
**High Performance**
**Real-Time Operating Systems**

---

# SCIOPTA Kernel Manual

# Copyright



# Disclaimer

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

# Trademark

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

**Corporate Headquarters**

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

Document No. S09332RL1

SCIOPTA

# Table of Contents


**1. SCIOPTA Real Time Operating System ..... 1-1**

1.1 Introduction ..... 1-1
1.2 CPU Families ..... 1-2
1.3 SCIOPTA Kernels ..... 1-2
1.4 About this Manual ..... 1-2
1.5 SCIOPTA System Manuals ..... 1-2

**2. SCIOPTA Architecture ..... 2-1**

2.1 Introduction ..... 2-1
2.1.1 SCIOPTA Kernel V2 ..... 2-2
2.2 Modules ..... 2-3
2.2.1 Introduction ..... 2-3
2.2.2 System Module ..... 2-3
2.2.3 Module Priority ..... 2-3
2.2.4 System Protection ..... 2-4
2.2.5 SCIOPTA Module Friend Concept ..... 2-4
2.2.6 Module Creation ..... 2-5
2.2.6.1 Static Module Creation ..... 2-5
2.2.6.2 Dynamic Module Creation ..... 2-5
2.2.6.3 Module System Calls ..... 2-6
2.3 Processes ..... 2-7
2.3.1 Introduction ..... 2-7
2.3.2 Process States ..... 2-7
2.3.2.1 Running ..... 2-7
2.3.2.2 Ready ..... 2-7
2.3.2.3 Waiting ..... 2-7
2.3.3 Static Processes ..... 2-8
2.3.4 Dynamic Processes ..... 2-8
2.3.5 Process Identity ..... 2-9
2.3.6 Prioritized Processes ..... 2-9
2.3.6.1 Creating and Declaring Prioritized Processes ..... 2-10
2.3.6.2 Process Priorities ..... 2-10
2.3.6.3 Writing Prioritized Processes ..... 2-11
2.3.7 Interrupt Processes ..... 2-12
2.3.7.1 Creating and Declaring Interrupt Processes ..... 2-12
2.3.7.2 Interrupt Process Priorities ..... 2-13
2.3.7.3 Writing Interrupt Processes ..... 2-13
2.3.8 Timer Processes ..... 2-16
2.3.9 Creating and Declaring Timer Processes ..... 2-16
2.3.10 Timer Process Priorities ..... 2-16
2.3.11 Writing Timer Processes ..... 2-16
2.3.12 Init Processes ..... 2-17
2.3.12.1 Creating and Declaring Init Processes ..... 2-17
2.3.12.2 Init Process Priorities ..... 2-17
2.3.12.3 Writing Init Processes ..... 2-17
2.3.13 Daemons ..... 2-19
2.3.13.1 Process Daemons ..... 2-19
2.3.13.2 Kernel Daemon ..... 2-20
2.3.14 Process Stacks ..... 2-21

2.3.14.1 Unified Interrupt Stack for ARM Architecture ... 2-21
2.3.14.2 Interrupt Nesting for ARM Architecture ... 2-21
2.3.15 Stack Protector ... 2-21
2.3.16 Addressing Processes ... 2-22
2.3.16.1 Introduction ... 2-22
2.3.16.2 Get Process IDs of Static Processes ... 2-22
2.3.16.3 Get Process IDs of Dynamic Processes ... 2-22
2.3.17 Process Variables ... 2-23
2.3.18 Process Observation ... 2-24
2.4 Messages ... 2-25
2.4.1 Introduction ... 2-25
2.4.2 Message Structure ... 2-25
2.4.3 Message Size ... 2-26
2.4.3.1 Example ... 2-26
2.4.4 Message Pool ... 2-26
2.4.5 Message Passing ... 2-27
2.4.6 Message Declaration ... 2-28
2.4.6.1 Message Identifier ... 2-28
2.4.6.2 Message Structure ... 2-29
2.4.6.3 Message Union ... 2-30
2.4.7 Message Number (ID) Organization ... 2-31
2.4.7.1 Global Message Number Defines File ... 2-31
2.4.8 Example ... 2-31
2.4.9 Messages and Modules ... 2-32
2.4.10 Returning Sent Messages ... 2-33
2.4.11 Message Passing and Scheduling ... 2-34
2.4.12 Message Sent to Unknown Process ... 2-35
2.4.12.1 Example ... 2-35
2.5 Pools ... 2-36
2.5.1 Message Pool Size ... 2-36
2.5.2 Creating Pools ... 2-37
2.5.2.1 Static Pool Creation ... 2-37
2.5.2.2 Dynamic Pool Creation ... 2-37
2.6 Hooks ... 2-38
2.6.1 Introduction ... 2-38
2.6.2 Error Hook ... 2-38
2.6.3 Message Hooks ... 2-38
2.6.4 Process Hooks ... 2-38
2.6.5 Pool Hooks ... 2-38
2.7 System Start and Setup ... 2-39
2.7.1 Start Sequence ... 2-39
2.7.2 Reset Hook ... 2-40
2.7.2.1 Syntax ... 2-40
2.7.2.2 Parameter ... 2-40
2.7.2.3 Return Value ... 2-40
2.7.2.4 Location ... 2-40
2.7.3 C Startup ... 2-41
2.7.4 Starting the SCIOPTA Simulator ... 2-41
2.7.4.1 Module Data RAM ... 2-41
2.7.5 Start Hook ... 2-42
2.7.5.1 Syntax ... 2-42
2.7.5.2 Parameter ... 2-42

2.7.5.3 Return Value ..... 2-42
2.7.5.4 Location ..... 2-42
2.7.6 Init Processes ..... 2-42
2.7.7 Module Start Functions ..... 2-43
2.7.7.1 System Module Start Function ..... 2-43
2.7.7.2 User Module Start Function ..... 2-43
2.8 SCIOPTA Trigger ..... 2-44
2.8.1 Description ..... 2-44
2.8.2 Using SCIOPTA Trigger ..... 2-44
2.8.3 Trigger Example ..... 2-45
2.9 Time Management ..... 2-46
2.9.1 Introduction ..... 2-46
2.9.2 System Tick ..... 2-46
2.9.3 Configuring the System Tick ..... 2-46
2.9.4 External Tick Interrupt Process ..... 2-46
2.9.5 Timeout Server ..... 2-47
2.9.5.1 Introduction ..... 2-47
2.9.5.2 Using the Timeout Server ..... 2-47
2.10 Error Handling ..... 2-48
2.10.1 Introduction ..... 2-48
2.10.2 Error Sequence Kernel V1 ..... 2-48
2.10.3 Error Sequence Kernel V2 and V2INT ..... 2-48
2.10.4 Error Hook Kernel V1 ..... 2-49
2.10.5 Error Hook Kernel V2 and V2INT ..... 2-50
2.10.6 Error Information ..... 2-51
2.10.7 Error Hook Registering ..... 2-51
2.10.8 Error Hook Declaration Syntax Kernel V1 ..... 2-52
2.10.8.1 Description ..... 2-52
2.10.8.2 Syntax ..... 2-52
2.10.8.3 Parameter ..... 2-52
2.10.8.4 Error Hook Example ..... 2-53
2.10.9 Error Hook Declaration Syntax Kernel V2 and V2INT ..... 2-54
2.10.9.1 Description ..... 2-54
2.10.9.2 Syntax ..... 2-54
2.10.9.3 Parameter ..... 2-54
2.10.9.4 Kernel Error Message Structure ..... 2-54
2.10.9.5 Structure Members ..... 2-55
2.10.9.6 Header Files ..... 2-55
2.10.9.7 Error Hook Example ..... 2-56
2.10.10 Error Hooks Return Behaviour Kernel V1 ..... 2-57
2.10.11 Error Process Kernel V2 and V2INT ..... 2-58
2.10.11.1 Error Process Registering ..... 2-58
2.10.11.2 Example of an Error Process ..... 2-58
2.10.12 The Error Proxy Kernel V2 and V2INT ..... 2-59
2.10.12.1 Example ..... 2-59
2.10.13 The errno Variable ..... 2-60
2.11 Distributed Systems ..... 2-61
2.11.1 Introduction ..... 2-61
2.11.2 Connectors ..... 2-61
2.11.3 Transparent Communication ..... 2-62
2.11.4 Unknown Process ..... 2-62

**3. System Calls Overview ..... 3-1**
3.1 Introduction ..... 3-1
3.2 Message System Calls ..... 3-1
3.3 Process System Calls ..... 3-3
3.4 Module System Calls ..... 3-6
3.5 Message Pool Calls ..... 3-7
3.6 Safe Data Type Calls ..... 3-7
3.7 Timing Calls ..... 3-8
3.8 System Tick Calls ..... 3-8
3.9 Process Trigger Calls ..... 3-9
3.10 CONNECTOR Process Calls ..... 3-9
3.11 Miscellaneous and Error Calls ..... 3-9
3.12 Simulator Calls ..... 3-10

**4. System Calls Reference ..... 4-1**
4.1 Introduction ..... 4-1
4.2 sc_connectorRegister ..... 4-1
4.2.1 Description ..... 4-1
4.2.2 Syntax ..... 4-1
4.2.3 Parameter ..... 4-1
4.2.4 Return Value ..... 4-1
4.2.5 Example ..... 4-1
4.2.6 Errors ..... 4-1
4.3 sc_connectorRemote2Local ..... 4-3
4.3.1 Description ..... 4-3
4.3.2 Syntax ..... 4-3
4.3.3 Parameter ..... 4-3
4.3.4 Return Value ..... 4-3
4.3.5 Example ..... 4-3
4.3.6 Errors ..... 4-3
4.4 sc_connectorUnregister ..... 4-4
4.4.1 Description ..... 4-4
4.4.2 Syntax ..... 4-4
4.4.3 Parameter ..... 4-4
4.4.4 Return Value ..... 4-4
4.4.5 Example ..... 4-4
4.4.6 Errors ..... 4-4
4.5 sc_miscCrc ..... 4-5
4.5.1 Description ..... 4-5
4.5.2 Syntax ..... 4-5
4.5.3 Parameter ..... 4-5
4.5.4 Return Value ..... 4-5
4.5.5 Example ..... 4-5
4.5.6 Errors ..... 4-5
4.6 sc_miscCrcContd ..... 4-6
4.6.1 Description ..... 4-6
4.6.2 Syntax ..... 4-6
4.6.3 Parameter ..... 4-6
4.6.4 Return Value ..... 4-6
4.6.5 Example ..... 4-6
4.6.6 Errors ..... 4-6

4.7 sc_miscCrc32 ... 4-7
4.7.1 Description ... 4-7
4.7.2 Syntax ... 4-7
4.7.3 Parameter ... 4-7
4.7.4 Return Value ... 4-7
4.7.5 Example ... 4-7
4.7.6 Errors ... 4-7
4.8 sc_miscCrc32Contd ... 4-8
4.8.1 Description ... 4-8
4.8.2 Syntax ... 4-8
4.8.3 Parameter ... 4-8
4.8.4 Return Value ... 4-8
4.8.5 Example ... 4-8
4.8.6 Errors ... 4-8
4.9 sc_miscErrnoGet ... 4-9
4.9.1 Description ... 4-9
4.9.2 Syntax ... 4-9
4.9.3 Parameter ... 4-9
4.9.4 Return Value ... 4-9
4.9.5 Example ... 4-9
4.9.6 Errors ... 4-9
4.10 sc_miscErrnoSet ... 4-10
4.10.1 Description ... 4-10
4.10.2 Syntax ... 4-10
4.10.3 Parameter ... 4-10
4.10.4 Return Value ... 4-10
4.10.5 Example ... 4-10
4.10.6 Errors ... 4-10
4.11 sc_miscError ... 4-11
4.11.1 Description ... 4-11
4.11.2 Syntax ... 4-11
4.11.3 Parameter ... 4-11
4.11.4 Return Value ... 4-11
4.11.5 Example ... 4-11
4.11.6 Errors ... 4-11
4.12 sc_miscError2 ... 4-12
4.12.1 Description ... 4-12
4.12.2 Syntax ... 4-12
4.12.3 Parameter ... 4-12
4.12.4 Return Value ... 4-12
4.12.5 Example ... 4-12
4.12.6 Errors ... 4-12
4.13 sc_miscErrorHookRegister ... 4-13
4.13.1 Description ... 4-13
4.13.2 Syntax ... 4-13
4.13.3 Parameter ... 4-13
4.13.4 Return Value ... 4-13
4.13.5 Example ... 4-13
4.13.6 Errors ... 4-13
4.14 sc_miscFlowSignatureGet ... 4-14
4.14.1 Description ... 4-14
4.14.2 Syntax ... 4-14

4.14.3 Parameter ..... 4-14
4.14.4 Return Value ..... 4-14
4.14.5 Example ..... 4-14
4.14.6 Errors ..... 4-14
4.15 sc_miscFlowSignatureInit ..... 4-15
4.15.1 Description ..... 4-15
4.15.2 Syntax ..... 4-15
4.15.3 Parameter ..... 4-15
4.15.4 Return Value ..... 4-15
4.15.5 Example ..... 4-15
4.15.6 Errors ..... 4-15
4.16 sc_miscFlowSignatureUpdate ..... 4-16
4.16.1 Description ..... 4-16
4.16.2 Syntax ..... 4-16
4.16.3 Parameter ..... 4-16
4.16.4 Return Value ..... 4-16
4.16.5 Example ..... 4-16
4.16.6 Errors ..... 4-16
4.17 sc_moduleCBChk ..... 4-17
4.17.1 Description ..... 4-17
4.17.2 Syntax ..... 4-17
4.17.3 Parameter ..... 4-17
4.17.4 Return Value ..... 4-17
4.17.5 Example ..... 4-17
4.17.6 Errors ..... 4-17
4.18 sc_moduleCreate ..... 4-18
4.18.1 Description ..... 4-18
4.18.2 Syntax ..... 4-18
4.18.3 Parameter ..... 4-19
4.18.4 Return Value ..... 4-19
4.18.5 Example ..... 4-20
4.18.6 Errors ..... 4-20
4.19 sc_moduleCreate2 ..... 4-21
4.19.1 Description ..... 4-21
4.19.2 Syntax ..... 4-21
4.19.3 Parameter ..... 4-21
4.19.4 Return Value ..... 4-21
4.19.5 Module Descriptor Block mdb ..... 4-22
4.19.6 Structure Members ..... 4-22
4.19.7 Module Address and Size ..... 4-23
4.19.8 Structure Members ..... 4-23
4.19.9 Example ..... 4-24
4.19.10 Errors ..... 4-24
4.20 sc_moduleFriendAdd ..... 4-26
4.20.1 Description ..... 4-26
4.20.2 Syntax ..... 4-26
4.20.3 Parameter ..... 4-26
4.20.4 Return Value ..... 4-26
4.20.5 Errors ..... 4-26
4.21 sc_moduleFriendAll ..... 4-27
4.21.1 Description ..... 4-27
4.21.2 Syntax ..... 4-27

4.21.3 Parameter ... 4-27
4.21.4 Return Value ... 4-27
4.21.5 Errors ... 4-27
4.22 sc_moduleFriendGet ... 4-28
4.22.1 Description ... 4-28
4.22.2 Syntax ... 4-28
4.22.3 Parameter ... 4-28
4.22.4 Return Value ... 4-28
4.22.5 Errors ... 4-28
4.23 sc_moduleFriendNone ... 4-29
4.23.1 Description ... 4-29
4.23.2 Syntax ... 4-29
4.23.3 Parameter ... 4-29
4.23.4 Return Value ... 4-29
4.23.5 Errors ... 4-29
4.24 sc_moduleFriendRm ... 4-30
4.24.1 Description ... 4-30
4.24.2 Syntax ... 4-30
4.24.3 Parameter ... 4-30
4.24.4 Return Value ... 4-30
4.24.5 Errors ... 4-30
4.25 sc_moduleIdGet ... 4-31
4.25.1 Description ... 4-31
4.25.2 Syntax ... 4-31
4.25.3 Parameter ... 4-31
4.25.4 Return Value ... 4-31
4.25.5 Example ... 4-31
4.25.5.1 Errors ... 4-31
4.26 sc_moduleInfo ... 4-32
4.26.1 Description ... 4-32
4.26.2 Syntax ... 4-32
4.26.3 Parameter ... 4-32
4.26.4 Return Value ... 4-32
4.26.5 Module Info Structure ... 4-33
4.26.6 Structure Members ... 4-33
4.26.7 Example ... 4-34
4.26.8 Errors ... 4-34
4.27 sc_moduleKill ... 4-35
4.27.1 Description ... 4-35
4.27.2 Syntax ... 4-35
4.27.3 Parameter ... 4-35
4.27.4 Return Value ... 4-35
4.27.5 Example ... 4-35
4.27.6 Errors ... 4-36
4.28 sc_moduleNameGet ... 4-37
4.28.1 Description ... 4-37
4.28.2 Syntax ... 4-37
4.28.3 Parameter ... 4-37
4.28.4 Return Value ... 4-37
4.28.5 Example ... 4-37
4.28.6 Errors ... 4-37
4.29 sc_modulePrioGet ... 4-38

4.29.1 Description ... 4-38
4.29.2 Syntax ... 4-38
4.29.3 Parameter ... 4-38
4.29.4 Return Value ... 4-38
4.29.5 Example ... 4-38
4.29.6 Errors ... 4-38
4.30 sc_moduleStop ... 4-39
4.30.1 Description ... 4-39
4.30.2 Syntax ... 4-39
4.30.3 Parameter ... 4-39
4.30.4 Return Value ... 4-39
4.30.5 Example ... 4-39
4.30.6 Errors ... 4-39
4.31 sc_msgAcquire ... 4-40
4.31.1 Description ... 4-40
4.31.2 Syntax ... 4-40
4.31.3 Parameter ... 4-40
4.31.4 Return Value ... 4-40
4.31.5 Example ... 4-40
4.31.6 Errors ... 4-41
4.32 sc_msgAddrGet ... 4-42
4.32.1 Description ... 4-42
4.32.2 Syntax ... 4-42
4.32.3 Parameter ... 4-42
4.32.4 Return Value ... 4-42
4.32.5 Example ... 4-42
4.32.6 Errors ... 4-43
4.33 sc_msgAlloc ... 4-44
4.33.1 Description ... 4-44
4.33.2 Syntax ... 4-44
4.33.3 Parameter ... 4-44
4.33.4 Return Value ... 4-45
4.33.5 Example ... 4-45
4.33.6 Errors ... 4-46
4.34 sc_msgAllocClr ... 4-47
4.34.1 Description ... 4-47
4.34.2 Syntax ... 4-47
4.34.3 Parameter ... 4-47
4.34.4 Return Value ... 4-47
4.34.5 Example ... 4-47
4.34.6 Error ... 4-47
4.35 sc_msgAllocTx ... 4-48
4.35.1 Description ... 4-48
4.35.2 Syntax ... 4-48
4.35.3 Parameter ... 4-48
4.35.4 Return Value ... 4-48
4.35.5 Example ... 4-48
4.35.6 Error ... 4-49
4.36 sc_msgDataCrcDis ... 4-50
4.36.1 Description ... 4-50
4.36.2 Syntax ... 4-50
4.36.3 Parameter ... 4-50

4.36.4 Return Value ..... 4-50
4.36.5 Example ..... 4-50
4.36.6 Error ..... 4-51
4.37 sc_msgDataCrcGet ..... 4-52
4.37.1 Description ..... 4-52
4.37.2 Syntax ..... 4-52
4.37.3 Parameter ..... 4-52
4.37.4 Return Value ..... 4-52
4.37.5 Example ..... 4-52
4.37.6 Error ..... 4-53
4.38 sc_msgDataCrcSet ..... 4-54
4.38.1 Description ..... 4-54
4.38.2 Syntax ..... 4-54
4.38.3 Parameter ..... 4-54
4.38.4 Return Value ..... 4-54
4.38.5 Example ..... 4-54
4.38.6 Error ..... 4-55
4.39 sc_msgFind ..... 4-56
4.39.1 Description ..... 4-56
4.39.2 Syntax ..... 4-56
4.39.3 Parameter ..... 4-56
4.39.4 Return Value ..... 4-57
4.39.5 Examples ..... 4-57
4.39.6 Errors ..... 4-57
4.40 sc_msgFlowSignatureUpdate ..... 4-58
4.40.1 Description ..... 4-58
4.40.2 Syntax ..... 4-58
4.40.3 Parameter ..... 4-58
4.40.4 Return Value ..... 4-58
4.40.5 Example ..... 4-58
4.40.6 Error ..... 4-59
4.41 sc_msgFree ..... 4-60
4.41.1 Description ..... 4-60
4.41.2 Syntax ..... 4-60
4.41.3 Parameter ..... 4-60
4.41.4 Return Value ..... 4-60
4.41.5 Example ..... 4-60
4.41.6 Errors ..... 4-61
4.42 sc_msgHdCheck ..... 4-62
4.42.1 Description ..... 4-62
4.42.2 Syntax ..... 4-62
4.42.3 Parameter ..... 4-62
4.42.4 Return Value ..... 4-62
4.42.5 Example ..... 4-62
4.42.6 Errors ..... 4-62
4.43 sc_msgHookRegister ..... 4-63
4.43.1 Description ..... 4-63
4.43.2 Syntax ..... 4-63
4.43.3 Parameter ..... 4-63
4.43.4 Return Value ..... 4-63
4.43.5 Example ..... 4-64
4.43.6 Errors ..... 4-64

4.44 sc_msgOwnerGet ... 4-65
4.44.1 Description ... 4-65
4.44.2 Syntax ... 4-65
4.44.3 Parameter ... 4-65
4.44.4 Return Value ... 4-65
4.44.5 Example ... 4-65
4.44.6 Errors ... 4-66
4.45 sc_msgPoolIdGet ... 4-67
4.45.1 Description ... 4-67
4.45.2 Systax ... 4-67
4.45.3 Parameter ... 4-67
4.45.4 Return Value ... 4-67
4.45.5 Example ... 4-67
4.45.6 Errors ... 4-68
4.46 sc_msgRx ... 4-69
4.46.1 Description ... 4-69
4.46.2 Syntax ... 4-69
4.46.3 Parameter ... 4-70
4.46.4 Return Value ... 4-70
4.46.5 Examples ... 4-70
4.46.6 Errors ... 4-71
4.47 sc_msgSizeGet ... 4-72
4.47.1 Description ... 4-72
4.47.2 Syntax ... 4-72
4.47.3 Parameter ... 4-72
4.47.4 Return Value ... 4-72
4.47.5 Example ... 4-72
4.47.6 Errors ... 4-73
4.48 sc_msgSizeSet ... 4-74
4.48.1 Description ... 4-74
4.48.2 Syntax ... 4-74
4.48.3 Parameter ... 4-74
4.48.4 Return Value ... 4-74
4.48.5 Example ... 4-74
4.48.6 Errors ... 4-75
4.49 sc_msgSndGet ... 4-76
4.49.1 Description ... 4-76
4.49.2 Syntax ... 4-76
4.49.3 Parameter ... 4-76
4.49.4 Return Value ... 4-76
4.49.5 Example ... 4-76
4.49.6 Errors ... 4-77
4.50 sc_msgTx ... 4-78
4.50.1 Description ... 4-78
4.50.2 Syntax ... 4-78
4.50.3 Parameter ... 4-78
4.50.4 Return Value ... 4-79
4.50.5 Example ... 4-79
4.50.6 Errors ... 4-79
4.51 sc_msgTxAlias ... 4-81
4.51.1 Description ... 4-81
4.51.2 Syntax ... 4-81

4.51.3 Parameter ... 4-81
4.51.4 Return Value ... 4-81
4.51.5 Example ... 4-82
4.51.6 Errors ... 4-82
4.52 sc_poolCBChk ... 4-83
4.52.1 Description ... 4-83
4.52.2 Syntax ... 4-83
4.52.3 Parameter ... 4-83
4.52.4 Return Value ... 4-83
4.52.5 Example ... 4-83
4.52.6 Errors ... 4-84
4.53 sc_poolCreate ... 4-85
4.53.1 Description ... 4-85
4.53.2 Syntax ... 4-85
4.53.3 Parameter ... 4-85
4.53.4 Return Value ... 4-86
4.53.5 Example ... 4-86
4.53.6 Errors ... 4-86
4.54 sc_poolDefault ... 4-88
4.54.1 Description ... 4-88
4.54.2 Syntax ... 4-88
4.54.3 Parameter ... 4-88
4.54.4 Return Value ... 4-88
4.54.5 Example ... 4-88
4.54.6 Errors ... 4-88
4.55 sc_poolHookRegister ... 4-89
4.55.1 Description ... 4-89
4.55.2 Syntax ... 4-89
4.55.3 Parameter ... 4-89
4.55.4 Return Value ... 4-89
4.55.5 Example ... 4-90
4.55.6 Errors ... 4-90
4.56 sc_poolIdGet ... 4-91
4.56.1 Description ... 4-91
4.56.2 Syntax ... 4-91
4.56.3 Parameter ... 4-91
4.56.4 Return Value ... 4-91
4.56.5 Example ... 4-91
4.56.6 Errors ... 4-91
4.57 sc_poolInfo ... 4-92
4.57.1 Description ... 4-92
4.57.2 Syntax ... 4-92
4.57.3 Parameter ... 4-92
4.57.4 Return Value ... 4-92
4.57.5 Pool Control Block Structure ... 4-93
4.57.6 Structure Members ... 4-93
4.57.7 Pool Statistics Info Structure ... 4-94
4.57.8 Structure Members ... 4-94
4.57.9 Example ... 4-95
4.57.10 Errors ... 4-95
4.58 sc_poolKill ... 4-96
4.58.1 Description ... 4-96

4.58.2 Syntax ..... 4-96
4.58.3 Parameter ..... 4-96
4.58.4 Return Value ..... 4-96
4.58.5 Example ..... 4-96
4.58.6 Errors ..... 4-97
4.59 sc_poolReset ..... 4-98
4.59.1 Description ..... 4-98
4.59.2 Syntax ..... 4-98
4.59.3 Parameter ..... 4-98
4.59.4 Return Value ..... 4-98
4.59.5 Example ..... 4-98
4.59.6 Errors ..... 4-98
4.60 sc_procAtExit ..... 4-100
4.60.1 Description ..... 4-100
4.60.2 Syntax ..... 4-100
4.60.3 Parameter ..... 4-100
4.60.4 Return Value ..... 4-100
4.60.5 Example ..... 4-100
4.60.6 Errors ..... 4-101
4.61 sc_procAttrGet ..... 4-102
4.61.1 Description ..... 4-102
4.61.2 Syntax ..... 4-102
4.61.3 Parameter ..... 4-102
4.61.4 Return Value ..... 4-103
4.61.5 Example ..... 4-103
4.61.6 Errors ..... 4-103
4.62 sc_procCBChk ..... 4-104
4.62.1 Description ..... 4-104
4.62.2 Syntax ..... 4-104
4.62.3 Parameter ..... 4-104
4.62.4 Return Value ..... 4-105
4.62.5 Example ..... 4-105
4.62.6 Errors ..... 4-105
4.63 sc_procCreate2 ..... 4-106
4.63.1 Description ..... 4-106
4.63.2 Syntax ..... 4-106
4.63.3 Parameter ..... 4-106
4.63.4 Return Value ..... 4-106
4.63.5 Process Descriptor Block pdb ..... 4-107
4.63.6 Structure Members Common for all Process Types ..... 4-108
4.63.7 Additional Structure Members for Prioritized Processes ..... 4-109
4.63.8 Additional Structure Members for Interrupt Processes ..... 4-109
4.63.9 Additional Structure Members for Timer Processes ..... 4-109
4.63.10 Example ..... 4-110
4.63.11 Errors ..... 4-110
4.64 sc_procDaemonRegister ..... 4-114
4.64.1 Description ..... 4-114
4.64.2 Syntax ..... 4-114
4.64.3 Parameter ..... 4-114
4.64.4 Return Value ..... 4-114
4.64.5 Errors ..... 4-114
4.65 sc_procDaemonUnregister ..... 4-115

4.65.1 Description ..... 4-115
4.65.2 Syntax ..... 4-115
4.65.3 Parameter ..... 4-115
4.65.4 Return Value ..... 4-115
4.65.5 Errors ..... 4-115
4.66 sc_procFlowSignatureGet ..... 4-116
4.66.1 Description ..... 4-116
4.66.2 Syntax ..... 4-116
4.66.3 Parameter ..... 4-116
4.66.4 Return Value ..... 4-116
4.66.5 Example ..... 4-116
4.66.6 Errors ..... 4-116
4.67 sc_procFlowSignatureInit ..... 4-117
4.67.1 Description ..... 4-117
4.67.2 Syntax ..... 4-117
4.67.3 Parameter ..... 4-117
4.67.4 Return Value ..... 4-117
4.67.5 Example ..... 4-117
4.67.6 Errors ..... 4-117
4.68 sc_procFlowSignatureUpdate ..... 4-118
4.68.1 Description ..... 4-118
4.68.2 Syntax ..... 4-118
4.68.3 Parameter ..... 4-118
4.68.4 Return Value ..... 4-118
4.68.5 Example ..... 4-118
4.68.6 Errors ..... 4-118
4.69 sc_procHookRegister ..... 4-119
4.69.1 Description ..... 4-119
4.69.2 Syntax ..... 4-119
4.69.3 Parameter ..... 4-119
4.69.4 Return Value ..... 4-119
4.69.5 Example ..... 4-120
4.69.6 Errors ..... 4-120
4.70 sc_procIdGet ..... 4-121
4.70.1 Description ..... 4-121
4.70.2 Syntax ..... 4-121
4.70.3 Parameter ..... 4-121
4.70.4 Return Value ..... 4-122
4.70.5 sc_procIdGet in Interrupt Processes ..... 4-122
4.70.6 Example ..... 4-122
4.70.7 Errors ..... 4-122
4.71 sc_procIntCreate ..... 4-123
4.71.1 Description ..... 4-123
4.71.2 Syntax ..... 4-123
4.71.3 Parameter ..... 4-123
4.71.4 Return Value ..... 4-124
4.71.5 Example ..... 4-124
4.71.6 Errors ..... 4-124
4.72 sc_procIrqRegister ..... 4-125
4.72.1 Description ..... 4-125
4.72.2 Syntax ..... 4-125
4.72.3 Parameter ..... 4-125

4.72.4 Return Value ..... 4-125
4.72.5 Example ..... 4-125
4.72.6 Errors ..... 4-126
4.73 sc_procIrqUnregister ..... 4-127
4.73.1 Description ..... 4-127
4.73.2 Syntax ..... 4-127
4.73.3 Parameter ..... 4-127
4.73.4 Return Value ..... 4-127
4.73.5 Example ..... 4-127
4.73.6 Errors ..... 4-127
4.74 sc_procKill ..... 4-128
4.74.1 Description ..... 4-128
4.74.2 Syntax ..... 4-128
4.74.3 Parameter ..... 4-128
4.74.4 Return Value ..... 4-128
4.74.5 Example ..... 4-128
4.74.6 Errors ..... 4-129
4.75 sc_procNameGet ..... 4-130
4.75.1 Description ..... 4-130
4.75.2 Syntax ..... 4-130
4.75.3 Parameter ..... 4-130
4.75.4 Return Value ..... 4-130
4.75.5 Example ..... 4-131
4.75.6 Errors ..... 4-131
4.76 sc_procObserve ..... 4-132
4.76.1 Description ..... 4-132
4.76.2 Syntax ..... 4-132
4.76.3 Parameter ..... 4-132
4.76.4 Return Value ..... 4-132
4.76.5 Example ..... 4-133
4.76.6 Errors ..... 4-133
4.77 sc_procPathCheck ..... 4-134
4.77.1 Description ..... 4-134
4.77.2 Syntax ..... 4-134
4.77.3 Parameter ..... 4-134
4.77.4 Return Value ..... 4-134
4.77.5 Example ..... 4-134
4.77.6 Errors ..... 4-134
4.78 sc_procPathGet ..... 4-135
4.78.1 Description ..... 4-135
4.78.2 Syntax ..... 4-135
4.78.3 Parameter ..... 4-135
4.78.4 Return Value ..... 4-136
4.78.5 Example ..... 4-136
4.78.6 Errors ..... 4-136
4.79 sc_procPpidGet ..... 4-137
4.79.1 Description ..... 4-137
4.79.2 Syntax ..... 4-137
4.79.3 Parameter ..... 4-137
4.79.4 Return Value ..... 4-137
4.79.5 Example ..... 4-138
4.79.6 Errors ..... 4-138

4.80 sc_procPrioCreate ..... 4-139
4.80.1 Description ..... 4-139
4.80.2 Syntax ..... 4-139
4.80.3 Parameter ..... 4-139
4.80.4 Return Value ..... 4-140
4.80.5 Example ..... 4-140
4.80.6 Errors ..... 4-140
4.81 sc_procPrioGet ..... 4-142
4.81.1 Description ..... 4-142
4.81.2 Syntax ..... 4-142
4.81.3 Parameter ..... 4-142
4.81.4 Return Value ..... 4-142
4.81.5 Example ..... 4-143
4.81.6 Errors ..... 4-143
4.82 sc_procPrioSet ..... 4-144
4.82.1 Description ..... 4-144
4.82.2 Syntax ..... 4-144
4.82.3 Parameter ..... 4-144
4.82.4 Return Value ..... 4-144
4.82.5 Example ..... 4-144
4.82.6 Errors ..... 4-145
4.83 sc_procSchedLock ..... 4-146
4.83.1 Description ..... 4-146
4.83.2 Syntax ..... 4-146
4.83.3 Parameter ..... 4-146
4.83.4 Return Value ..... 4-146
4.83.5 Example ..... 4-146
4.83.6 Errors ..... 4-146
4.84 sc_procSchedUnlock ..... 4-147
4.84.1 Description ..... 4-147
4.84.2 Syntax ..... 4-147
4.84.3 Parameter ..... 4-147
4.84.4 Return Value ..... 4-147
4.84.5 Example ..... 4-147
4.84.6 Errors ..... 4-147
4.85 sc_procSliceGet ..... 4-148
4.85.1 Description ..... 4-148
4.85.2 Syntax ..... 4-148
4.85.3 Parameter ..... 4-148
4.85.4 Return Value ..... 4-148
4.85.5 Example ..... 4-148
4.85.6 Errors ..... 4-148
4.86 sc_procSliceSet ..... 4-149
4.86.1 Description ..... 4-149
4.86.2 Syntax ..... 4-149
4.86.3 Parameter ..... 4-149
4.86.4 ..... 4-149
4.86.5 Example ..... 4-149
4.86.6 Errors ..... 4-150
4.87 sc_procStart ..... 4-151
4.87.1 Description ..... 4-151
4.87.2 Syntax ..... 4-151

4.87.3 Parameter ..... 4-151
4.87.4 Return Value ..... 4-151
4.87.5 Example ..... 4-151
4.87.6 Errors ..... 4-152
4.88 sc_procStop ..... 4-153
4.88.1 Description ..... 4-153
4.88.2 Syntax ..... 4-153
4.88.3 Parameter ..... 4-153
4.88.4 Return Value ..... 4-153
4.88.5 Example ..... 4-153
4.88.6 Errors ..... 4-154
4.89 sc_procTimCreate ..... 4-155
4.89.1 Description ..... 4-155
4.89.2 Syntax ..... 4-155
4.89.3 Parameter ..... 4-155
4.89.4 Return Value ..... 4-156
4.89.5 Example ..... 4-156
4.89.6 Errors ..... 4-156
4.90 sc_procUnobserve ..... 4-157
4.90.1 Description ..... 4-157
4.90.2 Syntax ..... 4-157
4.90.3 Parameter ..... 4-157
4.90.4 Return Value ..... 4-157
4.90.5 Example ..... 4-157
4.90.6 Errors ..... 4-157
4.91 sc_procVarDel ..... 4-158
4.91.1 Description ..... 4-158
4.91.2 Syntax ..... 4-158
4.91.3 Parameter ..... 4-158
4.91.4 Return Value ..... 4-158
4.91.5 Example ..... 4-158
4.91.6 Errors ..... 4-158
4.92 sc_procVarGet ..... 4-159
4.92.1 Description ..... 4-159
4.92.2 Syntax ..... 4-159
4.92.3 Parameter ..... 4-159
4.92.4 Return Value ..... 4-159
4.92.5 Example ..... 4-159
4.92.6 Errors ..... 4-159
4.93 sc_procVarInit ..... 4-160
4.93.1 Description ..... 4-160
4.93.2 Syntax ..... 4-160
4.93.3 Parameter ..... 4-160
4.93.4 Return Value ..... 4-160
4.93.5 Example ..... 4-160
4.93.6 Errors ..... 4-161
4.94 sc_procVarRm ..... 4-162
4.94.1 Description ..... 4-162
4.94.2 Syntax ..... 4-162
4.94.3 Parameter ..... 4-162
4.94.4 Return Value ..... 4-162
4.94.5 Errors ..... 4-162

4.95 sc_procVarSet ..... 4-163
4.95.1 Description ..... 4-163
4.95.2 Syntax ..... 4-163
4.95.3 Parameter ..... 4-163
4.95.4 Return Value ..... 4-163
4.95.5 Example ..... 4-163
4.95.6 Errors ..... 4-163
4.96 sc_procVectorGet ..... 4-164
4.96.1 Description ..... 4-164
4.96.2 Syntax ..... 4-164
4.96.3 Parameter ..... 4-164
4.96.4 Return Value ..... 4-164
4.96.5 Errors ..... 4-164
4.97 sc_procWakeupEnable ..... 4-165
4.97.1 Description ..... 4-165
4.97.2 Syntax ..... 4-165
4.97.3 Parameter ..... 4-165
4.97.4 Return Value ..... 4-165
4.97.5 Example ..... 4-165
4.97.6 Errors ..... 4-165
4.98 sc_procWakeupDisable ..... 4-166
4.98.1 Description ..... 4-166
4.98.2 Syntax ..... 4-166
4.98.3 Parameter ..... 4-166
4.98.4 Return Value ..... 4-166
4.98.5 Example ..... 4-166
4.98.6 Errors ..... 4-166
4.99 sc_procYield ..... 4-167
4.99.1 Description ..... 4-167
4.99.2 Syntax ..... 4-167
4.99.3 Parameter ..... 4-167
4.99.4 Return Value ..... 4-167
4.99.5 Example ..... 4-167
4.99.6 Errors ..... 4-167
4.100 sc_safe_charGet ..... 4-168
4.100.1 Description ..... 4-168
4.100.2 Syntax ..... 4-168
4.100.3 Parameter ..... 4-168
4.100.4 Return Value ..... 4-168
4.100.5 Example ..... 4-168
4.100.6 Errors ..... 4-168
4.101 sc_safe_charSet ..... 4-169
4.101.1 Description ..... 4-169
4.101.2 Syntax ..... 4-169
4.101.3 Parameter ..... 4-169
4.101.4 Return Value ..... 4-169
4.101.5 Example ..... 4-169
4.101.6 Errors ..... 4-169
4.102 sc_safe_<type>Get ..... 4-170
4.102.1 Description ..... 4-170
4.102.2 Syntax ..... 4-170
4.102.3 Parameter ..... 4-170

4.102.4 Return Value ..... 4-171
4.102.5 Example ..... 4-171
4.102.6 Errors ..... 4-171
4.103 sc_safe_<type>Set ..... 4-172
4.103.1 Description ..... 4-172
4.103.2 Syntax ..... 4-172
4.103.3 Parameter ..... 4-172
4.103.4 Return Value ..... 4-173
4.103.5 Example ..... 4-173
4.103.6 Errors ..... 4-173
4.104 sc_safe_shortGet ..... 4-174
4.104.1 Description ..... 4-174
4.104.2 Syntax ..... 4-174
4.104.3 Parameter ..... 4-174
4.104.4 Return Value ..... 4-174
4.104.5 Example ..... 4-174
4.104.6 Errors ..... 4-174
4.105 sc_safe_shortSet ..... 4-175
4.105.1 Description ..... 4-175
4.105.2 Syntax ..... 4-175
4.105.3 Parameter ..... 4-175
4.105.4 Return Value ..... 4-175
4.105.5 Example ..... 4-175
4.105.6 Errors ..... 4-175
4.106 sc_sleep ..... 4-176
4.106.1 Description ..... 4-176
4.106.2 Syntax ..... 4-176
4.106.3 Parameter ..... 4-176
4.106.4 Return Value ..... 4-176
4.106.5 Example ..... 4-176
4.106.6 Errors ..... 4-177
4.107 sc_tick ..... 4-178
4.107.1 Description ..... 4-178
4.107.2 Syntax ..... 4-178
4.107.3 Parameter ..... 4-178
4.107.4 Return Value ..... 4-178
4.107.5 Example ..... 4-178
4.107.6 Errors ..... 4-178
4.108 sc_tickActivationGet ..... 4-179
4.108.1 Description ..... 4-179
4.108.2 Syntax ..... 4-179
4.108.3 Parameter ..... 4-179
4.108.4 Return Value ..... 4-179
4.108.5 Example ..... 4-179
4.108.6 Errors ..... 4-179
4.109 sc_tickGet ..... 4-180
4.109.1 Description ..... 4-180
4.109.2 Syntax ..... 4-180
4.109.3 Parameter ..... 4-180
4.109.4 Return Value ..... 4-180
4.109.5 Example ..... 4-180
4.109.6 Errors ..... 4-180

4.110 sc_tickLength ... 4-181
4.110.1 Description ... 4-181
4.110.2 Syntax ... 4-181
4.110.3 Parameter ... 4-181
4.110.4 Return Value ... 4-181
4.110.5 Example ... 4-181
4.110.6 Errors ... 4-181
4.111 sc_tickMs2Tick ... 4-182
4.111.1 Description ... 4-182
4.111.2 Syntax ... 4-182
4.111.3 Parameter ... 4-182
4.111.4 Return Value ... 4-182
4.111.5 Example ... 4-182
4.111.6 Errors ... 4-182
4.112 sc_tickTick2Ms ... 4-183
4.112.1 Description ... 4-183
4.112.2 Syntax ... 4-183
4.112.3 Parameter ... 4-183
4.112.4 Return Value ... 4-183
4.112.5 Example ... 4-183
4.112.6 Errors ... 4-183
4.113 sc_tmoAdd ... 4-184
4.113.1 Description ... 4-184
4.113.2 Syntax ... 4-184
4.113.3 Parameter ... 4-184
4.113.4 Return Value ... 4-184
4.113.5 Example ... 4-184
4.113.6 Errors ... 4-185
4.114 sc_tmoRm ... 4-186
4.114.1 Description ... 4-186
4.114.2 Syntax ... 4-186
4.114.3 Parameter ... 4-186
4.114.4 Return Value ... 4-186
4.114.5 Example ... 4-186
4.114.6 Errors ... 4-187
4.115 sc_trigger ... 4-188
4.115.1 Description ... 4-188
4.115.2 Syntax ... 4-188
4.115.3 Parameter ... 4-188
4.115.4 Return Value ... 4-188
4.115.5 Example ... 4-188
4.115.6 Errors ... 4-189
4.116 sc_triggerValueGet ... 4-190
4.116.1 Description ... 4-190
4.116.2 Syntax ... 4-190
4.116.3 Parameter ... 4-190
4.116.4 Return Value ... 4-190
4.116.5 Example ... 4-190
4.116.6 Errors ... 4-190
4.117 sc_triggerValueSet ... 4-191
4.117.1 Description ... 4-191
4.117.2 Syntax ... 4-191

4.117.3 Parameter ..... 4-191
4.117.4 Return Value ..... 4-191
4.117.5 Example ..... 4-191
4.117.6 Errors ..... 4-191
4.118 sc_triggerWait ..... 4-192
4.118.1 Description ..... 4-192
4.118.2 Syntax ..... 4-192
4.118.3 Parameter ..... 4-192
4.118.4 Return Value ..... 4-192
4.118.5 Example ..... 4-193
4.118.6 Errors ..... 4-193
4.119 sciopta_end ..... 4-194
4.119.1 Desription ..... 4-194
4.119.2 Syntax ..... 4-194
4.119.3 Parameter ..... 4-194
4.119.4 Return Value ..... 4-194
4.119.5 Errors ..... 4-194
4.120 sciopta_start ..... 4-195
4.120.1 Syntax ..... 4-195
4.121 Parameter ..... 4-195

**5. Kernel Error Reference ..... 5-1**

5.1 Introduction ..... 5-1
5.2 Include Files ..... 5-1
5.3 Function Codes (Kernels V1) ..... 5-2
5.4 Function Codes (Kernels V2 and V2INT) ..... 5-5
5.5 Error Codes ..... 5-9
5.6 Error Types ..... 5-10

**6. Manual Versions ..... 6-1**

6.1 Manual Version 5.0 ..... 6-1

**7. Index ..... 7-1**

# 1 SCIOPTA Real Time Operating System

## 1.1 Introduction

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

Available modules:

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

SCIOPTA Real-Time Operating System contains design objects such as SCIOPTA modules, processes, messages and message pools. SCIOPTA is designed on a message based architecture allowing direct message passing between processes. Messages are mainly used for interprocess communication and synchronization. SCIOPTA messages are stored and maintained in memory pools. The memory pool manager is designed for high performance and memory fragmentation is avoided. Processes can be grouped in SCIOPTA modules, which allows you to design a very modular system. Modules can be static or created and killed during run-time as a whole. SCIOPTA modules can be used to encapsulate whole system blocks (such as a communication stack) and protect them from other modules in the system.

The SCIOPTA Real-Time Kernel has a very high performance. The SCIOPTA architecture is specifically designed to provide excellent real-time performance and small size. Internal data structures, memory management, interprocess communication and time management are highly optimized. SCIOPTA Real-Time kernels will also run on small single-chip devices without MMU.

## 1.2 CPU Families

SCIOPTA is delivered for many CPU architectures such as the various Arm Ltd. families, RX (Renesas ), Power architecture (NXP, STM), Blackfin (Analog Devices) and Aurix (Infineon).

Please consult the latest version of the SCIOPTA Price List for the complete list or ask our sales team if you are missing a specific architecture.

## 1.3 SCIOPTA Kernels

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

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

## 1.4 About this Manual

The purpose of this SCIOPTA Kernel Manual is to give all needed information how to use SCIOPTA Real-Time Kernel in an embedded project.

Chapter 2 "SCIOPTA Architecture" on page 2-1 is a detailed description and introduction into SCIOPTA working concepts, structures and elements.

Chapter 3 "System Calls Overview" on page 3-1 is the list of all system calls organized in functional groups.

Chapter 4 "System Calls Reference" on page 4-1 contains the reference of all system calls in alphabetical order.

Chapter 5 "Kernel Error Reference" on page 5-1 gives information about the SCIOPTA error handling and contains the list of all SCIOPTA error messages.

In Chapter 6 "Manual Versions" on page 6-1 you will find the version history of the manual.

## 1.5 SCIOPTA System Manuals

Detailed information on using SCIOPTA for specific CPU Families can be found in the SCIOPTA Systems Manuals. The following manuals are available:

- SCIOPTA System Manual for ARM
- SCIOPTA System Manual for ARM64
- SCIOPTA System Manual for PowerPC
- SCIOPTA System Manual for Aurix
- SCIOPTA System Manual for RX
- SCIOPTA System Manual for Blackfin

# 2 SCIOPTA Architecture

## 2.1 Introduction

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

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

In SCIOPTA processes communicate and cooperate by exchanging messages. Messages can have a content to move data from one process to the other or can be empty just to coordinate processes. Often, process switches can occur as a result of a message transfer. Besides data and some control structures, messages contain also an identity (number). This is used by processes to recognize the different message and also allows to select which messages to receive from the message queue FIFO. All other messages are kept back in the message queue of the process.

Messages are dynamically allocated from a message pool.

There are three Kernels within SCIOPTA, please see chapter 1.3 “SCIOPTA Kernels” on page 1-2 for more infprmation.

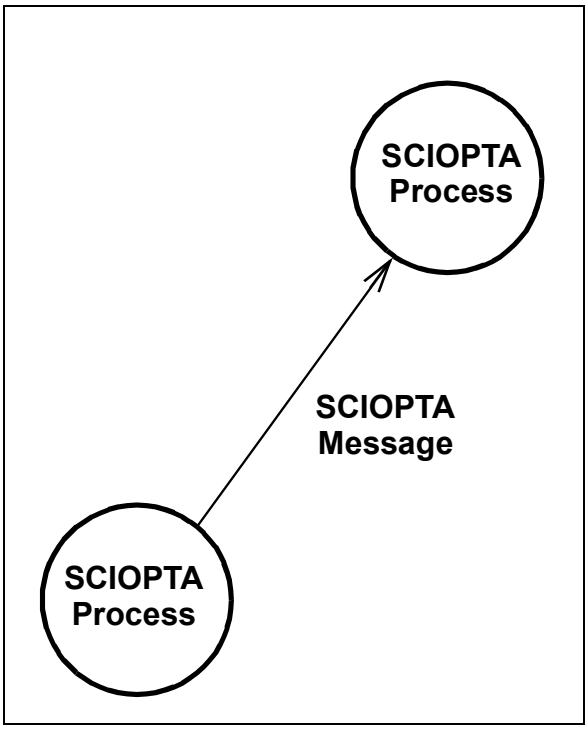


**Figure 2-1: SCIOPTA - Message Based RTOS**

### 2.1.1 SCIOPTA Kernel V2

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

Differences between SCIOPTA Kernel V2 and V1:

- Module priority: No process inside a module is allowed to have a higher priority than modules's maximum priority.
- Module friendship concept has been removed.
- System calls **sc_procPathGet** and **sc_procNameGet** return NULL if PID valid, but does not exist anymore.
- Time slice for prioritized processes is now fully supported and documented.
- The system call **sc_procSliceSet** is only allowed within same module.
- Parameter **n** in system call **sc_procVarInit** is now real maximum number of process variables. It was **n+1** before.
- Calls **sc_msgTx** and **sc_msgTxAlias** sets the activation time.
- The activation time is saved for **sc_msgRx** in prioritized processes.
- System call **sc_sleep** returns now the activation time.
- The error handling has been modified.
- Error hook API changed.
- Error process and error proxy introduced.
- New system calls:
    - **sc_moduleCreate2** call replaces the call sc_moduleCreate. It gets the module parameters now from a module descriptor block (mdb).
    - **sc_modulePrioGet**. Returns the priority of a module.
    - **sc_moduleStop**. Stops a whole module.
    - **sc_procAtExit**. Register a function to be called if a prioritized process is killed.
    - **sc_procAttrGet**. Returns specific process attributes.
    - **sc_procCreate2** call replaces the calls sc_procPrioCreate, procIntCreate and procTimCreate. It gets the process parameters now from a process descriptor block (pdb).
    - **sc_msgHdChk**. Integrity check of message header.
    - **sc_msgFind**. Finds messages which have been allocated or already received.
    - **sc_tickActivationGet**. Returns the tick time of last activation of the calling process.
    - **sc_miscCrc32**. Calculates a 32 bit CRC over a specified memory range.
    - **sc_miscCrc32Contd**. Calculates a 32 bit CRC over an additional memory range.

## 2.2 Modules

### 2.2.1 Introduction

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

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

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

### 2.2.2 System Module

There is always one static module in a SCIOPTA system.

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

### 2.2.3 Module Priority

SCIOPTA modules contain a (module) priority.

**Kernel V1:**

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

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

The effective priority has an upper limit of 31which 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).

**Kernels V2 and V2INT:**

This module priority defines a maximum priority level for all processes contained inside that module. The kernel will generate an error, if a process is created which has a higher priority than the module priority.

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

### 2.2.4 System Protection

In larger systems it is often necessary to protect certain areas to be accessed by others. In SCIOPTA the user can achieve this by grouping processes into modules creating sub-systems which can be protected.

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

System protection and MMU support is optional in SCIOPTA.

### 2.2.5 SCIOPTA Module Friend Concept

**Kernel V1:**

SCIOPTA supports also the “friend” concept. Modules can be “friends” of other modules. This has mainly consequences on whether message will be copied or not at message passing.

A module can be declared as friend by the 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.

**Kernels V2 and V2INT:**

Not supported!

### 2.2.6 Module Creation

#### 2.2.6.1 Static Module Creation

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

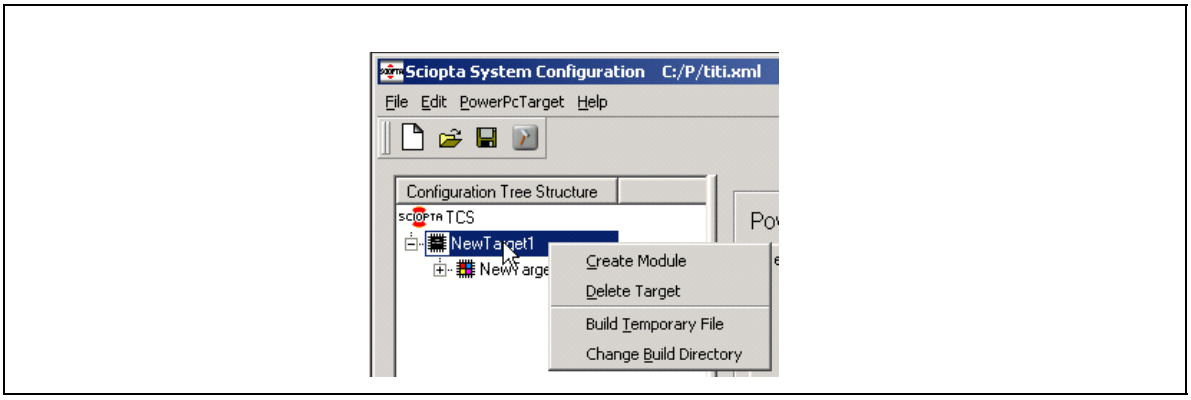


**Figure 2-2: Module Creation by SCONF**

#### 2.2.6.2 Dynamic Module Creation

Modules can also be created dynamically by the "sc_moduleCreate" (**Kernel V1**) or the "sc_moduleCreate2" (**Kernels V2 and V2INT**) system calls.

```
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
);
```

**Figure 2-3: Dynamic Module Creation Kernel V1**

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

**Figure 2-4: Dynamic Module Creation Kernels V2 and V2INT**

#### 2.2.6.3 Module System Calls

Please consult chapter 3.4 “Module System Calls” on page 3-6 for the list of Module System Calls.

## 2.3 Processes

### 2.3.1 Introduction

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

All SCIOPTA processes have system wide unique process identities.

A SCIOPTA process is always part of a SCIOPTA module. Please consult chapter 2.2 “Modules” on page 2-3 for more information about SCIOPTA modules.

### 2.3.2 Process States

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

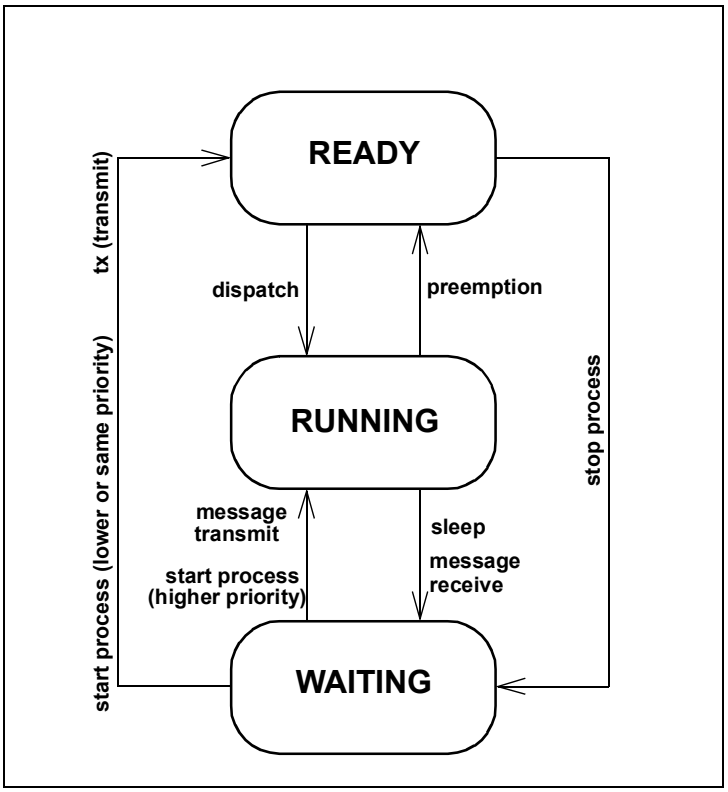


**Figure 2-1: State Diagram of SCIOPTA Kernel**

#### 2.3.2.1 Running

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

#### 2.3.2.2 Ready

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

#### 2.3.2.3 Waiting

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

- The process tried to receive a message which has (not yet) arrived.
- The process waits for a delay to expire..
- The process waits on a SCIOPTA trigger.
- The Process waits to be started.

### 2.3.3 Static Processes

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

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

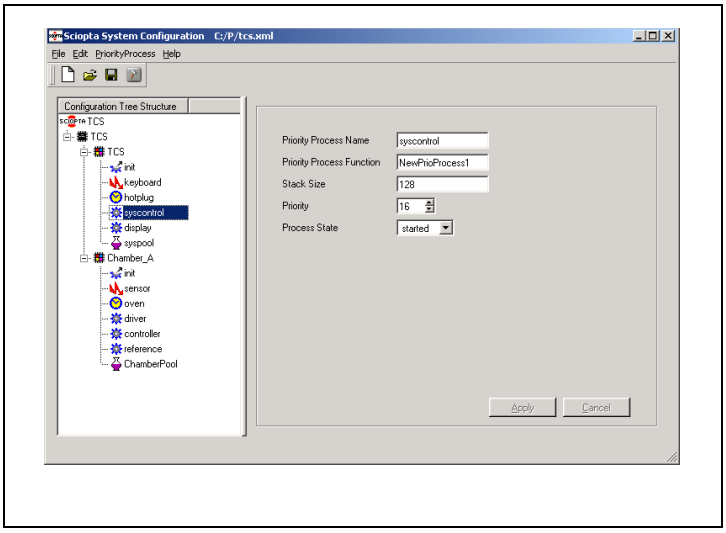


**Figure 2-2: Process Configuration Window for Static Processes**

### 2.3.4 Dynamic Processes

Dynamic processes can be created and killed during run-time. Often dynamic processes are used to run multiple instances of common code. The number of instances is only limited by system resources.

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

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

**Figure 2-3: Create Process System Call (prioritzed process for Kernel V1)**

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

**Figure 2-3: Create Process System Call (Kernel V2)**

### 2.3.5 Process Identity

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

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

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

### 2.3.6 Prioritized Processes

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

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

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

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

#### 2.3.6.1 Creating and Declaring Prioritized Processes

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

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

#### 2.3.6.2 Process Priorities

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

By assigning a priority the user designs groups of processes or parts of systems according to response time requirements. Ready processes with high priority are always interrupting processes with lower priority. Subsystems with high priority processes have therefore faster response time. Priority values for prioritized processes in SCIOPTA can be from 0 to 31. 0 is the highest and 31 the lowest priority level.

**Kernel V1:**

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

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

**Kernels V2 and V2INT:**

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

See also chapter 2.2.3 "Module Priority" on page 2-3.

### 2.3.6.3 Writing Prioritized Processes

**Process Declaration Syntax**

All prioritized processes in SCIOPTA must contain the following declaration:

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

**Process Template**

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

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

   /* Process initialization code */

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

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


      /* Code for process proc_name */


   }
}
```

### 2.3.7 Interrupt Processes

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

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

Interrupt process is the fastest process type in SCIOPTA and will respond almost immediately to events. As the system is blocked during interrupt handling interrupt processes must perform their task in the shortest time possible. By default, SCIOPTA does not allow interrupt nesting. If the user application needs it, the interrupts must be released inside the interrupt process. Consult the SCIOPTA Systems Manuals on how this can be done.

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

#### 2.3.7.1 Creating and Declaring Interrupt Processes

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

**Kernel V1:**

Dynamic interrupt process are created by using the "sc_procIntCreate" system call and killed dynamically with the "sc_procKill" system call.

**Kernels V2 and V2INT:**

Dynamic interrupt process are created by using the "sc_procCreate2" system call and killed dynamically with the "sc_procKill" system call.

#### 2.3.7.2 Interrupt Process Priorities

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

#### 2.3.7.3 Writing Interrupt Processes

##### Interrupt Process Declaration Syntax

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

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

##### Interrupt Source Parameter irq_src

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

##### Interrupt Source Parameter Values

| | |
|---|---|
| **SC_PROC_WAKEUP_HARDWARE** | The interrupt process is activated by a real hardware interrupt. |
| **SC_PROC_WAKEUP_MESSAGE** | The interrupt process is activated by a message sent to the interrupt process. |
| **SC_PROC_WAKEUP_TRIGGER** | The interrupt process is activated by a trigger event. |
| **SC_PROC_WAKEUP_START** | The interrupt process is activated when the process is started. (**Only for Kernels V2 and V2INT**) |
| **SC_PROC_WAKEUP_CREATE** | The interrupt process is activated when the process is created. This allows the interrupt process to execute some initialization code. |
| **SC_PROC_WAKEUP_KILL** | The interrupt process is activated when the process is killed. This allows the interrupt process to execute some exit code. |
| **SC_PROC_WAKEUP_STOP** | The interrupt process is activated when the process is stopped. (**Only for Kernels V2 and V2INT**). |
| **SC_PROC_WAKEUP_CALLBACK** | The callback is called. (**Only SCIOPTA Simulator**) |

##### Interrupt Vector Parameter vector

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

See "sc_procIrqRegister" for more information.

### Interrupt Process Template for Kernel V1

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

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

  switch (irq_src) {

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

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

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

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

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

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

### Interrupt Process Template for Kernels V2 and V2INT

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

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

  switch (irq_src) {

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

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

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

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

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

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

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

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

### 2.3.8 Timer Processes

A timer process in SCIOPTA is a specific interrupt process connected to the tick timer of the operating system. SCIOPTA is calling each timer process periodically.

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

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

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

### 2.3.9 Creating and Declaring Timer Processes

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

**Kernel V1:**

Dynamic interrupt process are created by using the "sc_procTimCreate" system call and killed dynamically with the "sc_procKill" system call.

**Kernel V2 and V2INT:**

Dynamic interrupt process are created by using the "sc_procCreate2" system call and killed dynamically with the "sc_procKill" system call.

### 2.3.10 Timer Process Priorities

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

### 2.3.11 Writing Timer Processes

Timer processes are written exactly the same way as interrupt processes. Please consult chapter 2.3.7.3 "Writing Interrupt Processes" on page 2-13 for information how to write interrupt processes.

Unlike interrupts timer processes can be stopped and started also in **Kernel V1**.

### 2.3.12 Init Processes

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

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

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

#### 2.3.12.1 Creating and Declaring Init Processes

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

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

#### 2.3.12.2 Init Process Priorities

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

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

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

#### 2.3.12.3 Writing Init Processes

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

Example: Module “dev”, module hook: void dev()

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

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

Template of a minimal init process of a dynamic module:

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

### 2.3.13 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.

#### 2.3.13.1 Process Daemons

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

Whenever you are using the "sc_procIdGet" system call you need to start the process daemon.

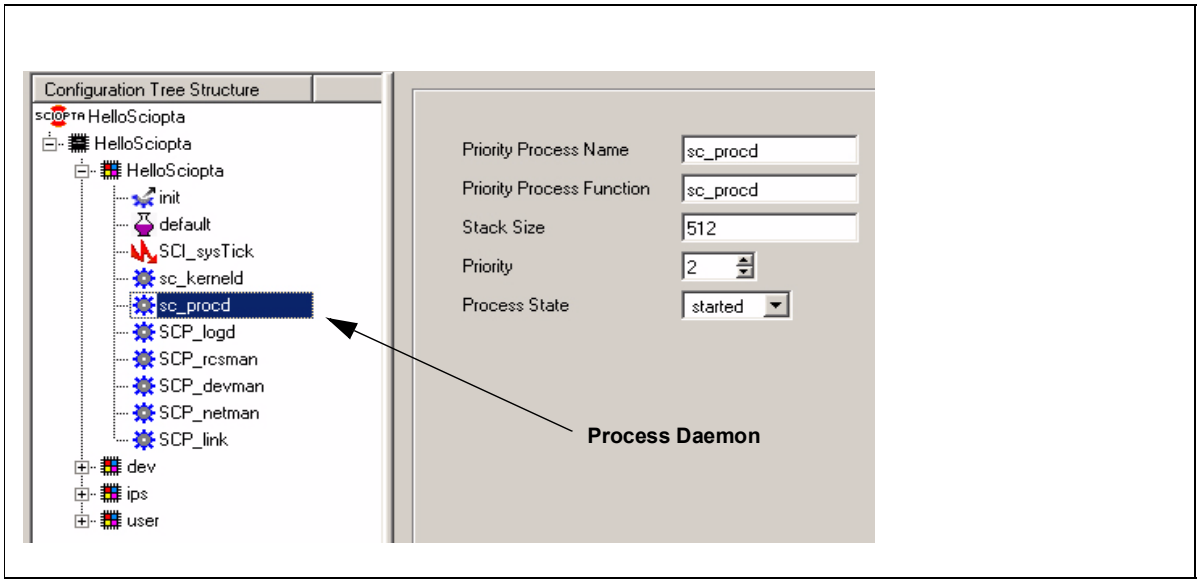


**Figure 2-4: Process Daemon Declaration in SCONF**

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

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

### 2.3.13.2 Kernel Daemon

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

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

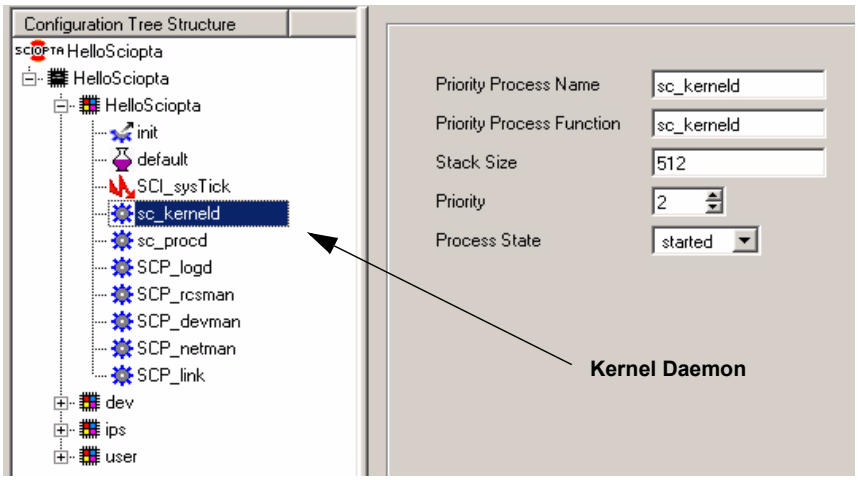


**Figure 2-5: Kernel Daemon Declaration in SCONF**

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

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

### 2.3.14 Process Stacks

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

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

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

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

#### 2.3.14.1 Unified Interrupt Stack for ARM Architecture

**Only for Kernel V1.**

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

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

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

#### 2.3.14.2 Interrupt Nesting for ARM Architecture

**Only for Kernel V1.**

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

### 2.3.15 Stack Protector

**Only for Kernel V2 and V2INT on PowerPC**

In a SCIOPTA system you can enable a stack protection called stack protector.

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

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

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

### 2.3.16 Addressing Processes

#### 2.3.16.1 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.

#### 2.3.16.2 Get Process IDs of Static Processes

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

You can address static process by appending

```
_pid
```

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

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

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

#### 2.3.16.3 Get Process IDs of Dynamic Processes

Dynamic processes can be created and killed during run-time. Often dynamic processes are used to run multiple instances of common code.

The process IDs of dynamic processes can be retrieved by using the system call "sc_procIdGet".

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

### 2.3.17 Process Variables

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

The process variable are usually maintained inside a SCIOPTA message and managed by the kernel. The user can access the process variable by specific system calls.

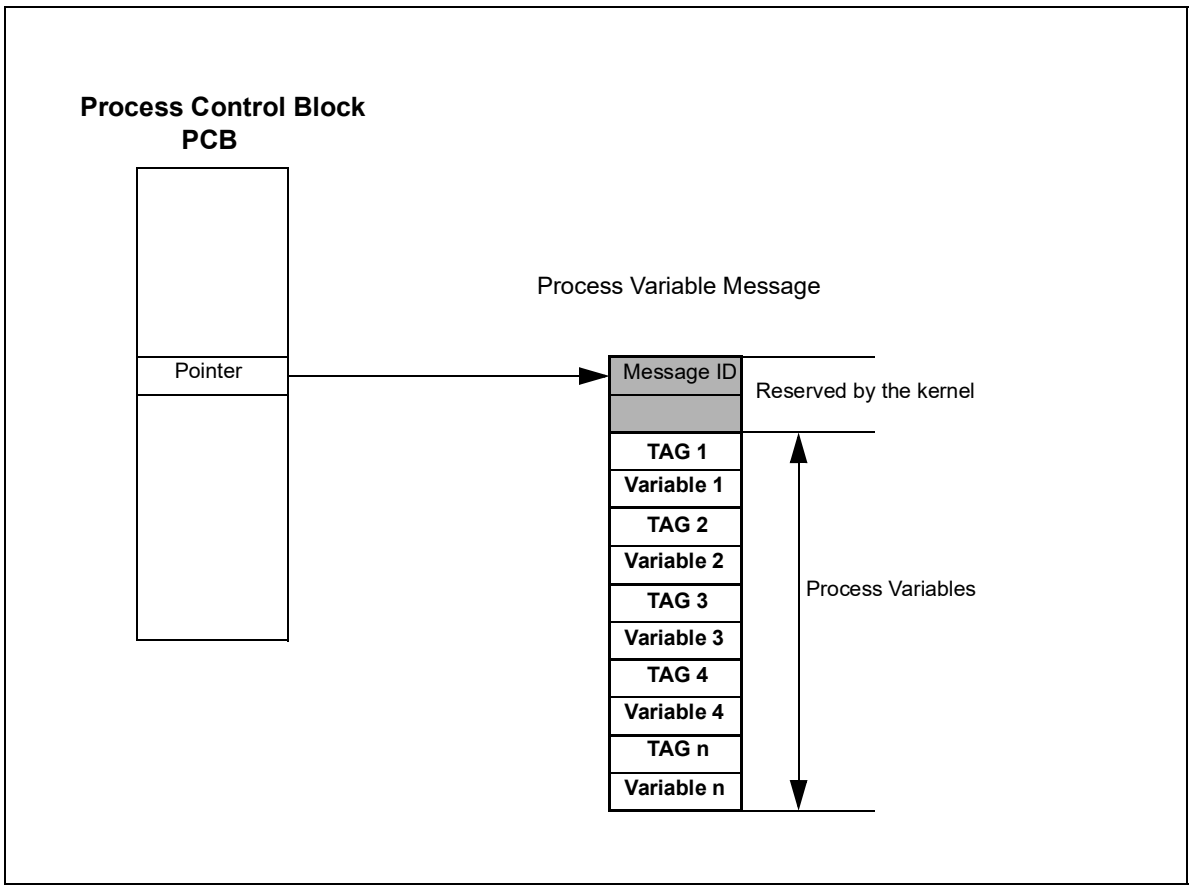


**Figure 2-6: SCIOPTA Process Variables**

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

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

### 2.3.18 Process Observation

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

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

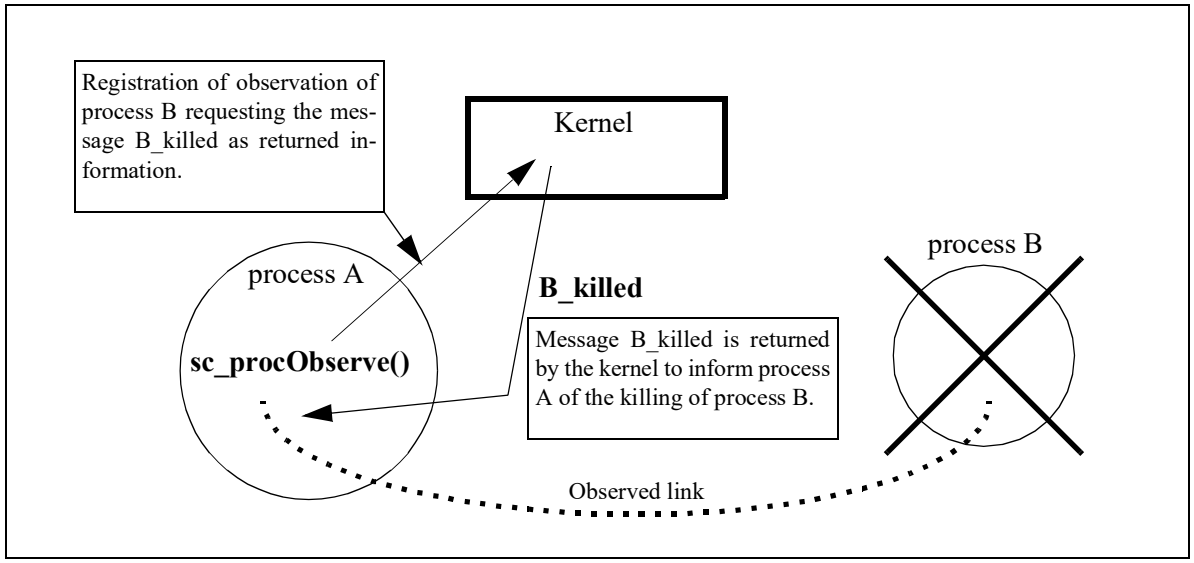


**Figure 2-7: SCIOPTA Observation**

## 2.4 Messages

### 2.4.1 Introduction

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

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

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

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

### 2.4.2 Message Structure

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

- Process ID of message owner
- Message size
- Process ID of transmitting process
- Process ID of addressed process

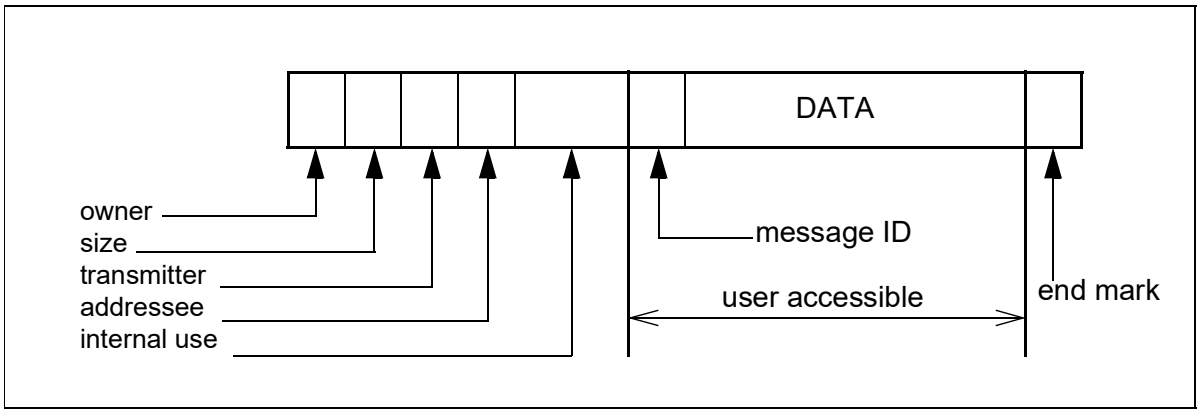


**Figure 2-5: SCIOPTA Message Structure**

When a process allocates a message it becomes the owner of the message. If the process transmits the message to another process, then this one becomes the owner. After transmitting, the sending process cannot access the message any more. This message ownership feature eliminates access conflicts in a clean and efficient way.

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

### 2.4.3 Message Size

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

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

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

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

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

#### 2.4.3.1 Example

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

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

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

### 2.4.4 Message Pool

Messages are the main data object in SCIOPTA. Messages are allocated by processes from message pools. If a process does not need the messages any longer it will free the message and return it to the pool. After this the process may no longer access the message.

Please consult chapter 2.5 “Pools” on page 2-36 for more information about message pools.

### 2.4.5 Message Passing

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

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

The "sc_msgAlloc" or the "sc_msgRx" calls returns a pointer to the allocated message. The pointer allows the user to access the message data to initialize or modify it.

The sending process transmits the message by calling the "sc_msgTx" system call. SCIOPTA changes the owner of the message to the kernel and puts the message in the queue of the receiver process. It is a linked list of all messages in the pool transmitted to this process.

If the receiving process is blocked at the "sc_msgRx" system call and is waiting on the transmitted message the kernel is performing a process swap and activates the receiving process. After reception, the process becomes owner of the process and gets the pointer to access the message contents. The "sc_msgRx" 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 shall be returned to the system by "sc_msgFree".

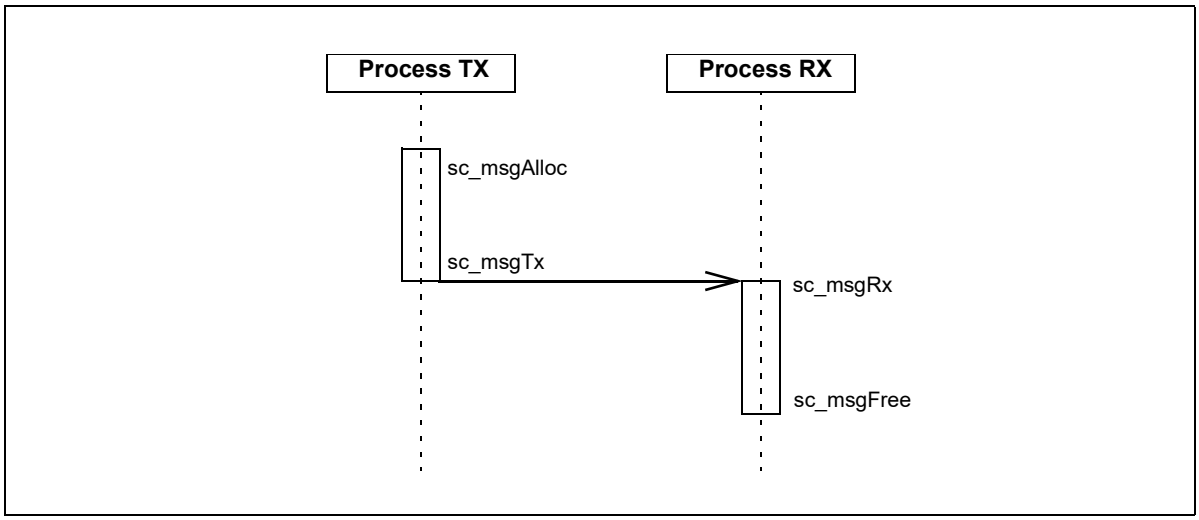


**Figure 2-6: Message Sequence Chart of a SCIOPTA Message Passing**

### 2.4.6 Message Declaration

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

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

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

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

#### 2.4.6.1 Message Identifier

**Description**

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

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

**Syntax**

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

**Parameter**

| | |
|---|---|
| **msg_nr** | Message Identifier (ID) |

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

#### Syntax

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

#### Parameter

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

### 2.4.6.3 Message Union

#### Description

All processes which are using SCIOPTA messages should include the following message union declaration.

The union sc_msg is used to standardize a message declaration for files using SCIOPTA messages.

#### Syntax

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

#### Parameter

| | |
|---|---|
| **id** | Message 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. |

### 2.4.7 Message Number (ID) Organization

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

#### 2.4.7.1 Global Message Number Defines File

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.

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

### 2.4.8 Example

```
#define CHAR_MSG(5)

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

union sc_msg
{
  sc_msgid_t id;
  char_msg_t char_msg;
};

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

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

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

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

### 2.4.9 Messages and Modules

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

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

Messages which are transmitted across modules boundaries are always copied if the Inter-Module setting in the system configuration utility is set to “always copy”. If it set to never copy, messages between modules are not copied.

**Kernel V1:**

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

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

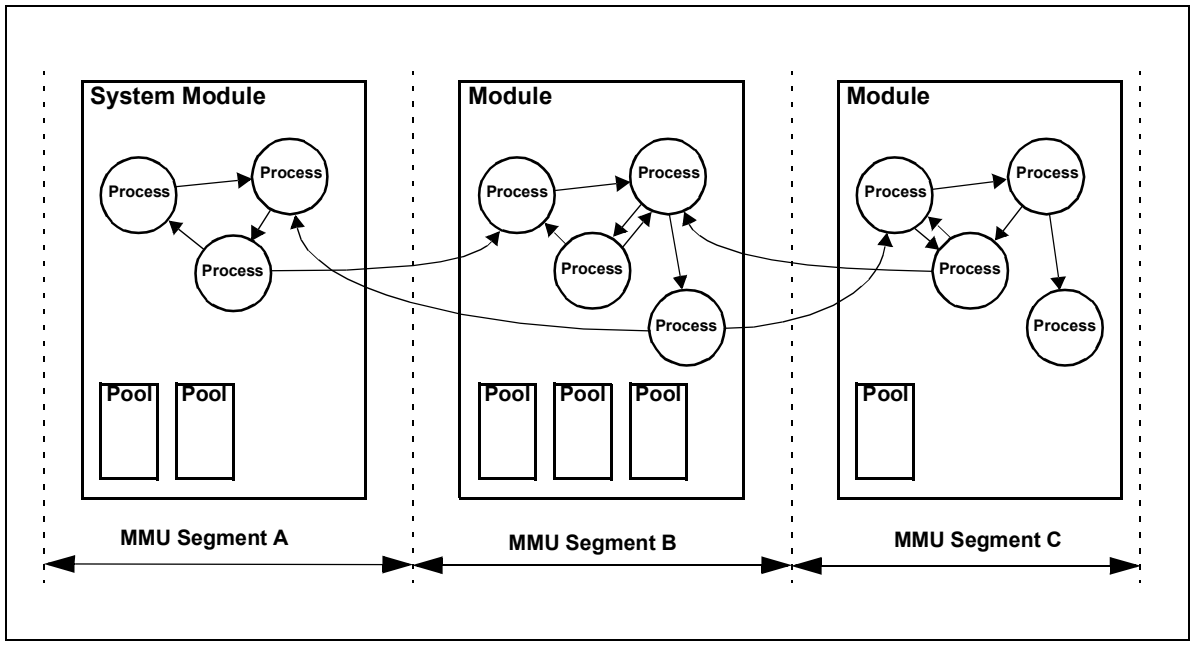


**Figure 2-7: SCIOPTA Messages and Modules**

### 2.4.10 Returning Sent Messages

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

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

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

### 2.4.11 Message Passing and Scheduling

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

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

Interrupt and timer process will always pre-empt prioritized processes. The intercepted prioritized process will continue after the interrupt has finished unless a process with a higher priority became ready (e.g. by a message fromt the interrupt process). In this case the CPU is given to the now highest priority process. The pre-emted process will continue where interrupted when it became highest priority process again..

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

The SCIOPTA kernel will do a re-scheduling at every, receive call, transmit call, process yield call, trigger wait call, sleep call and all system time-out which have elapsed. There is no forced re-scheduling based on a tick unless requested by the process by setting a time-slice.

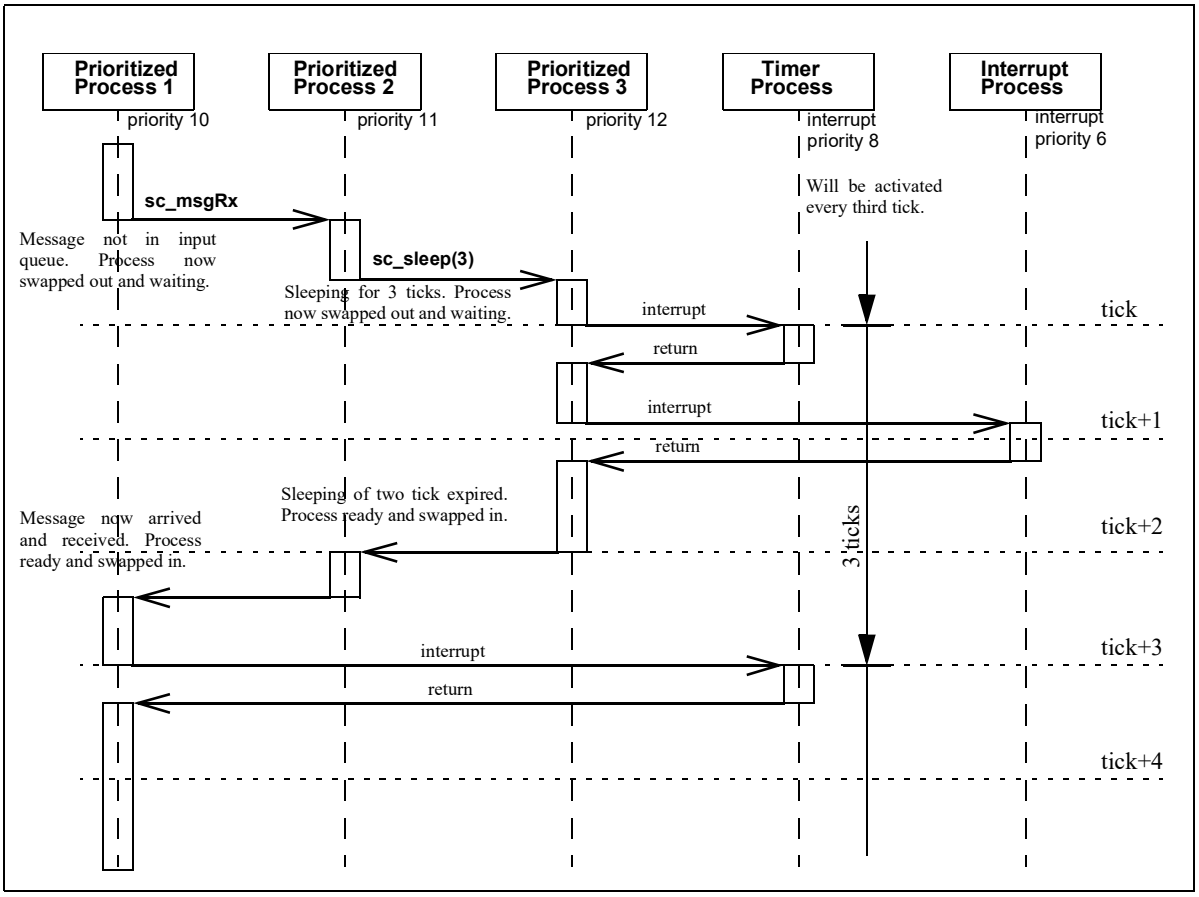


**Figure 2-8: Scheduling Sequence Example**

### 2.4.12 Message Sent to Unknown Process

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

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

If the message is sent with the flag **SC_MSGTX_RTN2SNDR**, then the message will stay with the sender.

#### 2.4.12.1 Example

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

    /* Error handling */

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


    }
}
```

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

## 2.5 Pools

Messages are the main data object in SCIOPTA. Messages are allocated by processes from message pools. If a process does not need the messages any longer it will free it.

There can be up to 127 pools per module for a standard kernel. Please consult chapter 2.2 “Modules” on page 2-3 for more information about the SCIOPTA module concept. The maximum number of pools will be defined at module creation. A message pool always belongs to the module where it was created.

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

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

The SCIOPTA kernel is managing all existing pools in a system. Messages are maintained by double linked list in the pool.

### 2.5.1 Message Pool Size

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

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

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

where:

| | |
|---|---|
| **n** | Number of buffer sizes (4, 8 or 16) |
| **stat** | Process statistics or message statistics are used (1) or not used (0) |

### 2.5.2 Creating Pools

#### 2.5.2.1 Static Pool Creation

Static pools are pools which are automatically created when the systems boots up. They are defined in the SCONF configuration tool.

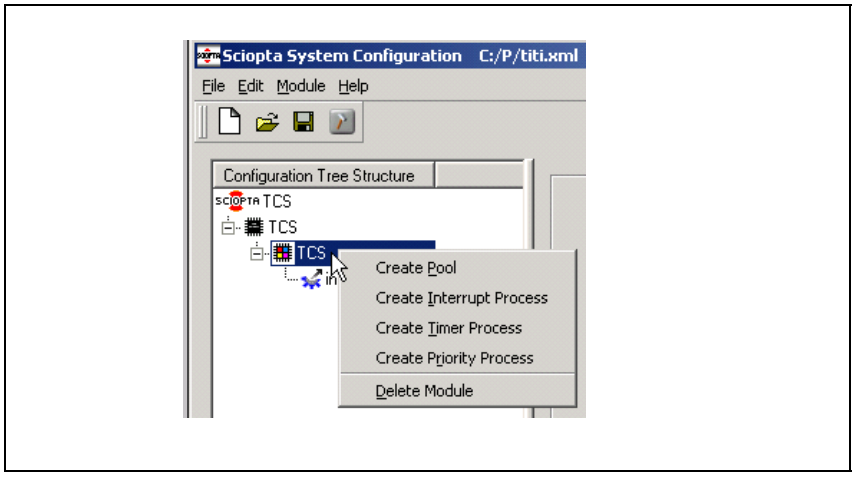


**Figure 2-2: Pool Creation by SCONF**

#### 2.5.2.2 Dynamic Pool Creation

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

```
static const sc_bufsize_t bufsizes[8]=
{
    4,
    8,
    16,
    32,
    64,
    128,
    256,
    700
};

myPool_plid = sc_poolCreate(
    /* start-address            */ 0,
    /* total size               */ 4000,
    /* number of buffers        */ 8,
    /* buffersizes              */ bufsizes,
    /* name                     */ "myPool"
);
```

**Figure 2-3: Dynamic Pool Creation**

## 2.6 Hooks

### 2.6.1 Introduction

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

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

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

### 2.6.2 Error Hook

The **Error Hook** is the most important user hook function and should normally be included in most of the systems. An error hook can be used to log the error and additional data on a logging device if the kernel has detected an error condition.

The error hook is described in chapter 2.10 “Error Handling” on page 2-48.

### 2.6.3 Message Hooks

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

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

### 2.6.4 Process Hooks

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

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

**Kernel V2**: If the IRQ Swap hook is activated, then the swap hook is also called before an interrupt process is entered.

Process hooks must be registered by using the “sc_procHookRegister” system call.

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

### 2.6.5 Pool Hooks

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

## 2.7 System Start and Setup

### 2.7.1 Start Sequence

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

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

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

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

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

### 2.7.2 Reset Hook

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

The reset hook must be written by the user.

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

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

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

#### 2.7.2.1 Syntax

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

#### 2.7.2.2 Parameter

None.

#### 2.7.2.3 Return Value

If it is set to !=0 then the kernel will immediately call the dispatcher. This will initiate a warm start.

If it is set to 0 then the kernel will jump to the C startup function. This will initiate a cold start.

#### 2.7.2.4 Location

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

resethook.S — Very early hardware initialization code written in assembler.
The extension .S is used in GCC for assembler source files. For other compiler packages the extensions for assembler source files might be different.

File location: <installation_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\src

### 2.7.3 C Startup

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

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

### 2.7.4 Starting the SCIOPTA Simulator

**Only for the SCIOPTA SCSIM Simulator:**

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

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

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

#### 2.7.4.1 Module Data RAM

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

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

### 2.7.5 Start Hook

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

The start hook is mainly used to do chip, board and system initialization. As the C environment is initialized it can be written in C. The start hook would also be the right place to include the registration of the system error hook (see chapter 2.10.7 “Error Hook Registering” on page 2-51) and other kernel hooks.

#### 2.7.5.1 Syntax

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

#### 2.7.5.2 Parameter

None.

#### 2.7.5.3 Return Value

None.

#### 2.7.5.4 Location

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

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

### 2.7.6 Init Processes

The init process is the first process in a module. Each module has at least one process and this is the init process. At module start the init process gets automatically the highest priority (0). After the init process has done some 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.

### 2.7.7 Module Start Functions

Please consult chapter 2.2 “Modules” on page 2-3 for general information about SCIOPTA modules.

#### 2.7.7.1 System Module Start Function

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

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

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

#### 2.7.7.2 User Module Start Function

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

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

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

## 2.8 SCIOPTA Trigger

### 2.8.1 Description

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

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

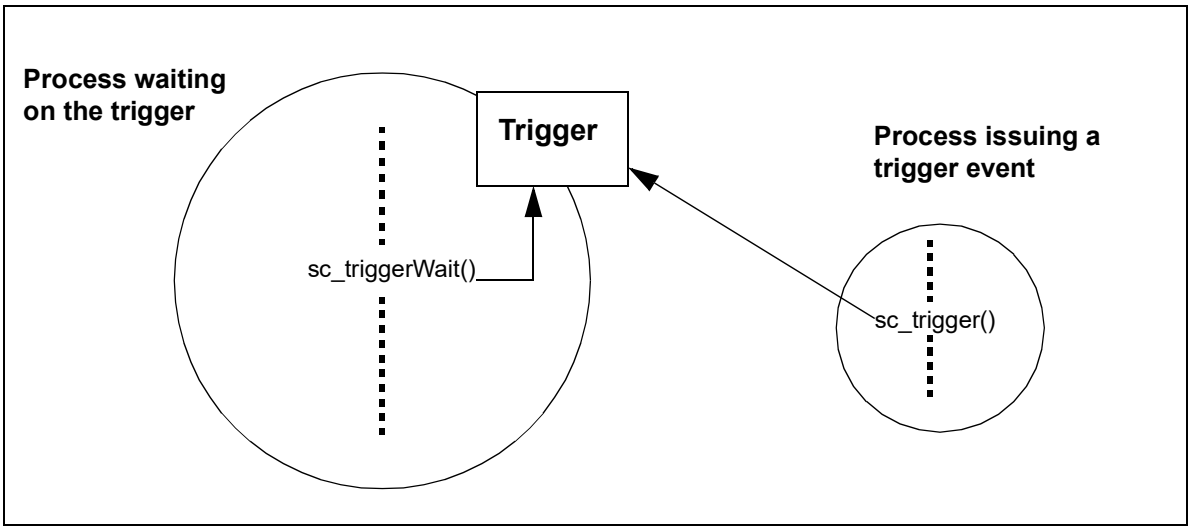


**Figure 2-4: SCIOPTA Trigger**

### 2.8.2 Using SCIOPTA Trigger

There are four system calls available to work with triggers. The "sc_triggerWait" call decrements the value of the trigger and the calling process will be blocked and swapped out if the value gets negative or equal to zero. Only the owner process of the trigger can wait for it. The process gets ready again when either the optional timeout expires or the trigger variable become greater than zero by other processes calling "sc_trigger". In case of an timeout, the trigger is incremented by the same amount as it has been decremented before. Note: "sc_trigger" calls which do not increment the variable to a value greater than zero will not be lost.

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

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

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

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

### 2.8.3 Trigger Example

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

extern sc_pid_t trigproc_pid

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

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

SC_PROCESS (trigproc)
{

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

  for (;;)
  {

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

## 2.9 Time Management

### 2.9.1 Introduction

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

### 2.9.2 System Tick

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

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

### 2.9.3 Configuring the System Tick

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

### 2.9.4 External Tick Interrupt Process

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

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

### 2.9.5 Timeout Server

#### 2.9.5.1 Introduction

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

#### 2.9.5.2 Using the Timeout Server

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

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

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

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

## 2.10 Error Handling

### 2.10.1 Introduction

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 identifiers 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 Hook. In traditional real-time operating systems, the user needs to check return values of system calls for a possible error condition. In SCIOPTA all error conditions will end up in the Error Hook. This guarantees that all errors are treated and that the error handling does not depend on individual error strategies which might vary from user to user.

### 2.10.2 Error Sequence Kernel V1

In SCIOPTA all error conditions will end up in an Error Hook. This guarantees that all errors are treated and that the error handling does not depend on individual error strategies which might vary from user to user.

There are two error hooks available:

A) Module Error Hook

B) Global Error Hook

If the kernel detect an error condition it will first call the module error hook and if it is not available call the global error hook. Error hooks are normal error handling functions and must be written by the user. Depending on the type of error (fatal or nonfatal) it will not be possible to return from an error hook. If there are no error hooks present the kernel will enter an infinite loop (at label **SC_ERROR**) and all interrupts are disabled.

Note: The use of module error hooks is deprecated

### 2.10.3 Error Sequence Kernel V2 and V2INT

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

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

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

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

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

### 2.10.4 Error Hook Kernel V1

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

An error hook can only use the following system calls:

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

### 2.10.5 Error Hook Kernel V2 and V2INT

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

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

The error hook can only use the following system calls:

| | |
|---|---|
| sc_miscCrc | Calculates a 16 bit CRC over a specified memory range. |
| sc_miscCrc32 | Calculates a 32 bit CRC over a specified memory range. |
| sc_miscCrcContd | Calculates a 16 bit CRC over an additional memory range. |
| sc_miscCrcContd32 | Calculates a 32 bit CRC over an additional memory range. |
| sc_miscErrnoGet | Returns the process error number (errno) variable. |
| sc_moduleIdGet | Returns the ID of a module. |
| sc_moduleInfo | Returns a snap-shot of a module control block (mcb). |
| sc_moduleNameGet | Returns the name of a module. |
| sc_poolIdGet | Returns the ID of a message pool. |
| sc_poolInfo | Returns a snap-shot of a pool control block. |
| sc_procAttrGet | Returns process attributes. |
| sc_procPpidGet | Returns the process ID of the parent (creator) of a process. |
| sc_procPrioGet | Returns the priority of a prioritized process. |
| sc_procSliceGet | Returns the time slice of a timer process. |
| sc_procVarDel | Removes a process variable from the process variable data area. |
| sc_procVarGet | Returns a process variable. |
| sc_procVarSet | Defines or modifies a process variable. |
| sc_tickGet | Returns the actual kernel tick counter value. |
| sc_tickLength | Sets the current system tick length in micro seconds. |
| sc_tickMs2Tick | Converts a time from milliseconds into system ticks. |
| sc_tickTick2Ms | Converts a time from system ticks into milliseconds. |
| sc_triggerValueGet | Returns the value of a process trigger. |

### 2.10.6 Error Information

When the error hook is called from the kernel, all information about the error are transferred in 32-bit error word (parameter errcode). Please consult chapter 5 “Kernel Error Reference” on page 5-1 for detailed description of the SCIOPTA error word. There are also up to four additional 32-bit extra words (parameters extra1 ... extra3) available to the user.

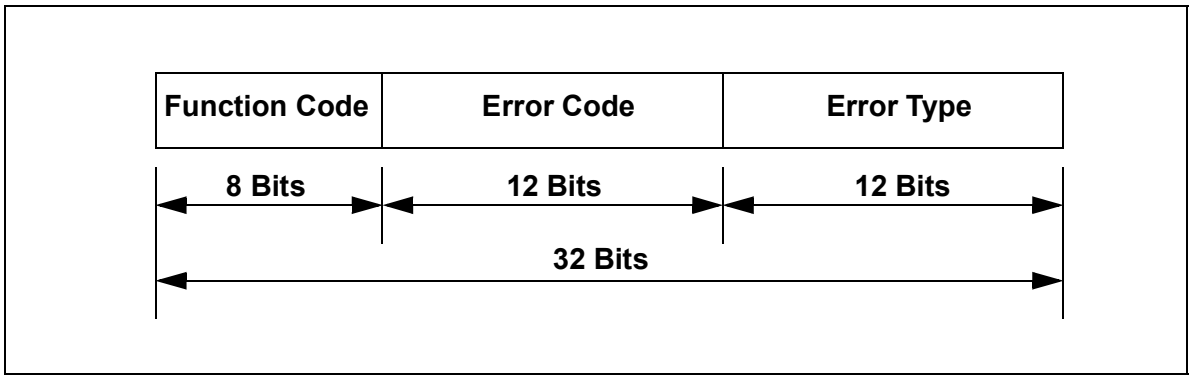


**Figure 2-5: 32-bit Error Word (Parameter: errcode)**

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

The **Error Code** contains the specific error information.

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

There are three error types in a SCIOPTA kernel.

- SC_ERR_SYSTEM_FATAL, system wide fatal error.
- SC_ERR_MODULE_FATAL, module wide fatal error.
- SC_ERR_PROCESS_FATAL, process wide fatal error.

There are three error warnings in a SCIOPTA kernel.

- SC_ERR_SYSTEM_WARNING, system wide warning.
- SC_ERR_MODULE_WARNING, module wide warning.
- SC_ERR_PROCESS_WARNING, process wide warning.

### 2.10.7 Error Hook Registering

An error hook is registered by using the “sc_miscErrorHookRegister” system call.

**Kernel V1:**

If the error hook is registered from within the system module it is registered as a global error hook. In this case the error hook registering will be done in the start hook.If the error hook is registered from within a module which is not the system module it will be registered as a module error hook.

Note: Module error hook is deprecated.

**Kernel V2 and V2INT:**

The error hook can only be registered in the start hook.

### 2.10.8 Error Hook Declaration Syntax Kernel V1

#### 2.10.8.1 Description

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

#### 2.10.8.2 Syntax

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

   ... error hook code

};
```

#### 2.10.8.3 Parameter

| | |
|---|---|
| **errcode** | Error word |
| | Error word containing the function code which defines from which SCIOPTA system call the error was initiated, the error code which contains the specific error information and the error type which informs about the source and type of error. |
| **extra** | Error extra word |
| | Gives additional information depending on the error code. |
| **user** | User/system error flag |
| != 0 | User error. |
| == 0 | System error. |
| **pcb** | Process control block |
| | Pointer to process control block of the process where the error occurred. Please consult **pcb.h** for more information about the module control block structure. |

#### 2.10.8.4 Error Hook Example

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

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

### 2.10.9 Error Hook Declaration Syntax Kernel V2 and V2INT

#### 2.10.9.1 Description

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

#### 2.10.9.2 Syntax

```
void <err_hook_name> (sc_errcode_t err, const sc_errMsg_t *errMsg)
{

   ... error hook code

};
```

#### 2.10.9.3 Parameter

| | |
|---|---|
| **err** | Error word |
| | Error word containing the function code which defines from which SCIOPTA system call the error was initiated, the error code which contains the specific error information and the error type which informs about the source and type of error. |
| **errMsg** | Pointer to the error message pointer |
| | Message is filled by the kernel error handling. |

#### 2.10.9.4 Kernel Error Message Structure

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

#### 2.10.9.5 Structure Members

| **user** | User/system error flag |
|---|---|
| != 0 | User error. |
| == 0 | System error. |

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

| **extra** | System specific extra error words |
|---|---|
| extra0<br>extra1<br>extra2<br>extra3 | Please consult chapter 5 “Kernel Error Reference” on page 5-1 for detailed description of the SCIOPTA extra error words. |

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

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

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

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

#### 2.10.9.6 Header Files

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

#### 2.10.9.7 Error Hook Example

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

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

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

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

### 2.10.10 Error Hooks Return Behaviour Kernel V1

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

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

### 2.10.11 Error Process Kernel V2 and V2INT

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

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

#### 2.10.11.1 Error Process Registering

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

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

#### 2.10.11.2 Example of an Error Process

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

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

void errorProcess(sc_errcode_t err,const sc_errMsg_t *errMsg)
{
#ifdef __DCC__
#pragma weak errorProxy_pid
#endif
  extern sc_pid_t errorProxy_pid;
  sc_msg_t msg;
  error_hook(err,errMsg);
  if ( err & SC_ERR_MODULE_FATAL ){
    kprintf(1,
      "Modul fatal error\n"
      "Request killing\n");
      msg = sc_msgAlloc(sizeof(sc_moduleKillMsg_t),
                        SC_MODULEKILLMSG,
                        0,
                        SC_NO_TMO);
    if ( !msg ){
      kprintf(0,"Could not get a message\n");
      return;
    }
    msg->mKill.mid = save_midGet(&errMsg->mcb->id);
    msg->mKill.flags = 0;
```

```
  } else if (  err & SC_ERR_PROCESS_FATAL ){
      kprintf(1,
      "Process fatal error\n"
      "Request killing\n");
      msg = sc_msgAlloc(sizeof(sc_procKillMsg_t),
                        SC_PROCKILLMSG,
                        0,
                        SC_NO_TMO);
      if ( !msg ){
        kprintf(1,"Could not get a message\n");
        return;
      }
        msg->pKill.pid = save_pidGet(&errMsg->pcb->pid);
        msg->pKill.flag = 0;
      } else {
        return;
  }
    if ( errorProxy_pid && errorProxy_pid != SC_ILLEGAL_PID ){
      sc_msgTx(&msg, errorProxy_pid, SC_MSGTX_RTN2SND);
    }
    if ( msg ){
      sc_msgFree(&msg);
      kprintf(0,"No errorProxy\n");
    }
}
```

### 2.10.12 The Error Proxy Kernel V2 and V2INT

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

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

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

#### 2.10.12.1 Example

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

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

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

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

### 2.10.13 The errno Variable

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

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

## 2.11 Distributed Systems

### 2.11.1 Introduction

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

### 2.11.2 Connectors

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

There must be one connector per remote system.

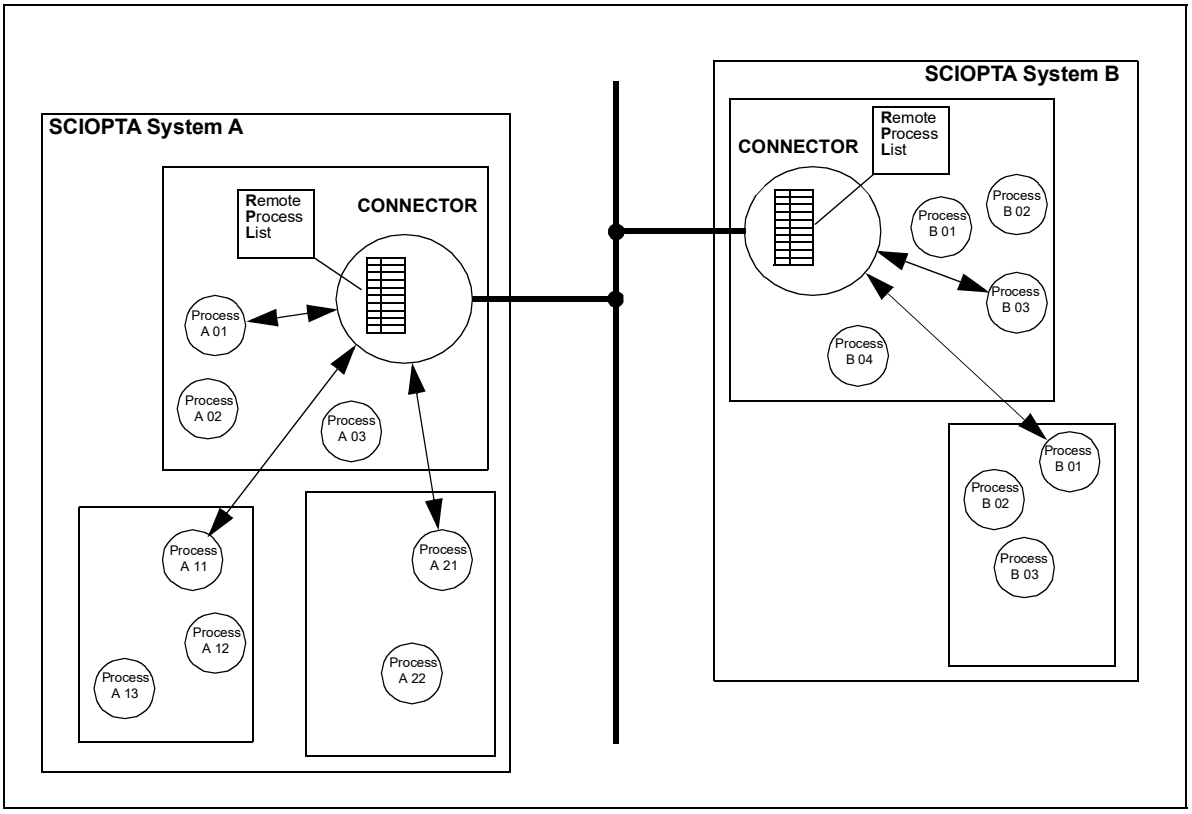


**Figure 2-6: SCIOPTA Distributed System**

### 2.11.3 Transparent Communication

If a process in one system (CPU) wants to communicate with a process in another system (CPU) it first will search for the remote process by using the "sc_procIdGet" system call. The parameter of this call includes the process name and the path to where to find it in the form:

```
//remote-system/module/procname
```

If the process does not reside on the same CPU as the caller, the kernel transmits a message to the CONNECTOR process including the inquiry. If the path and process is found in the remote process list, the CONNECTOR will assign a free PID for the system and send a reply message back to the kernel including the assigned PID. The kernel returns the PID to the caller process.

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

### 2.11.4 Unknown Process

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

Please consult chapter 2.4.12 "Message Sent to Unknown Process" on page 2-35 for a detailed description.

# 3 System Calls Overview

## 3.1 Introduction

This chapter list all SCIOPTA system calls in functional groups. Please consult chapter 4 “System Calls Reference” on page 4-1 for an alphabetical list.

**Please Note:**

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

If nothing is noted in the following lists below, the system call is valid for all three Kernel Technologies.

## 3.2 Message System Calls

| | |
|---|---|
| sc_msgAcquire | Changes the owner of the message. The caller becomes the owner of the message.<br>Chapter 4.31 “sc_msgAcquire” on page 4-40. |
| sc_msgAddrGet | Returns the process ID of the addressee of the message.<br>Chapter 4.32 “sc_msgAddrGet” on page 4-42. |
| sc_msgAlloc | Allocates a memory buffer of selectable size from a message pool.<br>Chapter 4.33 “sc_msgAlloc” on page 4-44. |
| sc_msgAllocClr | Allocates a memory buffer of selectable size from a message pool and initializes the message data to 0.<br>Chapter 4.34 “sc_msgAllocClr” on page 4-47. |
| sc_msgAllocTx | Allocates a message of 12 bytes from the default pool of the addressee and stores id, data1 and data2 in this message. Then sends the message to the addressee.<br>Chapter 4.35 “sc_msgAllocTx” on page 4-48. |
| sc_msgDataCrcDis | Disables the message data CRC check.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.36 “sc_msgDataCrcDis” on page 4-50. |
| sc_msgDataCrcGet | Checks and returns the message data checksum.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.37 “sc_msgDataCrcGet” on page 4-52. |
| sc_msgDataCrcSet | Calculates a CRC32 over the message data and stores it in the message header.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.38 “sc_msgDataCrcSet” on page 4-54. |
| sc_msgFind | Find a specific message in the allocated-messages queue of a process.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.39 “sc_msgFind” on page 4-56. |
| sc_msgFlowSignatureUpdate | Updates a global message flow signature.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.40 “sc_msgFlowSignatureUpdate” on page 4-58. |

| | |
|---|---|
| sc_msgFree | Returns a message to the message pool.<br>Chapter 4.41 “sc_msgFree” on page 4-60. |
| sc_msgHdCheck | The message header will be checked for plausibility.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.42 “sc_msgHdCheck” on page 4-62. |
| sc_msgHookRegister | Registers a message hook.<br>Chapter 4.43 “sc_msgHookRegister” on page 4-63. |
| sc_msgOwnerGet | Returns the process ID of the owner of the message.<br>Chapter 4.44 “sc_msgOwnerGet” on page 4-65. |
| sc_msgPoolIdGet | Returns the pool ID of a message.<br>Chapter 4.45 “sc_msgPoolIdGet” on page 4-67. |
| sc_msgRx | Receives a message.<br>Chapter 4.46 “sc_msgRx” on page 4-69. |
| sc_msgSizeGet | Returns the size of the message buffer.<br>Chapter 4.47 “sc_msgSizeGet” on page 4-72. |
| sc_msgSizeSet | Modifies the size of a message buffer.<br>Chapter 4.48 “sc_msgSizeSet” on page 4-74. |
| sc_msgSndGet | Returns the process ID of the sender of the message.<br>Chapter 4.49 “sc_msgSndGet” on page 4-76. |
| sc_msgTx | Sends a message to a process.<br>Chapter 4.50 “sc_msgTx” on page 4-78. |
| sc_msgTxAlias | Sends a message to a process by setting any process ID.<br>Chapter 4.51 “sc_msgTxAlias” on page 4-81. |

## 3.3 Process System Calls

| | |
|---|---|
| sc_procAtExit | Register a function to be called if a prioritized process is killed.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.60 “sc_procAtExit” on page 4-100. |
| sc_procAttrGet | Returns specific process attributes.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.61 “sc_procAttrGet” on page 4-102. |
| sc_procCBChk | Does a diagnostic test for all elements of the process control block of specific process.<br>**Only: Kernel Technologies: V2INT**<br>Chapter 4.62 “sc_procCBChk” on page 4-104. |
| sc_procCreate2 | Requests the kernel daemon to create process.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.63 “sc_procCreate2” on page 4-106. |
| sc_procDaemonRegister | Registers a process daemon which is responsible for pidGet request.<br>Chapter 4.64 “sc_procDaemonRegister” on page 4-114. |
| sc_procDaemonUnregister | Unregisters a process daemon.<br>Chapter 4.65 “sc_procDaemonUnregister” on page 4-115. |
| sc_procFlowSignatureGet | Gets the process program flow signature of the caller.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.66 “sc_procFlowSignatureGet” on page 4-116. |
| sc_procFlowSignatureInit | Initializes the process program flow signature of the caller.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.67 “sc_procFlowSignatureInit” on page 4-117. |
| sc_procFlowSignatureUpdate | Updates the process program flow signature of the caller.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.68 “sc_procFlowSignatureUpdate” on page 4-118. |
| sc_procHookRegister | Registers a process hook.<br>Chapter 4.69 “sc_procHookRegister” on page 4-119. |
| sc_procIdGet | Returns the process ID of a process.<br>Chapter 4.70 “sc_procIdGet” on page 4-121. |
| sc_procIntCreate | Request the kernel daemon to create an interrupt process.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.71 “sc_procIntCreate” on page 4-123. |
| sc_procIrqRegister | Registers an existing interrupt process for another interrupt vector.<br>Chapter 4.72 “sc_procIrqRegister” on page 4-125. |
| sc_procIrqUnregister | Unregisters previously registered interrupts.<br>Chapter 4.73 “sc_procIrqUnregister” on page 4-127. |
| sc_procKill | Requests the kernel daemon to kill a process.<br>Chapter 4.74 “sc_procKill” on page 4-128. |
| sc_procNameGet | Returns the full name of a process.<br>Chapter 4.75 “sc_procNameGet” on page 4-130. |

| | |
|---|---|
| sc_procObserve | Request a message to be sent if the given process pid dies (process supervision).<br>Chapter 4.76 “sc_procObserve” on page 4-132. |
| sc_procPathCheck | Checks if the construction of a path is correct.<br>Chapter 4.77 “sc_procPathCheck” on page 4-134. |
| sc_procPathGet | Returns the path of a process.<br>Chapter 4.78 “sc_procPathGet” on page 4-135. |
| sc_procPpidGet | Returns the process ID of the parent of a process.<br>Chapter 4.79 “sc_procPpidGet” on page 4-137. |
| sc_procPrioCreate | Requests the kernel daemon to create a prioritized process.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.80 “sc_procPrioCreate” on page 4-139. |
| sc_procPrioGet | Returns the priority of a process.<br>Chapter 4.81 “sc_procPrioGet” on page 4-142. |
| sc_procPrioSet | Sets the priority of a process.<br>Chapter 4.82 “sc_procPrioSet” on page 4-144. |
| sc_procSchedLock | Locks the scheduler and returns the number of times it has been locked before.<br>Chapter 4.83 “sc_procSchedLock” on page 4-146. |
| sc_procSchedUnlock | Unlocks the scheduler by decrementing the lock counter by one.<br>Chapter 4.84 “sc_procSchedUnlock” on page 4-147. |
| sc_procSliceGet | Returns the time slice or priority of a timer process.<br>Chapter 4.85 “sc_procSliceGet” on page 4-148. |
| sc_procSliceSet | Sets the time slice or priority of a timer process.<br>Chapter 4.86 “sc_procSliceSet” on page 4-149. |
| sc_procStart | Starts a process.<br>Chapter 4.87 “sc_procStart” on page 4-151. |
| sc_procStop | Stops a process.<br>Chapter 4.88 “sc_procStop” on page 4-153. |
| sc_procTimCreate | Requests the kernel daemon to create a timer process.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.89 “sc_procTimCreate” on page 4-155. |
| sc_procUnobserve | Cancels the observation of a process.<br>Chapter 4.90 “sc_procUnobserve” on page 4-157. |
| sc_procVarDel | Deletes a process variable.<br>Chapter 4.91 “sc_procVarDel” on page 4-158. |
| sc_procVarGet | Returns a process variable.<br>Chapter 4.92 “sc_procVarGet” on page 4-159. |
| sc_procVarInit | Initializes a process variable area.<br>Chapter 4.93 “sc_procVarInit” on page 4-160. |
| sc_procVarRm | Removes a process variable area.<br>Chapter 4.94 “sc_procVarRm” on page 4-162. |

| | |
|---|---|
| sc_procVarSet | Sets a process variable.<br>Chapter 4.95 “sc_procVarSet” on page 4-163. |
| sc_procVectorGet | Returns the interrupt vector of an interrupt process.<br>Chapter 4.96 “sc_procVectorGet” on page 4-164. |
| sc_procWakeupEnable | Enables the wakeup of a timer or interrupt process.<br>Chapter 4.97 “sc_procWakeupEnable” on page 4-165. |
| sc_procWakeupDisable | Disables the wakeup of a timer or interrupt process.<br>Chapter 4.98 “sc_procWakeupDisable” on page 4-166. |
| sc_procYield | Yields the CPU to the next ready process within the current process's priority group.<br>Chapter 4.99 “sc_procYield” on page 4-167. |

## 3.4 Module System Calls

| | |
|---|---|
| sc_moduleCBChk | Does a diagnostic test for all elements of the module control block of specific module.<br>**Only: Kernel Technologies: V2INT**<br>Chapter 4.17 “sc_moduleCBChk” on page 4-17. |
| sc_moduleCreate | Creates a module.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.18 “sc_moduleCreate” on page 4-18. |
| sc_moduleCreate2 | Creates a module.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.19 “sc_moduleCreate2” on page 4-21. |
| sc_moduleFriendAdd | Adds a module to the friendlist.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.20 “sc_moduleFriendAdd” on page 4-26. |
| sc_moduleFriendAll | Defines all existing modules in a system as friend.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.21 “sc_moduleFriendAll” on page 4-27. |
| sc_moduleFriendGet | Checks if a module is a friend.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.22 “sc_moduleFriendGet” on page 4-28. |
| sc_moduleFriendNone | Removes all modules from the friendlist.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.23 “sc_moduleFriendNone” on page 4-29. |
| sc_moduleFriendRm | Removes a module from the friendlist.<br>**Only: Kernel Technologies: V1**<br>Chapter 4.24 “sc_moduleFriendRm” on page 4-30. |
| sc_moduleIdGet | Returns the ID of a module.<br>Chapter 4.25 “sc_moduleIdGet” on page 4-31. |
| sc_moduleInfo | Returns a snap-shot of a module control block.<br>Chapter 4.26 “sc_moduleInfo” on page 4-32. |
| sc_moduleKill | Kills a module.<br>Chapter 4.27 “sc_moduleKill” on page 4-35. |
| sc_moduleNameGet | Returns the name of a module.<br>Chapter 4.28 “sc_moduleNameGet” on page 4-37. |
| sc_modulePrioGet | Returns the priority of a module.<br>Chapter 4.29 “sc_modulePrioGet” on page 4-38. |
| sc_moduleStop | Stops a whole module.<br>Chapter 4.30 “sc_moduleStop” on page 4-39.<br>**Only: Kernel Technologies: V2 and V2INT** |

## 3.5 Message Pool Calls

| | |
|---|---|
| sc_poolCBChk | Does a diagnostic test for all elements of the pool control block.<br>**Only: Kernel Technologies: V2INT**<br>Chapter 4.52 “sc_poolCBChk” on page 4-83. |
| sc_poolCreate | Creates a message pool.<br>Chapter .4.53 “sc_poolCreate” on page 4-85 |
| sc_poolDefault | Sets a message pool as default pool.<br>Chapter 4.54 “sc_poolDefault” on page 4-88. |
| sc_poolHookRegister | Registers a pool hook.<br>Chapter 4.55 “sc_poolHookRegister” on page 4-89. |
| sc_poolIdGet | Returns the ID of a message pool.<br>Chapter 4.56 “sc_poolIdGet” on page 4-91. |
| sc_poolInfo | Returns a snap-shot of a pool control block.<br>Chapter 4.57 “sc_poolInfo” on page 4-92. |
| sc_poolKill | Kills a whole message pool.<br>Chapter 4.58 “sc_poolKill” on page 4-96. |
| sc_poolReset | Resets a message pool to its original state.<br>Chapter 4.59 “sc_poolReset” on page 4-98. |

## 3.6 Safe Data Type Calls

| | |
|---|---|
| sc_safe_charGet | Returns safe data of specific char types.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.100 “sc_safe_charGet” on page 4-168. |
| sc_safe_charSet | Stores safe data of specific char types at a given address in memory.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.101 “sc_safe_charSet” on page 4-169. |
| sc_safe_intGet | Returns safe data of specific int types.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.102 “sc_safe_<type>Get” on page 4-170. |
| sc_safe_intSet | Stores safe data of specific int types at a given address in memory.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.103 “sc_safe_<type>Set” on page 4-172. |
| sc_safe_shortGet | Returns safe data of specific int types.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.104 “sc_safe_shortGet” on page 4-174. |
| sc_safe_shortSet | Stores safe data of specific int types at a given address in memory.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.105 “sc_safe_shortSet” on page 4-175. |

## 3.7 Timing Calls

| | |
|---|---|
| sc_sleep | Suspends a process for a defined time.<br>Chapter 4.106 “sc_sleep” on page 4-176. |
| sc_tmoAdd | Requests a time-out message after a defined time.<br>Chapter 4.113 “sc_tmoAdd” on page 4-184. |
| sc_tmoRm | Removes a time-out job.<br>Chapter 4.114 “sc_tmoRm” on page 4-186. |

## 3.8 System Tick Calls

| | |
|---|---|
| sc_tick | Calls the kernel tick function. Advances the kernel tick counter by one.<br>Chapter 4.107 “sc_tick” on page 4-178. |
| sc_tickActivationGet | Returns the tick time of last activation of the calling process.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.108 “sc_tickActivationGet” on page 4-179. |
| sc_tickGet | Returns the actual kernel tick counter value.<br>Chapter 4.109 “sc_tickGet” on page 4-180. |
| sc_tickLength | Returns/sets the current system tick-length.<br>Chapter 4.110 “sc_tickLength” on page 4-181. |
| sc_tickMs2Tick | Converts a time from milliseconds into ticks.<br>Chapter 4.111 “sc_tickMs2Tick” on page 4-182. |
| sc_tickTick2Ms | Converts a time from ticks into milliseconds.<br>Chapter 4.112 “sc_tickTick2Ms” on page 4-183. |

## 3.9 Process Trigger Calls

| | |
|---|---|
| sc_trigger | Signals a process trigger.<br>Chapter 4.115 “sc_trigger” on page 4-188. |
| sc_triggerValueGet | Returns the value of a process trigger.<br>Chapter 4.116 “sc_triggerValueGet” on page 4-190. |
| sc_triggerValueSet | Sets the value of a process trigger.<br>Chapter 4.117 “sc_triggerValueSet” on page 4-191. |
| sc_triggerWait | Waits on its process trigger.<br>Chapter 4.118 “sc_triggerWait” on page 4-192. |

## 3.10 CONNECTOR Process Calls

| | |
|---|---|
| sc_connectorRegister | Register Registers a connector process.<br>Chapter 4.2 “sc_connectorRegister” on page 4-1. |
| sc_connectorRemote2Local | Translates a remote PID into a local one.<br>Chapter 4.3 “sc_connectorRemote2Local” on page 4-3. |
| sc_connectorUnregister | Removes a registered connector process.<br>Chapter 4.4 “sc_connectorUnregister” on page 4-4. |

## 3.11 Miscellaneous and Error Calls

| | |
|---|---|
| sc_miscCrc | Calculates a 16 bit CRC over a specified memory range.<br>Chapter 4.5 “sc_miscCrc” on page 4-5. |
| sc_miscCrcContd | Calculates a 16 bit CRC over an additional memory range.<br>Chapter 4.6 “sc_miscCrcContd” on page 4-6. |
| sc_miscCrc32 | Calculates a 32 bit CRC over a specified memory range.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.7 “sc_miscCrc32” on page 4-7. |
| sc_miscCrc32Contd | Calculates a 32 bit CRC over an additional memory range.<br>**Only: Kernel Technologies: V2 and V2INT**<br>Chapter 4.8 “sc_miscCrc32Contd” on page 4-8. |
| sc_miscErrnoGet | Returns the process local errno variable.<br>Chapter 4.9 “sc_miscErrnoGet” on page 4-9. |
| sc_miscErrnoSet | Set Sets the error code.<br>Chapter 4.10 “sc_miscErrnoSet” on page 4-10. |
| sc_miscError | Calls the error hooks with an user error.<br>Chapter 4.11 “sc_miscError” on page 4-11. |
| sc_miscError2 | Calls the error hooks with an user error.<br>Chapter 4.12 “sc_miscError2” on page 4-12. |
| sc_miscErrorHookRegister | Registers an Error Hook.<br>Chapter 4.13 “sc_miscErrorHookRegister” on page 4-13. |

## 3.12 Simulator Calls

| | |
|---|---|
| sciopta_start | Starts a SCIOPTA Kernel Simulator application.<br>Chapter 4.120 “sciopta_start” on page 4-195. |
| sciopta_end | Starts a SCIOPTA Kernel Simulator application.<br>Chapter 4.119 “sciopta_end” on page 4-194. |

# 4 System Calls Reference

## 4.1 Introduction

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

## 4.2 sc_connectorRegister

### 4.2.1 Description

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 by the kernel to the local connector processes.

**Kernels: V1, V2 and V2INT**

### 4.2.2 Syntax

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

### 4.2.3 Parameter

| **defaultConn** | Defines the type of registered CONNECTOR. |
|---|---|
| == 0 | The caller becomes a connector process. The name of the process corresponds to the name of the target. |
| != 0 | The caller becomes the default connector for the system. The name of the process corresponds to the name of the target. |

### 4.2.4 Return Value

Process ID which is used to define the process ID for distributed processes.

### 4.2.5 Example

```
sc_pid_t connid;

connid = sc_connectorRegister(1);
```

### 4.2.6 Errors

| Error Code | Error Type | Extra Value |
|---|---|---|
| KERNEL_EILL_PROCTYPE | SC_ERR_SYSTEM_FATAL | |

| | |
|---|---|
| Caller is not a prioritized process. | e0 = Process type (see pcb.h). |

KERNEL_EALREADY_DEFINED ¦ SC_ERR_SYSTEM_FATAL

| | |
|---|---|
| Default CONNECTOR is already defined: | e0 = pid |
| Process is already a CONNECTOR: | e0 = 0 |

KERNEL_ENO_MORE_CONNECTOR

The maximum number of CONNECTORS is reached.

## 4.3 sc_connectorRemote2Local

### 4.3.1 Description

This system call is used to translate a remote PID to a local one.

**Kernels: V1, V2 and V2INT**

### 4.3.2 Syntax

```
void sc_connectorRemote2Local(void);
```

### 4.3.3 Parameter

None

### 4.3.4 Return Value

None

### 4.3.5 Example

```
sc_connectorRemote2Local;
```

### 4.3.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_CONNECTOR ¦ SC_ERR_SYSTEM_FATAL<br>Caller is not a connector process. | e0 = 0 |

## 4.4 sc_connectorUnregister

### 4.4.1 Description

This system call is used to remove a registered connector process. The caller becomes a normal prioritized process.

**Kernels: V1, V2 and V2INT**

### 4.4.2 Syntax

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

### 4.4.3 Parameter

None

### 4.4.4 Return Value

None

### 4.4.5 Example

```
sc_connectorUnregister;
```

### 4.4.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_CONNECTOR ¦ SC_ERR_SYSTEM_FATAL<br>Caller is not a connector process. | e0 = 0 |

## 4.5 sc_miscCrc

### 4.5.1 Description

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

The start value of the CRC is 0xFFFF.

**Kernels: V1, V2 and V2INT**

### 4.5.2 Syntax

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

### 4.5.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |

### 4.5.4 Return Value

The 16 bit CRC value.

### 4.5.5 Example

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

uint16_t     crc;
ips_socket_t *ref;

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

### 4.5.6 Errors

None

## 4.6 sc_miscCrcContd

### 4.6.1 Description

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

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

### 4.6.2 Syntax

```
uint16_t sc_miscCrcContd(
  const uint8_t *data,
  unsigned int  len,
  uint16_t      start
);
```

### 4.6.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |
| **start** | CRC start value. |

### 4.6.4 Return Value

The 16 bit CRC value.

### 4.6.5 Example

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

### 4.6.6 Errors

None

## 4.7 sc_miscCrc32

### 4.7.1 Description

This function calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3, polynominal: 0x04C11DB7) over a specified memory range.

The start value of the CRC is 0xFFFFFFFF.

**Kernels: V2 and V2INT**

### 4.7.2 Syntax

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

### 4.7.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |

### 4.7.4 Return Value

The inverted 32 bit CRC value.

### 4.7.5 Example

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

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

### 4.7.6 Errors

None

## 4.8 sc_miscCrc32Contd

### 4.8.1 Description

This function calculates a 32 bit cyclic redundancy check value (CRC-32-IEEE 802.3, polynomial: 0x04C11DB7) over an additional memory range.

The variable **init** is the CRC32 start value.

**Kernels: V2 and V2INT**

### 4.8.2 Syntax

```
uint32_t sc_miscCrc32Contd(
  const uint8_t *data,
  unsigned int  len,
  uint32_t      init
);
```

### 4.8.3 Parameter

| | |
|---|---|
| **data** | Pointer to the memory range. |
| **len** | Number of bytes. |
| **init** | CRC32 start value. |

### 4.8.4 Return Value

The inverted 32 bit CRC value.

### 4.8.5 Example

```
uint32_t b2crc;
uint32_t burst2[4];

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

### 4.8.6 Errors

None

## 4.9 sc_miscErrnoGet

### 4.9.1 Description

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.

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

**Kernels: V1, V2 and V2INT**

### 4.9.2 Syntax

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

### 4.9.3 Parameter

None

### 4.9.4 Return Value

Process error code.

### 4.9.5 Example

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

### 4.9.6 Errors

None

## 4.10 sc_miscErrnoSet

### 4.10.1 Description

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

Each SCIOPTA process has an errno variable.

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

**Kernels: V1, V2 and V2INT**

### 4.10.2 Syntax

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

### 4.10.3 Parameter

| | |
|---|---|
| **err** | User defined error code. |

### 4.10.4 Return Value

None.

### 4.10.5 Example

```
sc_miscErrnoSet(ENODEV);
```

### 4.10.6 Errors

None

## 4.11 sc_miscError

### 4.11.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.11.2 Syntax

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

### 4.11.3 Parameter

| **err** | User defined error code. |
|---|---|
| <error> | User error code. |
| SC_ERR_SYSTEM_FATAL | Declares error to be system fatal. Must be ored with <error> |
| SC_ERR_MODULE_FATAL | Declares error to be module fatal. Must be ored with <error> |
| SC_ERR_PROCESS_FATAL | Declares error to be process fatal. Must be ored with <error> |

| **misc** | Additional data to pass to the error hook. |
|---|---|

### 4.11.4 Return Value

None.

### 4.11.5 Example

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

### 4.11.6 Errors

None

## 4.12 sc_miscError2

### 4.12.1 Description

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

**Kernels: V2 and V2INT**

### 4.12.2 Syntax

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

### 4.12.3 Parameter

| **err** | User defined error code. |
|---|---|
| <error> | User error code. |
| SC_ERR_SYSTEM_FATAL | Declares error to be system fatal. Must be ored with <error> |
| SC_ERR_MODULE_FATAL | Declares error to be module fatal. Must be ored with <error> |
| SC_ERR_PROCESS_FATAL | Declares error to be process fatal. Must be ored with <error> |

**extra0...3** Additional data to pass to the error hook.

### 4.12.4 Return Value

None.

### 4.12.5 Example

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

### 4.12.6 Errors

None

## 4.13 sc_miscErrorHookRegister

### 4.13.1 Description

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.

**Kernels: V1, V2 and V2INT**

**Remark for Kernels V1:**

If the error hook is registered from within the system module it is registered as a global error hook. In this case the error hook registering will be done in the start hook. If the error hook is registered from within a module which is not the system module it will be registered as a module error hook.

### 4.13.2 Syntax

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

### 4.13.3 Parameter

| **newhook** | Function pointer to the hook. |
|---|---|
| <fptr> | Function pointer. |
| NULL | Will remove and unregister the hook. |

### 4.13.4 Return Value

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

0 if no error hook was registered.

### 4.13.5 Example

```
sc_errHook_t error_hook;

sc_miscErrorHookRegister( error_hook );
```

### 4.13.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_SYSTEM_FATAL<br>Process ID of caller is not valid (SC_ILLEGAL_PID) | e0 = pid |

## 4.14 sc_miscFlowSignatureGet

### 4.14.1 Description

This system call is used to get a global program flow signature at index id.

**Kernels: V2 and V2INT**

### 4.14.2 Syntax

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

### 4.14.3 Parameter

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

### 4.14.4 Return Value

Signature value.

### 4.14.5 Example

---

### 4.14.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Flow signature ID not valid (id >= SC_MAX_GFS_IDS). | e0 = Flow signature ID (id) |

## 4.15 sc_miscFlowSignatureInit

### 4.15.1 Description

This system call is used to initialize a global program flow signature at index id.

**Kernels: V2 and V2INT**

### 4.15.2 Syntax

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

### 4.15.3 Parameter

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

### 4.15.4 Return Value

None.

### 4.15.5 Example

---

### 4.15.6 Errors

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

## 4.16 sc_miscFlowSignatureUpdate

### 4.16.1 Description

This system call is used to update a global program flow signature.

**Kernels: V2 and V2INT**

### 4.16.2 Syntax

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

### 4.16.3 Parameter

| | |
|---|---|
| id | Global flow signature ID. |
| | Index of the global flow signature which is the index into the sc_globalFlowSignatures array. |
| token | Token value. |
| | Token value to calculate new signature. |

### 4.16.4 Return Value

Signature value.

### 4.16.5 Example

---

### 4.16.6 Errors

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

## 4.17 sc_moduleCBChk

### 4.17.1 Description

System call to do diagnostic test for all elements of the module control block of specific module.

**Kernels: V2INT**

### 4.17.2 Syntax

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

### 4.17.3 Parameter

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

### 4.17.4 Return Value

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

### 4.17.5 Example

---

### 4.17.6 Errors

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

## 4.18 sc_moduleCreate

### 4.18.1 Description

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.

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

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

$$p(eff) = min(p(module) + p(process), 31)$$

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

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

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

**Kernels: V1**

### 4.18.2 Syntax

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

### 4.18.3 Parameter

| | |
|---|---|
| **name** | Pointer to the module name. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore. |
| **init** | Function pointer to the init process function. |
| | This is the address where the init process of the module will start execution. |
| **stacksize** | Stacksize of the INIT process in bytes. |
| **moduleprio** | Module priority. |
| | The priority of the module which range from 0 to 31. |
| **size** | Size of the module in bytes. |
| | The minimum module size can be calculated according to the following formula (bytes):<br>size_mod = p * 256 + stack + pools + mcb + initsize<br>p = Number of static processes.<br>stack = Sum of stack sizes of all static processes.<br>pools = Sum of sizes of all message pools.<br>mcb = Size of module control block (see below)<br>initsize = Code (see parameter initsize) below<br>where: mcb can be calculated as follows:<br>mcb = 96 + friends + hooks * 4<br>friends = 0 if friends are notused, 16 if friends are used.<br>hooks = Number of hooks configured. |
| **initsize** | Size of the initialized data. |
| **max_pools** | Maximum number of pools in the module. |
| | The kernel will not allow to create more pools inside the module than stated here. Maximum value is 128. |
| **max_procs** | Maximum number of processes in the module. |
| | The kernel will not allow to create more processes inside the module than stated here. Maximum value is 16384. |

### 4.18.4 Return Value

Module ID.

### 4.18.5 Example

```
extern sc_module_addr_t m2mod;

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

### 4.18.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EMODULE_TOO_SMALL ¦ SC_ERR_SYSTEM_FATAL<br>Process control blocks and pool control blocks<br>do not fit in module size. | Module size |
| KERNEL_EILL_NAME ¦ SC_ERR_SYSTEM_FATAL<br>Requested name does not comply with SCIOPTA naming rules<br>or does already exist. | |
| KERNEL_ENO_MORE_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Maximum number of modules reached. | Number of modules |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Module addresses or sizes not valid.<br>Start address, size or initsize unaligned.<br>initsize > size. | |

## 4.19 sc_moduleCreate2

### 4.19.1 Description

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.

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

Each module has also a priority which can range between 0 (highest) to 31 (lowest) priority. This module priority defines a maximum priority level for all processes contained inside that module. The kernel will generate an error, if a process is created which has a higher priority than the module priority.

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

**Kernels: V2 and V2INT**

### 4.19.2 Syntax

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

### 4.19.3 Parameter

| | |
|---|---|
| **mdb** | Pointer to the module descriptor block (mdb) which defines the module to create. |
| | See chapter 4.19.5 “Module Descriptor Block mdb” on page 4-22. |

### 4.19.4 Return Value

Module ID.

### 4.19.5 Module Descriptor Block mdb

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

It is included in the header file modules.h.

```
struct sc_mdb_s {
  char              name[SC_MODULE_NAME_SIZE+1];
  sc_module_addr_t *maddr;
  sc_prio_t         maxPrio;
  unsigned int      maxPools;
  unsigned int      maxProcs;
  void              (*init)(void);
  sc_bufsize_t      stacksize;
  uint8_t              safetyFlag;
  uint8_t              spare_b;
  uint16_t             spare_h;
  uint32_t             *pt;
};
typedef struct sc_mdb_s sc_mdb_t;
```

### 4.19.6 Structure Members

**name** Name of the module to create.

The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore.

**maddr** Pointer to a structure containing the module addresses and size.

See chapter 4.19.7 “Module Address and Size” on page 4-23.

**maxPrio** Maximum module priority.

The priority of the module which range from 0 to 31. 0 is the highest priority.

**maxPools** Maximum number of pools in the module.

The kernel will not allow to create more pools inside the module than stated here. Maximum value is 128.

**maxProcs** Maximum number of processes in the module.

The kernel will not allow to create more processes inside the module than stated here. Maximum value is 16384.

**init** Function pointer to the init process function.

This is the address where the init process of the module will start execution.

**stacksize** Stack size of the module init process.

| **safetyFlag** | Module safety flag. |
|---|---|
| SC_KRN_FLAG_FALSE | None-safety module. |
| SC_KRN_FLAG_TRUE | Safety module. |

| **spare_b** | Spare, write 0. |
|---|---|

| **spare_h** | Spare, write 0. |
|---|---|

| **pt** | Pointer to the page table for MMU/MPU. |
|---|---|
| <pageptr> | Pointer to the MMU/MPU page table |
| NULL | If no MMU/MPU in the system. |

### 4.19.7 Module Address and Size

This is a structure which is defining the module addresses and the module size to be created. It is usually generated by the linker script.

It is defined in the header file modules.h.

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

### 4.19.8 Structure Members

| **start** | Start address of the module in RAM. |
|---|---|

| **size** | Size of the module in bytes (RAM). |
|---|---|
| | The minimum module size can be calculated as follows:<br>size = (sizeof(sc_pcb_t)+sizeof(sc_pcb_t *))*n(proc) + sum(stacks)<br>size+= (sizeof(sc_pool_cb_t *)*n(pool) + sum(pool)<br>size+= initsize + sizeof(sc_module_cb_t)<br><br>sc_pcb_t : Process Control Block<br>sc_pool_cb_t : Pool Control Block (size depends on configuration!)<br>sc_module_cb_t : Module Control Block (size depends on configuration!)<br>initsize : Module local data like initialized variables or code. Could be zero. |

| **initsize** | Size of the initialized data. |
|---|---|

### 4.19.9 Example

```
extern sc_module_addr_t M2_mod;

static const sc_mdb_t mdb = {
  /* module-name      */ "M2",
  /* module addresses */ &M2_mod, /* => linker-script */
  /* max. priority    */ 0,
  /* max. pools       */ 2,
  /* max. procs       */ 3,
  /* init-function    */ M2_init,/* init-stacksize */512,
  /* safety-flag      */ SC_KRN_FLAG_TRUE,
  /* spare values     */ 0,0,0
};

(void) sc_moduleCreate2(&mdb,0);
```

### 4.19.10 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter **mdb** not valid. 0 or SC_NIL. | |
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EMODULE_TOO_SMALL ¦ SC_ERR_SYSTEM_FATAL<br>Process control blocks and pool control blocks<br>do not fit in module size. | e0 = Module size |
| KERNEL_EILL_NAME ¦ SC_ERR_SYSTEM_FATAL<br>Requested name does not comply with SCIOPTA naming rules<br>or does already exist. | |
| KERNEL_ENO_MORE_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Maximum number of modules reached. | e0 = Number of modules |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_SYSTEM_FATAL<br>Parameter of module descriptor block not valid.<br>- maddr 0, SC_NIL or unaligned.<br>- maxPrio > 31<br>- maxPools > 128<br>- maxProcs > (MAX_PID+1)<br>- init = 0<br>- init not 4-byte alligned<br>- stacksize < SC_MIN_STACKSIZE<br>- safetyFlag neither true nor false<br>- pt not valid<br>- spares not 0 | |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Module addresses or sizes not valid.<br>Start address, size or initsize unaligned.<br>initsize > size. | |
| KERNEL_EMODULE_OVERLAP ¦ SC_ERR_SYSTEM_FATAL<br>Modules do overlap. | e0 = Requested start address<br>e1 = Module start address |

## 4.20 sc_moduleFriendAdd

### 4.20.1 Description

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

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

**Kernels: V1**

### 4.20.2 Syntax

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

### 4.20.3 Parameter

| | |
|---|---|
| **mid** | Module ID. |
| | Module ID of the new friend to add. |

### 4.20.4 Return Value

None.

### 4.20.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | Module ID |

## 4.21 sc_moduleFriendAll

### 4.21.1 Description

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

**Kernels: V1**

### 4.21.2 Syntax

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

### 4.21.3 Parameter

None.

### 4.21.4 Return Value

None.

### 4.21.5 Errors

None.

## 4.22 sc_moduleFriendGet

### 4.22.1 Description

This system call is used to check if a module is a friend. The caller will be informed if the module in parameter **mid** is a friend.

**Kernels: V1**

### 4.22.2 Syntax

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

### 4.22.3 Parameter

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

### 4.22.4 Return Value

0 if the module is not a friend (not included in the friend set)

!=0 if the mModule is a friend (included in the friend set)

### 4.22.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | Module ID |

## 4.23 sc_moduleFriendNone

### 4.23.1 Description

This system call is used to remove all modules from the friendlist.

**Kernels: V1**

### 4.23.2 Syntax

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

### 4.23.3 Parameter

None.

### 4.23.4 Return Value

None.

### 4.23.5 Errors

None.

## 4.24 sc_moduleFriendRm

### 4.24.1 Description

This system call is used to remove a module from the friendlist. The caller removes the module in parameter **mid** as friend.

**Kernels: V1**

### 4.24.2 Syntax

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

### 4.24.3 Parameter

| | |
|---|---|
| **mid** | Module ID. |
| | Module ID of the old friend to remove. |

### 4.24.4 Return Value

None.

### 4.24.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL | |
| Module ID not valid (mid >= SC_MAX_MODULE). | Module ID |

## 4.25 sc_moduleIdGet

### 4.25.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.25.2 Syntax

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

### 4.25.3 Parameter

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

### 4.25.4 Return Value

Module ID if the module name was found.

Current module ID (module ID of the caller) if parameter **name** is NULL.

SC_ILLEGAL_MID if the module name was not found.

### 4.25.5 Example

```
sc_moduleid_t mid;

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

#### 4.25.5.1 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE_NAME ¦ SC_ERR_PROCESS_WARNING<br>String pointed to by name too long. | e0 = name |

## 4.26 sc_moduleInfo

### 4.26.1 Description

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. A system level debugger or run-time debug code can use this system call to get a copy of the control blocks.

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

You cannot directly access the module control blocks.

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

**Kernels: V1, V2 and V2INT**

### 4.26.2 Syntax

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

### 4.26.3 Parameter

| | |
|---|---|
| **mid** | Module ID. |
| <mid> | mid. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |
| **info** | Pointer to a local structure of a module control block. |
| | See chapter 4.26.5 “Module Info Structure” on page 4-33. |

### 4.26.4 Return Value

1 if the module was found. In this case the **info** structure is filled with valid data.

0 if the module was not found.

### 4.26.5 Module Info Structure

The module info is a structure containing a snap-shot of the module control block.

It is included in the header file modules.h.

**For Kernels V1:**

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

**For Kernels V2 and V2INT:**

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

### 4.26.6 Structure Members

| | |
|---|---|
| **mid** | Module ID. |
| **ppid** | ID of the parent process. Process from where the module was created. |
| **name** | Name of the module. |
| **text** | Current address into module text segment. |
| **textsize** | Size of module text segment (initialized data asi t does also contain static variables). |

| | |
|---|---|
| **data** | Start address of the module's data area. |
| **datasize** | Total size of the module's data area. |
| **freesize** | Free size of module. |
| **max_process** | Maximum defined number of processes in the module. |
| **nprocess** | Actual number of processes. |
| **max_pools** | Maximum defined number of pools in the module. |
| **npools** | Actual number of pools. |

### 4.26.7 Example

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

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

### 4.26.8 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter **info** not valid (info == 0). | |
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |

## 4.27 sc_moduleKill

### 4.27.1 Description

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.

All processes and pools in the module will be killed and removed. The system call will return when the whole kill process is done. The system module cannot be killed.

**Kernels: V1, V2 and V2INT**

### 4.27.2 Syntax

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

### 4.27.3 Parameter

| **mid** | Module ID. |
|---|---|
| <mid> | Module ID of the module to be killed and removed. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

| **flags** | Module kill flags. |
|---|---|
| 0 | A cleaning up will be executed. |
| SC_MODULEKILL_KILL | No cleaning up will be done. |

### 4.27.4 Return Value

None.

### 4.27.5 Example

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

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

### 4.27.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Module to be killed is the system module.<br>MID is not valid (mid >= SC_MAX_MODULE).<br>MID is not valid ( mcb ==SC_NIL). | e0 = mid. |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_SYSTEM_FATAL<br>Caller is not a prioritized process. | e0 = pcb. |

## 4.28 sc_moduleNameGet

### 4.28.1 Description

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

The name will be returned as a 0 terminated string.

**Kernels: V1, V2 and V2INT**

### 4.28.2 Syntax

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

### 4.28.3 Parameter

| **mid** | Module ID. |
|---|---|
| <mid> | Module ID. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

### 4.28.4 Return Value

Name string of the module if the module was found.

0 if the module was not found.

### 4.28.5 Example

```
sc_moduleid_t mid;

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

### 4.28.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |

## 4.29 sc_modulePrioGet

### 4.29.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.29.2 Syntax

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

### 4.29.3 Parameter

| **mid** | Module ID. |
|---|---|
| <mid> | Module ID. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

### 4.29.4 Return Value

Module priority if the module was found and was valid.

SC_ILLEGAL_PRIO if the module was not valid (mcb == SC_NIL).

### 4.29.5 Example

```
sc_moduleid_t mid;

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

### 4.29.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |

## 4.30 sc_moduleStop

### 4.30.1 Description

This system call is used to stop a module.

It will stop all processes in a module.

The process stop will be done in the order of their process ID. First all interrupt and timer processes will be stopped and then all prioritized processes are stopped.

The stop behaves identically as the sc_procStop system call for the respective process types.

**Kernels: V2 and V2INT**

### 4.30.2 Syntax

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

### 4.30.3 Parameter

| **mid** | ID of module to be stopped. |
|---|---|
| <mid> | Module ID. |
| SC_CURRENT_MID | Current module ID (module ID of the caller). |

### 4.30.4 Return Value

None.

### 4.30.5 Example

```
sc_moduleid_t mid;

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

### 4.30.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_SYSTEM_FATAL<br>Module ID not valid (mid >= SC_MAX_MODULE). | e0 = mid |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Stopcounter overrun. | e0 = pcb |

## 4.31 sc_msgAcquire

### 4.31.1 Description

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 if the message resides not in a pool of the callers module and the callers module is not friend to the module where the message resides. In this case the message pointer (**msgptr**) will be modified.

Please use sc_msgAcquire with care. Transferring message buffers without proper ownership control by using sc_msgAcquire instead of transmitting and receiving messages with sc_msgTx and sc_msgRx will cause problems if you are killing processes.

**Kernels: V1, V2 and V2INT**

### 4.31.2 Syntax

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

### 4.31.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to the message buffer pointer. |

### 4.31.4 Return Value

None.

### 4.31.5 Example

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

sc_msg_t msg;
sc_msg_t msg2;

msg = sc_msgRx(SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID);

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

### 4.31.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EMSG_HD_CORRUPT ¦ SC_ERR_MODULE_FATAL | |
| Message header is corrupt. Kernel is the message owner. | e0 = Pointer to message. |
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL | |
| Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL | |
| Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL | |
| Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.32 sc_msgAddrGet

### 4.32.1 Description

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

This system call is used in communication software of distributed multi-CPU systems (using connector processes). It allows to retrieve the original addressee when you are forwarding a message by using the sc_msgTxAlias system call.

**Kernels: V1, V2 and V2INT**

### 4.32.2 Syntax

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

### 4.32.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.32.4 Return Value

Process ID of the addressee of the message.

### 4.32.5 Example

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

sc_msg_t msg;
sc_pid_t addr;

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

### 4.32.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.33 sc_msgAlloc

### 4.33.1 Description

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.

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

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

The caller can define how the system will respond to memory shortage in message pools (**tmo**).

**Kernels: V1, V2 and V2INT**

### 4.33.2 Syntax

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

### 4.33.3 Parameter

| | |
|---|---|
| **size** | The requested size of the message buffer. |
| **id** | Message ID. |
| | The message ID which will be placed at the beginning of the data buffer of the message. |

| **plid** | Pool ID from where the message will be allocated. |
|---|---|
| <pool id> | Pool ID from where the message will be allocated. |
| SC_DEFAULT_POOL | Message will be allocated from the default pool. The default pool can be set by the system call sc_poolDefault. |

| **tmo** | Allocation timing parameter. |
|---|---|
| SC_ENDLESS_TMO | Timeout is not used. Blocks and waits endless until a buffer is available from the message pool.<br>**Note**: This parameter is not recommended. Use with caution. |
| SC_NO_TMO | A NIL pointer will be returned if there is memory shortage in the message pool. |
| SC_FATAL_IF_TMO | A (fatal) kernel error will be generated if a message buffer of the requested size is not available. The function will not return. |
| 0 < tmo =< SC_TMO_MAX | Timeout value in system ticks. Alloc with timeout. Blocks and waits the specified number of ticks to get a message buffer. |

### 4.33.4 Return Value

Pointer to the allocated buffer or

NIL pointer if: **tmo** = SC_NO_TMO
**tmo** = > 0 The timeout expired before a buffer could be allocated..

### 4.33.5 Example

```
/* Allocate TEST_MSG from default pool */

sc_msg_t msg;

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

### 4.33.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_POOL_ID ¦ SC_ERR_PROCESS_FATAL<br>Pool index is not available. | e0 = Pool index |
| KERNEL_EILL_BUFSIZE ¦ SC_ERR_PROCESS_FATAL<br>Illegal message size was requested. | e0 = Requested size<br>e1 = Pool CB (or -1) |
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL<br>Request for number of bytes could not be fulfilled. | e0 = size<br>e1 = Pool CB |
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>tmo value not valid. | e0 = tmo value |
| KERNEL_EILL_DEFPOOL_ID ¦ SC_ERR_PROCESS_WARNING<br>Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook. | e0 = Pool index |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Process would swap but interrupts and/or scheduler are/is locked. | e0 = Lock counter value or<br>-1 if interrupt are locked. |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>tmo-flag with wrong value. Likely system is corrupt. | e0 = tmo-flag |

## 4.34 sc_msgAllocClr

### 4.34.1 Description

This system call works exactly the same as sc_msgAlloc but it will initialize the data area of the message to 0.

**Kernels: V1, V2 and V2INT**

### 4.34.2 Syntax

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

### 4.34.3 Parameter

Parameter values are the same as in chapter 4.33 “sc_msgAlloc” on page 4-44.

### 4.34.4 Return Value

Return values are the same as in chapter 4.33 “sc_msgAlloc” on page 4-44.

### 4.34.5 Example

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

sc_msg_t msg;

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

### 4.34.6 Error

Same as in chapter 4.33.6 “Errors” on page 4-46.

## 4.35 sc_msgAllocTx

### 4.35.1 Description

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

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

**Kernels: V1, V2 and V2INT**

### 4.35.2 Syntax

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

### 4.35.3 Parameter

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

### 4.35.4 Return Value

None.

### 4.35.5 Example

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

### 4.35.6 Error

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Addressee pid not valid (is init0).<br>bigger than MODULE_MAXPROCESS).<br>Illegal CONNECTOR pid.<br>Illegal addressees module-index. | e0 = Addressee process ID |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = Process type |
| KERNEL_EILL_DEFPOOL_ID ¦ SC_ERR_PROCESS_WARNING | |
| Illegal default pool index. This is a warning and will continue with pool 0 upon return from error hook.<br>Thrown in the addressee's module. | e0 = Pool index |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL | |
| Pool index in message header has an illegal value.<br>Thrown in the addressee's module. | e0 = Pool index |
| KERNEL_EOUTSIDE_POOL ¦ SC_ERR_MODULE_FATAL | |
| The pointer is outside the pool. Possible pool id corruption.<br>Thrown in the addressee's module. | e0 = Pointer to message header |
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL | |
| Request for number of bytes could not be fulfilled.<br>Thrown in the addressee's module. | e0 = size<br>e1 = Pool CB |

## 4.36 sc_msgDataCrcDis

### 4.36.1 Description

This system call is used to disable the message data CRC check for the message in the parameter..

**Kernels: V2 and V2INT**

### 4.36.2 Syntax

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

### 4.36.3 Parameter

| | |
|---|---|
| **msg** | Pointer to the message pointer. |

### 4.36.4 Return Value

None.

### 4.36.5 Example

---

### 4.36.6 Error

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = PID of owner<br>e1 = Pointer to message header |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.37 sc_msgDataCrcGet

### 4.37.1 Description

This system call is used to check the message data checksum.

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

**Kernels: V2 and V2INT**

### 4.37.2 Syntax

```
_u32 sc_msgDataCrcGet(
  sc_msgptr_t msg
);
```

### 4.37.3 Parameter

| | |
|---|---|
| **msg** | Pointer to the message pointer. |

### 4.37.4 Return Value

CRC of the message data if the CRC was successful.

0 if the message data is corrupt or no check is performed.

### 4.37.5 Example

---

### 4.37.6 Error

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = PID of owner<br>e1 = Pointer to message header |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.38 sc_msgDataCrcSet

### 4.38.1 Description

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

This system call is used to calculate a CRC32 over the message data and store it in the message header.

**Kernels: V2 and V2INT**

### 4.38.2 Syntax

```
_u32 sc_msgDataCrcSet(
  sc_msgptr_t msg
);
```

### 4.38.3 Parameter

| | |
|---|---|
| **msg** | Pointer to the message pointer. |

### 4.38.4 Return Value

CRC of the message data.

### 4.38.5 Example

---

### 4.38.6 Error

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = PID of owner<br>e1 = Pointer to message header |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.39 sc_msgFind

### 4.39.1 Description

This system call is used to find messages which have been allocated or already received. The allocated-messages queue of the caller will be searched for the desired messages. This call is similar to **sc_msgRx** (see chapter 4.46 “sc_msgRx” on page 4-69) but instead of searching the messages-input queue the allocated-messages queue will be scanned. This queue holds the messages which have already been received (by **sc_msgRx**) or messages which have been allocated by the caller process.

If a message matching the conditions is found the kernel will return to the caller. If the allocated-messages queue is empty or no wanted messages are available in the queue a NULL will be returned. The call sc_msgFind will not block the system.

A pointer to an array (**wanted**) containing the messages and/or process IDs which will be scanned by sc_msgFind must be given. The array must be terminated by 0.

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

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

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

**Kernels: V2 and V2INT**

### 4.39.2 Syntax

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

### 4.39.3 Parameter

| | |
|---|---|
| **mp** | Pointer to a message in the allocated-messages queue. |
| | Pointer to the message in the queue from where the find scanning will start. |
| !=NULL | Scanning starts from the head of the allocated-messages queue. |

| | |
|---|---|
| **wanted** | Pointer to the message (or pid) array. |
| <ptr> | Pointer to the message (or process ID) array. |
| SC_FIND_ALL | All messages will be searched. |

**flag** Find flag.

| | |
|---|---|
| | More than one value can be defined and must be separated by OR instructions. |
| SC_FIND_MSGID | An array of wanted message IDs is given. |
| SC_FIND_PID | An array of process ID´s from where sent messages are received is given. |
| SC_FIND_NOT | An array of message ID's is given which will be excluded from the search. |
| SC_FIND_BOTH | An array of pairs of message ID´s and process ID´s are given to search for specific messages from specific transmitting processes. |

### 4.39.4 Return Value

Pointer to the found message.

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

### 4.39.5 Examples

TBD

### 4.39.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal flags. | e0 = flags |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>tmo-flag with wrong value. Likely system is corrupt. | e0 = tmo-flag |

## 4.40 sc_msgFlowSignatureUpdate

### 4.40.1 Description

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

**Kernels: V2 and V2INT**

### 4.40.2 Syntax

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

### 4.40.3 Parameter

| | |
|---|---|
| **id** | Global flow signature ID. |
| | Index of the global flow signature which is the index into the sc_globalFlowSignatures array. |
| **msg** | Pointer to message. |

### 4.40.4 Return Value

Signature value.

### 4.40.5 Example

---

### 4.40.6 Error

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Flow signature ID not valid (id >= SC_MAX_GFS_IDS). | e0 = Flow signature ID (id) |
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Message pointer. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.41 sc_msgFree

### 4.41.1 Description

This system call is used to return a newly allocated message to the message pool. Message buffers which have been returned can be used again.

Only the owner of a message is allowed to free it by calling sc_msgFree. It is a fatal error to free a message owned by another process. 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.

**Kernels: V1, V2 and V2INT**

### 4.41.2 Syntax

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

### 4.41.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.41.4 Return Value

None.

### 4.41.5 Example

```
/* Free a message */

sc_msg_t msg;

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

### 4.41.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Process ID is wrong. | e0 = Process ID |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Module in message header has an illegal value. | e0 = Pointer to message header |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL<br>Pool index in message header has an illegal value. | e0 = Pointer to message header |
| KERNEL_EMSG_HD_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Either pool ID or buffersize index are corrupted. | e0 = Pointer to message header |
| KERNEL_EOUTSIDE_POOL ¦ SC_ERR_MODULE_FATAL<br>The pointer is outside the pool. Possible pool id corruption. | e0 = Pointer to message header |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Message is a timout message. | |

## 4.42 sc_msgHdCheck

### 4.42.1 Description

It is used to check the message header. The header will be checked for plausibility.

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

**Kernels: V2 and V2INT**

### 4.42.2 Syntax

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

### 4.42.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.42.4 Return Value

!= 0 if the message header is ok.

== 0 if the message header is corrupted.

### 4.42.5 Example

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

if (sc_msgHdCheck(&msg)) {
  printf ("Message ok!"
} else {
  printf ("Message corrupted!"
};
```

### 4.42.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message pointer or pointer to message are zero. | |

## 4.43 sc_msgHookRegister

### 4.43.1 Description

This system call will register a message hook.

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

**Kernels V1 only:** If sc_msgHookRegister is called from within a module a module message hook will be registered.

A global message hook will be registered when sc_msgHookRegister is called from the start hook function which is called before SCIOPTA is initialized.

Each time a message is sent or received (depending on the setting of parameter **type**) the module message hook of the caller will be called if such a hook exists. **Kernels V1 only:** First the module and then the global message hook will be called.

**Kernels: V1, V2 and V2INT**

### 4.43.2 Syntax

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

### 4.43.3 Parameter

| **type** | Defines the type of registerred CONNECTOR. |
|---|---|
| SC_SET_MSGTX_HOOK | Registers a message transmit hook. Every time a message is sent, this hook will be called. |
| SC_SET_MSGRX_HOOK | Registers a message receive hook. Every time a message is received, this hook will be called. |

| **newhook** | Message hook function pointer. |
|---|---|
| <funcptr> | Function pointer to the message hook. |
| NULL | Removes and unregisters the message hook. |

### 4.43.4 Return Value

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

0 if no message hook was registered.

### 4.43.5 Example

```
sc_msgHook_t oldMsgHook;

oldMsgHook = sc_msgHookRegister( SC_SET_MSGRX_HOOK,rxHook );
oldMsgHook = sc_msgHookRegister( SC_SET_MSGTX_HOOK,txHook );
```

### 4.43.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Wrong type (unknown or not active) | e0 = Requested message hook type. |

## 4.44 sc_msgOwnerGet

### 4.44.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.44.2 Syntax

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

### 4.44.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.44.4 Return Value

Process ID of the owner of the message.

### 4.44.5 Example

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

sc_msg_t msg;
sc_pid_t owner;

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

### 4.44.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.45 sc_msgPoolIdGet

### 4.45.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.45.2 Systax

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

### 4.45.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.45.4 Return Value

Pool ID where the message resides if the message is in the same module than the caller.

SC_DEFAULT_POOL if the message is not in the same module than the caller.

### 4.45.5 Example

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

sc_msg_t msg;
sc_poolid_t idx;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID );

idx = sc_msgPoolIdGet( &msg );
```

### 4.45.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.46 sc_msgRx

### 4.46.1 Description

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

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

A pointer to an array (**wanted**) containing the messages (and/or process IDs) which will be scanned by **sc_msgRx**. The array must be terminated by 0. **Kernel V2 only**: The kernel stores the pointer to the array in the process control block for debugging.

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

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

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

The caller can also specify a timeout value **tmo**. The caller will not wait (swapped out) longer than the specified time. If the timeout expires the process will be made ready again and sc_msgRx will return with **NULL**.

**Kernerl V2 only:** The activation time is saved for **sc_msgRx** in prioritized processes. The activation time is the absolute time (tick counter) value when the process became ready.

**Kernels: V1, V2 and V2INT**

### 4.46.2 Syntax

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

### 4.46.3 Parameter

| **tmo** | Timeout parameter. |
|---|---|
| SC_ENDLESS_TMO | Blocks and waits endless until the message is received. |
| SC_NO_TMO | No time out, returns immediately. Must be set for interrupt and timer processes. |
| 0 < tmo =< SC_TMO_MAX | Timeout value in system ticks. Receive with timeout. Blocks and waits a specified maximum number of ticks to receive the message. If the timeout expires the process will be made ready again and sc_msgRx will return with NULL. |

| **wanted** | Pointer to the message (or pid) array. |
|---|---|
| <ptr> | Pointer to the message (or process ID) array. |
| SC_MSGRX_ALL | All messages will be received. |

| **flag** | Receive flag. |
|---|---|
| | More than one value can be defined and must be separated by OR instructions. |
| SC_MSGRX_MSGID | An array of wanted message IDs is given. |
| SC_MSGRX_PID | An array of process ID´s from where sent messages are received is given. |
| 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. |
| SC_MSGRX_NOT | An array of message ID's is given which will be excluded from receive. |

### 4.46.4 Return Value

Pointer to the received message if the message has been received. The caller becomes owner of the received message.

NULL if timeout expired. The process will be made ready again.

### 4.46.5 Examples

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

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

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

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

```
/* wait for message from a certain process */

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

/* Wait for any message */

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

### 4.46.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>tmo value not valid. | e0 = tmo value |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Process would swap but interrupts and/or scheduler are/is locked. | e0 = Lock counter value or -1 if interrupt are locked. |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal flags. | e0 = flags |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>tmo-flag with wrong value. Likely system is corrupt. | e0 = tmo-flag |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>The calling process uses a timeout and is not a prioritized process. | |

## 4.47 sc_msgSizeGet

### 4.47.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.47.2 Syntax

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

### 4.47.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.47.4 Return Value

Requested size of the message.

### 4.47.5 Example

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

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

size = sc_msgSizeGet( &msg );
```

### 4.47.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.48 sc_msgSizeSet

### 4.48.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.48.2 Syntax

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

### 4.48.3 Parameter

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

### 4.48.4 Return Value

New requested buffer size if call without error condition.

Old requested buffer size if it was a wrong request such as requesting a higher buffer size as the old one.

### 4.48.5 Example

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

sc_msg_t msg:

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID );

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

sc_msgSizeSet( &msg, sizeof(reply_msg_t) ); /* reduce size before returning */

sc_msgTx( &msg, sc_msgSndGet(&msg), 0 );    /* return to sender (ACK) */
```

### 4.48.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_BUFSIZE ¦ SC_ERR_MODULE_FATAL<br>Illegal buffer sizes. | e0 = Buffer sizes |
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_EILL_VALUE ¦ SC_ERR_MODULE_FATAL<br>Parameter size is smaller than the size of a message id. | e0 = Buffer sizes |
| KERNEL_EENLARGE_MSG ¦ SC_ERR_MODULE_FATAL<br>Message would be enlarged. | e0 = Buffer size<br>e1 = Pointer to message |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.49 sc_msgSndGet

### 4.49.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.49.2 Syntax

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

### 4.49.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

### 4.49.4 Return Value

Process ID of the sender of the message if the message was sent at least once.

Process ID of the owner of the message if the message was never sent.

### 4.49.5 Example

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

sc_msg_t msg;
sc_pid_t sndr;

msg = sc_msgRx( SC_ENDLESS_TMO, SC_MSGRX_ALL , SC_MSGRX_MSGID );

sndr = sc_msgSndGet( &msg );
```

### 4.49.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |

## 4.50 sc_msgTx

### 4.50.1 Description

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 kernel will become the owner of the message. **NULL** is loaded into the caller´s message pointer **msgptr** to avoid unintentional message access by the caller after transmitting.

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

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

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

**Kernels: V1, V2 and V2INT**

### 4.50.2 Syntax

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

### 4.50.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |

| | |
|---|---|
| **addr** | The process ID of the addressee. |
| <pid> | Valid SCIOPTA PID |
| SC_CURRENT_PID | The caller himself. |

| | |
|---|---|
| **flags** | Transmitt flags. |
| SC_MSGTX_NO_FLAG | Normal sending. |
| SC_MSGTX_RTN2SNDR | Return message if addressee does not exist or if there is no memory to copy it into the addressee's module. |

### 4.50.4 Return Value

None.

### 4.50.5 Example

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

sc_msg_t msg;
sc_pid_t addr;

/* ... */
msg = sc_msgAlloc( sizeof(test_msg_t),TEST_MSG, SC_DEFAULT_POOL, SC_FATAL_IF_TMO );

sc_msgTx( &msg, sndr, 0 );
```

### 4.50.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_MODULE_FATAL<br>Either pointer to message or pointer to message pointer are zero. | e0 = Pointer to message pointer |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |
| KERNEL_EMSG_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Message endmark is corrupt. | e0 = Pointer to message. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Endmark of previous message is corrupt. | e0 = Pointer to previous message. |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Process type not valid. | e0 = pid of addressee<br>e1 = Process type |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Flag is neither 0 nor SC_MSGTX_RTN2SND | e0 = Flag value<br>e1 = 2 (position) |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_PROCESS_FATAL<br>Caller tries to send a timeout message but message is already a timeout message. | e0 = Pointer to message header |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Module in message header has an illegal value. | e0 = Pointer to message header |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL<br>Pool index in message header has an illegal value. | e0 = Pointer to message header |
| KERNEL_EMSG_HD_CORRUPT ¦ SC_ERR_MODULE_FATAL<br>Either pool ID or buffersize index are corrupted. | e0 = Pointer to message header |
| KERNEL_EOUTSIDE_POOL ¦ SC_ERR_MODULE_FATAL<br>The pointer is outside the pool. Possible pool id corruption. | e0 = Pointer to message header |

## 4.51 sc_msgTxAlias

### 4.51.1 Description

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

Otherwise sc_msgTxAlias works the same way as sc_msgTx.

**Kernels: V1, V2 and V2INT**

### 4.51.2 Syntax

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

### 4.51.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to message pointer. |
| **addr** | The process ID of the addressee. |
| <pid> | Valid SCIOPTA PID |
| SC_CURRENT_PID | The caller himself. |
| **flags** | Sending flags. |
| SC_MSGTX_NO_FLAG | Normal sending. |
| SC_MSGTX_RTN2SNDR | Return message if addressee does not exist or if there is no memory to copy it into the addressee's module. |
| **alias** | The process ID specified as sender. |

### 4.51.4 Return Value

None.

### 4.51.5 Example

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

sc_msg_t msg;
sc_pid_t addr;
sc_pid_t other;


/* ... */

msg = sc_msgAlloc( sizeof(test_msg_t),TEST_MSG, SC_DEFAULT_POOL, SC_FATAL_IF_TMO );

sc_msgTxAlias( &msg, sndr, 0, other );
```

### 4.51.6 Errors

Same errors as in chapter 4.50.6 “Errors” on page 4-79.

## 4.52 sc_poolCBChk

### 4.52.1 Description

System call to do diagnostic test for all elements of the pool control block of specific message pool.

**Kernels: V2INT**

### 4.52.2 Syntax

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

### 4.52.3 Parameter

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

### 4.52.4 Return Value

== 0 if the mid or idx is wrong.

== 1 if the pool control block is correct and therefore not corrupted.

== -1 if the pool control block is corrupted.

### 4.52.5 Example

---

### 4.52.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_PROCESS_FATAL<br>Parameter mid not valid (>= SC_MAX_MODULE). | e0 = mid |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_PROCESS_FATAL<br>Pool index too large. | e0 = pool index |

## 4.53 sc_poolCreate

### 4.53.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.53.2 Syntax

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

### 4.53.3 Parameter

| | |
|---|---|
| **start** | Start address of the pool. |
| <start> | Start address |
| 0 | The kernel will automatically take the next free address in the module |
| **size** | Size of the message pool. |
| | The minimum size must be the size of the maximum number of messages which ever are allocated at the same time plus the pool control block (pool_cb).<br>The size of the pool control block (pool_cb) can be calculated<br>according to the following formula:<br>pool_cb = 68 + n * 20 + stat * n * 20<br>n Maximum buffer sizes defined for the whole system<br>(and not the buffer sizes of the created pool). Value n can be 4, 8 or 16.<br>stat Process statistics or message statistics are used (1) or not used (0). |
| **nbufs** | The number of fixed buffer sizes. |
| | This can be 4, 8 or 16. It must always be lower or equal of the fixed buffer sizes which is defined for the whole system. |
| **bufsizes** | Pointer to an array of the fixed buffer sizes in ascending order. |
| **name** | Pointer to the name of the pool to create. |
| | The name is represented by a ASCII character string terminated by 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore. |

### 4.53.4 Return Value

Pool ID of the created message pool.

### 4.53.5 Example

```
static const sc_bufsize_t bufsizes[8]=
{
   4,
   8,
   16,
   32,
   64,
   128,
   256,
   700
};

myPool_plid = sc_poolCreate(
   /* start-address     */ 0,
   /* total size        */ 4000,
   /* number of buffers */ 8,
   /* buffersizes       */ bufsizes,
   /* name              */ "myPool"
);
```

### 4.53.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Illegal module.<br>module >= SC_MAX_MODULE<br>module == SC_NIL | e0 = Module ID |
| KERNEL_EILL_BUF_SIZES ¦ SC_ERR_MODULE_FATAL<br>Illegal buffer sizes. | e0 = Requested buffer sizes. |
| KERNEL_EILL_NAME ¦ SC_ERR_MODULE_FATAL<br>Illegal pool name requested. | e0 = Requested pool name |
| KERNEL_ENO_MORE_POOL ¦ SC_ERR_MODULE_FATAL<br>Maximum number of pools for module reached. | e0 = Number of pools in module cb. |
| KERNEL_EILL_NUM_SIZES ¦ SC_ERR_MODULE_FATAL<br>Illegal number of buffer sizes. | e0 = Number of requested buffer sizes. |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL<br>No more memory in module for pool. | e0 = Requested pool size. |
| KERNEL_EILL_POOL_SIZE ¦ SC_ERR_MODULE_FATAL<br>Size is not 4-byte aligned<br>Even the biggest buffer does not fit. | e0 = Requested pool size. |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_MODULE_FATAL<br>Pool start address is not 4-byte aligned. | e0 = Requested pool start address. |

## 4.54 sc_poolDefault

### 4.54.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.54.2 Syntax

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

### 4.54.3 Parameter

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

### 4.54.4 Return Value

Pool ID of the default pool.

### 4.54.5 Example

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

### 4.54.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_POOL_ID ¦ SC_ERR_PROCESS_WARNING | |
| Pool index not valid.<br>Pool index > 16<br>Pool index > MODULE_MAXPOOLS | e0 = Requested pool index. |

## 4.55 sc_poolHookRegister

### 4.55.1 Description

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

**V1 only:** There can be one pool create and one pool kill hook per module. If sc_poolHookRegister is called from within a module a module pool hook will be registered.

A global pool hook will be registered when sc_poolHookRegister is called from the start hook function which is called before SCIOPTA is initialized.

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

**Kernels: V1, V2 and V2INT**

### 4.55.2 Syntax

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

### 4.55.3 Parameter

| **type** | Defines the type of registered pool hook. |
|---|---|
| SC_SET_POOLCREATE_HOOK | Registers a pool create hook. Every time a pool is created, this hook will be called. |
| SC_SET_POOLKILL_HOOK | Registers a pool kill hook. Every time a pool is killed, this hook will be called. |
| **newhook** | Pool hook function pointer. |
| <funcptr> | Function pointer to the hook |
| NULL | The pool hook will be removed and unregistered. |

### 4.55.4 Return Value

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

NULL if no pool hook was registered.

### 4.55.5 Example

```
sc_poolHook_t oldPoolHook;

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

### 4.55.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Pool hook type not defined. | e0 = Pool hook type.<br>e1 = 0 |
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Pool hook function pointer not valid. | e0 = Pool hook type.<br>e1 = 1 |

## 4.56 sc_poolIdGet

### 4.56.1 Description

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

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

**Kernels: V1, V2 and V2INT**

### 4.56.2 Syntax

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

### 4.56.3 Parameter

| **name** | Pool name. |
|---|---|
| <name> | Pointer to the 0 terminated name string. |
| NULL | Default pool. |
| SC_NIL | Default pool. |
| Empty string | Default pool. |

### 4.56.4 Return Value

Pool ID if pool was found.

SC_ILLEGAL_POOLID if pool was not found.

### 4.56.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet( "fs_pool" );

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

### 4.56.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_NAME ¦ SC_ERR_PROCESS_FATAL<br>Illegal pool name. | e0 = pointer to pool name. |

## 4.57 sc_poolInfo

### 4.57.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.57.2 Syntax

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

### 4.57.3 Parameter

| | |
|---|---|
| **mid** | Module ID where the pool resides of which the control block will be returned. |
| **plid** | ID of the pool of which the pool control block data will be returned. |
| **info** | Pointer to a local structure of a pool control block. See chapter<br>This structure will be filled with the pool control block data.<br>It is included in the header file pool.h. |

### 4.57.4 Return Value

!= 0 if the pool control block data was successfully retrieved.

== 0 if the pool control block could not be retrieved.

### 4.57.5 Pool Control Block Structure

The pool info is a structure containing a snap-shot of the pool control block.

It is included in the header file pool.h.

```
struct sc_pool_cb_s{
  sc_save_poolid_t poolid;

  sc_save_voidptr_t start;
  sc_save_voidptr_t end;
  sc_save_voidptr_t cur;
  sc_save_uint_t lock;

  sc_save_uint_t nbufsizes;
  sc_save_plsize_t size;
  sc_save_pid_t creator;

  sc_save_bufsize_t bufsizes[SC_MAX_NUM_BUFFERSIZES];
  idbl_t freed[SC_MAX_NUM_BUFFERSIZES];
  idbl_t waiter[SC_MAX_NUM_BUFFERSIZES];

  char name[SC_POOL_NAME_SIZE+1];

#if SC_MSG_STAT == 1
  sc_pool_stat_t stat;
#endif

} sc_pool_cb_t;
```

### 4.57.6 Structure Members

| | |
|---|---|
| **poolid** | Pool ID. |
| **start** | Start of pool-data area. |
| **end** | End of pool (first byte not in pool). |
| **cur** | First free byte inside pool. |
| **lock** | Lock setting. Not locked if 0. |
| **nbufsizes** | Number of buffer sizes. |
| **size** | Complete pool size. |

| | |
|---|---|
| **creator** | Process which created the pool. |
| **bufsizes** | Array of buffers. |
| **freed** | List of free'd buffers. |
| **waiter** | List of processes waiting for a buffer. |
| **name** | Pointer to pool name. |
| **stat** | Statistics information. See chapter 4.57.7 "Pool Statistics Info Structure" on page 4-94. |

### 4.57.7 Pool Statistics Info Structure

The pool statistics info is a structure inside the pool control block containing containing pool statistics information.

It is included in the header file pool.h.

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

### 4.57.8 Structure Members

| | |
|---|---|
| **cnt_req** | Number of buffer requests for a specific size. |
| **cnt_alloc** | Number of buffer allocations of a specific size. |
| **cnt_free** | Number of buffer releases of a specific size. |
| **cnt_wait** | Number of unfullfilled buffer allocations of specific size. |
| **maxalloc** | Largest wanted size. |

### 4.57.9 Example

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

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

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

### 4.57.10 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Illegal pointer to **info** structure. | e0 = 0. |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL<br>Illegal pool ID. | e0 = pool ID. |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Illegal module. | e0 = module ID. |

## 4.58 sc_poolKill

### 4.58.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.58.2 Syntax

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

### 4.58.3 Parameter

| | |
|---|---|
| **plid** | ID of the pool to be killed. |

### 4.58.4 Return Value

None.

### 4.58.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet( "my_pool" );

sc_poolKill( pl );
```

### 4.58.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPOOL_IN_USE ¦ SC_ERR_MODULE_FATAL<br>Pool is in use and cannot be killed. | e0 = pool cb.<br>e1 = pool lock counter. |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL<br>Illegal pool ID. | e0 = pool ID. |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Illegal module. | e0 = module ID. |

## 4.59 sc_poolReset

### 4.59.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.59.2 Syntax

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

### 4.59.3 Parameter

| | |
|---|---|
| **plid** | ID of the pool to reset. |

### 4.59.4 Return Value

None.

### 4.59.5 Example

```
sc_poolid_t pl;

pl = sc_poolIdGet( "my_pool" );

sc_poolReset ( pl );
```

### 4.59.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPOOL_IN_USE ¦ SC_ERR_MODULE_FATAL<br>Pool is in use and no reset can be performed. | e0 = pool cb.<br>e1 = pool lock counter. |
| KERNEL_EILL_POOL_ID ¦ SC_ERR_MODULE_FATAL<br>Illegal pool ID. | e0 = pool ID. |

## 4.60 sc_procAtExit

### 4.60.1 Description

This system call is used to register a function to be called if a prioritized process is killed.

This allows to do some cleaning work if a process is killed. The sc_procAtExit system call is also used to register an error process from a module's init process.

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

**Kernels: V2 and V2INT**

### 4.60.2 Syntax

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

### 4.60.3 Parameter

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

### 4.60.4 Return Value

Returns the pointer to the old function or NULL if none was registerred.

### 4.60.5 Example

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

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

### 4.60.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Illegal function pointer. | e0 = function pointer. |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Can only be called within a prioritized process. | e0 = process type. |

## 4.61 sc_procAttrGet

### 4.61.1 Description

This system call is used to get specific attributes for a process.

**Kernels: V2 and V2INT**

### 4.61.2 Syntax

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

### 4.61.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to get the attribute. |
| SC_CURRENT_PID | Current running (caller) process. |

| **attribute** | Process attribute. See also sc_procAttr_t in proc.h. |
|---|---|
| SC_PROCATTR_NOP | Just checks if the process exists |
| SC_PROCATTR_STACKUSAGE | Stack usage in % |
| SC_PROCATTR_NALLOC | Number of messages allocated |
| SC_PROCATTR_NQUEUE | Number messages in the queue |
| SC_PROCATTR_NOBSERVE | Number of observations |
| SC_PROCATTR_TYPE | process type |
| SC_PROCATTR_STATE | Process state. It is a bit field which is zero or a combination of the following values:<br>SC_PROCATTR_STATE_WAIT_RX Process waits on message receive.<br>SC_PROCATTR_STATE_WAIT_TRG Process waits on trigger.<br>SC_PROCATTR_STATE_WAIT_ALLOC Process waits for message to be allocated.<br>SC_PROCATTR_STATE_WAIT_TMO Process waits with timeout.<br>SC_PROCATTR_STATE_READY process is ready. |
| SC_PROCATTR_IS_CONNECTOR | Process is a connector if value is TRUE. |
| SC_PROCATTR_STOPCNT | Stopcounter. Process is stopped if value is != 0 |
| SC_PROCATTR_NAME | Process name |

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

### 4.61.4 Return Value

0 if process is not found.

1 value successfully written.

### 4.61.5 Example

```
int msg_number

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

### 4.61.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal process attribute. | e0 = process attribute.<br>e1 = 1 |
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Pointer to value not valid (NULL). | |

## 4.62 sc_procCBChk

### 4.62.1 Description

System call to do diagnostic test for all elements of the process control block of specific process.

**Kernels: V2INT**

### 4.62.2 Syntax

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

### 4.62.3 Parameter

| **pid** | Process ID |
|---|---|
| <pid> | Process ID of the process to get the pcb. |
| SC_CURRENT_PID | Current running (caller) process. |
| **addr** | Address of the PCB. |
| **size** | Size of the PCB.. |

### 4.62.4 Return Value

== 0 if the pid is wrong.

== 1 if the process control block is correct and therefore not corrupted.

== -1 if the process control block is corrupted.

### 4.62.5 Example

TBD.

### 4.62.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Parameter pid not valid (== SC_ILLEGAL_PID). | e0 = pid |

## 4.63 sc_procCreate2

### 4.63.1 Description

This system call is used to request the kernel daemon to create a process. The standard kernel daemon (**sc_kerneld**) needs to be defined and started at system configuration.

The process will be created within the callers module.

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

**Kernels: V2 and V2INT**

### 4.63.2 Syntax

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

### 4.63.3 Parameter

| | |
|---|---|
| **pdb** | Pointer to the process descriptor block (pdb) which defines the process to create. |
| **state** | Process state after creation. |
| SC_PDB_STATE_RUN | The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes. |
| SC_PDB_STATE_STP | The process is stopped. Use the sc_procStart system call to start the process. |
| **plid** | Pool ID. |
| | Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated. |

### 4.63.4 Return Value

ID of the created process.

### 4.63.5 Process Descriptor Block pdb

The process descriptor block is a structure which is defining a process to be created.

It is included in the header file pcb.h.

It is devided into a common part valid for all process types and a process type specific part.

**For all CPUs except Aurix:**

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

**For Aurix CPUs:**

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


typedef struct sc_pdbcmn_s {
  PDB_COMMON(void);
} sc_pdbcommon_t;

typedef struct sc_pdbtim_s {
  PDB_COMMON(int);
  sc_ticks_t period;
  sc_ticks_t initial_delay;
} sc_pdbtim_t;

typedef struct sc_pdbint_s {
  PDB_COMMON(int);
  unsigned int ivector;
} sc_pdbint_t;
```

```
typedef struct sc_pdbprio_s {
  PDB_COMMON(void);

  sc_ticks_t slice;
  unsigned int prio;
} sc_pdbprio_t;

typedef union sc_pdb_u {
  sc_pdbcommon_t cmn;
  sc_pdbprio_t prio;
  sc_pdbint_t irq;
  sc_pdbtim_t tim;
} sc_pdb_t;
```

### 4.63.6 Structure Members Common for all Process Types

| **type** | Process type. |
|---|---|
| PCB_TYPE_PRI | Prioritized process. |
| PCB_TYPE_TIM | Timer process. |
| PCB_TYPE_INT | Interrupt process. |
| **name** | Pointer to process name. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore. |
| **entry** | Pointer to process function. |
| | This is the address where the created process will start execution. Processor typic alignment restriction apply. |
| **stacksize** | Process stack size. |
| | The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. |
| **pcb** | PCB pointer. |
| <pcb_ptr> | Pointer to a PCB |
| 0 | PCB will be allocated by the kernel. |
| **stack** | Stack address. |
| <stack_addr> | Pointer to a stack. Shall be taken from a pool or static memory within the module. |
| 0 | Stack will be allocated by the kernel. |

| | |
|---|---|
| **super** | Mode. |
| SC_KRN_FLAG_TRUE | Supervisor mode. |
| SC_KRN_FLAG_FALSE | User mode. |
| **fpu** | Floating point unit. |
| SC_KRN_FLAG_TRUE | Process does use FPU. |
| SC_KRN_FLAG_FALSE | Process does not use FPU. |
| **spare** | Not used, write 0. |
| **spare_b** | Not used, write 0 (for Aurix only). |
| **nCSA** | Number of CSAs (for Aurix only). |
| **csabtm** | Start of the CSA memory (for Aurix only). |
| | Must be in DSPR and aligned on 64 bytes. |

### 4.63.7 Additional Structure Members for Prioritized Processes

| | |
|---|---|
| **slice** | Time slice of the prioritized process. |
| **prio** | Process priority. |
| | The priority of the process which can be from 0 to 31. 0 is the highest priority. |

### 4.63.8 Additional Structure Members for Interrupt Processes

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

### 4.63.9 Additional Structure Members for Timer Processes

| | |
|---|---|
| **period** | Period of time between calls to the timer process in ticks. |
| **initdelay** | Initial delay in ticks before the first time call to the timer process. |

### 4.63.10 Example

```
static const sc_pdbprio_t pdb = {
  /* process-type    */ PCB_TYPE_PRI,
  /* process-name    */ "proc_A",
  /* function-name   */ proc_A,
  /* stacksize       */ 1024,
  /* pcb             */ 0,
  /* stack           */ 0,
  /* supervisor-flag */ SC_KRN_FLAG_TRUE,
  /* FPU-flag        */ SC_KRN_FLAG_FALSE,
  /* spare           */ 0,
  /* timeslice       */ 0,
  /* priority        */ 16
};

proc_A_pid = sc_procCreate2( (const sc_pdb_t *)&pdb, SC_PDB_STATE_RUN, 0x0) ;
```

### 4.63.11 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL | |
| Parameter **pdb** not valid (0 or SC_NIL). | |
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL | |
| mcb->freesize <= SIZEOF_PCB: | e0 = mcb->freesize |
| mcb->freesize <= stacksize element of pdb: | e0 = stacksize element of pdb<br>e1 = mcb->freesize |
| KERNEL_EILL_VECTOR ¦ SC_ERR_MODULE_FATAL | |
| Parameter vector (interrupt process) of pdb not valid (>SC_MAX_INT_VECTOR). | e0 = parameter: vector<br>e1 = pdb<br>e2 = 9 |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL | |
| Parameter slice (prioritized process) of pdb not valid: | e0 = parameter: slice |
| Parameter period (timer process) of pdb not valid: | e0 = parameter: period (timer process)<br>e1 = pdb<br>e2 = 9 |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL | |
| Parameter initial_dealy (timer process) of pdb not valid. | e0 = parameter: initial_delay<br>e1 = pdb<br>e2 = 10 |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Parameter type of pdb not valid. | e0 = parameter: type<br>e1 = pdb<br>e2 = 0 |
| KERNEL_ENO_MORE_PROC ¦ SC_ERR_MODULE_FATAL<br>Number of maximum processes reached. | e0 = No of process element of mcb<br>e1 = pdb<br>e2 = mcb |
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_MODULE_FATAL<br>Parameter name of pdb not valid. | e0 = pdb parameter: name<br>e1 = pdb<br>e2 = 1 |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Module cb is not valid (mcb == SC_NIL). | e0 = mid<br>e1 = 0<br>e2 = 1 |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Message which holds pcb or stack is not within current module. | e0 = Pointer to pcb or stack.<br>e1 = Pointer to mcb of current module.<br>e2 = Pointer to mcb of pcb/stack message buffer. |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_MODULE_FATAL<br>Module cb is not valid (mcb == SC_NIL). | e0 = pdb parameter: priority<br>e1 = pdb<br>e2 = 10 |
| KERNEL_EILL_STACKSIZE ¦ SC_ERR_MODULE_FATAL<br>Parameter stack of pdb not valid. | e0 = pdb parameter: stack<br>e1 = pdb<br>e2 = 3 |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter **pdb** not valid (!pdb \|\| pdb == SC_NIL). | e0 = Parameter **pdb**<br>e1 = 0<br>e2 = 0 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter **state** not valid. | e0 = Parameter **state**<br>e1 = 0<br>e2 = 1 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Illegal module ID.<br>mid low 24 bits != 0 or<br>midx >= SC_MAX_MODULES | e0 = mid<br>e1 = 0<br>e2 = 2 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter pcb of pdb not valid (==0). | e0 = pdb parameter: pcb<br>e1 = 0<br>e2 = 4 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter stack of pdb not valid (==0). | e0 = pdb parameter: stack<br>e1 = 0<br>e2 = 5 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter entry of pdb not valid. | e0 = pdb parameter: entry<br>e1 = pdb<br>e2 = 2 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter super not valid. | e0 = pdb parameter: super<br>e1 = pdb<br>e2 = 6 |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Parameter fpu not valid. | e0 = pdb parameter: fpu<br>e1 = pdb<br>e2 = 7 |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL | |
| Parameter spare not valid. | e0 = 0<br>e1 = pdb<br>e2 = 8 |
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL | |
| Not enough space for pcb or stack in selected message buffer. | e0 = Pointer to pcb or stack.<br>e1 = Pointer to mcb of current module.<br>e2 = Size of PCB or stacksize. |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_MODULE_FATAL | |
| Parameter vector (interrupt process) of pdb vector already defined. | e0 = pdb parameter: vector<br>e1 = pdb<br>e2 = 9 |

## 4.64 sc_procDaemonRegister

### 4.64.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.64.2 Syntax

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

### 4.64.3 Parameter

None.

### 4.64.4 Return Value

0 if the process daemon was successfully installed.

!=0 if the process daemon could not be installed.

### 4.64.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Calling process is not a prioritized process. | |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Calling process is not in system module. | |

## 4.65 sc_procDaemonUnregister

### 4.65.1 Description

This call is used by a process daemon to unregister.

**Kernels: V1, V2 and V2INT**

### 4.65.2 Syntax

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

### 4.65.3 Parameter

None

### 4.65.4 Return Value

0 if the process daemon was not a process daemon.

!=0 if the process daemon was successfully unregistered.

### 4.65.5 Errors

None

## 4.66 sc_procFlowSignatureGet

### 4.66.1 Description

This system call is used to get the caller's process program flow signature.

**Kernels: V2 and V2INT**

### 4.66.2 Syntax

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

### 4.66.3 Parameter

None.

### 4.66.4 Return Value

Signature value.

### 4.66.5 Example

---

### 4.66.6 Errors

None.

## 4.67 sc_procFlowSignatureInit

### 4.67.1 Description

This system call is used to initialize the caller's process program flow signature.

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

**Kernels: V2 and V2INT**

### 4.67.2 Syntax

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

### 4.67.3 Parameter

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

### 4.67.4 Return Value

None.

### 4.67.5 Example

---

### 4.67.6 Errors

None.

## 4.68 sc_procFlowSignatureUpdate

### 4.68.1 Description

System call to update the caller's program flow signature with the token given as parameter.

The result is returned.

**Kernels: V2 and V2INT**

### 4.68.2 Syntax

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

### 4.68.3 Parameter

| | |
|---|---|
| **token** | Token value. |
| | Token value to calculate new CRC16. |

### 4.68.4 Return Value

Signature value.

### 4.68.5 Example

---

### 4.68.6 Errors

None.

## 4.69 sc_procHookRegister

### 4.69.1 Description

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.

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

**Kernels: V1, V2 and V2INT**

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

### 4.69.2 Syntax

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

### 4.69.3 Parameter

| **type** | Type of process hook. |
|---|---|
| SC_SET_PROCCREATE_HOOK | Registers a process create hook. Every time a process is created, this hook will be called. |
| SC_SET_PROCKILL_HOOK | Registers a process kill hook. Every time a process is killed, this hook will be called. |
| SC_SET_PROCSWAP_HOOK | Registers a process swap hook. Every time a process swap is initiated by the kernel, this hook will be called. |

| **newhook** | Process hook function pointer. |
|---|---|
| <funcptr> | Function pointer to the process hook. |
| NULL | Removes and unregisters the process hook. |

### 4.69.4 Return Value

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

NULL if no process hook was registered.

### 4.69.5 Example

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

### 4.69.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_SYSTEM_FATAL<br>Illegal process hook type. | e0 = type |

## 4.70 sc_procIdGet

### 4.70.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.70.2 Syntax

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

### 4.70.3 Parameter

| **path** | Pointer to the path with the name of the process. |
|---|---|
| path::=process_name | Process resides within the caller's module. |
| path::='/'process_name | Process resides in the system module of the caller´s target. |
| path::='//'<system_name>'/'process_name | Process resides in the system module of an external target. |
| path::='/'<module_name>'/'process_name | Process resides in another than the system module of the caller's target. |
| path::='//'<system_name>'/'<module_name>'/'process_name | If the process resides in another than the system module of an external target. |

| **tmo** | Time to wait for a response in ticks. |
|---|---|
| | This parameter is not allowed if asynchronous timeout is disabled at system configuration (**SCONF**). |
| SC_NO_TMO | No timeout, returns immediately. |
| 0 < tmo < SC_TMO_MAX | Timeout value in system ticks. |
| SC_ENDLESS_TMO | Waits for ever. |

### 4.70.4 Return Value

Process ID of the found process if the process was found within the tmo time period or empty.

Current process ID (process ID of the caller) if parameter path is NULL.

SC_ILLEGAL_PID if process was not found within the tmo time period.

### 4.70.5 sc_procIdGet in Interrupt Processes

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

The reply message is defined as follows:

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

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

### 4.70.6 Example

```
sc_pid_t slave_pid;

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

### 4.70.7 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_PROCESS_FATAL<br>Illegal path. | e0 = pointer to path |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Process is PCB_TYPE_IDL. | e0 = process type |

## 4.71 sc_procIntCreate

### 4.71.1 Description

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. The interrupt process will be of type Sciopta. Interrupt processes of type Sciopta are handled by the kernel and may use (not blocking) system calls.

The process will be created within the callers module.

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

**Kernels: V1**

### 4.71.2 Syntax

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

### 4.71.3 Parameter

| | |
|---|---|
| **name** | Pointer to process name. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore. |
| **entry** | Pointer to process function. |
| | This is the address where the created process will start execution. |
| **stacksize** | Process stack size. |
| | The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize (160 bytes). |

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

### 4.71.4 Return Value

ID of the created process.

### 4.71.5 Example

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

### 4.71.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_MODULE_FATAL<br>Parameter name not valid. | e0 = Pointer to process name |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_MODULE_FATAL<br>Illegal interrupt vector. | e0 = Interrupt vector |

## 4.72 sc_procIrqRegister

### 4.72.1 Description

This system call is used to register an existing interrupt process for one more more other interrupt vectors.

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

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

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

It is a fatal error, if “vector” is already used.

**Kernels: V1, V2 and V2INT**

### 4.72.2 Syntax

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

### 4.72.3 Parameter

| | |
|---|---|
| **vector** | Interrupt vector. |
| | Must be in the range 0..SC_MAX_INT_VECTOR. |

### 4.72.4 Return Value

None.

### 4.72.5 Example

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

### 4.72.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE\|SC_ERR_PROCESS_FATAL<br>Not an interrupt process. | e0 = Process ID |
| KERNEL_EILL_VECTOR\|SC_ERR_PROCESS_FATAL<br>Illegal Interrupt Vector | e0 = vector |

## 4.73 sc_procIrqUnregister

### 4.73.1 Description

This system call is used to unregister previously registered interrupts.

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

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

**Kernels: V1, V2 and V2INT**

### 4.73.2 Syntax

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

### 4.73.3 Parameter

| | |
|---|---|
| **vector** | Interrupt vector. |
| | Must be in the range 0..SC_MAX_INT_VECTOR. |

### 4.73.4 Return Value

None.

### 4.73.5 Example

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

### 4.73.6 Errors

Error Code ¦ Error TypeExtra Value

KERNEL_EILL_PROCTYPE|SC_ERR_PROCESS_FATAL
Not an interrupt process. e0 = Process ID

KERNEL_EILL_VECTOR|SC_ERR_PROCESS_FATAL
Illegal Interrupt Vector e0 = vector

## 4.74 sc_procKill

### 4.74.1 Description

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.

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

If a cleaning-up is executed (depending on the **flag** parameter) all message buffers owned by the process will be returned to the message pool. If an observe is active on that process the observe messages will be sent to the observing processes. A significant time can elapse before a possible observe message is posted.

**Kernels: V1, V2 and V2INT**

### 4.74.2 Syntax

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

### 4.74.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to be killed. |
| SC_CURRENT_PID | Current running (caller) process. |

| **flag** | Process kill flag. |
|---|---|
| 0 | A cleaning up will be executed. |
| SC_PROCKILL_KILL | No cleaning up will be requested. |

### 4.74.4 Return Value

None

### 4.74.5 Example

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

### 4.74.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>pid == 0<br>pid == SC_ILLEGAL_PID<br>mid too large<br>process index too large. | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_WARNING<br>Process killed or pid not valid. | e0 = pid |

## 4.75 sc_procNameGet

### 4.75.1 Description

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

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

The user must free the message after use.

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

**Kernels: V1, V2 and V2INT**

### 4.75.2 Syntax

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

### 4.75.3 Parameter

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

### 4.75.4 Return Value

Message owned by the caller if the process exists.

If the return value is nonzero the returned message buffer is owned by the caller and the message is of type **sc_procNameGetMsgReply_t** and the message ID is **SC_PROCNAMEGETMSG_REPLY**. The message data contains the 0 terminated ASCII name string of the process. The message is defined in the **sciopta.msg** include file.

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

### 4.75.5 Example

```
sc_msg_t senderName;

senderName = sc_procNameGet( sender );
```

### 4.75.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>Illegal pid. | e0 = pid |

## 4.76 sc_procObserve

### 4.76.1 Description

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 supervising process.

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

**Kernels: V1, V2 and V2INT**

### 4.76.2 Syntax

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

### 4.76.3 Parameter

| | |
|---|---|
| **msgptr** | Pointer to the observe message pointer. |
| | Pointer to the message which will be returned if the supervised process disappears. The message must be of the following type:<br>`struct err_msg {`<br>`  sc_msgid_t  id;`<br>`  sc_errcode_t  error;`<br>`  /* user defined data */`<br>`};` |
| **pid** | Process ID. |
| <pid> | Process ID of the process where the name is requested. |

### 4.76.4 Return Value

None.

### 4.76.5 Example

```
struct dead_s {
  sc_msgid_t id;
  sc_errcode_t errcode;
};

union sc_msg{
  sc_msgid_t id;
  struct dead_s dead;
};

sc_msg_t msg;

msg = sc_msgAlloc( sizeof(struct dead_s), 0xdead, 0, SC_FATAL_IF_TMO );

sc_procObserve( &msg, slave_pid );
```

### 4.76.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>Illegal pid. | e0 = pid |

## 4.77 sc_procPathCheck

### 4.77.1 Description

This call is used to check if the construction of a path is correct. It checks the lengths of the system, module and process names and the number of slashes and if it contains only valid character (A-Z,a-z,0-9 and underscore)..

**Kernels: V1, V2 and V2INT**

### 4.77.2 Syntax

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

### 4.77.3 Parameter

| **path** | Pointer to the path with the name of the process. |
|---|---|
| path::=process_name | Process resides within the caller's module. |
| path::='/'process_name | Process resides in the system module of the caller´s target. |
| path::='//'<system_name>'/'process_name | Process resides in the system module of an external target. |
| path::='/'<module_name>'/'process_name | Process resides in another than the system module of the caller's target. |
| path::='//'<system_name>'/'<module_name>'/'process_name | If the process resides in another than the system module of an external target. |

### 4.77.4 Return Value

!= 0 if the path is correct.

== 0 if the path is wrong.

### 4.77.5 Example

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

### 4.77.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_SYSTEM_FATAL<br>Illegal path (pointer to path == 0). | e0 = pointer to path name. |

## 4.78 sc_procPathGet

### 4.78.1 Description

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, he system loops at the error label.

**Kernels: V1, V2 and V2INT**

### 4.78.2 Syntax

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

### 4.78.3 Parameter

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

### 4.78.4 Return Value

Message owned by the caller if the process exists.

If the return value is nonzero the returned message buffer is owned by the caller and the message is of type **sc_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;
```

### 4.78.5 Example

```
sc_msg_t msg;

msg = sc_procPathGet( SC_CURRENT_PID, 1 );
if ( strstr(msg->path.path,"node1") ){
   remote = "//node2/node2/echo";
} else {
   remote = "//node1/node1/echo";
}
```

### 4.78.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Illegal pid. | e0 = pid |

## 4.79 sc_procPpidGet

### 4.79.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.79.2 Syntax

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

### 4.79.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to be killed. |
| SC_CURRENT_PID | Current running (caller) process. |

### 4.79.4 Return Value

Process ID of the parent process if the parent process exists

Process ID of the parent process of the caller if parameter pid was SC_CURRENT_PID.

SC_ILLEGAL_PID if the parent process does no longer exist.

### 4.79.5 Example

```
typedef struct key_s {
  uint8_t scan;
  uint8_t cntrl;
} sckey_t;

#define KEYB_MSG 0x30000001
typedef struct keyb_msg_s{
  sc_msgid_t id;
  sckey_t data;
} keyb_msg_t;
sc_msg_t msg;

sc_pid_t ttyd_pid = sc_procPpidGet( SC_CURRENT_PID );

msg = sc_msgAlloc( sizeof(keyb_msg_t), KEYB_MSG, 0, SC_ENDLESS_TMO );
if ( msg ){
  msg->keyb.data.scan = key;
  msg->keyb.data.cntrl = control_keys;
  (&msg,ttyd_pid,0);
}
```

### 4.79.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL<br>Illegal pid. | e0 = pid |

## 4.80 sc_procPrioCreate

### 4.80.1 Description

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.

The process will be created within the callers module.

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

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

Kernels: V1

### 4.80.2 Syntax

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

### 4.80.3 Parameter

| | |
|---|---|
| **name** | Pointer to process name. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Allowed characters are A-Z, a-z, 0-9 and underscore. |
| **entry** | Pointer to process function. |
| | This is the address where the created process will start execution. |
| **stacksize** | Process stack size. |
| | The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize (160 bytes). |
| **slice** | Time slice of the prioritized process. |
| **prio** | Process priority. |
| | The priority of the process which can be from 0 to 31. 0 is the highest priority. |

| **state** | Process state after creation. |
|---|---|
| SC_PDB_STATE_RUN | The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes. |
| SC_PDB_STATE_STP | The process is stopped. Use the sc_procStart system call to start the process. |
| **plid** | Pool ID. |
| | Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated. |

### 4.80.4 Return Value

ID of the created process.

### 4.80.5 Example

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

### 4.80.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_MODULE_FATAL<br>Parameter name not valid. | e0 = Pointer to process name |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_MODULE_FATAL<br>Illegal priority (>=31) | e0 = Requested priority |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL<br>Illegal slice value | e0 = Slice |

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_STACKSIZE ¦ SC_ERR_MODULE_FATAL<br>Stack not valid. | e0 = Requested stacksize |
| KERNEL_EILL_MODULE ¦ SC_ERR_MODULE_FATAL<br>Module cb is not valid. | e0 = Module ID |
| KERNEL_ENO_MORE_PROC ¦ SC_ERR_MODULE_FATAL<br>Number of maximum processes reached. | e0 = No of processes |
| KERNEL_EOUT_OF_MEMORY ¦ SC_ERR_MODULE_FATAL<br>Size does not fir into module memory. | |

## 4.81 sc_procPrioGet

### 4.81.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.81.2 Syntax

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

### 4.81.3 Parameter

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

### 4.81.4 Return Value

Priority of any given process if parameter pid was any process.

Priority of the callers process if parameter pid was SC_CURRENT_PID.

32 if there was a warning of an invalid CONNECTOR process.

### 4.81.5 Example

```
// Create process "proc_A" with lower priority than caller

sc_pid_t proc_A_pid;

sc_prio_t prio = sc_procPrioGet( SC_CURRENT_PID ) + 1;

static const sc_pdbprio_t pdb = {
  /* process-type    */ PCB_TYPE_STATIC_PRI,
  /* process-name    */ "proc_A",
  /* function-name   */ proc_A,
  /* stacksize       */ 1024,
  /* pcb             */ 0,
  /* stack           */ 0,
  /* supervisor-flag */ SC_KRN_FLAG_TRUE,
  /* FPU-flag        */ SC_KRN_FLAG_FALSE,
  /* spare           */ 0,
  /* time-slice      */ 0,
  /* priority        */ prio
};

proc_A_pid = sc_ProcCreate2( (const sc_pdb_t *)&pdb, SC_PDB_STATE_RUN, 0x0);
```

### 4.81.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pid. | e0 = pid |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = pid<br>e1 = process type |

## 4.82 sc_procPrioSet

### 4.82.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.82.2 Syntax

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

### 4.82.3 Parameter

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

### 4.82.4 Return Value

None.

### 4.82.5 Example

```
// Switch caller to lowest-priority

sc_procPrioSet( 31 );
```

### 4.82.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = process type |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_PROCESS_FATAL | |
| Illegal priority. Priority == 32. | e0 = process type<br>e1 = module priority |
| KERNEL_EILL_PRIORITY ¦ SC_ERR_PROCESS_FATAL | |
| Illegal priority.<br>Priority > 32.<br>priority > max_prio. | e0 = process type<br>e1 = -1 |

## 4.83 sc_procSchedLock

### 4.83.1 Description

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 locked.

**Kernels: V1, V2 and V2INT**

### 4.83.2 Syntax

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

### 4.83.3 Parameter

None

### 4.83.4 Return Value

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

### 4.83.5 Example

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

### 4.83.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | e0 = process type |

## 4.84 sc_procSchedUnlock

### 4.84.1 Description

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 locked scheduler.

**Kernels: V1, V2 and V2INT**

### 4.84.2 Syntax

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

### 4.84.3 Parameter

None.

### 4.84.4 Return Value

None

### 4.84.5 Example

```
// count instances

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

### 4.84.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | e0 = process type |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Interrupts are locked. | |
| KERNEL_EUNLOCK_WO_LOCK ¦ SC_ERR_MODULE_FATAL<br>Lockcounter == 0. | |

## 4.85 sc_procSliceGet

### 4.85.1 Description

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

The time slice is the period of time between calls to the timer process in ticks or the the slice of round-robin scheduled prioritized processes on the same priority.

**Kernels: V1, V2 and V2INT**

### 4.85.2 Syntax

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

### 4.85.3 Parameter

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

### 4.85.4 Return Value

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

### 4.85.5 Example

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

### 4.85.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Process/Module disappeared | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pid. | e0 = pid |

## 4.86 sc_procSliceSet

### 4.86.1 Description

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

The modified time slice will become active after the current time slice expired or if the timer gets started. It can only be activated after the old time slice has elapsed.

**Kernels: V1, V2 and V2INT**

### 4.86.2 Syntax

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

### 4.86.3 Parameter

| **pid** | Process ID. |
|---|---|
| <pid> | Process ID of the process to set the time slice. |
| SC_CURRENT_PID | Current running (caller) process. |
| **slice** | New period of time between calls to the timer process in ticks. |
| !=0 | 0 is only allowed for priortized processes and disables the time-slice. |

### 4.86.4

Return Value

None.

### 4.86.5 Example

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

### 4.86.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Process/Module disappeared | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Illegal pid. | e0 = pid |

## 4.87 sc_procStart

### 4.87.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.87.2 Syntax

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

### 4.87.3 Parameter

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

### 4.87.4 Return Value

None.

### 4.87.5 Example

```
sc_procStart( proc_A_pid );
```

### 4.87.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized or timer process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller.<br>Process is init process. | e0 = pid |
| KERNEL_ESTART_NOT_STOPPED ¦ SC_ERR_MODULE_FATAL | |
| Stop counter already 0 | e0 = pid |

## 4.88 sc_procStop

### 4.88.1 Description

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.

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

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

**Kernels: V1, V2 and V2INT**

### 4.88.2 Syntax

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

### 4.88.3 Parameter

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

### 4.88.4 Return Value

None.

### 4.88.5 Example

```
sc_procStop( SC_CURRENT_PID );
```

### 4.88.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROC ¦ SC_ERR_MODULE_FATAL | |
| Caller is not a prioritized or timer process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller.<br>Process is init process. | e0 = pid |
| KERNEL_EILL_VALUE ¦ SC_ERR_MODULE_FATAL | |
| Stop counter already 0 | |

## 4.89 sc_procTimCreate

### 4.89.1 Description

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.

The process will be created within the callers module.

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

**Kernels: V1**

### 4.89.2 Syntax

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

### 4.89.3 Parameter

| | |
|---|---|
| **name** | Pointer to process name. |
| | The name is represented by a ASCII character string terminated be 0. The string can have up to 31 characters. Recommended characters are A-Z, a-z, 0-9 and underscore. |
| **entry** | Pointer to process function. |
| | This is the address where the created process will start execution. |
| **stacksize** | Process stack size. |
| | The kernel will use one of the fixed message buffer sizes from the message pool which is big enough to include the stack. Add 128 bytes for the process control block (pcb) and 32 bytes for the process name to your process stack to calculate the minimum stacksize (160 bytes). |
| **period** | Time period. |
| | Period of time between calls to the timer process in ticks. |

| | |
|---|---|
| **initdelay** | Initial time delay. |
| | Initial delay in ticks before the first time call to the timer process. |
| **state** | Process state after creation. |
| SC_PDB_STATE_RUN | The process will be on READY state. It is ready to run and will be swapped-in if it has the highest priority of all READY processes. |
| SC_PDB_STATE_STP | The process is stopped. Use the sc_procStart system call to start the process. |
| **plid** | Pool ID. |
| | Pool ID from where the message buffer (which holds the stack and the pcb) will be allocated. |

### 4.89.4 Return Value

ID of the created process.

### 4.89.5 Example

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

### 4.89.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENO_KERNELD ¦ SC_ERR_PROCESS_FATAL<br>There is no kernel daemon defined in the system. | |
| KERNEL_EILL_PROC_NAME ¦ SC_ERR_MODULE_FATAL<br>Parameter name not valid. | e0 = Pointer to process name |
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL<br>Illegal slice valueSlice | e0 = Slice |

## 4.90 sc_procUnobserve

### 4.90.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.90.2 Syntax

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

### 4.90.3 Parameter

| **pid** | Supervised process ID. |
|---|---|
| <pid> | Process ID of the process which is supervised. |
| **observer** | Observer process ID. |
| <pid> | Process ID of the observer process. |
| SC_CURRENT_PID | Current process is observer. |

### 4.90.4 Return Value

None.

### 4.90.5 Example

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

### 4.90.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller. | e0 = pid |
| Process is init process. | |

## 4.91 sc_procVarDel

### 4.91.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.91.2 Syntax

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

### 4.91.3 Parameter

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

### 4.91.4 Return Value

0 if the system call fails and the process variable could not be removed.

!=0 if the process variable was successfully removed.

### 4.91.5 Example

---

### 4.91.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No process variable set | |

## 4.92 sc_procVarGet

### 4.92.1 Description

This system call is used to read a process variable.

**Kernels: V1, V2 and V2INT**

### 4.92.2 Syntax

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

### 4.92.3 Parameter

| | |
|---|---|
| **tag** | Process variable tag. |
| | User defined tag of the process variable which was set by the sc_procVarSet call. |
| **value** | Process variable. |
| | Pointer to the variable where the process variable will be stored. |

### 4.92.4 Return Value

0 if the system call fails and the process variable could not be found and read.

!=0 if the process variable was successfully read.

### 4.92.5 Example

---

### 4.92.6 Errors

| Error Code | Error Type | Extra Value |
|---|---|---|
| KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL<br>No procVar set or value == NULL | |

## 4.93 sc_procVarInit

### 4.93.1 Description

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

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

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

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

```
size = sizeof(sc_varpool_t)+sizeof(sc_local_t)*n.
```

**Kernels: V1, V2 and V2INT**

### 4.93.2 Syntax

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

### 4.93.3 Parameter

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

| **n** | Maximum number of process variables. |
|---|---|

### 4.93.4 Return Value

None.

### 4.93.5 Example

---

### 4.93.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No procVar set or value == NULL | |
| KERNEL_ENOT_OWNER ¦ SC_ERR_PROCESS_FATAL<br>Process does not own the buffer | e0 = owner |
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Size too small. | e0 = size |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_PROCESS_FATAL<br>Process variable already set. | e0 = pointer to message buffer |

## 4.94 sc_procVarRm

### 4.94.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.94.2 Syntax

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

### 4.94.3 Parameter

None

### 4.94.4 Return Value

Pointer to the message buffer holding the process variables.

### 4.94.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>No procVar set or value == NULL | |

## 4.95 sc_procVarSet

### 4.95.1 Description

This system call is used to set or modify a process variable.

**Kernels: V1, V2 and V2INT**

### 4.95.2 Syntax

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

### 4.95.3 Parameter

| | |
|---|---|
| **tag** | Process variable tag. |
| | User defined tag of the process variable.<br>Valid values: All except 0. |
| **value** | Value of the process variable. |

### 4.95.4 Return Value

0 if the system call fails and the process variable could not be defined or modified.

!=0 if the process variable was successfully defined or modified.

### 4.95.5 Example

---

### 4.95.6 Errors

| Error Code | Error Type | Extra Value |
|---|---|---|
| KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL | No procVar set |

## 4.96 sc_procVectorGet

### 4.96.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.96.2 Syntax

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

### 4.96.3 Parameter

| | |
|---|---|
| **pid** | Process ID. |
| | Process ID of the interrupt process. |

### 4.96.4 Return Value

Interrupt vector of the interrupt process.

### 4.96.5 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL | |
| Caller is not an interrupt process. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING | |
| Illegal pcb | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_FATAL | |
| Process is caller.<br>Process is init process. | e0 = pid |

## 4.97 sc_procWakeupEnable

### 4.97.1 Description

This system call is used to enable the wakeup of a timer or interrupt process.

**Kernels: V1, V2 and V2INT**

**Please Note:**

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

### 4.97.2 Syntax

```
void sc_procWakeupEnable( void )
```

### 4.97.3 Parameter

None.

### 4.97.4 Return Value

None.

### 4.97.5 Example

---

### 4.97.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Caller is not an interrupt or timer process. | e0 = process type |

## 4.98 sc_procWakeupDisable

### 4.98.1 Description

This system call is used to disable the wakeup of a timer or interrupt process.

**Kernels: V1, V2 and V2INT**

### 4.98.2 Syntax

```
void sc_procWakeupDisable( void )
```

### 4.98.3 Parameter

None.

### 4.98.4 Return Value

None.

### 4.98.5 Example

---

### 4.98.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_MODULE_FATAL<br>Caller is not an interrupt or timer process. | e0 = process type |

## 4.99 sc_procYield

### 4.99.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.99.2 Syntax

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

### 4.99.3 Parameter

None.

### 4.99.4 Return Value

None.

### 4.99.5 Example

```
sc_procYield ();
```

### 4.99.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Scheduling not possible as it is locked. | |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | e0 = process type |

## 4.100 sc_safe_charGet

### 4.100.1 Description

System call to get safe data of specific char types. The data is stored once in normal and in inverted format.

**Kernels: V2 and V2INT**

### 4.100.2 Syntax

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

### 4.100.3 Parameter

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

### 4.100.4 Return Value

None.

### 4.100.5 Example

---

### 4.100.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter si not valid (== 0). | |

## 4.101 sc_safe_charSet

### 4.101.1 Description

System call to set safe data of specific char types at a given address in memory. The data is stored once in normal and in inverted format.

**Kernels: V2 and V2INT**

### 4.101.2 Syntax

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

### 4.101.3 Parameter

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

### 4.101.4 Return Value

None.

### 4.101.5 Example

---

### 4.101.6 Errors

| Error Code | Error Type | Extra Value |
|---|---|---|
| KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL<br>Parameter si not valid (== 0). | |

## 4.102 sc_safe_<type>Get

### 4.102.1 Description

System call to get safe data of specific types. The data is stored once in normal and in inverted format.

**Kernels: V2 and V2INT**

### 4.102.2 Syntax

```
int sc_safe_intGet( sc_safe_int_t *si );
unsigned int sc_safe_intGet( sc_safe_uint_t *si );
long sc_safe_intGet( sc_safe_long_t *si );
unsigned long sc_safe_intGet( sc_safe_ulong_t *si );
__s32 sc_safe_intGet( sc_safe_s32_t *si );
uint32_t sc_safe_intGet( sc_safe_u32_t *si );
sc_pool_cb_t *sc_safe_poolcb_ptrGet( sc_safe_poolcb_ptr_t *si );
sc_pcb_t *sc_safe_pcbptrGet( sc_safe_pcbptr_t *si );
sc_module_cb_t *sc_safe_mcbptrGet( sc_safe_mcbptr_t *si );
sc_modulesize_t sc_safe_modulesizeGet( sc_safe_modulesize_t*si );
sc_ticks_t sc_safe_ticksGet( sc_safe_ticks_t *si );
sc_time_t sc_safe_timeGet( sc_safe_time_t *si );
sc_pid_t sc_safe_pidGet( sc_safe_pid_t *si );
sc_mid_t sc_safe_midGet( sc_safe_mid_t *si );
sc_errcode_t sc_safe_errcodeGet( sc_safe_errcode_t *si );
void *sc_safe_voidptrGet( sc_safe_voidptr_t *si );
sc_triggerval_t sc_safe_triggervalGet( sc_safe_triggerval_t *si );
sc_plsize_t sc_safe_plsizeGet( sc_safe_plsize_t *si );
sc_poolid_t sc_safe_poolidGet( sc_safe_poolid_t *si );
sc_bufsize_t sc_safe_bufsizeGet( sc_safe_bufsize_t *si );
sc_prio_t sc_safe_prioGet( sc_safe_prio_t *si );
```

### 4.102.3 Parameter

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

### 4.102.4 Return Value

None.

### 4.102.5 Example

---

### 4.102.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter si not valid (== 0). | |

## 4.103 sc_safe_<type>Set

### 4.103.1 Description

System call to set safe data of specific types at a given address in memory. The data is stored once in normal and in inverted format.

**Kernels: V2 and V2INT**

### 4.103.2 Syntax

```
void sc_safe_intSet( sc_safe_int_t *si, int v );
void sc_safe_uintSet( sc_safe_uint_t *si, unsigned int v );
void sc_safe_longSet( sc_safe_long_t *si, long v );
void sc_safe_ulongSet( sc_safe_ulong_t *si, unsigned long v );
void sc_safe_s32Set( sc_safe_s32_t *si, __s32 v );
void sc_safe_u32Set( sc_safe_u32_t *si, uint32_t v );
void sc_safe_poolcb_ptrSet( sc_safe_poolcb_ptr_t *si, sc_pool_cb_t *v );
void sc_safe_pcbptrSet( sc_safe_pcbptr_t *si, sc_pcb_t *v );
void sc_safe_mcbptrSet( sc_safe_mcbptr_t *si, sc_module_cb_t *v );
void sc_safe_modulesizeSet( sc_safe_modulesize_t *si, sc_modulesize_t v );
void sc_safe_ticksSet( sc_safe_ticks_t *si, sc_ticks_t v );
void sc_safe_timeSet( sc_safe_time_t *si, sc_time_t v );
void sc_safe_pidSet( sc_safe_pid_t *si, sc_pid_t v );
void sc_safe_midSet( sc_safe_mid_t *si, sc_mid_t v );
void sc_safe_errcodeSet( sc_safe_errcode_t *si, sc_errcode_t v );
void sc_safe_voidptrSet( sc_safe_voidptr_t *si, void *v );
void sc_safe_triggervalSet( sc_safe_triggerval_t *si, sc_triggerval_t v );
void sc_safe_plsizeSet( sc_safe_plsize_t *si, sc_plsize_t v );
void sc_safe_poolidSet( sc_safe_poolid_t *si, sc_poolid_t v );
void sc_safe_bufsizeSet( sc_safe_bufsize_t *si, sc_bufsize_t v );
void sc_safe_prioSet( sc_safe_prio_t *si, sc_prio_t v );
```

### 4.103.3 Parameter

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

### 4.103.4 Return Value

None.

### 4.103.5 Example

---

### 4.103.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter si not valid (== 0). | |

## 4.104 sc_safe_shortGet

### 4.104.1 Description

System call to get safe data of specific short types. The data is stored once in normal and in inverted format.

**Kernels: V2 and V2INT**

### 4.104.2 Syntax

```
short sc_safe_shortGet( sc_safe_short_t *si )
unsigned short sc_safe_ushortGet( sc_safe_ushort_t *si )
__s16 sc_safe_s16Get( sc_safe_s16_t *si )
uint16_t sc_safe_u16Get( sc_safe_u16_t *si )
```

### 4.104.3 Parameter

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

### 4.104.4 Return Value

Retrieved safe data of specific short types.

### 4.104.5 Example

---

### 4.104.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Parameter si not valid (== 0). | |

## 4.105 sc_safe_shortSet

### 4.105.1 Description

System call to set safe data of specific short types at a given address in memory. The data is stored once in normal and in inverted format.

**Kernels: V2 and V2INT**

### 4.105.2 Syntax

```
void sc_safe_shortSet( sc_safe_short_t *si, short v )
void sc_safe_ushortSet( sc_safe_ushort_t *si, unsigned short v )
void sc_safe_s16Set( sc_safe_s16_t *si, __s16 v )
void sc_safe_u16Set( sc_safe_u16_t *si, uint16_t v )
```

### 4.105.3 Parameter

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

### 4.105.4 Return Value

None.

### 4.105.5 Example

---

### 4.105.6 Errors

| Error Code | Error Type | Extra Value |
|---|---|---|
| KERNEL_ENIL_PTR | SC_ERR_PROCESS_FATAL<br>Parameter si not valid (== 0). | |

## 4.106 sc_sleep

### 4.106.1 Description

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 timeout has elapsed the process will become ready again and will be swapped in if it has the highest priority of all ready processes.

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

**Kernels: V1, V2 and V2INT**

### 4.106.2 Syntax

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

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

### 4.106.3 Parameter

| | |
|---|---|
| **tmo** | Timeout. |
| | Number of system ticks to wait. |

### 4.106.4 Return Value

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

### 4.106.5 Example

```
void resetPHY(){
    // Setup some I/O pins
    sc_sleep( 2 );
    // Setup some other I/O pins
    sc_sleep( 2 );
    // Setup last I/O pins
    sc_sleep( 2 );
}
```

### 4.106.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Scheduler is locked. | |
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_MODULE_FATAL<br>Caller is not a prioritized process. | |
| KERNEL_EILL_SLICE ¦ SC_ERR_MODULE_FATAL<br>Illegal timeout value. | e0 = tmo |

## 4.107 sc_tick

### 4.107.1 Description

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.

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

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

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

**Kernels: V1, V2 and V2INT**

### 4.107.2 Syntax

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

### 4.107.3 Parameter

None.

### 4.107.4 Return Value

None.

### 4.107.5 Example

```
SC_INT_PROCESS( sysTick, src )
{
   if ( src == SC_PROC_WAKEUP_HARDWARE ){
      sc_tick();
         /* Handle timer irq */
   }
}
```

### 4.107.6 Errors

None

## 4.108 sc_tickActivationGet

### 4.108.1 Description

This functions returns the tick time of last activation of the calling process.

**Kernels: V2 and V2INT**

### 4.108.2 Syntax

```
sc_time_t sc_tickActivationGet(void)
```

### 4.108.3 Parameter

None

### 4.108.4 Return Value

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

### 4.108.5 Example

---

### 4.108.6 Errors

None

## 4.109 sc_tickGet

### 4.109.1 Description

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

**Kernels: V1, V2 and V2INT**

### 4.109.2 Syntax

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

### 4.109.3 Parameter

None

### 4.109.4 Return Value

Current value of the tick timer.

### 4.109.5 Example

```
t = sc_tickGet();
for(i = 0; i < 100; ++i ){
   cache_flush_range((char *)0x3000000,0x8000);
   memcpy32B((char *)0x2000000,(char *)0x3000000,0x100000);
}

t = sc_tickGet()-t;
kprintf(0,"Copy in 100MB in %d ms\n",sc_tickTick2Ms(t));
```

### 4.109.6 Errors

None

## 4.110 sc_tickLength

### 4.110.1 Description

This system call is used to set or get the current system tick length in microseconds.

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

**Kernels: V1, V2 and V2INT**

### 4.110.2 Syntax

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

### 4.110.3 Parameter

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

### 4.110.4 Return Value

Tick length in microseconds.

### 4.110.5 Example

```
kprintf(0,"Setting up system-timer ...");
pit_init(200, 0);// 200Hz == 5ms

sc_tickLength( 4999 );

pic_irqEnable( PIC_SRC_PIT0 );
kprintf( 0, "done\n" );
```

### 4.110.6 Errors

None

## 4.111 sc_tickMs2Tick

### 4.111.1 Description

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

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

**Kernels: V1, V2 and V2INT**

### 4.111.2 Syntax

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

### 4.111.3 Parameter

| | |
|---|---|
| **ms** | Time in milliseconds. |

### 4.111.4 Return Value

Time in system ticks.

### 4.111.5 Example

```
int tmo = 1000;

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

### 4.111.6 Errors

None

## 4.112 sc_tickTick2Ms

### 4.112.1 Description

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

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

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

**Kernels: V1, V2 and V2INT**

### 4.112.2 Syntax

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

### 4.112.3 Parameter

| | |
|---|---|
| **t** | Time in system ticks. |

### 4.112.4 Return Value

Time in milliseconds.

### 4.112.5 Example

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

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

### 4.112.6 Errors

None

## 4.113 sc_tmoAdd

### 4.113.1 Description

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

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

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

The registered timeout can be cancelled by the sc_tmoRm call before the timeout has expired. This system call returns the timeout ID which could be used later to cancel the timeout.

**Kernels: V1, V2 and V2INT**

### 4.113.2 Syntax

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

### 4.113.3 Parameter

| | |
|---|---|
| **tmo** | Timeout. |
| | Number of system tick after which the message will be sent back by the kernel. |
| **msgptr** | Pointer to the timeout message pointer. |
| | Pointer to the message pointer of the message which will be sent back by the kernel after the elapsed time. |

### 4.113.4 Return Value

Timeout ID.

### 4.113.5 Example

```
sc_tmoid_t tmoid;

msg = sc_msgAlloc( sizeof(ctrl_poll_t), TCS_CTRL_POLL, 0, SC_FATAL_IF_TMO );

tmoid = sc_tmoAdd( (sc_ticks_t)sc_tickMs2Tick( 1000 ), &msg );
```

### 4.113.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EPROC_NOT_PRIO ¦ SC_ERR_PROCESS_FATAL<br>Caller is not a prioritized process. | e0 = process type |
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>Illegal timeout value. | e0 = tmo |
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_EALREADY_DEFINED ¦ SC_ERR_PROCESS_FATAL<br>Message is already a timeout message. | e0 = Pointer to the message |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = Owner<br>e1 = Pointer to message |

## 4.114 sc_tmoRm

### 4.114.1 Description

This system call is used to remove a timeout before it is expired.

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

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

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

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

**Kernels: V1, V2 and V2INT**

### 4.114.2 Syntax

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

### 4.114.3 Parameter

| | |
|---|---|
| **id** | Timeout ID. |
| | Pointer to timeout ID which was given when the timeout was registered by the sc_tmoAdd call. |

### 4.114.4 Return Value

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

### 4.114.5 Example

```
sc_msg_t tmomsg;

tmomsg = sc_tmoRm( &tmoid );

sc_msgFree( &tmomsg );
```

### 4.114.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Timeout expired, but not received (not in the queue) | |
| KERNEL_EILL_PARAMETER ¦ SC_ERR_PROCESS_FATAL<br>Timeout ID is already cleared. | e0 = Pointer to timeout ID |
| KERNEL_ENIL_PTR ¦ SC_ERR_PROCESS_FATAL<br>Either pointer to message or pointer to message pointer are zero. | |
| KERNEL_ENOT_OWNER ¦ SC_ERR_MODULE_FATAL<br>Process does not own the message. | e0 = pid of the owner |

## 4.115 sc_trigger

### 4.115.1 Description

This system call is used to activate a process trigger.

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

**Kernels: V1, V2 and V2INT**

### 4.115.2 Syntax

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

### 4.115.3 Parameter

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

### 4.115.4 Return Value

None.

### 4.115.5 Example

```
sc_pid_t slave_pid;

slave_pid = sc_procIdGet( "slave", SC_NO_TMO );

sc_trigger( slave_pid );
```

### 4.115.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Illegal process type. | e0 = pid<br>e1 = process type |
| KERNEL_EILL_PID ¦ SC_ERR_MODULE_WARNING<br>Process disappeared | e0 = pid |
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>Process is an init or external process | e0 = pid |

## 4.116 sc_triggerValueGet

### 4.116.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.116.2 Syntax

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

### 4.116.3 Parameter

| | |
|---|---|
| **pid** | Process ID. |
| | ID of the process which trigger is returned. |

### 4.116.4 Return Value

Trigger value.

INT_MAX if no valid process.

### 4.116.5 Example

```
sc_pid_t slave_pid;
sc_triggerval_t slavetrig;

slave_pid = sc_procIdGet( "slave", SC_NO_TMO );

slave_trig = sc_triggerValueGet( slave_pid );
```

### 4.116.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PID ¦ SC_ERR_PROCESS_FATAL<br>Process is an init or external process | e0 = pid |

## 4.117 sc_triggerValueSet

### 4.117.1 Description

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.

**Kernels: V1, V2 and V2INT**

### 4.117.2 Syntax

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

### 4.117.3 Parameter

| | |
|---|---|
| **value** | Trigger value. |
| | The new trigger value to be stored. |

### 4.117.4 Return Value

None.

### 4.117.5 Example

```
sc_triggerValueSet( 1 );

sc_triggerWait( 1, SC_ENDLESS_TMO );
```

### 4.117.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL | |
| Illegal trigger value. | e0 = Trigger value |

## 4.118 sc_triggerWait

### 4.118.1 Description

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.

The caller can also specify a timeout value **tmo**. The caller will not wait longer than the specified time for the trigger. If the timeout expires the process will be ready in again and the trigger value will be incremented by the amount it has been decrement before.

**Kernels V2 only:** The activation time is saved for sc_triggerWait in prioritized processes. The activation time is the absolute time (tick counter) value when the process became ready.

**Kernels: V1, V2 and V2INT**

### 4.118.2 Syntax

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

### 4.118.3 Parameter

| **dec** | Decrease value. |
|---|---|
| | The number to decrease the process trigger value. |

| **tmo** | Timeout. |
|---|---|
| SC_ENDLESS_TMO | Timeout is not used. Blocks and waits endless until trigger. |
| SC_NO_TMO | Generates a system error. |
| SC_FATAL_IF_TMO | Generates a system error. |
| 0 < tmo =< SC_TMO_MAX | Timeout value in system ticks. Waiting on trigger with timeout. Blocks and waits the specified number of ticks for trigger. |

### 4.118.4 Return Value

SC_TRIGGER_TRIGGERED if the trigger occurred.

SC_TRIGGER_NO_WAIT if the process did not swap out

SC_TRIGGER_TMO if a timeout occurred

SC_TRIGGER_WAKEUP if the kernel will wakeup a timer or interrupt process.

### 4.118.5 Example

```
sc_triggerValueSet( 1 );

sc_triggerWait( 1, SC_ENDLESS_TMO );
```

### 4.118.6 Errors

| Error Code ¦ Error Type | Extra Value |
|---|---|
| KERNEL_EILL_PROCTYPE ¦ SC_ERR_PROCESS_FATAL<br>Illegal process type. | e0 = Process type |
| KERNEL_ELOCKED ¦ SC_ERR_MODULE_FATAL<br>Interrupts and/or scheduler are/is locked. | e0 = Lock counter value or<br>-1 if interrupt are locked. |
| KERNEL_EILL_VALUE ¦ SC_ERR_PROCESS_FATAL<br>Illegal trigger decrement value (<=0). | e0 = Decrement value |
| KERNEL_EILL_SLICE ¦ SC_ERR_PROCESS_FATAL<br>tmo value not valid. | e0 = tmo value |

## 4.119 sciopta_end

### 4.119.1 Desription

This function ends a SCIOPTA Simulator application.

The control is returned to the Windows operating system.

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

**Kernel: V1**

### 4.119.2 Syntax

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

### 4.119.3 Parameter

None.

### 4.119.4 Return Value

None.

### 4.119.5 Errors

None.

## 4.120 sciopta_start

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

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

**Kernel: V1**

### 4.120.1 Syntax

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

## 4.121 Parameter

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

# 5 Kernel Error Reference

## 5.1 Introduction

SCIOPTA has many built-in error check functions.

Contrary to most conventional real-time operating systems, SCIOPTA uses a centralized mechanism for error reporting, called Error Hook. In traditional real-time operating systems, the user needs to check return values of system calls for a possible error condition. In SCIOPTA all error conditions will end up in the Error Hook. This guarantees that all errors are treated and that the error handling does not depend on individual error strategies which might vary from user to user.

Please consult chapter 2.10 “Error Handling” on page 2-48 for more information about the SCIOPTA error handling.

When the error hook is called from the kernel, all information about the error are transferred in 32-bit error word parameter. There are also additional 32-bit extra words (parameters **e0, e1, e2, ...**) available to the user.

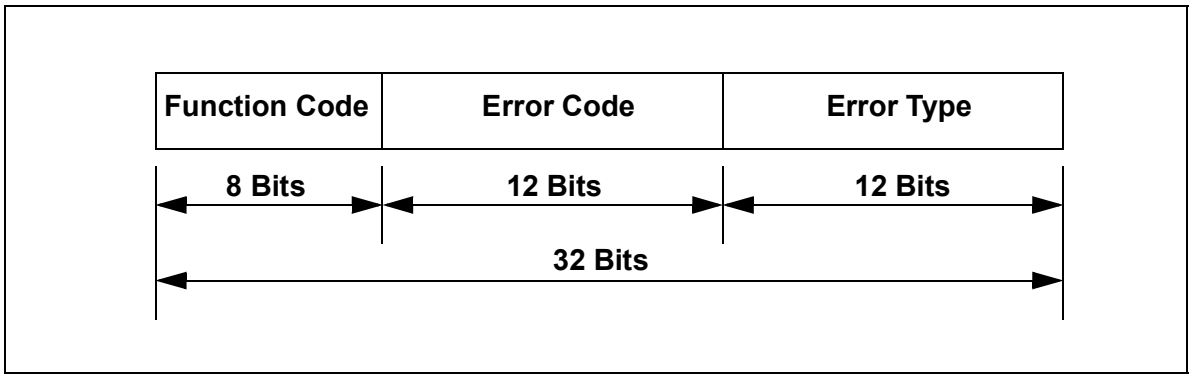


**Figure 5-1: 32-bit Error Word**

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.

## 5.2 Include Files

The error codes are defined in the **err.h** include file.

File location: <install_folder>\sciopta\<version>\include\kernel\ (Kernel Technology V1)

File location: <install_folder>\sciopta\<version>\include\kernel2\ (Kernel Technology V2)

File location: <install_folder>\sciopta\<version>\include\ikernel\ (Kernel Technology V2INT)

The error descriptions are defined in the **errtxt.h** include file.

File location: <install_folder>\sciopta\<version>\include\ossys\

## 5.3 Function Codes (Kernels V1)

| Name | Number | Error Source |
|---|---|---|
| SC_MSGALLOC | 0x01 | sc_msgAlloc |
| SC_MSGFREE | 0x02 | sc_msgFree |
| SC_MSGADDRGET | 0x03 | sc_msgAddrGet |
| SC_MSGSNDGET | 0x04 | sc_msgSndGet |
| SC_MSGSIZEGET | 0x05 | sc_msgOwnerGet |
| SC_MSGSIZESET | 0x06 | sc_msgSizeGet |
| SC_MSGOWNERGET | 0x07 | sc_msgSizeSet |
| SC_MSGTX | 0x08 | sc_msgTx |
| SC_MSGTXALIAS | 0x09 | sc_msgTxAlias |
| SC_MSGRX | 0x0A | sc_msgRx |
| SC_MSGPOOLIDGET | 0x0B | sc_poolIdGet |
| SC_MSGACQUIRE | 0x0C | sc_msgAcquire |
| SC_MSGALLOCCLR | 0x0D | sc_msgAllocClr |
| SC_MSGHOOKREGISTER | 0x0E | sc_msgHookRegister |
| SC_MSGHDCHECK | 0x0F | sc_msgHdCheck |
| SC_POOLCREATE | 0x10 | sc_poolCreate |
| SC_POOLRESET | 0x11 | sc_poolReset |
| SC_POOLKILL | 0x12 | sc_poolKill |
| SC_POOLINFO | 0x13 | sc_poolInfo |
| SC_POOLDEFAULT | 0x14 | sc_poolDefault |
| SC_POOLIDGET | 0x15 | sc_poolIdGet |
| SC_SYSPOOLKILL | 0x16 | sc_sysPoolKill (Internal) |
| SC_POOLHOOKREGISTER | 0x17 | sc_poolHookRegister |
| SC_MISCERRORHOOKREGISTER | 0x18 | sc_miscErrorHookRegister |
| SC_MISCKERNELDREGISTER | 0x19 | sc_miscKerneldRegister (Internal) |
| SC_MISCCRCCONTD | 0x1A | sc_miscCrcContd |
| SC_MISCCRC | 0x1B | sc_miscCrc |
| SC_MISCERRNOSET | 0x1C | sc_miscErrnoSet |
| SC_MISCERRNOGET | 0x1D | sc_miscErrnoGet |
| SC_PROCWAKEUPENABLE | 0x1E | sc_procWakeupEnable |
| SC_PROCWAKEUPDISABLE | 0x1F | sc_procWakeupDisable |
| SC_PROCPRIOGET | 0x20 | sc_procPrioGet |
| SC_PROCPRIOSET | 0x21 | sc_procPrioSet |
| SC_PROCSLICEGET | 0x22 | sc_procSliceGet |
| SC_PROCSLICESET | 0x23 | sc_procSliceSet |
| SC_PROCIDGET | 0x24 | sc_procIdGet |
| SC_PROCPPIDGET | 0x25 | sc_procPpidGet |
| SC_PROCNAMEGET | 0x26 | sc_procNameGet |

| Name | Number | Error Source |
|---|---|---|
| SC_PROCSTART | 0x27 | sc_procStart |
| SC_PROCSTOP | 0x28 | sc_procStop |
| SC_PROCVARINIT | 0x29 | sc_procVarInit |
| SC_PROCSCHEDUNLOCK | 0x2A | sc_procSchedUnlock |
| SC_PROCPRIOCREATESTATIC | 0x2B | sc_procPrioCreateStatic (Internal) |
| SC_PROCINTCREATESTATIC | 0x2C | sc_procIntCreateStatic (Internal) |
| SC_PROCTIMCREATESTATIC | 0x2D | sc_procTimCreateStatic (Internal) |
| SC_PROCUSRINTCREATESTATIC | 0x2E | sc_procUsrIntCreateStatic (Internal) |
| SC_PROCPRIOCREATE | 0x2F | sc_procPrioCreate |
| SC_PROCINTCREATE | 0x30 | sc_procIntCreate |
| SC_PROCTIMCREATE | 0x31 | sc_procTimCreate |
| SC_PROCUSRINTCREATE | 0x32 | sc_procUsrIntCreate |
| SC_PROCKILL | 0x33 | sc_procKill |
| SC_PROCYIELD | 0x34 | sc_procYield |
| SC_PROCOBSERVE | 0x35 | sc_procObserve |
| SC_SYSPROCCREATE | 0x36 | sc_sysProcCreate (Internal) |
| SC_PROCSCHEDLOCK | 0x37 | sc_procSchedLock |
| SC_PROCVARGET | 0x38 | sc_procVarGet |
| SC_PROCVARSET | 0x39 | sc_procVarSet |
| SC_PROCVARDEL | 0x3A | sc_procVarDel |
| SC_PROCVARRM | 0x3B | sc_procVarRm |
| SC_PROCUNOBSERVE | 0x3C | sc_procUnobserve |
| SC_PROCPATHGET | 0x3D | sc_procPathGet |
| SC_PROCPATHCHECK | 0x3E | sc_procPathCheck |
| SC_PROCHOOKREGISTER | 0x3F | sc_procHookRegister |
| SC_MODULECREATE | 0x40 | sc_moduleCreate |
| SC_MODULEKILL | 0x41 | sc_moduleKill |
| SC_MODULENAMEGET | 0x42 | sc_moduleNameGet |
| SC_MODULEIDGET | 0x43 | sc_moduleIdGet |
| SC_MODULEINFO | 0x44 | sc_moduleInfo |
| SC_MODULEPRIOSET | 0x45 | sc_modulePrioSet (Internal) |
| SC_MODULEPRIOGET | 0x46 | sc_modulePrioGet |
| SC_MODULEFRIENDADD | 0x47 | sc_moduleFriendAdd |
| SC_MODULEFRIENDRM | 0x48 | sc_moduleFriendRm |
| SC_MODULEFRIENDGET | 0x49 | sc_moduleFriendGet |
| SC_MODULEFRIENDNONE | 0x4A | sc_moduleFriendNone |
| SC_MODULEFRIENDALL | 0x4B | sc_moduleFriendAll |
| SC_PROCIRQREGISTER | 0x4C | sc_procIrqRegister |
| SC_PROCIRQUNREGISTER | 0x4D | sc_procIrqUnregister |
| SC_PROCDEAMONUNREGISTER | 0x4E | sc_procDaemonUnregister |

| Name | Number | Error Source |
|---|---|---|
| | | |
| SC_PROCDEAMONREGISTER | 0x4F | sc_procDaemonRegister |
| SC_TRIGGERVALUESET | 0x50 | sc_triggerValueSet |
| SC_TRIGGERVALUEGET | 0x51 | sc_triggerValueGet |
| SC_TRIGGER | 0x52 | sc_trigger |
| SC_TRIGGERWAIT | 0x53 | sc_triggerWait |
| SC_SYSERROR | 0x54 | sc_sysError (Internal) |
| SC_MISCERROR | 0x55 | sc_miscError |
| SC_MODULECREATE2 | 0x56 | sc_moduleCreate2 |
| SC_TICK | 0x57 | sc_tick |
| SC_TMOADD | 0x58 | sc_tmoAdd |
| SC_TMO | 0x59 | sc_tmo (Internal) |
| SC_SLEEP | 0x5A | sc_sleep |
| SC_TMORM | 0x5B | sc_tmoRm |
| SC_TICKGET | 0x5C | sc_tickGet |
| SC_TICKLENGTH | 0x5D | sc_tickLength |
| SC_TICKMS2TICK | 0x5E | sc_tickMs2Tick |
| SC_TICKTICK2MS | 0x5F | sc_tickTick2Ms |
| SC_CONNECTORREGISTER | 0x60 | sc_connectorRegister |
| SC_CONNECTORUNREGISTER | 0x61 | sc_connectorUnregister |
| | 0x62 | <dispatcher> |
| | 0x63 | reserved |
| | 0x64 | reserved |
| SC_MSGALLOCTX | 0x65 | sc_msgAllocTx |
| SC_CONNECTORREMOTE2LOCAL | 0x66 | sc_connectorRemote2local (Internal) |

## 5.4 Function Codes (Kernels V2 and V2INT)

| Name | Number | Error Source |
|---|---|---|
| SC_MSGALLOC | 0x01 | sc_msgAlloc |
| SC_MSGFREE | 0x02 | sc_msgFree |
| SC_MSGADDRGET | 0x03 | sc_msgAddrGet |
| SC_MSGSNDGET | 0x04 | sc_msgSndGet |
| SC_MSGSIZEGET | 0x05 | sc_msgSizeGet |
| SC_MSGSIZESET | 0x06 | sc_msgSizeSet |
| SC_MSGOWNERGET | 0x07 | sc_msgOwnerGet |
| SC_MSGTX | 0x08 | sc_msgTx |
| SC_MSGTXALIAS | 0x09 | sc_msgTxAlias |
| SC_MSGRX | 0x0A | sc_msgRx |
| SC_MSGPOOLIDGET | 0x0B | sc_poolIdGet |
| SC_MSGACQUIRE | 0x0C | sc_msgAcquire |
| SC_MSGALLOCCLR | 0x0D | sc_msgAllocClr |
| SC_MSGHOOKREGISTER | 0x0E | sc_msgHookRegister |
| SC_MSGHDCHECK | 0x0F | sc_msgHdCheck |
| SC_TMOADD | 0x10 | sc_tmoAdd |
| SC_TMORM | 0x11 | sc_tmoRm |
| SC_MSGFIND | 0x12 | sc_msgFind |
| SC_MSGALLOCTX | 0x13 | sc_msgAllocTx |
| SC_MSGDATACRCSET | 0x14 | sc_msgDataCrcSet |
| SC_MSGDATACRCGET | 0x15 | sc_msgDataCrcGet |
| SC_MSGDATACRCDIS | 0x16 | sc_msgDataCrcDis |
| SC_MSGFLOWSIGNATUREUPDATE | 0x17 | sc_msgFlowSignatureUpdate |
| SC_POOLCREATE | 0x18 | sc_poolCreate |
| SC_POOLRESET | 0x19 | sc_poolReset |
| SC_POOLKILL | 0x1A | sc_poolKill |
| SC_POOLINFO | 0x1B | sc_poolInfo |
| SC_POOLDEFAULT | 0x1C | sc_poolDefault |
| SC_POOLIDGET | 0x1D | sc_procIdGet |
| SC_POOLHOOKREGISTER | 0x1E | sc_poolHookRegister |
| SC_POOLCBCHK | 0x1F | sc_poolCBChk |
| SC_MISCERRORHOOKREGISTER | 0x20 | sc_miscErrorHookRegister |
| SC_MISCKERNELDREGISTER | 0x21 | sc_miscKerneldRegister (Internal) |
| SC_MISCCRCCONTD | 0x22 | sc_miscCrcContd |
| SC_MISCCRC | 0x23 | sc_miscCrc |
| SC_MISCCRC32CONTD | 0x24 | sc_miscCrc32Contd |
| SC_MISCCRC32 | 0x25 | sc_miscCrc32 |

| Name | Number | Error Source |
|---|---|---|
| SC_MISCERRNOSET | 0x26 | sc_miscErrnoSet |
| SC_MISCERRNOGET | 0x27 | sc_miscErrnoGet |
| SC_MISCERROR | 0x28 | sc_miscError |
| SC_MISCCRCSTRING | 0x29 | sc_miscCrcString |
| | 0x2A | reserved |
| | 0x2B | reserved |
| | 0x2C | reserved |
| SC_MISCFLOWSIGNATUREINIT | 0x2D | sc_miscFlowSignatureInit |
| SC_MISCFLOWSIGNATUREUPDATE | 0x2E | sc_miscFlowSignatureUpdate |
| SC_MISCFLOWSIGNATUREGET | 0x2F | sc_miscFlowSignatureGet |
| SC_PROCWAKEUPENABLE | 0x30 | sc_procWakeupEnable |
| SC_PROCWAKEUPDISABLE | 0x31 | sc_procWakeupDisable |
| SC_PROCPRIOGET | 0x32 | sc_procPrioGet |
| SC_PROCPRIOSET | 0x33 | sc_procPrioSet |
| SC_PROCSLICEGET | 0x34 | sc_procSliceGet |
| SC_PROCSLICESET | 0x35 | sc_procSliceSet |
| SC_PROCIDGET | 0x36 | sc_procIdGet |
| SC_PROCPPIDGET | 0x37 | sc_procPpidGet |
| SC_PROCNAMEGET | 0x38 | sc_procNameGet |
| SC_PROCPATHGET | 0x39 | sc_procPathGet |
| SC_PROCATTRGET | 0x3A | sc_procAttrGet |
| SC_PROCVECTORGET | 0x3B | sc_procVectorGet |
| SC_PROCPATHCHECK | 0x3C | sc_procPathCheck |
| SC_PROCIRQREGISTER | 0x3D | sc_procIrqRegister |
| SC_PROCIRQUNREGISTER | 0x3E | sc_procIrqUnregister |
| | 0x3F | reserved |
| SC_PROCSTART | 0x40 | sc_procStart |
| SC_PROCSTOP | 0x41 | sc_procStop |
| SC_PROCSCHEDLOCK | 0x42 | sc_procSchedLock |
| SC_PROCSCHEDUNLOCK | 0x43 | sc_procSchedUnlock |
| SC_PROCYIELD | 0x44 | sc_procYield |
| SC_PROCCREATE2 | 0x45 | sc_procCreate2 |
| SC_PROCKILL | 0x46 | sc_procKill |
| SC_PROCOBSERVE | 0x47 | sc_procObserve |
| SC_PROCUNOBSERVE | 0x48 | sc_procUnobserve |
| SC_PROCVARINIT | 0x49 | sc_procVarInit |
| SC_PROCVARGET | 0x4A | sc_procVarGet |
| SC_PROCVARSET | 0x4B | sc_procVarSet |
| SC_PROCVARDEL | 0x4C | sc_procVarDel |
| SC_PROCVARRM | 0x4D | sc_procVarRm |

| Name | Number | Error Source |
|---|---|---|
| SC_PROCATEXIT | 0x4E | sc_procAtExit |
| SC_PROCHOOKREGISTER | 0x4F | sc_procHookRegister |
| SC_PROCDAEMONUNREGISTER | 0x50 | sc_procDaemonUnregister |
| SC_PROCDAEMONREGISTER | 0x51 | sc_procDaemonRegister |
| | 0x52 | |
| | 0x53 | |
| | 0x54 | |
| | 0x55 | |
| | 0x56 | |
| | 0x57 | |
| | 0x58 | |
| | 0x59 | |
| | 0x9A | |
| | 0x9B | |
| SC_PROCCBCHK | 0x5C | sc_procCBChk |
| SC_PROCFLOWSIGNATUREINIT | 0x5D | sc_procFlowSignatureInit |
| SC_PROCFLOWSIGNATUREUPDATE | 0x5E | sc_procFlowSignatureUpdate |
| SC_PROCFLOWSIGNATUREGET | 0x5F | sc_procFlowSignatureGet |
| SC_MODULECREATE2 | 0x60 | sc_moduleCreate2 |
| SC_MODULEKILL | 0x61 | sc_moduleKill |
| SC_MODULENAMEGET | 0x62 | sc_moduleNameGet |
| SC_MODULEIDGET | 0x63 | sc_moduleIdGet |
| SC_MODULEINFO | 0x64 | sc_moduleInfo |
| SC_MODULEPRIOGET | 0x65 | sc_modulePrioGet |
| SC_MODULEFRIENDADD | 0x66 | sc_moduleFriendAdd |
| SC_MODULEFRIENDRM | 0x67 | sc_moduleFriendRm |
| SC_MODULEFRIENDGET | 0x68 | sc_moduleFriendGet |
| SC_MODULEFRIENDNONE | 0x69 | sc_moduleFriendNone |
| SC_MODULEFRIENDALL | 0x6A | sc_moduleFriendAll |
| SC_MODULESTART | 0x6B | sc_moduleStart (Internal) |
| SC_MODULESTOP | 0x6C | sc_moduleStop |
| | 0x6D | reserved |
| | 0x6E | reserved |
| SC_MODULECBCHK | 0x6F | sc_moduleCBChk |
| SC_TRIGGERVALUESET | 0x70 | sc_triggerValueSet |
| SC_TRIGGERVALUEGET | 0x71 | sc_triggerValueGet |
| SC_TRIGGER | 0x72 | sc_trigger |
| SC_TRIGGERWAIT | 0x73 | sc_triggerWait |
| | 0x74 | reserved |
| | 0x75 | reserved |

| Name | Number | Error Source |
|---|---|---|
| | 0x76 | reserved |
| | 0x77 | reserved |
| SC_TICKACTIVATIONGET | 0x78 | sc_tickActivationGet |
| SC_TICK | 0x79 | sc_tick |
| SC_TICKGET | 0x7A | sc_tickGet |
| SC_TICKLENGTH | 0x7B | sc_tickLength |
| SC_TICKMS2TICK | 0x7C | sc_tickMs2Tick |
| SC_TICKTICK2MS | 0x7D | sc_tickTick2Ms |
| SC_SLEEP | 0x7E | sc_sleep |
| | 0x7F | reserved |
| SC_CONNECTORREGISTER | 0x80 | sc_connectorRegister |
| SC_CONNECTORUNREGISTER | 0x81 | sc_connectorUnregister |
| | 0x82 | reserved |
| | 0x83 | reserved |
| | 0x84 | reserved |
| | 0x85 | reserved |
| | 0x86 | reserved |
| | 0x87 | reserved |
| | 0x88 | dispatcher (Internal) |
| | 0x89 | irq_dispatcher (Internal) |
| | 0x8A | kernel_private (Internal) |
| SC_SYSERROR | 0x8B | sc_sysError (many) |
| | 0x8C | sc_safe_* (many) |
| SC_SYSLDATACORRUPT | 0x8D | many |
| | 0x8E | reserved |
| | 0x8F | reserved |

## 5.5 Error Codes

| Name | Number | Description |
|---|---|---|
| KERNEL_EILL_POOL_ID | 0x001 | Illegal pool ID. |
| KERNEL_ENO_MOORE_POOL | 0x002 | No more pool. |
| KERNEL_EILL_POOL_SIZE | 0x003 | Illegal pool size. |
| KERNEL_EPOOL_IN_USE | 0x004 | Pool still in use. |
| KERNEL_EILL_NUM_SIZES | 0x005 | Illegal number of buffer sizes. |
| KERNEL_EILL_BUF_SIZES | 0x006 | Illegal buffersizes. |
| KERNEL_EILL_BUFSIZE | 0x007 | Illegal buffersize. |
| KERNEL_EOUTSIDE_POOL | 0x008 | Message outside pool. |
| KERNEL_EMSG_HD_CORRUPT | 0x009 | Message header corrupted. |
| KERNEL_ENIL_PTR | 0x00A | NIL pointer. |
| KERNEL_EENLARGE_MSG | 0x00B | Message enlarged. |
| KERNEL_ENOT_OWNER | 0x00C | Not owner of the message. |
| KERNEL_EOUT_OF_MEMORY | 0x00D | Out of memory. |
| KERNEL_EILL_VECTOR | 0x00E | Illegal interrupt vector. |
| KERNEL_EILL_SLICE | 0x00F | Illegal time slice. |
| KERNEL_ENO_KERNELD | 0x010 | No kernel daemon started. |
| KERNEL_EMSG_ENDMARK_CORRUPT | 0x011 | Message endmark corrupted. |
| KERNEL_EMSG_PREV_ENDMARK_CORRUPT | 0x012 | Previous message's endmark corrupted. |
| KERNEL_EILL_DEFPOOL_ID | 0x013 | Illegal default pool ID. |
| KERNEL_ELOCKED | 0x014 | Illegal system call while scheduler locked. |
| KERNEL_EILL_PROCTYPE | 0x015 | Illegal process type. |
| KERNEL_EILL_INTERRUPT | 0x016 | Illegal interrupt. |
| KERNEL_EILL_EXCEPTION | 0x017 | Illegal unhandled exception. |
| KERNEL_EILL_SYSCALL | 0x018 | Illegal syscall number. |
| KERNEL_EILL_NESTING | 0x019 | Illegal interrupt nesting. |
| KERNEL_EUNLOCK_WO_LOCK | 0x01F | Unlock without lock. |
| KERNEL_EILL_PID | 0x020 | Illegal process ID. |
| KERNEL_ENO_MORE_PROC | 0x021 | No more processes. |
| KERNEL_EMODULE_TOO_SMALL | 0x022 | Module size too small. |
| KERNEL_ESTART_NOT_STOPPED | 0x023 | Starting of a not stopped process. |
| KERNEL_EILL_PROC | 0x024 | Illegal process. |
| KERNEL_EILL_NAME | 0x025 | Illegal name. |
| 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_EMODULE_OVERLAP | 0x033 | Module memory overlaps. |
| KERNEL_EPROC_TERMINATE | 0xFFF | Process terminated. |

## 5.6 Error Types

| Name | Bit | Description |
|---|---|---|
| SC_ERR_SYSTEM_FATAL | 0x01 | This type of error will stop the whole target. |
| SC_ERR_MODULE_FATAL | 0x02 | This type of error results in killing the module if an error hook returns a value of !=0. |
| SC_ERR_PROCESS_FATAL | 0x04 | This type of error results in killing the process if an error hook returns a value of !=0. |
| SC_ERR_SYSTEM_WARNING | 0x10 | Warning on target level.The system continues if an error hook is installed. |
| SC_ERR_MODULE_WARNING | 0x20 | Warning on module level. The system continues if an error hook is installed. |
| SC_ERR_PROC_WARNING | 0x40 | Warning on process level. The system continues if an error hook is installed. |

# 6 Manual Versions

## 6.1 Manual Version 5.0

- Whole manual restructured and rewritten.

# 7 Index


**A**

Activate a Process Trigger ..... 4-188
Add a Friend Module ..... 4-26
Add a Time-Out Request ..... 4-184
Addressee of a Message ..... 4-42
Allocate a Message ..... 4-44
Allocate a Message and Clear Content ..... 4-47
Allocate a Message with Time-Out ..... 4-44
Allocate Memory ..... 4-44
allocated-messages queue ..... 4-56

**C**

Calculate a 16 Bit CRC ..... 4-5
Calculate a 32 Bit CRC ..... 4-7
Calculates an Additional CRC ..... 4-6
Calculates an Additional CRC32 ..... 4-8
Call Kernel Tick ..... 4-178
Call the Error Hook with a User Error ..... 4-11
Cancel a Process Observation ..... 4-157
Cancel a Process Supervision ..... 4-157
Change the Owner of a Message ..... 4-40
Change the Sender of a Message ..... 4-81
Check message header ..... 4-62
Check the Path of a Process ..... 4-134
Clear a Message Pool ..... 4-98
CONNECTOR Process ..... 4-1
CONNECTOR Process Calls ..... 3-9
Convert Milliseconds in System Ticks ..... 4-182
Convert System Ticks in Milliseconds ..... 4-183
CRC ..... 4-5, 4-6
CRC32 ..... 4-7, 4-8
Create a Message Pool ..... 4-85
Create a Module ..... 4-18, 4-21
Creates a process ..... 4-106

**D**

Default Pool ..... 4-88
Define a Process Variable ..... 4-163
Define all Modules as Friends ..... 4-27
Delete Process Variable ..... 4-158
Disable wakeup of a timer or interrupt process ..... 4-166

**E**

Enable wakeup of a timer or interrupt process ..... 4-165
err.h ..... 5-1
errno Variable ..... 4-9, 4-10
Error Check ..... 5-1
Error Code ..... 5-1

Error Codes ..... 5-9
Error Function Codes ..... 5-2
Error Hook ..... 4-11, 4-12, 4-13
Error Include Files ..... 5-1
Error Number ..... 4-9, 4-10
Error Type ..... 5-1
Error Types ..... 5-10
errtxt.h ..... 5-1
External Process ..... 4-1

## F

Find a message in the allocated-messages queue ..... 4-56
Free a Message ..... 4-60
Function Code ..... 5-1

## G

Get a Process Variable ..... 4-159
Get Information About a Message Pool ..... 4-92
Get Information About a Module ..... 4-32
Get Module Friend Information ..... 4-28
Get the Addressee of a Message ..... 4-42
Get the Creator Process ..... 4-137
Get the ID of a Message Pool ..... 4-91
Get the ID of a Module ..... 4-31
Get the ID of a Process ..... 4-121
Get the Interrupt Vector ..... 4-164
Get the Name of a Module ..... 4-37
Get the Name of Process ..... 4-130
Get the Owner of a Message ..... 4-65
Get the Parent Process ..... 4-137
Get the Path of a Process ..... 4-135
Get the Pool ID of a Message ..... 4-67
Get the Priority of a Process ..... 4-142
Get the Process Error Number ..... 4-9
Get the Requested Size of a Message ..... 4-72
Get the Sender of a Message ..... 4-76
Get the Tick Counter Value ..... 4-180
Get the Time Slice of a Timer Process ..... 4-148
Get the Value of a Process Trigger ..... 4-190
Get time of last activation ..... 4-179

## I

Initialize Process Variable Area ..... 4-160
Installed files ..... 6-1
Interrupt Source Parameter ..... 2-13
Interrupt Vector Parameter ..... 2-13

## K

Kernel Daemon ..... 4-18, 4-21, 4-35, 4-106, 4-128
Kernel Error Codes ..... 5-1

Kernel Tick Counter ..... 4-178
Kernel Tick Function ..... 4-178
KERNEL_EALREADY_DEFINED ..... 5-10
KERNEL_EENLARGE_MSG ..... 5-9
KERNEL_EILL_BUF_SIZES ..... 5-9
KERNEL_EILL_BUFSIZE ..... 5-9
KERNEL_EILL_DEFPOOL_ID ..... 5-9
KERNEL_EILL_EXCEPTION ..... 5-9
KERNEL_EILL_INTERRUPT ..... 5-9
KERNEL_EILL_MODULE ..... 5-9
KERNEL_EILL_MODULE_NAME ..... 5-9
KERNEL_EILL_NAME ..... 5-9
KERNEL_EILL_NESTING ..... 5-9
KERNEL_EILL_NUM_SIZES ..... 5-9
KERNEL_EILL_PARAMETER ..... 5-10
KERNEL_EILL_PID ..... 5-9
KERNEL_EILL_POOL_ID ..... 5-9
KERNEL_EILL_POOL_SIZE ..... 5-9
KERNEL_EILL_PRIORITY ..... 5-9
KERNEL_EILL_PROC ..... 5-9
KERNEL_EILL_PROC_NAME ..... 5-10
KERNEL_EILL_PROCTYPE ..... 5-9
KERNEL_EILL_SLICE ..... 5-9
KERNEL_EILL_STACKSIZE ..... 5-9
KERNEL_EILL_SYSCALL ..... 5-9
KERNEL_EILL_TARGET_NAME ..... 5-9
KERNEL_EILL_VALUE ..... 5-10
KERNEL_EILL_VECTOR ..... 5-9
KERNEL_ELOCKED ..... 5-9
KERNEL_EMODULE_OVERLAP ..... 5-10
KERNEL_EMODULE_TOO_SMALL ..... 5-9
KERNEL_EMSG_ENDMARK_CORRUPT ..... 5-9
KERNEL_EMSG_HD_CORRUPT ..... 5-9
KERNEL_EMSG_PREV_ENDMARK_CORRUPT ..... 5-9
KERNEL_ENIL_PTR ..... 5-9
KERNEL_ENO_KERNELD ..... 5-9
KERNEL_ENO_MOORE_POOL ..... 5-9
KERNEL_ENO_MORE_CONNECTOR ..... 5-10
KERNEL_ENO_MORE_MODULE ..... 5-9
KERNEL_ENO_MORE_PROC ..... 5-9
KERNEL_ENOT_OWNER ..... 5-9
KERNEL_EOUT_OF_MEMORY ..... 5-9
KERNEL_EOUTSIDE_POOL ..... 5-9
KERNEL_EPOOL_IN_USE ..... 5-9
KERNEL_EPROC_NOT_PRIO ..... 5-10
KERNEL_EPROC_TERMINATE ..... 5-10
KERNEL_ESTACK_OVERFLOW ..... 5-10
KERNEL_ESTACK_UNDERFLOW ..... 5-10
KERNEL_ESTART_NOT_STOPPED ..... 5-9
KERNEL_EUNLOCK_WO_LOCK ..... 5-9
Kill a Message Pool ..... 4-96
Kill a Module ..... 4-35

Kill a Process ..... 4-128

**L**

Lock the Scheduler ..... 4-146

**M**

mdb ..... 4-22
Message Ownership ..... 4-44
Message Pool ..... 4-44, 4-67
Message pool ..... 4-85
Message Pool Calls ..... 3-7
Message Queue ..... 4-69
Message Sent to Unknown Process ..... 2-35
Message Size ..... 4-74
Message System Calls ..... 3-1
Miscellaneous and Error Calls ..... 3-9, 3-10
Modify a Process Variable ..... 4-163
Module Address and Size ..... 4-23
Module Control Block ..... 4-32
Module Descriptor Block ..... 4-22
Module Info Structure ..... 4-33
Module System Calls ..... 3-6
module.h ..... 4-32

**N**

Name of a process ..... 4-130

**O**

Observe a Process ..... 4-132
Owner of a Message ..... 4-40, 4-65

**P**

Path of a process ..... 4-135
Plausibility check ..... 4-62
Pool Control Block ..... 4-92
Pool Create Hook ..... 4-89
Pool ID ..... 4-67
Pool Info Structure ..... 4-93
Pool Kill Hook ..... 4-89
Pool Statistics Info Structure ..... 4-94
Priority Range ..... 4-142
Process Daemon ..... 4-114, 4-121
Process Hook ..... 4-119
Process ID ..... 4-121
Process Path ..... 4-121, 4-128, 4-134
Process System Calls ..... 3-3
Process Ticks ..... 4-148
Process Trigger ..... 4-188
Process Trigger Calls ..... 3-9

## R

Read a Process Variable ..... 4-159
Receive a Message ..... 4-56, 4-69
Receive a Message with Time-Out ..... 4-69
Register a CONNECTOR Process ..... 4-1
Register a Message Hook ..... 4-63
Register a Pool Hook ..... 4-89
Register a Process Daemon ..... 4-114
Register a process exit function ..... 4-100, 4-117, 4-118
Register a Process Hook ..... 4-119
Register an Error Hook ..... 4-13
Remove a CONNECTOR Process ..... 4-3, 4-4
Remove a Module from Friends ..... 4-30
Remove a Process Variable ..... 4-158
Remove a Time-Out Request ..... 4-186
Remove a Whole Process Variable Area ..... 4-162
Reset a Message Pool ..... 4-98
Return a Message ..... 4-60
Return specific process attributes ..... 4-102, 4-104

## S

SC_CONNECTORREGISTER ..... 5-4, 5-8
sc_connectorRegister ..... 4-1
SC_CONNECTORUNREGISTER ..... 5-4, 5-8
sc_connectorUnregister ..... 4-3, 4-4
SC_DEFAULT_POOL ..... 4-88
SC_DISPATCHER ..... 5-4, 5-8
SC_ENDLESS_TMO ..... 4-45, 4-70
SC_ERR_MODULE_FATAL ..... 5-10
SC_ERR_MODULE_WARNING ..... 5-10
SC_ERR_PROC_WARNING ..... 5-10
SC_ERR_PROCESS_FATAL ..... 5-10
SC_ERR_TARGET_FATAL ..... 5-10
SC_ERR_TARGET_WARNING ..... 5-10
SC_FATAL_IF_TMO ..... 4-45
SC_INT_PROCESS ..... 2-13
SC_INT_PROCESS_EX ..... 2-13
SC_IRQDISPATCHER ..... 5-4, 5-8
SC_MISCCRC ..... 5-2, 5-5
sc_miscCrc ..... 4-5
SC_MISCCRC32 ..... 5-5
sc_miscCrc32 ..... 4-7
SC_MISCCRC32CONTD ..... 5-5
sc_miscCrc32Contd ..... 4-8
SC_MISCCRCCONTD ..... 5-2, 5-5
sc_miscCrcContd ..... 4-6
SC_MISCERRNOGET ..... 5-2, 5-6
sc_miscErrnoGet ..... 4-9
SC_MISCERRNOSET ..... 5-2, 5-6
sc_miscErrnoSet ..... 4-10
SC_MISCERROR ..... 5-6

sc_miscError ..... 4-11
SC_MISCERRORHOOKREGISTER ..... 5-2, 5-5
sc_miscErrorHookRegister ..... 4-13
SC_MISCFLOWSIGNATUREGET ..... 5-6
SC_MISCFLOWSIGNATUREINIT ..... 5-6
SC_MISCFLOWSIGNATUREUPDATE ..... 5-6
SC_MISCKERNELDREGISTER ..... 5-2, 5-5
SC_MODULECBCHK ..... 5-7
SC_MODULECREATE2 ..... 5-3, 5-7
sc_moduleCreate2 ..... 4-21
SC_MODULEIDGET ..... 5-3, 5-7
sc_moduleIdGet ..... 4-31
SC_MODULEINFO ..... 5-3, 5-7
sc_moduleInfo ..... 4-32
SC_MODULEKILL ..... 5-3, 5-7
sc_moduleKill ..... 4-35
SC_MODULENAMEGET ..... 5-3, 5-7
sc_moduleNameGet ..... 4-37
SC_MODULEPRIOGET ..... 5-3, 5-7
sc_modulePrioGet ..... 4-38
SC_MODULESTOP ..... 5-4, 5-7
sc_moduleStop ..... 4-39
SC_MSGACQUIRE ..... 5-2, 5-5
sc_msgAcquire ..... 4-40
SC_MSGADDRGET ..... 5-2, 5-5
sc_msgAddrGet ..... 4-42
SC_MSGALLOC ..... 5-2, 5-5
sc_msgAlloc ..... 4-44
sc_msgAllocClr ..... 4-47
SC_MSGDATACRCDIS ..... 5-5
SC_MSGDATACRCGET ..... 5-5
SC_MSGDATACRCSET ..... 5-5
sc_msgFind ..... 4-56
SC_MSGFLOWSIGNATUREUPDATE ..... 5-5
SC_MSGFREE ..... 5-2, 5-5
sc_msgFree ..... 4-60
sc_msgHookRegister ..... 4-63
SC_MSGOWNERGET ..... 5-2, 5-5
sc_msgOwnerGet ..... 4-65
SC_MSGPOOLIDGET ..... 5-2, 5-5
sc_msgPoolIdGet ..... 4-67
SC_MSGRX ..... 5-2, 5-5
sc_msgRx ..... 4-69
SC_MSGRX_ALL ..... 4-57, 4-70
SC_MSGRX_BOTH ..... 4-57, 4-70
SC_MSGRX_MSGID ..... 4-57, 4-70
SC_MSGRX_NOT ..... 4-57, 4-70
SC_MSGRX_PID ..... 4-57, 4-70
SC_MSGSIZEGET ..... 5-2, 5-5
sc_msgSizeGet ..... 4-72
SC_MSGSIZESET ..... 5-2, 5-5
sc_msgSizeSet ..... 4-74

SC_MSGSNDGET ..... 5-2, 5-5
sc_msgSndGet ..... 4-76
SC_MSGTX ..... 5-2, 5-5
sc_msgTx ..... 4-78
SC_MSGTXALIAS ..... 5-2, 5-5
sc_msgTxAlias ..... 4-81
SC_NO_TMO ..... 4-45, 4-70
SC_POOLCBCHK ..... 5-5
SC_POOLCREATE ..... 5-5
sc_poolCreate ..... 4-85
SC_POOLDEFAULT ..... 5-2, 5-5
sc_poolDefault ..... 4-88
SC_POOLHOOKREGISTER ..... 5-2, 5-5
sc_poolHookRegister ..... 4-89
SC_POOLIDGET ..... 5-2, 5-5
sc_poolIdGet ..... 4-91
SC_POOLINFO ..... 5-2, 5-5
sc_poolInfo ..... 4-92
SC_POOLKILL ..... 5-2, 5-5
sc_poolKill ..... 4-96
SC_POOLRESET ..... 5-5
sc_poolReset ..... 4-98
SC_PROC_WAKEUP_CALLBACK ..... 2-13
SC_PROC_WAKEUP_CREATE ..... 2-13
SC_PROC_WAKEUP_HARDWARE ..... 2-13
SC_PROC_WAKEUP_KILL ..... 2-13
SC_PROC_WAKEUP_MESSAGE ..... 2-13
SC_PROC_WAKEUP_START ..... 2-13
SC_PROC_WAKEUP_STOP ..... 2-13
SC_PROC_WAKEUP_TRIGGER ..... 2-13
SC_PROCATEXIT ..... 5-7
sc_procAtExit ..... 4-100, 4-117, 4-118
SC_PROCATTRGET ..... 5-6
sc_procAttrGet ..... 4-102, 4-104
SC_PROCCBCHK ..... 5-7
sc_procCreate2 ..... 4-106
SC_PROCDAEMONREGISTER ..... 5-7
sc_procDaemonRegister ..... 4-114
SC_PROCDAEMONUNREGISTER ..... 5-7
sc_procDaemonUnregister ..... 4-115
SC_PROCFLOWSIGNATUREGET ..... 5-7
SC_PROCFLOWSIGNATUREINIT ..... 5-7
SC_PROCFLOWSIGNATUREUPDATE ..... 5-7
SC_PROCHOOKREGISTER ..... 5-3, 5-7
sc_procHookRegister ..... 4-119
SC_PROCIDGET ..... 5-2, 5-6
sc_procIdGet ..... 4-121
sc_procIdGet in Interrupt Processes ..... 4-122
SC_PROCKILL ..... 5-6
sc_procKill ..... 4-128
SC_PROCNAMEGET ..... 5-2, 5-6
sc_procNameGet ..... 4-130

SC_PROCNAMEGETMSG_REPLY ..... 4-130, 4-136
SC_PROCOBSERVE ..... 5-3, 5-6
sc_procObserve ..... 4-132
SC_PROCPATHCHECK ..... 5-6
sc_procPathCheck ..... 4-134
SC_PROCPATHGET ..... 5-6
sc_procPathGet ..... 4-135
SC_PROCPPIDGET ..... 5-2, 5-6
sc_procPpidGet ..... 4-137
SC_PROCPRIOGET ..... 5-2, 5-6
sc_procPrioGet ..... 4-142
SC_PROCPRIOSET ..... 5-2, 5-6
sc_procPrioSet ..... 4-144
SC_PROCSCHEDLOCK ..... 5-6
sc_procSchedLock ..... 4-146
SC_PROCSCHEDUNLOCK ..... 5-3, 5-6
sc_procSchedUnLock ..... 4-147
SC_PROCSLICEGET ..... 5-2, 5-6
sc_procSliceGet ..... 4-148
SC_PROCSLICESET ..... 5-2, 5-6
sc_procSliceSet ..... 4-149
SC_PROCSTART ..... 5-3, 5-6
sc_procStart ..... 4-151
SC_PROCSTOP ..... 5-3, 5-6
sc_procStop ..... 4-153
sc_procUnobserve ..... 4-157
SC_PROCVARDEL ..... 5-3, 5-6
sc_procVarDel ..... 4-158
SC_PROCVARGET ..... 5-3, 5-6
sc_procVarGet ..... 4-159
SC_PROCVARINIT ..... 5-6
sc_procVarInit ..... 4-160
SC_PROCVARRM ..... 5-3, 5-6
sc_procVarRm ..... 4-162
SC_PROCVARSET ..... 5-3, 5-6
sc_procVarSet ..... 4-163
SC_PROCVECTORGET ..... 5-6
sc_procVectorGet ..... 4-164
SC_PROCWAKEUPDISABLE ..... 5-2, 5-6
sc_procWakeupDisable ..... 4-166
SC_PROCWAKEUPENABLE ..... 5-2, 5-6
sc_procWakeupEnable ..... 4-165
SC_PROCYIELD ..... 5-3, 5-6
sc_procYield ..... 4-167, 4-168
SC_SET_MSGRX_HOOK ..... 4-63
SC_SET_MSGTX_HOOK ..... 4-63
SC_SET_POOLCREATE_HOOK ..... 4-89
SC_SET_POOLKILL_HOOK ..... 4-89
SC_SET_PROCCREATE_HOOK ..... 4-119
SC_SET_PROCKILL_HOOK ..... 4-119
SC_SET_PROCSWAP_HOOK ..... 4-119
SC_SLEEP ..... 5-8

sc_sleep ..... 4-176
SC_SYSERROR ..... 5-4, 5-8
SC_SYSLDATACORRUPT ..... 5-4, 5-8
SC_TICK ..... 5-4, 5-8
sc_tick ..... 4-178
SC_TICKACTIVATIONGET ..... 5-4, 5-8
sc_tickActivationGet ..... 4-179
SC_TICKGET ..... 5-4, 5-8
sc_tickGet ..... 4-180
SC_TICKLENGTH ..... 5-4, 5-8
sc_tickLength ..... 4-181
SC_TICKMS2TICK ..... 5-4, 5-8
sc_tickMs2Tick ..... 4-182
SC_TICKTICK2MS ..... 5-4, 5-8
sc_tickTick2Ms ..... 4-183
SC_TMO_MAX ..... 4-45, 4-70
sc_tmoAdd ..... 4-184
sc_tmoRm ..... 4-186
SC_TRIGGER ..... 5-4, 5-7
sc_trigger ..... 4-188
SC_TRIGGERVALUEGET ..... 5-4, 5-7
sc_triggerValueGet ..... 4-190
SC_TRIGGERVALUESET ..... 5-4, 5-7
sc_triggerValueSet ..... 4-191
SC_TRIGGERWAIT ..... 5-4, 5-7
sc_triggerWait ..... 4-192
Scheduler Lock Counter ..... 4-146, 4-147
SCIOPTA System Framework ..... 2-3
Send a Message ..... 4-78, 4-83
Sender of a Message ..... 4-76
Set a Message Pool as Default Pool ..... 4-88
Set a Process Error Number ..... 4-10
Set all Module as No-Friends ..... 4-29
Set the Priority of a Process ..... 4-144
Set the Size of a Message ..... 4-74
Set the Tick Length ..... 4-181
Set the Time Slice of Timer Process ..... 4-149
Set the Value of a Trigger ..... 4-191
Setup a Process Variable Area ..... 4-160
Size of a Message ..... 4-72
Start a Process ..... 4-151
Start/Stop Counter ..... 4-151, 4-153
Stop a module ..... 4-39
Stop a Process ..... 4-153, 4-155
Stop all processes in a module ..... 4-39
Supervise a Process ..... 4-132
System call reference ..... 4-1
System Calls Overview ..... 3-1
System error ..... 4-11
System Tick Calls ..... 3-8
System Ticks ..... 4-176

**T**
Time-out Expired ..... 4-186
Timing Calls ..... 3-7
Transmitt a Message ..... 4-78, 4-83
Trigger ..... 4-188

**U**
Unknown Process ..... 2-35
Unlock the Scheduler ..... 4-147
Unregister a Process Daemon ..... 4-115

**W**
Wait on the Process Trigger ..... 4-192
Wanted Array in Receive Call ..... 4-69

**Y**
Yield the CPU ..... 4-167, 4-168, 4-169, 4-170, 4-172, 4-174, 4-175