [7.11 “Error Hook” on page 7-24](#) and [5.11 “Error Handling” on page 5-19](#) for more information about error handling and error hooks.

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

Parameter	Description
none	

Return Value	
Read error code.	

## 6.18.3 sc\_miscErrnoSet

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

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

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

Please consult chapter [7.11 “Error Hook” on page 7-24](#) and [5.11 “Error Handling” on page 5-19](#) for more information about error handling and error hooks.

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

Parameter	Description
err	User defined error code.

Return Value	
none	



## 6.19 Hook Registering System Calls

### 6.19.1 sc\_miscErrorHookRegister

This system call will register an error hook

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

Please consult chapter [7.11 “Error Hook” on page 7-24](#) and [5.11 “Error Handling” on page 5-19](#) for more information about error handling and error hooks.

If the error hook is registered from within the system module it is registered as a global error hook. In this case the error hook registering will be done in the start hook.

If the error hook is registered from within a module which is not the system module it will be registered as a module error hook.

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

Parameter	Description
<b>newhook</b>	Function pointer to the hook. <b>0</b> will remove and unregister the hook.

Return Value	Condition
Function pointer to the previous error hook.	Error hook was registered.
<b>0</b>	No error hook was registered.

## 6 Application Programming Interface

### 6.19.2 sc\_msgHookRegister

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

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

If [sc\\_msgHookRegister](#) is called from within a module a module message hook will be registered.

A global message hook will be registered when [sc\\_msgHookRegister](#) is called from the start hook function which is called before SCIOPTA is initialized.

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

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

Parameter	Description	
<b>type</b>		
	<b>Value</b>	<b>Description</b>
	SC_SET_MSGTX_HOOK	Registers a message transmit hook. Every time a message is sent, this hook will be called.
	SC_SET_MSGRX_HOOK	Registers a message receive hook. Every time a message is received, this hook will be called.
<b>newhook</b>	Function pointer to the hook. A zero value will remove and unregister the hook.	

Return Value	Condition
Function pointer to the previous message hook.	Message hook was registered.
0	No message hook was registered.

## 6.19.3 sc\_poolHookRegister

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

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

If [sc\\_poolHookRegister](#) is called from within a module a module pool hook will be registered.

A global pool hook will be registered when [sc\\_poolHookRegister](#) is called from the start hook function which is called before SCIOPTA is initialized.

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

### Syntax

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

Parameter	Description	
type		
	Value	Description
	SC_SET_POOLCREATE_HOOK	Registers a pool create hook. Every time a pool is created, this hook will be called.
	SC_SET_POOLKILL_HOOK	Registers a pool kill hook. Every time a pool is killed, this hook will be called.
newhook	Function pointer to the hook. A zero value will remove and unregister the hook.	

Return Value	Condition
Function pointer to the previous pool create hook.	Pool create hook was registered.
0	No pool create hook was registered.

## 6 Application Programming Interface

### 6.19.4 sc\_procHookRegister

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

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

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

If [sc\\_procHookRegister](#) is called from within a module a module process hook will be registered.

A global process hook will be registered when [sc\\_procHookRegister](#) is called from the start hook function which is called before SCIOPTA is initialized.

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

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

Parameter	Description	
<b>type</b>		
	Value	Description
	SC_SET_PROCCREATE_HOOK	Registers a process create hook. Every time a process is created, this hook will be called.
	SC_SET_PROCKILL_HOOK	Registers a process kill hook. Every time a process is killed, this hook will be called.
	SC_SET_PROCSWAP_HOOK	Registers a process swap hook. Every time a process swap is initiated by the kernel, this hook will be called.
<b>newhook</b>	Function pointer to the hook. A zero value will remove and unregister the hook.	

Return Value	Condition
Function pointer to the previous process hook.	Process hook was registered.
0	No process hook was registered.

## 7 Application Programming

### 7.1 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.

### 7.2 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.

### 7.3 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.

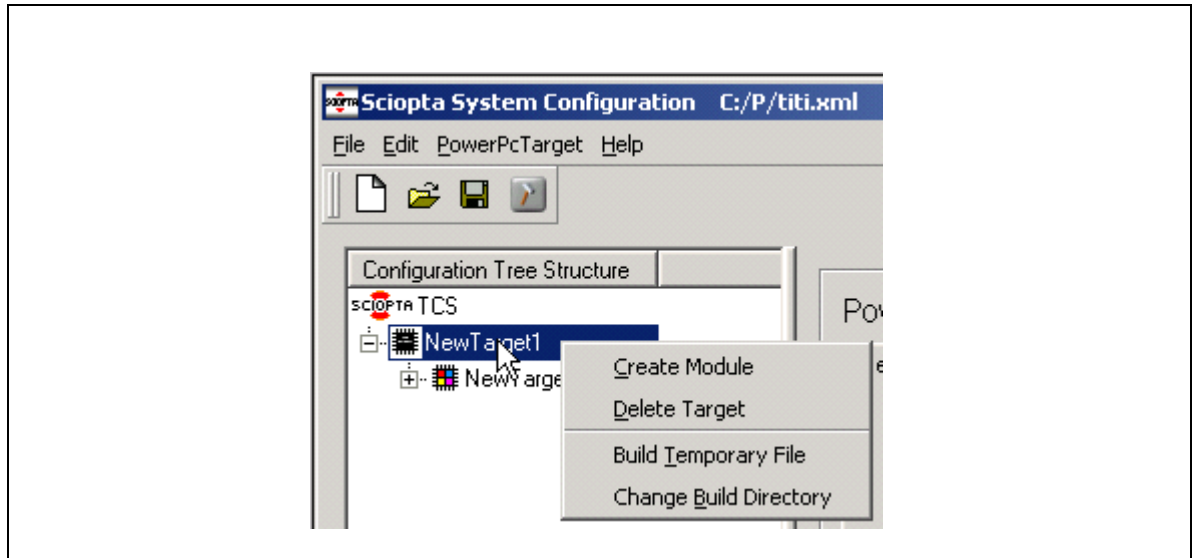


Figure 7-1: Module Creation by SCONF

Please consult the [16.10 “Creating Modules” on page 16-12](#) for more information about module creation by the SCONF tool.

Another way is to create modules dynamically by the [sc\\_moduleCreate](#) system call.

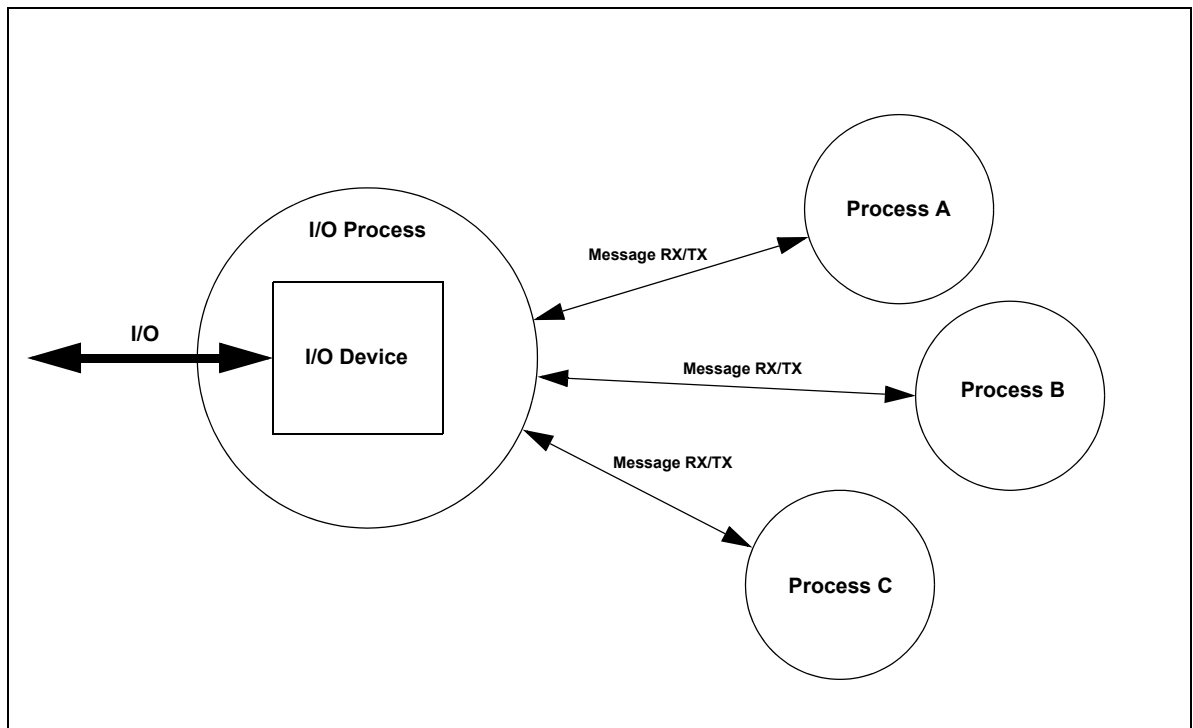
```
sc_moduleid_t sc_moduleCreate(const char *name,          /* name of module */
                              void (*init)(void),       /* init-process */
                              sc_bufsize_t stacksize,   /* stack-size of init process */
                              sc_prio_t prio,         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

Figure 7-2: Module Creation by [sc\\_moduleCreate](#) System Call

### 7.4 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 7-3: Resource Encapsulation**

Please consult the SCIOPTA ARM Cortex - Device Driver, User's Guide for more information.

### 7.5 Processes

#### 7.5.1 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 [sc\\_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.

#### 7.5.2 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.



## 7 Application Programming

### 7.5.2.1 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.

### 7.5.2.2 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 */

    }
}
```

### 7.5.3 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.

## 7 Application Programming

### 7.5.3.1 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	Description												
<b>proc_name</b>	Name of the interrupt process.												
<b>irq_src</b>	<p>Interrupt source. Depending of this value the interrupt process can execute different code for different interrupt sources. The following values are defined:</p> <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>The interrupt process is activated by a real hardware interrupt.</td></tr> <tr> <td>1</td><td>The interrupt process is activated by a message sent to the interrupt process.</td></tr> <tr> <td>2</td><td>The interrupt process is activated by a trigger event.</td></tr> <tr> <td>-1</td><td>The interrupt process is activated when the process is created. This allows the interrupt process to execute some initialization code.</td></tr> <tr> <td>-2</td><td>The interrupt process is activated when the process is killed. This allows the interrupt process to execute some exit code.</td></tr> </table> <p><b>irq_src</b> is of type int.</p>	Value	Description	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.
Value	Description												
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.												

### 7.5.3.2 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 */

    }
}
```

### 7.5.4 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 [7.5.3 “Interrupt Processes” on page 7-6](#) are also valid for timer processes.

#### 7.5.4.1 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 [7.5.3.1 “Interrupt Process Declaration Syntax” on page 7-7](#).

#### 7.5.4.2 Timer Process Template

You can use the same template as for interrupt processes (see chapter [7.5.3.2 “Interrupt Process Template” on page 7-8](#)).

### 7.5.5 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.

#### 7.5.5.1 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 is called **Module Hook** and 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_PROCURINTCREATESTATIC);

    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;
}
```

### 7.5.5.2 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 */
}
```

### 7.5.6 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.

#### 7.5.6.1 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.

#### 7.5.6.2 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.

#### 7.5.6.3 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.



### 7.6 Addressing Processes

#### 7.6.1 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.

#### 7.6.2 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);
```

#### 7.6.3 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\\_procPrioCreate](#), [sc\\_procIntCreate](#) and [sc\\_procTimCreate](#) will also return the process IDs which can be used for further addressing.

### 7.7 Interprocess Communication

#### 7.7.1 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.

#### 7.7.2 SCIOPTA Messages

##### 7.7.2.1 Description

Please consult chapter [5.3 “Messages” on page 5-7](#) 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**.

##### 7.7.2.2 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

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

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

## 7 Application Programming

### 7.7.2.5 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	Description
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.

### 7.7.2.6 Message Number (ID) organization

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

All message IDs greater than 0x8000000 are reserved for SCIOPTA internal modules and functions and may not be used by the application. These messages are defined in the file `defines.h`. Please consult this file for managing and organizing the message IDs of your application.

<code>defines.h</code>	System wide constant definitions.
------------------------	-----------------------------------

File location: `<installation_folder>\sciopta\<version>\include\ossys\`

### 7.7.2.7 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_NO_TMO);    /* Get process ID */
                                                /* for process display */

    for (;;)
    {
        msg = msgAlloc(sizeof (char_msg_t), CHAR_MSG, SC_DEAFULT_POOL, 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 */
    }
}
```

### 7.7.3 SCIOPTA Trigger

#### 7.7.3.1 Description

Please consult chapter [5.5 “Trigger” on page 5-12](#) for an 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 too slow.

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

#### 7.7.3.2 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 */
        .
        .
    }
}
```

### 7.8 SCIOPTA Memory Manager - Message Pools

#### 7.8.1 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 [5.4 “Modules” on page 5-10](#) 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.

#### 7.8.2 Message Pool size

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

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

**Standard (32-Bit) kernel:**

$$\text{pool\_cb} = 68 + n * 20 + \text{stat} * n * 20$$

where:

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

### 7.8.3 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.

### 7.8.4 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.

### 7.8.5 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 administration block. Please consult chapter [5.3 “Messages” on page 5-7](#) for more information about SCIOPTA messages. The size of the message header is 32 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. The size of the end-mark is 4 bytes.



## 7 Application Programming

### 7.9 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.

#### 7.9.1 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.

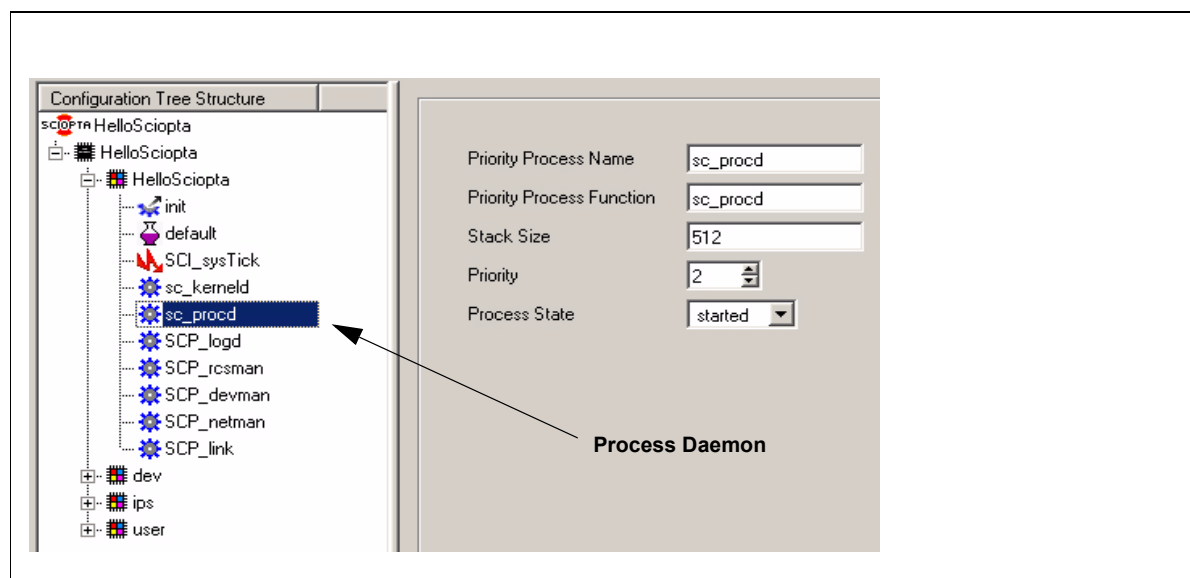


Figure 7-4: Process Daemon Declaration in SCONF

### 7.9.2 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.

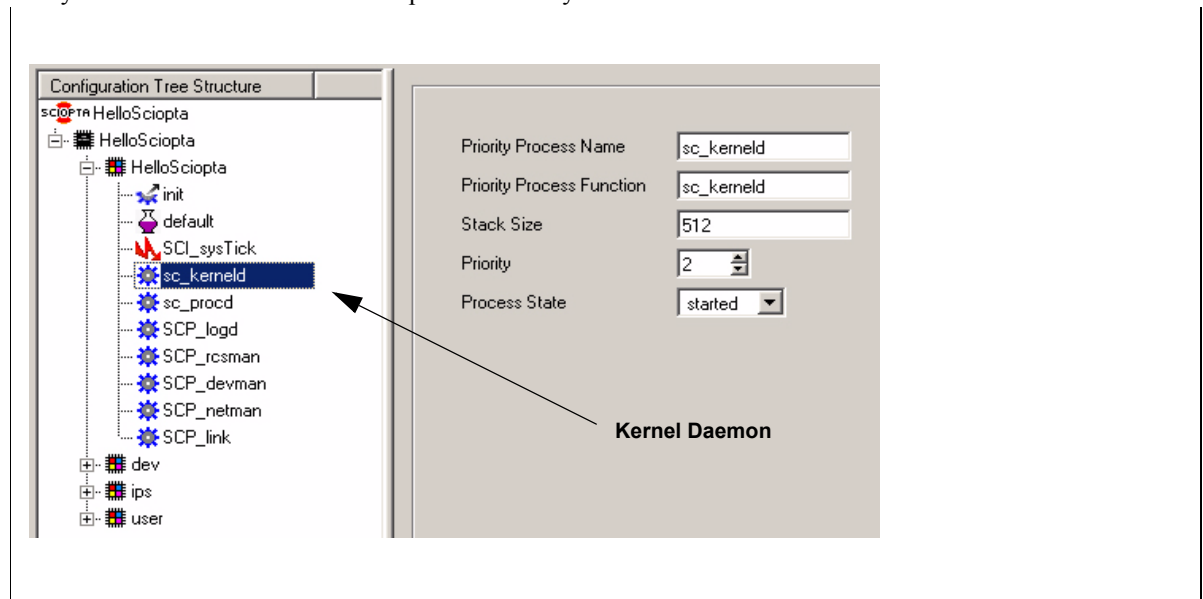


Figure 7-5: Kernel Daemon Declaration in SCONF

## 7.10 Trap Interface

Trap Interface is actually not yet supported.

In a typical monolithic SCIOPTA systems the kernel functions are directly called.

In more complex dynamic systems using load modules or MMU protected modular systems the kernel functions cannot be accessed any more by direct calls. SCIOPTA offers a trap interface. In such systems you need to assemble the CPU dependent file syscall.S which can be found in the \machine sub-directory of the include directory.

syscall.S	SCIOPTA kernel trap interface trampoline functions for GNU GCC.
-----------	---

File location: <install\_folder>\sciopta\<version>\include\machine\arm\

This file includes another file with the same name containing CPU independent trap interface functions.

syscall.S	SCIOPTA kernel trap interface trampoline functions, not CPU dependent.
-----------	--

File location: <install\_folder>\sciopta\<version>\include\machine\

7.11 Error Hook

7.11.1 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.

7.11.2 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**).

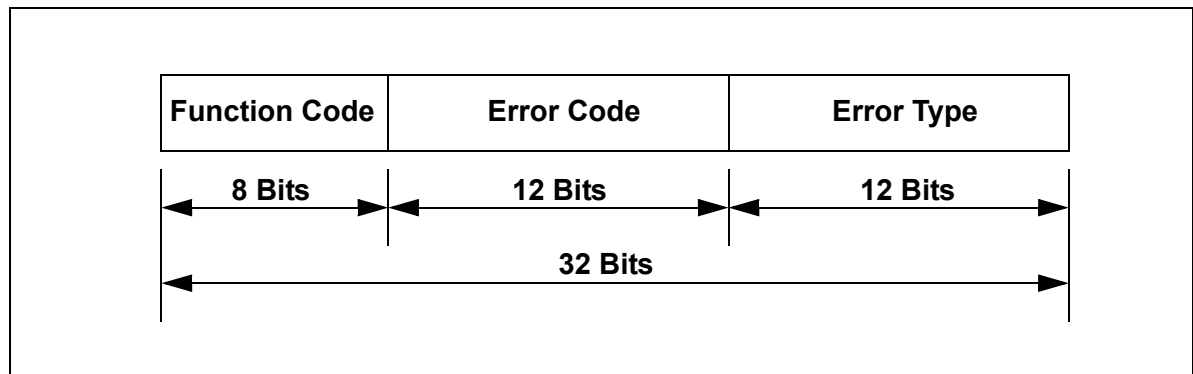


Figure 7-6: 32-bit Error Word (Parameter: **errcode**)

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 chapter [15.4 “Error Codes” on page 15-4](#) for a detailed description of the error codes.

## 7 Application Programming

### 7.11.3 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. In this case the error hook registering will be done in the start hook.

If the error hook is registered from within a module which is not the system module it will be registered as a module error hook.

### 7.11.4 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	Description	
<b>errcode</b>	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.	
<b>extra</b>	Gives additional information depending on the error code.	
<b>user</b>	<b>Value</b>	<b>Description</b>
	user != 0	User error.
	user == 0	System error.
<b>pcb</b>	Pointer to the Process Control Block PCB of the process which generated the error.	

Return Value	Description
!= 0	Continue/resume if error was not fatal. The kernel first kills the module or process.
== 0	Jumps to infinite loop at label SC_ERROR.

## 7.11.5 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)&0xffff) <= SC_MAXERR &&
        (err>>24) <= SC_MAXFUNC )
    {
        kprintf(0,"Function: %s\nError: %s\n",
            func_txt[err>>24],
            err_txt[(err>>12)&0xffff]);
    }
    return 0;
}
#endif
```

## 7.11.6 Error Hooks Return Behaviour

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

Global Error Hook		Module Error Hook		Error Type	Action
exists	return value	exists	return value	Module Error Fatal	
No	-	No	-	X	Endless loop.
Yes	0	No	-	Yes	Endless loop.
				No	Endless loop.
	1		-	Yes	Kill module and swap out.
				No	Return & continue.
Yes	0	Yes	0	Yes	Endless loop.
				No	Endless loop.
	1		0	Yes	Kill module and swap out.
				No	Return & continue.
	0		1	Yes	Kill module and swap out.
				No	Return & continue.
	1		1	Yes	Kill module and swap out.
				No	Return & continue.
No	-	Yes	0	Yes	Endless loop.
				No	Endless loop.
	-		1	Yes	Kill module and swap out.
				No	Return & continue.

### 7.12 SCIOPTA ARM Cortex Exception Handling

#### 7.12.1 Introduction

Exception handling in SCIOPTA ARM Cortex is identically for all Cortex CPUs. Vectorized interrupts are implemented.

#### 7.12.2 Interrupt Handler

The interrupt handler is included in the kernel. The interrupt vector table is supplied SCIOPTA (cortexm3\_vector.S) and no user intervention is needed.

#### 7.12.3 SCIOPTA Interrupt Process

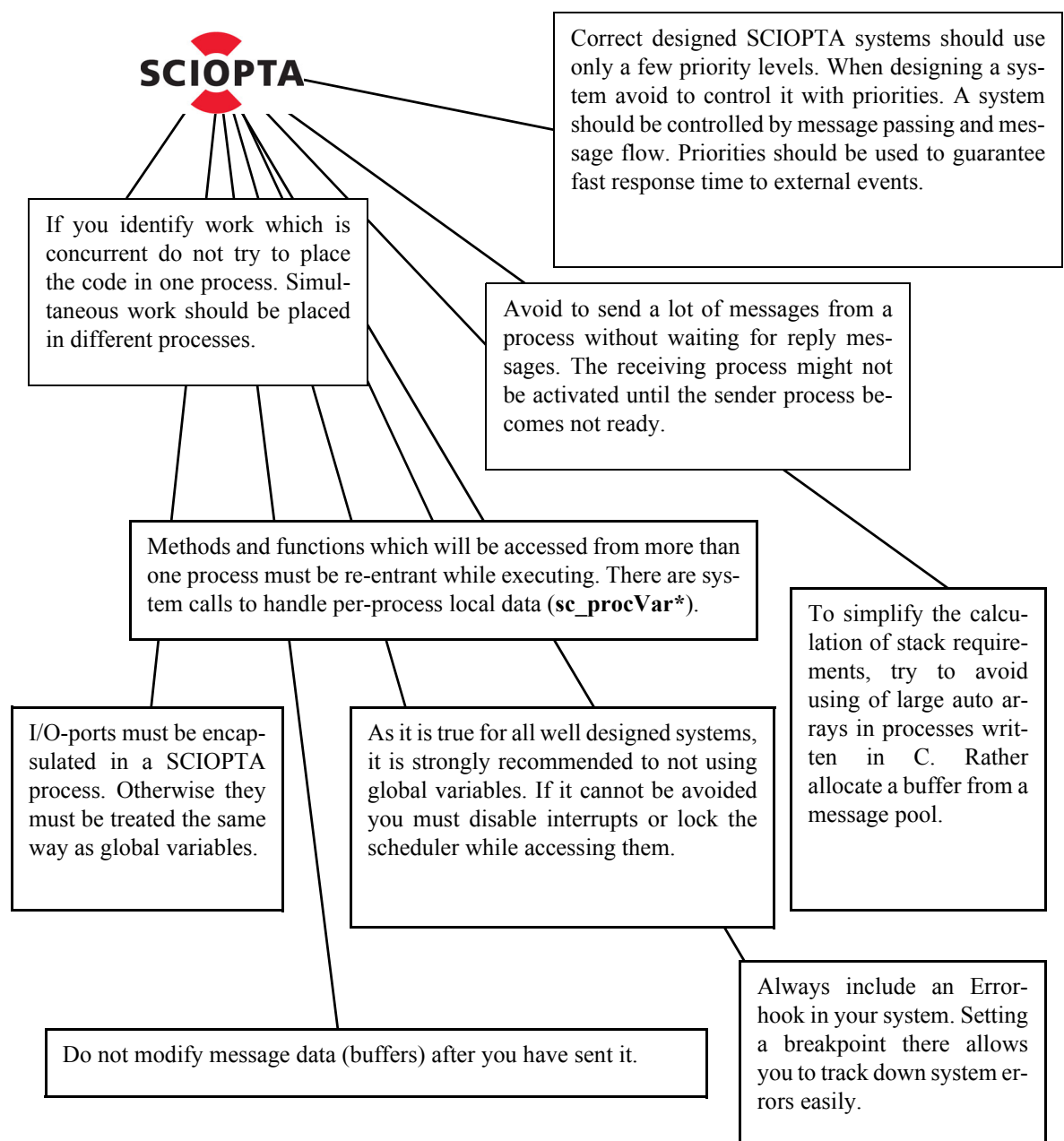
SCIOPTA interrupt processes shall do the usual interrupt service work (including clearing possible interrupt flags and acknowledge) and are statically defined by the **SCONF** utility or dynamically created by the [sc\\_procIntCreate](#) system call. The vector declared in SCONF or by [sc\\_procIntCreate](#) must be the correct interrupt vector of you Cortex interrupt.

You can find examples of interrupt processes in the board support packages (e.g. systick.c for the tick timer).



### 7.13 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 powerful and strong design technology. Nevertheless the SCIOPTA user has to follow some rules to design message based systems efficiently and easy to debug.



## 8 System and Application Configuration

### 8.1 Introduction

Besides the kernel configuration described in chapter [10 “Kernel Configuration” on page 10-1](#) which defines system characteristics, modules, processes and message pools, you need to setup and initialize your board and your application.

System and application configuration is done in some specific files such as `resethook.S` and `cstartup.S`.

Other system and application configuration functions for the system module such as start hooks, system module hooks and hook registration is usually done in a specific file called `system.c` which can be found in the SCIOPTA examples deliveries.

System and application configuration functions for other modules (module hooks and hook registration) is usually done in specific files having the same name as the module (`dev.c`, `ips.c` etc.) which can also be found in the SCIOPTA examples deliveries.

### 8.2 System Start

#### 8.2.1 Start Sequence

After a system hardware reset the following sequence will be executed:

1. The kernel calls the function `reset_hook`.
2. The kernel performs some internal initialization.
3. The kernel calls the C startup function `cstartup`.
4. The kernel calls the function `start_hook`.
5. The kernel calls the function `TargetSetup`. The code of this function is automatically generated by the **SCONF** configuration utility and included in the file `sconf.c`. `TargetSetup` creates the system module.
6. The kernel calls the dispatcher.
7. The first process (init process of the system module) is swapped in.

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

8. The INIT process of the system module creates all static modules, processes and pools.
9. The INIT process of the system module calls the system module start function.
10. The process priority of the INIT process of the system module is set to 32 and loops for ever.
11. The INIT Process of each created static module calls the user module hook of each module.
12. The process priority of the INIT process of each created static module is set to 32 and loops for ever.
13. The process with the highest system priority will be swapped-in and executed.

## 8.2.2 Reset Hook

### Description

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

The reset hook is a function in the file **boardSetup.c** and 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.

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

Please consult also chapter [9 “Board Support Packages” on page 9-1](#) for more information.

### Source Files

boardSetup.c	Board setup.
--------------	--------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\src\

### Syntax

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

Parameter	Description
none	

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

## 8 System and Application Configuration

### 8.2.3 C Startup

After a cold start the kernel will call the C startup function. The C startup function is written in assembler and has the name `cstartup`. It initializes the C system and replaces the library C startup function. C startup functions are compiler specific. Please note that this file is not needed for **IAR EW**.

#### Source Files

<code>cortexm3_cstartup.S</code>	C startup assembler source for GNU GCC
----------------------------------	--

File location: `<install_folder>\sciopta\<version>\bsp\arm\src\gnu\`

<code>cortexm3_cstartup.s</code>	C startup assembler source for ARM RealView
----------------------------------	---

File location: `<install_folder>\sciopta\<version>\bsp\arm\src\arm\`

### 8.2.4 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. The start hook would also be the right place to include the registration of the system error hook (see chapter [7.11.3 “Error Hook Registering” on page 7-25](#)) and other kernel hooks.

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

<code>system.c</code>	System configuration file including hooks and other setup code.
-----------------------	---

File location: `<installation_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\`

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

#### Prototype

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

### 8.2.5 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.

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

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

**8.2.6 Module Hook**

Please consult chapter [5.4 “Modules” on page 5-10](#) for general information about SCIOPTA modules.

**8.2.6.1 System Module Hook**

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

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

system.c	System configuration file including hooks and other setup code.
----------	---

File location: `<installation_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\`

**8.2.6.2 User Modules Hooks**

All other user modules have also own individual User Module Hooks. These are functions with the same name of the respective defined and configured modules which will be called by the INIT Process of each respective module.

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

## 9 Board Support Packages

### 9.1 Introduction

In this chapter all officially supported board support packages are listed. After a short description of the board the included files, processes, hooks and processes are listed. Information about configuration and board set-up are also given.

The board support functions and drivers for other SCIOPTA products such for IPS Internet Protocols, DRUID System Level Debugger, File System etc. are described in the manuals for these products.

Please consult also the SCIOPTA - Device Driver, User Guide for information about the SCIOPTA device driver concept.

### 9.2 General System Functions

System functions which do not depend on a specific board and a specific CPU and are common for SCIOPTA systems. Files for external chips and controllers.

#### Project Files

winIDEA_gnu.ind	iSYSTEM winIDEA indirection file for GNU GCC
-----------------	--

File location: <install\_folder>\sciopta\<version>\bsp\common\include\

### 9.3 ARM Cortex Family System Functions

Setup and driver descriptions which do not depend on a specific board and are common for all ARM based processors and controllers.

#### Project Files

module.ld	Linker script: Module sections (common to all boards) for GNU GCC
-----------	---

File location: <install\_folder>\sciopta\<version>\bsp\arm\include\

#### Include Files

memcpy.h	Header for fast memcpy
----------	------------------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\include\

### Source Files

cortexm3_cstartup.S	Cortex M3 C startup assembler source for GNU GCC.
cortexm3_exception.S	Cortex M3 exception handler for GNU GCC.
cortexm3_vector.S	Cortex M3 vector table for GNU GCC.

File location: <install\_folder>\sciopta\<version>\bsp\arm\src\gnu\

cortexm3_cstartup.s	Cortex M3 C startup assembler source for ARM RealView.
cortexm3_exception.s	Cortex M3 Exception handler for ARM RealView.
cortexm3_vector.s	Cortex M3 vector table for ARM RealView.

File location: <install\_folder>\sciopta\<version>\bsp\arm\src\arm\

cortexm3_exception.s	Cortex M3 exception handler for IAR Embedded Workbench Version 5.
cortexm3_exception.s79	Cortex M3 exception handler for IAR Embedded Workbench Version 4.
cortexm3_vector.s	Cortex M3 vector table for IAR Embedded Workbench Version 5.
cortexm3_vector.s79	Cortex M3 vector table for IAR Embedded Workbench Version 5.

File location: <install\_folder>\sciopta\<version>\bsp\arm\src\iar\

9.4 STM32 System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with STM32 based processors.

9.4.1 General STM32 Functions and Definitions

For the STMicroelectronics STM32 Family CPU we are using the STMicroelectronics STM32 Firmware Library **STM32F10xFWLib**.

Include Files

cortexm3_macro.h	Header file for cortexm3_macro.s.
stm32f10x_xxxx.h	Cortex STM32 definitions (26 files).

File location: <stm32\_firmware\_library\_install\_folder>\FWLib\library\inc\

Source Files

stm32f10x_xxxx.c	Cortex STM32 firmware functions (26 files).
------------------	---

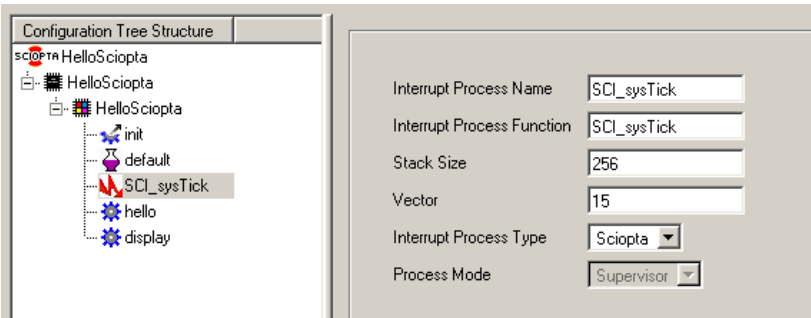
File location: <stm32\_firmware\_library\_install\_folder>\FWLib\library\src\



### 9.4.2 STM32 System Tick Driver

#### 9.4.2.1 STM32 System Tick Interrupt Process

You need to declare the Interrupt Process `SCI_sysTick` in the SCIOPTA `SCONF` Utility:



#### Source Files

<code>systick.c</code>	System timer setup.
------------------------	---------------------

File location: `<install_folder>\sciopta\<version>\bsp\arm\stm32\src\`

### 9.4.3 STM32 UART Functions

This is not a full featured serial driver. Just some basic UART routines for the STM32 controllers are supplied to be used for system logging and `printf` debugging.

#### Include Files

<code>simple_uart.h</code>	Simple UART routines.
----------------------------	-----------------------

File location: `<install_folder>\sciopta\<version>\bsp\arm\stm32\include\`

#### Source Files

<code>simple_uart.c</code>	Simple polling uart function for <code>printf</code> debugging or logging.
----------------------------	--

File location: `<install_folder>\sciopta\<version>\bsp\arm\stm32\src\`

### 9.5 Stellaris System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with Stellaris based processors.

#### 9.5.1 General Stellaris Functions and Definitions

For the Luminary Stellaris Cortex-M3 Family CPU we are using the Luminary Stellaris Cortex-M3 Family Driver Library.

##### Include Files

lm3sxxx.h	Cortex Stellaris definitions (138 files).
-----------	---

File location: <stellaris\_firmware\_library\_install\_folder>\inc\

*.h	Driver include files (25 files).
-----	----------------------------------

File location: <stellaris\_firmware\_library\_install\_folder>\src\

##### Source Files

*.c	Stellaris Cortex-M3 drivers (20 files).
-----	---

File location: <stellaris\_firmware\_library\_install\_folder>\src\

##### Project Files

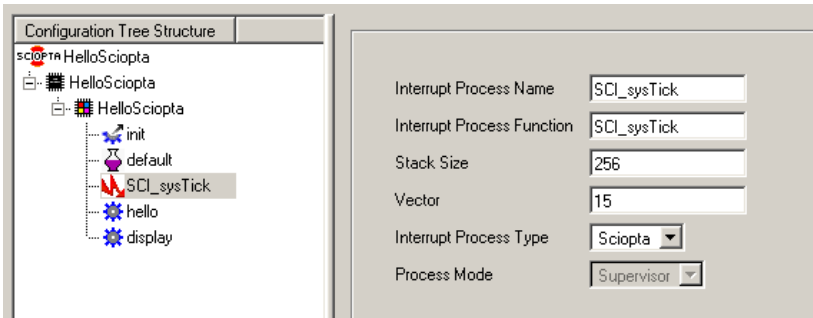
*.*	Stellaris Cortex-M3 project and makefiles for different development environments.
-----	---

File location: <stellaris\_firmware\_library\_install\_folder>\src\

### 9.5.2 Stellaris System Tick Driver

#### 9.5.2.1 Stellaris System Tick Interrupt Process

You need to declare the Interrupt Process `SCI_sysTick` in the SCIOPTA `SCONF` Utility:



#### Source Files

<code>systick.c</code>	System timer setup.
------------------------	---------------------

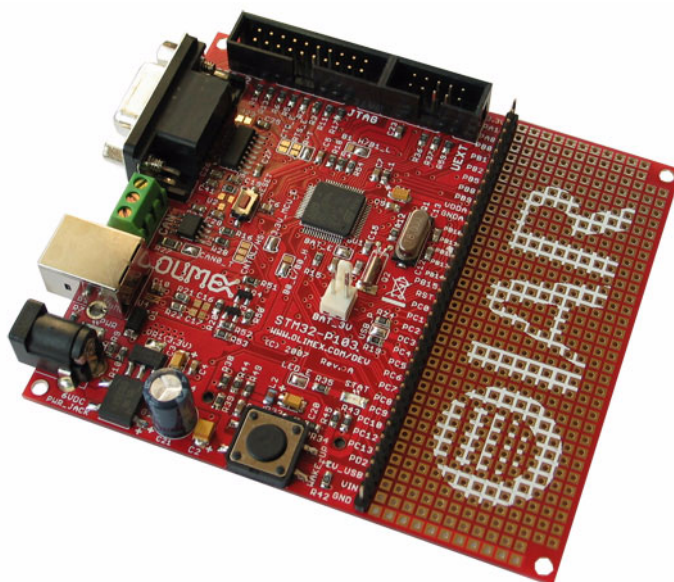
File location: `<install_folder>\sciopta\<version>\bsp\arm\stellaris\src\`

### 9.6 Olimex STM32-P103 Board

#### 9.6.1 Description

The ARM Cortex-M3 processor is the latest generation of ARM processors for embedded systems. It has been developed to provide a low-cost platform that meets the needs of MCU implementation, with a reduced pin count and low-power consumption, while delivering outstanding computational performance and an advanced system response to interrupts. The ARM Cortex-M3 32-bit RISC processor features exceptional code-efficiency, delivering the high-performance expected from an ARM core in the memory size usually associated with 8- and 16-bit devices.

The STM32F103 Performance Line family has an embedded ARM core and is therefore compatible with all ARM tools and software. It combines the high performance ARM Cortex-M3 CPU with an extensive range of peripheral functions and enhanced I/O capabilities. STM32-P103 is good start-up board for learning the new ST Cortex-M3 based microcontrollers STM32F103RBT6. It have RS232 and both USB and CAN , the prototype area with all microcontrollers port near it allow customer easy to implement his own schematics and add-ons.



### Features:

- MCU: STM32F103RBT6 ARM 32 bit CORTEX M3™ with 128K Bytes Program Flash, 20K Bytes RAM, USB, CAN, x2 I2C, x2 ADC 12 bit, x3 UART, x2 SPI, x3 TIMERS, up to 72Mhz operation
- standard JTAG connector with ARM 2x10 pin layout for programming/debugging with ARM-JTAG
- USB connector
- CAN driver and connector
- RS232 driver and connector
- UEXT connector which allow different modules to be connected (as MOD-MP3, MOD-NRF24LR, etc)
- SD-MMC connector
- backup battery connector
- RESET button
- status LED
- power supply LED
- on board voltage regulator 3.3V with up to 800mA current
- single power supply: takes power from USB port or power supply jack
- 8 Mhz crystal oscillator
- 32768 Hz crystal and RTC backup battery connector
- extension headers for all uC ports
- PCB: FR-4, 1.5 mm (0,062"), soldermask, silkscreen component print
- Dimensions: 100 x 90mm (3.94 x 3.5")

### 9.6.2 STM32-P103 General Board Functions and Definitions

#### Project Files

stm32-p103.ld	GCC linker script example.
stm32-p103.sct	ARM RealView linker script example.
stm32-p103.xcl	IAR EW linker script example.

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32-p103\include\

#### Include Files

config.h	Board configuration definitions.
stm32f10x_conf.h	Driver library configuration definitions.

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32-p103\include\

#### Source Files

boardSetup.c	Board setup.
--------------	--------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32-p103\src\

### 9.6.3 STM32-P103 LED Driver

Simple functions to access the STM32-P103 board LEDs.

#### Include Files

led.h	Defines for the STM32-P103 board's LED routines.
-------	--

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32-p103\include\

#### Source Files

led.c	Routines to access the LEDs on the STM32-P103 board.
-------	--

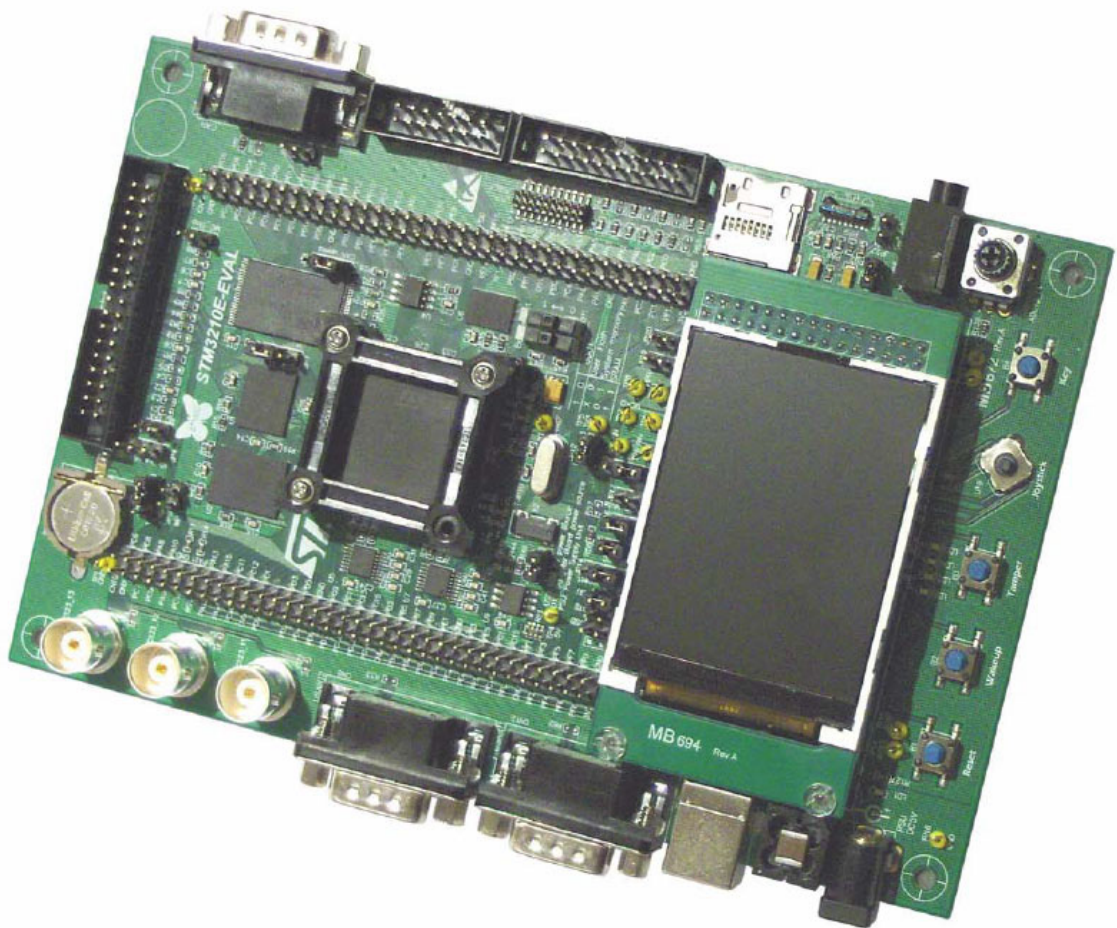
File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32-p103\src\

### 9.7 STMicroelectronics STM3210E-EVAL Evaluation Board

#### 9.7.1 Description

The STM3210E-EVAL evaluation board is designed as a complete development platform for STMicroelectronic's ARM Cortex-M3 core-based STM32F103Z microcontroller with full speed USB2.0, CAN2.0A/B compliant interface, two I2S channels, two I2C channels, five USART channels with smartcard support, three SPI channels, two DAC channels, FSMC interface, SDIO, internal 64 KB SRAM and 512 KB Flash, JTAG and SWD debug support.

The full range of hardware features on the board helps you to evaluate all peripherals (USB, motor control, CAN, MicroSD card, smartcard, USART, NOR Flash, NAND flash, SRAM) and develop your own applications. Extension headers make it possible to easily connect a daughter board or wrapping board for your specific application.



## 9 Board Support Packages

---

### Features:

- Three 5 V power supply options: power jack, USB connector or daughter board
- Boot from user Flash, system memory or SRAM
- I2S Audio DAC, stereo audio jack
- 128 Mbyte MicroSD card
- Both A and B type smartcard support
- 64 or 128 Mbit serial Flash, 512 Kx16 SRAM, 512 Mbit or 1 Gbit NAND Flash and 128 Mbit NOR Flash
- I2C/SMBus compatible serial interface temperature sensor
- Two RS-232 channels with RTS/CTS handshake support on one channel IrDA transceiver
- USB2.0 full speed connection
- CAN2.0A/B compliant connection
- Inductor motor control connector
- JTAG and trace debug support
- 240x320 TFT color LCD
- Joystick with 4-direction control and selector
- Reset, wakeup, tamper and user buttons
- 4 color LEDs
- RTC with backup battery



## 9.7.2 STM3210E-EVAL General Board Functions and Definitions

### Project Files

stm32103-eval.ld	GCC linker script example.
stm32103-eval.sct	ARM RealView linker script example.
stm32103-eval.xcl	IAR EW linker script example.

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32103-eval\include\

### Include Files

config.h	Board configuration definitions.
stm32f10x_conf.h	Driver library configuration definitions.

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32103-eval\include\

### Source Files

boardSetup.c	Board setup.
--------------	--------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32103-eval\src\

## 9.7.3 STM3210E-EVAL LED Driver

Simple functions to access the STM3210E-EVAL board LEDs.

### Include Files

led.h	Defines for the STM3210E-EVAL board's LED routines.
-------	---

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32103-eval\include\

### Source Files

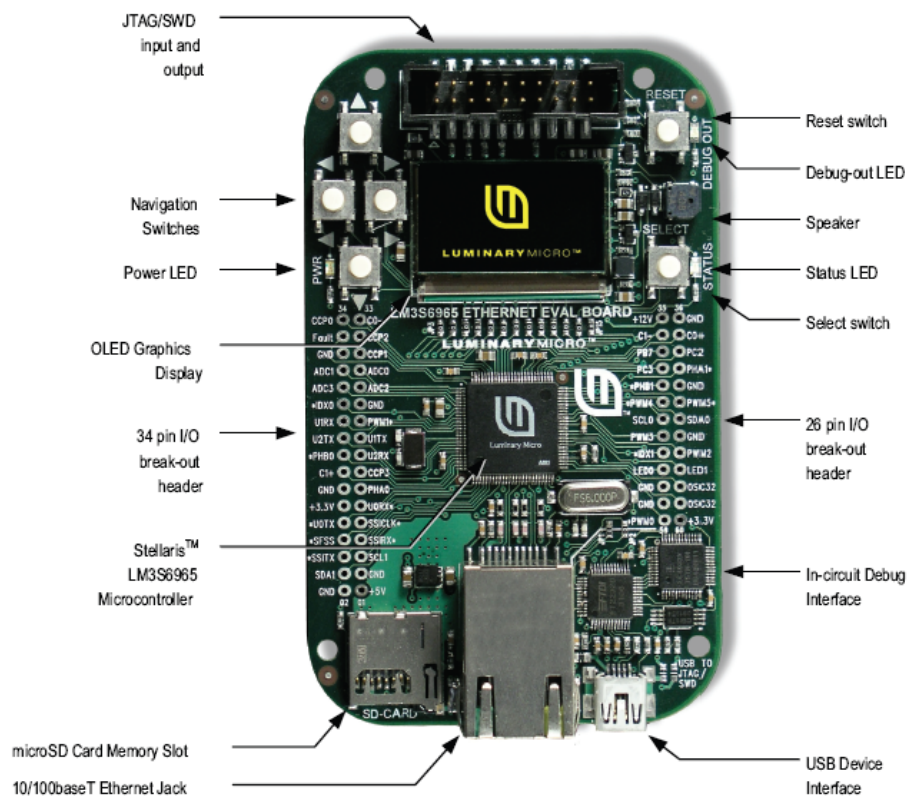
led.c	Routines to access the LEDs on the STM3210E-EVAL board.
-------	---

File location: <install\_folder>\sciopta\<version>\bsp\arm\stm32\stm32103-eval\src\

### 9.8 Luminary LM3S6965 Board

#### 9.8.1 Description

Stellaris LM3S6965 Evaluation Kits provide a compact and versatile evaluation platform for Ethernet enabled Stellaris ARM® Cortex™-M3-based microcontrollers. Each board has an In-Circuit Debug Interface (ICDI) that provides hardware debugging functionality not only for the on-board Stellaris devices, but also for any Stellaris microcontroller-based target board. The evaluation kits contain all cables, software, and documentation needed to develop and run applications for Stellaris microcontrollers easily and quickly.



## 9 Board Support Packages

---

### Features:

- LM3S6965 Evaluation Board
- Stellaris LM3S6965 microcontroller with fully integrated 10/100 (MAC+PHY) Ethernet controller
- Simple setup: USB cable provides serial communication, debugging, and power
- OLED graphics display with 128 x 64 pixel resolution and 16 shades of gray
- User LED, navigation switches, and select pushbuttons
- Magnetic speaker
- All LM3S6965 I/O available on labeled break-out pads
- Standard ARM® 20-pin JTAG debug connector with input and output modes
- MicroSD card slot
- Retracting Ethernet cable, USB cable, and JTAG cable

9.8.2 LM3S6965 General Board Functions and Definitions

Project Files

ek-lm3s6965.ld	GCC linker script example.
ek-lm3s6965.sct	ARM RealView linker script example.
ek-lm3s6965.xcl	IAR EW linker script example.

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\include\

Include Files

config.h	Board configuration definitions.
----------	----------------------------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\include\

Source Files

boardsetup.c	Board setup.
--------------	--------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\src\

resethook.S	Board setup for GNU GCC.
-------------	--------------------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\src\gnu\

resethook.s	Board setup for ARM RealView.
-------------	-------------------------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\src\arm\

resethook.s	Board setup for IAR EW.
-------------	-------------------------

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\src\iar\

9.8.3 LM3S6965 LED Driver

Simple functions to access the LM3S6965 board LEDs.

Source Files

led.c	Routines to access the LEDs on the LM3S6965 board.
-------	--

File location: <install\_folder>\sciopta\<version>\bsp\arm\stellaris\ek-lm3s6965\src\

### 9.9 Luminary® Stellaris Family Driver Library

The Luminary Micro® Stellaris® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the Stellaris family of ARM® Cortex™-M3 based microcontrollers. While they are not drivers in the pure operating system sense (that is, they do not have a common interface and do not connect into a global device driver infrastructure), they do provide a mechanism that makes it easy to use the device's peripherals.

The capabilities and organization of the drivers are governed by the following design goals:

- They are written entirely in C except where absolutely not possible.
- They demonstrate how to use the peripheral in its common mode of operation.
- They are easy to understand.
- They are reasonably efficient in terms of memory and processor usage.
- They are as self-contained as possible.
- Where possible, computations that can be performed at compile time are done there instead of at run time.
- They can be built with more than one tool chain.

Some consequences of these design goals are:

- The drivers are not necessarily as efficient as they could be (from a code size and/or execution speed point of view). While the most efficient piece of code for operating a peripheral would be written in assembly and custom tailored to the specific requirements of the application, further size optimizations of the drivers would make them more difficult to understand.
- The drivers do not support the full capabilities of the hardware. Some of the peripherals provide complex capabilities which can not be utilized by the drivers in this library, though the existing code can be used as a reference upon which to add support for the additional capabilities.
- The APIs have a means of removing all error checking code. Since the error checking is usually only useful during initial program development, it can be removed to improve code size and speed.

For many applications, the drivers can be used as is. But in some cases, the drivers will have to be enhanced or rewritten in order to meet the functionality, memory, or processing requirements of the application. If so, the existing driver can be used as a reference on how to operate the peripheral.

Please install Luminary® Stellaris Family Driver Library as described in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6.](#)

## 10 Kernel Configuration

### 10.1 Introduction

The SCIOPTA ARM Cortex kernel needs to be configured before you can build and download your application. 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.

For a detailed description of the **SCONF** configuration utility, please consult chapter [16 “SCONF Kernel Configuration Utility” on page 16-1](#).

### 10.2 System

For a SCIOPTA project it would be possible to define more than one system. You could configure a SCIOPTA ARM Cortex target system and other SCIOPTA target systems from within the same **SCONF** configuration window. But usually you will define just one target system.

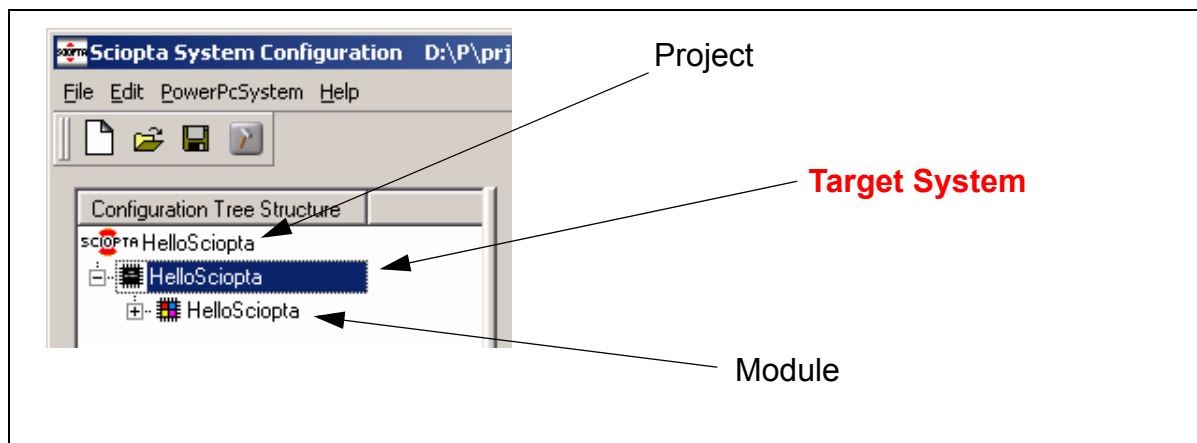


Figure 10-1: SCIOPTA System

Please consult chapter [16.9 “Configuring ARM Cortex Target Systems” on page 16-8](#) for a detailed description of the **SCONF** target system configuration and the configuration parameters.

### 10.3 Modules

Processes can be grouped into modules to improve system structure. A SCIOPTA system must have at least one module, also called module 0 or system module. The system module gets automatically the name of the system.

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.

Please consult chapter [5.4 “Modules” on page 5-10](#) for an introduction into the SCIOPTA module concept.

In chapter [7.3 “Modules” on page 7-1](#) you will find information how to use SCIOPTA modules.

Please consult chapter [16.11 “Configuring Modules” on page 16-13](#) for a detailed description of the **SCONF** module configuration and the configuration parameters.

#### 10.3.1 Small Systems

Small or simple system can be put into one module. This keeps the system and memory map on a very neat level.

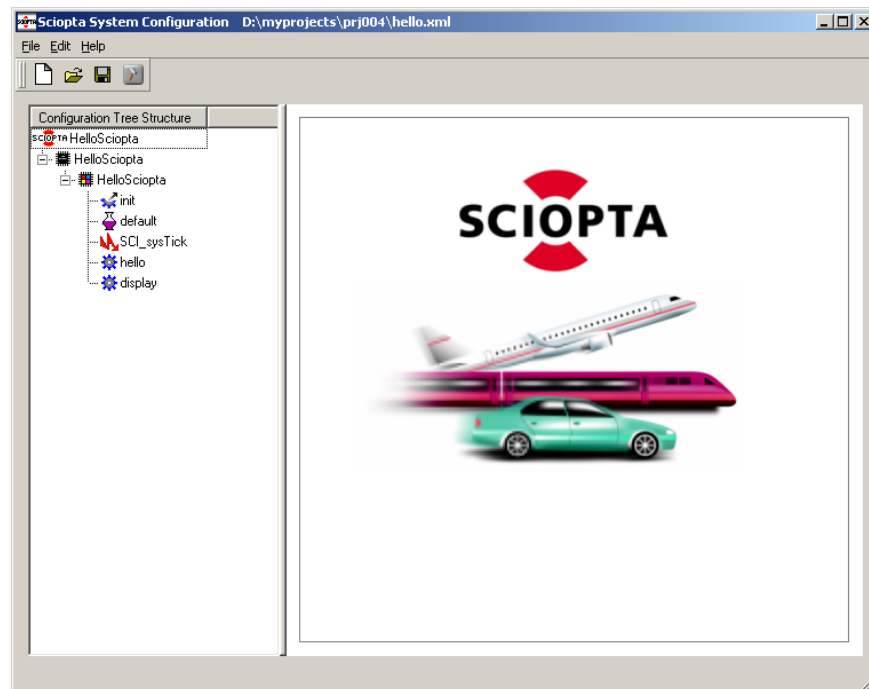


Figure 10-2: One-Module System

10.3.2 Multi-Module Systems

In larger or more complex system it is good design practice to partition the system up into more modules.

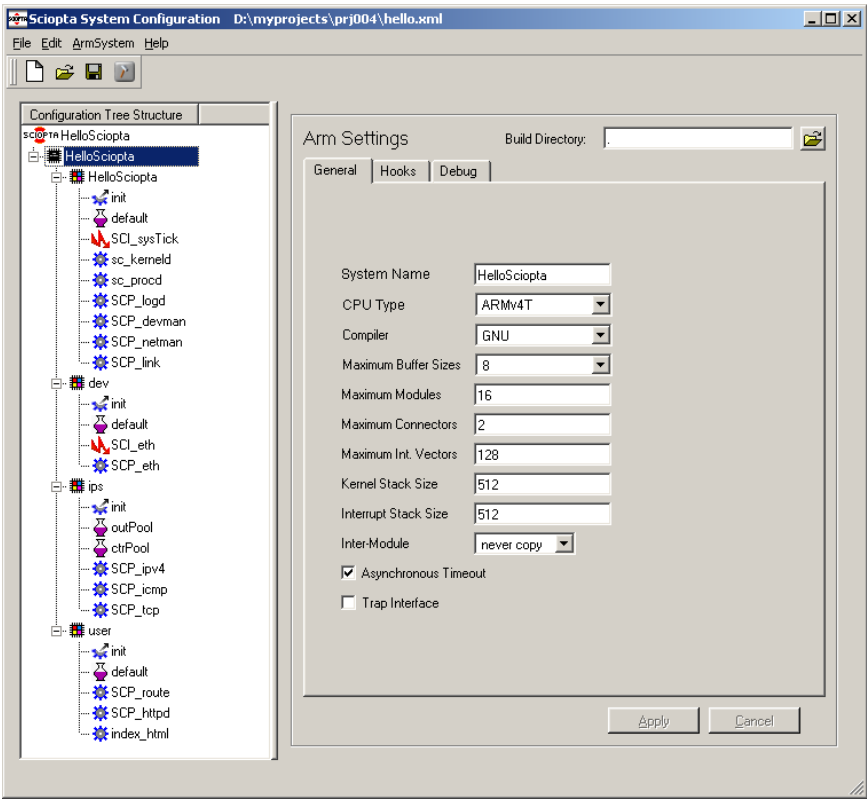


Figure 10-3: Multi-Module System

Above example system consists of four modules.

HelloSciOpta	This is the system module and contains the daemons (kernel daemon sc_kerneld, process daemon sc_procd and log daemon SCP_logd), the SDD managers (device manager SCP_devman and network device manager SCP_netman) and other system pools and system processes. The system module gets automatically the name of the system.
dev	This module holds the device driver processes and device driver pools.
ips	This example includes the SCIOPTA IPS TCP/IP stack. The processes of this communication stack are located in the ips module.
user	In this user module the application processes and pools are placed.



## 10 Kernel Configuration

### 10.4 Processes

#### 10.4.1 INIT Process

Only the stack size can be configured for the initialization process. Define a size which is big enough for the whole initialization procedure. Start to give a quite high value. You can later optimize the stack size with the help of the DRUID System Level Debugger.

Please consult chapter [5.2.4.4 “Init Process” on page 5-5](#) for an introduction into the SCIOPTA INIT Processes.

In chapter [7.5.5 “Init Process” on page 7-10](#) you will find information how to use INIT Processes.

Please consult chapter [16.13 “Configuring the Init Process” on page 16-17](#) for a detailed description of the SCONF INIT Process configuration and the configuration parameters.

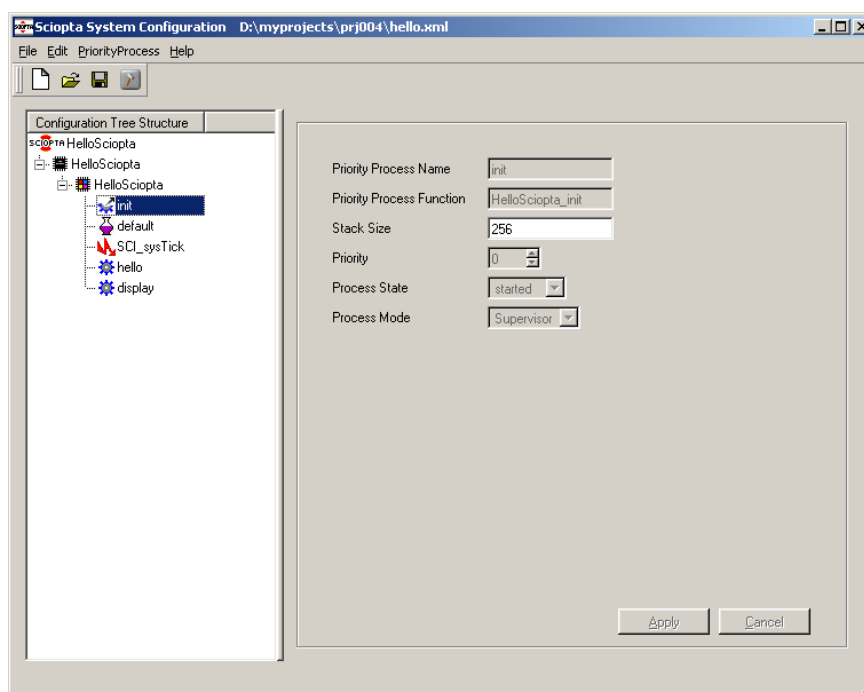


Figure 10-4: Init Process Configuration

## 10.4.2 Interrupt Processes

In a very minimal SCIOPTA system the tick timer Interrupt Process (SCI\_sysTick) is the only Interrupt Process.

Define all Interrupt Process of your device driver system and make sure that the allocated Vectors are correct. Select stack sizes which are big enough for the Interrupt Processes. Start to give a quite high value. You can later optimize the stack size with the help of the DRUID System Level Debugger.

Please consult chapter [5.2.4.2 “Interrupt Process” on page 5-4](#) for an introduction into the SCIOPTA Interrupt Processes.

In chapter [7.5.3 “Interrupt Processes” on page 7-6](#) you will find information how to use Interrupt Processes.

Please consult chapter [16.14 “Interrupt Process Configuration” on page 16-18](#) for a detailed description of the SCONF Interrupt Process Configuration and the configuration parameters.

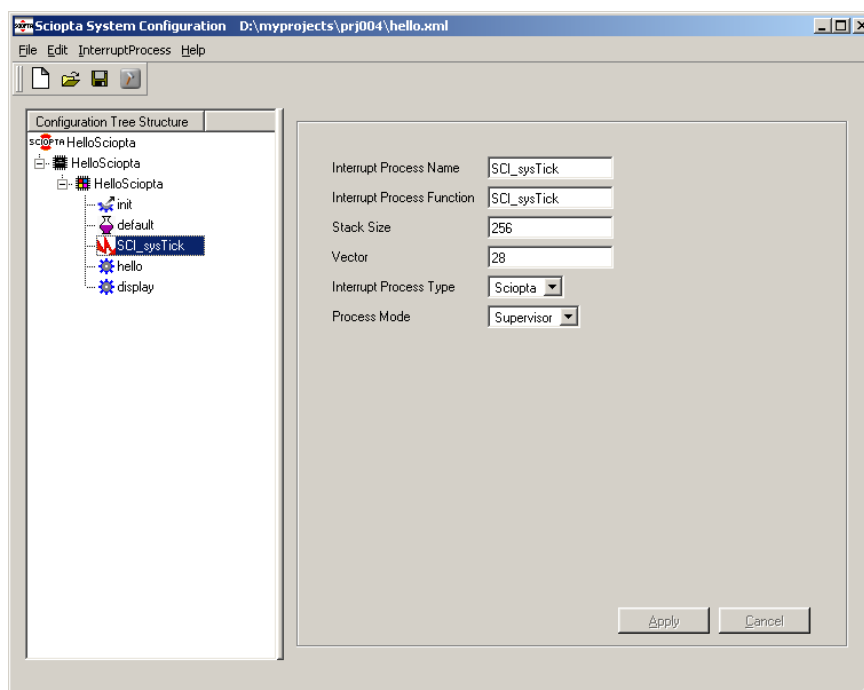


Figure 10-5: System Tick Interrupt Process Configuration

### 10.4.3 Timer Processes

Define all timer processes of your system. Select stack sizes which are big enough for the timer processes. Start to give a quite high value. You can later optimize the stack size with the help of the DRUID System Level Debugger.

Please consult chapter [5.2.4.3 “Timer Process” on page 5-4](#) for an introduction into the SCIOPTA timer processes.

In chapter [7.5.4 “Timer Process” on page 7-9](#) you will find information how to use timer processes.

Please consult chapter [16.15 “Timer Process Configuration” on page 16-20](#) for a detailed description of the SCONF timer process configuration and the configuration parameters.

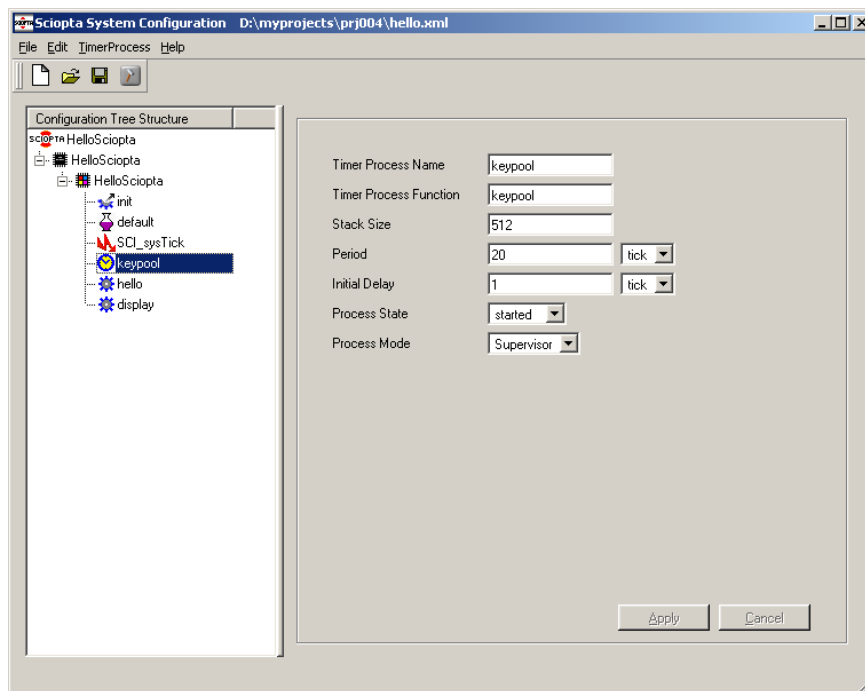


Figure 10-6: Timer Process Configuration

#### 10.4.4 Prioritized Processes

Define all prioritized processes of your system. Select stack sizes which are big enough for the processes. Start to give a quite high value. You can later optimize the stack size with the help of the DRUID System Level Debugger.

Please consult chapter [5.2.4.1 “Prioritized Process” on page 5-4](#) for an introduction into the SCIOPTA prioritized processes.

In chapter [7.5.2 “Prioritized Processes” on page 7-4](#) you will find information how to use prioritized processes.

Please consult chapter [16.16 “Prioritized Process Configuration” on page 16-22](#) for a detailed description of the SCONF prioritized process configuration and the configuration parameters.

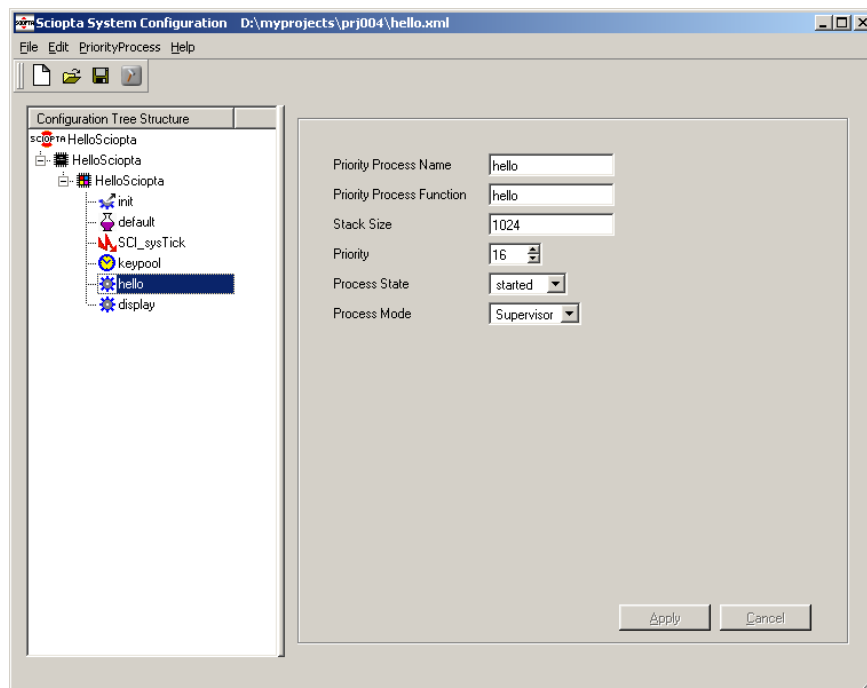


Figure 10-7: Prioritized Process Configuration

10.4.5 Message Pools

Define all pools and message sizes for your system. You could leave the default pool sizes and optimize it later with the help of the system level debugger.

Please consult chapter [5.3 “Messages” on page 5-7](#) for an introduction into the SCIOPTA messages and message pools.

In chapter 8.8 [7.8 “SCIOPTA Memory Manager - Message Pools” on page 7-19](#) you will find information how to use message pools.

Please consult chapter [16.17 “Pool Configuration” on page 16-24](#) for a detailed description of the **SCONF** prioritized process configuration and the configuration parameters.

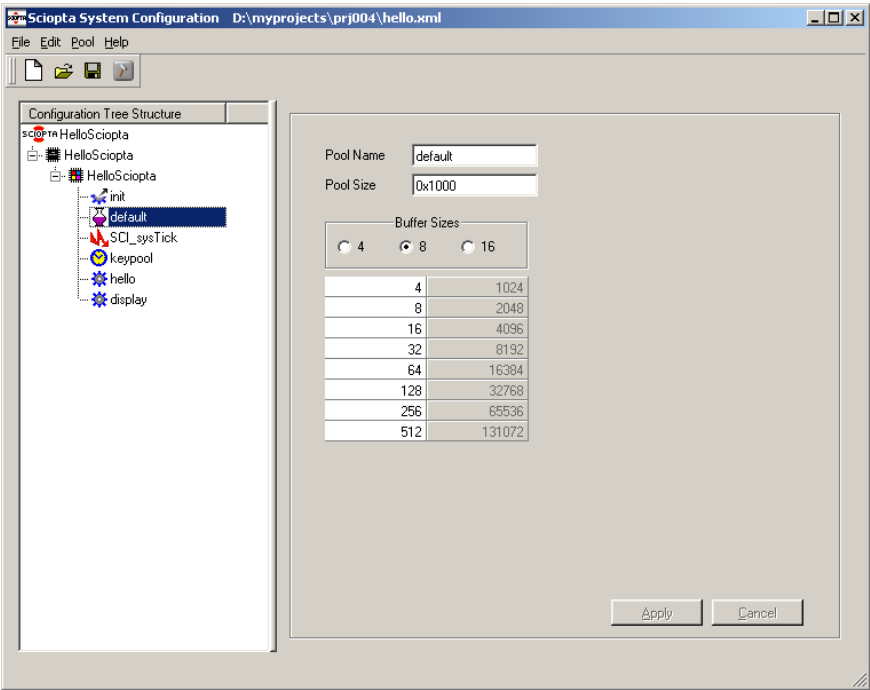


Figure 10-8: Pool Configuration

### 10.5 Generating the Configuration Files

After your configuration is correct the **SCONF** utility will generate three files `sciopta.cnf`, `sconf.h` and `sconf.c` which need to be included into your SCIOPTA project.

#### 10.5.1 Generated Kernel Configuration Files

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

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.

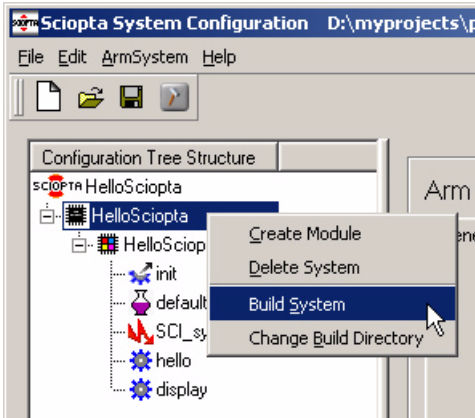


Figure 10-9: Build System

Please consult chapter [16.18 “Build” on page 16-26](#) for more information about configuration file generation.

## 10 Kernel Configuration

---

### 10.5.2 sciopta.cnf

This file contains the configured part of the kernel which will be included when the SCIOPTA kernel is assembled.

### 10.5.3 sconf.h

The file sconf.h is a header file which contains the configuration settings. sconf.h must be included by files which need to have knowledge about system configuration.

### 10.5.4 sconf.c

This is a C source file which contains the system initialization and startup code. The following example shows a very small SCIOPTA system consisting of one interrupt process, one message pool and two prioritized processes located in one single module.

#### Please Note

- Do never modify this automatically generated file sconf.c.
- Some of the system calls used in the sconf.c file are not available to the user.
- The process HelloSciopta\_init is the INIT Process of the system module HelloSciopta.
- For each module in the system such an module INIT Process with the name <module\_name>\_init will be created.
- The INIT Process of the system module creates all static message pools and processes.
- For each module a module start function called module hook will be called with the same name as the module name.
- At the end the INIT Processes will change the priority to the lowest level (32) and enter an endless loop. Priority 32 is only allowed for the INIT Processes.
- The function TargetSetup will be called by the kernel after the start\_hook has returned. TargetSetup creates the system module.
- Please consult chapter      for more information about the system start sequence.

## 11 Libraries and Include Files

### 11.1 Kernel

The kernel is delivered as a stripped and undocumented assembler source file for each supported compiler.

#### SCIOPTA Kernel

sciopta.S	SCIOPTA kernel for GNU GCC.
sciopta.s	SCIOPTA kernel for IAR EW Version 5.
sciopta.s79	SCIOPTA kernel for IAR EW Version 4.
sciopta_ads.s	SCIOPTA kernel for ARM RealView

File location: <installation\_folder>\sciopta\<version>\lib\arm\krm\

### 11.2 Kernel Libraries

For the SCIOPTA generic device driver (GDD) functions, the shell functions and the SCIOPTA utilities some prebuilt libraries are included in the delivery.

#### 11.2.1 GNU GCC Kernel Libraries

The file name of the libraries have the following format:

lib<name>\_Xt.a

File location: <installation\_folder>\sciopta\<version>\lib\arm\gnu\

##### 11.2.1.1 Optimization “X”

The libraries are delivered for three different compiler optimization. The letter **X** defines one of three compiler optimization levels.

“X” can have a value of 0,1 and 2 and defines the optimization.

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

##### 11.2.1.2 “t”

If the SCIOPTA Trap Interface is used, the libraries with the letter “t” after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.



## 11 Libraries and Include Files

### 11.2.1.3 Included SCIOPTA Kernel Libraries

Utilities	libutil_Xt.a
Generic device driver	libgdd_Xt.a
Shell	libsh_Xt.a

### 11.2.1.4 Building Kernel Libraries for GCC

The example makefiles and project files are supposing to use libraries for the generic device driver (GDD) and utility (util) modules. As described above, there are some libraries delivered for specific compiler settings.

We have included source files and makefiles which allows you to build the libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

#### Procedures to generate the libraries

- Open a Command Prompt window.
- Navigate to the route directory of the product of which you want to build the library:

<b>Generic Device Drivers</b>	<installation_folder>\sciopta\<version>\gdd\
<b>Utilities</b>	<installation_folder>\sciopta\<version>\util\

- Execute the makefile for GNU GCC for each product:  
For thumb mode: gnu-make -f Makefile.tarm delivery\_lib  
For ARM mode: gnu-make -f Makefile.arm delivery\_lib
- The libraries will be installed in the directory: <installation\_folder>\sciopta\<version>\lib\arm\gnu\

## 11 Libraries and Include Files

---

### 11.2.2 IAR Kernel Libraries

The file name of the libraries have the following format:

<name>\_Xt.r79

File location: <installation\_folder>\sciopta\<version>\lib\arm\iar\

#### 11.2.2.1 Optimization “X”

The libraries are delivered for three different compiler optimization. The letter **X** defines one of three compiler optimization levels.

“X” can have a value of 0,1 and 2 and defines the optimization.

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

#### 11.2.2.2 “t”

If the SCIOPTA Trap Interface is used, the libraries with the letter “t” after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

#### 11.2.2.3 Included SCIOPTA Kernel Libraries

Utilities	util_Xt.r79
Generic device driver	gdd_Xt.r79
Shell	sh_Xt.r79

## 11 Libraries and Include Files

---

### 11.2.2.4 Building Kernel Libraries for IAR

The example makefiles and project files are supposing to use libraries for the generic device driver (GDD) and utility (util) modules. As described above, there are some libraries delivered for specific compiler settings.

We have included source files and makefiles which allows you to build the libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

#### Procedures to generate the libraries

- Open a Command Prompt window.
- Navigate to the route directory of the product of which you want to build the library:

<b>Generic Device Drivers</b>	<installation_folder>\sciopta\<version>\gdd\
<b>Utilities</b>	<installation_folder>\sciopta\<version>\util\

- Execute the makefile for GNU GCC for each product:  
For thumb mode: `make -f Makefile.tarmiar delivery_lib`
- The libraries will be installed in the directory: <installation\_folder>\sciopta\<version>\lib\arm\iar\

## 11 Libraries and Include Files

---

### 11.2.3 ARM RealView Kernel Libraries

The file name of the libraries have the following format:

<name>\_Xt.a

File location: <installation\_folder>\sciopta\<version>\lib\arm\arm\<version>\

#### 11.2.3.1 Optimization “X”

The libraries are delivered for three different compiler optimization. The letter **X** defines one of three compiler optimization levels.

“X” can have a value of 0,1 and 2 and defines the optimization.

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

#### 11.2.3.2 “t”

If the SCIOPTA Trap Interface is used, the libraries with the letter “t” after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

#### 11.2.3.3 Included SCIOPTA Kernel Libraries

Utilities	util_Xt.l
Generic device driver	gdd_Xt.l
Shell	sh_Xt.l

## 11.2.3.4 Building Kernel Libraries for ARM RealView

The example makefiles and project files are supposing to use libraries for the generic device driver (GDD) and utility (util) modules. As described above, there are some libraries delivered for specific compiler settings.

We have included source files and makefiles which allows you to build the libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

### Procedures to generate the libraries

- Open a Command Prompt window.
- Navigate to the route directory of the product of which you want to build the library:

<b>Generic Device Drivers</b>	<installation_folder>\sciopta\<version>\gdd\
<b>Utilities</b>	<installation_folder>\sciopta\<version>\util\

- Execute the makefile for ARM RealView for each product:  
For thumb mode: `gnu-make -f Makefile.tarmads delivery_lib`  
For ARM mode: `gnu-make -f Makefile.armads delivery_lib`
- The libraries will be installed in the directory: <installation\_folder>\sciopta\<version>\lib\arm\arm\<version>\

### 11.3 Luminary® Stellaris Family Driver Libraries

The Luminary Micro® Stellaris® Peripheral Driver Library is a set of drivers for accessing the peripherals found on the Stellaris family of ARM® Cortex™-M3 based microcontrollers.

Please consult also [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#) and [9.9 “Luminary® Stellaris Family Driver Library” on page 9-16](#).

#### 11.3.1 GNU GCC Stellaris Family Driver Libraries

The file name of the libraries have the following format:

libdriver.a

File location: <installation\_folder>\src\gcc\

#### 11.3.2 IAR Stellaris Family Driver Libraries

The file name of the libraries have the following format:

driverlib.a

File location: <installation\_folder>\src\ewarm\Exe\

#### 11.3.3 ARM RealView Stellaris Family Driver Libraries

The file name of the libraries have the following format:

driverlib.lib

File location: <installation\_folder>\src\rvmdk\

## 11.4 Include Files

### 11.4.1 Include Files Search Directories

Please make sure that the environment variable **SCIOPTA\_HOME** is defined as explained in chapter [2.4.6 “SCIOPTA\\_HOME Environment Variable” on page 2-4](#).

If you are using a Luminary Stellaris Cortex-M3 Family CPU, please make sure that the environment variable **LMI\_DRIVER** is defined as explained in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#).

If you are using a STMicroelectronics STM32 Family CPU, please make sure that the environment variable **STM32\_FWLIB** is defined as explained in chapter [2.4.11 “STMicroelectronics STM32 Firmware Library” on page 2-6](#).

Define the following entries the include files search directories field of your IDE:

```
.
%(SCIOPTA_HOME)\include
%(SCIOPTA_HOME)\include\sciopta\arm
```

For the Luminary® Stellaris Family Driver:

```
%(LMI_DRIVER)
```

Depending on the CPU and board you are using:

```
%(SCIOPTA_HOME)\bsp\arm\include
%(SCIOPTA_HOME)\bsp\arm\<CPU>\include
%(SCIOPTA_HOME)\bsp\arm\<CPU>\<BOARD>\include
```

### 11.4.2 Main Include File sciopta.h

This file contains some main definitions and the SCIOPTA Application Programming Interface.

Each module or file which is using SCIOPTA system calls and definitions must include the file **sciopta.h**.

sciopta.h	Main include file.
-----------	--------------------

File location: <installation\_folder>\sciopta\<version>\include\

The file sciopta.h includes all specific API header files.

File location: <installation\_folder>\sciopta\<version>\include\kernel

### 11.4.3 Configuration Definitions sconf.h

Files or modules which are SCIOPTA configuration dependent need to include first the file **sconf.h** and then the file **sciopta.h**.

The file **sconf.h** needs to be included if for instance you want to know the maximum number of modules allowed in a system. This information is stored in **SC\_MAX\_MODULES** in the file **sconf.h**. Please remember that **sconf.h** is automatically generated by the **sconf** configuration tool.

## 11 Libraries and Include Files

---

### 11.4.4 Main Data Types `types.h`

These types are introduced to allow portability between various SCIOPTA implementations.

The main data types are defined in the file `types.h` located in `ossys`. These types are not target processor dependent.

<code>types.h</code>	Processor independent data types.
----------------------	-----------------------------------

File location: `<installation_folder>\sciopta\<version>\include\ossys\`

### 11.4.5 ARM Data Types `types.h`

The ARM specific data types are defined in the file `types.h` located in `\arm\arch`.

<code>types.h</code>	ARM data types.
----------------------	-----------------

File location: `<installation_folder>\sciopta\<version>\include\sciopta\arm\arch\`

### 11.4.6 Global System Definitions `defines.h`

System wide definitions are defined in the file `defines.h`. Among other global definitions, the base addresses of the IDs of the SCIOPTA system messages are defined in this file. Please consult this file for managing and organizing the message IDs of your application (see also chapter [7.7.2.6 “Message Number \(ID\) organization” on page 7-16](#)).

<code>defines.h</code>	System wide constant definitions.
------------------------	-----------------------------------

File location: `<installation_folder>\sciopta\<version>\include\ossys\`



## 12 Linker Scripts and Memory Map

### 12.1 Introduction

A linker script is controlling the link in the build process. The linker script is written in a specific linker command language. The linker script and linker command language are compiler specific.

The linker script describes how the defined memory sections in the link input files are mapped into the output file which will be loaded in the target system. Therefore the linker script controls the memory layout in the output file.

SCIOPTA uses the linker scripts to define and map SCIOPTA modules into the global memory map.

### 12.2 GCC Linker Script

You can find examples of linker scripts in the SCIOPTA examples and getting started projects. In these examples there is usually a main linker script which includes a second linker script. The main linker scripts are board dependent.

<board>.ld	Linker script for GNU GCC
------------	---------------------------

File location: <installation\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

This linker script file includes another linker script which is usually located at:

module.ld	Linker script for GNU GCC
-----------	---------------------------

File location: <installation\_folder>\sciopta\<version>\bsp\arm\include\

Study these linker script files to get full information how to locate a SCIOPTA system in the embedded memory space.

#### 12.2.1 Memory Regions

The main linker script contains the allocation of all available memory and definition of the memory regions including the regions for all modules. Sections are assigned to SCIOPTA specific memory regions. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name.

The following regions are typically defined by the user:

rom	destination for read-only data
sram	internal SRAM
system_mod	Memory region for the system module
<module_name>_mod	Memory region for other modules (with name: <module_name>).

## 12 Linker Scripts and Memory Map

### 12.2.2 Module Sizes

The sizes used by SCIOPTA of each module must be defined by the user in the linker script. This size determines the memory which will be used by SCIOPTA for message pools, PCBs and other system data structures.

The name of the size is usually defined as follows: `<module_name>_size`

The module name for the system module is **system**.

Typical definitions (for modules **system**, **dev**, **ips** and **user**) might look as follows:

```
system_size = 0x4000;
dev_size    = 0x4000;
ips_size    = 0x4000;
user_size   = 0x4000;
```

size calculation:

$size\_mod = p * 128 + stack + pools + mcb + textsize$

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

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

$size\_mcb = 96 + friends + hooks * 4 + 8$

where:

friend	if friends are not used: 0 if friends are used 16 bytes
hooks	number of hooks configured

Please consult the configuration chapter (**SCONF** utility) of the SCIOPTA Kernel, User's Guide for information about friend and hook settings.

### 12.2.3 Specific Module Values

For each module four values are calculated in the linker script:

<module_name>_start	Start address of module RAM
<module_name>_initsize	Size of initialized RAM
<module_name>_size	Complete size of the module
<module_name>_mod	A structure which contains the above three addresses.

The SCIOPTA configuration utility **SCONF** is using these definitions to pass the module addresses to the kernel.

Example in the linker script:

```
.module_init :
{
    system_mod = .;
    LONG(system_start);
    LONG(system_size);
    LONG(system_initsize);
    dev_mod = .;
    LONG(dev_start);
    LONG(dev_size);
    LONG(dev_initsize);
    ips_mod = .;
    LONG(ips_start);
    LONG(ips_size);
    LONG(ips_initsize);
    user_mod = .;
    LONG(user_start);
    LONG(user_size);
    LONG(user_initsize);
} > rom
```

## 12 Linker Scripts and Memory Map

### 12.3 IAR Embedded Workbench Linker Script

You can find examples of linker scripts in the SCIOPTA examples and getting started projects. The linker scripts are board dependent and can be found at:

<board>.xcl	Linker script for IAR.
-------------	------------------------

File location: <installation\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

Study these linker script files to get full information how to locate a SCIOPTA system in the embedded memory space.

For IAR you need to define the free RAM of the modules in a separate file. In this area there are no initialized data. Module Control Block (ModuleCB), Process Control Blocks (PCBs), Stacks and Message Pools are placed in this free RAM.

#### Source File

map.c	Module mapping definitions for IAR.
-------	-------------------------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\<example>\<board>\

### 12.4 ARM RealView Linker Script

You can find examples of linker scripts in the SCIOPTA examples and getting started projects. The linker scripts are board dependent and can be found at:

<board>.sct	Linker script for ARM RealView
-------------	--------------------------------

File location: <installation\_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

Study these linker script files to get full information how to locate a SCIOPTA system in the embedded memory space.

12.5 Data Memory Map

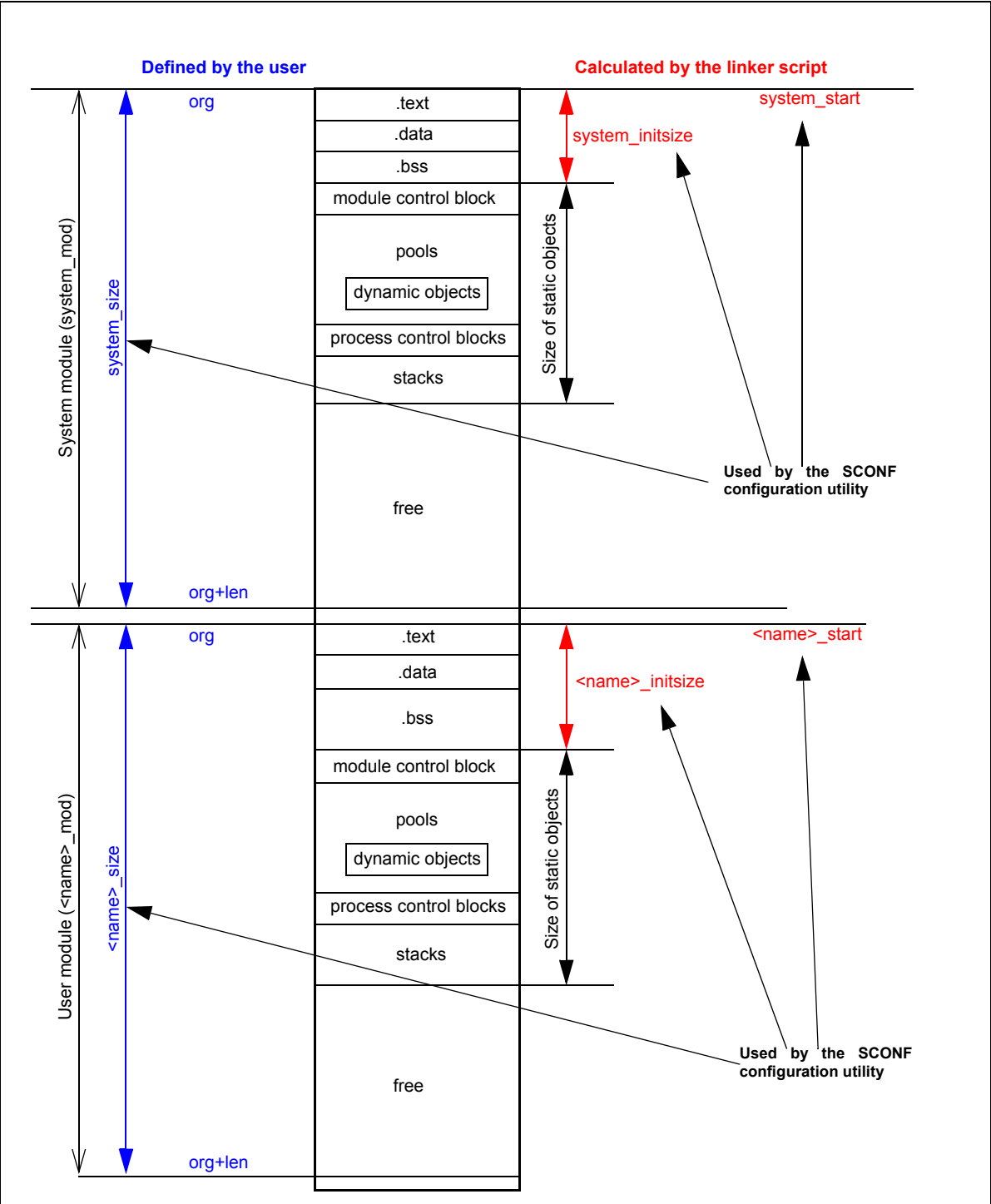


Figure 12-1: SCIOPTA Memory Map

## 13 System Building

### 13.1 Introduction

A SCIOPTA System can be built by using many different compilers and integrated development environments.

You can only use compilers which are officially supported by SCIOPTA.

In a typical SCIOPTA delivery you will find project files for different integrated development environments. An integrated development environment (IDE) is computer software to help computer programmers develop software.

They normally consist of a source code editor, a compiler and/or interpreter, build-automation tools, and (usually) a debugger. Sometimes a version control system and various tools to simplify the construction of a GUI are integrated as well. Many modern IDEs also integrate a class browser, an object inspector and a class hierarchy diagram, for use with object oriented software development. Although some multiple-language IDEs are in use, such as the Eclipse IDE or Microsoft Visual Studio, typically an IDE is devoted to a specific programming language.

### 13.2 Makefile and GNU GCC

#### 13.2.1 Tools

You will need the Sourcery G++ Lite Edition GNU Tool Chain Package for ARM.

This product can be found on the SCIOPTA Cortex CD.

To run the makefile we are using the GNU Make utility which we have included in the \sciopta\bin\win32 directory. The GNU make consists of the following files:

- gnu-make.exe
- rm.exe
- sed.exe
- libiconv2.dll
- libintl3.dll

#### 13.2.2 Environment Variables

The following environment variables need to be defined:

- **SCIOPTA\_HOME** needs to point to the SCIOPTA delivery. Please consult chapter [2.4.6 “SCIOPTA\\_HOME Environment Variable” on page 2-4](#) for more information.
- If you are using a Luminary Stellaris Cortex-M3 Family CPU, please make sure that the environment variable **LMI\_DRIVER** is defined as explained in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#).
- If you are using a STMicroelectronics STM32 Family CPU, please make sure that the environment variable **STM32\_FWLIB** is defined as explained in chapter [2.4.11 “STMicroelectronics STM32 Firmware Library” on page 2-6](#).
- Include the CodeSourcery GNU C & C++ compiler bin directory in the PATH environment variable as described in chapter [2.4.9 “GNU Tool Chain Installation” on page 2-5](#).
- Include the SCIOPTA bin directory in the PATH environment variable as described in chapter [2.4.7 “Setting SCIOPTA Path Environment Variable” on page 2-4](#).

## 13 System Building

---

### 13.2.3 Makefile Location

You will find typical SCIOPTA makefiles for GNU GCC in the example deliveries.

Makefile	Example makefile.
----------	-------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\

Usually these example makefiles include a board specific makefile called board.mk located here:

board.mk	Board dependent makefiles.
----------	----------------------------

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\hello\<board>\

### 13.2.4 Project Settings

Please consult the delivered example makefiles for detailed information about compiler, assembler and linker calls, options and settings.

## 13 System Building

### 13.3 Eclipse IDE and GNU GCC

#### 13.3.1 Introduction

Please consult chapter [2.4.12 “Eclipse C/C++ Development Tooling - CDT” on page 2-7](#).

#### 13.3.2 Tools

You will need

- Sourcery G++ Lite Edition GNU Tool Chain Package for ARM
- Eclipse Platform Version 3.4.1 (Ganymede) with Eclipse C/C++ Development Tools (CDT) Version 5.0.1.
- Eclipse CDT Plug-Ins for SCIOPTA Cortex:
  - com.sciopta.ui\_x.x.x.jar
  - com.sciopta.gnu.cortex\_x.x.x.jar

The “x.x.x” represent the actual version.

These products can be found on the SCIOPTA Cortex CD.

In order to run the Eclipse Platform you also need the Sun Java 2 SDK, Standard Edition for Microsoft Windows.

To run the Eclipse build and make we are using the GNU Make utility which we have included in the \sciopta\bin\win32 directory. The GNU make consists of the following files:

- gnu-make.exe
- rm.exe
- sed.exe
- libiconv2.dll
- libintl3.dll

#### 13.3.3 Environment Variables

The following environment variables need to be defined:

- **SCIOPTA\_HOME** needs to point to the SCIOPTA delivery. Please consult chapter [2.4.6 “SCIOPTA\\_HOME Environment Variable” on page 2-4](#) for more information.
- If you are using a Luminary Stellaris Cortex-M3 Family CPU, please make sure that the environment variable **LMI\_DRIVER** is defined as explained in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#).
- If you are using a STMicroelectronics STM32 Family CPU, please make sure that the environment variable **STM32\_FWLIB** is defined as explained in chapter [2.4.11 “STMicroelectronics STM32 Firmware Library” on page 2-6](#).
- Include the CodeSourcery GNU C & C++ compiler bin directory in the PATH environment variable as described in chapter [2.4.9 “GNU Tool Chain Installation” on page 2-5](#).
- Include the SCIOPTA bin directory in the PATH environment variable as described in chapter [2.4.7 “Setting SCIOPTA Path Environment Variable” on page 2-4](#).



### 13.3.4 Project Settings and Building

Please consult the getting started example [3.2.2 “Eclipse IDE and GNU GCC” on page 3-2](#) for detailed information about compiler, assembler and linker calls, options and settings.

### 13.4 iSYSTEM© winIDEA

#### 13.4.1 Introduction

winIDEA is the IDE for all iSYSTEMS emulators. It is the a Integrated Development Environment, which contains all the necessary tools in one shell. winIDEA consists of a project manager, a 3rd party tools integrator, a multi-file C source editor and a high-level source debugger.

Please consult <http://www.isystem.com/> for more information about the iSYSTEM emulator/debugger.

#### 13.4.2 Tools

You will need:

- The Sourcery G++ Lite Edition GNU Tool Chain Package for ARM.  
This product can be found on the SCIOPTA ARM CD.
- iSYSTEM winIDEA

#### 13.4.3 Environment Variables

The following environment variables need to be defined:

- **SCIOPTA\_HOME** needs to point to the SCIOPTA delivery. Please consult chapter [2.4.6 “SCIOPTA\\_HOME Environment Variable” on page 2-4](#) for more information.
- If you are using a Luminary Stellaris Cortex-M3 Family CPU, please make sure that the environment variable **LMI\_DRIVER** is defined as explained in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#).
- If you are using a STMicroelectronics STM32 Family CPU, please make sure that the environment variable **STM32\_FWLIB** is defined as explained in chapter [2.4.11 “STMicroelectronics STM32 Firmware Library” on page 2-6](#).
- Include the CodeSourcery GNU C & C++ compiler bin directory in the PATH environment variable as described in chapter [2.4.9 “GNU Tool Chain Installation” on page 2-5](#).

## 13 System Building

---

### 13.4.4 winIDEA Project Files Location

You will find typical winIDEA project files for SCIOPTA in the example deliveries.

iC3000.xjrf	iSYSTEM winIDEA project file
iC3000.xqrf	iSYSTEM winIDEA project file

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\<example>\<board>

### 13.4.5 Project Settings

Selecting **Projects > Settings...** from the menu (or press **Alt+F7**) opens the **Project Settings** window.

After expanding the **C/C++ Build** entry you can select **Settings** to open the project specific settings window.

Please consult the delivered example winIDEA project for detailed information about compiler, assembler and linker calls, options and settings.

### 13.5 IAR Embedded Workbench for ARM

#### 13.5.1 Introduction

IAR Embedded Workbench for ARM is a set of development tools for building and debugging embedded system applications using assembler, C and C++. It provides a completely integrated development environment that includes a project manager, editor, build tools and the C-SPY debugger.

Please consult <http://www.iar.com/> for more information about the IAR Embedded Workbench.

#### 13.5.2 Tools

You will need

- IAR Embedded Workbench for ARM including the following main components:
  - IAR Assembler for ARM
  - IAR C/C++ Compiler for ARM
  - IAR Embedded Workbench IDE
  - IAR XLINK

#### 13.5.3 Environment Variables

The following environment variables need to be defined:

- **SCIOPTA\_HOME** needs to point to the SCIOPTA delivery. Please consult chapter [2.4.6 “SCIOPTA\\_HOME Environment Variable” on page 2-4](#) for more information.
- If you are using a Luminary Stellaris Cortex-M3 Family CPU, please make sure that the environment variable **LMI\_DRIVER** is defined as explained in chapter [2.4.10 “Luminary Stellaris Cortex-M3 Family Driver Library” on page 2-6](#).
- If you are using a STMicroelectronics STM32 Family CPU, please make sure that the environment variable **STM32\_FWLIB** is defined as explained in chapter [2.4.11 “STMicroelectronics STM32 Firmware Library” on page 2-6](#).
- Include the SCIOPTA bin directory in the PATH environment variable as described in chapter [2.4.7 “Setting SCIOPTA Path Environment Variable” on page 2-4](#). This will give access to the sconf.exe utility. Some IAREW examples might call sconf.exe directly from the workbench to do the SCIOPTA configuration.

13.5.4 IAR EW Project Files Location

You will find typical IAR EW project files for SCIOPTA in the example deliveries.

<board>.ewd	IAR EW project file
<board>.eww	IAR EW project file

File location: <installation\_folder>\sciopta\<version>\exp\krm\arm\<example>\<board>

13.5.5 Project Settings

Selecting **Projects > Options...** from the menu (or press **Alt+F7**) opens the **Options** window.

Please consult the delivered example IAR EW project for detailed information about compiler, assembler and linker calls, options and settings.

## 14 SCIOPTA Debugging

### 14.1 Introduction

For debugging embedded systems with the SCIOPTA real-time operating system you have the choice of several different debugging levels:

1. **pure source-level debugging** by using a third-party debugger.  
This is the classic style. The user is debugging on assembler, C or C++ level. The debugger has no specific knowledge about SCIOPTA kernel objects. But as the structures and addresses of the SCIOPTA kernel objects such as process control blocks and message pools are accessible by the user this is still a good and efficient method for debugging SCIOPTA systems.
2. Third-party source-level debugging with **SCIOPTA kernel awareness**.  
The debugger has specific knowledge about SCIOPTA kernel objects. This is provided by DLLs, plug-ins or macros delivered by SCIOPTA or by the debugger manufacturer. The user has direct access to module, process, pool and message information. Process aware breakpoints can be handled.  
The level of built-in SCIOPTA functionality depends on the possibilities of the RTOS kernel awareness SDK provided by the debugger manufacturer.
3. **DRUID system level debugger**.  
DRUID is a system level debugger supplied by SCIOPTA and is used in addition to a source-level debugger. It is not replacing the source-level debugger as DRUID gives you another view into a SCIOPTA system on a higher level. DRUID connects to the target system over a separate channel which can be a simple serial line, a TCP/IP ethernet link, a USB interface or a transparent channel of the source-level debugger.

System level debugging with DRUID allows the user to

- Get process states
- Analyse message pools
- Trace SCIOPTA messages and other kernel events
- Set watch- and breakpoints on system events
- Read message sequence charts from the target system

Please consult the DRUID System Level Debugger User's Guide and Reference Manual for more information.

14.2 Kernel Awareness for Third-Party Debuggers

14.2.1 Kernel Awareness for Lauterbach Trace32

There are three files needed for the SCIOPTA kernel awareness in Lauterbach Trace32 debuggers:

sciopta.cmm	Kernel awareness setup for Lauterbach Trace32
sciopta.men	RTOS specific menu
sciopta.t32	TRACE32 configuration file

The files must reside in the debugger/project working directory.

File location: <install\_folder>\sciopta\<version>\3rdParty\Lauterbach\

Some board dependent files \*.cmm can also be found in the include directories of the board support packages for specific boards.

## 15 Kernel Error Codes

### 15.1 Introduction

SCIOPTA uses a centralized mechanism for error reporting called Error Hooks.

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.

When the error hook is called from the kernel, all information about the error are transferred in 32-bit error word parameter.

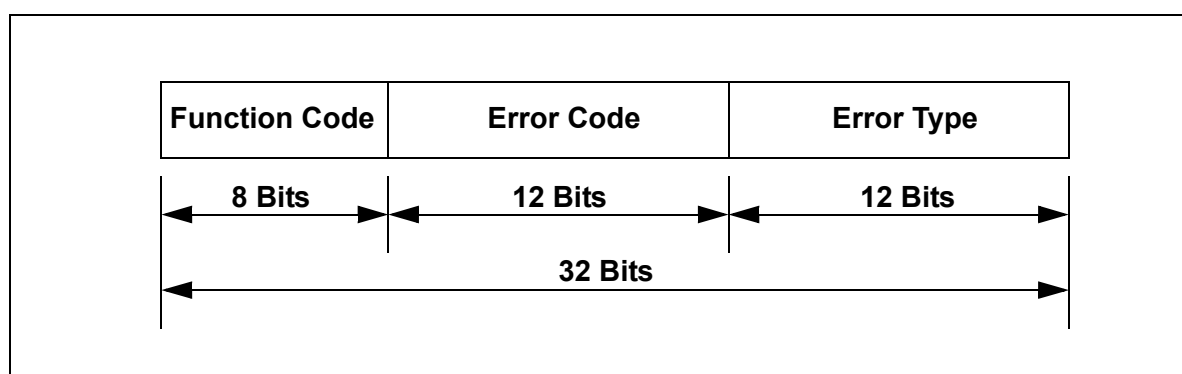


Figure 15-1: 32-bit Error Word

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

Please consult the SCIOPTA - Kernel, User's Guide for more information about error hooks.

### 15.2 Include Files

The error codes are defined in the **err.h** include file.

File location: <install\_folder>\sciopta\<version>\include\kernel\

The error descriptions are defined in the **errtxt.h** include file.

File location: <install\_folder>\sciopta\<version>\include\ossys\



## 15.3 Function Codes

Name	Number	Error Source
SC_MSGALLOC	0x01	<a href="#">sc_msgAlloc</a>
SC_MSGFREE	0x02	<a href="#">sc_msgFree</a>
SC_MSGADDRGET	0x03	<a href="#">sc_msgAddrGet</a>
SC_MSGSNDGET	0x04	<a href="#">sc_msgSndGet</a>
SC_MSGSIZEGET	0x05	<a href="#">sc_msgOwnerGet</a>
SC_MSGSIZESET	0x06	<a href="#">sc_msgSizeGet</a>
SC_MSGOWNERGET	0x07	<a href="#">sc_msgSizeSet</a>
SC_MSGTX	0x08	<a href="#">sc_msgTx</a>
SC_MSGTXALIAS	0x09	<a href="#">sc_msgTxAlias</a>
SC_MSGRX	0x0A	<a href="#">sc_msgRx</a>
SC_MSGPOOLIDGET	0x0B	<a href="#">sc_poolIdGet</a>
SC_MSGACQUIRE	0x0C	<a href="#">sc_msgAcquire</a>
SC_MSGALLOCCLR	0x0D	<a href="#">sc_msgAllocClr</a>
SC_MSGHOOKREGISTER	0x0E	<a href="#">sc_msgHookRegister</a>
SC_POOLCREATE	0x10	<a href="#">sc_poolCreate</a>
SC_POOLRESET	0x11	<a href="#">sc_poolReset</a>
SC_POOLKILL	0x12	<a href="#">sc_poolKill</a>
SC_POOLINFO	0x13	<a href="#">sc_poolInfo</a>
SC_POOLDEFAULT	0x14	<a href="#">sc_poolInfo</a>
SC_POOLIDGET	0x15	<a href="#">sc_procIdGet</a>
SC_SYSPOLKILL	0x16	<a href="#">sc_sysPoolKill</a>
SC_PROCPRIOGET	0x20	<a href="#">sc_procPrioGet</a>
SC_PROCPRIOSET	0x21	<a href="#">sc_procPrioSet</a>
SC_PROCSLICEGET	0x22	<a href="#">sc_procSliceGet</a>
SC_PROCSLICESET	0x23	<a href="#">sc_procSliceSet</a>
SC_PROCIDGET	0x24	<a href="#">sc_procIdGet</a>
SC_PROCPPIDGET	0x25	<a href="#">sc_procPpidGet</a>
SC_PROCNAMEGET	0x26	<a href="#">sc_procNameGet</a>
SC_PROCSTART	0x27	<a href="#">sc_procStart</a>
SC_PROCTOP	0x28	<a href="#">sc_procStop</a>
SC_PROCVARINIT	0x29	<a href="#">sc_procVarInit</a>
SC_PROCSCHEDUNLOCK	0x2A	<a href="#">sc_procSchedUnlock</a>
SC_PROCPRIOCREATESTATIC	0x2B	<a href="#">sc_procPrioCreate</a>
SC_PROCINTCREATESTATIC	0x2C	<a href="#">sc_procIntCreate</a>
SC_PROCTIMCREATESTATIC	0x2D	<a href="#">sc_procTimCreate</a>
SC_PROCURINTCREATESTATIC	0x2E	<a href="#">sc_procUsrIntCreate</a> (former call)
SC_PROCPRIOCREATE	0x2F	<a href="#">sc_procPrioCreate</a>
SC_PROCINTCREATE	0x30	<a href="#">sc_procIntCreate</a>

Name	Number	Error Source
SC_PROCTIMCREATE	0x31	<a href="#">sc_procTimCreate</a>
SC_PROCUSRINTCREATE	0x32	sc_procUsrIntCreate (former call)
SC_PROCKILL	0x33	<a href="#">sc_procKill</a>
SC_PROCYIELD	0x34	<a href="#">sc_procYield</a>
SC_PROCOBSERVE	0x35	<a href="#">sc_procObserve</a>
SC_SYSPROCCREATE	0x36	sc_sysProcCreate
SC_PROCSCHEDLOCK	0x37	<a href="#">sc_procSchedLock</a>
SC_PROCVARGET	0x38	<a href="#">sc_procVarGet</a>
SC_PROCVARSET	0x39	<a href="#">sc_procVarSet</a>
SC_PROCVARDEL	0x3A	<a href="#">sc_procVarDel</a>
SC_PROCVARRM	0x3B	<a href="#">sc_procVarRm</a>
SC_PROCUOBSERVE	0x3C	<a href="#">sc_procUnobserve</a>
SC_PROCPATHGET	0x3D	<a href="#">sc_procPathGet</a>
SC_PROCPATHCHECK	0x3E	sc_procPathCheck
SC_PROCHOOKREGISTER	0x3F	<a href="#">sc_procHookRegister</a>
SC_MODULECREATE	0x40	<a href="#">sc_moduleCreate</a>
SC_MODULEKILL	0x41	<a href="#">sc_moduleKill</a>
SC_MODULENAMEGET	0x42	<a href="#">sc_moduleNameGet</a>
SC_MODULEIDGET	0x43	<a href="#">sc_moduleIdGet</a>
SC_MODULEINFO	0x44	<a href="#">sc_moduleInfo</a>
SC_MODULEPRIOSSET	0x45	sc_sysModulePrioSet
SC_MODULEPRIOGET	0x46	sc_sysModulePrioGet
SC_MODULEFRIENDADD	0x47	<a href="#">sc_moduleFriendAdd</a>
SC_MODULEFRIENDRM	0x48	<a href="#">sc_moduleFriendRm</a>
SC_MODULEFRIENDGET	0x49	<a href="#">sc_moduleFriendGet</a>
SC_MODULEFRIENDNON	0x4A	<a href="#">sc_moduleFriendNone</a>
SC_MODULEFRIENDALL	0x4B	<a href="#">sc_moduleFriendAll</a>
SC_TRIGGERVALUESET	0x50	<a href="#">sc_triggerValueSet</a>
SC_TRIGGERVALUEGET	0x51	<a href="#">sc_triggerValueGet</a>
SC_TRIGGER	0x52	<a href="#">sc_trigger</a>
SC_TRIGGERWAIT	0x53	<a href="#">sc_triggerWait</a>
SC_TMOADD	0x58	<a href="#">sc_tmoAdd</a>
SC_TMO	0x59	sc_tmo
SC_SLEEP	0x5A	<a href="#">sc_sleep</a>
SC_TMORM	0x5B	<a href="#">sc_tmoRm</a>
SC_CONNECTORREGISTER	0x60	<a href="#">sc_connectorRegister</a>
SC_CONNECTORUNREGISTER	0x61	<a href="#">sc_connectorUnregister</a>
SC_DISPATCHER	0x62	dispatcher

## 15.4 Error Codes

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

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

## 15.5 Error Types

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

## 16 SCONF Kernel Configuration Utility

### 16.1 Introduction

The kernel of a SCIOPTA system needs to be configured before you can generate 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 settings 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.

### 16.2 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.

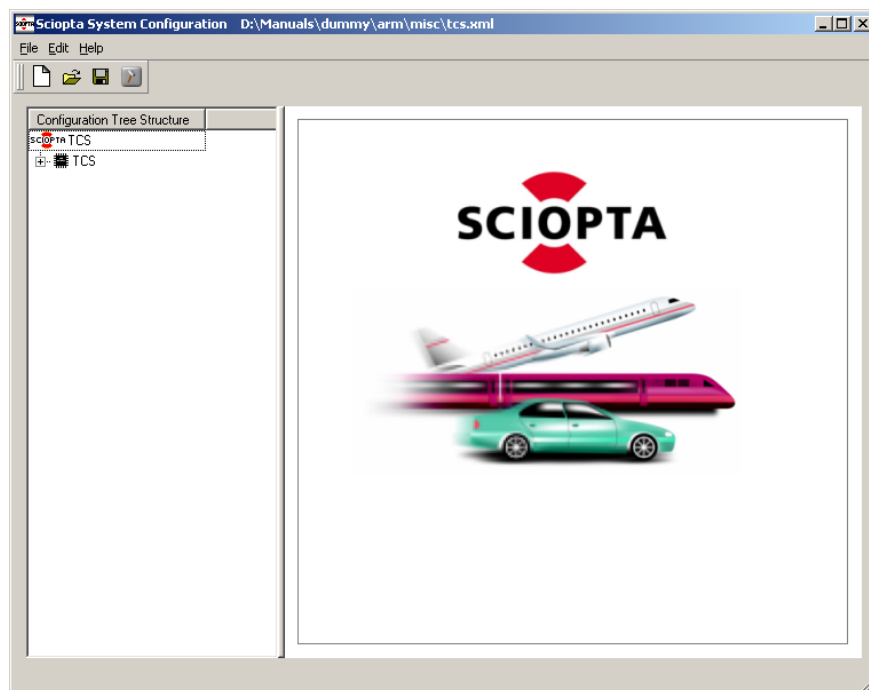


Figure 16-1: SCIOPTA Configuration Utility Start Screen

### 16.3 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.

### 16.4 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.

## 16.5 SCONF Windows

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

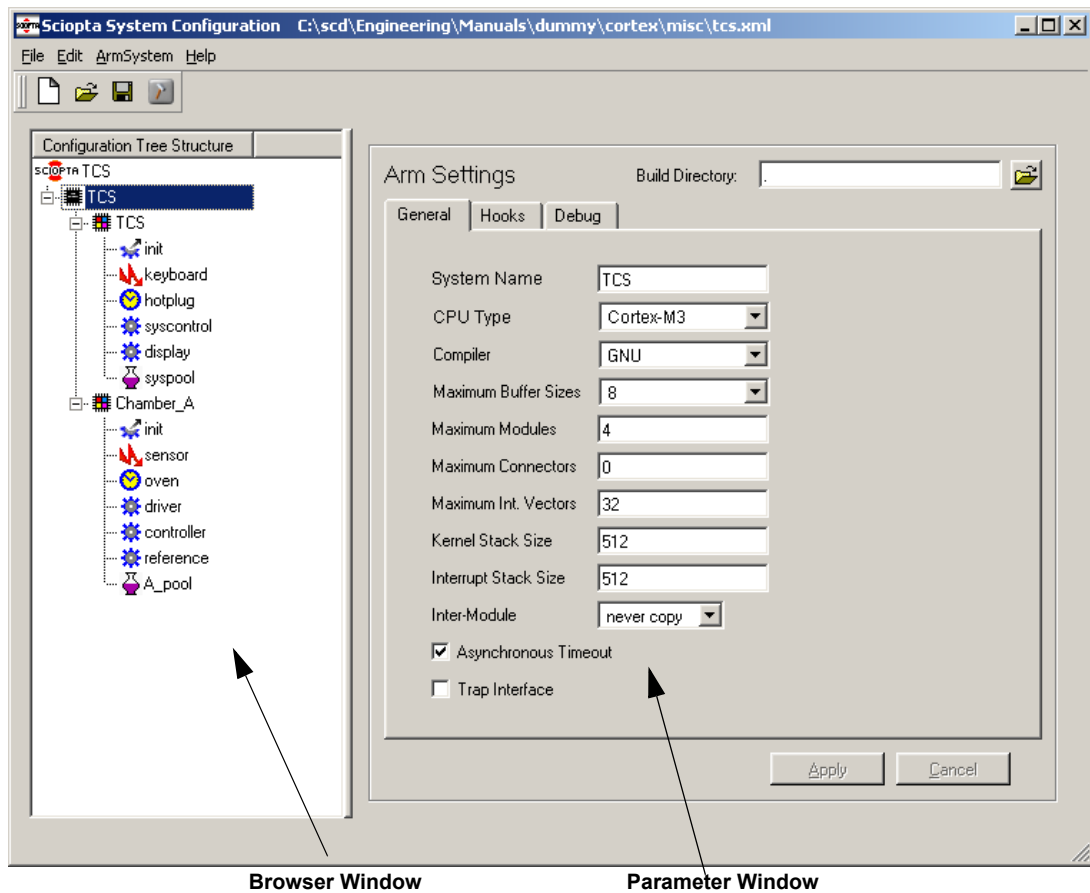


Figure 16-2: SCONF Windows

### 16.5.1 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.

### 16.5.2 Browser Window

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

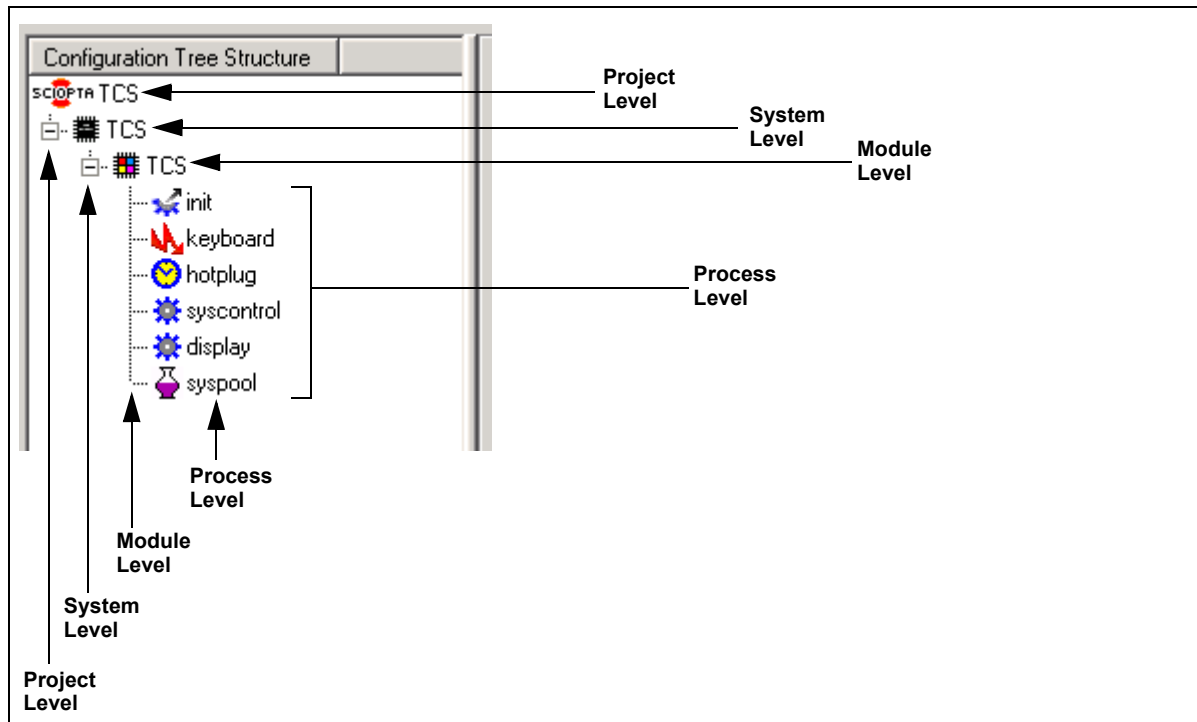


Figure 16-3: Browser Windows

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.

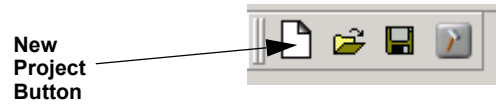
- The uppermost level is the **Project Level** where all project configurations will be done. The project name can be defined and you can create new systems for the project.
- In the **System Level** you are configuring the system for one CPU. You can create the static modules for the system and configure 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**.



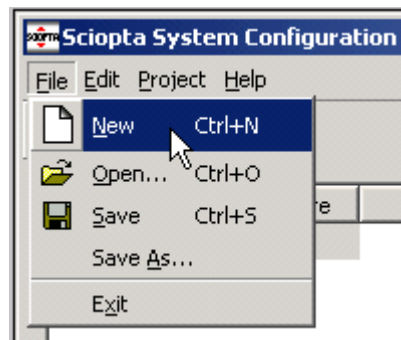
## 16 SCONF Kernel Configuration Utility

### 16.6 Creating a New Project

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

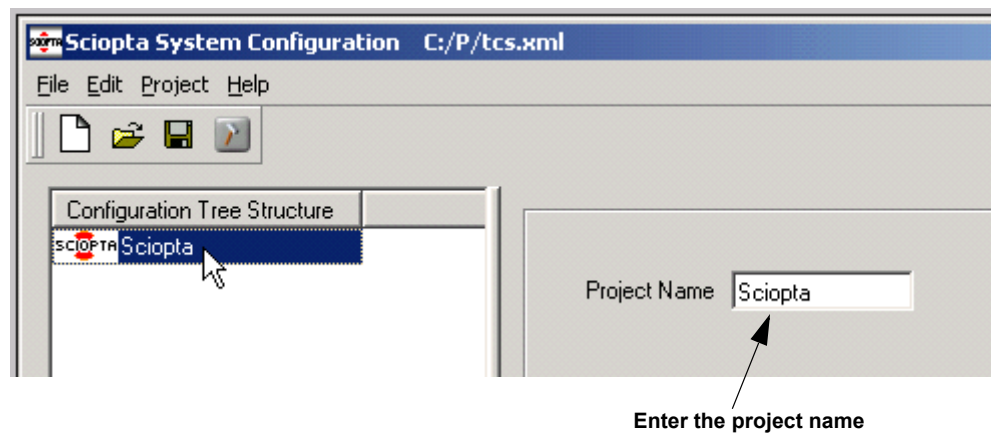


You also can create a new project from the file menu or by the Ctrl+N keystroke:



### 16.7 Configure the Project

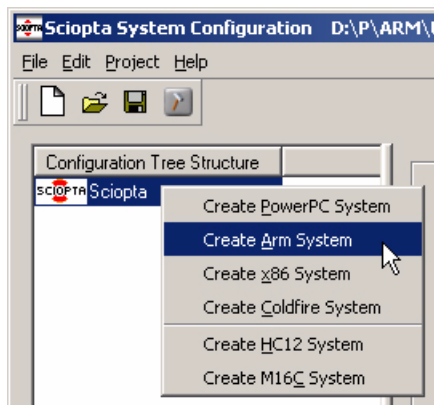
You can define and modify the project name. Click on the project name on the right side of the SCIOPTA logo and enter the project name in the parameter window.



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

### 16.8 Creating Systems

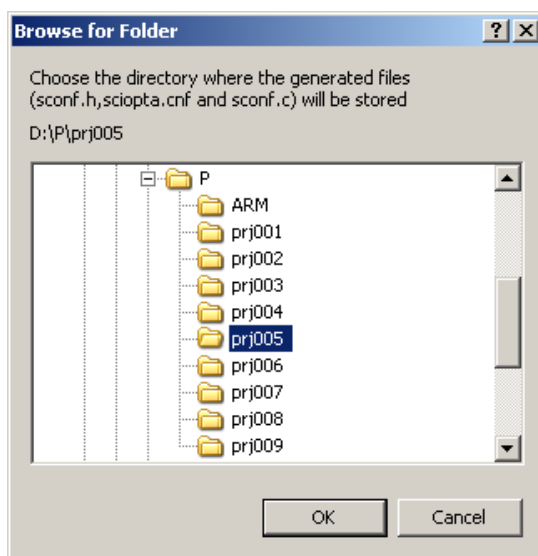
From the project level you can create new systems. Move the mouse pointer over the project and right-click the mouse.



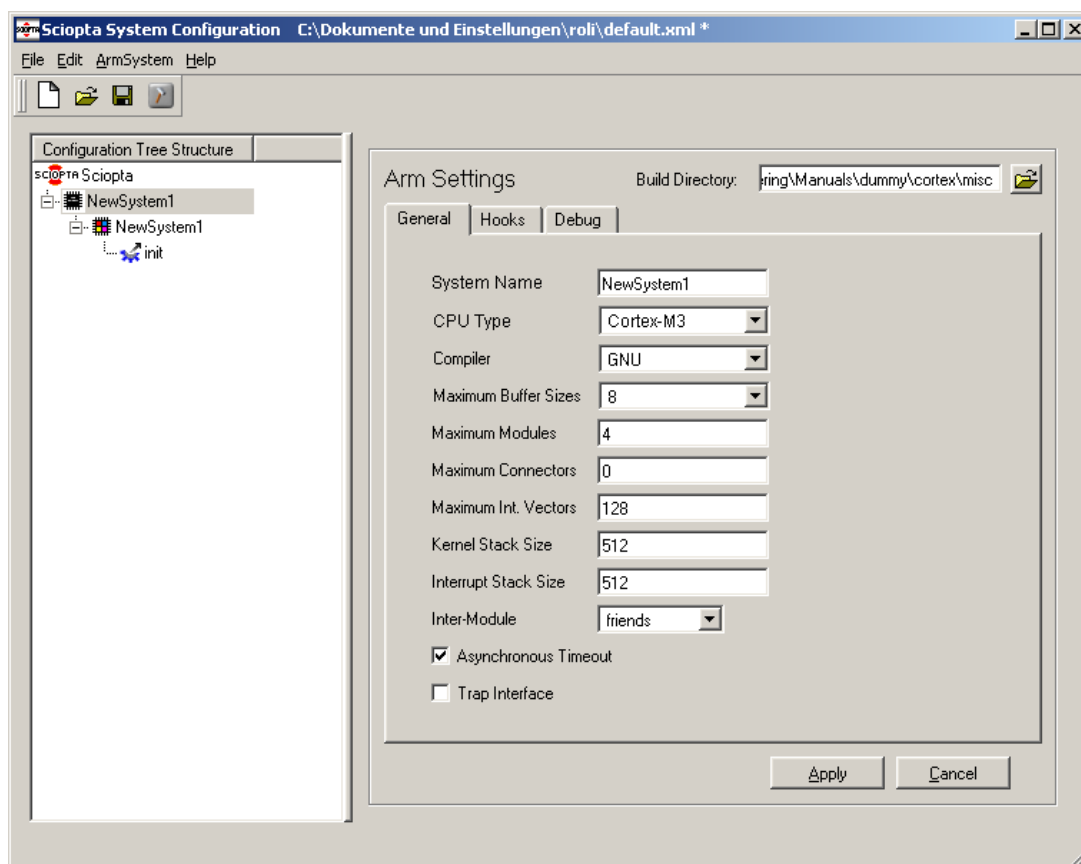
A pop-up menu appears and allows you to select a system out of all SCIOPTA supported target CPUs.

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.

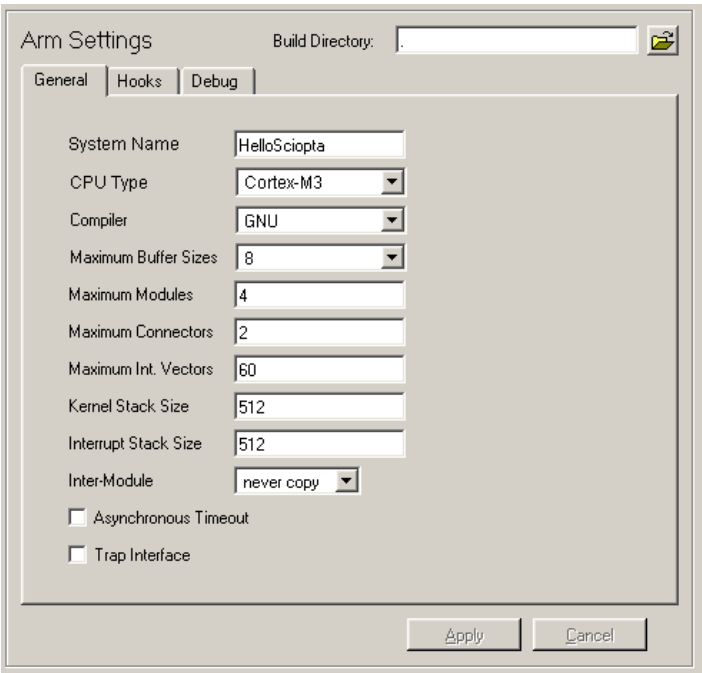
You are now ready to configure the individual targets.

### 16.9 Configuring ARM Cortex 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.

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

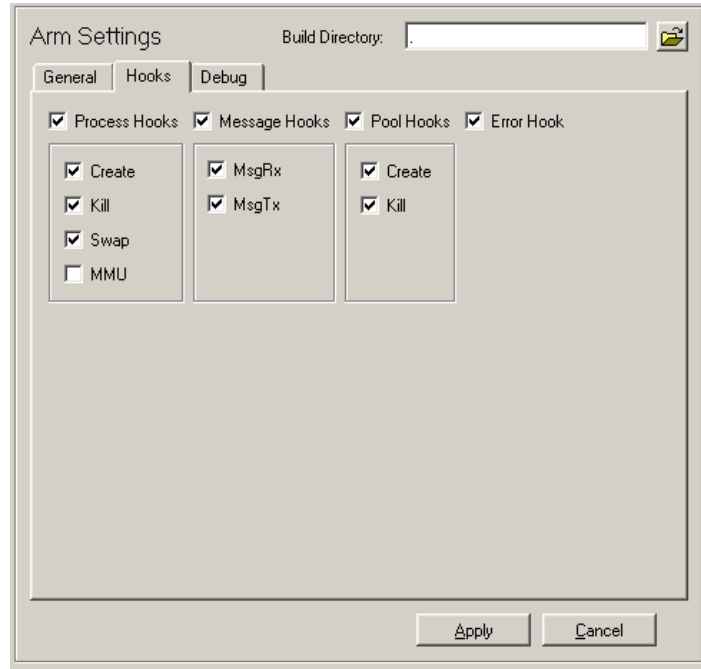
#### 16.9.1 General Configuration



Parameter	Description
<b>System Name</b>	Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.
<b>CPU Type</b>	Give the name of you specific target processor derivative.
<b>Compiler</b>	Select the C/C++ Cross compiler which you are using for this SCIOPTA system.
<b>Maximum Buffer Sizes</b>	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.
<b>Maximum Modules</b>	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.

Parameter	Description	
<b>Maximum Connectors</b>	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.	
<b>Kernel Stack Size</b>	Currently not used. Entered values are not considered.	
<b>Interrupt Stack Size</b>	Currently not used. Entered values are not considered.	
<b>Inter-Module</b>	Selects the module relations and defines the message copy behaviour.	
	<b>Value</b>	<b>Description</b>
	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 <a href="#">5.4.1 “SCIOPTA Module Friend Concept”</a> on page 5-10 for more information.
<b>Asynchronous Timeout</b>	When checked the SCIOPTA ARM Cortex kernel includes timeout-server functionality.	
<b>Trap Interface</b>	The SCIOPTA Trap Interface (system calls by software traps) will be used.	

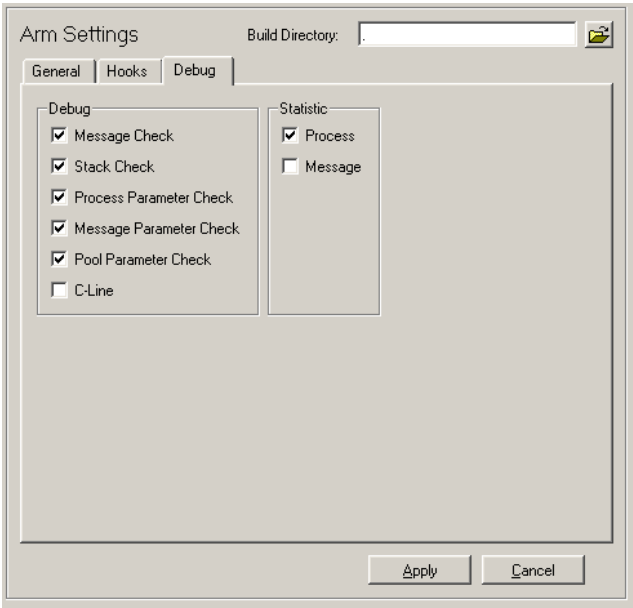
### 16.9.2 Configuring Hooks



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.

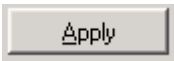
16.9.3 Debug Configuration



Parameter	Description
<b>Message Check</b>	If you are selecting this check box some test functions on messages will be included in the kernel.
<b>Stack Check</b>	If you are selecting this check box a stack check will be included in the kernel.
<b>Process Parameter Check</b>	If you are selecting this check box the kernel will do some testing on the parameters of the process system calls.
<b>Message Parameter Check</b>	If you are selecting this check box the kernel will do some testing on the parameters of the message system calls.
<b>Pool Parameter Check</b>	If you are selecting this check box the kernel will do some testing on the parameters of the pool system calls.
<b>C-Line</b>	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.
<b>Process Statistics</b>	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.
<b>Message Statistics</b>	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

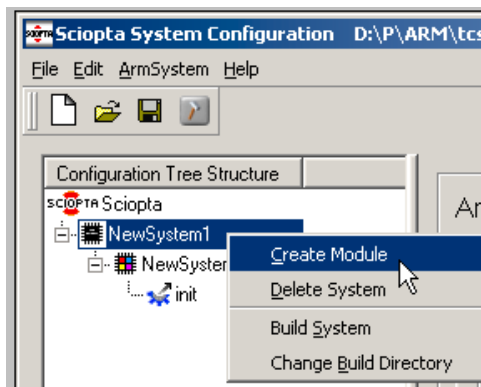
Click on the Apply button to accept the target configuration settings.



## 16 SCONF Kernel Configuration Utility

### 16.10 Creating Modules

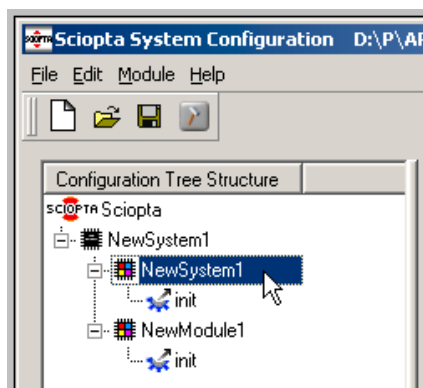
From the system level you can create new modules. Move the mouse pointer over the system and right-click the mouse.



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.



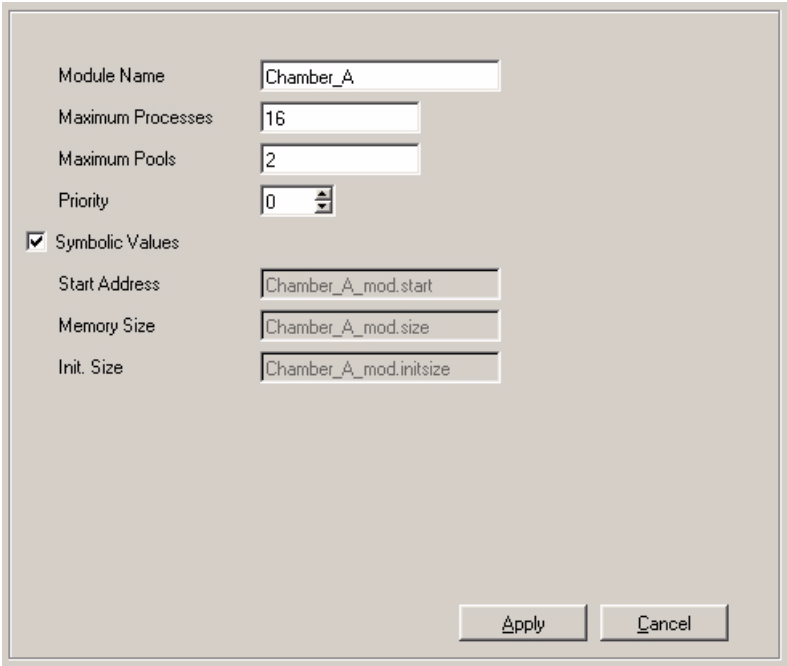
You can create up to 127 modules.

You are now ready to configure the individual modules.



### 16.11 Configuring Modules

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



The screenshot shows a configuration window with the following fields and values:

- Module Name:
- Maximum Processes:
- Maximum Pools:
- Priority:
- ☒ Symbolic Values
- Start Address:
- Memory Size:
- Init. Size:

Buttons at the bottom right:

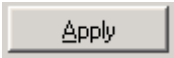
Parameter	Description
<b>Module Name</b>	Enter the name of the module. If you have selected the system module (the first module or the module with the id 0) you cannot give or modify the name as it will have the same name as the target system.
<b>Maximum Processes</b>	Maximum number of processes in the module. The kernel will not allow to create more processes inside the module than stated here. The maximum value is 16383.
<b>Maximum Pools</b>	Maximum number of pools in the module. The kernel will not allow to create more pools inside the module than stated here. The maximum value is 128.
<b>Priority</b>	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.

Parameter	Description																				
<b>Start Address</b>	<p>This is the start address of the module in RAM.</p> <p>Best is to specify a label which will be resolved at link time (e.g. <code>&lt;module_name&gt;_mod</code>). The specified label will be used in the linker script. Therefore all memory allocation for all modules is controlled by the linker script.</p> <p>You may specify an absolute address, but you need to be very carefully check with the linker script to avoid overlapping.</p>																				
<b>Memory Size</b>	<p>Size of the module in bytes (RAM).</p> <p>Best is to specify a label which will be resolved at link time (e.g. <code>&lt;module_name&gt;_size</code>). The specified label will be used in the linker script. Therefore all memory allocation for all modules is controlled by the linker script.</p> <p>You may specify an absolute address, but you need to be very carefully check with the linker script to avoid overlapping.</p> <p>The minimum module size can be calculated according to the following formula (bytes): <math>\text{size\_mod} = p * 128 + \text{stack} + \text{pools} + \text{mcb} + \text{textsize}</math></p> <p>where:</p> <table> <tr> <th>Parameter</th><th>Description</th></tr> <tr> <td>p</td><td>Number of static processes</td></tr> <tr> <td>stack</td><td>Sum of stack sizes of all static processes</td></tr> <tr> <td>pools</td><td>Sum of sizes of all message pools</td></tr> <tr> <td>mcb</td><td>module control block (see below)</td></tr> <tr> <td>textsize</td><td>Init size (see below)</td></tr> </table> <p>The size of the module control block (mcb) can be calculated according to the following formula (bytes): <math>\text{size\_mcb} = 96 + \text{friends} + \text{hooks} * 4 + c</math></p> <p>where:</p> <table> <tr> <th>Parameter</th><th>Description</th></tr> <tr> <td>friend</td><td>if friends are not used: 0 if friends are used 16 bytes</td></tr> <tr> <td>hooks</td><td>number of hooks configured</td></tr> <tr> <td>c</td><td>= 8 (for PowerPC)</td></tr> </table>	Parameter	Description	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)	Parameter	Description	friend	if friends are not used: 0 if friends are used 16 bytes	hooks	number of hooks configured	c	= 8 (for PowerPC)
Parameter	Description																				
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)																				
Parameter	Description																				
friend	if friends are not used: 0 if friends are used 16 bytes																				
hooks	number of hooks configured																				
c	= 8 (for PowerPC)																				

Parameter	Description
Init Size	<p>Size of the memory which is initialized by the C-startup function (cstartup.S).</p> <p>Best is to specify a label which will be resolved at link time (e.g. &lt;module_name&gt;_initsize). The specified label will be used in the linker script. Therefore all memory allocation for all modules is controlled by the linker script.</p> <p>You may specify an absolute address, but you need to be very carefully check with the linker script to avoid overlapping.</p>
Symbolic Values	<p>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).</p> <p>The labels (&lt;module_name&gt;_mod, &lt;module_name&gt;_size and &lt;module_name&gt;_initsize) will be used by the linker script and resolved at link time.</p> <p>Therefore all memory allocation for all modules is controlled by the linker script.</p>

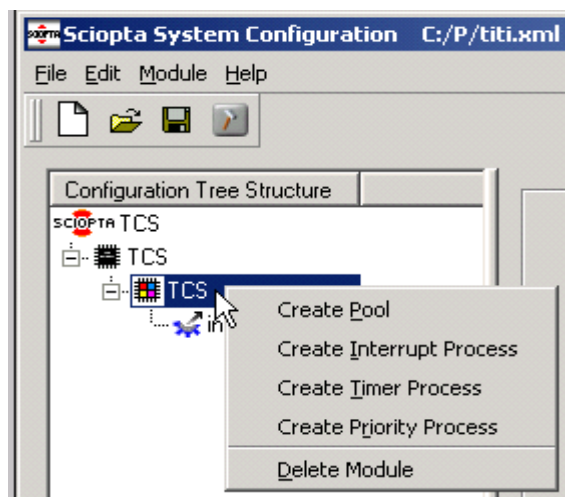
Applying Module Configuration

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



### 16.12 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.

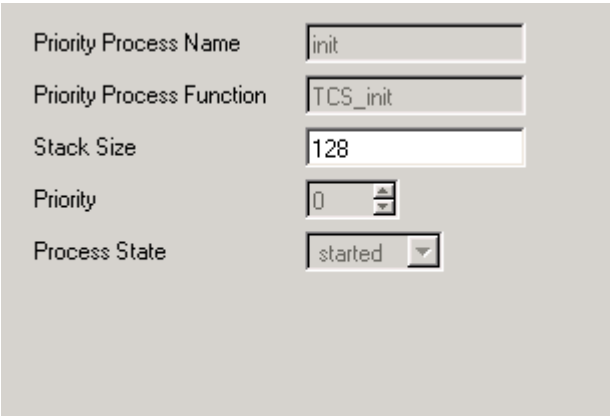


A pop-up menu appears and allows you to create pools, interrupt processes, timer processes and prioritized processes.

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

### 16.13 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.

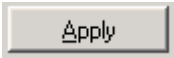


Priority Process Name	init
Priority Process Function	TCS_init
Stack Size	128
Priority	0
Process State	started

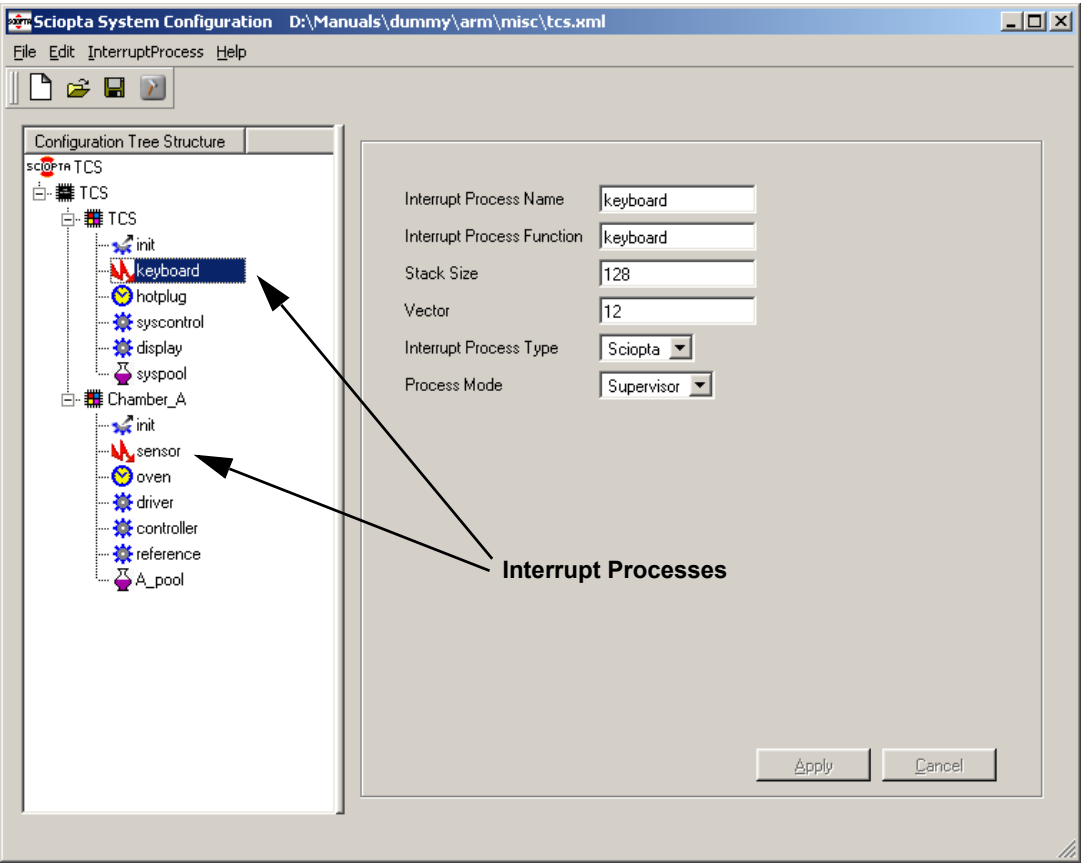
Parameter	Description
Priority Process Name	Name “init” defined automatically by the kernel.
Priority Process Function	Function <module_name>_init defined automatically by the kernel.
Stack Size	Enter a stack size for the init process.
Priority	Priority controlled automatically by the kernel.
Process State	Process state controlled automatically by the kernel.

#### Applying Init Process Configuration

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



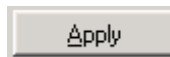
16.14 Interrupt Process Configuration



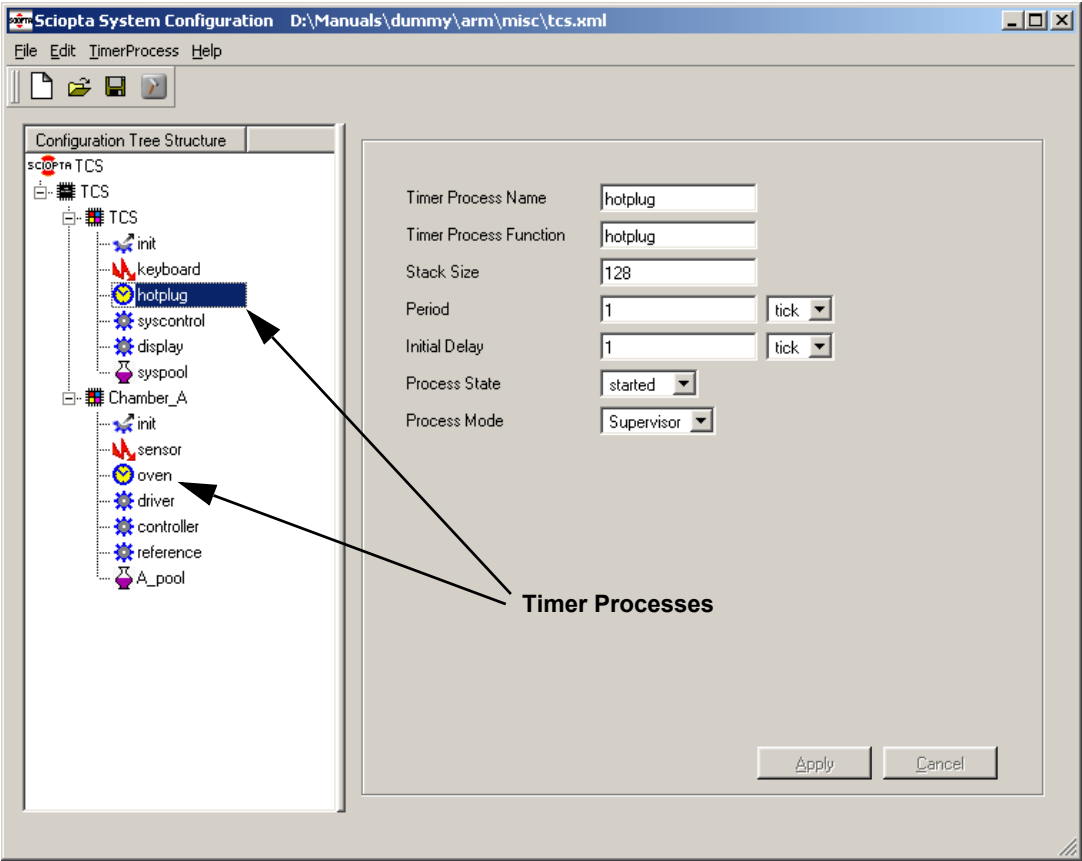
Parameter	Description	
Interrupt Process Name	Enter the name of the interrupt process.	
Interrupt Process Function	Function name of the interrupt process function. This is the address where the created process will start execution.	
Stack Size	Stacksize of the created process in bytes.	
Vector	Enter the interrupt vector connected to the interrupt process.	
Interrupt Process Type	Only interrupt processes of type Sciopta are supported. They are handled by the kernel and may use (not blocking) system calls.	
Process Mode	Enter the process mode.	
	Value	Description
	Supervisor	The process runs in CPU supervisor mode.
	User	The process runs in CPU user mode.

## Applying the Interrupt Process Configuration

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



16.15 Timer Process Configuration



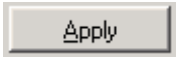
Parameter	Description
<b>Timer Process Name</b>	Enter the name of the timer process.
<b>Timer Process Function</b>	Function name of the timer process function. This is the address where the created process will start execution.
<b>Stack Size</b>	Stacksize of the created process in bytes.
<b>Period</b>	Period of time between calls to the timer process in ticks or in milliseconds.
<b>Initial Delay</b>	Initial delay in ticks before the first time call to the timer process in ticks or milliseconds.



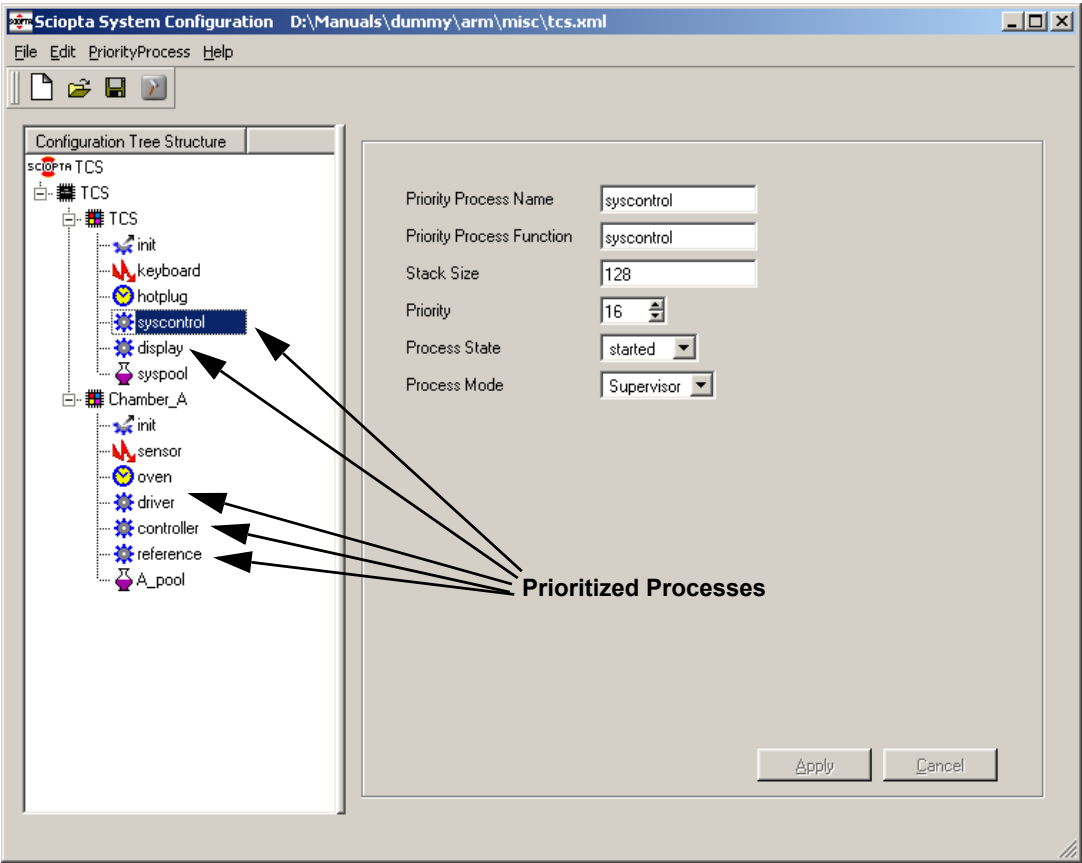
Parameter	Description	
Process State	Enter the state which the process should have after creation.	
	Value	Description
	started	The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes.
	stopped	The process is stopped. Use the <a href="#">sc_procStart</a> system call to start the process.
Process Mode	Enter the process mode.	
	Value	Description
	Supervisor	The process runs in CPU supervisor mode.
	User	The process runs in CPU user mode.

Applying the Timer Process Configuration

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



16.16 Prioritized Process Configuration

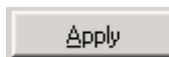


Parameter	Description
Prioritized Process Name	Enter the name of the prioritized process.
Prioritized Process Function	Function name of the prioritized process function. This is the address where the created process will start execution.
Stack Size	Stacksize of the created process in bytes.
Priority	Enter the process priority.  For process scheduling SCIOPTA uses a combination of the module priority and process priority called effective priority.

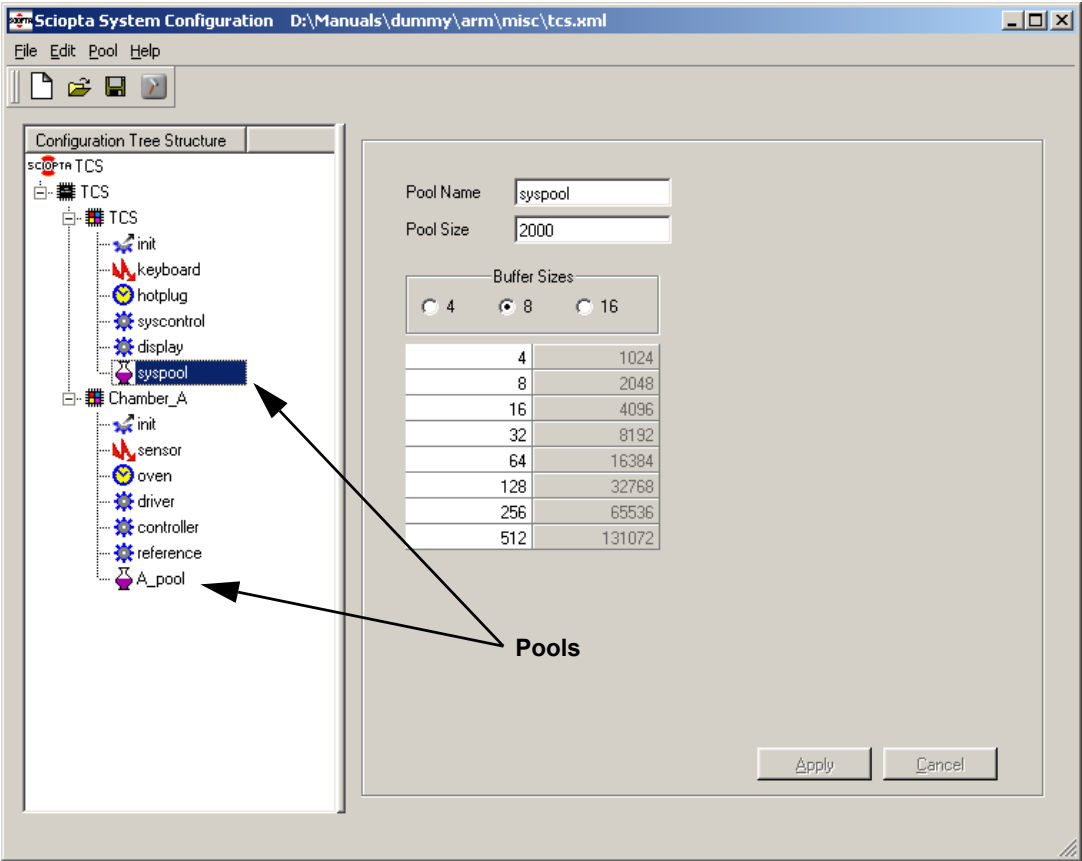
Parameter	Description	
Process State	Enter the state which the process should have after creation.	
	Value	Description
	started	The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes.
	stopped	The process is stopped. Use the <a href="#">sc_proeStart</a> system call to start the process.
Process Mode	Enter the process mode.	
	Value	Description
	Supervisor	The process runs in CPU supervisor mode.
	User	The process runs in CPU user mode.

## Applying the Priority Process Configuration

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



16.17 Pool Configuration

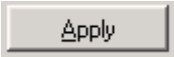


Parameter	Description
Pool Name	Enter the name of the message pool.

Parameter	Description					
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).					
	The size of the pool control block (pool_cb) can be calculated according to the following formula:					
	$\text{pool\_cb} = 68 + n * 20 + \text{stat} * n * 20$ <p>where:</p> <table> <tr> <th>Parameter</th><th>Description</th></tr> <tr> <td>n</td><td>buffer sizes (4, 8 or 16)</td></tr> <tr> <td>stat</td><td>process statistics or message statistics are used (1) or not used (0).</td></tr> </table>	Parameter	Description	n	buffer sizes (4, 8 or 16)	stat
Parameter	Description					
n	buffer sizes (4, 8 or 16)					
stat	process statistics or message statistics are used (1) or not used (0).					
Buffer Sizes	Select 4, 8 or 16 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.



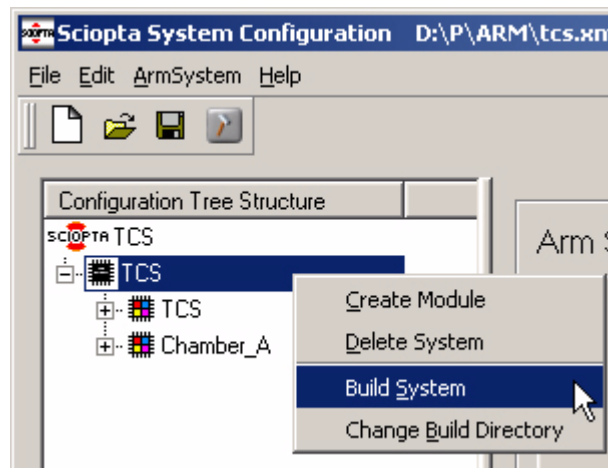
## 16.18 Build

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

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

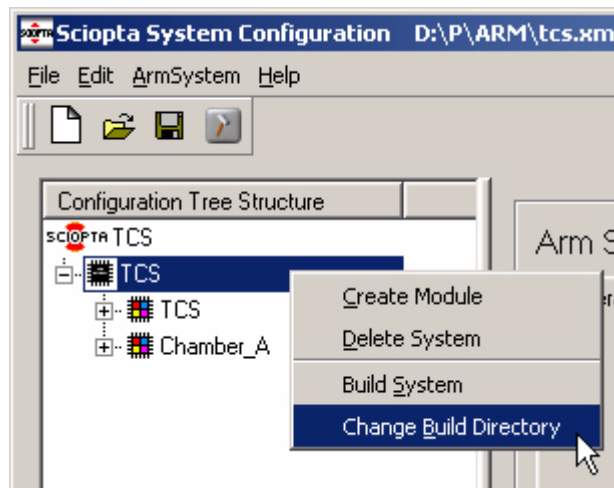
### 16.18.1 Build System

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.



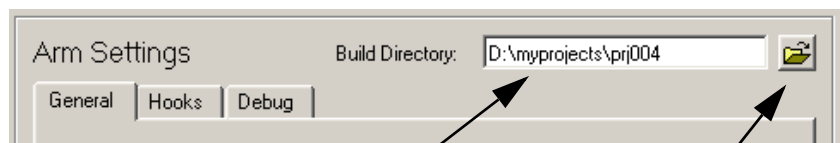
### 16.18.2 Change Build Directory

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:



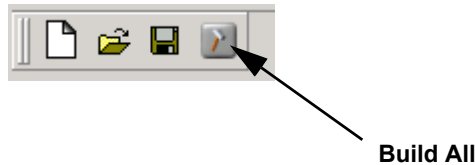
You can change the Build Directory also from the System Settings Window by entering directly the Build Directory Path.

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

### 16.18.3 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.**



### 16.19 Command Line Version

#### 16.19.1 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.

#### 16.19.2 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.

## 17 Manual Versions

### 17.1 Manual Version 1.1

- Chapter 2.4.11 STMicroelectronics STM32 Firmware Library, added.
- Chapter 2.4.12 Eclipse C/C++ Development Tooling - CDT, rewritten.
- Chapter 3 Getting Started, rewritten.
- Chapter 4.2 Files, missing files “cortexm3\_vector.x” included in the list. Files resethook.\* removed. File board-setup.c renamed to boardSetup.c.
- Chapter 4.11 Board Functions and Drivers, 9.7 “STMicroelectronics STM3210E-EVAL Evaluation Board” added. Files resethook.\* removed. File stm32f10x\_conf.h added.
- Chapter 4.15.2 Eclipse, rewritten.
- Chapter 4.6 Error Handling, error.c now new location.
- Chapter 4.9 ARM Cortex Family System Functions, files for IAR EW Version 4 included.
- Chapter 4.13 Kernel, kernel source file names corrected.
- Chapter 5.2.5 Priorities, upper limit effective priority of 31 described.
- Chapter 6.3.6 sc\_msgRx and chapter 6.3.1 sc\_msgAlloc, parameter tmo, value modified.
- Chapter 6.4.1 sc\_poolCreate, parameter size, pool calculation value n better defined. Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.4.3 sc\_poolIdGet, return value “poolID of default pool” added.
- Chapter 6.5.6 sc\_procPrioSet, paragraph: “If there are already existing processes at the same priority, the process which has just moved to that priority will be put at the end of the list and swapped-out.” added.
- Chapter 6.6.1 sc\_procPrioCreate, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.6.2 sc\_procIntCreate, “User” interrupt process type removed, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.6.3 sc\_procTimCreate, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.10.1 sc\_moduleIdGet, parameter name text: “...or zero for current module” appended at description and return value for parameter name=NULL added. Return value SC\_NOSUCH\_MODULE modified to SC\_ILLEGAL\_MID if Module name was not found.
- Chapter 6.11.1 sc\_moduleCreate, parameter textsize renamed into initsize. Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.12.3 sc\_moduleFriendGet, return value “1” modified to “!=”.
- Chapter 6.13.2 sc\_tick, last paragraph added.
- Chapter 6.15.2 sc\_triggerWait, parameter tmo better described.
- Chapter 6.19.1 sc\_miscErrorHookRegister, function and parameter newhook are of type pointer. Global and Module error hook difference described.
- Chapter 6.19.2 sc\_msgHookRegister, function and parameter newhook are of type pointer.
- Chapter 6.19.3 sc\_poolHookRegister, function and parameter newhook are of type pointer. Global and Module pool hook difference described.

- Chapter 6.19.4 `sc_procHookRegister`, function and parameter `newhook` are of type pointer. Global and Module process hook difference described.
- Chapter 7.7.2.7 Example, system call `sc_procIdGet` parameter corrected.
- Chapter 7.11.4 Error Hook Declaration Syntax, return value “!= 0” modified.
- Chapter 8.2.2 Reset Hook. Rewritten as the function `resethook()` is now included in the file `boardSetup.c`.
- Chapter 9.3 ARM Cortex Family System Functions, files for IAR EW Version 4 added.
- Chapter 9.4.1 General STM32 Functions and Definitions, device driver libraries are now outside of the `sciopta` delivery.
- Chapter 9.6.2 STM32-P103 General Board Functions and Definitions, file `stm32f10x_conf.h` added. File `reset-hook.S` replace by `boardSetup.c`.
- Chapter 9.7 STMicroelectronics STM3210E-EVAL Evaluation Board, added.
- Chapter 11.1 Kernel, file `sciopta.s79` added.
- Chapter 11.4.1 Include Files Search Directories, environment variable `STM32_FWLIB` added.
- Chapter 13.2.2 Environment Variables, environment variables `LMI_DRIVER` and `STM32_FWLIB` added.
- Chapter 13.3 Eclipse IDE and GNU GCC, rewritten.
- Chapter 13.4.3 Environment Variables, environment variables `LMI_DRIVER` and `STM32_FWLIB` added.
- Chapter 13.5.3 Environment Variables, environment variables `LMI_DRIVER` and `STM32_FWLIB` added.
- Chapter 15.3 Function Codes, code `0x0E`, `0x0D`, `0x3E`, `0x3F`, `0x57`, `0x5C`, `0x5D`, `0x5E` and `0x5F` added.
- Chapter 15.4 Error Codes, code `0x016`, `0x017` and `0x018` added.

### 17.2 Manual Version 1.0

- Initial Version

## 18 Index

### Symbols

__init_size .....	12-3
__mod .....	12-3
__pid .....	7-13
__size .....	12-3
__start .....	12-3

### A

Activate a Process Trigger .....	6-77
Add a Friend Module .....	6-64
Add a Time-Out Request .....	6-75
Addressed Process .....	5-7
Addressee of a Message .....	6-18
Addressing Processes .....	7-13
Allocate a Message .....	6-4
Allocate a Message and Clear Content .....	6-7
Allocate a Message with Time-Out .....	6-4
Allocate Memory .....	6-4
Application Configuration .....	8-1
Application Programming .....	7-1
ARM Cortex Family System Function .....	9-1
ARM CPU System Functions and Drivers .....	4-6
ARM Data Types types.h .....	11-9
ARM Family System Functions .....	4-5
ARM RealView Kernel Libraries .....	11-5
ARM RealView Linker Script .....	12-4
ARM RealView Stellaris Family Driver Libraries .....	11-7

### B

Board Configuration .....	4-7
Board Functions and Drivers .....	4-6
Board Setup .....	4-6
Board Support Package .....	4-6, 8-2, 9-1
Browser Window .....	16-4
BSP General System Functions .....	9-1
Build .....	16-26
Build a SCIOPTA System .....	4-9
Build All .....	16-28
Build Directory .....	16-27
Build Target .....	16-26
Build Temporary File .....	16-26
Building Kernel Libraries for ARM RealView .....	11-6
Building Kernel Libraries for GCC .....	11-2
Building Kernel Libraries for IAR .....	11-4
Building the Project .....	4-9

### C

C Environment .....	8-2
---------------------	-----

C Startup .....	8-3
C Startup Function .....	8-3
-c switch .....	16-29
Calculate a 16 Bit CRC .....	6-83
Calculates an Additional CRC .....	6-84
Call Kernel Tick .....	6-70
Call the Error Hook with a User Error .....	6-85
Cancel a Process Observation .....	6-52
Cancel a Process Supervision .....	6-52
Change Build Directory .....	16-27
Change the Owner of a Message .....	6-17
Change the Sender of a Message .....	6-12
Clear a Message Pool .....	6-29
C-Line .....	16-11
Coding .....	7-1
Compiler Optimization .....	11-1, 11-3, 11-5
Concurrent .....	7-29
config.exe .....	16-1
config.h .....	4-7
Configure the Project .....	16-5
Configuring Hooks .....	16-10
Configuring Modules .....	16-13
Configuring Target Systems .....	16-8
Configuring the INIT Process .....	16-17
CONNECTOR .....	1-1
CONNECTOR Process .....	6-81, 7-14
CONNECTOR System Calls .....	6-81
CONNECTOR, User's Guide .....	2-1
Convert Milliseconds in System Ticks .....	6-73
Convert System Ticks in Milliseconds .....	6-74
CPU Management .....	1-4
CRC .....	6-83, 6-84
CRC System Calls .....	6-83
Create a Message Pool .....	6-24
Create a Module .....	6-61, 7-2
Create a Prioritized Process .....	6-45
Create a Timer Process .....	6-47
Create an Interrupt Process .....	6-46
Creating a New Project .....	16-5
Creating Modules .....	16-12
Creating Pools .....	16-16
Creating Processes .....	16-16
Creating Systems .....	16-6
<b>D</b>	
Daemon .....	5-5, 7-21
Data Memory Map .....	12-5
Debug Configuration .....	16-11
Debugger Board Setup .....	4-7
Debugging .....	14-1
Default Pool .....	6-27

Define a Process Variable .....	6-54
Define all Modules as Friends .....	6-65
defines.h .....	11-9
Delete Process Variable .....	6-56
Distributed Multi-CPU Systems .....	1-1
Distributed Systems .....	5-16
Double Linked List .....	5-8
DRUID .....	1-1, 2-3, 14-1
DRUID, User's Guide .....	2-1
Dynamic Process .....	5-3, 7-13

## E

Eclipse .....	2-7, 4-9
Eclipse IDE and GNU GCC .....	3-2
Effective Priority .....	5-6
Embedded System .....	7-1
err.h .....	15-1
errno Variable .....	5-19, 6-86, 6-87
Error Check .....	5-19
Error Code .....	7-24, 15-1, 15-4
Error Function Code .....	7-24, 15-2
Error Handling .....	4-4, 5-19
Error Hook .....	4-4, 5-19, 5-20, 6-85, 6-88, 7-24, 7-25, 7-29, 15-1
Error Hook Declaration Syntax .....	7-25
Error Hook Example .....	7-26
Error Hook Registering .....	7-25
Error Hooks Return Behaviour .....	7-27
Error Include Files .....	15-1
Error Information .....	7-24
Error Number .....	6-86, 6-87
Error System Calls .....	6-85
Error Type .....	7-24, 15-1, 15-5
Error Word .....	7-24
error.c .....	4-4
errtxt.h .....	15-1
Exception handling .....	7-28
exit .....	7-6
External Process .....	6-81

## F

FAT File System .....	1-1
FAT File System, User's Guide .....	2-1
FLASH File System .....	1-1
FLASH Safe File System, User's Guide .....	2-1
Free a Message .....	6-8
Function Call .....	1-4
Function Code .....	15-1

## G

GCC Linker Script .....	12-1
GDD Generic Device Driver .....	11-2, 11-3, 11-5

General Configuration .....	16-8
General Stellaris Functions and Definitions .....	9-5
General STM32 Functions and Definitions .....	9-3
General System Functions .....	9-1
General System Functions and Drivers .....	4-5
Generating the Configuration Files .....	10-9
Get a Process Variable .....	6-55
Get Information About a Message Pool .....	6-28
Get Information About a Module .....	6-60
Get Module Friend Information .....	6-66
Get the Addressee of a Message .....	6-18
Get the Creator Process .....	6-32
Get the ID of a Message Pool .....	6-26
Get the ID of a Module .....	6-58
Get the ID of a Process .....	6-30
Get the Interrupt Vector .....	6-43
Get the Name of a Module .....	6-59
Get the Name of Process .....	6-33
Get the Owner of a Message .....	6-19
Get the Parent Process .....	6-32
Get the Path of a Process .....	6-34
Get the Pool ID of a Message .....	6-20
Get the Priority of a Process .....	6-35
Get the Process Error Number .....	6-86
Get the Process ID of a Static Process .....	7-13
Get the Requested Size of a Message .....	6-22
Get the Sender of a Message .....	6-21
Get the Tick Counter Value .....	6-71
Get the Time Slice of a Timer Process .....	6-39
Get the Value of a Process Trigger .....	6-79
Getting Started - Hello Example .....	3-1
Global Error Hook .....	5-19, 7-24, 15-1
Global Message Hook .....	6-89
Global Variable .....	1-4
Global Variables .....	7-29
GNU GCC Kernel Libraries .....	11-1
GNU GCC Stellaris Family Driver Libraries .....	11-7
GNU Tool Chain .....	2-5
<b>H</b>	
hardware .....	7-6
Hook .....	5-20
Hook Registering System Calls .....	6-88
<b>I</b>	
I/O-Ports .....	7-29
IAR Embedded Workbench for ARM .....	4-10
IAR Embedded Workbench Linker Script .....	12-4
IAR EW project file .....	4-10
IAR Kernel Libraries .....	11-3
IAR Stellaris Family Driver Libraries .....	11-7

iC3000.xjrf .....	4-9
iC3000.xqrf .....	4-9
IDE .....	13-1
Include Files .....	11-8
Include Files Search Directories .....	11-8
init .....	7-6
INIT Process .....	8-3, 10-4
Init Process .....	5-5, 7-4, 7-10, 16-7
INIT Process Stack Size .....	16-17
Initialize Process Variable Area .....	6-53
Input/Output management .....	1-4
Installation .....	2-1
Installation Location .....	2-3
Installation Procedure .....	2-2
Installed files .....	17-1
Internet Protocols .....	1-1
Interprocess Communication .....	1-3, 1-4, 5-7, 5-9, 7-14
Interrupt Handler .....	7-28
Interrupt Process .....	5-4, 7-4, 7-6, 7-12, 7-28, 10-5
Interrupt Process Configuration .....	16-18
Interrupt Process Declaration Syntax .....	7-7
Interrupt Process Priority .....	5-6
Interrupt Process Template .....	7-8
Interrupt Service Routine .....	7-6
interrupt vector table .....	7-28
IPS .....	1-1
IPS Applications .....	1-1
IPS Internet Protocols Applications, User's Guide .....	2-1
IPS Internet Protocols, User's Guide .....	2-1
iSYSTEM winIDEA .....	4-9
iSYSTEM© .....	13-5, 13-7

## K

Kernel Awareness .....	14-1
Kernel Awareness for Lauterbach Trace32 .....	14-2
Kernel Configuration .....	4-8, 10-1
Kernel Daemon .....	5-5, 6-45, 6-46, 6-47, 6-48, 6-61, 6-63, 7-22
Kernel Error Codes .....	15-1
Kernel Libraries .....	11-1
Kernel Tick Counter .....	6-70
Kernel Tick Function .....	6-70
Kernel, User's Guide .....	2-1
KERNEL_EALREADY_DEFINED .....	15-5
KERNEL_EENLARGE_MSG .....	15-4
KERNEL_EILL_BUF_SIZES .....	15-4
KERNEL_EILL_BUFSIZE .....	15-4
KERNEL_EILL_DEFPOOL_ID .....	15-4
KERNEL_EILL_INTERRUPT .....	15-4
KERNEL_EILL_MODULE .....	15-4
KERNEL_EILL_MODULE_NAME .....	15-4
KERNEL_EILL_NAME .....	15-4



KERNEL_EILL_NUM_SIZES .....	15-4
KERNEL_EILL_PARAMETER .....	15-5
KERNEL_EILL_PID .....	15-4
KERNEL_EILL_POOL_ID .....	15-4
KERNEL_EILL_POOL_SIZE .....	15-4
KERNEL_EILL_PRIORITY .....	15-4
KERNEL_EILL_PROC .....	15-4
KERNEL_EILL_PROC_NAME .....	15-5
KERNEL_EILL_PROCTYPE .....	15-4
KERNEL_EILL_SLICE .....	15-4
KERNEL_EILL_STACKSIZE .....	15-4
KERNEL_EILL_TARGET_NAME .....	15-4
KERNEL_EILL_VALUE .....	15-5
KERNEL_EILL_VECTOR .....	15-4
KERNEL_ELOCKED .....	15-4
KERNEL_EMODULE_TOO_SMALL .....	15-4
KERNEL_EMMSG_ENDMARK_CORRUPT .....	15-4
KERNEL_EMMSG_HD_CORRUPT .....	15-4
KERNEL_EMMSG_PREV_ENDMARK_CORRUPT .....	15-4
KERNEL_ENIL_PTR .....	15-4
KERNEL_ENO_KERNELD .....	15-4
KERNEL_ENO_MOORE_POOL .....	15-4
KERNEL_ENO_MORE_CONNECTOR .....	15-5
KERNEL_ENO_MORE_MODULE .....	15-4
KERNEL_ENO_MORE_PROC .....	15-4
KERNEL_ENOT_OWNER .....	15-4
KERNEL_EOUT_OF_MEMORY .....	15-4
KERNEL_EOUTSIDE_POOL .....	15-4
KERNEL_EPOOL_IN_USE .....	15-4
KERNEL_EPROC_NOT_PRIO .....	15-5
KERNEL_EPROC_TERMINATE .....	15-5
KERNEL_ESTACK_OVERFLOW .....	15-5
KERNEL_ESTACK_UNDERFLOW .....	15-5
KERNEL_ESTART_NOT_STOPPED .....	15-4
KERNEL_EUNLOCK_WO_LOCK .....	15-4
Kill a Message Pool .....	6-25
Kill a Module .....	6-63
Kill a Process .....	6-48

## L

Libraries and Include Files .....	4-8
Linker Control File .....	4-7
Linker Script .....	4-7
Linker script for ARM RealView .....	12-4
Linker script for GNU GCC .....	12-1
Linker script for IAR .....	12-4
Linux .....	2-1
LM3S6965 General Board Functions and Definitions .....	9-15
LM3S6965 LED Driver .....	9-15
Lock the Scheduler .....	6-37
Luminary LM3S6965 Board .....	9-13

Luminary® Stellaris Family Driver Libraries .....	11-7
Luminary® Stellaris Family Driver Library .....	9-16

## M

Mailbox Interprocess Communication .....	5-9
Main Installation Window .....	2-2
Makefile .....	4-9
map.c .....	12-4
Maximum Pools .....	16-15
Memory .....	7-3
Memory Fragmentation .....	5-8
Memory Management .....	1-4
Memory Management Unit .....	1-1
Memory Regions .....	12-1
Message .....	5-1, 5-7
Message Administration Block .....	7-20
Message Based RTOS .....	5-7
Message Declaration .....	7-14
Message Hook .....	5-20
Message Number .....	7-15
Message Number Definition .....	7-14
Message Owner .....	5-7
Message Ownership .....	6-4
Message passing .....	5-9
Message Pool .....	5-8, 6-4, 6-20, 7-19, 10-8
Message pool .....	6-24
Message Pool and Module .....	5-8
Message Queue .....	6-14
Message Size .....	5-7, 5-8, 6-23, 7-19
Message Statistics .....	16-11
Message Structure .....	7-15
Message Structure Definition .....	7-14
Message System Calls .....	6-4
Message Union .....	7-16
Message Union Declaration .....	7-14
MESSAGE_NAME .....	7-15
Messages and Modules .....	5-11
Methods .....	7-1
Microsoft® Windows XP .....	2-1
MMU .....	5-11, 7-23, 11-1, 11-3, 11-5
Modify a Process Variable .....	6-54
Module .....	5-10, 7-10, 10-2
Module Control Block .....	6-60
Module Create/Kill System Calls .....	6-61
Module Error Hook .....	5-19, 7-24, 15-1
Module Friend Concept .....	5-10
Module Friendship System Calls .....	6-64
Module Hook .....	8-4
Module Level .....	16-3, 16-4
Module Message Hook .....	6-89
Module Priority .....	5-6

Module size .....	12-2
Module Sizes .....	12-2
Module Status System Calls .....	6-58
module.h .....	6-60
Modules .....	7-1
msg .....	7-6
msg_nr .....	7-15
Multi-Module Systems .....	10-3
Mutual Exclusion .....	7-3
<b>N</b>	
Name of a process .....	6-33
New Button .....	16-5
<b>O</b>	
Observation .....	5-18
Observe a Process .....	6-51
Olimex STM32-P103 Board .....	9-7
On-chip Timer .....	6-70
Owner of a Message .....	6-17, 6-19
<b>P</b>	
Parameter Window .....	16-3
Password .....	2-2
Path of a process .....	6-34
PEG+, User's Guide .....	2-1
Peripheral Devices .....	7-3
Pool .....	7-19
Pool Configuration .....	16-24
Pool Control Block .....	6-28
Pool Create Hook .....	6-90
Pool Hook .....	5-20
Pool ID .....	6-20
Pool Kill Hook .....	6-90
Pool Name .....	16-24
Pool Parameter Check .....	16-11
Pool System Calls .....	6-24
Portable Embedded GUI .....	1-1
Pre-emptive Prioritized Scheduling .....	5-15
Prioritized Process .....	5-4, 7-4, 7-12, 10-7
Prioritized process .....	7-4
Prioritized Process Configuration .....	16-22
Prioritized Process Priority .....	5-6
Priority .....	5-6
Priority Levels .....	7-29
Priority Range .....	6-35
Process .....	7-4, 10-4
Process Categories .....	5-3
Process Create/Kill System Calls .....	6-45
Process Daemon .....	5-5, 6-30, 6-49, 6-50, 7-21
Process Daemon System Calls .....	6-49

Process Declaration Syntax .....	7-5
Process Hook .....	5-20, 6-91
Process ID .....	6-30, 7-4
Process Level .....	16-3, 16-4
Process Observation System Calls .....	6-51
Process Path .....	6-30
Process Priority .....	5-6
Process State .....	5-2
Process Statistics .....	16-11
Process Status System Calls .....	6-30
Process Synchronisation .....	5-12, 7-18
Process Ticks .....	6-39
Process Trigger .....	6-77
Process Type .....	5-4
Process Variable .....	5-13
Process Variables System Calls .....	6-53
Processes .....	5-2
Project Level .....	16-3, 16-4
Project Menu .....	16-6
Protected Memory Segment .....	5-11
<b>R</b>	
Read a Process Variable .....	6-55
Ready .....	5-2
Real-time Requirements .....	7-1
Receive a Message .....	6-14
Receive a Message with Time-Out .....	6-14
Re-entrant .....	7-29
Register a CONNECTOR Process .....	6-81
Register a Message Hook .....	6-89
Register a Pool Hook .....	6-90
Register a Process Daemon .....	6-49
Register a Process Hook .....	6-91
Register an Error Hook .....	6-88
Remote Process .....	5-17
Remove a CONNECTOR Process .....	6-82
Remove a Module from Friends .....	6-68
Remove a Process Variable .....	6-56
Remove a Time-Out Request .....	6-76
Remove a Whole Process Variable Area .....	6-57
Reply Message .....	7-29
Reset a Message Pool .....	6-29
Reset Hook .....	8-2
resethook.S .....	4-6
Resource Management .....	1-3, 7-3
Return a Message .....	6-8
Running .....	5-2
<b>S</b>	
sc_config.cfg .....	16-2
SC_CONNECTORREGISTER .....	15-3

sc_connectorRegister .....	6-81
SC_CONNECTORUNREGISTER .....	15-3
sc_connectorUnregister .....	6-82
SC_DEFAULT_POOL .....	6-27
SC_DISPATCHER .....	15-3
SC_ENDLESS_TMO .....	6-5, 6-14
SC_ERR_MODULE_FATAL .....	15-5
SC_ERR_MODULE_WARNING .....	15-5
SC_ERR_PROC_WARNING .....	15-5
SC_ERR_PROCESS_FATAL .....	15-5
SC_ERR_TARGET_FATAL .....	15-5
SC_ERR_TARGET_WARNING .....	15-5
SC_FATAL_IF_TMO .....	6-5
SC_INT_PROCESS .....	7-7, 7-8, 7-9
sc_kerneld .....	5-5
sc_miscCrc .....	6-83
sc_miscCrcContd .....	6-84
sc_miscErrnoGet .....	6-86
sc_miscErrnoSet .....	6-87
sc_miscError .....	6-85
sc_miscErrorHookRegister .....	6-88
SC_MODULECREATE .....	15-3
sc_moduleCreate .....	6-61
SC_MODULEFRIENDADD .....	15-3
sc_moduleFriendAdd .....	6-64
SC_MODULEFRIENDALL .....	15-3
sc_moduleFriendAll .....	6-65
SC_MODULEFRIENDGET .....	15-3
sc_moduleFriendGet .....	6-66
SC_MODULEFRIENDNON .....	15-3
sc_moduleFriendNone .....	6-67
SC_MODULEFRIENDRM .....	15-3
sc_moduleFriendRm .....	6-68
SC_MODULEIDGET .....	15-3
sc_moduleIdGet .....	6-58
SC_MODULEINFO .....	15-3
sc_moduleInfo .....	6-60
SC_MODULEKILL .....	15-3
sc_moduleKill .....	6-63
SC_MODULENAMEGET .....	15-3
sc_moduleNameGet .....	6-59
SC_MODULEPRIOGET .....	15-3
SC_MODULEPRIOSET .....	15-3
SC_MSGACQUIRE .....	15-2
sc_msgAcquire .....	6-17
SC_MSGADDRGET .....	15-2
sc_msgAddrGet .....	6-18
SC_MSGALLOC .....	15-2
sc_msgAlloc .....	6-4
sc_msgAllocClr .....	6-7
SC_MSGFREE .....	15-2
sc_msgFree .....	6-8

sc_msgHookRegister .....	6-89
SC_MSGOWNERGET .....	15-2
sc_msgOwnerGet .....	6-19
SC_MSGPOOLIDGET .....	15-2
sc_msgPoolIdGet .....	6-20
SC_MSGRX .....	15-2
sc_msgRx .....	6-14
SC_MSGRX_ALL .....	6-14
SC_MSGRX_BOTH .....	6-15
SC_MSGRX_MSGID .....	6-15
SC_MSGRX_NOT .....	6-15
SC_MSGRX_PID .....	6-15
SC_MSGSIZEGET .....	15-2
sc_msgSizeGet .....	6-22
SC_MSGSIZESET .....	15-2
sc_msgSizeSet .....	6-23
SC_MSGSNDGET .....	15-2
sc_msgSndGet .....	6-21
SC_MSGTX .....	15-2
sc_msgTx .....	6-10
SC_MSGTXALIAS .....	15-2
sc_msgTxAlias .....	6-12
SC_NO_TMO .....	6-5, 6-14
SC_POOLCREATE .....	15-2
sc_poolCreate .....	6-24
SC_POOLDEFAULT .....	15-2
sc_poolDefault .....	6-27
sc_poolHookRegister .....	6-90
SC_POOLIDGET .....	15-2
sc_poolIdGet .....	6-26
SC_POOLINFO .....	15-2
sc_poolInfo .....	6-28
SC_POOLKILL .....	15-2
sc_poolKill .....	6-25
SC_POOLRESET .....	15-2
sc_poolReset .....	6-29
sc_procCreate .....	7-13
sc_procd .....	5-5
sc_procDaemonRegister .....	6-49
sc_procDaemonUnregister .....	6-50
SC_PROCESS .....	7-5
sc_procHookRegister .....	6-91
SC_PROCIDGET .....	15-2
sc_procIdGet .....	6-30, 7-13
sc_procIdGet in Interrupt Processes .....	6-31
SC_PROCINTCREATE .....	15-2
sc_procIntCreate .....	6-46
SC_PROCINTCREATESTATIC .....	15-2
SC_PROCKILL .....	15-3
sc_procKill .....	6-48
SC_PROCNAMEGET .....	15-2
sc_procNameGet .....	6-33

SC_PROCNAMEGETMSG_REPLY .....	6-33, 6-34
SC_PROCOBSERVE .....	15-3
sc_procObserve .....	6-51
SC_PROCPATHGET .....	15-3
sc_procPathGet .....	6-34
SC_PROCPPIDGET .....	15-2
sc_procPpidGet .....	6-32
SC_PROCPRIOCREATE .....	15-2
SC_PROCPRIOCREATESTATIC .....	15-2
SC_PROCPRIOGET .....	15-2
sc_procPrioGet .....	6-35
SC_PROCPRIOSET .....	15-2
sc_procPrioSet .....	6-36
SC_PROCSCHEDLOCK .....	15-3
sc_procSchedLock .....	6-37
SC_PROCSCHEDUNLOCK .....	15-2
sc_procSchedUnLock .....	6-38
SC_PROCSLICEGET .....	15-2
sc_procSliceGet .....	6-39
SC_PROCSLICESET .....	15-2
sc_procSliceSet .....	6-40
SC_PROCSTART .....	15-2
sc_procStart .....	6-41
SC_PROCTOP .....	15-2
sc_procStop .....	6-42
SC_PROCTIMCREATE .....	15-3
sc_procTimCreate .....	6-47
SC_PROCTIMCREATESTATIC .....	15-2
SC_PROCUUNOBSERVE .....	15-3
sc_procUnobserve .....	6-52
SC_PROCURINTCREATE .....	15-3
SC_PROCURINTCREATESTATIC .....	15-2
SC_PROCVARDEL .....	15-3
sc_procVarDel .....	6-56
SC_PROCVARGET .....	15-3
sc_procVarGet .....	6-55
SC_PROCVARINIT .....	15-2
sc_procVarInit .....	6-53
SC_PROCVARRM .....	15-3
sc_procVarRm .....	6-57
SC_PROCVARSET .....	15-3
sc_procVarSet .....	6-54
sc_procVectorGet .....	6-43
SC_PROCYIELD .....	15-3
sc_procYield .....	6-44
SC_SET_MSGRX_HOOK .....	6-89
SC_SET_MSGTX_HOOK .....	6-89
SC_SET_POOLCREATE_HOOK .....	6-90
SC_SET_POOLKILL_HOOK .....	6-90
SC_SET_PROCCREATE_HOOK .....	6-91
SC_SET_PROCKILL_HOOK .....	6-91
SC_SET_PROCSWAP_HOOK .....	6-91

SC_SLEEP .....	15-3
sc_sleep .....	6-69
SC_SYSPPOOLKILL .....	15-2
SC_SYSPROCCREATE .....	15-3
sc_tick .....	6-70
sc_tickGet .....	6-71
sc_tickLength .....	6-72
sc_tickMs2Tick .....	6-73
sc_tickTick2Ms .....	6-74
SC_TMO .....	15-3
SC_TMO_MAX .....	6-5, 6-14
SC_TMOADD .....	15-3
sc_tmoAdd .....	6-75
SC_TMORM .....	15-3
sc_tmoRm .....	6-76
SC_TRIGGER .....	15-3
sc_trigger .....	6-77
SC_TRIGGERVALUEGET .....	15-3
sc_triggerValueGet .....	6-79
SC_TRIGGERVALUESET .....	15-3
sc_triggerValueSet .....	6-80
SC_TRIGGERWAIT .....	15-3
sc_triggerWait .....	6-78
SCAPI .....	1-1
Scheduler Lock Counter .....	6-37, 6-38
SCIOPTA API .....	1-1, 4-3
SCIOPTA Application .....	4-3
SCIOPTA ARM Cortex Exception Handling .....	7-28
SCIOPTA Connector .....	5-16
SCIOPTA delivery .....	2-1
SCIOPTA Design Rules .....	7-29
SCIOPTA Kernel .....	4-8, 11-1
SCIOPTA Kernel Project Overview .....	4-1
SCIOPTA Memory Manager .....	5-8
SCIOPTA Message .....	7-14
SCIOPTA Message Structure .....	5-7
SCIOPTA PEG .....	1-1
SCIOPTA Scheduling .....	5-15
SCIOPTA Simulator .....	1-1
SCIOPTA Start Sequence .....	8-1
SCIOPTA System Call .....	4-3
SCIOPTA Techniques and Concepts .....	4-2
SCIOPTA Technology and Methods .....	5-1
SCIOPTA Trigger .....	5-12, 7-18
sciopta.cmm .....	14-2
sciopta.cnf .....	4-8, 10-9, 10-10, 16-6
sciopta.h .....	7-5, 7-8, 11-8
sciopta.men .....	14-2
sciopta.S .....	4-8
sciopta.s .....	4-8
sciopta.s79 .....	4-8
sciopta.t32 .....	14-2



SCIOPTA_HOME Environment Variable .....	2-4
SCONF .....	4-8, 16-1, 16-27, 16-29
sconf .....	2-3
SCONF Command Line Version .....	16-29
sconf.c .....	4-8, 10-9, 10-10, 16-6
sconf.h .....	4-8, 10-9, 10-10, 11-8, 16-6
SCSIM .....	1-1
Selecting Process Type .....	7-12
Send a Message .....	6-10
Sender of a Message .....	6-21
Set a Message Pool as Default Pool .....	6-27
Set a Process Error Number .....	6-87
Set all Module as No-Friends .....	6-67
Set the Priority of a Process .....	6-36
Set the Size of a Message .....	6-23
Set the Tick Length .....	6-72
Set the Time Slice of Timer Process .....	6-40
Set the Value of a Trigger .....	6-80
Setting SCIOPTA Path Environment Variable .....	2-4
Setup a Process Variable Area .....	6-53
SFATFS .....	1-1
SFFS .....	1-1
Shared Resource .....	7-3
Shell .....	11-2, 11-3, 11-5
Short Cut .....	2-3
simple_uart.c .....	4-6
Size of a Message .....	6-22
Small Systems .....	10-2
SMMS .....	1-1
SMMS Memory Protection, User's Guide .....	2-1
Specific Module Values .....	12-3
Specification .....	7-1
Stack Requirements .....	7-29
Start a Process .....	6-41
Start Hook .....	4-4, 8-3
Start/Stop Counter .....	6-41, 6-42
Starting SCONF .....	16-1
Static Module .....	7-10
Static Process .....	5-3
Stellaris System Functions and Drivers .....	9-5
Stellaris System Tick Driver .....	9-6
Stellaris System Tick Interrupt Process .....	9-6
Step-By-Step Tutorial .....	3-3
STM32 System Functions and Drivers .....	9-3
STM32 System Tick Driver .....	9-4
STM32 System Tick Interrupt Process .....	9-4
STM32 UART Functions .....	9-4
STM32_FWLIB .....	2-6
STM3210E-EVAL General Board Functions and Definitions .....	9-12
STM3210E-EVAL LED Driver .....	9-12
STM32F10xFWLib .....	2-6
STM32-P103 General Board Functions and Definitions .....	9-9

STM32-P103 LED Driver .....	9-9
STMicroelectronics STM32 Firmware Library .....	2-6
Stop a Process .....	6-42
Structures .....	7-1
Supervise a Process .....	6-51
Supervisor Process .....	5-5, 7-4
syscall.S .....	7-23
System Analysis .....	7-1
System Call Group .....	6-1
System call reference .....	6-1
System Calls List .....	6-1
System Configuration .....	8-1
System Configuration and Initialization .....	4-4
System Design .....	7-1
System design .....	8-1
System error .....	6-85
System Latency .....	5-8
System Level Debugger .....	14-1
System Module .....	5-10
System Module Hook .....	8-4
System Partition .....	7-1
System Protection .....	5-11
System Requirements .....	2-1
System Reset .....	8-2
System Start .....	8-1
System Structure .....	5-10
System Tick .....	5-14, 7-9
System Ticks .....	6-69
System Timer .....	4-6
system.c .....	4-4
systick.c .....	4-6
<b>T</b>	
Target Level .....	16-3, 16-4
Target System .....	10-1
Techniques .....	7-1
Testing .....	7-1
Tick Timer .....	5-4
Time Management .....	1-3, 1-4, 5-14
Time-out Expired .....	6-76
Time-Out Server .....	5-14
Time-out Server .....	6-75
Time-out Server System Calls .....	6-75
Timer Process .....	5-4, 7-4, 7-9, 7-12, 10-6
Timer Process Configuration .....	16-20
Timer Process Declaration Syntax .....	7-9
Timer Process Priority .....	5-6
Timing System Calls .....	6-69
Transmitt a Message .....	6-10
Transmitting Process .....	5-7
Transparent Communication .....	5-17

Trap Interface .....	7-23, 11-1, 11-3, 11-5
Trigger .....	6-77, 7-18
trigger .....	7-6
Trigger System Calls .....	6-77
types.h .....	11-9
Typical files .....	4-1
<b>U</b>	
Universal Serial Bus .....	1-1
Unlock the Scheduler .....	6-38
Unregister a Process Daemon .....	6-50
USB Device, User's Guide .....	2-1
USB Host, User's Guide .....	2-1
USBD .....	1-1
USBH .....	1-1
User Module Hook .....	8-4
Utilities .....	11-2, 11-3, 11-5
<b>V</b>	
Vectorized interrupt .....	7-28
Version Number .....	2-3
<b>W</b>	
Wait on the Process Trigger .....	6-78
Waiting .....	5-2
Wanted Array in Receive Call .....	6-14
Windows .....	2-1
winIDEA .....	13-5, 13-7
<b>X</b>	
XML File .....	16-29
<b>Y</b>	
Yield the CPU .....	6-44