SCIOPTA

High Performance
Real-Time Operating Systems

Kernel V2

User’s Manual

Copyright

Disclaimer

Trademark

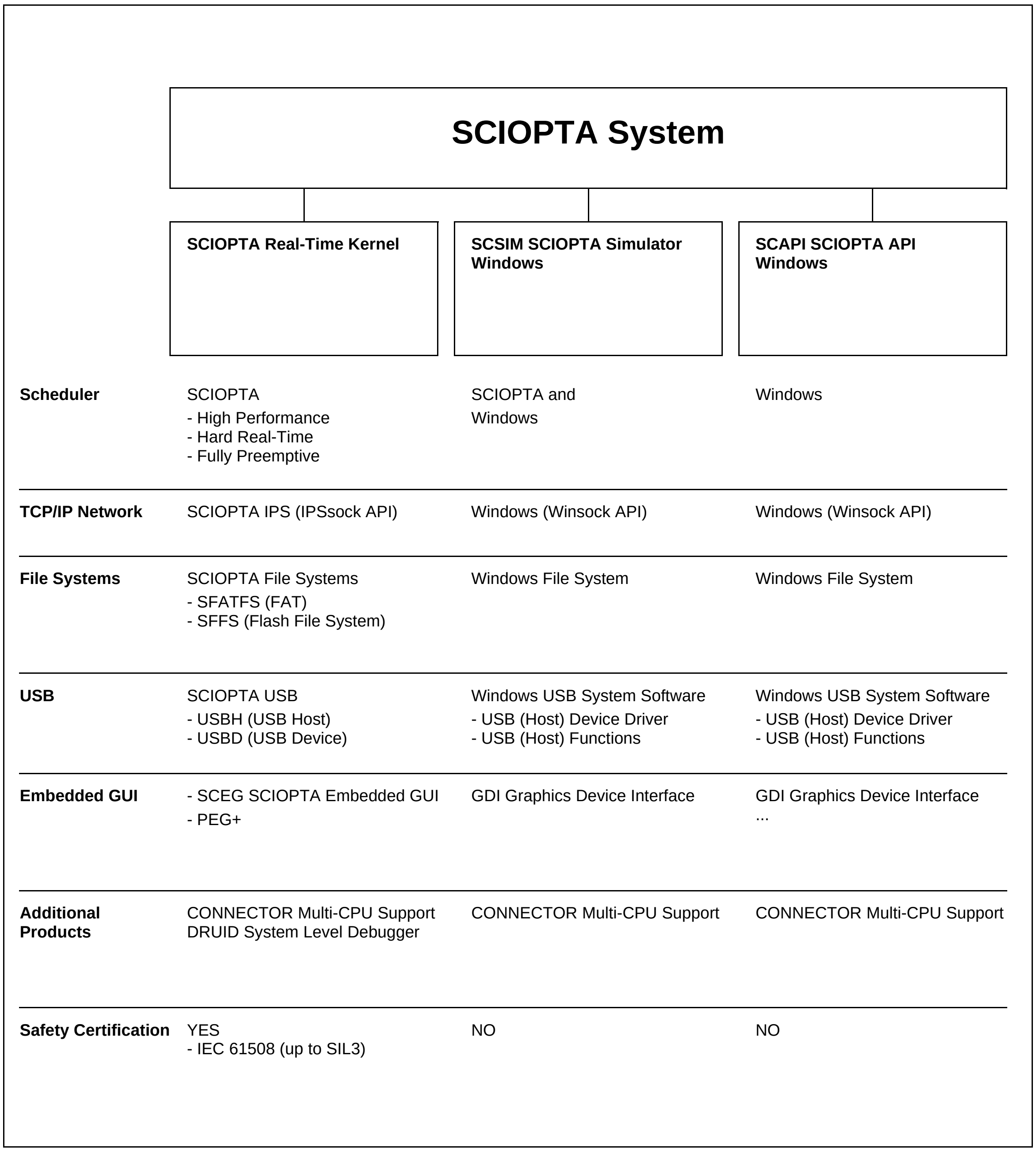
Headquarters

SCIOPTA Systems AG
Fiechthagstrasse 19
4103 Bottmingen
Switzerland
Tel. +41 61 423 10 62
Fax +41 61 423 10 63
email: sales@sciopta.com
www.sciopta.com

Document No. S09333RL1

Table of Contents

1 SCIOPTA System



	SCIOPTA Real-Time Kernel	SCSIM SCIOPTA Simulator Windows	SCAPI SCIOPTA API Windows
Scheduler	SCIOPTA - High Performance - Hard Real-Time - Fully Preemptive	SCIOPTA and Windows	Windows
TCP/IP Network	SCIOPTA IPS (IPSsock API)	Windows (Winsock API)	Windows (Winsock API)
File Systems	SCIOPTA File Systems - SFATFS (FAT) - SFFS (Flash File System)	Windows File System	Windows File System
USB	SCIOPTA USB - USBH (USB Host) - USBD (USB Device)	Windows USB System Software - USB (Host) Device Driver - USB (Host) Functions	Windows USB System Software - USB (Host) Device Driver - USB (Host) Functions
Embedded GUI	- SCEG SCIOPTA Embedded GUI - PEG+	GDI Graphics Device Interface	GDI Graphics Device Interface ...
Additional Products	CONNECTOR Multi-CPU Support DRUID System Level Debugger	CONNECTOR Multi-CPU Support	CONNECTOR Multi-CPU Support
Safety Certification	YES - IEC 61508 (up to SIL3)	NO	NO

Figure 1-1: The SCIOPTA System

1.1 The SCIOPTA System

1.1.1 SCIOPTA System

SCIOPTA System is the name for a SCIOPTA framework which includes 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 kernel 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.

1.1.2 SCIOPTA Real-Time Kernels

SCIOPTA System Framework together with specific SCIOPTA scheduler results in very high performance real-time operating systems for many CPU architectures. The kernels and scheduler are written 100% in assembler. SCIOPTA is the fastest real-time operating system on the market. 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.

1.1.3 SCIOPTA Simulator and API for Windows

The **SCIOPTA System** is available on top of Windows. **SCSIM** is a SCIOPTA simulator including SCIOPTA scheduling together with the Windows scheduler. This allows realistic system behaviour in a simulated environment.

SCAPI is a SCIOPTA API allowing message passing in a windows system. SCAPI is mainly used to design distributed systems together with CONNECTOR processes. Scheduling in SCAPI is done by the underlying operating system.

1.2 About This Manual

The purpose of this **SCIOPTA - Kernel V2, User´s Manual** is to give all needed information how to use the SCIOPTA real-time kernel V2 in an embedded project.

After a description of the installation procedure, detailed information about the technologies and methods used in the SCIOPTA V2 Kernel are given. Descriptions of Getting Started examples will allow a fast and smooth introduction into SCIOPTA.

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

Please consult also the **SCIOPTA - Kernel V2, Reference Manual** which contains a complete description of all system calls and error messages.

1.3 Supported Processors

1.3.1 Architectures

SCIOPTA - Kernel V2 supports the following processor architectures. Architectures are referenced in this document as **<arch>**):

- ppc (power pc)
- rx (Renesas RX600)

1.3.2 CPU Families

SCIOPTA - Kernel V2 supports the following CPU families. CPU families are referenced in this document as **<cpu>**):

Architecture **<arch>**	CPU Family **<cpu>**	Description
ppc	mpx5xx	**Freescale PowerPC MPC500** MPC53x, MPC55x, MPC56x and all other derivatives of the Freescale MPC500 family.
	mpc5500	**Freescale PowerPC MPC55xx** MPC5516, MPC5534, MPC5554, MPC5567 and all other derivatives of the Freescale MPC55xx family.
	mpc8xx	**Freescale PowerPC PowerQUICC I** MPC823, MPC850, MPC852T, MPC855T, MPC857, MPC859, MPC860, MPC862, MPC866 and all other derivatives of the Freescale MPC8xx family.
	mpc82xx	**Freescale PowerPC PowerQUICC II** MPC8250, MPC8255, MPC8260, MPC8264, MPC8265, MPC8266 and all other derivatives of the Freescale MPC82xx family.
	mpc83xx	**Freescale PowerPC PowerQUICC II Pro** MPC8313, MPC8314, MPC8315 and all other derivatives of the Freescale MPC83xx family.
	mpc52xx	**Freescale PowerPC MPC5200** MobileGT MPC5200 and all other derivatives of the Freescale MPC52xx and 51xx family.
	ppc4xx	**AMCC PowerPC 4xx** PowerPC 405, 440, 460 and all other derivatives of the AMCC PowerPC 4xx family.
rx	rx	**Renesas RX600** RX610, RX621, RX62N, RX62T and all other derivatives of the Renesas RX600 family.

1.4 SCIOPTA Real-Time Kernel V2

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

Differences between SCIOPTA Real-Time Kernel V2 and SCIOPTA Real-Time Kernel Standard:

- 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 full 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.
- Flag **SC_MSGTX_RTN2SND** introduced for **sc_msgTx** and **sc_msgTxAlias** system calls.
- 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 called.
    - **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_procWakeupDisable**. Disables the wakeup of a timer or interrupt process.
    - **sc_procWakeupEnable**. Enables the wakeup of a timer or interrupt process.
    - **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 Installation

2.1 Introduction

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

2.2 The SCIOPTA Delivery

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

- CD-ROM containing SCIOPTA for your CPU family.
- Installation password.
- Manuals of your installed products:
    - SCIOPTA - Kernel V2, User’s Manual (this document).
    - SCIOPTA - Kernel V2, Reference Manual.
    - SCIOPTA - Device Driver, User’s and Reference Manual.
    - SCIOPTA - DRUID, User’s and Reference Manual.
    - SCIOPTA - IPS Internet Protocols, User’s and Reference Manual.
    - SCIOPTA - IPS Internet Protocols Applications, User’s and Reference Manual.
    - SCIOPTA - FAT File System, User’s and Reference Manual.
    - SCIOPTA - FLASH Safe File System, User’s and Reference Manual.
    - SCIOPTA - USB Device, User’s and Reference Manual.
    - SCIOPTA - USB Host, User’s and Reference Manual.
    - SCIOPTA - PEG+, User’s and Reference Manual.
    - SCIOPTA - SMMS Memory Protection, User’s and Reference Manual.
    - SCIOPTA - CONNECTOR, User’s and Reference Manual.

2.3 System Requirements

2.3.1 Windows

Personal Computer or Workstation with:

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

2.3.2 Linux

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

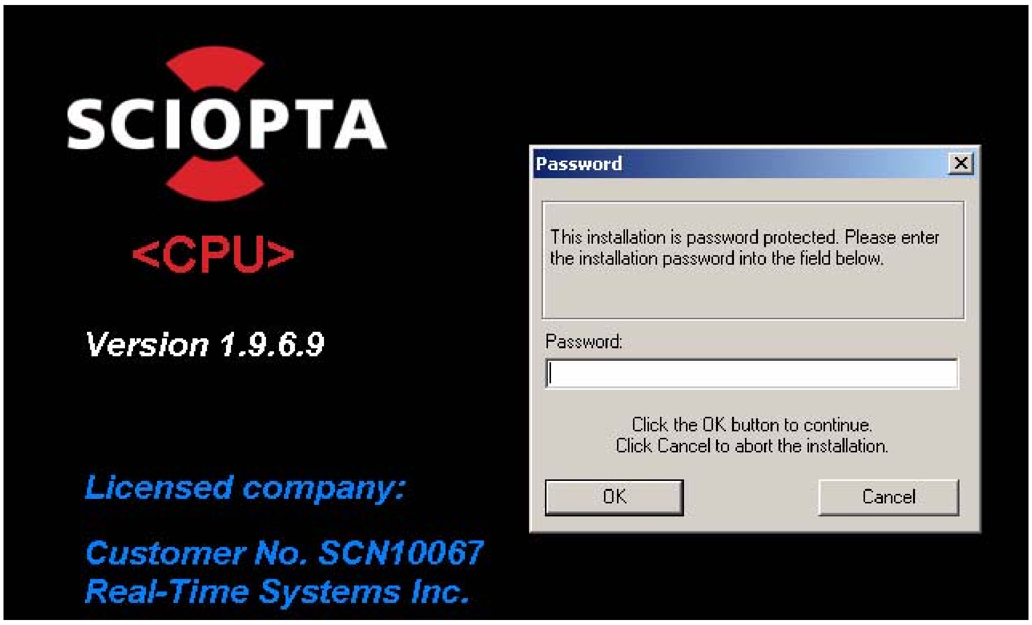
2.4 Installation Procedure Windows Hosts

2.4.1 Main Installation Window

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

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

The following main installation window will appear on your screen:


SCIOPTA

<CPU>

Version 1.9.6.9

Licensed company:

Customer No. SCN10067
Real-Time Systems Inc.

Password

This installation is password protected. Please enter the installation password into the field below.

Password:

Click the OK button to continue.
Click Cancel to abort the installation.

Figure 2-1: Main Installation Window

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

The program will guide you through the installation process.

2.4.2 Product Versions

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

"X.Y.Z.F"

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

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

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

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

2.4.3 Installation Location

The SCIOPTA products will be installed at the following location:

<Destination Folder>\sciopta\<version> (in this manual also referred as <install_folder>\sciopta\<version>)

The expression <version> stands for the SCIOPTA four digit version number (e.g. 1.9.6.9)

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

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

2.4.4 Release Notes

This SCIOPTA Kernel delivery for your CPU family includes a text file named RN_<cpu_family>_KRN.txt which contains a description of the changes of the actual version compared to the last delivered version. It allows you to decide if you want to install and use the new version.

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

File location: <installation_folder>\sciopta\<version>\doc\

2.4.5 Short Cuts

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

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

2.4.6 SCIOPTA_HOME Environment Variable

The **SCIOPTA** system building process needs the **SCIOPTA_HOME** environment variable to be defined.

Please define the **SCIOPTA_HOME** environment variable and set it to the following value:

```
<install_folder>\sciopta\<version>
```

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

2.4.7 Setting SCIOPTA Path Environment Variable

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

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

Please include

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

in your **path** environment variable.

2.4.8 Uninstalling SCIOPTA

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

For each **SCIOPTA** product use the following procedure:

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

Please Note:

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

2.4.9 GNU Tool Chain Installation

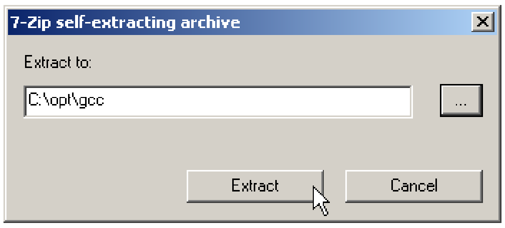
SCIOPTA is supporting the GNU Tool Chain. The Sourcery G++ Lite Edition from CodeSourcery is directly supported. The Lite Edition contains only command-line tools and is available at no cost. A ready to install version is available from SCIOPTA.

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

- GNU C & C++ Compilers for the specific CPU family
- GNU Assembler and Linker
- GNU C & C++ Runtime Libraries

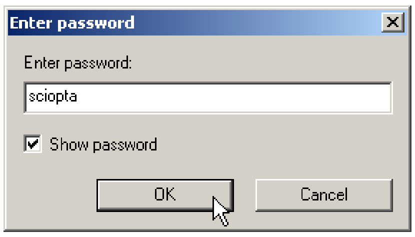
Run the compiler installation file which can be found on the SCIOPTA CD.

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


7-Zip self-extracting archive

Extract to: C:\opt\gcc

The installer asks for password. Please enter "sciopta".


Enter password

Enter password: sciopta

[✓] Show password

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

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

2.4.10 Eclipse IDE for C/C++ Developers.

The **Eclipse IDE for C/C++ Developers** project provides a fully functional C and C++ Integrated Development Environment (IDE).

Please consult http://www.eclipse.org/ for more information about Eclipse. You can download **Eclipse IDE for C/C++ Developers** from the download page of this site.

Please consult http://www.eclipse.org/cdt for more information about Eclipse CDT (C/C++ Development Tools) project.

For all delivered SCIOPTA examples there are Makefiles included. Eclipse is easy to configure for working with external makefiles. Please consult chapter 3 "Getting Started" on page 3-1 for a detailed description how to setup Eclipse to build SCIOPTA systems.

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

3 Getting Started

3.1 Introduction

These is a small tutorial example which gives you a good introduction into typical SCIOPTA systems and products. It would be a good starting point for more complex applications and your real projects.

3.2 Example Description

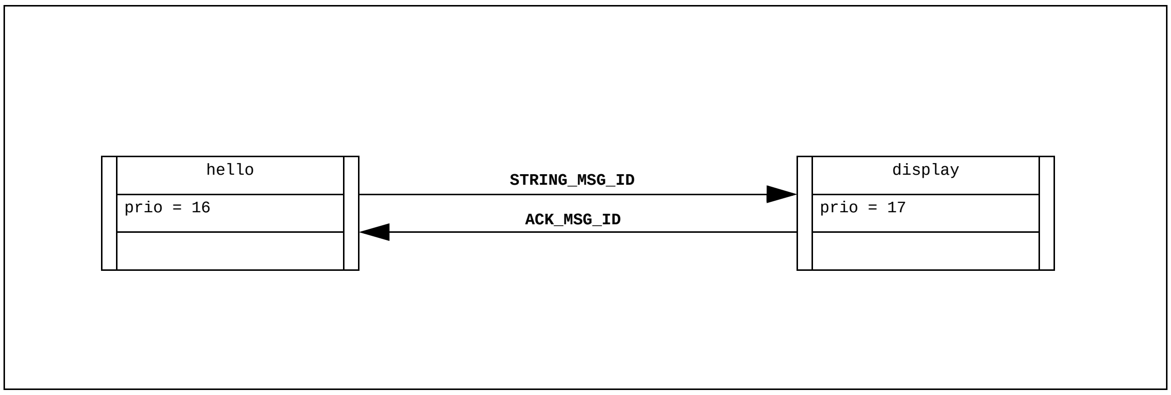


Figure 3-1: Process-Message Diagram of a Simple Hello World Example

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

After all four messages have been sent process **hello** sleeps for a while and restarts the whole cycle again for ever. Each message is received, displayed and freed by process **display**. Process **display** sends back an acknowledge message (**ACK_MSG_ID**) for every received message.

3.3 Getting Started Eclipse and GNU GCC

This is a getting started project including a step-by-step tutorial for the SCIOPTA Real-Time Kernels (<arch>=ppc). Not yet available for Renesas RX600 (<arch>=rx).

3.3.1 Equipment

For the PowerPC architecture the following equipment is used to run this getting started example:

- Microsoft® Windows Personal Computer or Workstation.
- Compiler package:

  For **PowerPC** CodeSourcery GNU C & C++ Sourcery G++ Lite Edition for Power PC. Architecture (arch): ppc

  This package can be found on the SCIOPTA CD.
- A debugger/emulator for the CPU on your target board which supports GNU GCC such as the iSYSTEM winIDEA Emulator/Debugger or the Lauterbach TRACE32 debugger.
- Target board which is supported by SCIOPTA examples. For each supported board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\.
- SCIOPTA - Real-Time Kernel for your selected architecture.
- Eclipse IDE for C/C++ Developers. You can download Eclipse from here http://www.eclipse.org/.
- In order to run the Eclipse Platform you also need the Sun Java 2 SDK, Standard Edition for Microsoft Windows.
- This getting started example is sending some specific example messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board. The selected UART port can be found in chapter 14 "Board Support Packages" on page 14-1 where the **Log Port** for each board is given. If you want to change the port you may modify the files system.c, simple_uart.c, simple_uart.h and config.h.

3.3.2 Step-By-Step Tutorial

This is a step-by-step tutorial for the SCIOPTA Real-Time Kernels (<arch>=ppc)

1. Check that the environment variable SCIOPTA_HOME is defined as described in chapter 2.4.6 "SCIOPTA_HOME Environment Variable" on page 2-4.
2. Be sure that the GNU GCC compiler bin directory is included in the PATH environment variable as described in chapter 2.4.9 "GNU Tool Chain Installation" on page 2-5.
3. Be sure that the SCIOPTA \win32\bin directory is included in the PATH environment variable as described in chapter 2.4.7 "Setting SCIOPTA Path Environment Variable" on page 2-4.
4. Create a project folder to hold all project files (e.g. d:\myprojects\sciopta) if you have not already done it for other getting-started projects.
5. Launch Eclipse. The Workspace Launcher window opens.
6. Select your created project folder (e.g. c:\myproject\sciopta) as your workspace (by using the Browse button).
7. Click the **OK** button. The workbench windows opens.
8. Close the "Welcome" window.

9. Deselect **“Build Automatically”** in the **Project** menu.
10. Open the **New Project** window (menu: **File -> New -> C Project**). We will create Makefile project.
11. Click the **Finish** button. You will see the krn_hello project folder in the Project Explorer window.
12. The next steps we will executed outside Eclipse.
13. Copy the script **copy_files.bat** from the example directory of your selected target board:

    <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\<board>\

    to your project folder.
14. Open a command window (windows cmd.exe) and go to your project folder.
15. Type copy_files to execute the **copy_files.bat** batch file. All needed project files will be copied from the delivery to your project folder.
16. Close the command window and return to Eclipse.
17. Swap back to the Eclipse workbench. Make sure that the kernel hello project (**krn_hello**) is highlighted.
18. Type the F5 key (or menu: **File > Refresh**) to refresh the project.
19. Expand the project by clicking on the **[+]** button and make sure that the **krn_hello** project is highlighted.
20. Now you can see all files in the Eclipse Navigator window.
21. Get the board number (**<board_number>)** for the makefile. This number (**BOARD_SEL**) can be found in chapter 14 “Board Support Packages” on page 14-1 where it is given for each board.
22. Click on the krn_hello project in the Project Explorer window to make sure that the kernel hello project (**krn_hello**) is highlighted.
23. Open the **Properties** window (menu: **File -> Properties** or **Alt+Enter** button).
24. Click on “C/C++ Build.
25. Deselect “Use default build command” in the Builder Settings Tab. Now you can enter a customized Build command.
26. Enter the following Build command: **gnu-make BOARD_SEL=<board_number> V=1**

    Enter the retrieved board number (**<board_number>)** as option of the make call.
    For example: **gnu-make BOARD_SEL=5 V=1** will build the project for board number 5.
27. Click the **OK** button.
28. Activate the Console window at the bottom of the Eclipse workbench to see the project building output.
29. Be sure that the project (**krn_hello**) is high-lighted and build the project (menu: **Project > Build Project** or **Build** button).
30. The executable (**sciopta.elf**) will be created in the debug folder of the project.

31. Launch your source-level emulator/debugger and load the resulting sciopta.elf.
32. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. Parameters are 115200Bd, 8 Bit, no parity, 1 stop bit, no flow-control.
33. For some emulators/debuggers specific project and board initialization files can be found in the created project folder or in other example directories.
34. Now you can start the system and check the log messages on your host terminal window.
35. You can also set breakpoints anywhere in the example system and watch the behaviour.

3.3.3 Please Note

- Study carefully the makefile to get a good understanding of the whole build process and the needed files in a specific SCIOPTA project.
- The makefile calls the SCIOPTA configuration utility directly (command line version, see chapter 16.19 “Command Line Version” on page 16-33). In a standard project you will rather use the normal GUI version as the utility is not used very often and you should not directly edit the sconf.xml file.
- The V=1 switch in the build call is just used to show you the full build information.

3.4 Getting Started iSYSTEM winIDEA

This is a getting started project including a step-by-step tutorial for the SCIOPTA Real-Time Kernels (<arch>=ppc).

3.4.1 Equipment

For the PowerPC architecture the following equipment is used to run this getting started example:

- Microsoft® Windows Personal Computer or Workstation.
- Compiler package:

  For **PowerPC** CodeSourcery GNU C & C++ Sourcery G++ Lite Edition for Power PC . Architecture (arch): ppc

  These packages can be found on the SCIOPTA CD.
- iSYSTEM winIDEA Emulator/Debugger for your selected target processor.
- Target board which is supported by SCIOPTA examples. For each supported board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\.
- SCIOPTA - Real-Time Kernel for your selected architecture.
- This getting started example is sending some specific example messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board. The selected UART port can be found in chapter 14 “Board Support Packages” on page 14-1 where the **Log Port** for each board is given. If you want to change the port you may modify the files system.c, simple_uart.c, simple_uart.h and config.h.

3.4.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in the chapter 2.4.6 “SCIOPTA_HOME Environment Variable” on page 2-4.
2. Be sure that the GNU GCC compiler bin directory is included in the PATH environment variable as described in the chapter 2.4.9 “GNU Tool Chain Installation” on page 2-5.
3. Create an example working directory at a suitable place.
4. Copy the script **copy_files.bat** from the example directory for your selected target boards:

   <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\<board>\

   to your project folder.
5. Double click **copy_files.bat** to execute the batch file. All needed project files will be copied from the delivery to your project folder.

6. Launch the SCIOPTA configuration utility **SCONF** from the desktop or the Start menu.
7. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF**.
   File > **Open**
8. Click on the **Build All** button or press **CTRL-B** to build the kernel configuration files.

   The following files will be created in your project folder:

   - sconf.c
   - sconf.h.
9. Launch the iSYSTEM - winIDEA Emulator/Debugger.
10. Open the example workspace (menu: **File** > **Workspace** > **Open Workspace...**). Browse to your example project directory and select the workspace file <project_file_name>.jrf.
11. Make the project (menu: **Project** > **Make**) or type the **F7** button.
12. The executable (sciopta.elf) will be created in the debug folder of the project.
13. Download the sciopta.elf file into the target system (menu: **Debug** > **Download**) or type the **Ctrl+F3** button.
14. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. Parameters are 115200Bd, 8 Bit, no parity, 1 stop bit, no flow-control.
15. Run the system (menu: **Debug** > **Run**) or type the **F5** button.
16. Now you can check the log messages on your host terminal window.
17. You can also set breakpoints anywhere in the example system and watch the behaviour.

3.5 Getting Started IAR Systems Embedded Workbench

This is a getting started project including a step-by-step tutorial for the SCIOPTA Real-Time Kernels (<arch>=rx).

3.5.1 Equipment

For architectures Renesas RX600 the following equipment is used to run this getting started example:

- Microsoft® Windows Personal Computer or Workstation.
- Compiler package:

For **RX**	IAR Embedded Workbench for RX. Architecture (arch): rx

- IAR CSpy Debugger for your selected target processor.
- Target board which is supported by SCIOPTA examples. For each supported board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\.
- SCIOPTA - Real-Time Kernel for your selected architecture.
- This getting started example is sending some specific example messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board. The selected UART port can be found in chapter 14 “Board Support Packages” on page 14-1 where the **Log Port** for each board is given. If you want to change the port you may modify the files system.c, simple_uart.c, simple_uart.h and config.h.

3.5.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in chapter 2.4.6 “SCIOPTA_HOME Environment Variable” on page 2-4.
2. Be sure that the SCIOPTA \win32\bin directory is included in the PATH environment variable as described in chapter 2.4.7 “Setting SCIOPTA Path Environment Variable” on page 2-4. This will give access to the sconf.exe utility. Some IAREW examples might call sconf.exe directly from the workbench to do the SCIOPTA configuration.
3. Be sure that the IAR Embedded Workbench is installed as described in the IAREW user manuals.
4. Create an example working directory at a suitable place.
5. Copy the script **copy_files_iar.bat** from the example directory for your selected target boards:

   <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\<board>\

   to your project folder.
6. Double click **copy_files.bat** to execute the batch file. All needed project files will be copied from the delivery to your project folder.

7. Launch the SCIOPTA configuration utility **SCONF** from the desktop or the Start menu.
8. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF**.
   File > **Open**
9. Choose the **Build All** button or press **CTRL-B** to build the kernel configuration files.

   The following files will be created in your project folder:

   - sconf.c
   - sconf.h.
10. Launch the IAR Embedded Workbench.
11. Click on the **Open existing workbench** button in the Embedded Workbench Startup window.
12. Browse to your example project directory and select the IAR Embedded Workbench file for your selected board: **<file_name>.eww**.
13. Select the project in the Workspace and Make the project (menu: **Project** > **Make**) or type the **F7** button.
14. The executable (sciopta.elf) will be created in the Output folder of the project.
15. Download and debug the sciopta.elf file into the target system (menu: **Project** > **Debug**) or type the **Ctrl+D** button.
16. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. Parameters are 115200Bd, 8 Bit, no parity, 1 stop bit, no flow-control.
17. Run the system (menu: **Debug** > **Go**) or type the **Go** button.
18. Now you can check the log messages on your host terminal window.
19. You can also set breakpoints anywhere in the example system and watch the behaviour.

4 Modules

4.1 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. Modules can be moved and copied between CPUs and systems

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.

4.2 System Module

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

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

4.3 Module Priority

SCIOPTA modules contain a (module) 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.

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

4.4 Module Memory

The module start address and the module size must be given when declaring a static module in the SCIOPTA configuration utility (SCONF) or as parameters when dynamically creating a module.

The best method to handle module addresses is to use the linker script to calculate the module address boundaries. Please consult chapter 15.8.2.2 “Module Sizes” on page 15-9 for more information.

4.5 System Protection

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

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

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

4.6 Creating Modules

4.6.1 Static Module Creation

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

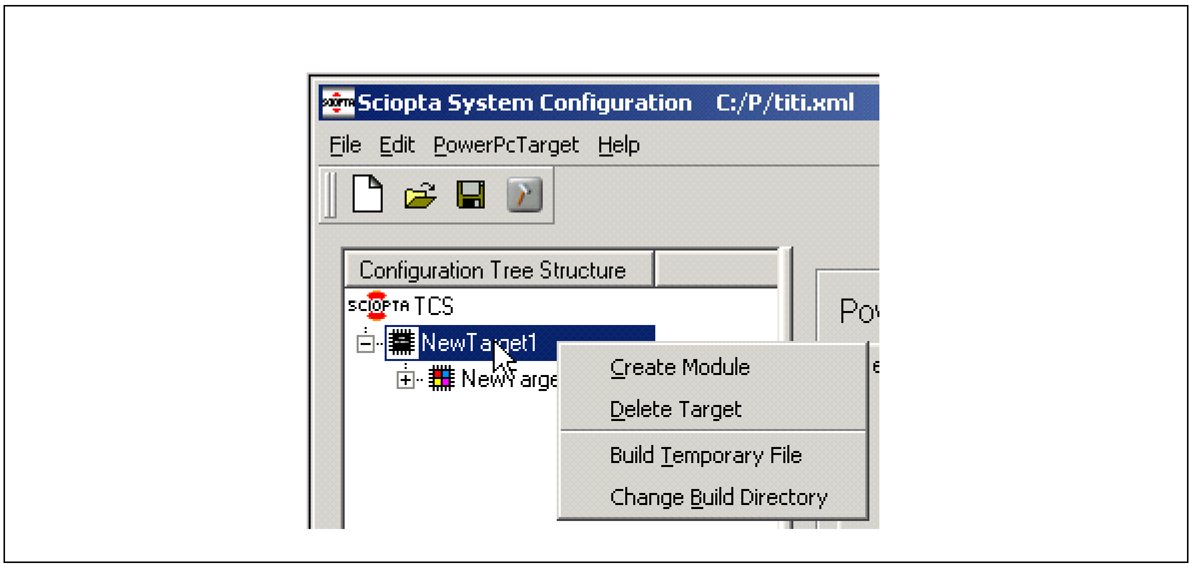

Sciopta System Configuration C:/P/titi.xml
File Edit PowerPcTarget Help
Configuration Tree Structure
TCS
NewTarget1
NewTarge
Create Module
Delete Target
Build Temporary File
Change Build Directory

Figure 4-1: Module Creation by SCONF

Please consult chapter 16.10 “Creating Modules” on page 16-15 for more information about module creation by the **SCONF** tool.

4.6.2 Dynamic Module Creation

Another way is to create modules dynamically by the **sc_moduleCreate2()** system call.

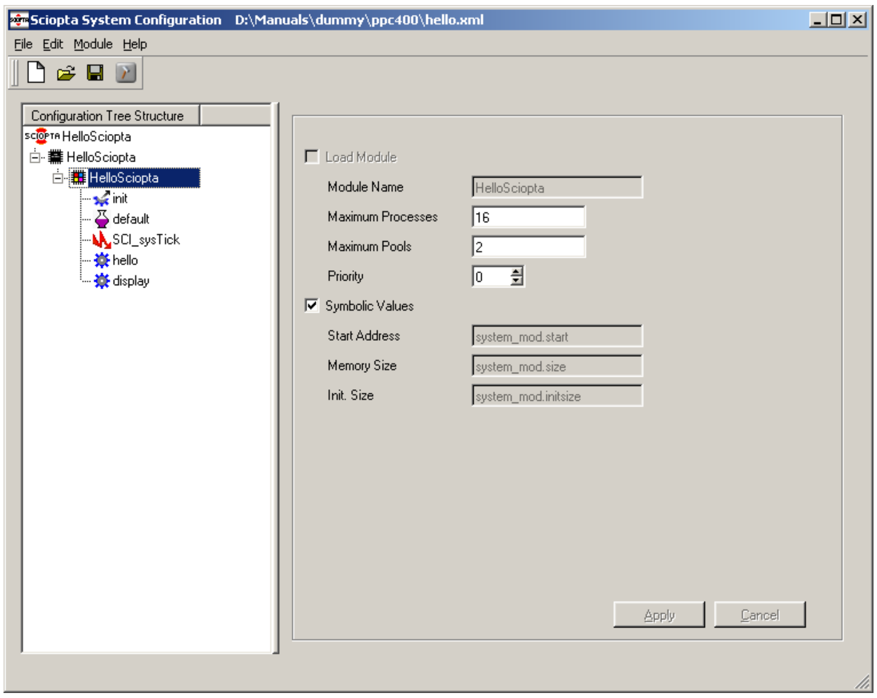
```
sc_moduleid_t sc_moduleCreate2 (
    sc_mdb_t *mdb      // Pointer to the module descriptor block
);
```

Figure 4-2: Dynamic Module Creation

4.7 Module Layout Examples

4.7.1 Small Systems

Small or simple system can be put into one module. This keeps the system and memory map on a very neat level.



Load Module: ☐

Module Name	HelloSciopta
Maximum Processes	16
Maximum Pools	2
Priority	0

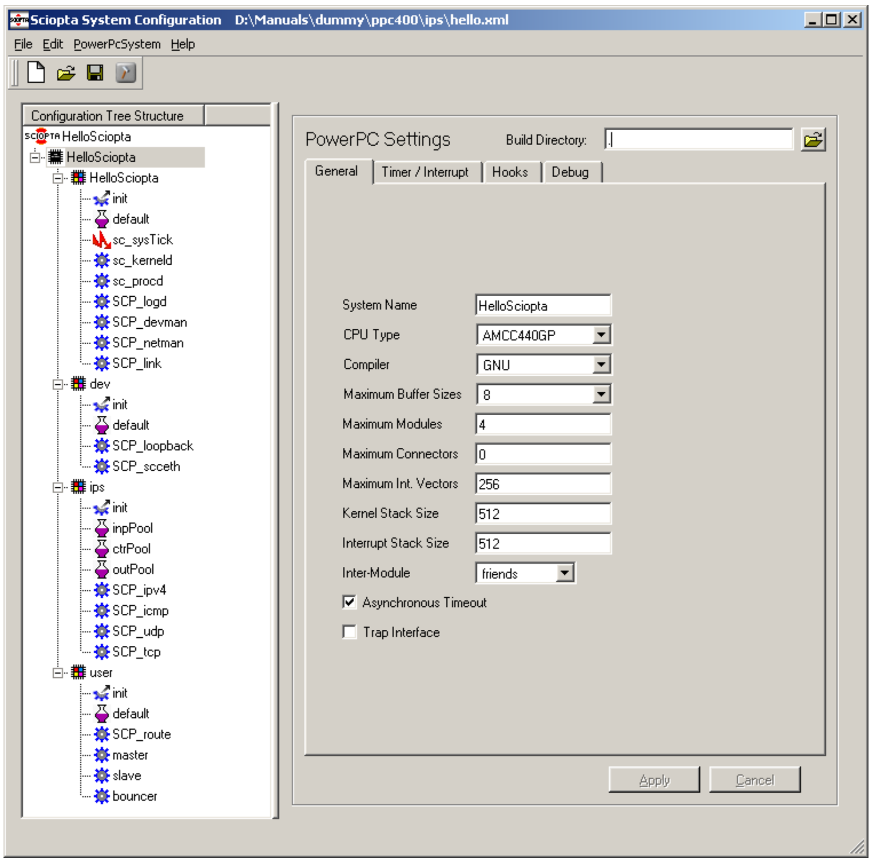
Symbolic Values: ☑

Start Address	system_mod.start
Memory Size	system_mod.size
Init. Size	system_mod.initsize

Figure 4-3: One-Module System

4.7.2 Multi-Module Systems

In larger or more complex system it is good design practice to partition the system up into more modules.


Sciopta System Configuration D:\Manuals\dummy\ppc400\ips\hello.xml
File Edit PowerPcSystem Help
Configuration Tree Structure
PowerPC Settings
Build Directory:
General
Timer / Interrupt
Hooks
Debug
Apply
Cancel

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      sc_sysTick
      sc_kerneld
      sc_procd
      SCP_logd
      SCP_devman
      SCP_netman
      SCP_link
    dev
      init
      default
      SCP_loopback
      SCP_scceth
    ips
      init
      inpPool
      ctrPool
      outPool
      SCP_ipv4
      SCP_icmp
      SCP_udp
      SCP_tcp
    user
      init
      default
      SCP_route
      master
      slave
      bouncer
```

Build Directory:	.
System Name	HelloSciopta
CPU Type	AMCC440GP
Compiler	GNU
Maximum Buffer Sizes	8
Maximum Modules	4
Maximum Connectors	0
Maximum Int. Vectors	256
Kernel Stack Size	512
Interrupt Stack Size	512
Inter-Module	friends
Asynchronous Timeout	✓
Trap Interface	☐

Figure 4-4: Multi-Module System

Above example system consists of four modules.

HelloSciopta	This is the system module and contains the daemons (kernel daemon sc_kerneld, process daemon sc_procd and log daemon SCP_logd), the SDD managers (device manager SCP_devman and network device manager SCP_netman) and other system pools and system processes. The system module gets automatically the name of the system.
dev	This module holds the device driver processes and device driver pools.
ips	This example includes the SCIOPTA IPS TCP/IP stack. The processes of this communication stack are located in the ips module.
user	In this user module the application processes and pools are placed.

4.8 Module System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_moduleCreate2	Creates a module.
sc_moduleIdGet	Returns the ID of a module.
sc_moduleInfo	Returns a snap-shot of a module control block.
sc_moduleKill	Kills a module.
sc_moduleNameGet	Returns the full name of a module.
sc_modulePrioGet	Returns the priority of a module.
sc_moduleStop	Stops a whole module.

5 Processes

5.1 Introduction

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

All SCIOPTA processes have system wide unique process identities.

A SCIOPTA process is always part of a SCIOPTA module. Please consult chapter 4 “Modules” on page 4-1 for more information about SCIOPTA modules.

5.2 Process States

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

5.2.1 Running

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

5.2.2 Ready

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

5.2.3 Waiting

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

- The process tried to receive a message which has (not yet) arrived.
- The process called the sleep system call and waits for the delay to expire.
- The process waits on a SCIOPTA trigger.
- The Process waits on a start system call if it was previously stopped.

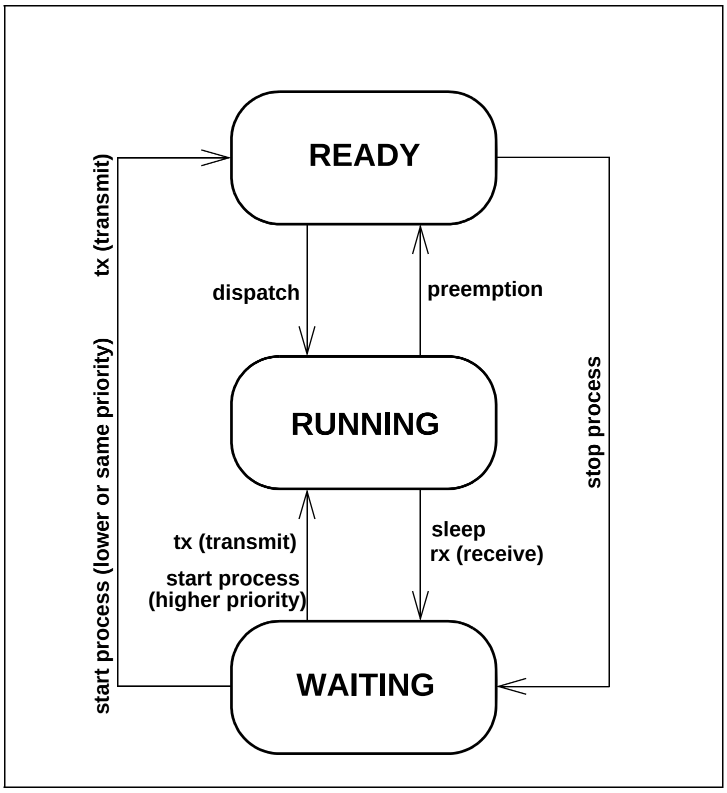
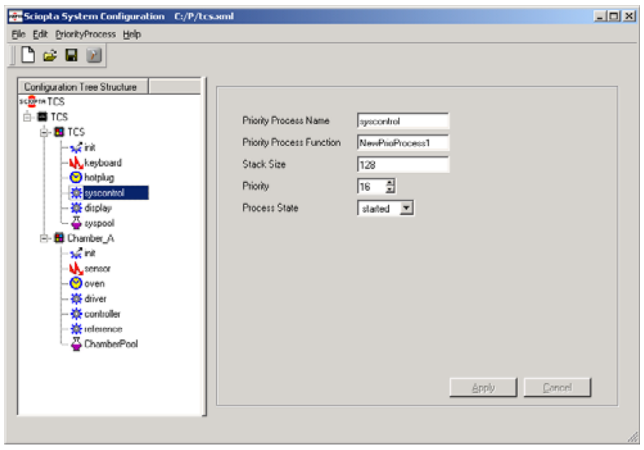


Figure 5-1: State Diagram of SCIOPTA Kernel

5.3 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 such as priority and process stack sizes. At start-up the kernel puts all static created processes into READY or WAITING (stopped) state.

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



Priority Process Name: syscontrol
Priority Process Function: NewPrioProcess1
Stack Size: 128
Priority: 16
Process State: started

Figure 5-2: Process Configuration Window for Static Processes

5.4 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 and does not to be known before running the system.

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

```
sc_pid_t sc_procPrioCreate2(
             sc_pdb_t *pdb,
             int state,
             sc_poolid_t plid
);
```

Figure 5-3: Create Process System Call

5.5 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 a **sc_procIdGet** system call including the process name.

5.6 Prioritized Processes

In SCIOPTA a process can be seen as an independent program which executes as if it has the whole CPU available. The operating systems guarantees that always the most important process at a certain moment is executing. In a typical SCIOPTA system prioritized processes are the most common used process types. Each prioritized process has a priority and the SCIOPTA scheduler is running ready processes according to these priorities. The process with higher priority 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.

5.6.1 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. See also chapter 16.12 “Creating Processes and Pools” on page 16-18.

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

5.6.2 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.

The process priority cannot be higher than the module priority of the module where the process resides. See also chapter 4.3 “Module Priority” on page 4-1.

5.6.3 Writing Prioritized Processes

5.6.3.1 Process Declaration Syntax

All prioritized processes in SCIOPTA must contain the following declaration:

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

5.6.3.2 Process Template

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

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

   /* Process initialization code */

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

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


      /* Code for process proc_name */


   }

}
```

5.7 Interrupt Processes

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

The 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.

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.

In some SCIOPTA systems there might be two type of interrupt processes. Interrupt processes of type **Sciopta** are handled by the kernel and may use (not blocking) system calls while interrupt processes of type **User** are handled outside the kernel and may not use system calls.

5.7.1 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 16.12 “Creating Processes and Pools” on page 16-18.

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

5.7.2 Interrupt Process Priorities

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

5.7.3 Writing Interrupt Processes

5.7.3.1 Interrupt Process Declaration Syntax

All interrupt processes in SCIOPTA must contain the following declaration:

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

5.7.3.2 Interrupt Source Parameter

The interrupt process declaration has beside the process name a second parameter (<irq_src> see chapter 5.7.3.1 “Interrupt Process Declaration Syntax” on page 5-5) which defines the interrupt source. This parameter is set by the kernel depending on the interrupt source.

Interrupt Source Parameter Values

0	The interrupt process is activated by a real hardware interrupt.
1	The interrupt process is activated by a message sent to the interrupt process.
2	The interrupt process is activated by a trigger event.
3	The interrupt process is activated when the process is started.
-1	The interrupt process is activated when the process is created. This allows the interrupt process to execute some initialization code.
-2	The interrupt process is activated when the process is killed. This allows the interrupt process to execute some exit code.
-3	The interrupt process is activated when the process is stopped.

5.7.3.3 Interrupt Process Template

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

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

SC_INT_PROCESS (proc_name, irq_src)/* Declaration for interrupt process proc_name */
{
   /* Local variables */
   if (irq_src == 0)          /* Generated by hardware */
   {
      /* Code for hardware interrupt handling */
   }
   else if (irq_src == -1)          /* Generated when process created */
   {
      /* Initialization code */
   }
   else if (irq_src == -2)       /* Generated when process killed */
   {
      /* Exit code */
   }
   else if (irq_src == 1)           /* Generated by a message sent to this interrupt process */
   {
      /* Code for receiving a message */
   }
   else if (irq_src == 2)           /* Generated by a SCIOPTA trigger event */
   {
      /* Code for trigger event handling */
   }
   else if (irq_src == -3)          /* Generated when process is stopped */
   {
      /* Stop code */
   }
   else if (irq_src == 3)           /* Generated when process is started */
   {
      /* Start code */
   }
}
```

5.8 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 derived from the operating system tick counter.

When configuring or creating a timer process, the user defines the number of system ticks to expire from one call to the other individually for each process.

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

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

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

5.8.1 Creating and Declaring Timer Processes

Static timer processes are defined in the SCIOPTA configuration utility (SCONF) and created by the kernel automatically at system startup. See also chapter 16.12 “Creating Processes and Pools” on page 16-18.

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

5.8.2 Timer Process Priorities

The priority of an interrupt process is assigned by hardware of the interrupt source which is used for the timer process. Whenever a timer interrupt occurs the assigned timer interrupt process is called, assuming that no other interrupt of higher priority is running.

5.8.3 Writing Timer Processes

Timer processes are written exactly the same way as interrupt processes. Please consult chapter 5.7.3 “Writing Interrupt Processes” on page 5-5 for information how to write interrupt processes.

5.9 Init Processes

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

At module start the init process gets automatically the highest priority (0). After the init process has done some its work it will change its 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.

5.9.1 Creating and Declaring Init Processes

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

In dynamic modules the init process is also created and started automatically. But the code of the init process must be written by the user. The entry point of the init process is given as parameter of the **sc_moduleCreate2** system call. Please see below for more information how to write init processes for dynamic modules.

5.9.2 Init Process Priorities

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

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

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

5.9.3 Writing Init Processes

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 **sc_moduleCreate2** system call. 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 */
}
```

5.10 Daemons

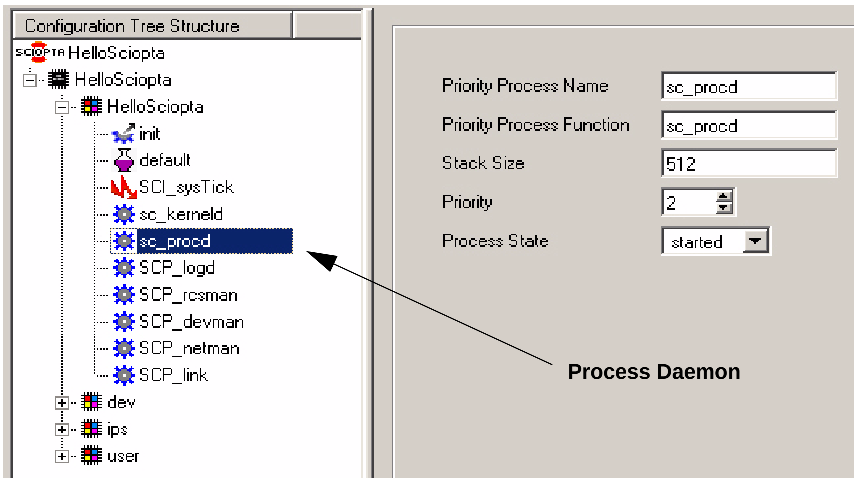
Daemons are internal processes in SCIOPTA and are structured the same way as ordinary processes. They have a process control block (pcb), a process stack and a priority.

Not all SCIOPTA daemons are part of the standard SCIOPTA delivery.

5.10.1 Process Daemon

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

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



Priority Process Name: sc_procd
Priority Process Function: sc_procd
Stack Size: 512
Priority: 2
Process State: started

Figure 5-4: Process Daemon Declaration in SCONF

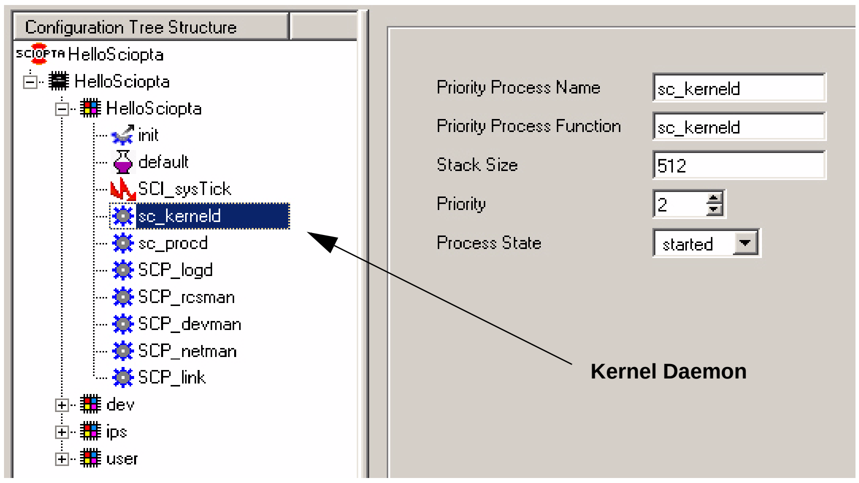
The process daemon is part of the kernel. But to use it you need to define and declare it in the **SCONF** configuration utility.

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

5.10.2 Kernel Daemon

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

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



Priority Process Name: sc_kerneld
Priority Process Function: sc_kerneld
Stack Size: 512
Priority: 2
Process State: started

Figure 5-5: 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 creared and placed in the system module.

5.11 Supervisor Processes

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

5.12 Process Stacks

When creating processes either statically in the SCONF configuration tool or dynamically with the **sc_procCreate2** system call 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 big as possible. In a later stage you can measure the used stack with the SCIOPTA DRUID system level debugger and reduce the stacks if needed.

5.13 Stack Protector

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 need 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 (see chapter 16.9.4 “Debug Configuration Tab” on page 16-13)

5.14 Addressing Processes

5.14.1 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.

5.14.2 Get Process IDs of Static Processes

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

You can address static process by appending the string

```
_pid
```

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

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

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

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

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

5.14.3 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 call **sc_procCreate2** will also return the process IDs which can be used for further addressing.

5.15 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.

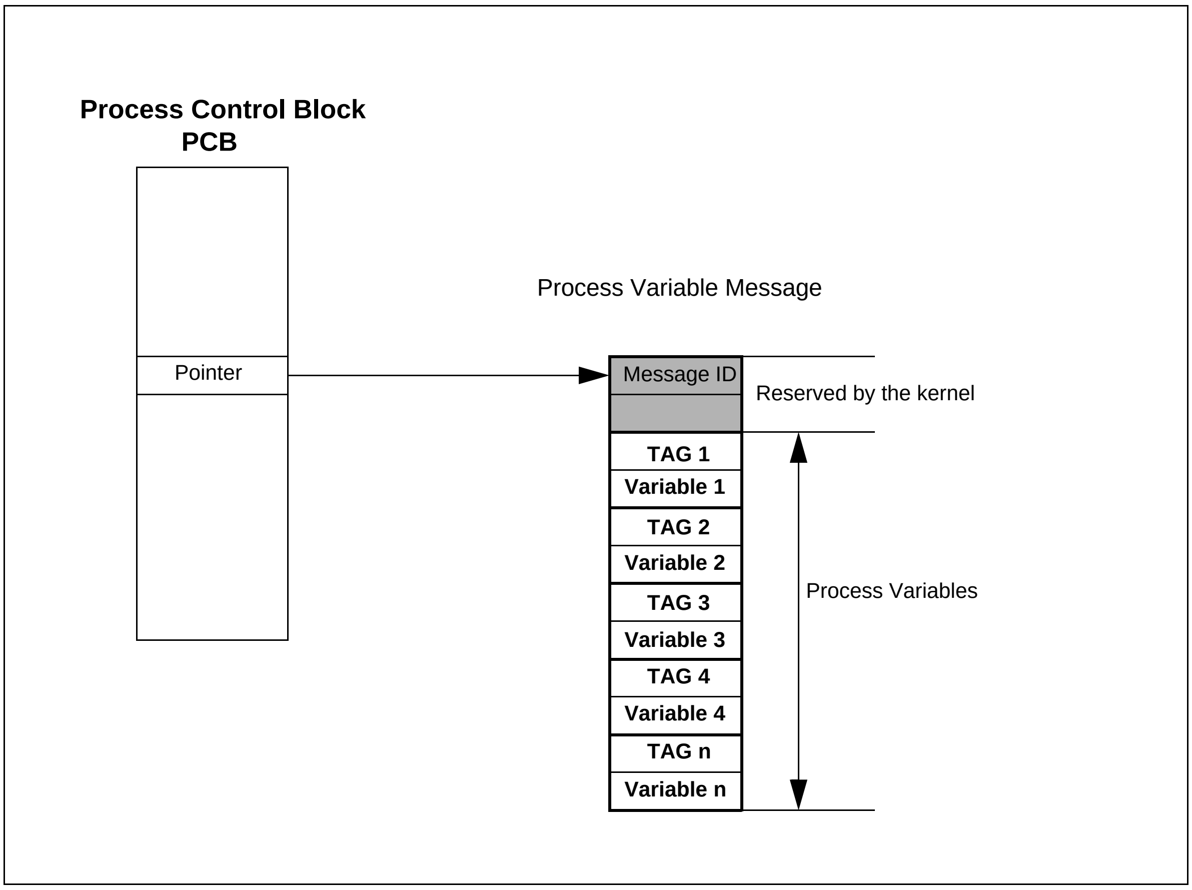


Figure 5-6: SCIOPTA Process Variables

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.

5.16 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.

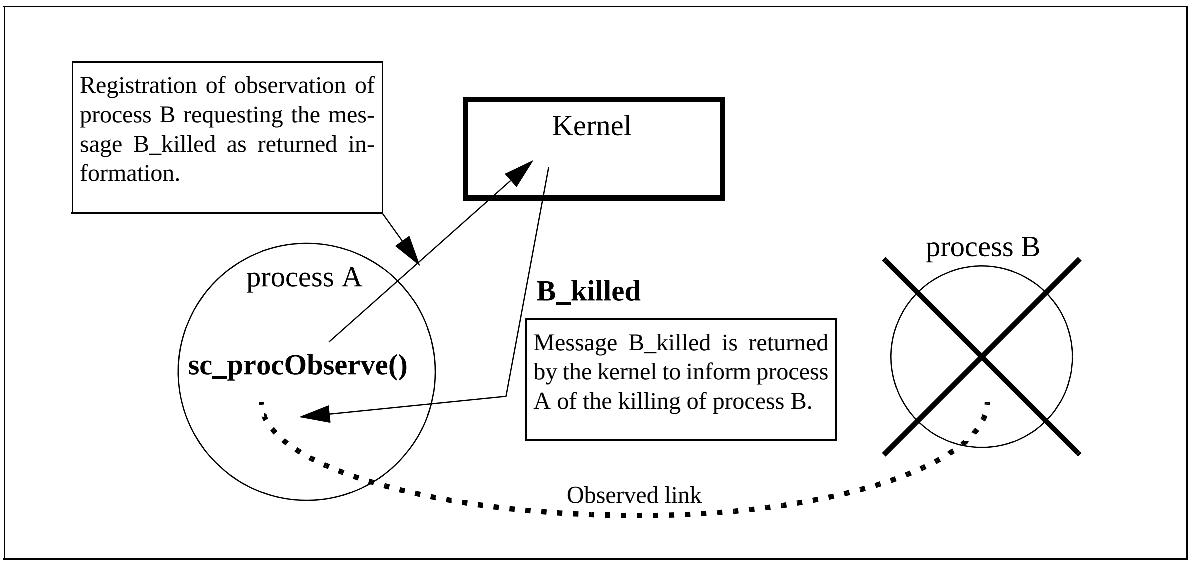

Registration of observation of process B requesting the message B_killed as returned information.
Kernel
process A
sc_procObserve()
B_killed
Message B_killed is returned by the kernel to inform process A of the killing of process B.
process B
Observed link

Figure 5-7: SCIOPTA Observation

5.17 Process System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_procAtExit	Register a function to be called if a prioritized process is called.
sc_procAttrGet	Returns specific process attributes.
sc_procCreate2	Requests the kernel daemon to create process.
sc_procDaemonRegister	Registers a process daemon which is responsible for pid get request.
sc_procDaemonUnregister	Unregisters a process daemon.
sc_procHookRegister	Registers a process hook.
sc_procIdGet	Returns the process ID of a process.
sc_procKill	Requests the kernel daemon to kill a process.
sc_procNameGet	Returns the full name of a process.
sc_procObserve	Request a message to be sent if the given process pid dies (process supervision).
sc_procPathCheck	Checks if the construction of a path is correct.
sc_procPathGet	Returns the path of a process.
sc_procPpidGet	Returns the process ID of the parent of a process.
sc_procPrioGet	Returns the priority of a process.
sc_procPrioSet	Sets the priority of a process.
sc_procSchedLock	Locks the scheduler and returns the number of times it has been locked before.
sc_procSchedUnlock	Unlocks the scheduler by decrementing the lock counter by one.
sc_procSliceGet	Returns the time slice of a timer process.
sc_procSliceSet	Sets the time slice of a timer process.
sc_procStart	Starts a process.
sc_procStop	Stops a process.
sc_procUnobserve	Cancels the observation of a process.
sc_procVarDel	Deletes a process variable.
sc_procVarGet	Returns a process variable.
sc_procVarInit	Initializes a process variable area.
sc_procVarRm	Removes a process variable area.
sc_procVarSet	Sets a process variable.
sc_procVectorGet	Returns the interrupt vector of an interrupt process.
sc_procWakeupEnable	Enables the wakeup of a timer or interrupt process.
sc_procWakeupDisable	Disables the wakeup of a timer or interrupt process.
sc_procYield	Yields the CPU to the next ready process within the current priority group.

6 Messages

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

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

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

6.2 Message Structure

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

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

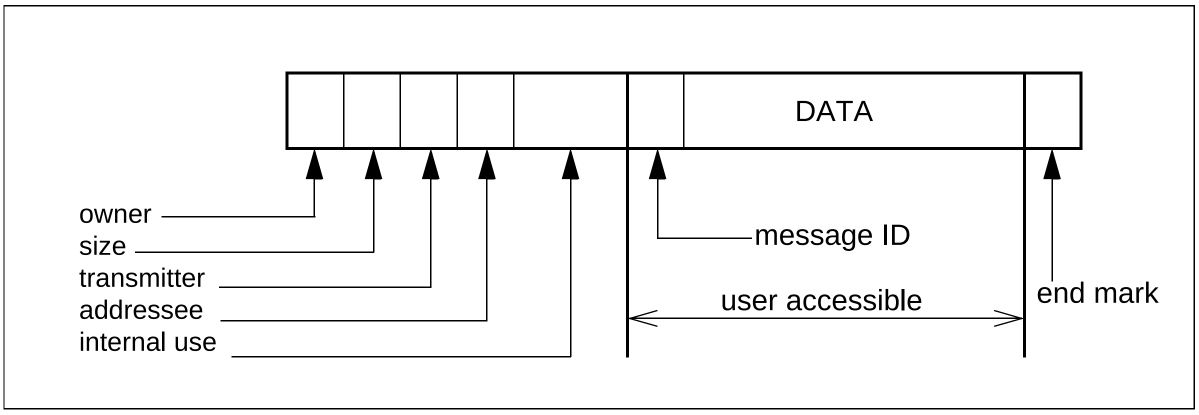


Figure 6-5: SCIOPTA Message Structure

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

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

6.3 Message Sizes

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

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

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

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

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

6.3.1 Example

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

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

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

6.4 Message Pool

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

Please consult chapter 7 “Pools” on page 7-1 for more information about message pools.

6.5 Message Passing

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

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

The size is given in bytes and the **sc_msgAlloc** function of SCIOPTA chooses an internal size out of a number of 4, 8 or 16 fixed sizes (see also chapter 6.3 “Message Sizes” on page 6-2).

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

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

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

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

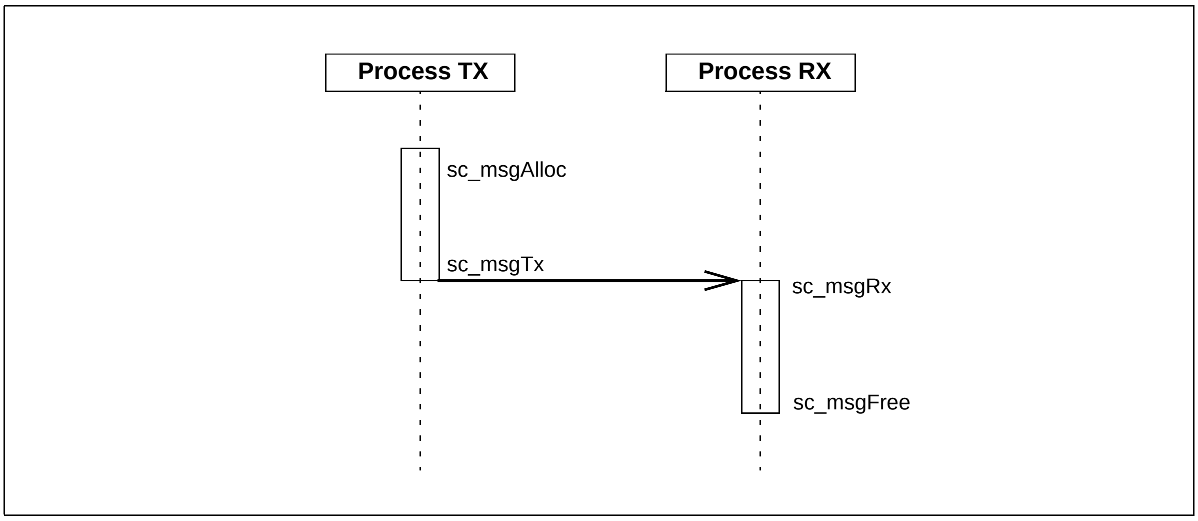


Figure 6-6: Message Sequence Chart of a SCIOPTA Message Passing

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

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

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

6.6.1 Message Number

6.6.1.1 Description

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

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

6.6.1.2 Syntax

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

6.6.1.3 Parameter

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

6.6.2 Message Structure

6.6.2.1 Description

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

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

6.6.2.2 Syntax

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

6.6.2.3 Parameter

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

6.6.3 Message Union

6.6.3.1 Description

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

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

6.6.3.2 Syntax

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

6.6.3.3 Parameter

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

6.7 Message Number (ID) organization

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

6.7.1 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\

6.8 Example

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

```
#define       CHAR_MSG         (5)

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

union     sc_msg
{
   sc_msgid_t             id;
   char_msg_t             char_msg;
};

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

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

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

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

6.9 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 (see chapter 16.9.1 “General System Configuration Tab” on page 16-8).

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.

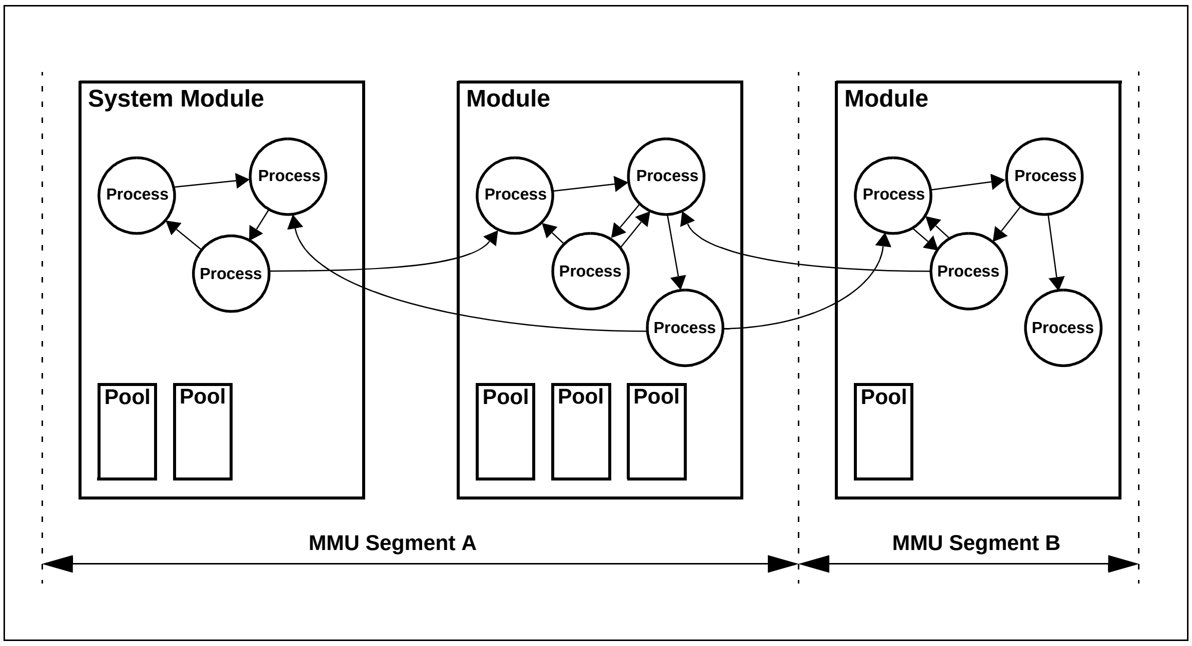


Figure 6-7: SCIOPTA Messages and Modules

6.10 Returning Sent Messages

There is a specific flag available (**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.

6.11 Message Passing and Scheduling

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

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

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

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

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

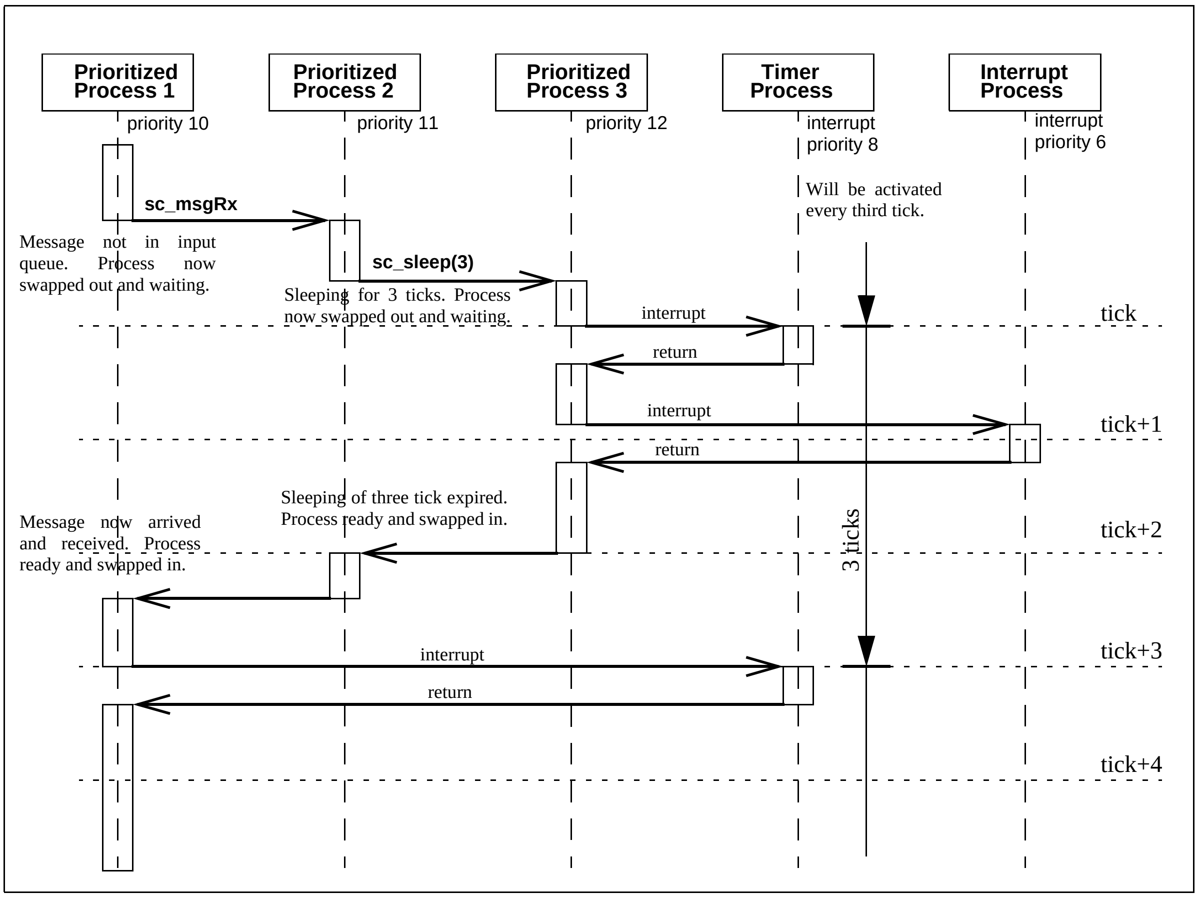

Prioritized Process 1
priority 10
Prioritized Process 2
priority 11
Prioritized Process 3
priority 12
Timer Process
interrupt priority 8
Interrupt Process
interrupt priority 6
Will be activated every third tick.
sc_msgRx
Message not in input queue. Process now swapped out and waiting.
sc_sleep(3)
Sleeping for 3 ticks. Process now swapped out and waiting.
interrupt
return
tick
interrupt
return
tick+1
Sleeping of three tick expired. Process ready and swapped in.
Message now arrived and received. Process ready and swapped in.
3 ticks
tick+2
interrupt
return
tick+3
tick+4

Figure 6-8: Scheduling Sequence Example

6.12 Message System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_msgAlloc	Allocates a memory buffer of selectable size from a message pool.
sc_msgAllocClr	Allocates a memory buffer of selectable size from a message pool and initializes the message data to 0.
sc_msgFind	Find a message in allocated queue.
sc_msgTx	Sends a message to a process.
sc_msgTxAlias	Sends a message to a process by setting any process ID.
sc_msgRx	Receives one ore more defined messages.
sc_msgFree	Returns a message to the message pool.
sc_msgAcquire	Changes the owner of the message. The caller becomes the owner of the message.
sc_msgAddrGet	Returns the process ID of the addressee of the message.
sc_msgHdCheck	The message header will be checked for plausibility.
sc_msgHookRegister	Registers a message hook.
sc_msgOwnerGet	Returns the process ID of the owner of the message.
sc_msgPoolIdGet	Returns the pool ID of a message.
sc_msgSizeGet	Returns the size of the message buffer.
sc_msgSizeSet	Modifies the size of a message buffer.
sc_msgSndGet	Returns the process ID of the sender of the message.

7 Pools

7.1 Introduction

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

There can be up to 127 pools per module for a standard kernel (32-bit) and up to 15 pools for a compact kernel (16-bit). Please consult chapter 4 "Modules" on page 4-1 for more information about the SCIOPTA module concept. The maximum number of pools will be defined at module creation. A message pool always belongs to the module from where it was created.

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

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

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

7.2 Message Pool size

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

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

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)

7.3 Pool Message Buffer Memory Manager

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

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

The 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.

7.3.1 Example

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

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

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

7.3.2 Message Administration Block

Each SCIOPTA message contains a hidden data structure which will be used by the kernel. The user can access these message information only by specific SCIOPTA system calls. Information such as the process ID of the message owner, the message size, the process ID of the transmitting process and the process ID of the addressed process are included in the message header administration block. Please consult chapter 5.3 “Messages” on page 5-7 for more information about SCIOPTA messages. The size of the message header is 32 bytes.

Each SCIOPTA message can contain an end-mark. This end-mark is used for the kernel message check if the message check option is enabled at kernel configuration. Please consult the configuration chapter of the SCIOPTA target manual for more information about message check. The size of the end-mark is 4 bytes.

Please consult chapter 6 “Messages” on page 6-1 for more information about message sizes and message memory management.

7.4 Creating Pools

7.4.1 Static Pool Creation

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

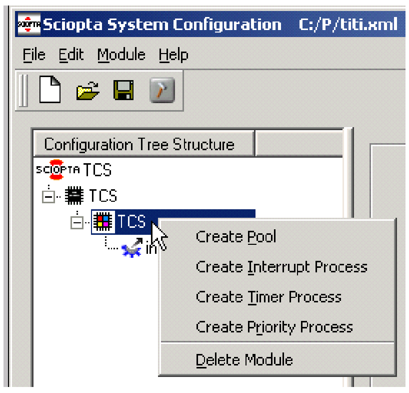


Figure 7-2: Pool Creation by SCONF

Please consult chapter 16.12 “Creating Processes and Pools” on page 16-18 for more information about module creation by the **SCONF** tool.

7.4.2 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 7-3: Dynamic Module Creation

7.5 Pool System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

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

8 SCIOPTA Trigger

8.1 Description

The trigger in SCIOPTA is a method which allows to synchronise processes even faster as it would be possible with messages. With a trigger a process will be notified and woken-up by another process. Trigger are used only for process co-ordination and synchronisation and cannot carry data. 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 basically a integer variable owned by the process. At process creation the value of the trigger is initialized to one.

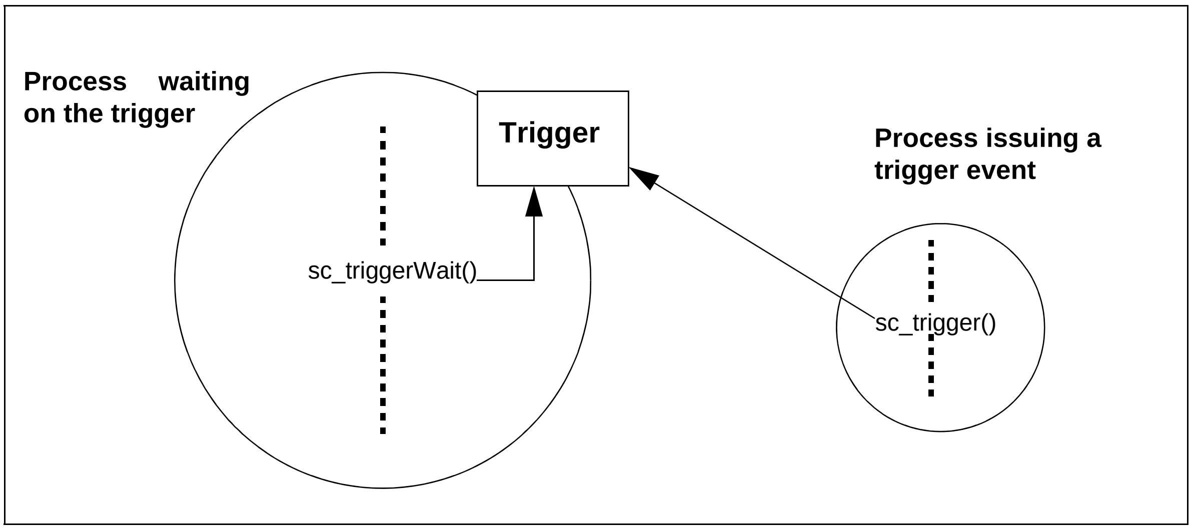

Process waiting on the trigger
Trigger
Process issuing a trigger event
sc_triggerWait()
sc_trigger()

Figure 8-1: SCIOPTA Trigger

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 zero. Only the owner process of the trigger can wait for it. An interrupt process cannot wait on its trigger. The process waiting on the trigger will become ready when another process triggers it by issuing a **sc_trigger** call which will make the value of the trigger non-negative.

The process which is waiting on a trigger can define a time-out value. If the time-out has elapsed it will be triggered (become non-negative) by the operating system (actually: The previous state of the trigger is restored). If the now ready process has a higher priority than the actual running process the operating system will pre-empt the running process and execute the triggered process.

The **sc_triggerValueSet** system calls allows to sets the value of a trigger. Only the owner of the trigger can set the value. Processes can also read the values of trigger by the **sc_triggerValueGet** call.

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

8.3 Trigger Example

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

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

extern sc_pid_t        trigproc_pid

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


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

SC_PROCESS (trigproc)
{

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

   for (;;)
   {

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

8.4 Trigger System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

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

9 Time Management

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 from another process or process can be suspended for a specific time or timer processes can be defined which are activated at specific time intervals.

9.2 System Tick

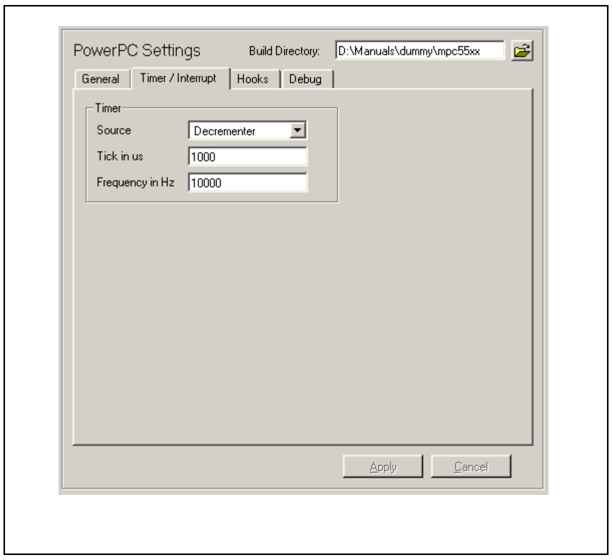
Time is managed by SCIOPTA by a tick timer which can be selected and configured by the user.

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

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

9.2.1 Configuring the System Tick

The system tick is configured by the sciopta configuration utility (see chapter 16.9.2 “Timer and Interrupt Configuration Tab” on page 16-10).


PowerPC Settings

Build Directory: D:\Manuals\dummy\mpc55xx

Timer / Interrupt

Timer

Source: Decrementer

Tick in us: 1000

Frequency in Hz: 10000

Figure 9-1: Tick Timer Configuration

You can use either an internal specific timer for tick or configure an external timer. If an external timer is used a tick interrupt process must be specified which will call **sc_tick** at regular intervals.

9.2.2 External Tick Interrupt Process

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

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

9.3 Timing System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_sleep	Suspends a process for a defined time.
sc_tick	Calls the kernel tick function. Advances the kernel tick counter by 1.
sc_tickActivationGet	Returns the tick time of last activation of the calling process.
sc_tickGet	Returns the actual kernel tick counter value.
sc_tickLength	Returns/sets the current system tick-length.
sc_tickMs2Tick	Converts a time from milliseconds into ticks.
sc_tickTick2Ms	Converts a time from ticks into milliseconds.

9.4 Timeout Server

9.4.1 Introduction

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

9.4.2 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.

9.5 Timeout Server System Calls

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_tmoAdd	Request a time-out message after a defined time.
sc_tmoRm	Remove a time-out job.

10 Error Handling

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 identities are verified in different kernel functions.
- Ownership of messages are checked.
- Parameters and sources of system calls are validated.
- The kernel will detect if messages and stacks have been over written beyond its length.

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

10.2 Error Sequence

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

For all other types of error 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.

10.3 Error Hook

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:

(Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA 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.

10.3.1 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 the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA error word. There are also four additional 32-bit extra words (parameters extra1 ... extra3) available to the user.

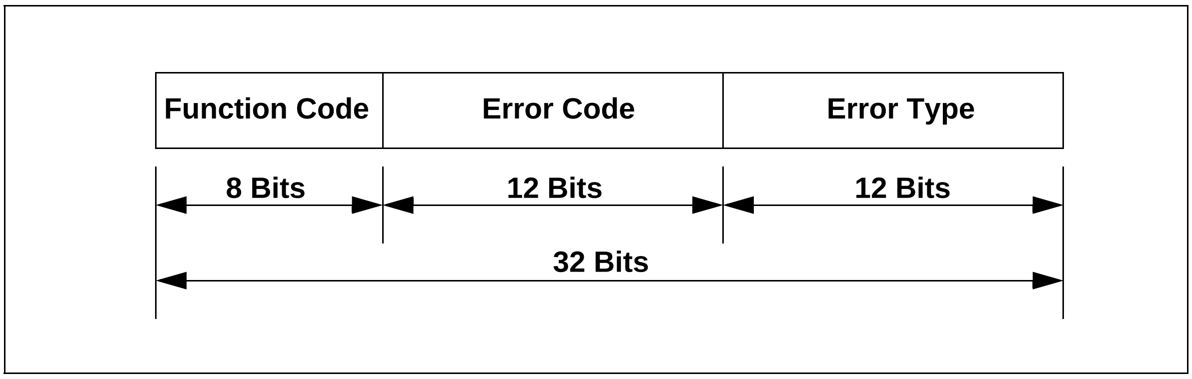


Figure 10-1: 32-bit Error Word (Parameter: errcode)

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.

10.3.2 Error Hook Registering

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

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

10.3.3 Error Hook Declaration Syntax

10.3.3.1 Description

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

10.3.3.2 Syntax

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

   ... error hook code

};
```

10.3.3.3 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.

10.3.4 Kernel Error Message Structure

```
typedef struct sc_errMsg_s {
  __u32 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;
```

10.3.4.1 Structure Members

user	User/system error flag.
!= 0	User error.
== 0	System error.
error	Error word.
	Error word containing the function code which defines from which SCIOPTA system call the error was initiated, the error code which contains the specific error information and the error type which informs about the source and type of error.
extra	System specific extra error words.
extra0 extra1 extra2 extra3	Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA extra error words.
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.

10.3.4.2 Header Files

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

10.3.5 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");
}
```

10.4 Error Process

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.

10.4.1 Error Process Registering

The error process will be registered by calling **sc_procAtExit** in the **init** process (see chapter 5.9 “Init Processes” on page 5-8) of the system module. It behaves like an interrupt process and should be as short as possible.

Actually it must delegate the error-recovery and error-handling to a error proxy.

10.4.2 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");
  }
}
```

10.5 The Error Proxy

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.

10.5.1 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);
    }
  }
}
```

10.6 The errno Variable

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

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

11 System Start and Setup

11.1 Start Sequence

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

1. The kernel calls the function **reset_hook**.
2. The kernel performs some internal initialization.
3. The kernel calls **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.

11.2 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**.

11.2.1 Syntax

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

11.2.2 Parameter

None.

11.2.3 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.

11.2.4 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.
	File location: <installation_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\src

11.3 C Startup

After a cold start the kernel will call the C startup function. The C startup function is written in assembler and has the name **cstartup**. It initializes the C system and replaces the library C startup function. C startup functions are compiler specific.

11.3.1 Location

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

cstartup.S	C System initialization. File location: <installation_folder>\sciopta\<version>\bsp\<arch>\src

11.4 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 10.3.2 “Error Hook Registering” on page 10-3) and other kernel hooks.

11.4.1 Syntax

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

11.4.2 Parameter

None.

11.4.3 Return Value

None.

11.4.4 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>\

11.5 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.

Please consult chapter 5.9 "Init Processes" on page 5-8.

11.6 Module Start Functions

Please consult chapter 4 "Modules" on page 4-1 for general information about SCIOPTA modules.

11.6.1 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>\

11.6.2 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.

12 Additional Functions

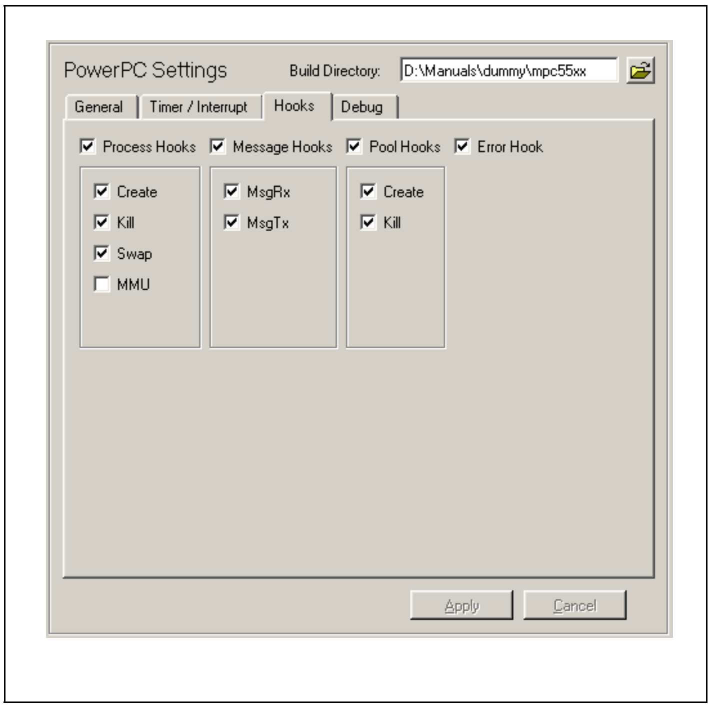
12.1 Introduction

In this chapter we are listing some additional functions of the kernel which are used for specific needs and projects.

12.2 Hooks

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).


PowerPC Settings
Build Directory: D:\Manuals\dummy\mpc55xx

☑ Process Hooks ☑ Message Hooks ☑ Pool Hooks ☑ Error Hook

☑ Create ☑ MsgRx ☑ Create
☑ Kill ☑ MsgTx ☑ Kill
☑ Swap
☐ MMU

Figure 12-1: Hook Configuration

Please consult chapter 16.9.3 “Hooks Configuration Tab” on page 16-12 for more information.

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

12.3 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 has been described in chapter 10.3 “Error Hook” on page 10-2.

12.4 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.

12.4.1 Registering Message Hooks

Message hooks must be registered by specific system calls. Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_msgHookRegister	Registers a message hook.

12.5 Process Hooks

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

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

12.5.1 Registering Process Hooks

Process hooks must be registered by specific system calls. Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_procHookRegister	Registers a process hook.

12.6 Pool Hooks

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

12.6.1 Registering Pool Hooks

Pool hooks must be registered by specific system calls. Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

sc_poolHookRegister	Registers a pool hook.

12.7 Exception Handling

12.7.1 Introduction

Exception handling for SCIOPTA is mainly done inside the kernel. Depending on the CPU family there might be some external functions needed. They are usually supplied by SCIOPTA and included in the board support package of the delivery.

12.7.2 SCIOPTA PowerPC Exception Handling

12.7.2.1 PowerPC CPU Family Exception Handling Files

exception.<ext>	PowerPC kernel - exception handling.
File extensions **<ext>**:	**S** GNU GCC and Windriver File location: <install_folder>\sciopta\<version>\bsp\ppc\<cpu>\src\

Usually there is no need to modify the exception handlers.

The vector table is included int the file **exception.<ext>**.

Interrupt handling is included int the file **exception.<ext>**.

12.7.2.2 PowerPC Interrupt Macros

The file **exception.<ext>** uses some specific interrupt macros which are defined in **irq.S**.

irq.S	Interrupt macros. File location: <installation_folder>\sciopta\<version>\include\machine\ppc\

12.8 Trap Interface

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

In more complex dynamic systems using load modules or MMU protected modular systems the kernel functions cannot be accessed any more by direct calls.

SCIOPTA offers a trap interface. In such systems you need to assemble the CPU dependent file syscall.S.

syscall.S	SCIOPTA kernel trap interface trampoline functions. File location: <installation_folder>\sciopta\<version>\include\machine\<arch>\

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

syscall.S	SCIOPTA kernel trap interface trampoline functions, not CPU dependent. File location: <installation_folder>\sciopta\<version>\include\machine\

12.9 Distributed Systems

12.9.1 Introduction

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

12.9.2 CONNECTORS

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

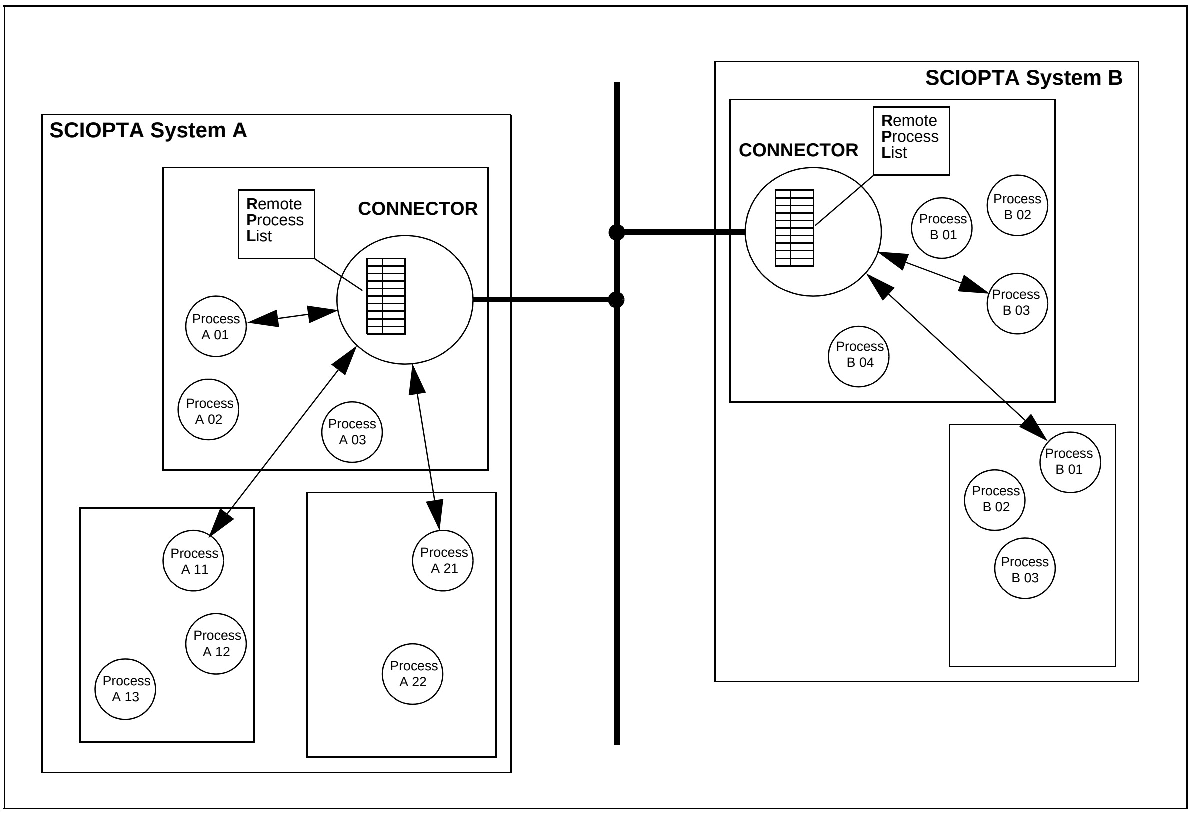


Figure 12-2: SCIOPTA Distributed System

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

12.9.3 Transparent Communication

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

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

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

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

13 SCIOPTA Design Hints and Tips

13.1 Introduction

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

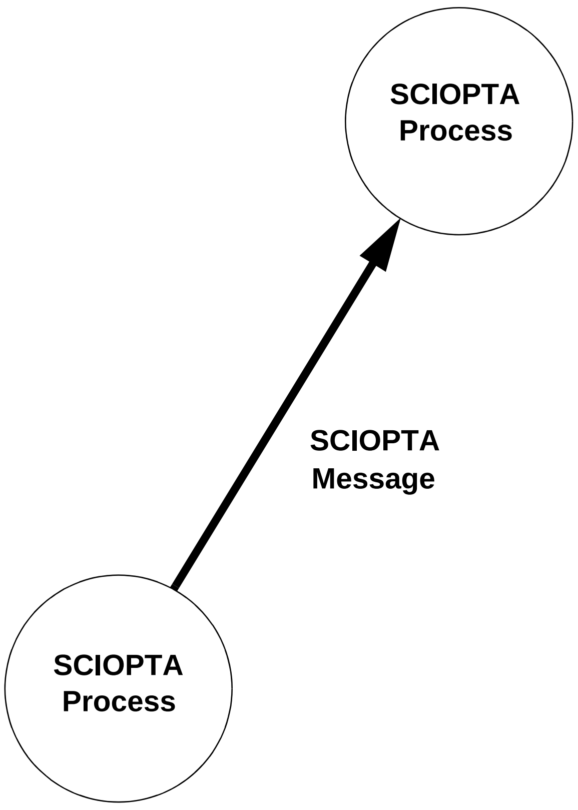
A typical system controlled by SCIOPTA consists of a number of more or less independent programs called processes. Each process can be seen as if it had the whole CPU for its own use. SCIOPTA controls the system by activating the correct processes according to their priority assigned by the user. Occurred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

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

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

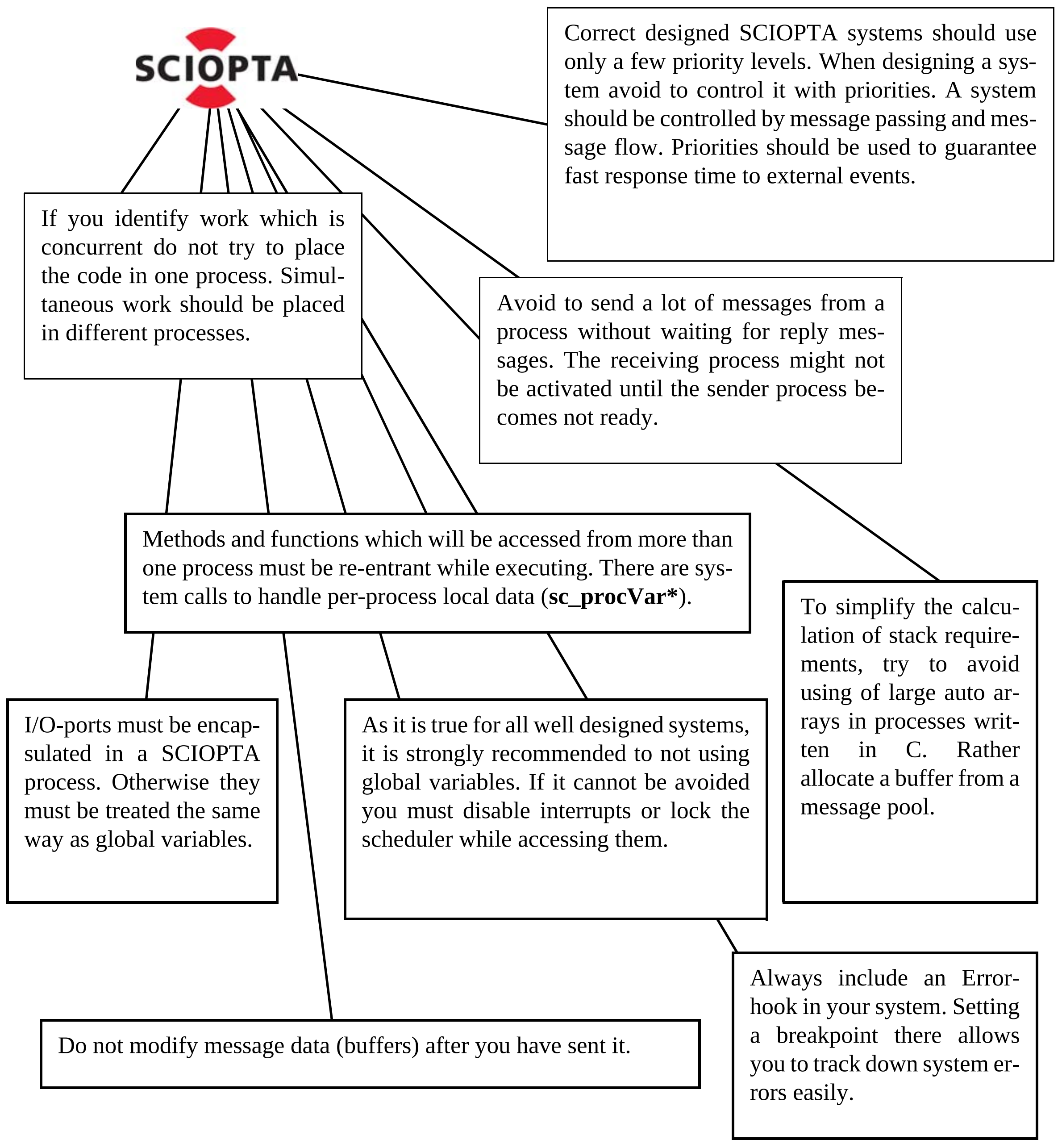
This can be used by a process for selecting specific messages to receive at a certain moment. All other messages are kept back in the message queue of the receiving process.

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



13.2 Some SCIOPTA Design Rules

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



Correct designed SCIOPTA systems should use only a few priority levels. When designing a system avoid to control it with priorities. A system should be controlled by message passing and message flow. Priorities should be used to guarantee fast response time to external events.

If you identify work which is concurrent do not try to place the code in one process. Simultaneous work should be placed in different processes.

Avoid to send a lot of messages from a process without waiting for reply messages. The receiving process might not be activated until the sender process becomes not ready.

Methods and functions which will be accessed from more than one process must be re-entrant while executing. There are system calls to handle per-process local data (**sc_procVar***).

To simplify the calculation of stack requirements, try to avoid using of large auto arrays in processes written in C. Rather allocate a buffer from a message pool.

I/O-ports must be encapsulated in a SCIOPTA process. Otherwise they must be treated the same way as global variables.

As it is true for all well designed systems, it is strongly recommended to not using global variables. If it cannot be avoided you must disable interrupts or lock the scheduler while accessing them.

Always include an Error-hook in your system. Setting a breakpoint there allows you to track down system errors easily.

Do not modify message data (buffers) after you have sent it.

14 Board Support Packages

14.1 Introduction

A SCIOPTA board support package (BSP) consists of number of files containing device drivers and project files such as makefiles and linker script for specific boards.

The BSPs are included in the delivery in a specific folder and organized in different directory levels depending on CPU dependency:

1. General System Functions
2. Architecture System Functions
3. CPU Family System Functions
4. Board System Functions

All BSP files can be found at the following top-level location after you have installed SCIOPTA:

File location: <install_folder>\sciopta\<version>\bsp\

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

14.2 General System Functions

General System Functions are functions which are common to all architectures, all CPUs and all boards.

It contains mainly include and source device drivers files for external (not on-chip) controllers.

Generic debugger files might also be placed here.

File location: <install_folder>\sciopta\<version>\bsp\common\include\
File location: <install_folder>\sciopta\<version>\bsp\common\src\

14.3 Architecture System Functions

Architecture System Functions are functions which are architecture (<arch>) specific (please consult chapter 1.3.1 "Architectures" on page 1-2 for the list of supported architectures) and are common to all CPUs and all boards.

It contains generic linker script include files (module.ld), C startup files (cstartup.S) and other architecture files.

File location: <install_folder>\sciopta\<version>\bsp\<arch>\include\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\src\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\src\<compiler>\

14.4 CPU Family System Functions

CPU Family System Functions are functions which are architecture (<arch>) specific and CPU family specific (please consult chapter 1.3.2 “CPU Families” on page 1-2 for the list of supported CPU families) and are common to all boards.

It contains mainly include and source device drivers files for on-chip controllers.

File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\include\
File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\src\

14.5 Board System Functions

Board System Functions are functions which are architecture (<arch>) specific, CPU family specific and board specific.

It contains mainly include, source and project files for board setup.

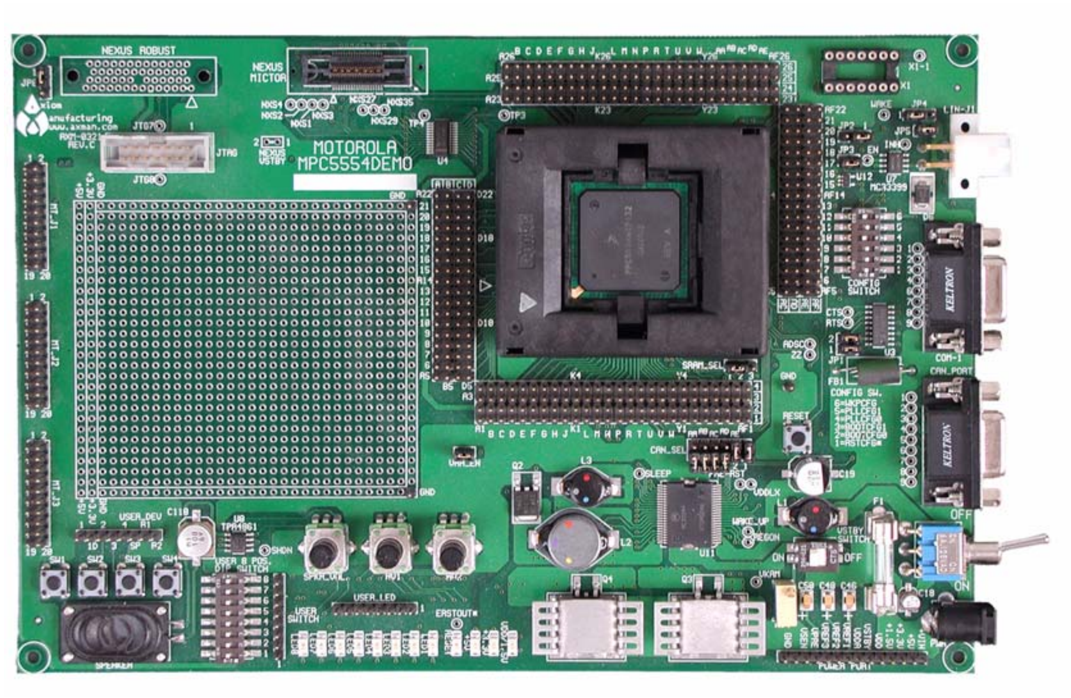
Debugger initialization files and linker scripts might also placed here.

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

14.6 Standard MPC55xx Boards

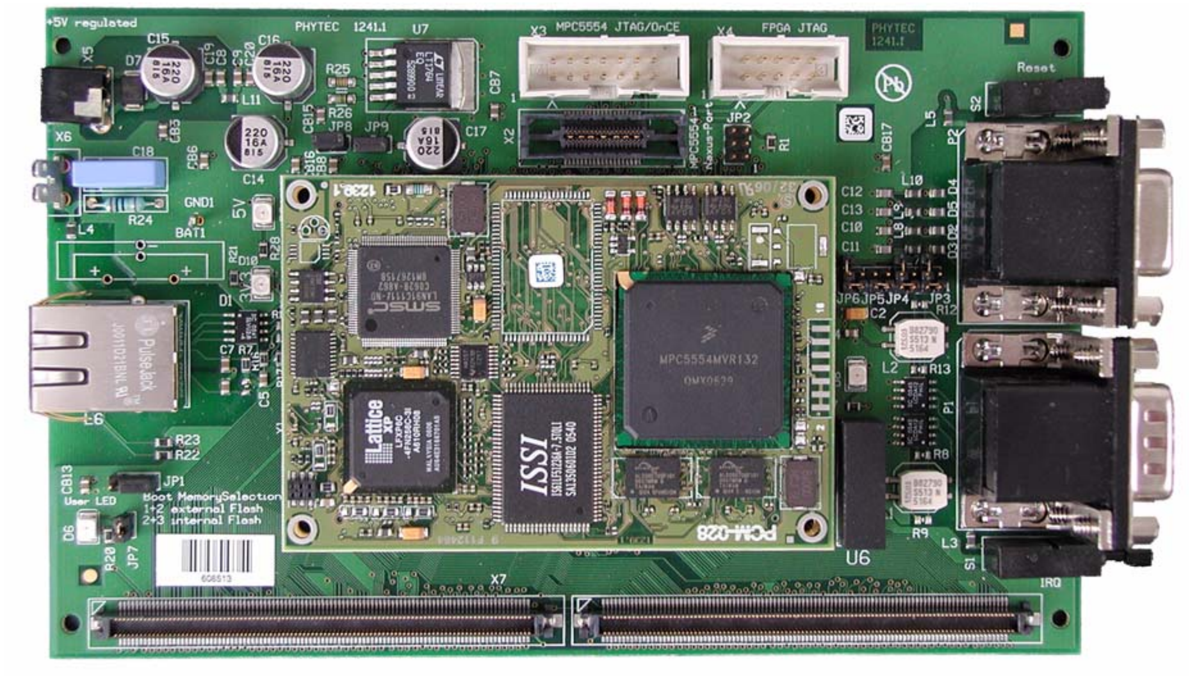
Please note that we are supporting many MPC55xx based boards. The boards listed here are included in the standard delivery and maintained within the shipped versions.

14.6.1 Motorola MPC5554DEMO Board



CPU	Freescale MPC5554
BOARD_SEL	Makefile board selection number.
9	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = 9) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\arm\hello\mpc5554demo\
Log Port	Log message port.
COM-1	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

14.6.2 Phytec phyCORE-MPC5554 Board



CPU	Freescale MPC5554
BOARD_SEL	Makefile board selection number.
9	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = 9) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\arm\hello\phyCoreMPC5554\
Log Port	Log message port.
P1 (Lower)	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

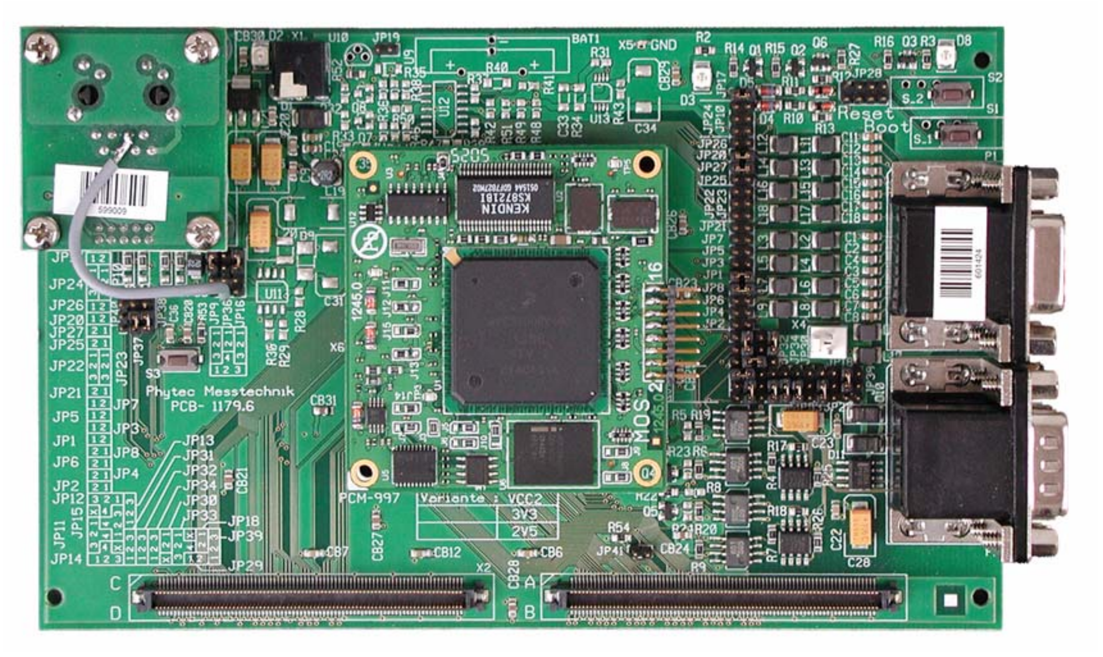
14.6.3 Freescale MPC5567EVB Board

CPU	Freescale MPC5567
BOARD_SEL	Makefile board selection number.
12	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = 12) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\arm\hello\mpc5567evb\
Log Port	Log message port.
xxx	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

14.7 Standard MPC5200 Boards

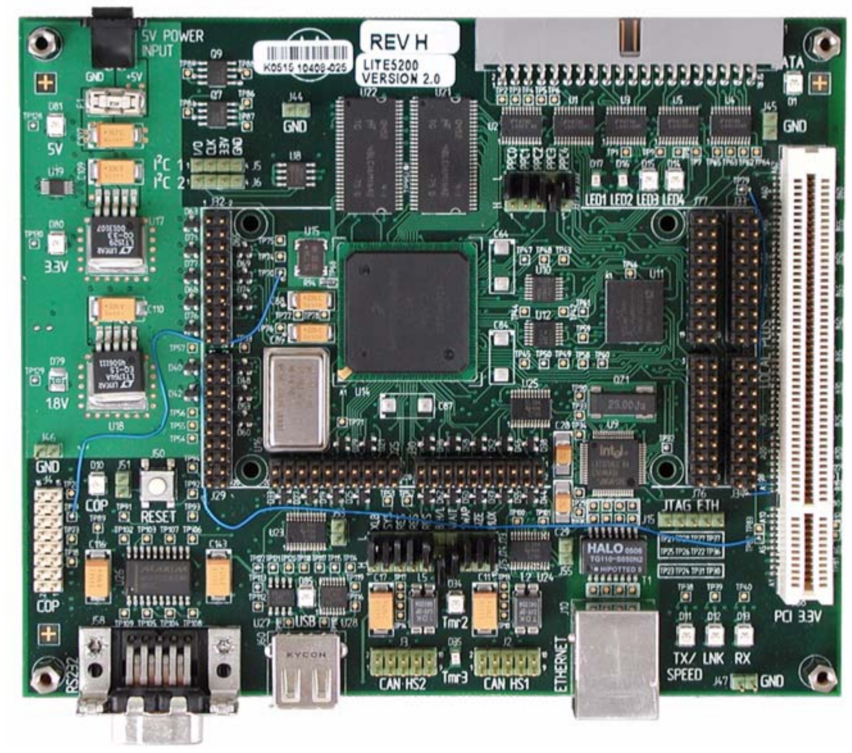
Please note that we are supporting many MPC5200 based boards. The boards listed here are included in the standard delivery and maintained within the shipped versions.

14.7.1 Phytec phyCORE-MPC5200B Tiny Board



CPU	Freescale MPC5200B
BOARD_SEL	Makefile board selection number.
5	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = 5) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\arm\hello\phyCoreMPC5200B-tiny\
Log Port	Log message port.
P1 (Lower)	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

14.7.2 Freescale Lite5200 Board

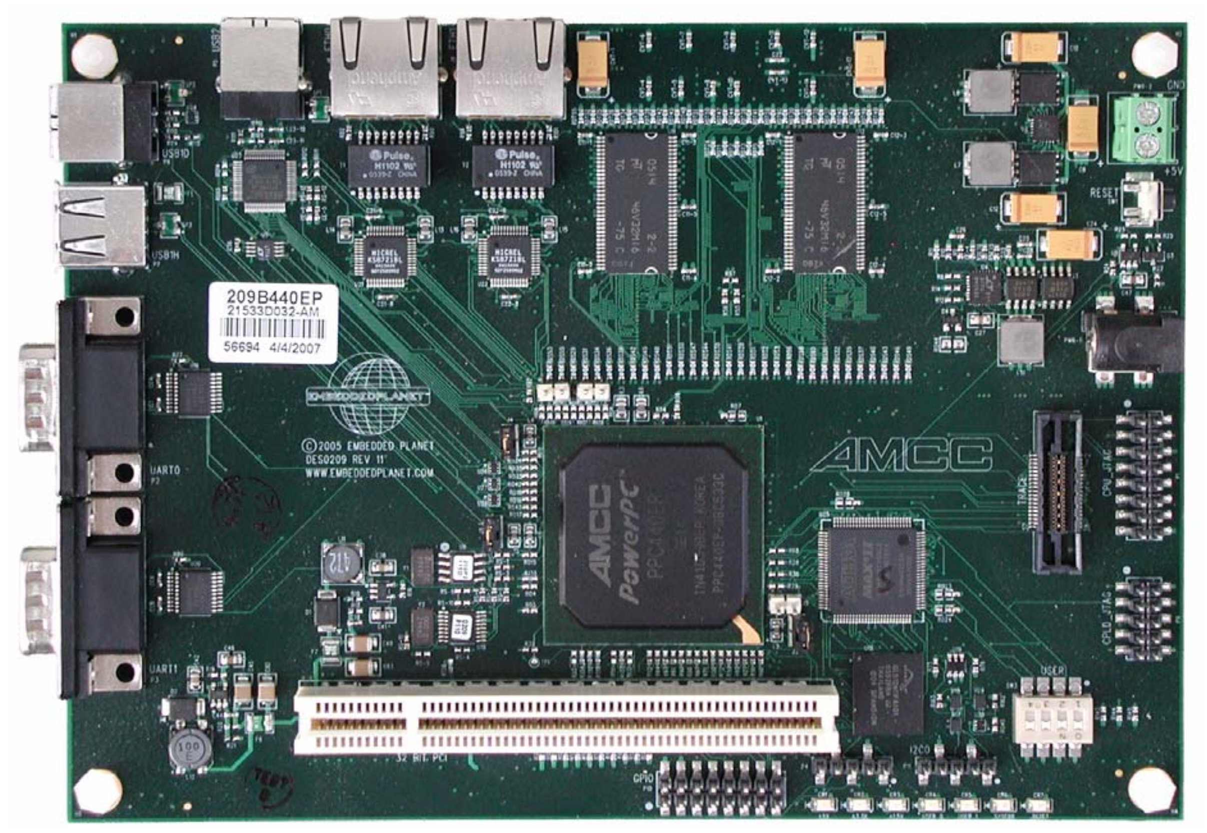


CPU	Freescale MPC5200B
BOARD_SEL	Makefile board selection number.
2	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = 2) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\arm\hello\lite5200\
Log Port	Log message port.
RS232	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

14.8 Standard PPC400 Boards

Please note that we are supporting many PPC400 based boards. The boards listed here are included in the standard delivery and maintained within the shipped versions.

14.8.1 AMCC Yosemite 440EP Evaluation Board

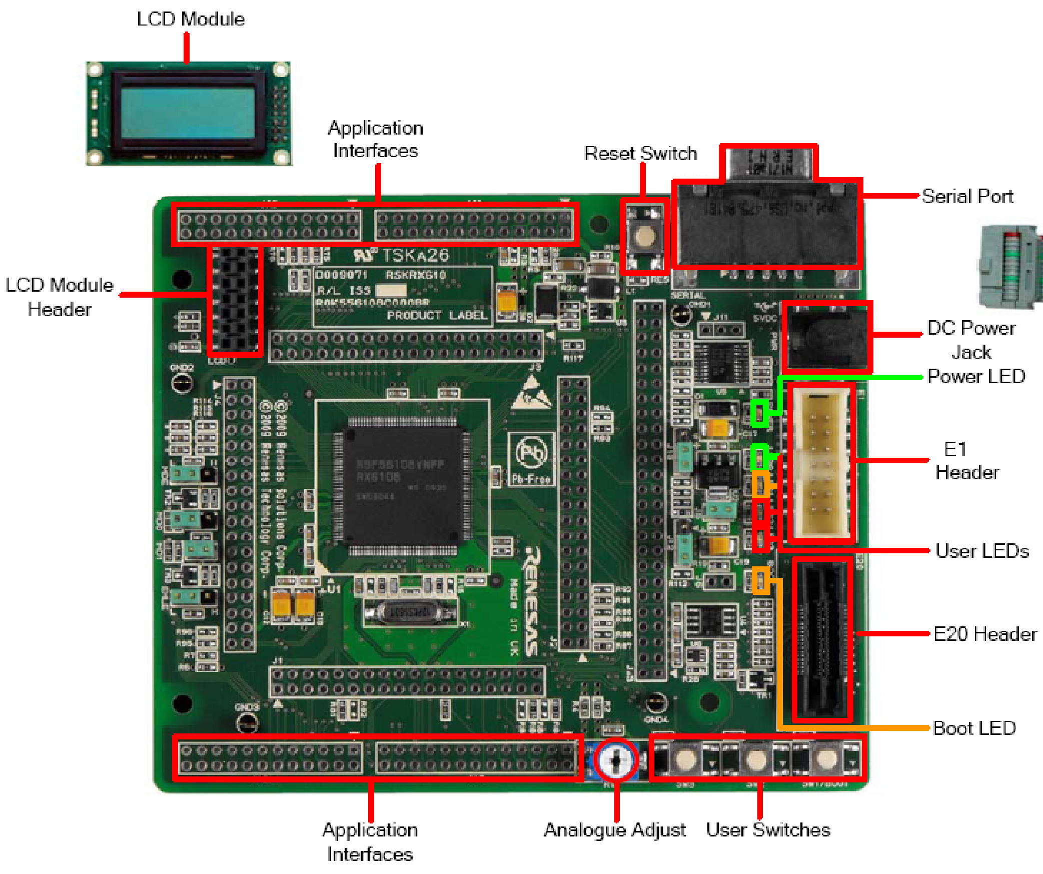


CPU	AMCC 440EP
BOARD_SEL	Makefile board selection number.
8	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = 8) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\arm\hello\yosemite440ep\
Log Port	Log message port.
UART0	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

14.9 Standard RX600 Boards

Please note that we are supporting many Renesas RX600 based boards. The boards listed here are included in the standard delivery and maintained within the shipped versions.

14.9.1 Renesas RSKRX610 Evaluation Board



CPU	RX610
BOARD_SEL	Makefile board selection number.
TBD	To be used as parameter when calling the makefile from Eclipse or from a shell. (Example: gnu_make BOARD_SEL = TBD) Makefile location for kernel example (hello): <install_folder>\sciopta\<version>\exp\krn\rx\hello\rx610sk\
Log Port	Log message port.
Serial Port	The getting started examples are sending some specific example log messages to a selected UART of the board. Also log daemon message are sent to this port.

15 Building SCIOPTA Systems

15.1 Introduction

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

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

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

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

One of the strength of SCIOPTA is that it is easy-to-use. A large part of a typical SCIOPTA application can be written by using the system calls which are handling the interprocess communication: **sc_msgAlloc**, **sc_msgTx**, **sc_msgRx** and **sc_msgFree**. These four system calls together with **sc_msgOwnerGet** which returns the owner of a message and **sc_sleep** which is used to suspend a process for a defined time, are often sufficient to write whole SCIOPTA applications.

Please consult the SCIOPTA - Kernel V2, Reference Manual for detailed description of the SCIOPTA system calls.

The SCIOPTA building procedure consists of the following steps:

- Configuring the system with the **SCONF** configuration tool (sconf.exe see chapter 16 “SCONF Kernel Configuration” on page 16-1). The SCONF tool generates the C file **sconf.c** (system defines and start) and the include file **sconf.h**.
- Locate the include files and define the include paths.
- Assemble the kernel.
- Locate and get all assembler source files and assemble it.
- Locate and get all C/C++ source files and compile them.
- Design the linker script to map your system into the target memory.
- Select and define the correct libraries for the SCIOPTA **Generic Device Driver** (gdd) and **Utilities** (util) functions.
- Link the system.

The Getting Started project (see chapter 3 “Getting Started” on page 3-1) is a good example for the needed files of a SCIOPTA application. This example project as a good starting point for your system design.

15.2 Configuration

The kernel of a SCIOPTA system needs to be configured before you can generated the whole system.

In the SCIOPTA configuration utility **SCONF** (sconf.exe) you will define the parameters for SCIOPTA systems such as name of systems, static modules, processes and pools etc. Configure the system with the **SCONF** configuration tool (see chapter 16 “SCONF Kernel Configuration” on page 16-1).

The SCONF tool generates the C file **sconf.c** (system defines and startup) and the include file **sconf.h**.

15.3 Include Files

15.3.1 Include Files Search Directories

Please make sure that the environment variable **SCIOPTA_HOME** is defined as explained in chapter 2.4.6 “SCIOPTA_HOME Environment Variable” on page 2-4.

Define the following entries the include files search directories field of your IDE:

```
.
%(SCIOPTA_HOME)\include
%(SCIOPTA_HOME)\include\sciopta\<arch>
```

Depending on the CPU and board you are using:

```
%(SCIOPTA_HOME)\bsp\<arch>\include
%(SCIOPTA_HOME)\bsp\<arch>\<cpu>\include
%(SCIOPTA_HOME)\bsp\<arch>\<cpu>\<board>\include
```

15.3.2 Main Include File sciopta.h

The file **sciopta.h** contains some main definitions and the SCIOPTA Application Programming Interface. Each module or file which is using SCIOPTA system calls and definitions must include this file. The file **sciopta.h** includes all specific API header files.

sciopta.h	Main SCIOPTA include file.
	File location: <installation_folder>\sciopta\<version>\include\

15.3.3 Configuration Definitions sconf.h

Files or modules which are SCIOPTA configuration dependent need to include first the file **sconf.h** and then the file **sciopta.h**.

The file **sconf.h** needs to be included if for instance you want to know the maximum number of modules allowed in a system. This information is stored in **SC_MAX_MODULES** in the file **sconf.h**.

Please remember that **sconf.h** is automatically generated by the **SCONF** configuration tool and placed at a convenient location in your project folder.

15.3.4 Main Data Types types.h

These types are introduced to allow portability between various SCIOPTA implementations.

The main data types are defined in the file types.h. These types are not target processor dependent.

types.h	Processor independent data types. File location: <installation_folder>\sciopta\<version>\include\ossys\

15.3.5 Architecture Dependent Data Types types.h

The architecture specific data types are defined in the file types.h.

types.h	Architecture dependent data types. File location: <installation_folder>\sciopta\<version>\include\sciopta\<arch>\arch\

15.3.6 Global System Definitions defines.h

System wide definitions are defined in the file defines.h. Among other global definitions, the base addresses of the IDs of the SCIOPTA system messages are defined in this file. Please consult this file for managing and organizing the message IDs of your application.

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

15.3.7 Board Configuration

It is good design practice to include specific board configurations, defines and settings in a file. In the SCIOPTA board support package deliveries such example files (**config.h**) are available.

config.h	Board configuration defines. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\include\

15.4 Assembling the PowerPC Kernel

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

15.4.1 Kernels for PowerPC Architectures

Architecture (<arch>): **ppc** (see chapter 1.3.1 “Architectures” on page 1-3).

sciopta.S	Kernel source file for GNU GCC and Windriver
sciopta_mmu.S	Kernel source file for GNU GCC and Windriver including MMU support
sciopta_vle.S	Kernel source file for GNU GCC and Windriver including PowerPC VLE code support

File location: <install_folder>\sciopta\<version>\lib\ppc\krn\

15.5 Renesas RX600 Kernel

The SCIOPTA kernel for Renesas RX600 is provided as object libraries. It is compiler manufacturer and device specific. For different configuration settings (see chapter) there are specific libraries included.

Architecture (<arch>): **rx** (see chapter 1.3.1 “Architectures” on page 1-3).

File locations for **Renesas HEW** and RX600 and RX620:
<installation_folder>\sciopta\<version>\lib\rx\hew_rx600\

File locations for **Renesas HEW** and RX610:
<installation_folder>\sciopta\<version>\lib\rx\hew_rx610\

File locations for **IAR Embedded Workbench** and RX600 and RX620:
<installation_folder>\sciopta\<version>\lib\rx\iar_rx600\

File locations for **IAR Embedded Workbench** and RX610:
<installation_folder>\sciopta\<version>\lib\rx\hew_rx610\

The file name of the libraries have the following format:

krnrx_**dh**.a

15.5.1 Kernel Library Versions

d	SCIOPTA debug configuration level (see chapter 16.9.4 “Debug Configuration Tab” on page 16-13)
0	No debug
1	C debug only
2	All except message checks
3	All

h	SCIOPTA hooks configuration (see chapter 16.9.3 “Hooks Configuration Tab” on page 16-12)
0	No hooks
1	Error hook only
2	All hooks

15.6 Assembling the Assembler Source Files

Usually there are not many assembler source file in a project. Sometimes system files from the board support packages might be needed.

15.6.1 PowerPC Architecture Assembler Source Files

atomic.<ext>	Routine to atomically increment a variable.
cstartup.<ext>	C startup code for PowerPC.
cstartup_vle.<ext>	C startup code for PowerPC with VLE code support.
md5.<ext>	MD5 Sum file for PowerPC.

File extensions **<ext>**: **S** GNU GCC and Windriver
File location: <install_folder>\sciopta\<version>\bsp\ppc\src\

15.6.2 RX600 Architecture Assembler Source Files

cstartup.<ext>	C startup code for RX600.
vectors.<ext>	Vector table for RX600.

File extensions **<ext>**: **s** IAR
File location: <install_folder>\sciopta\<version>\bsp\rx\src\iar\

asm Renesas HEW
File location: <install_folder>\sciopta\<version>\bsp\rx\src\hew\

15.6.3 PowerPC CPU Family Assembler Source Files

Typical files for PowerPC architecture and specific CPU families.

exception.<ext>	PowerPC kernel - exception handling.
core.<ext>	Some helper function to access SPFRs.
systick.<ext>	System tick interrupt handler.
systick_vle.<ext>	System tick interrupt handler including PowerPC VLE code support.

File extensions **<ext>**: **S** GNU GCC and Windriver
File location: <install_folder>\sciopta\<version>\bsp\ppc\<cpu>\src\

Please Note: Not all files are needed for all CPU families. Please consult the examples in the SCIOPTA deliveries, the sources and the project requirements to include the needed files.

Please Note: Not all files are needed for all CPU families. Please consult the examples in the SCIOPTA deliveries, the sources and the project requirements to include the needed files.

15.6.4 PowerPC Boards Assembler Source Files

Typical files for PowerPC architecture and specific boards.

resethook.<ext>	Resethook early startup functions.
led.<ext>	LED access functions.

File extensions **<ext>**: **S** GNU GCC and Windriver
File location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\src\

Please Note: Not all files are needed for all boards. Please consult the examples in the SCIOPTA deliveries, the sources and the project requirements to include the needed files.

15.7 Compiling the C/C++ Source Files

15.7.1 CPU Families C/C++ Source Files

Typical CPU family files:

druid_uart.c	Druid UART driver.
simple_uart.c	Simple polling UART function for printf debugging or logging.
serial.c	Serial driver.
systick.c	System tick (RX600).
<driver_name>.c	CPU family specific device drivers. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\src\

15.7.2 Chip Driver C/C++ Source Files

Typical chip and device files:

<driver_name>.c	Specific device drivers for controllers and devices which are not board specific. File location: <install_folder>\sciopta\<version>\bsp\common\src\

15.7.3 Boards C/C++ Source Files

Typical board files:

<driver_name>.c	Board specific device drivers. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\src\

15.7.4 Configuration C/C++ Files

Typical system configuration and initialization file:

system.c	System configuration file including the start_hook the system module start function and other setup code. File example location: <install_folder>\sciopta\<version>\exp\<arch>\<example>\<board\

Typical user modules system configuration and initialization files:

<module_name>.c	User module start function and other user module setup code. File example location: <install_folder>\sciopta\<version>\exp\<arch>\<example>\<board\

15.7.5 User Application C/C++ Files

User application files:

<application>.c	User application files containing processes and other application functions.
<application>.msg	User application files containing message declarations and definitions. File example location: <project_folder>

15.8 Linker Scripts

15.8.1 Introduction

A linker script is controlling the link in the build process. The linker script is written in a specific linker command language. The linker script and linker command language are compiler specific.

The linker script describes how the defined memory sections in the link input files are mapped into the output file which will be loaded in the target system. Therefore the linker script controls the memory layout in the output file.

SCIOPTA uses the linker scripts to define and map SCIOPTA modules into the global memory map.

15.8.2 GCC Linker Scripts

You can find examples of linker scripts in the SCIOPTA examples and getting started projects. In these examples there is usually a main linker script which includes a second linker script. The main linker scripts are board dependent.

<board_name>.ld	Board specific linker script for GNU GCC. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\include\

Usually the main linker scripts includes another standard linker script:

module.ld	Module linker script for GNU GCC. File location: <install_folder>\sciopta\<version>\bsp\<arch>\include\

Study these linker script files to get full information how to locate a SCIOPTA system in the embedded memory space.

15.8.2.1 Memory Regions

The main linker script contains the allocation of all available memory and definition of the memory regions including the regions for all modules. Sections are assigned to SCIOPTA specific memory regions. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name.

The following regions are typically defined by the user:

rom	destination for read-only data
sram	internal SRAM
system_mod	Memory region for the system module
<module_name>_mod	Memory region for other modules (with name: <module_name>).

15.8.2.2 Module Sizes

The sizes used by SCIOPTA of each module must be defined by the user in the linker script. This size determines the memory which will be used by SCIOPTA for message pools, PCBs and other system data structures.

The name of the size is usually defined as follows: **<module_name>_size**

The module name for the system module is **system**.

Typical definitions (for modules **system**, **dev**, **ips** and **user**) might look as follows:

```
system_size = 0x4000;
dev_size    = 0x4000;
ips_size    = 0x4000;
user_size   = 0x4000;
```

size calculation:

size_mod = p * 256 + stack + pools + mcb + textsize
where:

p	Number of static processes
stack	Sum of stack sizes of all static processes
pools	Sum of sizes of all message pools
mcb	module control block = 200 bytes
textsize	Size of the memory which is initialized by the C-Startup function (cstartup.S)

Please consult the configuration chapter (**SCONF** utility) of the SCIOPTA Kernel, User's Guide for information about friend and hook settings.

15.8.2.3 Specific Module Values

For each module four values are calculated in the linker script:

<module_name>_start	Start address of module RAM
<module_name>_initsize	Size of initialized RAM
<module_name>_size	Complete size of the module
<module_name>_mod	A structure which contains the above three addresses.

The SCIOPTA configuration utility **SCONF** is using these definitions to pass the module addresses to the kernel.

Example in the linker script:

```
.module_init :
  {
        system_mod = .;
        LONG(system_start);
        LONG(system_size);
        LONG(system_initsize);
        dev_mod = .;
        LONG(dev_start);
        LONG(dev_size);
        LONG(dev_initsize);
        ips_mod = .;
        LONG(ips_start);
        LONG(ips_size);
        LONG(ips_initsize);
        user_mod = .;
        LONG(user_start);
        LONG(user_size);
        LONG(user_initsize);
   } > rom
```

15.8.2.4 GCC Data Memory Map

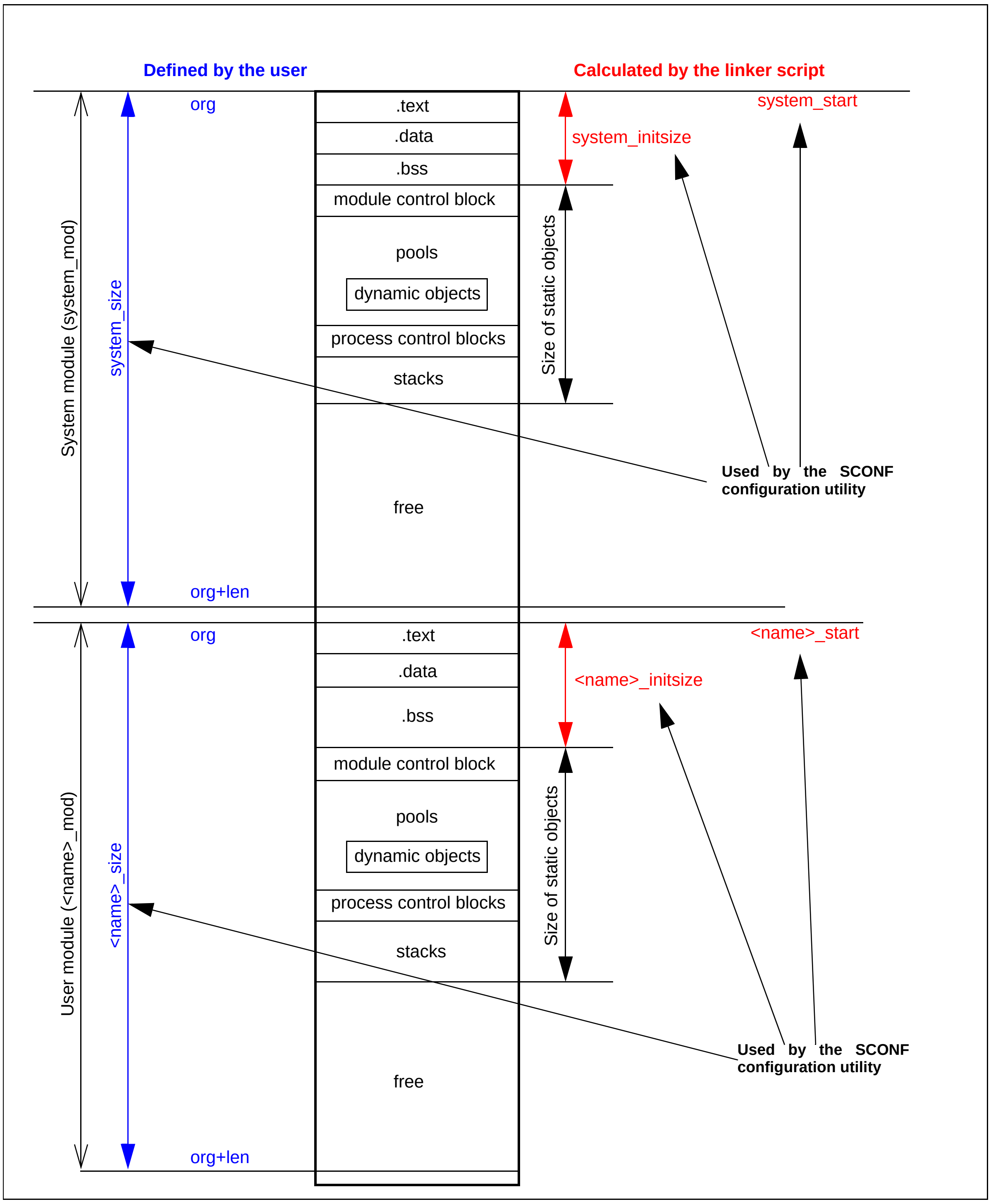


Figure 15-1: SCIOPTA Memory Map

15.8.3 Windriver Linker Scripts

You can find examples of Windriver C/C++ Compiler Package linker scripts in the SCIOPTA examples and getting started projects. In these examples there is usually a main linker script which includes a second linker script. The main linker scripts are board dependent.

<board_name>.dld	Board specific linker script for Windriver. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\include\

Usually the main linker scripts includes another standard linker script:

module.dld	Module linker script for Windriver. File location: <install_folder>\sciopta\<version>\bsp\<arch>\include\

Study these linker script files to get full information how to locate a SCIOPTA system in the embedded memory space.

15.8.4 IAR Embedded Workbench Linker Scripts

You can find examples of IAR Embedded Workbench linker scripts in the SCIOPTA examples and getting started projects.

<board_name>.icf	Board specific Linker script for IAR. File location: <install_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\include\

For IAR you need to define the free RAM of the modules in a separate file. In this area there are no initialized data. Module Control Block (ModuleCB), Process Control Blocks (PCBs), Stacks and Message Pools are placed in this free RAM:

map.c	Module mapping definitions for IAR. File location: <installation_folder>\sciopta\<version>\exp\krn\arm\<example>\<board>\

Study these linker script files to get full information how to locate a SCIOPTA system in the embedded memory space.

15.9 GCC Kernel Libraries

For the SCIOPTA generic device driver (gdd) functions, the shell functions (sh) and the SCIOPTA utilities (util) some prebuilt libraries are included in the delivery

Please consult the SCIOPTA - Device Driver, User’s and Reference Manual for more information about generic device driver (gdd) functions.

The file name of the libraries have the following format:

libgdd_**xt**.a
libsh_**xt**.a
libutil_**xt**.a

File location: <installation_folder>\sciopta\<version>\lib\<arch>\gnu\

15.9.1 Library Versions

x	Compiler optimization level
0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

t	Trap interface.
	If the SCIOPTA Trap Interface is used, the libraries containing the letter “t” after the Optimization letter x must be included.

15.9.2 Building Kernel Libraries for GCC

The example makefiles and project files are supposing to use libraries for the generic device driver (gdd) and utility (util) modules. As described above, there are some libraries delivered for specific compiler settings. We have included source files and makefiles which allows you to build the libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

Procedures to generate the libraries

- Open a Command Prompt window.
- For Generic Device Driver libraries (gdd):
  - Navigate to the route source directory: <installation_folder>\sciopta\<version>\gdd\
  - Execute the makefile for each architecture:
    ppc: gnu-make -f Makefile.ppc gdd
- For Utilities and Shell libraries (util):
  - Navigate to the route source directory: <installation_folder>\sciopta\<version>\util\
  - Execute the makefile for each architecture:
    ppc: gnu-make -f Makefile.ppc util
- The libraries will be installed in the directory:
  <installation_folder>\sciopta\<version>\lib\<arch>\gnu\

15.10 Windriver Kernel Libraries

For the SCIOPTA generic device driver (gdd) functions, the shell functions (sh) and the SCIOPTA utilities (util) some prebuilt libraries are included in the delivery.

Please consult the SCIOPTA - Device Driver, User’s and Reference Manual for more information about generic device driver (gdd) functions.

The file name of the libraries have the following format:

libgdd_**xt**.a
libsh_**xt**.a
libutil_**xt**.a

File location: <installation_folder>\sciopta\<version>\lib\<arch>\diab\

File location for Windriver PowerPC VLE code:
<installation_folder>\sciopta\<version>\lib\ppc\diab_vle\

15.10.1 Library Versions

x	Compiler optimization level
0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

t	Trap interface.
	If the SCIOPTA Trap Interface is used, the libraries containing the letter “t” after the Optimization letter x must be included.

15.10.2 Building Kernel Libraries for Windriver

The example makefiles and project files are supposing to use libraries for the generic device driver (gdd) and utility (util) modules. As described above, there are some libraries delivered for specific compiler settings. We have included source files and makefiles which allows you to build the libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

Procedures to generate the libraries

- Open a Command Prompt window.
- For Generic Device Driver libraries (gdd):
  - Navigate to the route source directory: <installation_folder>\sciopta\<version>\gdd\
  - Execute the makefile for each architecture:

    ppc: gnu-make -f Makefile.ppcdiab gdd
    gnu-make -f Makefile.ppcdiabv gdd (PowerPC VLE Code)

- For Utilities and Shell libraries (util):
  - Navigate to the route source directory: <installation_folder>\sciopta\<version>\util\
  - Execute the makefile for each architecture:

    ppc: gnu-make -f Makefile.ppcdiab util
    gnu-make -f Makefile.ppcdiabv util (PowerPC VLE Code)

- The libraries will be installed in the directory:
  <installation_folder>\sciopta\<version>\lib\<arch>\diab\
- The libraries for PowerPC VLE Code will be installed in the directory:
  <installation_folder>\sciopta\<version>\lib\ppc\diab_vle\

15.11 IAR Kernel Libraries

For the SCIOPTA generic device driver (gdd) functions, the shell functions (sh) and the SCIOPTA utilities (util) some prebuilt libraries are included in the delivery

Please consult the SCIOPTA - Device Driver, User's and Reference Manual for more information about generic device driver (gdd) functions.

The file name of the RX libraries have the following format:

gdd_**x**.a
sh_**x**.a
util_**x**.a

File location RX600 and RX620:
<installation_folder>\sciopta\<version>\lib\rx\iar_rx600\
File location for RX610:
<installation_folder>\sciopta\<version>\lib\rx\iar_rx610\

15.11.1 Library Versions

x	Compiler optimization level
0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

15.11.2 Building Kernel Libraries for IAR

The example makefiles and project files are supposing to use libraries for the generic device driver (gdd) and utility (util) modules. As described above, there are some libraries delivered for specific compiler settings. We have included source files and IAR EW project files which allows you to build the libraries yourself. If you want to change compiler switches or other system settings you need to modify the IAR EW project file.

Procedures to generate the libraries

- For Generic Device Driver libraries (gdd):
    - Navigate to the route source directory: <installation_folder>\sciopta\<version>\gdd\
    - Load the gdd_rx.eww into the IAR Embedded Workbench. Select the appropiate target and build the libraries.

- For Utilities and Shell libraries (util):
    - Navigate to the route source directory: <installation_folder>\sciopta\<version>\util\
    - Load the util_rx.eww into the IAR Embedded Workbench. Select the appropiate target and build the libraries.

15.12 Linking the System

Now you are ready to link the generated object files from the assembler and C/C++ source files.

Make sure that you include the correct gdd and util libraries for your project and the correct linker script for you target environment.

15.13 Integrated Development Environments

15.13.1 Eclipse and GNU GCC

The **Eclipse IDE for C/C++ Developers** project provides a fully functional C and C++ Integrated Development Environment (IDE).

Please consult http://www.eclipse.org/ for more information about Eclipse. You can download **Eclipse IDE for C/C++ Developers** from the download page of this site.

Please consult http://www.eclipse.org/cdt for more information about Eclipse CDT (C/C++ Development Tools) project.

For all delivered SCIOPTA examples for the ARM, PowerPC and ColdFire architectures there are Makefiles included. Eclipse is easy to configure for working with external makefiles. Please consult chapter 3 "Getting Started" on page 3-1 for a detailed description how to setup Eclipse to build SCIOPTA systems.

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

15.13.1.1 Tools

The following tools re needed to build a SCIOPT project with Eclipse and GNU GCC.

- Compiler package:

  For **PowerPC** CodeSourcery GNU C & C++ Sourcery G++ Lite Edition for Power PC Version 4.2. Architecture (arch): ppc

  These packages can be found on the SCIOPTA CD.

- Eclipse IDE for C/C++ Developers. You can download Eclipse from here http://www.eclipse.org/.
- SCIOPTA - Kernel for your selected architecture.

15.13.1.2 Environment Variables

The following environment variables need to be defined:

- Check that the environment variable SCIOPTA_HOME is defined as described in chapter 2.4.6 "SCIOPTA_HOME Environment Variable" on page 2-4.
- Be sure that the GNU GCC compiler bin directory is included in the PATH environment variable as described in chapter 2.4.9 "GNU Tool Chain Installation" on page 2-5.
- Be sure that the SCIOPTA \win32\bin directory is included in the PATH environment variable as described in chapter 2.4.7 "Setting SCIOPTA Path Environment Variable" on page 2-4.

15.13.1.3 Eclipse Project Files

We are using **"makefile projects"** (contrary to "managed make projects") in Eclipse. For all delivered SCIOPTA examples for the PowerPC architecture there are makefiles included. Eclipse is easy to configure for working with external makefiles.

You will find typical makefiles for SCIOPTA in the example deliveries.

Makefile	Makefile for GNU GCC

File location: <installation_folder>\sciopta\<version>\exp\krn\<arch>\<example>\

Usually these example makefiles include a board specific makefile called board.mk located here:

board.mk	Board dependent makefiles.

File location: <installation_folder>\sciopta\<version>\exp\krn\<arch>\<example>\<board>

15.13.1.4 Project Settings in Eclipse

You just need to define the make call in the “Build command”.

- Click on the project in the Project Explorer window to make sure that the project is highlighted.
- Open the **Properties** window (menu: **File -> Properties** or **Alt+Enter** button).
- Click on “C/C++ Build.
- Deselect “Use default build command” in the Builder Settings Tab. Now you can enter a customized Build command.
- Enter the following Build command: **gnu-make**

  Enter make options according to you project need
- Click the **OK** button.

Now you can build the project from the menu (**Project > Build Project)** or by clicking on the **Build** button.

15.13.1.5 Debugger Board Setup Files

Board setup files for Lauterbach Trace32 and iSYSTEM winIDEA can be found in the SCIOPTA delivery.

<file_name>.cmm	Lauterbach Trace32 board setup file.
<file_name>.ini	iSYSTEM winIDEA board setup file.

The **file_name** is often the board name <board>.

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

15.13.2 iSYSTEM© winIDEA

The program winIDEA is the IDE for all iSYSTEMS emulators. It is the a Integrated Development Environment, which contains all the necessary tools in one shell. winIDEA consists of a project manager, a 3rd party tools integrator, a multi-file C source editor and a high-level source debugger.

Please consult http://www.isystem.com/ for more information about the iSYSTEM emulator/debugger.

15.13.2.1 Tools

The following tools are needed to build a SCIOPT project with iSYSTEM and GNU GCC.

- GNU GCC Compiler package:

For **PowerPC**	CodeSourcery GNU C & C++ Sourcery G++ Lite Edition for Power PC Version 4.2. Architecture (arch): ppc

  These packages can be found on the SCIOPTA CD.
- iSYSTEM debugger including winIDEA software.
- SCIOPTA - Kernel for your selected architecture.

15.13.2.2 Environment Variables

The following environment variables need to be defined:

- **SCIOPTA_HOME** needs to point to the SCIOPTA delivery. Please consult chapter 2.4.6 “SCIOPTA_HOME Environment Variable” on page 2-4 for more information.
- Include the GNU GCC compiler bin directory in the PATH environment variable as described in chapter 2.4.9 “GNU Tool Chain Installation” on page 2-5.

15.13.2.3 winIDEA Project Files

You will find typical winIDEA project files for SCIOPTA in the example deliveries.

<file_name>.xjrf	iSYSTEM winIDEA project file
<file_name>.xqrf	iSYSTEM winIDEA project file

File location: <installation_folder>\sciopta\<version>\exp\krn\<arch>\<example>\<board>

15.13.2.4 winIDEA Project Settings

Selecting **Projects > Settings...** from the menu (or press **Alt+F7**) opens the **Project Settings** window.

After expanding the **C/C++ Build** entry you can select **Settings** to open the project specific settings window.

Please consult the delivered example winIDEA project for detailed information about compiler, assembler and linker calls, options and settings.

15.13.2.5 winIDEA Board Setup Files

Board setup files for winIDEA can be found in the SCIOPTA delivery.

<file_name>.ini	iSYSTEM winIDEA board setup file.

The **file_name** is often the board name <board>.

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

15.13.3 IAR Embedded Workbench

IAR Embedded Workbench is a set of development tools for building and debugging embedded system applications using assembler, C and C++. It provides a completely integrated development environment that includes a project manager, editor, build tools and the C-SPY debugger.

Please consult http://www.iar.com/ for more information about the IAR Embedded Workbench.

15.13.3.1 Tools

The following tools re needed to build a SCIOPT project with IAR Embedded Workbench.

- **For RX:** IAR Embedded Workbench for **Renesas RX600** including the following main components:
    - IAR Assembler for ARM
    - IAR C/C++ Compiler for ARM
    - IAR Embedded Workbench IDE
    - IAR XLINK
- SCIOPTA - Real-Time Kernel for your selected architecture.

15.13.3.2 Environment Variables

The following environment variables need to be defined:

- **SCIOPTA_HOME** needs to point to the SCIOPTA delivery. Please consult chapter 2.4.6 “SCIOPTA_HOME Environment Variable” on page 2-4 for more information.

15.13.3.3 IAR EW Project Files

You will find typical IAR EW project files for SCIOPTA in the example deliveries.

<file_name>.ewd	IAR EW project file
<file_name>.ewp	IAR EW project file
<file_name>.eww	IAR EW project file

The **file_name** is often the board name <board>.

File location: <installation_folder>\sciopta\<version>\exp\krn\<arch>\<example>\<board>

15.13.3.4 IAR EW Project Settings

Selecting **Projects > Options...** from the menu (or press **Alt+F7**) opens the **Options** window.

Please consult the delivered example IAR EW project for detailed information about compiler, assembler and linker calls, options and settings.

15.13.3.5 IAR C-SPY Board Setup File

C-SPY is the name of the IAR Embedded Workbench debugger. Board setup file for C-SPY can be found in the SCIOPTA delivery.

<file_name>.mac	IAR EW C-SPY board setup file.

The **file_name** is often the board name <board>.

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

16 SCONF Kernel Configuration

16.1 Introduction

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

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

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

16.2 Starting SCONF

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

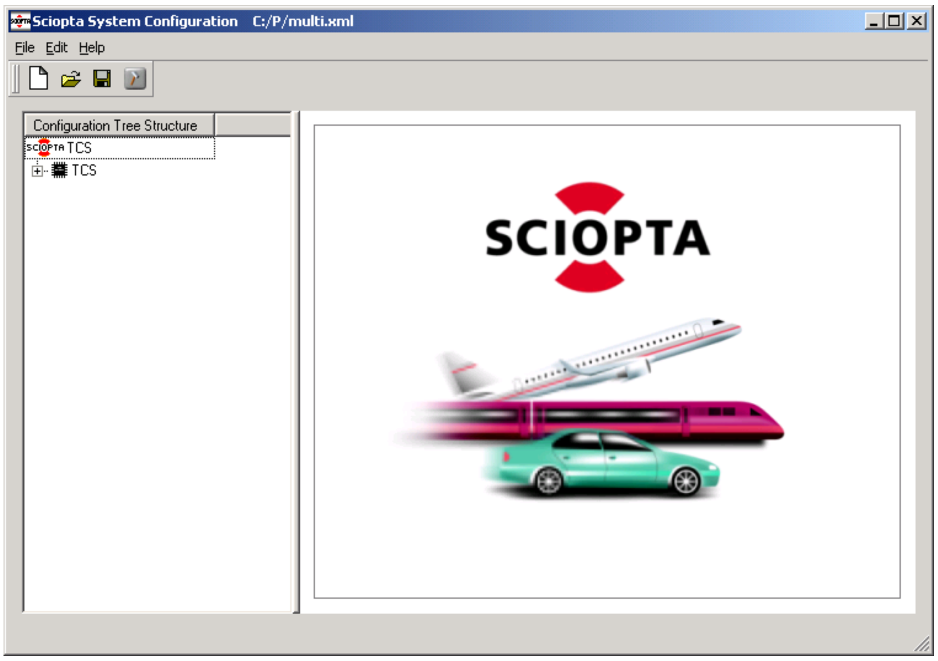

Sciopta System Configuration C:/P/multi.xml
File Edit Help
Configuration Tree Structure
TCS
TCS
SCIOPTA

Figure 16-1: SCIOPTA Configuration Utility Start Screen

16.3 Preference File sc_config.cfg

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

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

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

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

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

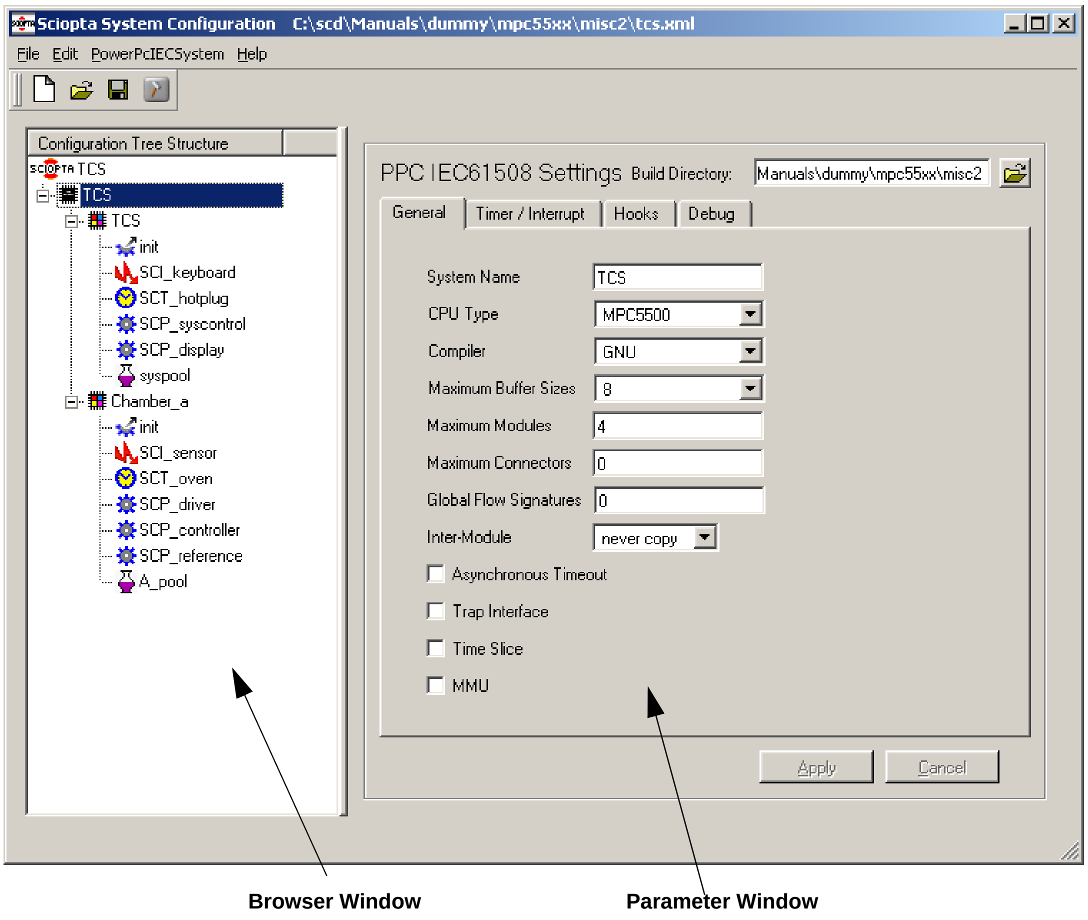
At every project save the file **sc_config.cfg** is updated.

16.4 Project File

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

16.5 SCONF Windows

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



Build Directory:	Manuals\dummy\mpc55xx\misc2
System Name	TCS
CPU Type	MPC5500
Compiler	GNU
Maximum Buffer Sizes	8
Maximum Modules	4
Maximum Connectors	0
Global Flow Signatures	0
Inter-Module	never copy
Asynchronous Timeout	☐
Trap Interface	☐
Time Slice	☐
MMU	☐

Figure 16-2: SCONF Windows

16.5.1 Parameter Window

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

16.5.2 Browser Window

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

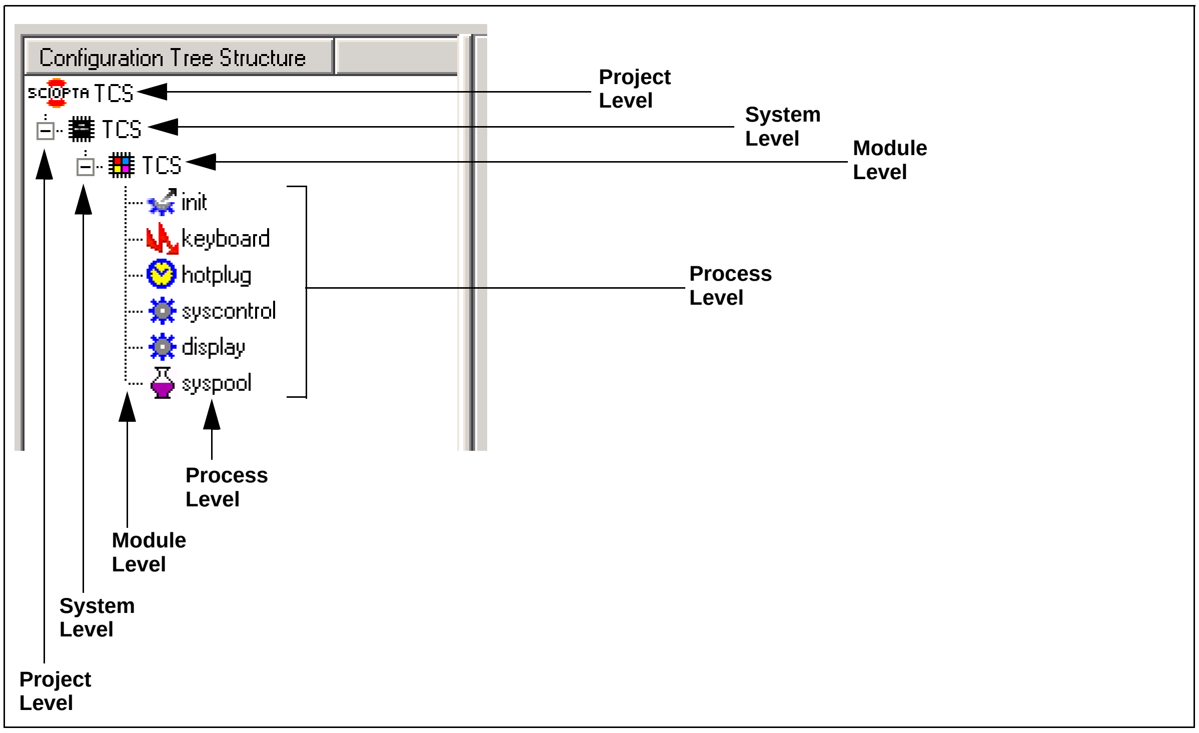


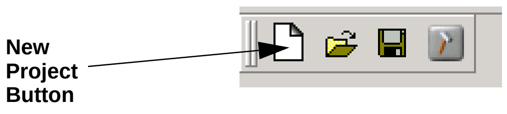
Figure 16-3: Browser Windows

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

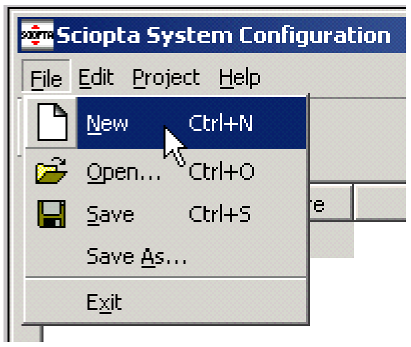
- The uppermost level is the **Project Level** where all project configurations will be done. The project name can be defined and you can create new systems for the project.
- In the **System Level** you are configuring the system for one CPU. You can create the static modules for the system and configure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

16.6 Creating a New Project

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

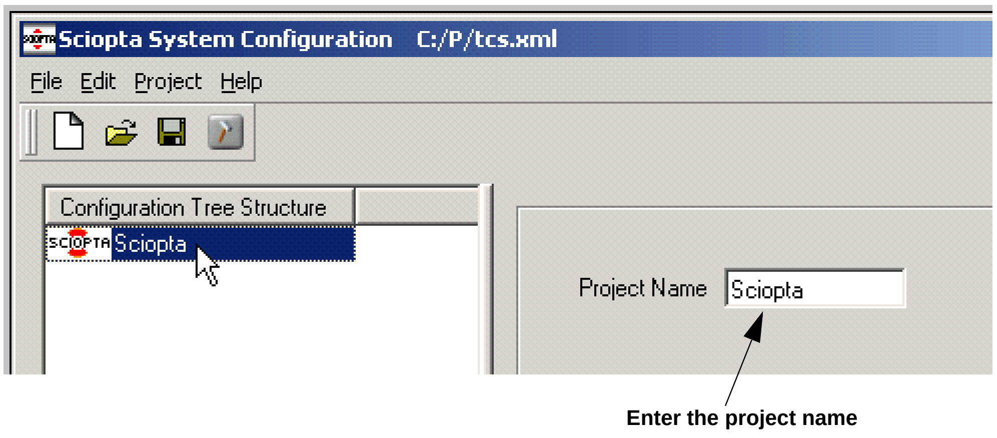


You also can create a new project from the file menu or by the Ctrl+N keystroke:



16.7 Configure the Project

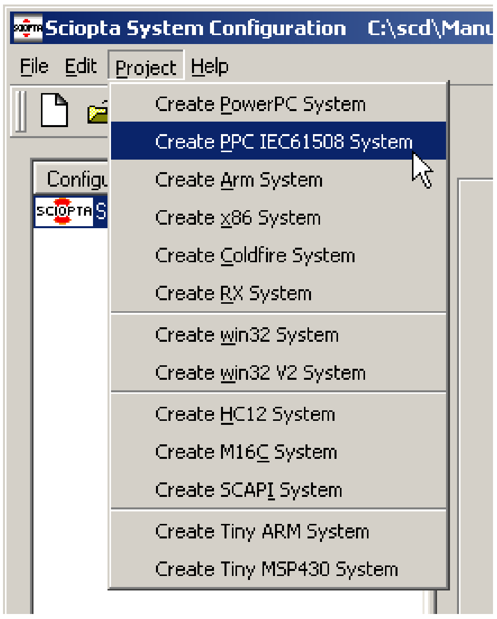
You can define and modify the project name. Click on the project name on the right side of the SCIOPTA logo and enter the project name in the parameter window.



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

16.8 Creating Systems

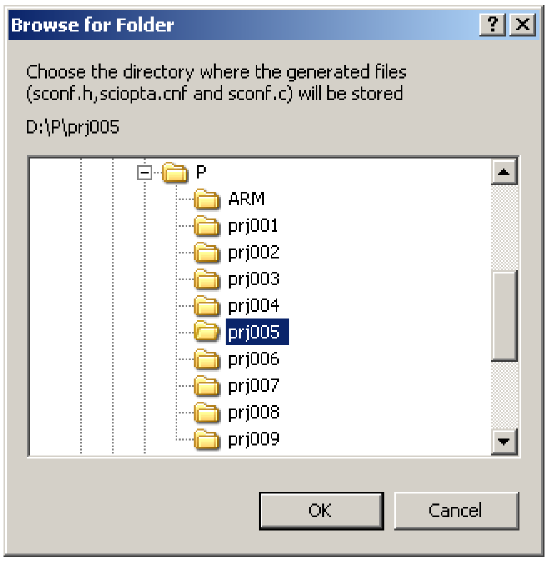
From the project level you can create new systems. Move the mouse pointer over the project and right-click the mouse.



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

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

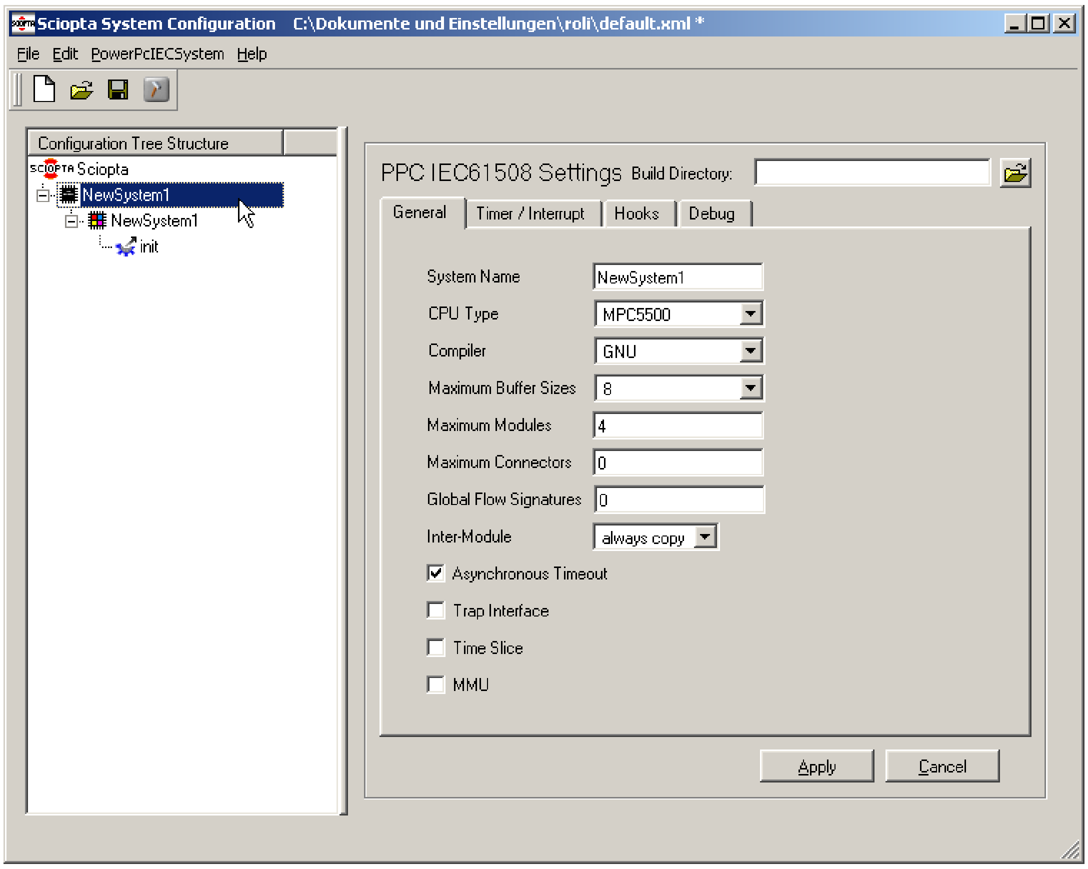
SCONF asks you to enter a directory where the generated files (sconf.h and sconf.c) will be stored:


Browse for Folder

Choose the directory where the generated files
(sconf.h,sciopta.cnf and sconf.c) will be stored

D:\P\prj005

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


Sciopta System Configuration C:\Dokumente und Einstellungen\roli\default.xml *
File Edit PowerPcIECSystem Help
Configuration Tree Structure
Sciopta
NewSystem1
NewSystem1
init
PPC IEC61508 Settings Build Directory:
General
Timer / Interrupt
Hooks
Debug
Apply
Cancel

System Name: NewSystem1
CPU Type: MPC5500
Compiler: GNU
Maximum Buffer Sizes: 8
Maximum Modules: 4
Maximum Connectors: 0
Global Flow Signatures: 0
Inter-Module: always copy
[✓] Asynchronous Timeout
[ ] Trap Interface
[ ] Time Slice
[ ] MMU

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

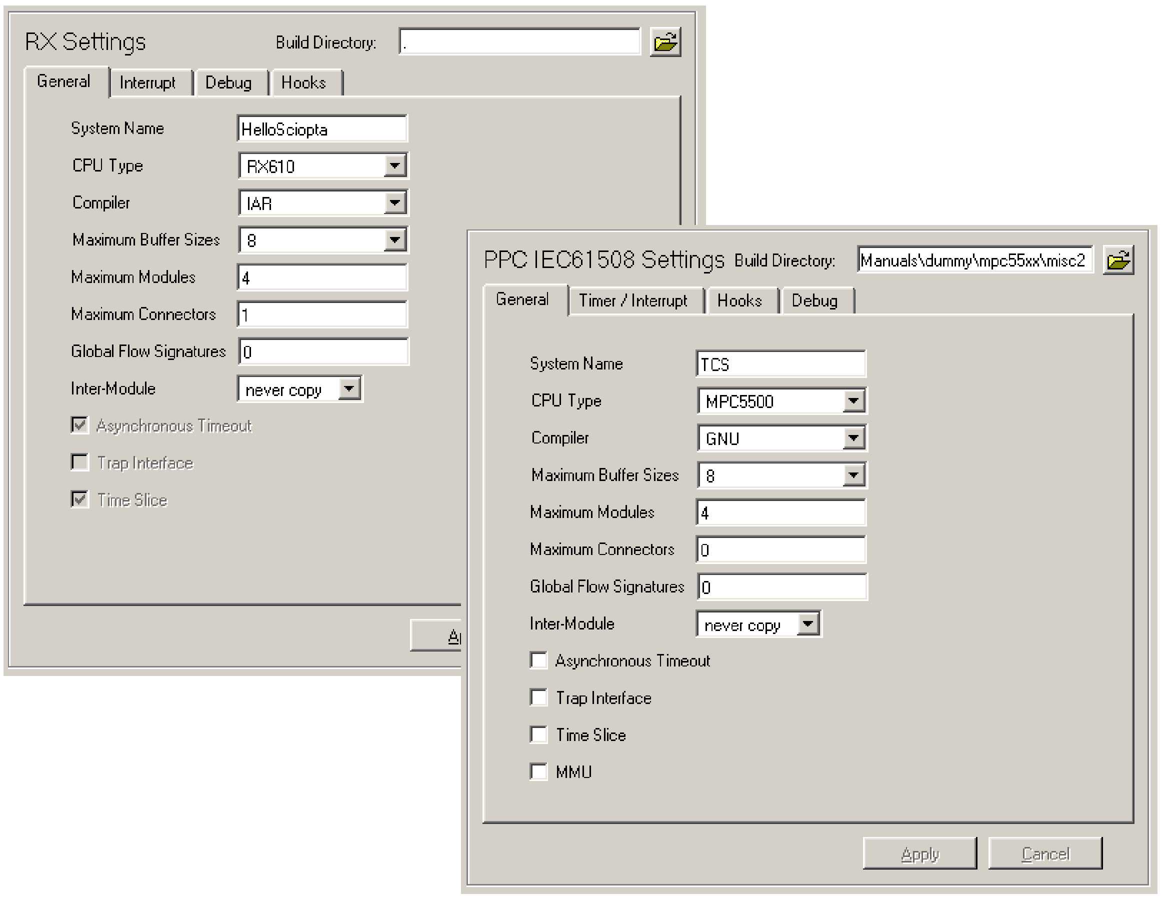
You are now ready to configure the individual targets.

16.9 Configuring Target Systems

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

The system configuration for PowerPCand Renesas RX600 target systems is divided into 4 tabs: General, Timer/ Interrupt, Hooks and Debug.

16.9.1 General System Configuration Tab



Build Directory: .

System Name	HelloSciopta
CPU Type	RX610
Compiler	IAR
Maximum Buffer Sizes	8
Maximum Modules	4
Maximum Connectors	1
Global Flow Signatures	0
Inter-Module	never copy
Asynchronous Timeout	✓
Trap Interface	☐
Time Slice	✓

Build Directory: Manuals\dummy\mpc55xx\misc2

System Name	TCS
CPU Type	MPC5500
Compiler	GNU
Maximum Buffer Sizes	8
Maximum Modules	4
Maximum Connectors	0
Global Flow Signatures	0
Inter-Module	never copy
Asynchronous Timeout	☐
Trap Interface	☐
Time Slice	☐
MMU	☐

16.9.1.1 Parameter

System Name	Name of the target system.
	Enter the name of your system. Please note that the system module (module 0) in this system will use the same name.
CPU Type	CPU family for the selected architecture.

Compiler	C/C++ Compiler selection.
GNU	Also for Windriver/DIAB
IAR	IAR Embedded Workbench for RX600.
HEW	Renesas HEW for RX600

Maximum Buffer Sizes	Maximum number of message buffer sizes.
4, 8 or 16	If a process allocates a message there is also the size to be given. The user just gives the number of bytes needed. SCIOPTA is not returning the exact amount of bytes requested but will select one of a list of buffer sizes which is large enough to contain the requested number. This list can contain 4, 8 or 16 sizes which is configured here in the maximum buffer sizes entry. See 7 “Pools” on page 7-1. **For Renesas RX600 only a setting for 8 buffer sizes is available.**

Maximum Modules	Maximum number of SCIOPTA modules in the system.
	Here you can define a maximum number of modules which can be created in this system. The maximum value is 127 modules. It is important that you give here a realistic value of maximum number of modules for your system as SCIOPTA is initializing some memory statically at system start for the number of modules given here. See also 4 “Modules” on page 4-1.

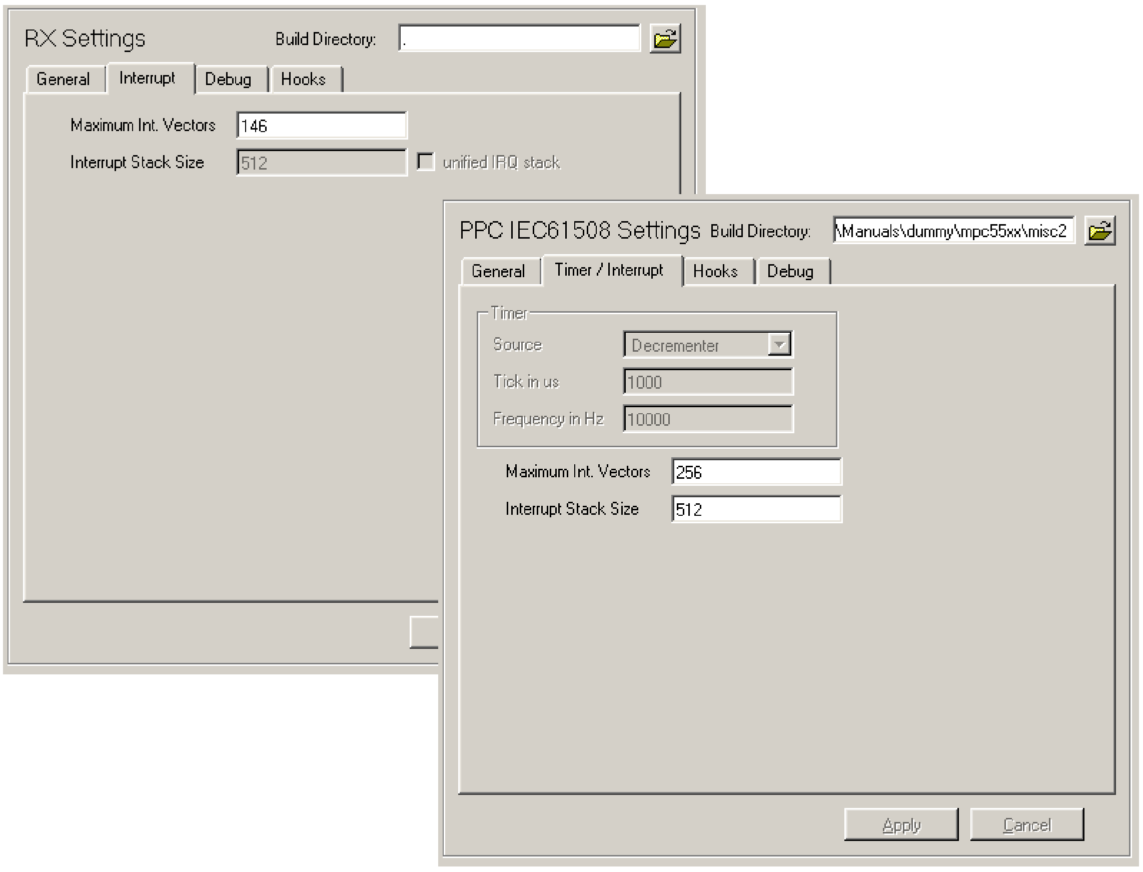
Maximum CONNECTORS	Maximum number of CONNECTOR processes in the system.
	CONNECTORS are specific SCIOPTA processes and responsible for linking a number of SCIOPTA systems. The maximum number of connectors in a system may not exceed 127 which correspond to the maximum number of systems. See also chapter 12.9 “Distributed Systems” on page 12-5.

Global Flow Signatures	Number of global flow control signatures
0	Only supported in the safety kernel. Please consult the SCIOPTA safety manual for more information. Set to 0 for non-safety systems.

Inter-Module	Defines if messages between modules are copied or not.
never copy	Messages between modules are never copied.
always copy	Messages between modules are always copied. Please note: Do not select **friends**. The friend concept is not supported in the kernel version 2.

Asynchronous Timeout	Enables the time-out server. See chapter 9.4 “Timeout Server” on page 9-3. **Allways selected in the Renesas RX600 kernel.**
Trap Interface	Enables the trap interface. See chapter 12.8 “Trap Interface” on page 12-4. **Actually not available in the Renesas RX600 kernel.**
Time Slice	Enables time slice in prioritized processes. **Allways selected in the Renesas RX600 kernel.**
MMU	Select if an MMU is used in the system. **Not available in the Renesas RX600 kernel.**

16.9.2 Timer and Interrupt Configuration Tab



RX Settings

Build Directory: .

General | Interrupt | Debug | Hooks

Maximum Int. Vectors: 146

Interrupt Stack Size: 512

☐ unified IRQ stack

PPC IEC61508 Settings

Build Directory: \Manuals\dummy\mpc55xx\misc2

General | Timer / Interrupt | Hooks | Debug

Timer

Source: Decrementer

Tick in us: 1000

Frequency in Hz: 10000

Maximum Int. Vectors: 256

Interrupt Stack Size: 512

Apply | Cancel

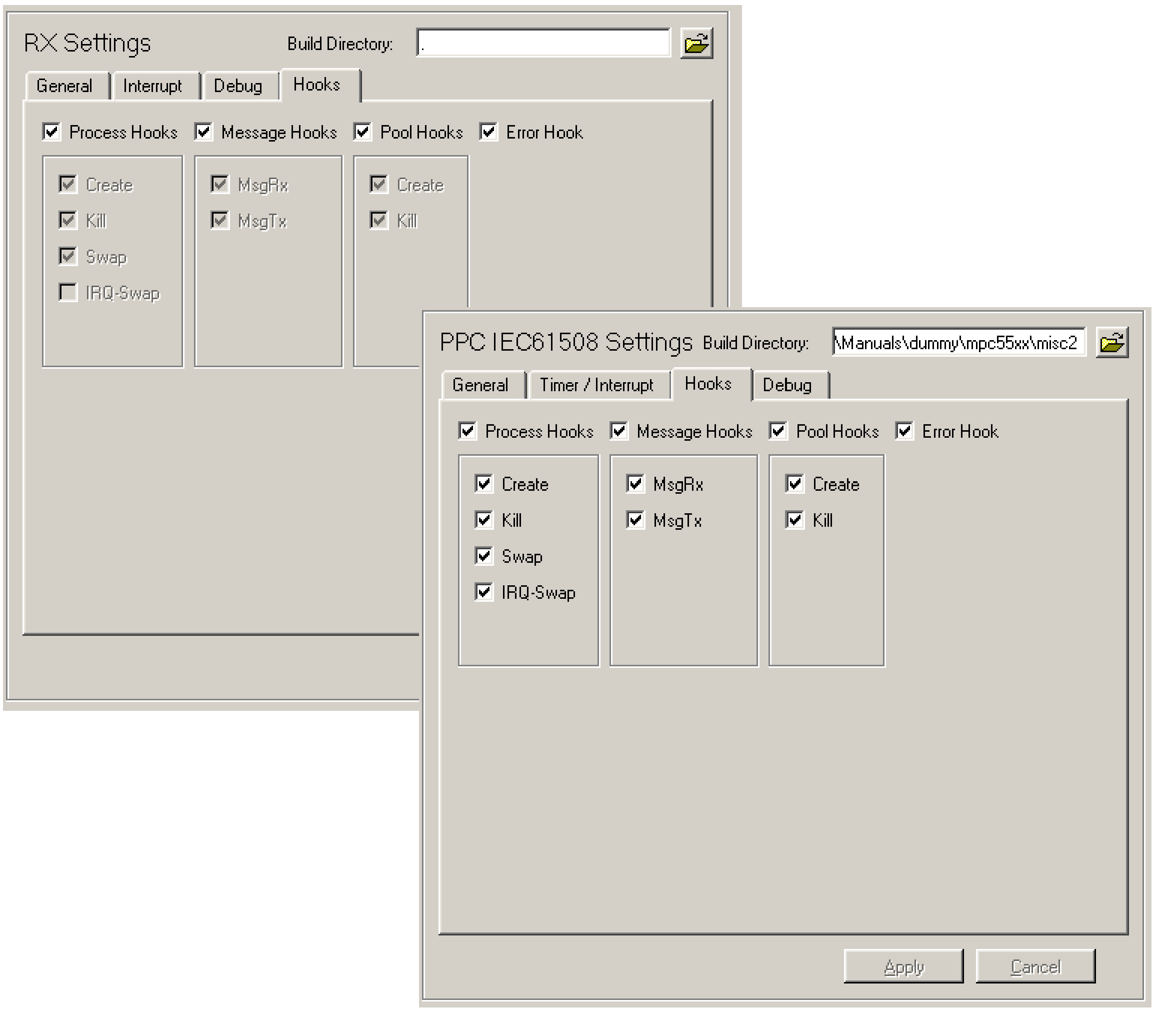
16.9.2.1 Timer Parameter

Timer	Tick timer configuration. **Only for PowerPC**.
	The decrementer timer of the CPU is automatically selected. The tick timer process is included in the file **systick.S**: File location: <installation_folder>\sciopta\<version>\bsp\<arch>\<cpu>\src\ The tick timer values are defined in the file **config.h**: File location: <installation_folder>\sciopta\<version>\bsp\<arch>\<cpu>\<board>\include\

16.9.2.2 Interrupt Parameter

Maximum Int. Vectors	Maximum number of interrupt vectors.
	Enter here the maximum number of interrupt vectors used in your system.
Interrupt Stack Size	Size of interrupt stack.
	You need to define the size of the interrupt stack. **Fix selected to 512 bytes in the Renesas RX600 kernel.**

16.9.3 Hooks Configuration Tab



RX Settings, Hooks tab:

- [✓] Process Hooks: [✓] Create, [✓] Kill, [✓] Swap, [ ] IRQ-Swap
- [✓] Message Hooks: [✓] MsgRx, [✓] MsgTx
- [✓] Pool Hooks: [✓] Create, [✓] Kill
- [✓] Error Hook

PPC IEC61508 Settings, Hooks tab:

- [✓] Process Hooks: [✓] Create, [✓] Kill, [✓] Swap, [✓] IRQ-Swap
- [✓] Message Hooks: [✓] MsgRx, [✓] MsgTx
- [✓] Pool Hooks: [✓] Create, [✓] Kill
- [✓] Error Hook

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

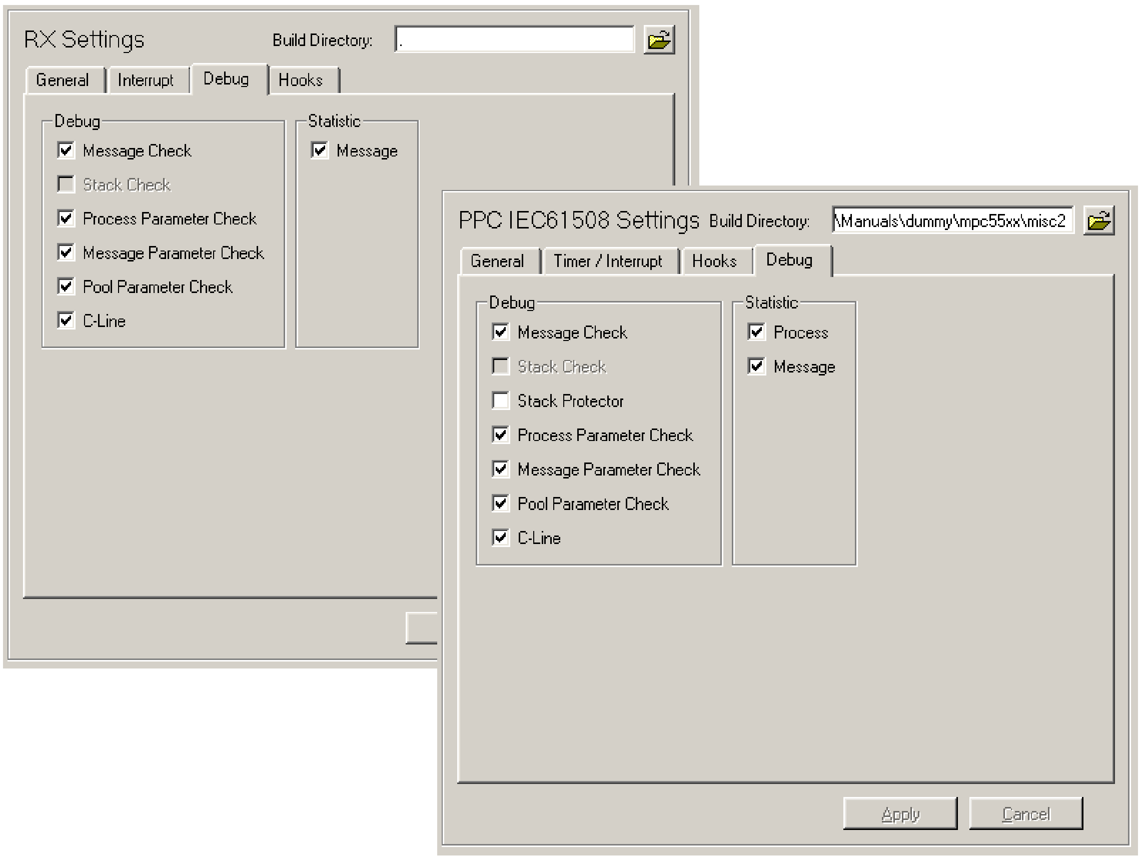
You can enable all existing hooks separately by selecting the corresponding check box.

Please consult chapter 12.2 “Hooks” on page 12-1 for more information about SCIOPTA hooks.

Please Note:

For Renesas RX600 there is only limited hook configuration possible. For each setting a specific kernel object file is needed. Please consult chapter 15.5.1 “Kernel Library Versions” on page 15-4 for possible selections.

16.9.4 Debug Configuration Tab


RX Settings Build Directory: .

General | Interrupt | Debug | Hooks

Debug
- [✓] Message Check
- [ ] Stack Check
- [✓] Process Parameter Check
- [✓] Message Parameter Check
- [✓] Pool Parameter Check
- [✓] C-Line

Statistic
- [✓] Message

PPC IEC61508 Settings Build Directory: \Manuals\dummy\mpc55xx\misc2

General | Timer / Interrupt | Hooks | Debug

Debug
- [✓] Message Check
- [ ] Stack Check
- [ ] Stack Protector
- [✓] Process Parameter Check
- [✓] Message Parameter Check
- [✓] Pool Parameter Check
- [✓] C-Line

Statistic
- [✓] Process
- [✓] Message

16.9.4.1 Debug Parameter

Message Check	Enables the message check functions in the kernel.
	Some test functions on messages will be included in the kernel.
Stack Protector	Enables the stack protection functions. **Only for PowerPC**.
	Allows to check if the stack frame of a called function still fits in the stack. This is compiler specific.
Process Parameter Check	Enables process parameter checks.
	Parameter check of the process system calls will be included in the kernel.
Message Parameter Check	Enables message parameter checks.
	Parameter check of the message system calls will be included in the kernel.

Pool Parameter Check	Enables pool parameter checks.
	Parameter check of the pool system calls will be included in the kernel.
C-Line	Enables C-line informations.
	If you are selecting this check box the kernel will include line number information which can be used by the SCIOPTA DRUID Debug System or an error hook. Line number and file of the last system call is recorded in the per process data.

16.9.4.2 Statistics Parameter

Process Statistics	Includes process statistics. **Only for PowerPC**.
	The kernel will maintain a process statistics data field where information such as number of process swaps can be read.
Message Statistics	Includes message statistics.
	The kernel will maintain a message statistics data field in the pool control block where information such as number of message allocation can be read.

Please Note:

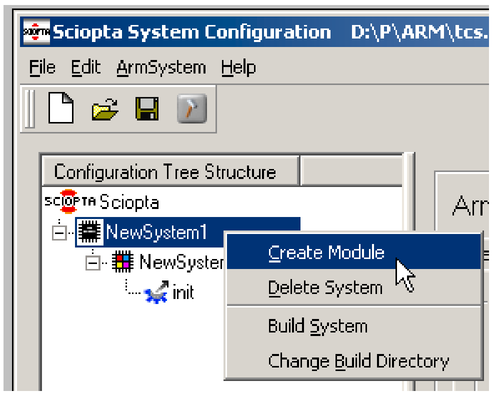
For Renesas RX600 there is only limited debug configuration possible. For each setting a specific kernel object file is needed. Please consult chapter 15.5.1 “Kernel Library Versions” on page 15-4 **for possible selections.**

Applying Target Configuration

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

16.10 Creating Modules

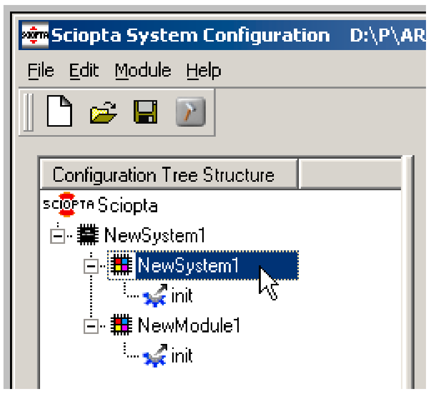
From the system level you can create new modules. Move the mouse pointer over the system and right-click the mouse.



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

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

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

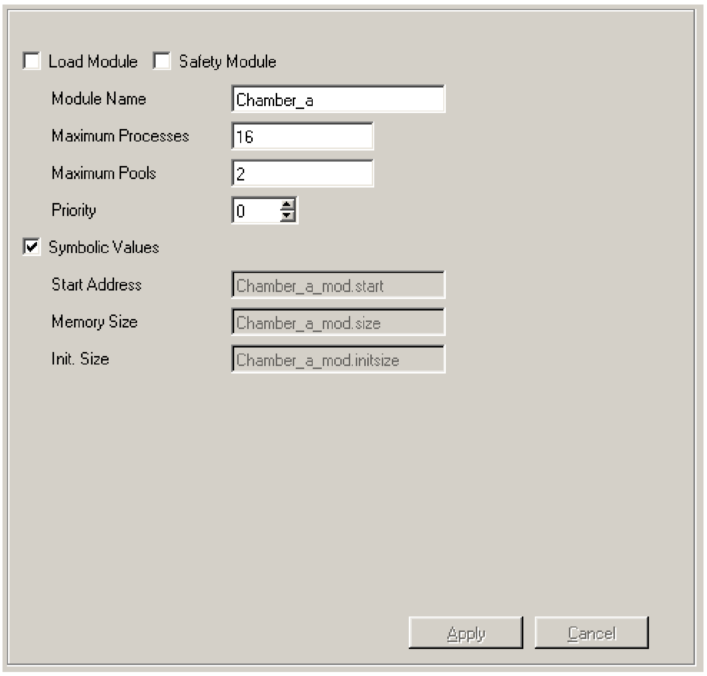


You can create up to 127 modules.

You are now ready to configure the individual modules.

16.11 Configuring Modules

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



☐ Load Module	☐ Safety Module
Module Name	Chamber_a
Maximum Processes	16
Maximum Pools	2
Priority	0
☑ Symbolic Values	
Start Address	Chamber_a_mod.start
Memory Size	Chamber_a_mod.size
Init. Size	Chamber_a_mod.initsize

16.11.1 Parameter

Load Module	Module is a load module.
	Check this box if the module will be loaded at run-time into the system. This check box is not available for the system module.
Safety Module	Module is a safety module.
	Check this box if the module will be certified safety related module. This check box is not available for the system module.
Module Name	Name of the module.
	Enter the name of the module. If you have selected the system module (the first module or the module with the id 0) you cannot give or modify the name as it will have the same name as the target system.
Maximum Processes	Maximum number of processes in the module.
	The kernel will not allow to create more processes inside the module than stated here. The maximum value is 16383.

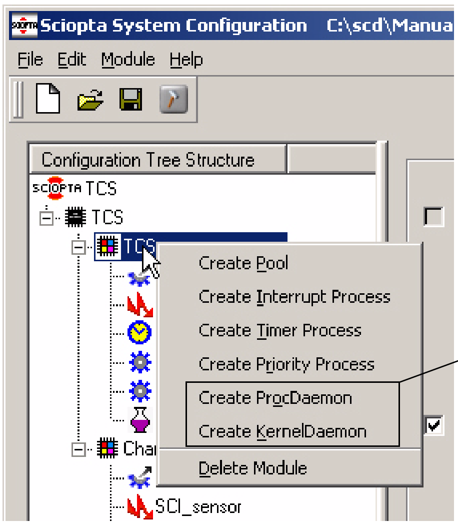
Maximum Pools	Maximum number of message pools.
	Enter the maximum number of pools in the module. The kernel will not allow to create more pools inside the module than stated here. The maximum value is 128.
Priority	Module priority.
	Enter the priority of the module. Each module has a priority which can range between 0 (highest) to 31 (lowest) priority. See also chapter 4.3 “Module Priority” on page 4-1.
Symbolic Values	Module memory map defined by the linker script.
	You need to select this checkbox if you want to specify labels instead of absolute values for the module addresses and module size (start address, memory size and init size). The following labels will be used by the linker script and resolved at link time: <module_name>_mod <module_name>_size <module_name>_initsize) See chapter 15.8.2.3 “Specific Module Values” on page 15-10. Therefore all memory allocation for all modules is controlled by the linker script.
Start Address	Module start address.
	If **Symbolic Values** is not selected the module start address can be defined here. You need to be very carefully when entering an absolute address here and need to check with the linker script to avoid overlapping.
Memory Size	Size of the module.
	If **Symbolic Values** is not selected the module size can be defined here. Please consult chapter 15.8.2.2 “Module Sizes” on page 15-9 for more information about module size calculation.
Init Size	Size of initialized module memory.
	If **Symbolic Values** is not selected the size of the module memory which is initialized by the C-startup function (cstartup.S) can be defined here. You need to be very carefully when entering an absolute address here and need to check with the linker script to avoid overlapping.

Applying Module Configuration

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

16.12 Creating Processes and Pools

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



The process and kernel daemon can only be created in the syse module

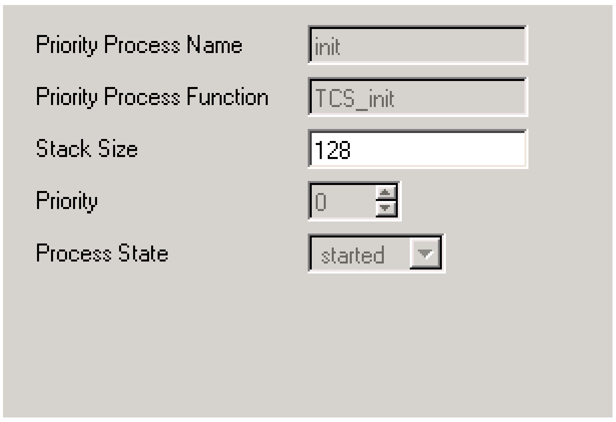
A pop-up menu appears and allows you to create pools, interrupt processes, timer processes, prioritized processes and if it is the system module also the process daemon and the kernel daemon.

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

16.13 Configuring the Init Process

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

Please consult chapter 5.9 “Init Processes” on page 5-8 for more information about init processes.



Priority Process Name	init
Priority Process Function	TCS_init
Stack Size	128
Priority	0
Process State	started

16.13.1 Parameter

Stack Size	Init process stack size.
	Enter a big enough stack size.

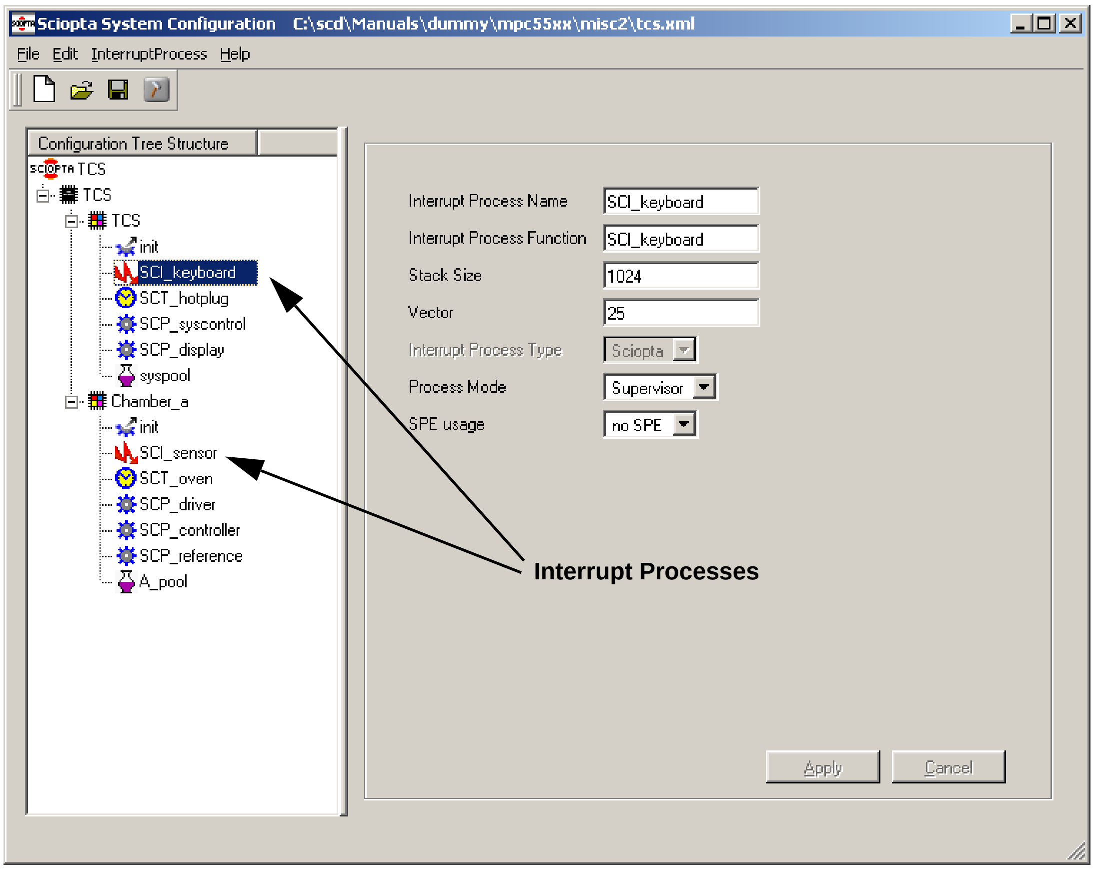
Applying Init Process Configuration

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

16.14 Interrupt Process Configuration

After selecting an interrupt process with your mouse the parameter window on the right side will show the configuration parameters for the interrupt process.

Please consult chapter 5.7 "Interrupt Processes" on page 5-5 for more information about interrupt processes.



Interrupt Process Name	SCI_keyboard
Interrupt Process Function	SCI_keyboard
Stack Size	1024
Vector	25
Interrupt Process Type	Sciopta
Process Mode	Supervisor
SPE usage	no SPE

16.14.1 Parameter

Interrupt Process Name	Interrupt process name.
	It is suggested to use a name such as **SCI_<name>**.
Interrupt Process Function	Interrupt process function entry.
	Function name of the interrupt process function. This is the address where the created process will start execution. More than one interrupt processes (names) can have the same interrupt process function.

Stack Size	Interrupt process stack size.
	Enter a big enough stack size of the created interrupt process in bytes.

Vector	Interrupt vector.
	Enter the interrupt vector connected to the interrupt process.

Processor Mode	Selects interrupt processor mode.
Supervisor	The process runs in CPU supervisor mode.
User	The process runs in CPU user mode.

SPE Usage	Selects if PowerPC Signal Processing Engine is used or not.
no SPE	SPE not used.
SPE	SPE is used.

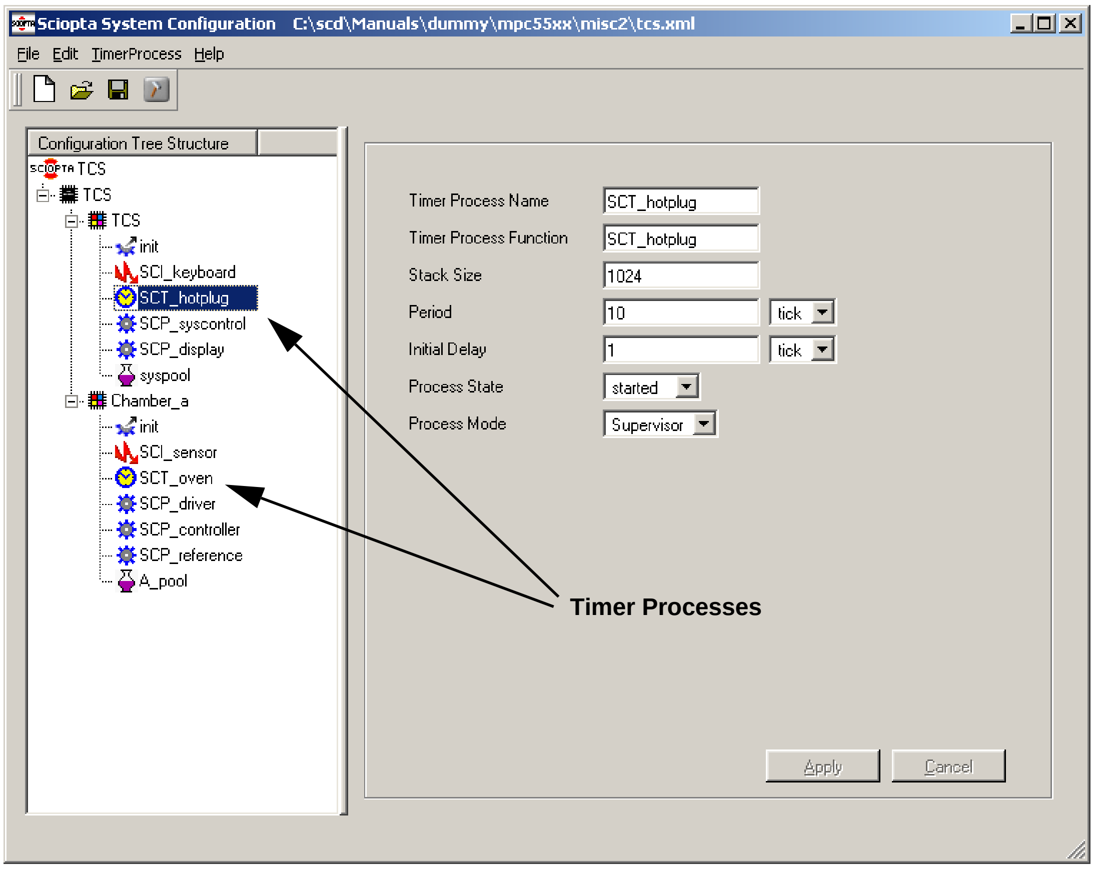
Applying the Interrupt Process Configuration

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

16.15 Timer Process Configuration

After selecting a timer process with your mouse the parameter window on the right side will show the configuration parameters for the timer process.

Please consult chapter 5.8 “Timer Processes” on page 5-7 for more information about timer processes.



Timer Process Name	SCT_hotplug	
Timer Process Function	SCT_hotplug	
Stack Size	1024	
Period	10	tick
Initial Delay	1	tick
Process State	started	
Process Mode	Supervisor	

16.15.1 Parameter

Timer Process Name	Timer process name.
	It is suggested to use a name such as **SCT_<name>**.
Timer Process Function	Timer process function entry.
	Function name of the timer process function. This is the address where the created process will start execution.

Stack Size	Timer process stack size.
	Enter a big enough stack size of the created interrupt process in bytes.

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

Initial Delay	Initial shift delay time.
	Initial delay before the first time call to the timer process in ticks or milliseconds. To avoid tick overload due to timer processes having the same period.

Process State	Starting state of the timer process.
started	The timer process will be started after creation.
stopped	The process is stopped after creation. Use the **sc_procStart** system call to start the process.

Processor Mode	Selects interrupt processor mode.
Supervisor	The process runs in CPU supervisor mode.
User	The process runs in CPU user mode.

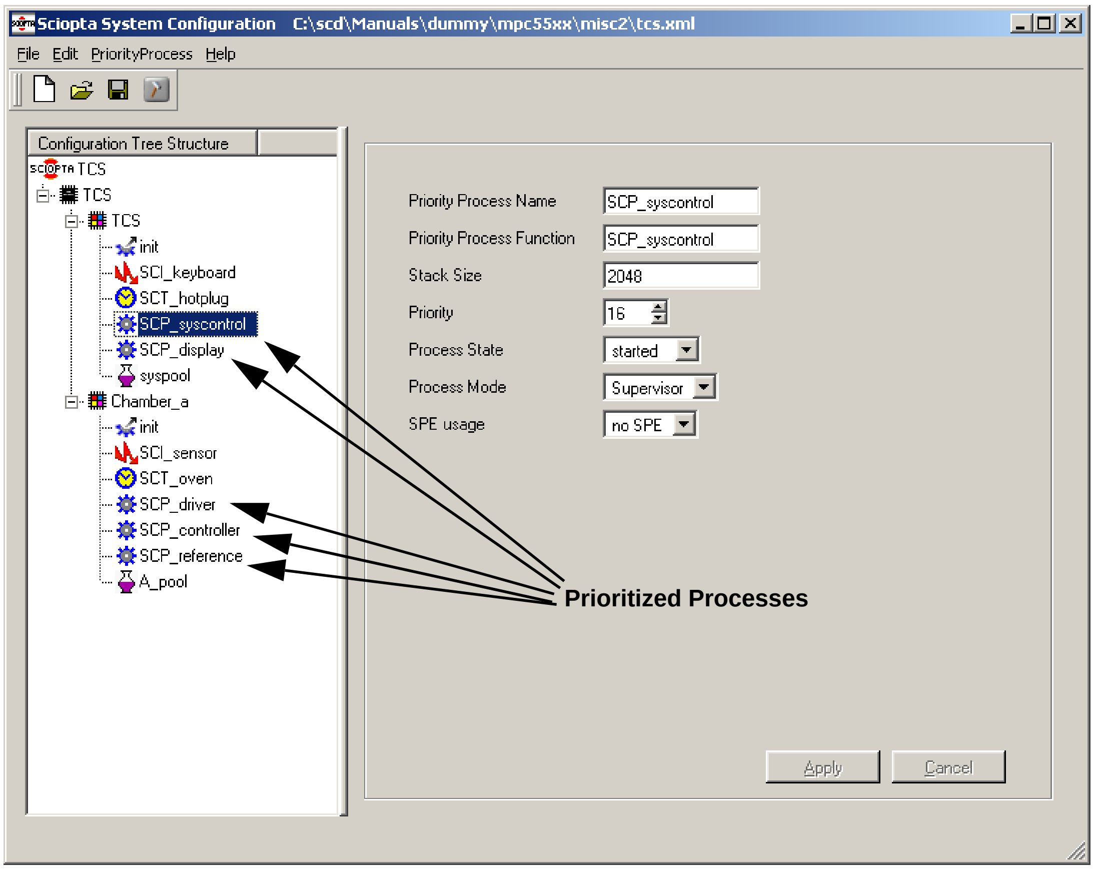
Applying the Timer Process Configuration

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

16.16 Prioritized Process Configuration

After selecting a prioritized process with your mouse the parameter window on the right side will show the configuration parameters for the prioritized process.

Please consult chapter 5.6 “Prioritized Processes” on page 5-3 for more information about timer processes.



Priority Process Name: SCP_syscontrol
Priority Process Function: SCP_syscontrol
Stack Size: 2048
Priority: 16
Process State: started
Process Mode: Supervisor
SPE usage: no SPE

16.16.1 Parameter

Priority Process Name	Process name.
	It is suggested to use a name such as **SCP_<name>**.
Priority Process Function	Process function entry.
	Function name of the prioritized process function. This is the address where the created process will start execution. More than one interrupt processes (names) can have the same interrupt process function.

Stack Size	Process stack size.
	Enter a big enough stack size of the created prioritized process in bytes.

Priority	Priority of the process.
	An error will be generated if the priority is higher than the module priority.

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

Processor Mode	Selects interrupt processor mode.
Supervisor	The process runs in CPU supervisor mode.
User	The process runs in CPU user mode.

SPE Usage	Selects if PowerPC Signal Process Engine is used or not.
no SPE	SPE not used.
SPE	SPE is used.

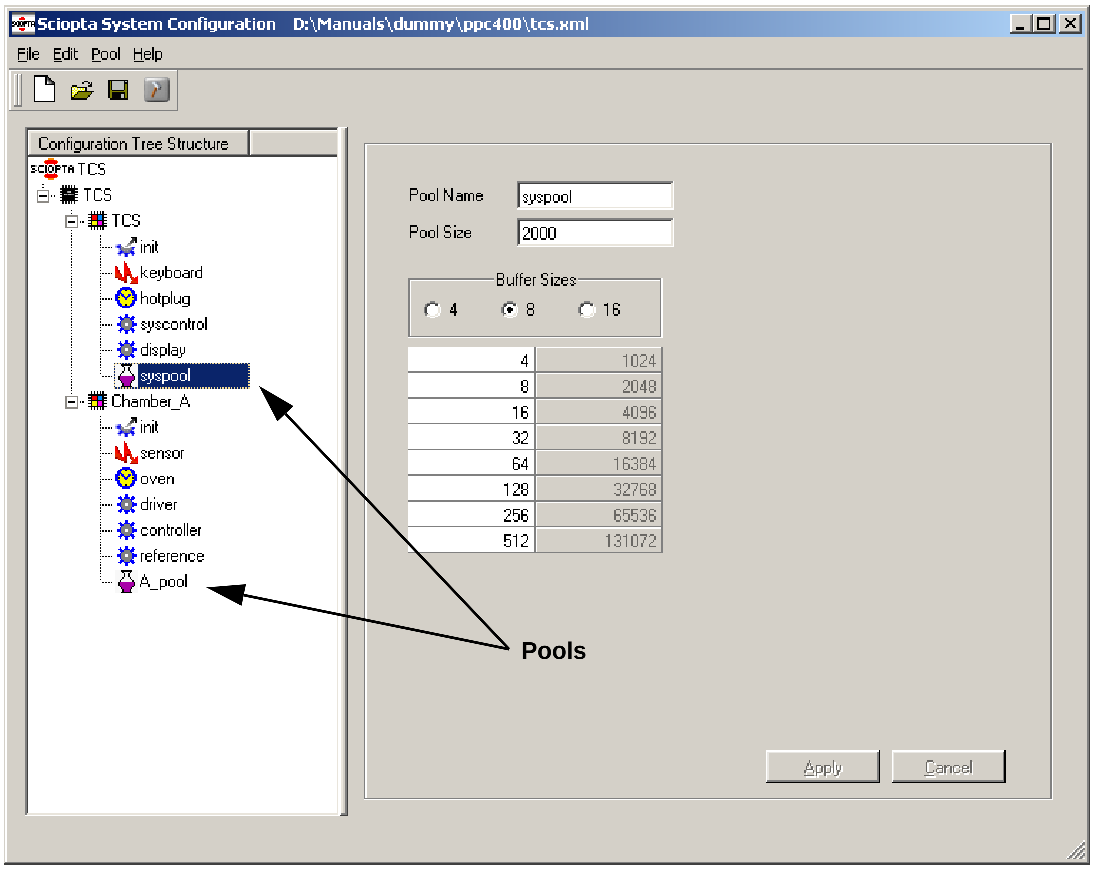
Applying the Priority Process Configuration

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

16.17 Pool Configuration

After selecting a pool with your mouse the parameter window on the right side will show the configuration parameters for the pool.

Please consult chapter 7 “Pools” on page 7-1 for more information about timer processes.



Pool Name: syspool

Pool Size: 2000

Buffer Sizes: 4 [✓ 8] 16

4	1024
8	2048
16	4096
32	8192
64	16384
128	32768
256	65536
512	131072

16.17.1 Parameter

Pool Name	Name of the message pool.
Pool Size	Size of the message pool.
	See chapter 7.2 “Message Pool size” on page 7-1.

Buffer Sizes	Number of buffer sizes.
4, 8 or 16	Define the different buffer sizes for your selection. See chapter 7.3 “Pool Message Buffer Memory Manager” on page 7-2. **For Renesas RX600 only Buffer Sizes of 8 is possible.**

Applying the Pool Configuration

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

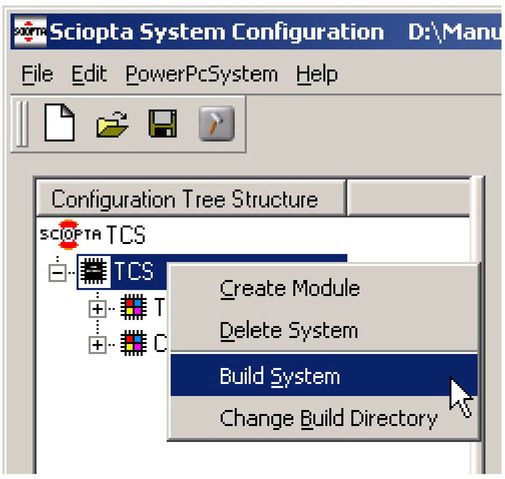
16.18 Build

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

sconf.h	This is a header file which contains some configuration settings. This file will be included by the kernel during assembling. You need to include this into all your files which need configuration information.
sconf.c	This is a C source file which contains the system initialization code. You need to compile this file in the system building process.

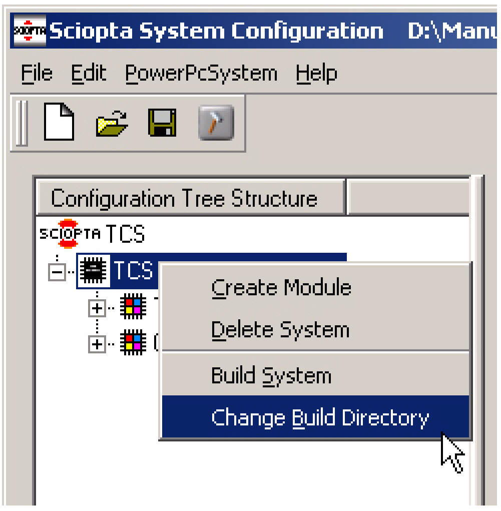
16.18.1 Build System

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


Sciopta System Configuration D:\Manu
File Edit PowerPcSystem Help
Configuration Tree Structure
SCIOPTA TCS
TCS
T
C
Create Module
Delete System
Build System
Change Build Directory

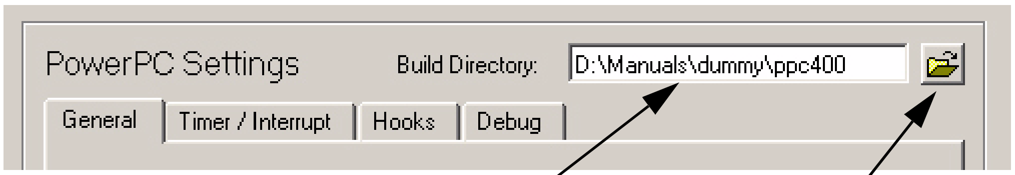
16.18.2 Change Build Directory

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


Sciopta System Configuration D:\Manu
File Edit PowerPcSystem Help
Configuration Tree Structure
TCS
TCS
Create Module
Delete System
Build System
Change Build Directory

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

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



Build Directory: D:\Manuals\dummy\ppc400

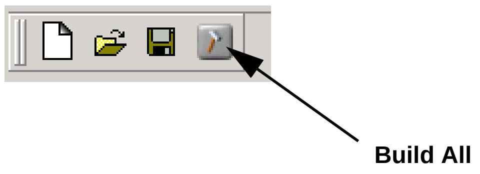
You can change the Build Directory also from the System Settings Window. Click on the Browse Button and select the new directory.

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

16.18.3 Build All

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

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



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

SCONF will prompt for generating the files for each system.

Please note:

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

16.19 Command Line Version

16.19.1 Introduction

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

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

16.19.2 Syntax

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

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

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

17 Manual Versions

17.1 Manual Version 4.1

- Chapter 1.3.2 CPU Families, Renesas RX600 added.
- Chapter 3.5 Getting Started IAR Systems Embedded Workbench, added.
- Chapter 14.3 Architecture System Functions, file location ..\<compiler>\ added.
- Chapter 14.9 Standard RX600 Boards, added.
- Chapter 15.5 Renesas RX600 Kernel, added.
- Chapter 15.6.2 RX600 Architecture Assembler Source Files, added.
- Chapter 15.11 IAR Kernel Libraries, added.
- Chapter 15.13.3 IAR Embedded Workbench, added.
- Chapter 16 SCONF Kernel Configuration, support for Renesas RX600 added.

17.2 Manual Version 4.0

- Support for SCIOPTA Version 2 Kernel.
- SCIOPTA Manual system restructured and rewritten.
- Splitted into a Kernel Reference Manual and User's Manual.
- No more target specific manuals. Target specific information are included in the Kernel Reference Manual and User's Manual.

17.3 Manual Version 3.2

- Chapter 2.4.9 GNU Tool Chain Installation, rewritten due to CodeSourcery GCC tool chain support.
- Chapter 3.2.1 Class Diagram, replaces old block diagram.
- Chapter 3.2.3.1 Equipment, CodeSourcery GCC tool chain now used.
- Chapter 6.6.4 sc_procKill, "The sc_procKill system calls returns before the cleaning work begins." removed from the last paragraph.
- Chapter 13.2.1 Tools, CodeSourcery GCC tool chain now used.
- Chapter 13.3.2 Tools, CodeSourcery GCC tool chain now used.
- Chapter 13.4.2 Tools, CodeSourcery GCC tool chain now used.

17.4 Manual Version 3.1

- Chapter 2.4.10 Eclipse C/C++ Development Tooling - CDT, rewritten.
- Chapter 3 Getting Started, rewritten.
- Chapter 4.15.2 Eclipse, rewritten.
- Chapter 5.2.5 Priorities, upper limit effective priority of 31 described.
- Chapter 6, Application Programming Interface,

- Chapter 6.3.6 sc_msgRx and chapter 6.3.1 sc_msgAlloc, parameter tmo, value modified.
- Chapter 6.4.1 sc_poolCreate, parameter size, pool calculation value n better defined. Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.4.3 sc_poolIdGet, return value “poolID of default pool” added.
- Chapter 6.5.6 sc_procPrioSet, paragraph: “If there are already existing processes at the same priority, the process which has just moved to that priority will be put at the end of the list and swapped-out.” added.
- Chapter 6.6.1 sc_procPrioCreate, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.6.2 sc_procIntCreate, “User” interrupt process type removed, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.6.3 sc_procTimCreate, Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.10.1 sc_moduleIdGet, parameter name text: “...or zero for current module” appended at description and return value for parameter name=NULL added. Return value SC_NOSUCH_MODULE modified to SC_ILLEGAL_MID if Module name was not found.
- Chapter 6.11.1 sc_moduleCreate, parameter textsize renamed into initsize. Parameter name, “Valid characters” modified to “Recommended characters”.
- Chapter 6.12.3 sc_moduleFriendGet, return value “1” modified to “!=”.
- Chapter 6.13.2 sc_tick, last paragraph added.
- Chapter 6.15.2 sc_triggerWait, parameter tmo better described.
- Chapter 6.19.1 sc_miscErrorHookRegister, function and parameter newhook are of type pointer. Global and Module error hook difference described.
- Chapter 6.19.2 sc_msgHookRegister, function and parameter newhook are of type pointer.
- Chapter 6.19.3 sc_poolHookRegister, function and parameter newhook are of type pointer. Global and Module pool hook difference described.
- Chapter 6.19.4 sc_procHookRegister, function and parameter newhook are of type pointer. Global and Module process hook difference described.
- Chapter 7.7.2.7 Example, system call sc_procIdGet parameter corrected.
- Chapter 7.11.4 Error Hook Declaration Syntax, return value “!= 0” modified.
- Chapter 7.11.6 Error Hooks Return Behaviour, added.
- Chapter 9.10 i.MX27 System Functions and Drivers, added.
- Chapter 9.22 LOGIC i.MX27 LITEKIT, added.
- Chapter 13.3 Eclipse IDE and GNU GCC, rewritten.
- Chapter 15.3 Function Codes, code 0x0E, 0x0D, 0x3E, 0x3F, 0x57, 0x5C, 0x5D, 0x5E and 0x5F added.
- Chapter 15.4 Error Codes, code 0x016, 0x017 and 0x018 added.
- Chapter 16.9.3 Debug Configuration, Stack Check added to the list.

17.5 Manual Version 3.0

- Manual restructured and rewritten.

17.6 Manual Version 2.1

- Chapter 2.4 Installation Procedure Windows Hosts, now modified for customer specific deliveries.
- Chapter 2.4.5 SCIOPTA_HOME Environment Variable, UNIX Shell versions removed.

17.7 Manual Version 2.0

- The following manuals have been combinded in this new SCIOPTA ARM - Kernel, User's Guide:
  - SCIOPTA - Kernel, User's Guide Version 1.8
  - SCIOPTA - Kernel, Reference Manual Version 1.7
  - SCIOPTA - ARM Target Manual

17.8 Former SCIOPTA - Kernel, User's Guide Versions

17.8.1 Manual Version 1.8

- Back front page, Litronic AG became SCIOPTA Systems AG.
- Chapter 2.3.4.1 Prioritized Process, icon now correct.
- Chapter 2.3.4.2 Interrupt Process, last paragrpah added.
- Chapter 2.3.4.5 Supervisor Process, rewritten.
- Chapter 2.5.2 System Module, rewritten.
- Chapter 4.11.1 Start Sequence, added.
- Chapter 4.11.3 C Startup, rewritten.
- Chapter 4.11.5 INIT Process, added.
- Chapter 4.11.6 Module Start Function, added.
- Chapter 4.7.2.6 Example, in system call msg_alloc SC_ENDLESS_TMO replaced by SC_DEFAULT_POOL.

17.8.2 Manual Version 1.7

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

17.8.3 Manual Version 1.6

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

17.8.4 Manual Version 1.5

- All **union sc_msg *** changed to **sc_msg_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **union sc_msg **** changed to **sc_msgptr_t** to support SCIOPTA 16 Bit systems (NEAR pointer).

- Manual now splitted into a User’s Guide and Reference Manual.

17.8.5 Manual Version 1.4

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

17.8.6 Manual Version 1.3

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

17.8.7 Manual Version 1.2

- Top cover back side: Address of SCIOPTA France added.
- Chapter 2.6 Trigger, second paragraph: At process creation the value of the trigger is initialized to **one**.
- Chapter 2.6 Trigger, third paragraph: The **sc_triggerWait**() call decrements the value of the trigger and the calling process will be blocked and swapped out if the value gets negative **or equal zero**.
- Chapter 2.7 Process Variables, second paragraph: The tag and the process variable have a fixed size **large enough to hold a pointer**.
- Chapter 2.7 Process Variables, third paragraph: Last sentence rewritten.
- Chapter 4.5.3.1 Interrupt Process Declaration Syntax, irg_src is of type **int** added.
- Chapter 4.5.6 Idle Process, added.

- Chapter 4.10.4 Error Hook Declaration Syntax, Parameter **user** : user != 0 (User error).
- System call **sc_procRegisterDaemon** changed to **sc_DaemonRegister** and **sc_procUnregisterDaemon** changed to **sc_procDaemonUnregister**.
- System call sc_miscErrorHookRegister, return values better specified.
- System call sc_moduleCreate, parameter **size** value "code" added in Formula.
- System call sc_moduleNameGet, return value **NULL** added.
- System call sc_msgAcquire, condition modified.
- System Call sc_msgAlloc, **SC_DEFAULT_POOL** better specified.
- Systme Call sc_msgHookRegister, description modified and return value better specified.
- System call sc_msgRx, parameters better specified.
- System call sc_poolHookRegister, return value better specified.
- System call sc_procHookRegister, return value better specified.
- System call sc_procIdGet, last paragraph in **Description** added.
- System calls sc_procVarDel, sc_procVarGet and procVarSet, return value **!=0** introduced.
- Chapter 7.3 Function Codes, errors **0x38** to **0x3d** added.
- System call **sc_procUnobserve** added.
- Chapters 2.5.2 System Module and 4.3 Modules, the following sentence was removed: The system module runs always on supervisor level and has all access rights.
- Chapter 2.5.3 Messages and Modules, third paragraph rewritten.
- Chapter 6.31 sc_msgTx, fifth paragraph rewritten.

17.8.8 Manual Version 1.1

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

17.8.9 Manual Version 1.0

Initial version.

17.9 Former SCIOPTA - Kernel, Reference Manual Versions

17.9.1 Manual Version 1.7

- Back front page, Litronic AG became SCIOPTA Systems AG.
- Chapter 3.24 sc_msgHookRegister, text: There can be one module message hook per module replaced by: There can be one module message hook of each type (transmitt/receive) per module.
- Chapter 3.27 sc_msgRx, **flag** parameter **SC_MSGRX_NOT** : text: An array of messages is given which will be excluded from receive replaced by: An array of message ID's is given which will be excluded from receive.
- Chapter 3.47 sc_procNameGet and chapter 3.49 sc_procPathGet, chapter "**Return Value**" rewritten.

17.9.2 Manual Version 1.6

- Chapter 3.27 sc_msgRx, **tmo** parameter, SC_TMO_NONE replaced by SC_NO_TMO. Parameter better specified.
- Chapter 3.27 sc_msgRx, **wanted** parameter, NULL replaced by SC_MSGRX_ALL.
- Chapter 3.7 sc_miscError, **err** parameter, bits 0, 1 and 2 documented.
- Chapter 3.44 sc_procIdGet, if paramter **path** is NULL and parameter **tmo** is SC_NO_TMO this system call returns the callers process ID.

17.9.3 Manual Version 1.5

- All **union sc_msg *** changed to **sc_msg_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **union sc_msg **** changed to **sc_msgptr_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- Chapter 6, System Call Reference, page layout for all system calls modified.
- Chapter 6.81 sc_triggerWait, third paragraph rewritten.
- Chapters 6.9 sc_moduleCreate and 6.17 sc_moduleKill, information about **sc_kerneld** are given.
- Chapter 6.44 sc_procIdGet, added text: this parameter is not allowed if **asynchronous timeout** is disabled at system configuration (**sconf**).
- Manual split into a User's Guide and a Reference Manual.

17.9.4 Manual Version 1.4

- Chapter 4.7.3.2 Example, OS_INT_PROCESS changed into correct SC_INT_PROCESS.
- Chapter 2.3.4.4 Init Process, rewritten.
- Chapter 4.5 Processes, former chapters **4.5.6 Idle Process** and **4.5.7 Supervisor Process** removed.
- Chapter 4.5.1 Introduction, last paragraph about supervisor processes added.
- Chapter 4.5.5 Init Process, rewritten.
- Chapter 6.8 sc_miscErrorHookRegister, syntax corrected.
- Chapter 6.21 sc_mscAlloc, time-out parameter **tmo** better specified.
- Chapter 6.27 sc_msgRx, time-out parameter **tmo** better specified.

- Chapter 4.10.4 Error Hook Declaration Syntax, user !=**0** user error.
- Chapter 4.9 SCIOPTA Daemons, moved from chapter 2.9 and rewritten.
- Chapter 6.41 sc_procDaemonRegister, last paragraph of the description rewritten.
- Chapters 6.45 sc_procIntCreate, 6.46 sc_procKill, 6.51 sc_procPrioCreate, 6.60 sc_procTimCreate and 6.62 sc_procUsrIntCreate, information about **sc_kerneld** are given.
- Chapter 4.10.5 Example, added.

17.9.5 Manual Version 1.3

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

17.9.6 Manual Version 1.2

- Top cover back side: Address of SCIOPTA France added.
- Chapter 2.6 Trigger, second paragraph: At process creation the value of the trigger is initialized to **one**.
- Chapter 2.6 Trigger, third paragraph: The **sc_triggerWait**() call decrements the value of the trigger and the calling process will be blocked and swapped out if the value gets negative **or equal zero**.
- Chapter 2.7 Process Variables, second paragraph: The tag and the process variable have a fixed size **large enough to hold a pointer**.
- Chapter 2.7 Process Variables, third paragraph: Last sentence rewritten.
- Chapter 4.5.3.1 Interrupt Process Declaration Syntax, irg_src is of type **int** added.
- Chapter 4.5.6 Idle Process, added.
- Chapter 4.10.4 Error Hook Declaration Syntax, Parameter **user** : user != 0 (User error).
- System call **sc_procRegisterDaemon** changed to **sc_DaemonRegister** and **sc_procUnregisterDaemon** changed to **sc_procDaemonUnregister**.
- System call sc_miscErrorHookRegister, return values better specified.
- System call sc_moduleCreate, parameter **size** value “code” added in Formula.
- System call sc_moduleNameGet, return value **NULL** added.
- System call sc_msgAcquire, condition modified.
- System Call sc_msgAlloc, **SC_DEFAULT_POOL** better specified.
- Systme Call sc_msgHookRegister, description modified and return value better specified.
- System call sc_msgRx, parameters better specified.
- System call sc_poolHookRegister, return value better specified.

- System call sc_procHookRegister, return value better specified.
- System call sc_procIdGet, last paragraph in **Description** added.
- System calls sc_procVarDel, sc_procVarGet and procVarSet, return value **!=0** introduced.
- Chapter 7.3 Function Codes, errors **0x38** to **0x3d** added.
- System call **sc_procUnobserve** added.
- Chapters 2.5.2 System Module and 4.3 Modules, the following sentence was removed: The system module runs always on supervisor level and has all access rights.
- Chapter 2.5.3 Messages and Modules, third paragraph rewritten.
- Chapter 6.31 sc_msgTx, fifth paragraph rewritten.

17.9.7 Manual Version 1.1

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

17.9.8 Manual Version 1.0

Initial version.

17.10 Former SCIOPTA ARM - Target Manual Versions

17.10.1 Manual Version 2.2

- Back front page, Litronic AG became SCIOPTA Systems AG.
- Chapter 2.2 The SCIOPTA ARM Delivery and chapter 2.4.1 Main Installation Window, tiny kernel added.
- Chapter 3 Getting Started, in the example folder, additional directories for boards have been introduced.
- Chapter 3 Getting Started, the Eclipse project files and the file **copy_files.bat** are now stored in the "\phyCore2294" board sub-directory of the example folder.
- Chapter 3 Getting Started, the SCIOPTA SCONF configuration file is now called **hello.xml** (was hello_phyCore2294.xml before).
- Chapter 5.8.3 Assembling with IAR Systems Embedded Workbench, added.
- Chapter 5.10.3 Compiling with IAR Systems Embedded Workbench, added.
- Chapter 5.12.3 Linking with IAR Systems Embedded Workbench, added.

- Chapter 5.13.1.1 Memory Regions, last paragraph added.
- Chapter 5.13.1.2 Module Sizes, name is now **<module_name>_size** (was <module_name>_free before).
- Chapter 5.13.3 IAR Systems Embedded Workbench Linker Script, added.
- Chapter 5.14 Data Memory Map, redesigned and now one memory map for all environments.
- Chapter 5.14.4 IAR Systems Embedded Workbench©, added.
- Chapter 6 Board Support Packages, file lists modified for SCIOPTA ARM version 1.7.2.5
- Chapter 6.3 ATMEL AT96SAM7S-EK Board, added.
- Chapter 6.4 ATMEL AT96SAM7X-EK Board, added.
- Chapter 6.5 IAR Systems STR711-SK Board, added.

17.10.2 Manual Version 2.1

- Chapter 1.1 About this Manual, SCIOPTA product list updated.
- Chapter 2.4.1 Main Installation Window, Third Party Products, new version for GNU Tool Chain (version 1.4) and MSys Build Shell (version 1.0.10).
- Chapter 2.4.7 GNU Tool Chain Installation, new GCC Installation version 1.4 including new gcc version 3.4.4, new binutils version 2.16.1 and new newlib version 1.13.1. The installer creates now two directories (and not three).
- Chapter 2.4.8 MSYS Build Shell, new version 1.0.10.
- Chapter 3, Getting Started: Equipment, new versions for GNU GCC and MSys.
- Chapter 3, Getting Started: List of copied files (after executed copy_files.bat) removed.
- Chapter 3.5.1 Description (Web Server), paragraph rewritten.
- Chapter 3.13.2.1 Equipment, serial cable connection correctly described.
- Chapter 3.13.2.2 Step-By-Step Tutorial, DRUID and DRUID server setup rewritten.
- Chapter 5.16 Integrated Development Environments, new chapter.

17.10.3 Manual Version 2.0

- Manual rewritten.
- Own manual version, moved to version 2.0

17.10.4 Manual Version 1.7.2

- Installation: all IPS Applications such as Web Server, TFTP etc. in one product.
- Getting started now for all products.
- Chapter 4, Configuration now moved into Kernel User’s Guide.
- New BSP added: Phytec phyCORE-LPC2294.
- Uninstallation now separately for every SCIOPTA product.
- Eclipse included in the SCIOPTA delivery.

- New process SCP_proxy introduced in Getting Started - DHCP Client Example.
- IPS libraries now in three verisons (standard, small and full).

17.10.5 Manual Version 1.7.0

- All **union sc_msg *** changed to **sc_msg_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **union sc_msg **** changed to **sc_msgptr_t** to support SCIOPTA 16 Bit systems (NEAR pointer).
- All **sdd_obj_t *** changed to **sdd_obj_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **sdd_netbuf_t *** changed to **sdd_netbuf_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **sdd_objInfo_t *** changed to **sdd_objInfo_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **ips_dev_t *** changed to **ips_dev_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **ipv4_arp_t *** changed to **ipv4_arp_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **ipv4_route_t *** changed to **ipv4_route_t NEARPTR** to support SCIOPTA 16 Bit systems.
- IAR support added in the kernel.
- Web server modifiied.
- TFTP server added (in addition to client).
- DHCP server added (in addition to client).
- DRUID System Level Debugger added.

18 Index