SCIOPTA

High Performance
Real-Time Operating Systems

ARM
IPS
Internet Protocols

User's Guide

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. S07008RL1

Table of Contents

1 Introduction

1.1 SCIOPTA ARM Real-Time Operating System

SCIOPTA ARM is a high performance real-time operating system for microcontrollers using the ARM cores ARM7TDMI, ARM7TDMIS, ARM966E-S, ARM940T, ARM946E-S, ARM720T, ARM920T, ARM922T, ARM926E-JS and other derivatives of the ARM 7/9 family. Including:

Atmel AT91SAM
AT91SAM7S, AT91SAM7SE, AT91SAM7X, AT91SAM9 and all other ARM7/9 based microcontrollers.

NXP LPC2000
LPC21xx, LPC22xx, LPC212x, LPC23xx, LPC24xx, LPC28xx, LPC3180, and all other ARM7/9 based microcontrollers.

Sharp ARM7 and ARM9 MCU and SoC
LH754xx, LH795xx, LH7A4xx and all other ARM7/9 based microcontrollers.

STMicroelectronics STR7 and STR9
STR71x, STR72x, STR75x, STR91x and all other ARM7/9 based microcontrollers.

Including all microcontrollers from other suppliers which have ARM7/9 cores.

The operating system environment includes:

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

1.2 About This Manual

The SCIOPTA Real-time Operating System includes a TCP/IP communication stack (SCIOPTA IPS) specifically designed for embedded systems.

The purpose of this SCIOPTA ARM - IPS Internet Protocols - User´s Guide is to give all needed information how to use the SCIOPTA TCP/IP communication stack in an ARM embedded project.

After an introduction into the techniques and concepts used in SCIOPTA IPS, information about the interfaces to access the IPS functionality are given. Furthermore you will find useful information about system design and configuration. You can also find a complete description of the IPS Function Interface.

Please consult also the SCIOPTA ARM - Kernel, User’s Guide.

1.3 IPS Overview

The IPS Internet Protocol Suite allows embedded systems running SCIOPTA to communicate with other embedded systems or computers running SCIOPTA or different operating systems.

SCIOPTA IPS is a high performance TCP/IP communication stack for embedded systems. IPS supports TCP, UDP, ICMP, IGMP, IP, IGMP, Fragmentation, PPP and Ethernet. This product has been specially designed to meet the requirement of modern internet protocol network applications in embedded systems. This gives IPS the advantages over traditional internet stacks, of having higher performance and a lower memory footprint.

SCIOPTA IPS is a scalable stack. Memory footprint and RAM needs depend on the IPS configuration and can be adapted to specific applications needs.

1.4 IPS Protocol Layers

The SCIOPTA IPS internet protocols are developed in layers the same way as all standard TCP/IP protocols. Each layer is responsible for a different task of the communications. The IPS protocol suite is the combination of specific protocols at different layers. IPS can be considered to be a four-layer system.

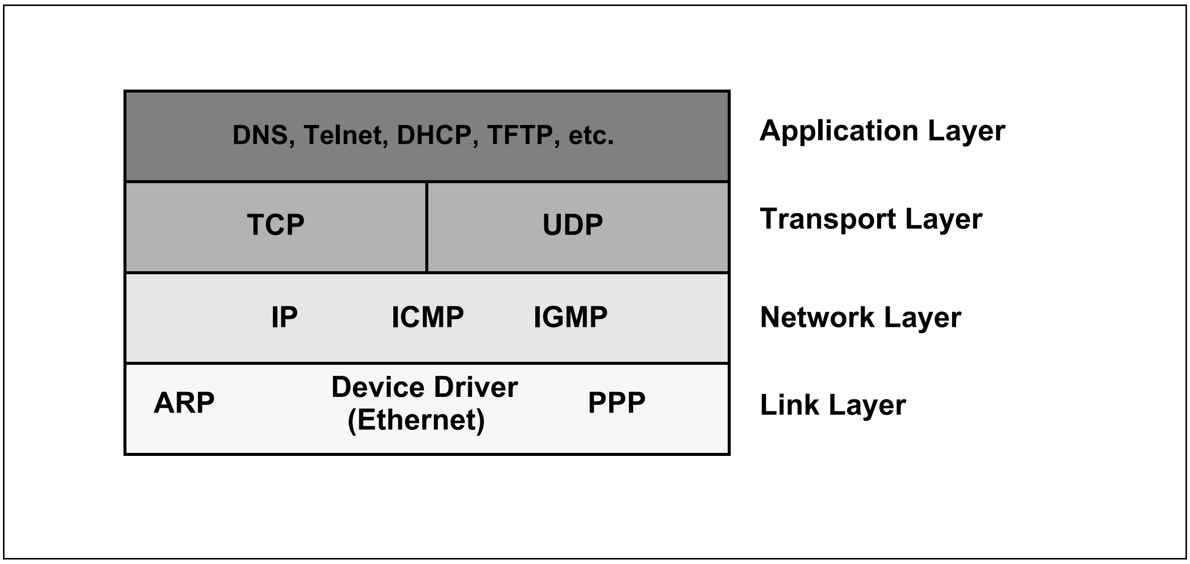


Figure 1-1: IPS Protocol Layers

1.4.1 Application Layer

The application layer controls and manages a specific application. There are many standard applications available for SCIOPTA IPS such as DNS, Telnet, DHCP, TFTP etc. For each application there is a specific SCIOPTA IPS manual available.

1.4.2 Transport Layer

The transport layer is responsible to provide a flow of data between two systems for the application on the upper layer. As in all TCP/IP stack, there are two different transport protocols available in IPS. The User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP).

1.4.2.1 User Datagram Protocol

UDP provides a quite simple service to the application layer by just sending packets of data called datagrams from one system to the other. UDP provides no reliability, there is no guarantee that the datagrams reach the other end. The application layer must provide any desired reliability.

1.4.2.2 Transmission Control Protocol

TCP on the other hand provides a reliable flow of data between two systems. It handles functions such as

- setting timeouts to make sure the other system acknowledges packages that are sent,
- acknowledging received packages,
- dividing the data passed to it from the application into data sizes appropriate for the layer below,
- and many other things.

As TCP provides reliable data package flow, the application layer does not need to add reliability.

1.4.3 Network Layer

The network layer manages the sending and transmitting of packages around the network. For example, the routing is handled by the network layer. Internet Protocol (IP), Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP) are provided in the network layer.

1.4.4 Link Layer

The link layer includes the device driver and the network device (such as Ethernet for example). The link layer handles all the hardware details of physically interfacing with the network media (such as the cable).

The Address Resolution Protocol (ARP) is included in the link layer and is used to convert between the IP address and network interface address.

There is also a Point-to-Point Protocol (PPP) available for SCIOPTA IPS which is placed in the link layer. PPP provides a way to encapsulate IP datagrams on a serial link. Please consult the SCIOPTA - IPS PPP, User´s Guide and Reference Manual for more information about PPP.

1.5 IPS Processes

All parts of the SCIOPTA IPS stack are encapsulated in SCIOPTA static or dynamic processes which can be started and stopped individually. This gives a highly modular design which can be scaled for specific applications.

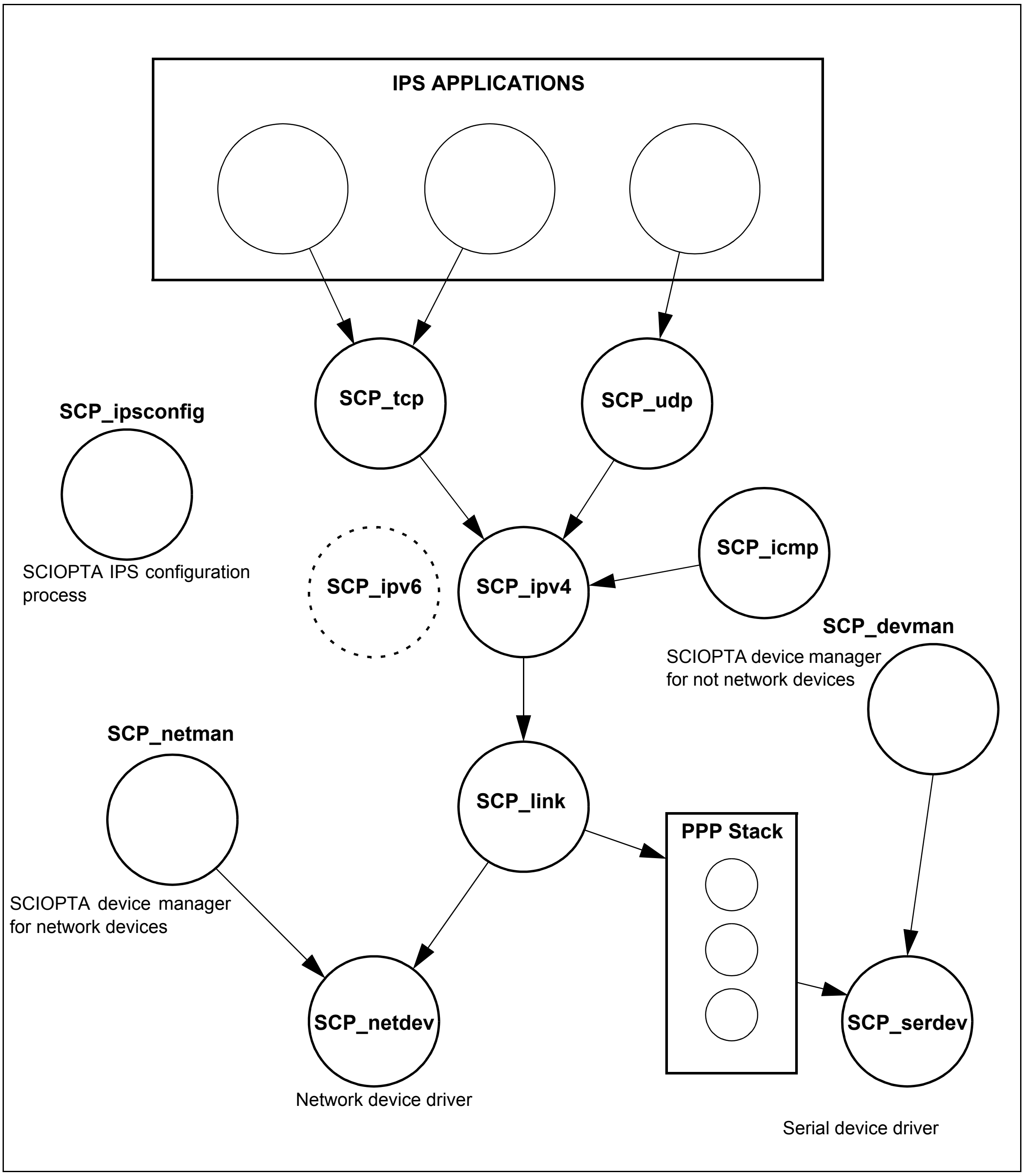


Figure 1-2: IPS Processes

1.6 Using IPS

The SCIOPTA IPS processes manage the protocols, add and maintain the protocol headers and control the interfaces. During the protocol management and while the data transfer occurs the user can still perform some concurrent work. The IPS function interface library must be linked to the process which need to access the IPS stack.

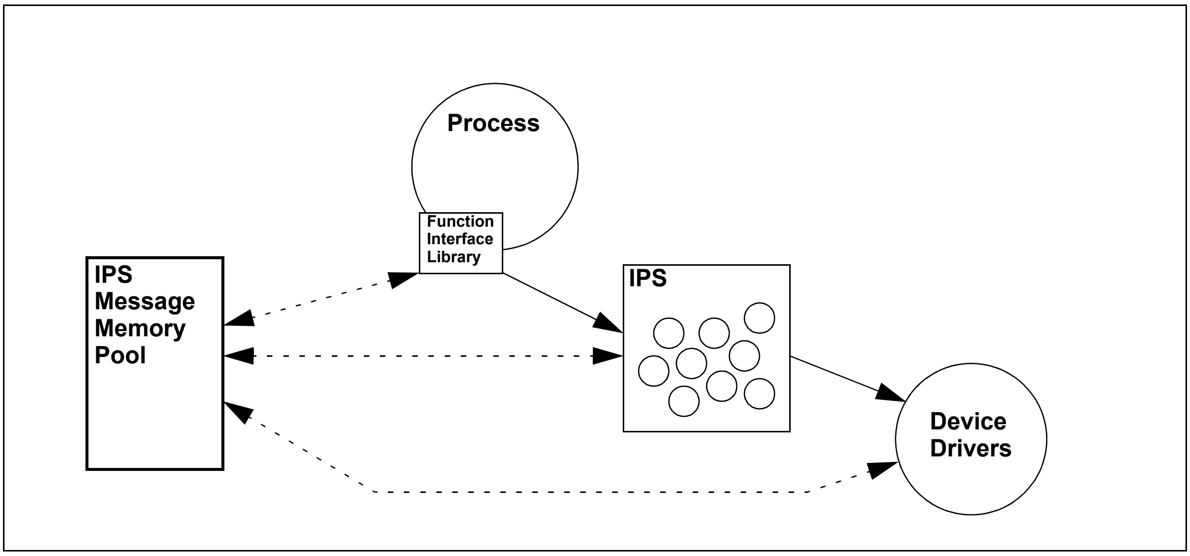


Figure 1-3: IPS Processes

For programming communication applications the programmer can use the well known standard socket function calls such as socket(), accept(), connect(), read().

In addition some specific interface methods are provided in SCIOPTA IPS which helps to increase the communication performance:

- A socket with the socket option **SO_SC_ASYNC** causes IPS to use the SCIOPTA message queue (message pool) for receiving data independent of how the data was sent. This method allows the user to concurrently work with ordinary SCIOPTA messages while waiting and handling received TCP or UDP data packages.
- SCIOPTA IPS includes an efficient flow-control.
- SCIOPTA IPS features a zero copy throughput.

1.7 IPS Application Programmers Interface

There are three different interfaces which can be used to access the SCIOPTA IPS functionality.

The SCIOPTA IPS is based on the SCIOPTA message passing technology. You can access the IPS functionality by exchanging messages. This results in a very efficient, fast and direct way of working with IPS. An application programmer can use the SCIOPTA message passing to send and receive network data for high speed asynchronous communication.

The IPS Function Interface is a function layer on top of the message interface. The message handling and event control are encapsulated in these functions.

Another convenient way is to use the BSD Socket Interface as it is a standardized API for most Internet Protocols applications. There are also IPS specific system calls included for supporting asynchronous mode.

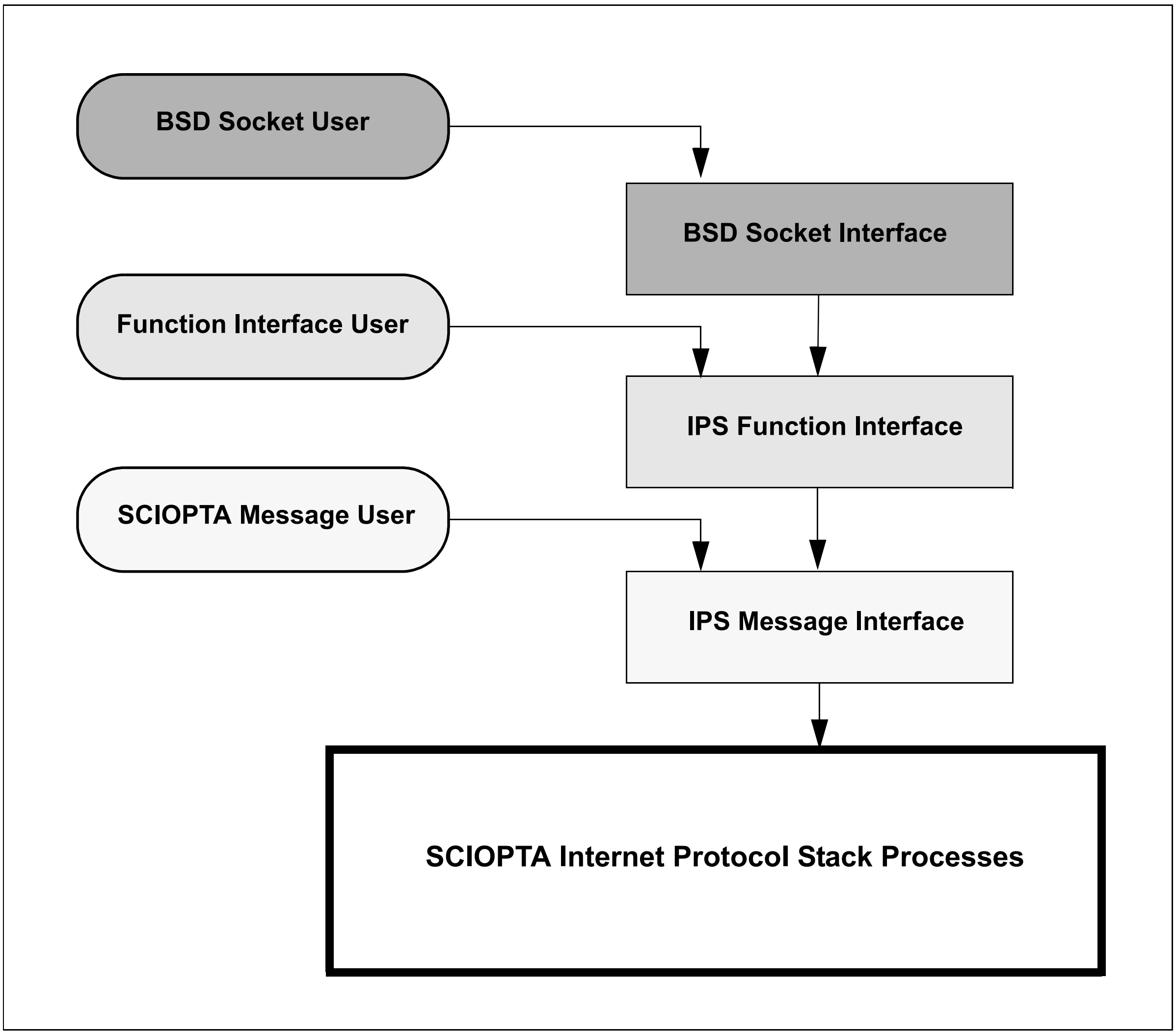


Figure 1-4: SCIOPTA IPS API

2 Installation

Please consult chapter 2 Installation of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description and guidelines of the SCIOPTA installation.

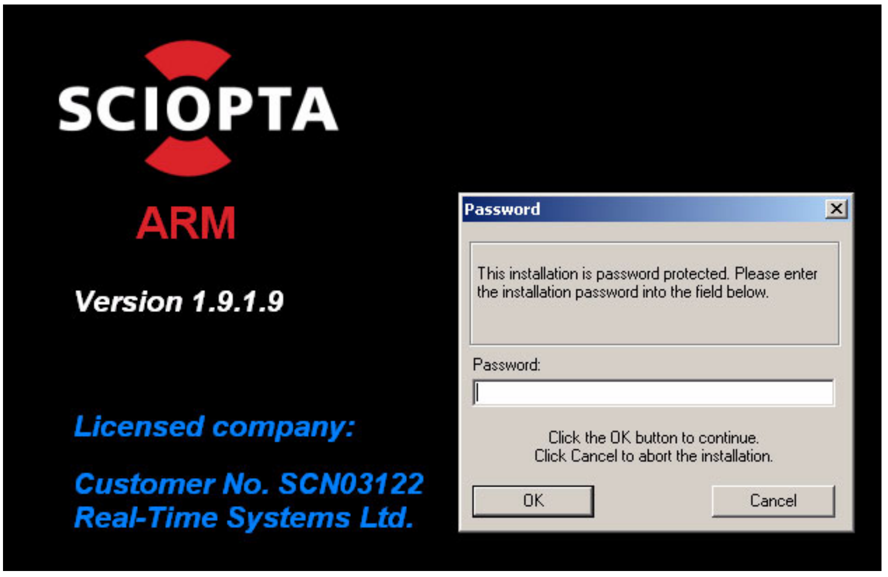


Figure 2-1: Main Installation Window

3 Getting Started

3.1 Introduction

These are small tutorial examples which gives you a good introduction into typical SCIOPTA systems and products.

They can be used as a starting point for more complex applications and your real projects.

Please note:

- The Getting-Started examples are using the Eclipse IDE, the MSys Make Utility, the GNU GCC Cross Compiler, the SCIOPTA BSPs (Board Support Packages) and the SCIOPTA examples. If you are using another board, CPU, compiler or IDE you might have to adapt the examples and you might need to change some or all the following files:

  Project SCIOPTA configuration file: hello.xml

  Project file: system.c

  Linker script: <board_name>.ld for GNU GCC

  Board assembler files such as: led.S, resethook.S

  BSP C files such as: druid_uart.c, eth.c, serial.c, simple_uart.c systick.c

  C Startup assembler file: cstartup.S

- Install **GCC version 3.4.4**, GNU Binutils version 2.16.1 and **MSys Build Shell version 1.0.10**. These products can be found on the SCIOPTA CD delivery.
- In the getting started examples we are using the **ECLIPSE** IDE. Install the Eclipse Platform including the **CDT** C/C++ Development Toolkit. These products can be downloaded from the **http://www.eclipse.org/** and the **http://www.eclipse.org/cdt/** web sites.
- We will use the MSys make utility for the Eclipse Platform. Therefore you need to add the MSys **bin** directory in your **PATH environment variable** (e.g. c:\msys\1.0\bin)
- Include the GNU GCC compiler **bin** directory in your **PATH environment variable** as described in chapter "GNU Tool Chain Installation" of the SCIOPTA ARM - Kernel, User's Guide.
- Check that the **environment variable $SCIOPTA_HOME** is defined as described in chapter "SCIOPTA_HOME Environment Variable" of the SCIOPTA ARM - Kernel, User's Guide.
- For every example we are copying all needed files into a local folder. This is not very useful in normal project development, but here it is done so, to show you what files are needed for the examples.
- You might need to setup your source-level emulator/debugger to initialize the memory. Please check the example linker script to fit to your board memory map.

3.2 Getting Started - IPS Internet Protocols Example

3.2.1 Description

The Getting Started System for the SCIOPTA IPS Internet Protocols consists of two isolated and separated subsystems. The two subsystems are not connected to each other and are included to show two different application levels.

The first system is a very simple target process which is just returning received data on a real network device. The target process is listening on TCP port 23 (telnet port). The user opens a telnet application on the host system and does a connect to the target. The target process first sends a welcome message and continues to send every received character back to the host. The target process terminates the telnet session if a single quote (".") character is received (only if telnet is not working in "line mode"). If telnet is working in "line mode" the session is terminated on a single quote character at the beginning of a line. This example is included to demonstrate a working real network connection between a host and the target system.

The second system consists of two processes (master and slave) which are connected over the SCIOPTA IPS TCP/IP stack. The two processes could be placed on two different target systems connected to a network. But to simplify the system both processes are placed on the same target system and connected to a loopback device. The slave process sends the data packages to localhost (127.0.0.1:2000). A loopback process handles such local IPS data packages by sending each IPS package back to the IPS stack.

The process master is bound on UDP <any_address>:port 2000 and waits for IPS data packets from the slave process. After the first packet is received, the master retrieves the slave's IP address and port number which is included in the received data packet. The master process connects to the slave and sends back the received data packet. The data package is now exchanged between process master and slave as long as the system is running.

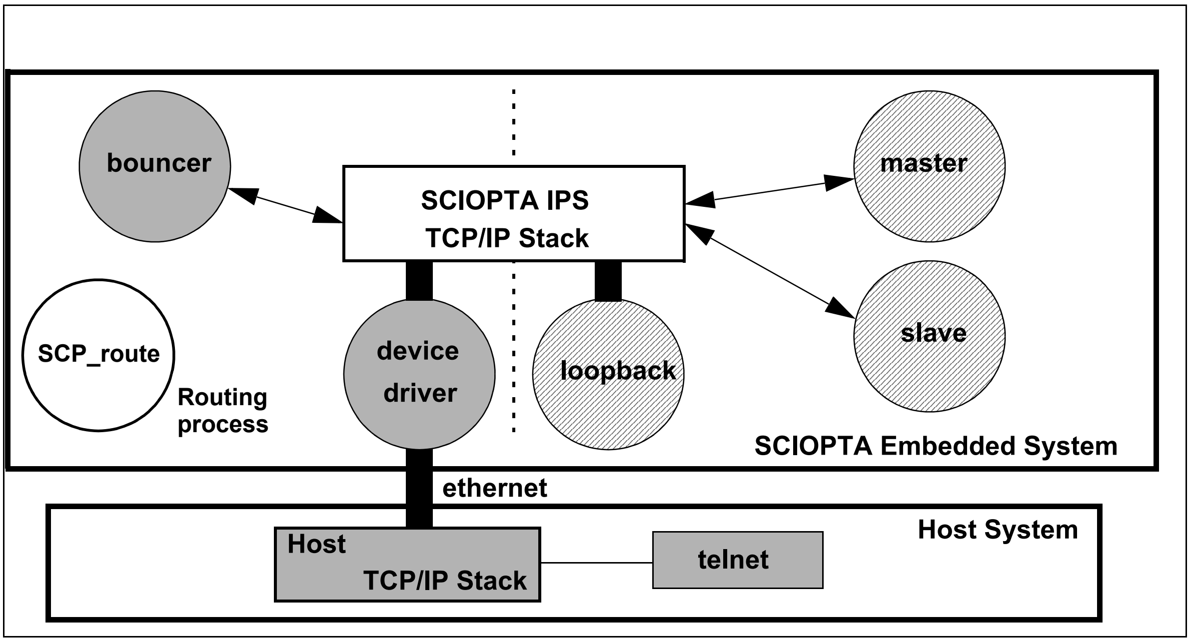


Figure 3-1: Getting Started IPS Example System

3.2.2 IPS Example - Windows Host

3.2.2.1 Equipment

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

- Personal Computer or Workstation with: Intel® Pentium® processor, Microsoft® Windows XP, 256 MB of RAM and 200 MB of available hard disk space.
- Target board which is supported by a SCIOPTA board support package (BSP). For each supported board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\ips\arm\hello\
- Network cable connected between your target board and your host workstation or to your network.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- GCC version 3.4.4, GNU Binutils version 2.16.1 and MSys Build Shell version 1.0.10. These products can be found on the SCIOPTA CD delivery.
- Eclipse Platform Version 3.2.1 including CDT C/C++ Development Toolkit version 3.1.1. These products can be downloaded from the **http://www.eclipse.org/** and the **http://www.eclipse.org/cdt/** web sites.
  In order to run the Eclipse Platform you also need the Sun Java 2 SDK, Standard Edition for Microsoft Windows.

3.2.2.2 Step-By-Step Tutorial

1. Create a project folder to hold all project files (e.g. c:\myproject\sciopta) if you have not already done it for other getting-started projects.
2. Launch Eclipse. The Workspace Launcher window opens.
3. Select your created project folder (e.g. c:\myproject\sciopta) as your workspace (by using the Browse button).
4. Click the OK button. The workbench windows opens. Close the Welcome Tab.
5. Maximize the workbench and deselect **"Build Automatically"** in the **Project** menu.
6. Open the **New Project** window (menu: **File -> New -> Project ...**).
7. Expand the **C** folder and select **Managed Make C Project**. Click the **Next** button.
8. Enter the project name (e.g. **ips_hello**) and click on the **Finish** button.
9. **Confirm Perspective Switch** by clicking on the **Yes** button. A new workbench opens and you can see the crated project in the Navigator window. Eclipse has crated a project folder (e.g. c:\myproject\sciopta\ips_hello).
10. Select the new created project in the browser window and close the project (menu: **Project -> Close Project**).
11. The next steps we will execute outside Eclipse.
12. Open a **Windows Explorer** or a **Command Prompt** window.
13. Copy the batch file **copy_files.bat** from the example directory of your board: <install_folder>\sciopta\<version>\exp\ips\arm\hello\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_hello).

14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_hello).
15. Double click on the **copy_files.bat** file to execute the batch file and the file copy process.
16. Launch the SCIOPTA configuration utility **sconf** from the desktop.
17. Load the SCIOPTA example project file **hello.xml** from your project folder into sconf.
    Menu: **File->Open**.
18. 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: **sciopta.cnf**, **sconf.c** and **sconf.h**.
19. Swap back to the Eclipse workbench. Make sure that the kernel hello project is highlighted (ips_hello) and open the project (menu: **Project -> Open Project**).
20. Expand the project by selecting the [+] button and make sure that the kernel hello project is highlighted (ips_hello).
21. Type the F5 key (or menu **File -> Refresh**). Now you can see all files in the Eclipse Navigator window.
22. Edit the file **route.c** by double-clicking this file. Edit and check the target IP settings (IP address, network mask and gateway to suit your network configuration:

```
#define ETH0_IP         "10.0.2.222"        // your_target_IP_address
#define ETH0_MASK       "255.255.255.0"     // your_target_network_mask
#define ETH0_GW         "10.0.2.2"          // your_target_gateway
```

23. Open the Console window at the bottom of the Eclipse workbench to see the project building output.
24. Be sure that the project (**ips_hello**) is high-lighted and build the project (menu **Project -> Build Project**). The executable (**sciopta.elf**) will be created in the debug folder of the project
    (e.g. c:\myproject\sciopta\ips_hello\debug).
25. Launch your source-level emulator/debugger.
26. For some specific emulators/debuggers you might found project and board initialization files in the original example directories.
27. Download the resulting **sciopta.elf** on your target board.
28. Check that your target board is connected to your network.
29. Start the target program.
30. Open a telnet session on your host system. From the Windows desktop select menu **Start -> Run...** and type **telnet**. The standard Telnet window opens.
31. Connect to the target by typing **open <your_target_IP_address>**
32. Watch now the typed characters returned from the target system.
33. You can also set breakpoints in the target ips application and watch the behaviour.

4 Configuration

4.1 Introduction

IPS Configuration is divided in two parts:

1. Configuring and defining all static objects (modules, pools and processes) with the help of the SCIOPTA configuration utility SCONF (sconf.exe). Please consult the SCIOPTA ARM - Kernel Manual for more information about using sconf.exe and the static configuration process.

   The static objects will be generated and started automatically at system start.

2. SCIOPTA IPS stack configuration which configure all parts in the IPS stack. This includes setting-up Link, ICMP, IPv4, UDP, TCP, PPP, Routing, Name Resolving etc. dependent on your protocol requirements.

4.2 Modules

For SCIOPTA systems using IPS you will normally put some basic processes (Driver Managers) in the System Module (each SCIOPTA application has a System Module sometimes also called module 0).

All other remaining IPS processes might reside in a specific IPS Module. But you could place all IPS processes in the System Module.

In our delivered getting started example we are using 4 modules. The systems module (“HelloSciopta”) contains the basic device and protocol managers, the “dev” module contains the device driver processes, in the “ips” modules reside the process for the TCP/IP stack and in the “user” module you can find the user’s application processes.

Please consult the SCIOPTA ARM - Kernel, User’s Guide for information about the module concept.

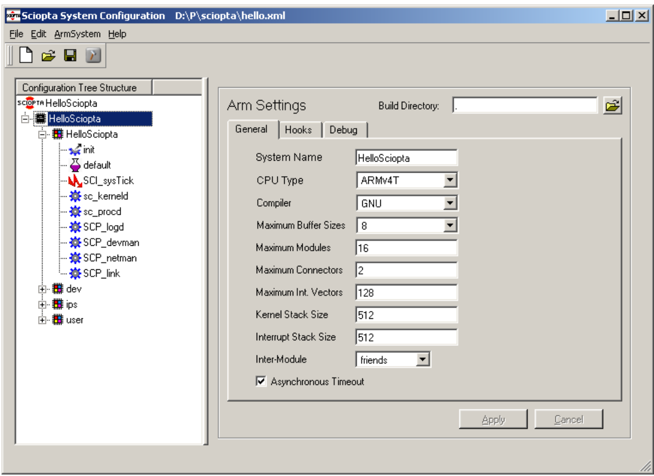


Figure 4-1: IPS System configuration with four modules

4.3 System Module

There is always one static system module in a SCIOPTA system. This module is called system module (sometimes also named module 0) and is the only static module in a system.

The following IPS processes must reside in the system module:

- SCP_devman
- SCP_link
- SCP_netman

All other IPS processes can either be placed in a separate IPS module or also reside in the system module.

4.3.1 System Module Configuration

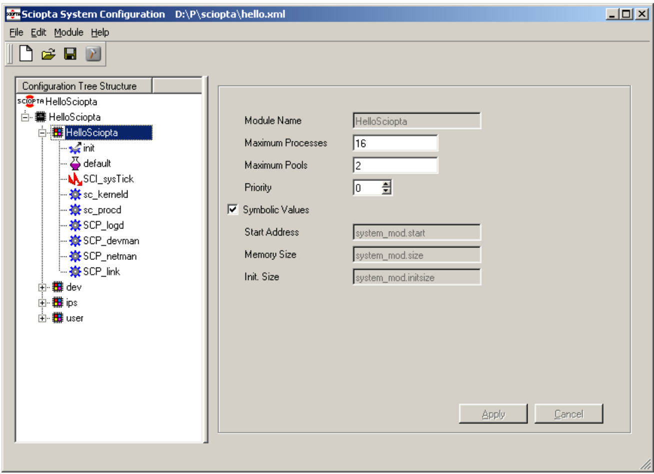


Figure 4-2: IPS system module “HelloSciopta”

Please consult the SCIOPTA ARM - Kernel, User’s Guide for information about system module parameters.

4.3.2 System Module init Process

The init process is the first process in a module. Each module has at least one process and this is the init process. At module start the init process gets automatically the highest priority (0). After the init process has done some important work it will change its priority to the lowest level (32) and enter an endless loop. Priority 32 is only allowed for the init process. All other processes are using priority 0 - 31. The init process acts therefore also as idle process which will run when all other processes of a module are in the waiting state.

The system module init process is automatically generated by the SCIOPTA SCONF tool. The process code can be found in the file sconf.c.

For the system module init process you only need to define the stack. A good starting point would be 512 bytes. You could optimize this stack by doing a stack analysis using the DRUID system level debugger.

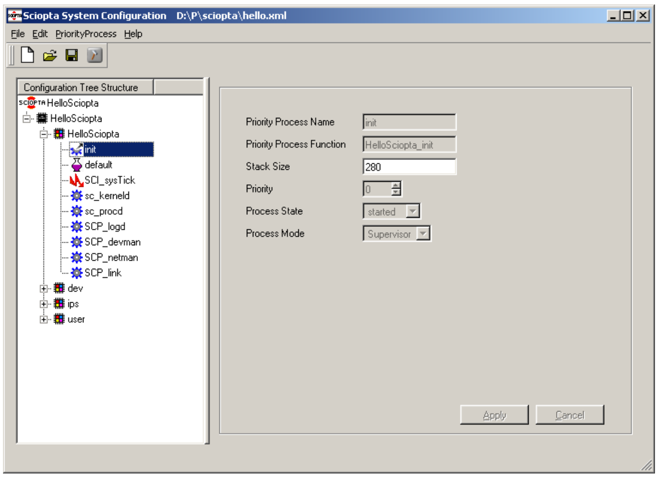


Figure 4-3: IPS system module init process

4.3.3 System Module Default Pool

In IPS process SCP_link allocates messages (network buffers) from a system module pool. SCP_link allocates the buffers from the first pool in the list of pools of the system module (**SC_DEFAULT_POOL**). In addition to the

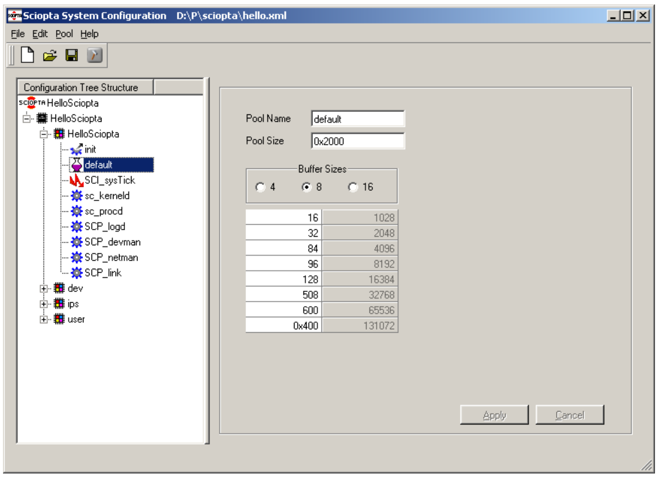


Figure 4-4: IPS system module default pool

SCP_link process, also device drivers are using this pool. The size depends on system design and timing issues.

4.3.4 System Tick Interrupt Process

The interrupt process for the system tick needs to be declared.

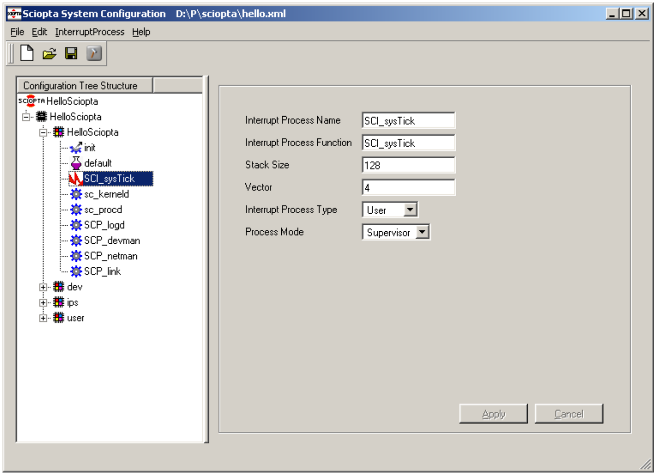


Figure 4-5: IPS system tick interrupt process

4.3.5 System Daemons

The SCIOPTA kernel daemon (sc_kerneld process) and process daemon (sc_procd process) are needed in dynamic system and are placed in the system module. The SCIOPTA log daemon (sc_logd process) is optional and logs the IPS events.

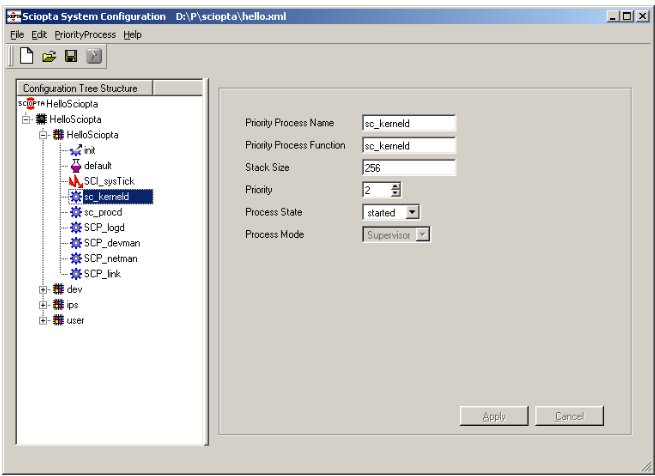


Figure 4-6: IPS system module daemons (kernel, process and log daemon)

4.3.6 Device Manager Process

SCP_devman is the device manager process for all devices excluding network devices. SCP_devman is a prioritized static process and must be located in the System Module.

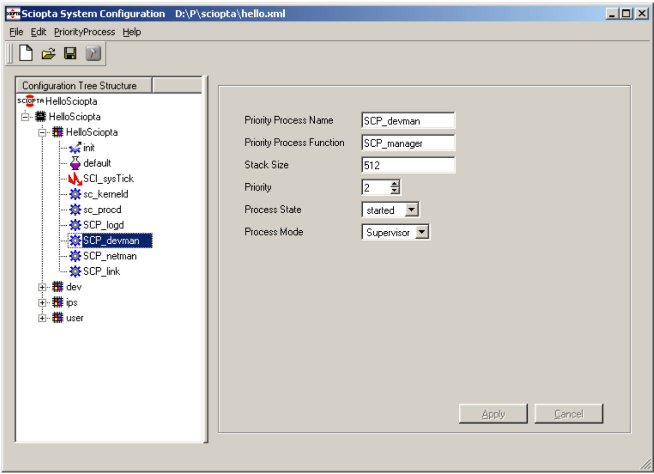


Figure 4-7: IPS system module device manager SCP_devman

4.3.7 Network Device Manager Process

SCP_netman is the device manager process for all network devices. SCP_netman is a prioritized static process and must be located in the System Module.

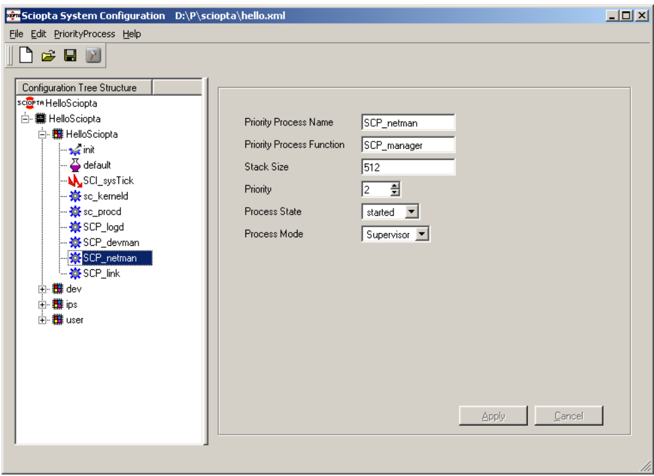


Figure 4-8: IPS system module network device manager SCP_netman

4.3.8 Link Process

SCP_link is a root manager process which links network device drivers to stacks (actually only IPv4 stack, but later also IPv6 stack). Please consult the SCIOPTA - Device Driver, User’s Guide for information about root managers. SCP_link is a prioritized static process and must be located in the System Module.

The user has to write the SCP_link process to define the LINK configuration parameters. Please consult chapter 4.7.1 “Link Configuration” on page 4-16 for more information how to write SCP_link.

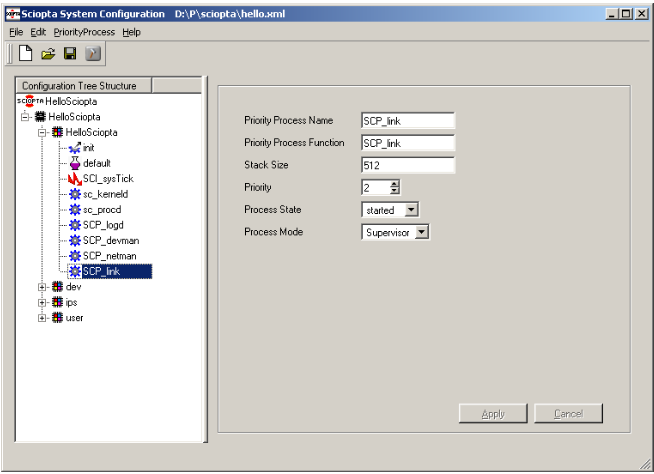


Figure 4-9: IPS system module link manager process

4.4 Device Driver Module

In a well structured IPS system and if there is enough target memory available it is good design practice to place the device drivers in a separate module. In the SCIOPTA IPS example we have done so and called this module “dev”.

4.4.1 Dev Module Configuration

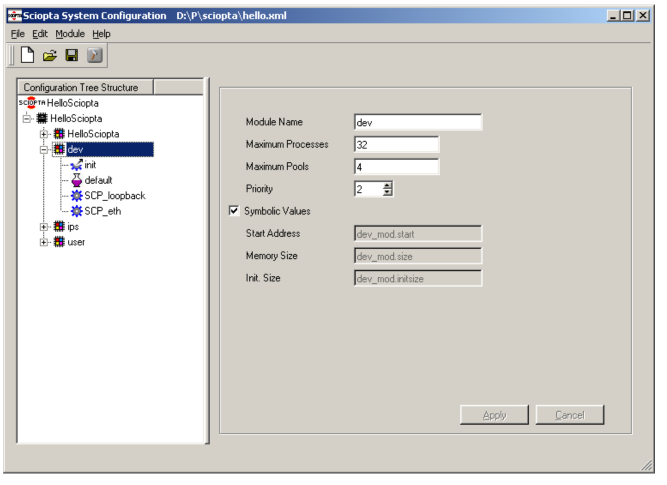


Figure 4-10: IPS device driver module “dev”

4.4.2 Device Driver Module Init Process

Similar settings as for the system module init process. Please consult chapter **4.3.2 “System Module init Process” on page 4-2**.

4.4.3 Device Driver Module Default Pool

The pool parameters must be designed to fit the requirements of the device driver message buffer sizes. The main device driver for IPS is the ethernet driver. Therefore define one buffer size to be slightly higher than a ethernet frame.

4.4.4 Ethernet Driver Process

For the ARM on-chip ethernet controller you need to declare a prioritized process (SCP_fec).

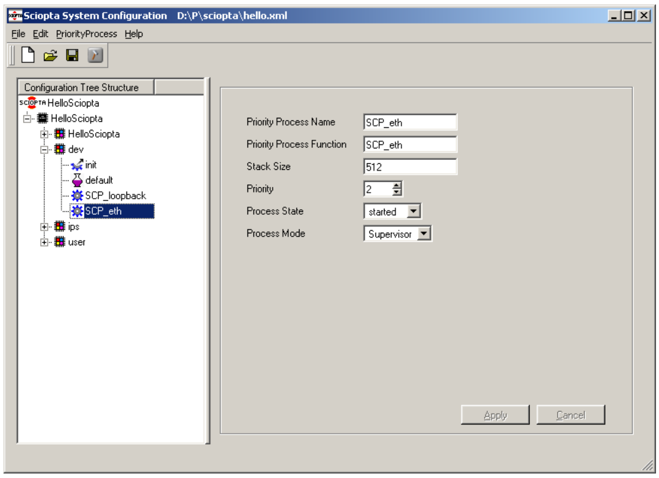


Figure 4-11: IPS ethernet device driver processes

4.4.5 Loopback process

The loopback process is for local host connections and is declared in the device driver module.

The user has to write the SCP_loopback process to define the loopback configuration parameters. Please consult chapter 4.7.6 "Loopback Configuration" on page 4-20 for more information how to write SCP_loopback.

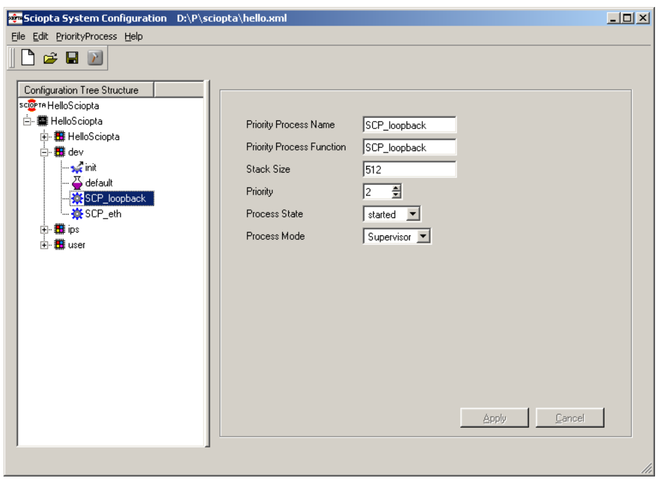


Figure 4-12: IPS loopback process

4.5 IPS Module

Usually the IPS TCP/IP stack is implemented in a separate module. To avoid message copy between the system module and the IPS module you have to declare the IPS module to be **friend** of the system module. Please consult the SCIOPTA - Kernel, User's Guide for more information about the module friend concept.

For systems with hardware MMU and the SCIOPTA SMMS product the IPS module can be protected from the other system. In such a system the messages to and from the IPS module will always be copied.

The IPS module includes the following processes:

- SCP_ipv4
- SCP_icmp (if ICMP is used)
- SCP_udp (if UDP is used)
- SCP_tcp (if TCP is used)

4.5.1 IPS Module Configuration

The SCIOPTA IPS TCP/IP stack resides in the "ips" module.

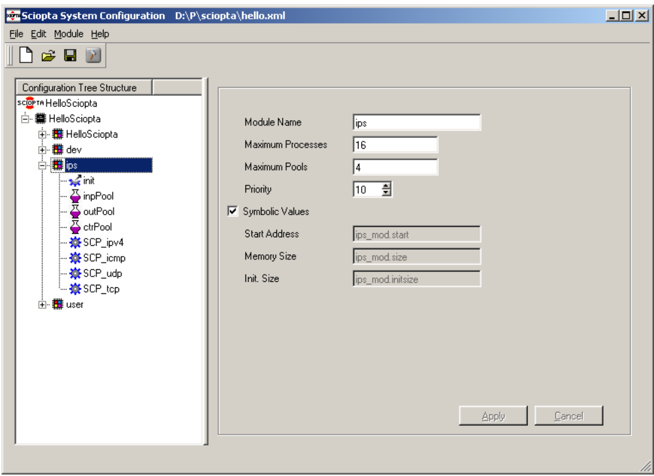


Figure 4-13: IPS stack module "ips"

Important: The order of declared IPS processes must be observed. Declare first **SCP_ipv4** followed by **SCP_icmp** and then **SCP_udp** and/or **SCP_tcp**.

4.5.2 IPS Module Init Process

Similar settings as for the system module init process. Please consult chapter **4.3.2 "System Module init Process" on page 4-2**.

4.5.3 IPS Module Input Message Pool

This is the message pool for incoming network buffers (packages). The size depends on the transmit window size, the protocol (TCP/UDP) and the number of connection. The size depends also on system design and timing issues. A good starting point is to use all remaining data memory of the module and divide it between ipsPool and ctrPool. Use the DRUID pool analyser to optimize the pool size.

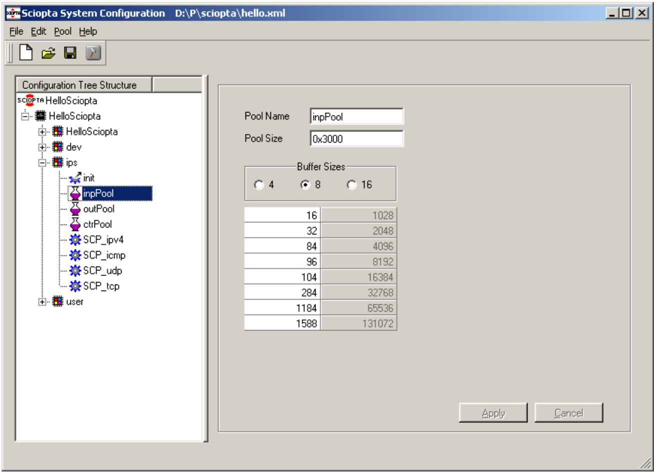


Figure 4-14: IPS module input message pool

4.5.4 IPS Module Output Pool

This is the message pool for outgoing network buffers (packages).

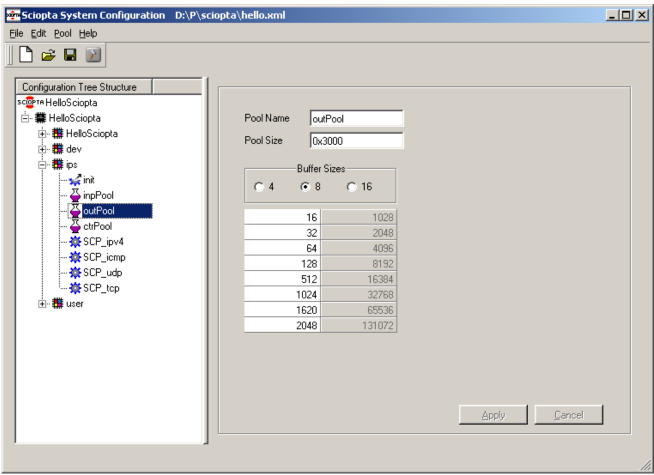


Figure 4-15: IPS module output message pool

4.5.5 IPS Module Control Pool

This is a specific message pool for all control messages in IPS.

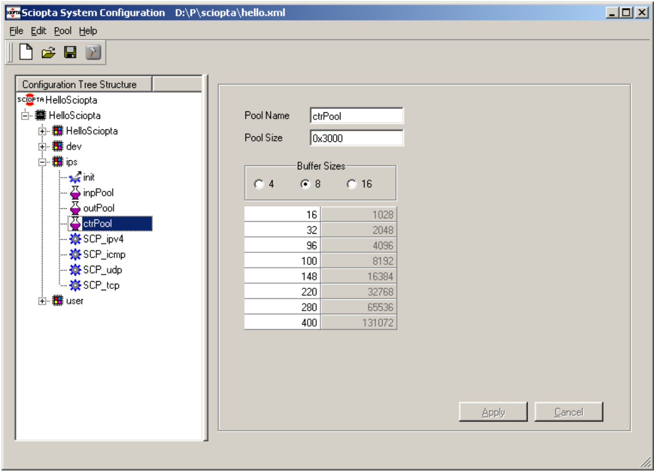


Figure 4-16: IPS module control pool

4.5.6 IPv4 Process

SCP_ipv4 does all the IP (version 4) protocol functions in IPS and provides the unreliable, connectionless datagram delivery service. The user has to write the SCP_ipv4 process to define the IPv4 configuration parameters. Please consult chapter **4.7.3 "IPv4 Configuration" on page 4-17** for more information how to write SCP_ipv4.

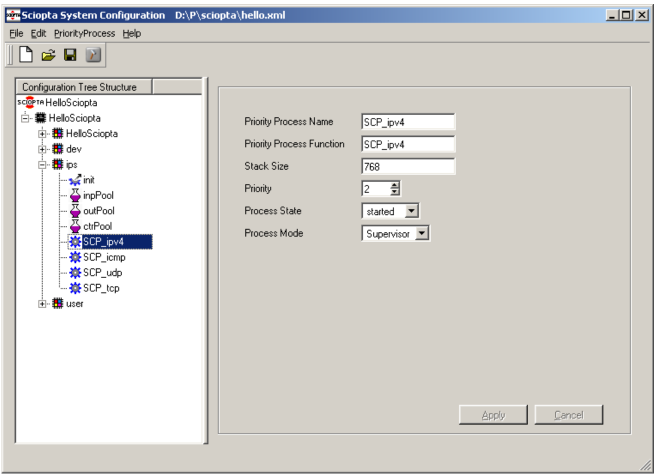


Figure 4-17: IPS module IPv4 process

4.5.7 SCP_icmp Process

SCP_icmp implements the Internet Control Message Protocol in IPS. It communicates error messages and other conditions that require attention. This process is optional but it is strongly recommended to include it. The user has to write the SCP_icmp process to define the ICMP configuration parameters. Please consult chapter **4.7.2 “ICMP Configuration” on page 4-17** for more information how to write SCP_icmp.

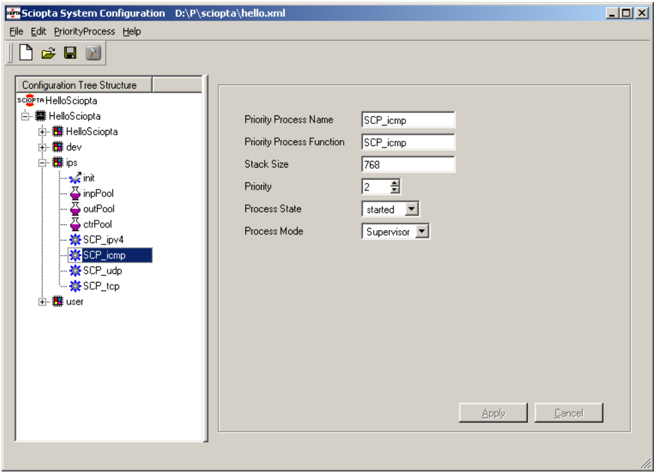


Figure 4-18: IPS module ICMP process

4.5.8 SCP_udp Process

SCP_udp implements the User Datagram Protocol in IPS. It needs only to be included if UDP is used. The user has to write the SCP_udp process to define the UDP configuration parameters. Please consult chapter 4.7.4 "UDP Configuration" on page 4-18 for more information how to write SCP_udp.

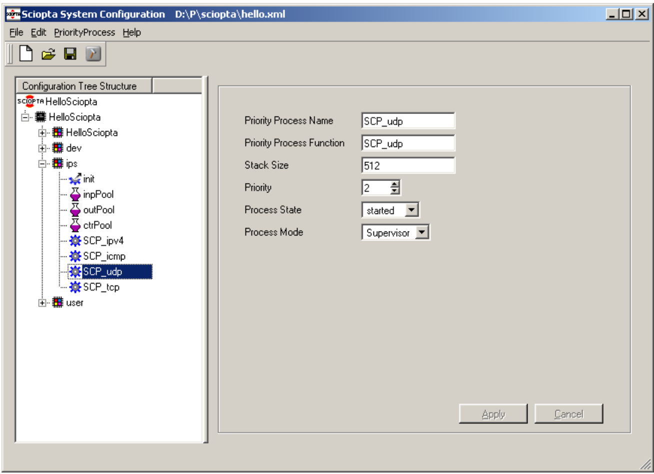


Figure 4-19: IPS module UDP process

4.5.9 SCP_tcp Process

SCP_tcp implements the Transmission Control Protocol in IPS. It needs only to be included if TCP is used. The user has to write the SCP_tcp process to define the TCP configuration parameters. Please consult chapter 4.7.5 "TCP Configuration" on page 4-19 for more information how to write SCP_tcp.

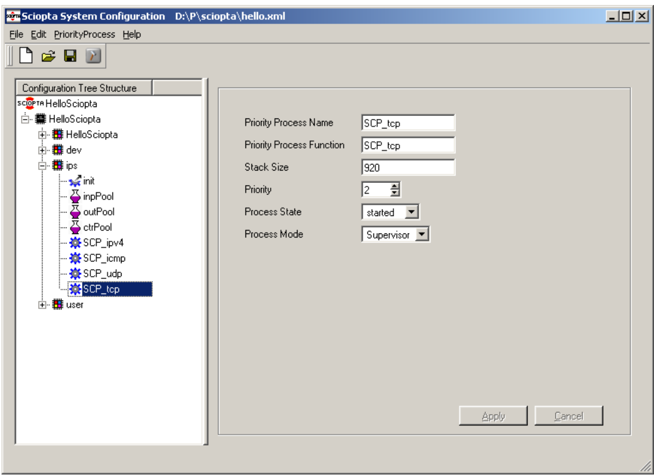


Figure 4-20: IPS module TCP process

4.6 User Module

For the SCIOPTA IPS examples we have placed the application processes in a separate module called “user”.

4.6.1 User Module Configuration

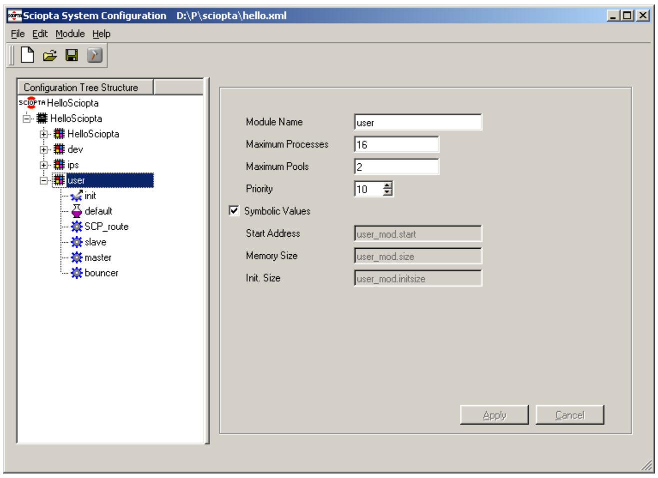


Figure 4-21: User module “user”

4.6.2 User Module Init Process

Similar settings as for the system module init process. Please consult chapter **4.3.2 “System Module init Process” on page 4-2**.

4.6.3 User Module Default Pool

The pool parameters must be designed to fit the requirements of the application message buffer sizes.

4.6.4 User Processes

Declare here all static user processes and choose the parameters to fit the requirements of your application.

4.6.5 Routing Process

A network device must have at least an IP address and a netmask to be able to send data on a TCP/IP network.

This can be done in a routing process (SCP_route) which we have placed in the “user” module in our standard IPS examples. The code of this routing process can be found in the route.c file.

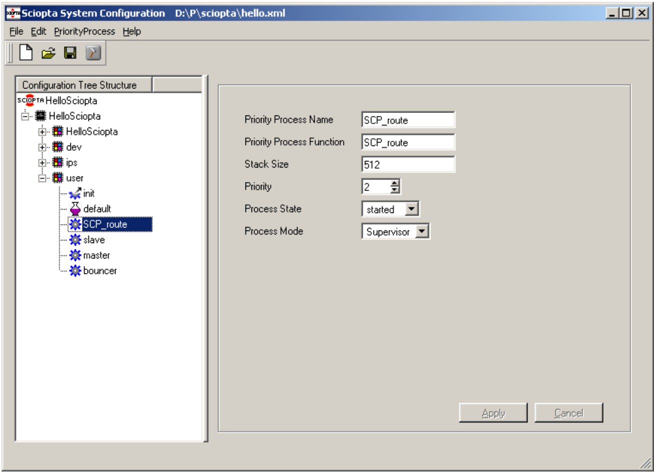


Figure 4-22: Routing process SCP_route

4.7 IPS Stack Configuration

In the previous chapters we have described how to configure the static modules, pools and processes for SCIOPTA IPS. This consisted mainly in declaring the static modules, pools and process and configuring the names, sizes and priorities in the SCIOPTA configuration utility SCONF.

To run the IPS TCP/IP stack you need also to configure some network parameters depending on the network specified properties. These parameters are defined by calling the SCIOPTA IPS stack processes at start-up:

- Link Configuration (SCP_link process)
- ICMP Configuration (SCP_icmp process)
- IPv4 Configuration (SCP_ipv4 process)
- UDP Configuration (SCP_udp process)
- TCP Configuration (SCP_tcp process)
- Loopback Configuration (SCP_loopback process)

In the SCIOPTA IPS example we have include the declaration of these processes in the file **ips.c** which can be found in the examples delivery. This file contains also the start-function of the “ips” module (void ips(void)). Please remember that the init process of each module calls a start-function with the same name as the module name.

File location: <install_folder>\sciopta\<version>\exp\ips\ppc\<example>\

4.7.1 Link Configuration

You need to declare a link initialization process SCP_link (see also chapter **4.3.8 “Link Process” on page 4-6**). The only function of this process is to call the IPS-internal LINK function **link_process** including the LINK configuration parameter.

Syntax link_process

```
void link_process (int maxProtocols);
```

Parameters

maxProtocols	Maximum number of allowed link protocols.

Example:

```
SC_PROCESS (SCP_link)
{
   link_process (2);
}
```

4.7.2 ICMP Configuration

You need to declare an ICMP initialization process SCP_icmp (see also chapter **4.5.7 "SCP_icmp Process" on page 4-12**). The only function of this process is to call the IPS-internal ICMP function **icmp_process** including the ICMP configuration parameter.

Syntax icmp_process

```
void icmp_process (int maxProtocols);
```

Parameters

maxProtocols	Maximum number of allowed ICMP protocols.

Example:

```
SC_PROCESS (SCP_icmp)
{
   icmp_process (1);
}
```

4.7.3 IPv4 Configuration

You need to declare an IPv4 initialization process SCP_ipv4 (see also chapter **4.5.6 "IPv4 Process" on page 4-11**). The only function of this process is to call the IPS-internal IPv4 function **ipv4_process** including the IPv4 configuration parameter.

Syntax ipv4_process

```
void ipv4_process (int maxProtocols);
```

Parameters

maxProtocols	Maximum number of allowed IPv4 protocols.

Example:

```
SC_PROCESS (SCP_ipv4)
{
   ipv4_process (3);
}
```

4.7.4 UDP Configuration

If you have UDP applications and you therefore want to have UDP supported by IPS you need to declare an UDP initialization process SCP_udp (see also chapter **4.5.8 "SCP_udp Process" on page 4-13**). The only function of this process is to call the IPS-internal UDP function **udp_process** including the two UDP configuration parameters.

Syntax udp_process

```
void udp_process (int maxConn, int maxPktQueue);
```

Parameters

maxConn	Maximum number of allowed UDP connections. -1 = no limitation.
maxPktQueue	Maximum number of allowed UDP packages inside the udp_process queue. If there is not any more space left in the queue all incoming packages are discarded.

Example:

```
SC_PROCESS (SCP_udp)
{
   udp_process (4, 4);
}
```

4.7.5 TCP Configuration

If you have TCP applications and you therefore want to have TCP supported by IPS you need to declare a TCP initialization process SCP_tcp (see also chapter 4.5.9 “SCP_tcp Process” on page 4-13). The only function of this process is to call the IPS-internal TCP function **tcp_process** including the six TCP configuration parameters.

Syntax tcp_process

```
void tcp_process (int rtoAlgorithm,
                  int rtoMin,
                  int rtoMax,
                  int maxConn,
                  int sndWnd,
                  int rcvWnd);
```

Parameters

rtoAlgorithm	TCP_MIB_TCP_RTO_ALGORITHM_VANJ Van Jacobson Timeout Algorithm.
rtoMin	Minimum timeout value (starting point of the Van Jacobson Timeout Algorithm) in milliseconds
rtoMax	Maximum timeout value in milliseconds.
maxConn	Maximum number of allowed TCP connections. -1 = no limitation.
sndWnd	Transmit window size.
rcvWnd	Maximum allowed receive window size.

Example:

```
SC_PROCESS (SCP_tcp)
{
   tcp_process (TCP_MIB_TCP_RTO_ALGORITHM_VANJ, 1500, 4800, 4, 1024, 1024);
}
```

4.7.6 Loopback Configuration

For local host connections you need to declare a loopback initialization process SCP_loopback (see also chapter 4.4.5 “Loopback process” on page 4-8). The only function of this process is to call the IPS-internal loopback function **lo_process** including the loopback configuration parameter.

Syntax lo_process

```
void lo_process ( const char   *name,
                  int          mtu,
                  int          mru,
                  sc_poolid_t  plid );
```

Parameters

name	Name of the loopback device (lo0, lo1 ...).
mtu	Max. transmitt unit.
mru	Max. receive unit.
plid	Pool ID for message allocation in lo_process.

Example:

```
SC_PROCESS (SCP_loopback)
{
  lo_process ("lo", 1500, 1500, SC_DEFAULT_POOL);
}
```

4.8 Routing Configuration

A network device must have at least an IP address and a netmask to be able to send data on a TCP/IP network. In the SCIOPTA IPS this is done with a message of type **ipv4_route_t** defined in the file router.msg which can be found in the IPS include directory.

The network device must be started before you can configure the router.

Example of the routing function is included in the file **route.c** of SCIOPTA IPS example delivery:

File location: <install_folder>\sciopta\<version>\exp\ips\common\

4.8.1 Routing Add Function

The best way to configure the routing is to use the following function:

```
Syntax          int ipv4_routeAdd (ipv4_route_t **route)
```

The parameter **route** is a pointer to a pointer to a message of type ips_ipv4Route_t. Please consult the SCIOPTA IPS Internet Protocols, Function Interface Manual for information about the ips_ipv4RouteAdd function.

Before calling this function you need to allocate the message and fill in the routing parameters.

4.8.2 Routing Configuration Example

In the following code fragment a network device for the subnet 10.0.1.0 will be configured. We are supposing that the address 10.0.1.45 is free in the subnet and the name of the device is eth0.

Includes

```
#include <ips/router.h>
#include <ips/router.msg>
#include <ips/device.h>
```

Message

```
union sc_msg {
  sc_msgid_t id;
  ipv4_route_t route;
}

union sc_msg routemsg;
```

Configuration code fragment to be included inside the configuration file:

```
  ips_dev_t *dev;
  routemsg = sc_msgAlloc (sizeof (ipv4_route_t),
                               0,
                               SC_DEFAULT_POOL,
                               SC_FATAL_IF_TMO);
```

```
dev = ips_devGetByName ("eth0");
memcpy (routemsg, dev, sizeof (ips_dev_t));
sc_msgFree ((union sc_msg **) &dev);

/* my IP */
routemsg->route.source[0] = 10;
routemsg->route.source[1] = 0;
routemsg->route.source[2] = 1;
routemsg->route.source[2] = 45;

/* my destination subnet */
routemsg->route.destination[0] = 10;
routemsg->route.destination[1] = 0;
routemsg->route.destination[2] = 1;
routemsg->route.destination[3] = 0;

/* my router */
/* set to zero (no router) */
/* as subnet mounted directly to the target in this example */
routemsg->route.router[0] = 0;
routemsg->route.router[1] = 0;
routemsg->route.router[2] = 0;
routemsg->route.router[3] = 0;

/* my netmask */
routemsg->route.netmask [0] = 255;
routemsg->route.netmask [1] = 255;
routemsg->route.netmask [2] = 255;
routemsg->route.netmask [3] = 0;

/* metric (default = 0) */
routemsg->route.metric = 0;

/* if mtu and mru need to be configures */
routemsg->route.dev.mru = 512;
routemsg->route.dev.mtu = 512;
/* please be sure that the device driver can handle the given mru and mtu values */

ips_ipv4RouteAdd (&routemsg);
```

4.8.3 Routing Remove Example

To remove a route you need to specify it explicitly. The best way to remove the route is to use the following function:

```
Syntax                int ipv4_routeRm (ipv4_route_t *route)
```

The parameter **route** is a pointer to a message of type ipv4_route_t.

If you want to remove for example the route of the preceding config process you need to specify the source IP address, the netmask and the metric.

```
SCP_PROCESS (shutdown)
{
  ipv4_route_t route;

  /* Specify the route as follows */
  /* my IP */
  route.source[0] = 10;
  route.source[1] = 0;
  route.source[2] = 1;
  route.source[2] = 45;
  /* my netmask */
  route.netmask [0] = 255;
  route.netmask [1] = 255;
  route.netmask [2] = 255;
  route.netmask [3] = 0;
  /* my metric */
  route.metric = 0;

  /* Removes the defined route */
  ipv4_routeRm (&route);

  sc_procKill (SC_CURRENT_PID, 0);

}
```

4.9 Name Resolving

To be able to resolve IP addresses to host names (and vice versa) the following actions need to be implemented.

1. Declare the process SCP_resolver as a static process.
2. You need to declare the name and path to the resolver process in the string variable resolver (to be declared in the ips configuration file **ips.c**):

   ```
   char *resolver = {"path"};
   ```

   path is declared as /<module_name>/<process_name>

   ```
   example: char *resolver = {"/ips/SCP_resolver"};
   ```

3. Define the name servers. You can declare as many as you want, but it does not make sense to declare more than four.

   The variable res_noofNameServers contains the number of declared name servers and the array res_nameServers contains the name server entries. Both variables might be declared in the file **ipsconfig.c**.

   ```
   example:

   include <ips\addr.h>

   int res_noofNameServers = 3;

   ips_addr_t res_nameServers[] = {
      { 4, {10,0,1,11}},
      { 4, {147,86,..,..}},
      { 4, {..,..,..,..}}
   };
   ```

   The number 4 in front of the 3 addresses declares name resolving for IPv4.

5 System Design

5.1 Introduction

In this chapter you will get information how to design embedded systems using the SCIOPTA IPS Internet Protocols. After a description of IPS system configuration useful hints and information about specific design issues are given.

5.2 Client-Server Connection Setup

5.2.1 UDP Client-Server Setup

A connection in the true sense of the word does not exist in UDP. UDP is a connectionless communication. Nevertheless the BSD functions bind and connect are available in IPS and must actually be used for the asynchronous connection as the sendto() and the recvfrom() calls are not yet implemented.

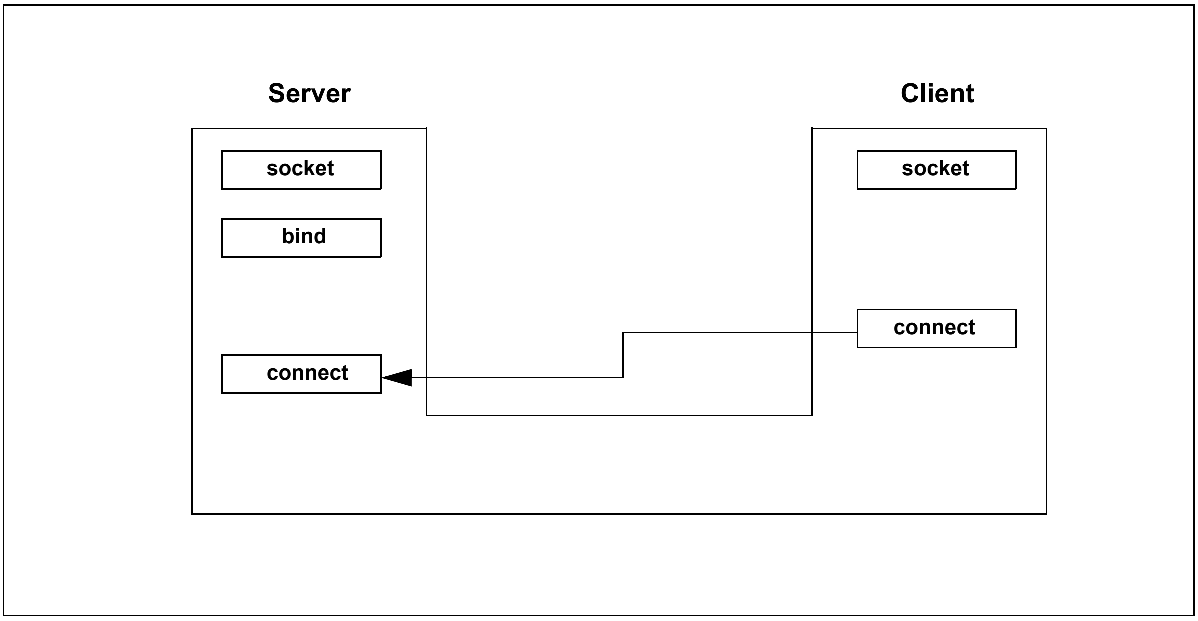


Figure 5-1: UDP Connection Building

5.2.2 TCP Client-Server Setup

In contrast to UDP, TCP is a connection oriented communication and requires a fully qualified connection set-up.

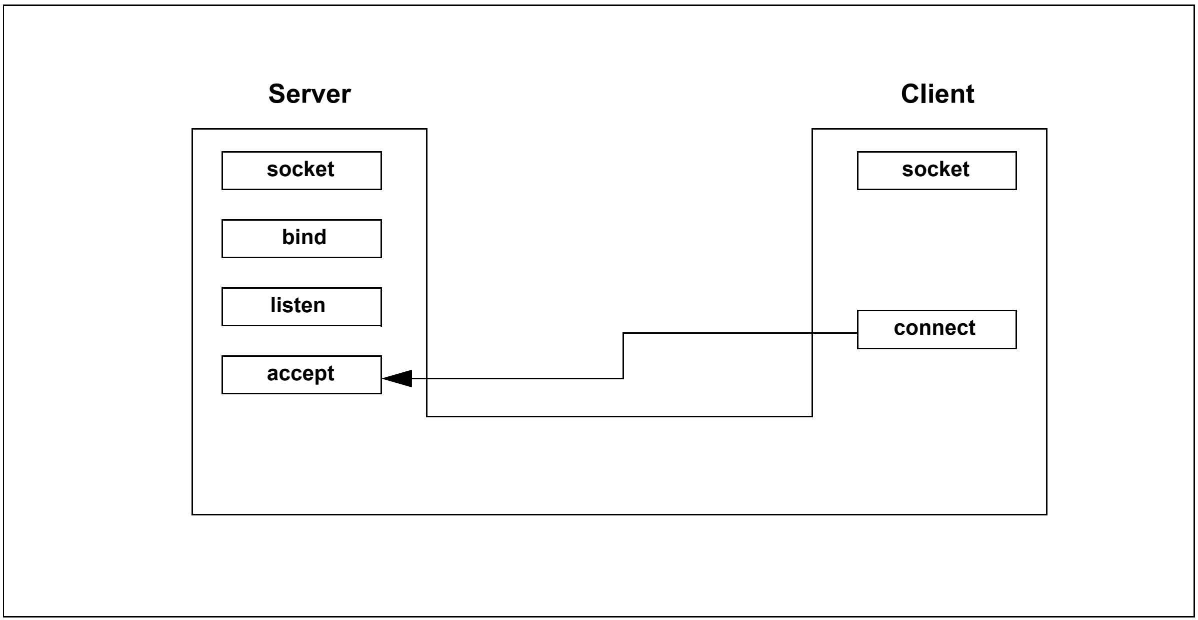


Figure 5-2: TCP Connection Building

5.2.3 Asynchronous versus Synchronous Connection

Traditional TCP/IP stacks using socket programming are based on synchronous communication. A read() is blocking the function as long as the requested number of bytes are not received in its buffer. In this case the data of UDP or TCP package are always copied. Some stacks (UNIX) allow to reach an asynchronous mode by using the O_NONBLOCK option and signals. But this will not avoid data copying.

The SCIOPTA IPS TCP/IP stack allows an asynchronous communication by using the standard SCIOPTA message queue. The data packages are directly maintained in the SCIOPTA message queue and accessed by the application program. A copy of the data is avoided as the received buffer is owned by the user. Nevertheless, a copy will of course occur if the data crosses module borders. By using the SCIOPTA friend concept you can also prevent this copy. Please consult the SCIOPTA Kernel, User's Guide for more information about the module and friend concept.

The SCIOPTA IPS asynchronous communication allows to receive UDP as well as TCP packages in the same message queue which might be useful for some applications. As each buffer contains the handle and the process id of the sender the user can find out if it is an UDP or TCP package. To do this you need to get the sdd_obj_t from the socket descriptor. Please consult chapter **5.9 "BSD Descriptor and SDD Descriptor" on page 5-33** for more information.

5.3 SDD Objects

SDD objects are specific system objects in a SCIOPTA real-time operating system such as:

SDD devices and SDD device drivers	Objects and methods controlling I/O devices
SDD managers	Objects and methods managing other SDD objects. SDD managers are maintaining SDD object databases. There are for instance **SDD device managers** which managing **SDD devices** and **SDD device drivers** and **SDD file managers** which are managing **files** in the SCIOPTA SFS file system.
SDD protocols	Objects and methods representing network protocols such as SCIOPTA IPS TCP/IP internet protocols.
SDD directories and files	The file object in the SCIOPTA SFS file system.

5.4 SDD Descriptors

5.4.1 Introduction

SDD Descriptors are data structures in SCIOPTA containing information about specific drivers or objects. Drivers are not only programs managing and controlling devices (device drivers) they can also represent objects which are managing protocols, file system files or other system resources.

SDD descriptors are stored as standard SCIOPTA messages inside message pools. That is why SDD descriptors contain a message ID structure element.

Please consult also the SCIOPTA - BSP and Device Driver, User’s Guide and Reference Manual.

5.4.2 General SDD Descriptor Definition

An **SDD Object Descriptor** contains information about a **SCIOPTA Object**.

5.4.3 Specific SDD Descriptors

- **SDD device descriptors** contain information about **SDD devices**.
- **SDD device manager descriptors** contain information about **SDD device managers**.
- **SDD network device descriptors** contain information about **SDD network devices**.
- **SDD protocol descriptors** contain information about **SDD protocols**.
- **SDD file manager descriptors** contain information about **SDD file managers**.
- **SDD file device descriptors** contain information about **SDD file devices**.
- **SDD directory descriptors** contain information about **SDD directories**.
- **SDD file descriptors** contain information about **SDD files**.

- Please consult chapter for more information about SDD object descriptor structures.

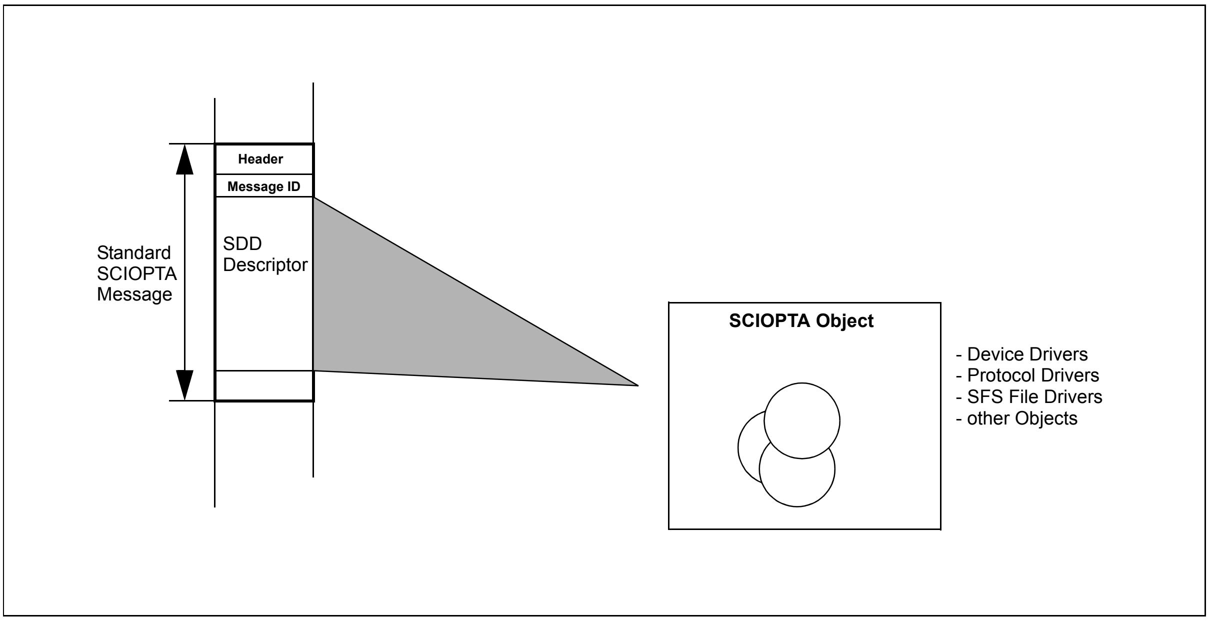


Figure 5-3: SDD Descriptor

5.5 IPS Application Programmers Interface

There are three different interfaces which can be used to access the SCIOPTA IPS functionality.

The SCIOPTA IPS is based on the SCIOPTA message passing technology. You can access the IPS functionality by exchanging messages. This results in a very efficient, fast and direct way of working with IPS. An application programmer can use the SCIOPTA message passing to send and receive network data for high speed asynchronous communication. Please consult chapter **5.6 "Using the IPS Message Interface" on page 5-7** for more information.

The IPS Function Interface is a function layer on top of the message interface. The message handling and event control are encapsulated in these functions. Please consult chapter **5.7 "Using the IPS Function Interface" on page 5-22** for more information.

Another convenient way is to use the BSD Socket Interface as it is a standardized API for most Internet Protocols applications. There are also IPS specific system calls included for supporting asynchronous mode. Please consult chapter **5.8 "Using The IPS BSD Socket Interface" on page 5-27** for more information.

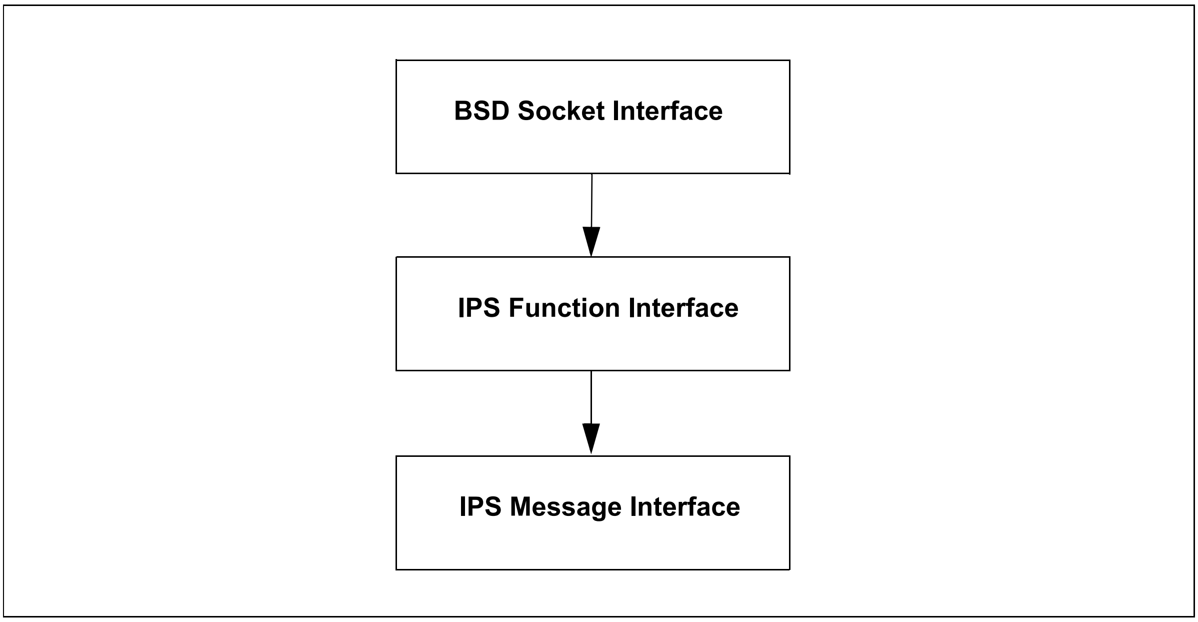


Figure 5-4: SCIOPTA IPS API

5.6 Using the IPS Message Interface

5.6.1 Introduction

We will just give some simple examples how to build up UDP and TCP connection and how to send and receive data by using the IPS Message Interface. Please consult the SCIOPTA - Kernel, User's Guide and Reference Manual for information how to define and use SCIOPTA messages.

Before you can send and receive network data with UDP and TCP you need to setup the routes (see chapter **4.8 "Routing Configuration" on page 4-21**).

5.6.2 UDP Sending Using the IPS Message Interface

1. Getting the IPv4 SDD protocol descriptor

   To be able to access the IPv4 protocol driver we need to get the IPv4 SDD protocol descriptor from the IPS SCP_link process. The SCP_link process has automatically registered the IPv4 SDD protocol descriptor and can be accessed like manager process with a SDD_MAN_GET message.

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_obj_t          netIPV4;
};

sc_msg_t     ipv4msg;
```

   Allocate an SDD_MAN_GET message of type sdd_obj_t.

```
ipv4msg = sc_msgAlloc (sizeof (sdd_obj_t),
                         SDD_MAN_GET,
                         SC_DEFAULT_POOL,
                         SC_FATAL_IF_TMO);
```

   Fill the message body.

ipv4msg->netIPV4.base.id	SDD_MAN_GET (Filled by sc_msgAlloc())
ipv4msg->netIPV4.base.error	0
ipv4msg->netIPV4.base.handle	0
ipv4msg->netIPV4.manager	0 (SCP_link is a Root Manager)
ipv4msg->netIPV4.type	0
ipv4msg->netIPV4.name	"ipv4" (We need the IPv4 SDD protocol descriptor)
ipv4msg->netIPV4.controller	0
ipv4msg->netIPV4.sender	0
ipv4msg->netIPV4.receiver	0

   Send the message to the IPS manager process /**SCP_link**. If SCP_link is a static process (which is usually the case) you can address it by just append **_pid** to the process name.

```
sc_msgTx (&ipv4msg, SCP_link_pid, 0);
```

   Receive the SDD_MAN_GET_REPLY message from SCP_link.

```
static const sc_msgid_t select[2] = { SDD_MAN_GET_REPLY, 0 };

ipv4msg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The SDD_MAN_GET_REPLY message is sent by SCP_link and received. The received message is the IPv4 SDD protocol descriptor and contains all information how to access the IPv4 protocol driver.

ipv4msg->netIPV4.base.id	SDD_MAN_GET_REPLY
ipv4msg->netIPV4.base.error	Possible error returned by SCP_link
ipv4msg->netIPV4.base.handle	Manager Handle of the IPv4 protocol driver
ipv4msg->netIPV4.manager	0 (not modified)
ipv4msg->netIPV4.type	0 (not modified)
ipv4msg->netIPV4.name	“ipv4” (not modified)
ipv4msg->netIPV4.controller	controller process ID of the IPv4 protocol driver
ipv4msg->netIPV4.sender	sender process ID of the IPv4 protocol driver
ipv4msg->netIPV4.receiver	receiver process ID of receiver of IPv4 protocol driver

2. Getting the UDP SDD protocol descriptor

To be able to access the UDP protocol driver we need to get the UDP SDD protocol descriptor from the IPv4 protocol driver. The IPv4 protocol driver has automatically registered the SDD protocol descriptors and can be accessed like manager process with a SDD_MAN_GET message.

Message definition:

```
union sc_msg {
   sc_msgid_t          id;
   sdd_obj_t           netUDP;
};

sc_msg_t      udpmsg;
```

Allocate an SDD_MAN_GET message of type sdd_obj_t.

```
udpmsg = sc_msgAlloc (sizeof (sdd_obj_t),
                      SDD_MAN_GET,
                      SC_DEFAULT_POOL,
                      SC_FATAL_IF_TMO);
```

Fill the message body.

udpmsg->netUDP.base.id	SDD_MAN_GET
udpmsg->netUDP.base.error	0
udpmsg->netUDP.base.handle	0
udpmsg->netUDP.manager	Manager Handle of the IPv4 protocol driver (copied from ipv4msg->netIPv4.base.handle)
udpmsg->netUDP.type	0
udpmsg->netUDP.name	“udp” (We need the UDP SDD protocol descriptor)
udpmsg->netUDP.controller	0
udpmsg->netUDP.sender	0
udpmsg->netUDP.receiver	0

Sent this message to the controller process of the IPv4 protocol driver.

```
sc_msgTx (&udpmsg, ipv4msg->netIPV4.controller, 0);
```

Receive the SDD_MAN_GET_REPLY message from the IPv4 protocol driver.

```
static const sc_msgid_t select[2] = { SDD_MAN_GET_REPLY, 0 };

udpmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The SDD_MAN_GET_REPLY message is sent by the IPv4 protocol driver and received. The received message is the UDP SDD protocol descriptor and contains all information how to access the UDP protocol driver.

udpmsg->netUDP.base.id	SDD_MAN_GET_REPLY
udpmsg->netUDP.base.error	Possible error returned by the IPv4 protocol driver
udpmsg->netUDP.base.handle	Handle of the UDP protocol driver
udpmsg->netUDP.manager	not modified
udpmsg->netUDP.type	0 (not modified)
udpmsg->netUDP.name	“udp” (not modified)
udpmsg->netUDP.controller	controller process ID of the UDP protocol driver
udpmsg->netUDP.sender	sender process ID of the UDP protocol driver
udpmsg->netUDP.receiver	receiver process ID of the UDP protocol driver

3. **Opening** the UDP protocol driver

To be able to connect and to communicate with the SDD UDP protocol driver we need to open it. This will return the UDP SDD protocol descriptor which contains the access handle.

Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_netOpen_t      netOpen;
};

sc_msg_t      openmsg;
```

Allocate a SDD_NET_OPEN message of type sdd_netOpen_t. This message is documented in the SCIOPTA - BSP and Device Driver manual.

```
openmsg = sc_msgAlloc (sizeof (sdd_netOpen_t),
                  SDD_NET_OPEN,
                  SC_DEFAULT_POOL,
                  SC_FATAL_IF_TMO);
```

Fill the message body.

openmsg->netOpen.base.id	SDD_NET_OPEN (Filled by sc_msgAlloc())
openmsg->netOpen.base.error	0
openmsg->netOpen.base.handle	handle of the UDP protocol driver (copied fromudpmsg->netUDP.base.handle).
openmsg->netOpen.context	0

Send this message to the controller process of the UDP protocol driver.

```
sc_msgTx (&openmsg, udpmsg->netUDP.controller, 0);
```

Receive the SDD_NET_OPEN_REPLY message from the UDP protocol driver.

```
static const sc_msgid_t select[2] = { SDD_NET_OPEN_REPLY, 0 };

openmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The SDD_NET_OPEN_REPLY message is sent by UDP protocol driver and received. The received message contains the access handle of the UDP protocol driver.

openmsg->netOpen.base.id	SDD_NET_OPEN_REPLY
openmsg->netOpen.base.error	Possible error returned by the UDP protocol driver
openmsg->netOpen.base.handle	Access handle of the UDP protocol driver
openmsg->netOpen.flags	not modified

We have now all information to access the UDP protocol driver.

1. UDP controller process of the UDP protocol driver: **`udpmsg->netUDP.controller`**
2. Access handle: **`openmsg->netOpen.base.handle`**

3. **Binding** (server side)

   Binding is used to define a specific IP address and port number where UDP receives network packages. We will send an IPS_BIND message to the UDP protocol driver.

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   ips_bind_t         netBind;
};

sc_msg_t      bindmsg;
```

   Allocate an IPS_BIND message of type ips_bind_t.

```
bindmsg = sc_msgAlloc (sizeof (ips_bind_t),
                       IPS_BIND,
                       SC_DEFAULT_POOL,
                       SC_FATAL_IF_TMO);
```

   Fill the message body.

bindmsg->netBind.base.id	IPS_BIND (Filled by sc_msgAlloc())
bindmsg->netBind.base.error	0
bindmsg->netBind.base.handle	Access handle: openmsg->netOpen.base.handle
bindmsg->netBind.srcPort	Source port, netbyte order htons.
bindmsg->netBind.srcAddr.len	4 (for IPv4 adress).
bindmsg->netBind.srcAddr.addr[]	Source address.

   Send the message to the controller process of the UDP protocol driver.

```
sc_msgTx (&bindmsg, udpmsg->netUDP.controller, 0);
```

   Receive the IPS_BIND_REPLY message from the UDP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_BIND_REPLY, 0 };

bindmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

   Check bindmsg->netBind.base.error for a returned error condition.

4. **Connecting** (client and server side)

   Connecting is used to set the default destination peer. We will send an IPS_CONNECT message to the UDP protocol driver.

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   ips_connect_t      netConnect;
};

sc_msg_t      connectmsg;
```

Allocate an IPS_CONNECT message of type ips_connect_t.

```
connectmsg = sc_msgAlloc (sizeof (ips_connect_t),
                          IPS_BIND,
                          SC_DEFAULT_POOL,
                          SC_FATAL_IF_TMO);
```

Fill the message body.

connectmsg->netConnect.base.id	IPS_CONNECT (Filled by sc_msgAlloc())
connectmsg->netConnect.base.error	0
connectmsg->netConnect.base.handle	Access handle: openmsg->netOpen.base.handle
connectmsg->netConnect.dstPort	Destination port, netbyte order htons.
connectmsg->netConnect.dstAddr.len	4 (for IPv4 adress).
connectmsg->netConnect.dstAddr.addr[]	Destination address.

Send the message to the controller process of the UDP protocol driver.

```
sc_msgTx (&connectmsg, udpmsg->netUDP.controller, 0);
```

Receive the IPS_CONNECT_REPLY message from udp or tcp.

```
static const sc_msgid_t select[2] = { IPS_CONNECT_REPLY, 0 };

connectmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

Check connectmsg->netConnect.base.error for a returned error condition.

5. Setting-Up the **Network Buffer**

Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_netbuf_t       netBuf;
};

sc_msg_t      netmsg;
```

Use the ips_alloc() function to allocate a network buffer.

```
netmsg = (sc_msg_t) ips_alloc (netbufsize,
                               SC_DEFAULT_POOL,
                               SC_FATAL_IF_TMO)
```

Set the message id :

```
netmsg->netBuf.base.id = SDD_NET_SEND;
```

Set the Access handle:

```
netmsg->netBuf.base.handle = openmsg->netOpen.base.handle;
```

Fill the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

6. **Sending** the Network Buffer

Send the message to the sender process of the UDP protocol driver

```
sc_msgTx (&netmsg, udpmsg->netUDP.sender, 0);
```

5.6.3 UDP Receiving Using the IPS Message Interface

The device driver will send a SDD_NET_RECEIVE message to the SCP_link process of the IPS stack if the network data came in. The user process waits on a SDD_NET_RECEIVE message coming from the IPS stack.

Before you can receive UDP data you should get the UDP SDD protocol descriptor, open the UDP protocol driver and connect or bind as described in chapter 5.6.2 "UDP Sending Using the IPS Message Interface" on page 5-7.

1. **Receive Mode** Setting

   IPS is configured for synchronous communication by default. As we are working with messages we will set IPS to asynchronous mode. We will send an IPS_SET_OPTION message to the UDP protocol driver.

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   ips_option_t       netOpt;
};

sc_msg_t      optmsg;
```

   Allocate an IPS_SET_OPTION message of type ips_option_t.

```
optmsg = sc_msgAlloc (sizeof (ips_option_t)+optlen,
                        IPS_SET_OPTION,
                        SC_DEFAULT_POOL,
                        SC_FATAL_IF_TMO);
```

   Fill the message body.

optmsg->netOpt.base.id	IPS_SET_OPTION (Filled by sc_msgAlloc())
optmsg->netOpt.base.error	0
optmsg->netOpt.base.handle	Access handle: openmsg->netOpen.base.handle
optmsg->netOpt.level	SOL_SOCKET
optmsg->netOpt.optname	SO_SC_ASYNC (The process will now receive all messages from the UDP protocol driver
optmsg->netOpt.optlen	Size of char as optval is a char. Could also be an int or long but at the minimum size of char.
optmsg->netOpt.optval[0]	1 (async mode will be switched on)

   Send the message to the controller process of the UDP protocol driver.

```
sc_msgTx (&optmsg, udpmsg->netUDP.controller, 0);
```

   Receive the IPS_SET_OPTION_REPLY message from the UDP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_SET_OPTION_REPLY, 0 };

optmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

   Check optmsg->netOpt.base.error for a returned error condition.

2. Network buffer message definition:

```
union sc_msg {
   sc_msgid_t          id;
   sdd_netbuf_t        netBuf;
};
sc_msg_t      netmsg;
```

3. Receive the SDD_NET_RECEIVE message from the IPS stack.

```
static const sc_msgid_t select[2] = { SDD_NET_RECEIVE, 0 };

netmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

4. You can retrieve the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

5. Sending the IPS_ACK message.

   Message definition:

```
union sc_msg {
   sc_msgid_t          id;
   ips_ack_t           netAck;
};

sc_msg_t      ackmsg;
```

   Allocate an IPS_ACK message of type ips_ack_t.

```
ackmsg = sc_msgAlloc (sizeof (ips_option_t),
                        IPS_ACK,
                        SC_DEFAULT_POOL,
                        SC_FATAL_IF_TMO);
```

   Fill the message body.

ackmsg->netAck.base.id	IPS_ACK (Filled by sc_msgAlloc())
ackmsg->netAck.base.error	0
ackmsg->netAck.base.handle	Access handle: openmsg->netOpen.base.handle
ackmsg->netAck.size	Size of the received network buffer.

   Send the message to the controller process of the UDP protocol driver.

```
sc_msgTx (&ackmsg, udpmsg->netUDP.controller, 0);
```

SCIOPTA ARM - IPS

5.6.4 TCP Sending Using the IPS Message Interface

1. Getting the TCP SDD protocol descriptor

To be able to access the TCP protocol driver we need to get the TCP SDD protocol descriptor from the IPv4 protocol driver. The IPv4 protocol driver has automatically registered the SDD protocol descriptors and can be accessed like manager process with a SDD_MAN_GET message.

Message definition:

```
union sc_msg {
   sc_msgid_t          id;
   sdd_obj_t           netTCP;
};

sc_msg_t     tcpmsg;
```

Allocate an SDD_MAN_GET message of type sdd_obj_t.

tcpmsg->netTCP.base.id	SDD_MAN_GET
tcpmsg->netTCP.base.error	0
tcpmsg->netTCP.base.handle	0
tcpmsg->netTCP.manager	Manager Handle of the IPv4 protocol driver (copied from ipv4msg->netIPv4.base.handle)
tcpmsg->netTCP.type	0
tcpmsg->netTCP.name	“tcp” (We need the TCP SDD protocol descriptor)
tcpmsg->netTCP.controller	0
tcpmsg->netTCP.sender	0
tcpmsg->netTCP.receiver	0

Sent this message to the controller process of the IPv4 protocol driver.

```
sc_msgTx (&tcpmsg, ipv4msg->netIPV4.controller, 0);
```

Receive the SDD_MAN_GET_REPLY message from the IPv4 protocol driver.

```
static const sc_msgid_t select[2] = { SDD_MAN_GET_REPLY, 0 };

msg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The SDD_MAN_GET_REPLY message is sent by the IPv4 protocol driver and received. The received message is the TCP SDD protocol descriptor and contains all information how to access the TCP protocol driver.

tcpmsg->netTCP.base.id	SDD_MAN_GET_REPLY
tcpmsg->netTCP.base.error	Possible error returned by the IPv4 protocol driver
tcpmsg->netTCP.base.handle	Handle of the TCP protocol driver
tcpmsg->netTCP.manager	not modified
tcpmsg->netTCP.type	0 (not modified)
tcpmsg->netTCP.name	“tcp” (not modified)
tcpmsg->netTCP.controller	controller process ID of the TCP protocol driver
tcpmsg->netTCP.sender	sender process ID of the TCP protocol driver
tcpmsg->netTCP.receiver	receiver process ID of the TCP protocol driver

2. Opening the TCP protocol driver

To be able to connect and to communicate with the TCP protocol driver we need to open it. This will return the TCP SDD protocol descriptor which contains the access handle.

Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_netOpen_t      netOpen;
};

sc_msg_t      openmsg;
```

Allocate a SDD_NET_OPEN message of type sdd_netOpen_t. This message is documented in the SCIOPTA - BSP and Device Driver manual.

```
openmsg = sc_msgAlloc (sizeof (sdd_netOpen_t),
                     SDD_NET_OPEN,
                     SC_DEFAULT_POOL,
                     SC_FATAL_IF_TMO);
```

Fill the message body.

openmsg->netOpen.base.id	SDD_NET_OPEN (Filled by sc_msgAlloc())
openmsg->netOpen.base.error	0
openmsg->netOpen.base.handle	handle of the TCP protocol driver (copied fromtcpmsg->netTCP.base.handle).
openmsg->netOpen.context	0

Send this message to the controller process of the TCP protocol driver.

```
sc_msgTx (&openmsg, tcpmsg->netTCP.controller, 0);
```

Receive the SDD_NET_OPEN_REPLY message from the TCP protocol driver.

```
static const sc_msgid_t select[2] = { SDD_NET_OPEN_REPLY, 0 };

openmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The SDD_NET_OPEN_REPLY message is sent by TCP protocol driver and received. The received message contains the access handle of the TCP protocol driver.

openmsg->netOpen.base.id	SDD_NET_OPEN_REPLY
openmsg->netOpen.base.error	Possible error returned by the TCP protocol driver
openmsg->netOpen.base.handle	Access handle of the TCP protocol driver
openmsg->netOpen.flags	not modified

We have now all information to access the TCP protocol driver.

1. TCP controller process of the TCP protocol driver: **`tcpmsg->netTCP.controller`**
2. Access handle: **`openmsg->netOpen.base.handle`**

3. **Binding** (server and client side)

   Binding is used to define a specific slot where TCP will receive connections after a listen. Binds the access handle to a local IP address and port number. We will send an IPS_BIND message to the TCP protocol driver.

   Message definition:

```
union sc_msg {
   sc_msgid_t          id;
   ips_bind_t          netBind;
};

sc_msg_t      bindmsg;
```

   Allocate an IPS_BIND message of type ips_bind_t.

```
bindmsg = sc_msgAlloc (sizeof (ips_bind_t),
                  IPS_BIND,
                  SC_DEFAULT_POOL,
                  SC_FATAL_IF_TMO);
```

   Fill the message body.

bindmsg->netBind.base.id	IPS_BIND (Filled by sc_msgAlloc())
bindmsg->netBind.base.error	0
bindmsg->netBind.base.handle	Access handle: openmsg->netOpen.base.handle
bindmsg->netBind.srcPort	Source port, netbyte order htons.
bindmsg->netBind.srcAddr.len	4 (for IPv4 adress).
bindmsg->netBind.srcAddr.addr[]	Source address.

   Send the message to the controller process of the TCP protocol driver.

```
sc_msgTx (&bindmsg, tcpmsg->netTCP.controller, 0);
```

   Receive the IPS_BIND_REPLY message from the TCP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_BIND_REPLY, 0 };

bindmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

   Check bindmsg->netBind.base.error for a returned error condition.

4. **Listening** (server side)

   Listen is waiting on connection requests of destination peers on the binded source port and IP address. We will send an IPS_LISTEN message to the controller process of the TCP protocol driver.

   Message definition:

```
union sc_msg {
   sc_msgid_t          id;
   ips_listen_t        netListen;
};

sc_msg_t      listenmsg;
```

   Allocate an IPS_LISTEN message of type ips_listen_t.

```
listenmsg = sc_msgAlloc (sizeof (ips_listen_t),
                  IPS_LISTEN,
                  SC_DEFAULT_POOL,
                  SC_FATAL_IF_TMO);
```

Fill the message body.

listenmsg->netListen.base.id	IPS_LISTEN (Filled by sc_msgAlloc())
listenmsg->netListen.base.error	0
listenmsg->netListen.base.handle	Access handle: openmsg->netOpen.base.handle
listenmsg->netListen.backlog	Maximum number of connection request by the peer which will be queued. Value: 0 ... n (but not more thane specified in the TCP protocol driver).

Send the message to the controller process of the TCP protocol driver.

```
sc_msgTx (&listenmsg, tcpmsg->netTCP.controller, 0);
```

Receive the IPS_LISTEN_REPLY message from TCP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_LISTEN_REPLY, 0 };

listenmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

Check listenmsg->netListen.base.error for a returned error condition.

5. **Accepting** (server side)

Accepting is used to accept a connection request of a destination peer and to establish the connection. We will send an IPS_ACCEPT message to the TCP protocol driver.

Message definition:

```
union sc_msg {
   sc_msgid_t        id;
   ips_accept_t      netAccept;
};

sc_msg_t     acceptmsg;
```

Allocate an IPS_ACCEPT message of type ips_accept_t.

```
acceptmsg = sc_msgAlloc (sizeof (ips_accept_t),
                           IPS_ACCEPT,
                           SC_DEFAULT_POOL,
                           SC_FATAL_IF_TMO);
```

Fill the message body.

acceptmsg->netAccept.base.id	IPS_ACCEPT (Filled by sc_msgAlloc())
acceptmsg->netAaccept.base.error	0
acceptmsg->netAccept.base.handle	Access handle: openmsg->netOpen.base.handle
acceptmsg->netAccept.dstPort	not used
acceptmsg->netAccept.dstAddr.len	not used
acceptmsg->netAccept.dstAddr.addr[]	not used

Send the message to the controller process of the TCP protocol driver.

```
sc_msgTx (&acceptmsg, tcpmsg->netTCP.controller, 0);
```

Receive the IPS_ACCEPT_REPLY message from the TCP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_ACCEPT_REPLY, 0 };

acceptmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

The IPS_ACCEPT_REPLY message is sent by the TCP protocol driver and received. The received message is a new (accepted) SDD protocol descriptor and contains information how to access an accepted TCP connection:

acceptmsg->netAccept.base.id	IPS_ACCEPT_REPLY
acceptmsg->netAccept.error	Possible returned error
acceptmsg->netAccept.base.handle	**Accepted access handle** for the connection
acceptmsg->netAccept.dstPort	Destination port of the accepted connection, netbyte order.
acceptmsg->netAccept.dstAddr.len	4 (for IPv4 adress), atons for reading of the ports.
acceptmsg->netAccept.dstAddr.addr[]	Destination IP address of the accepted connection.

We have now the access handle to communicate with the accepted and connected destination peer:

Accepted access handle: **`acceptmsg->netAopen.base.handle`**

6. **Connecting** (client side)

This used to connect to the destination peer. We will send an IPS_CONNECT message to the TCP protocol driver.

Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   ips_connect_t      netConnect;
};

sc_msg_t      connectmsg;
```

Allocate an IPS_CONNECT message of type ips_connect_t.

```
connectmsg = sc_msgAlloc (sizeof (ips_connect_t),
                   IPS_BIND,
                   SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);
```

Fill the message body.

connectmsg->netConnect.base.id	IPS_CONNECT (Filled by sc_msgAlloc())
connectmsg->netConnect.base.error	0
connectmsg->netConnect.base.handle	Access handle: openmsg->netOpen.base.handle
connectmsg->netConnect.dstPort	Destination port, netbyte oder htons
connectmsg->netConnect.dstAddr.len	4 (for IPv4 adress).
connectmsg->netConnect.dstAddr.addr[]	Destination address.

Send the message to the controller process of the TCP protocol driver.

```
sc_msgTx (&connectmsg, tcpmsg->netTCP.controller, 0);
```

Receive the IPS_CONNECT_REPLY message from TCP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_CONNECT_REPLY, 0 };

connectmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

Check connectmsg->netConnect.base.error for a returned error condition.

7. Setting-Up the **Network Buffer**

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_netbuf_t       netBuf;
};

sc_msg_t      netmsg;
```

   Use the **ips_alloc** function to allocate a network buffer.

```
netmsg = (sc_msg_t) ips_alloc (netbufsize,
                                     SC_DEFAULT_POOL,
                                     SC_FATAL_IF_TMO)
```

   Set the message id :

```
netmsg->netBuf.base.id = SDD_NET_SEND;
```

   Set the access handle:

   For a TCP connection on a server side (**accepted** access handle):

```
netmsg->netBuf.base.handle = aopenmsg->netAopen.base.handle;
```

   For a TCP connection on a client side:

```
netmsg->netBuf.base.handle = openmsg->netOpen.base.handle;
```

   Fill the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

8. **Sending** the Network Buffer

   Send the message to the sender process of the TCP protocol driver.

```
sc_msgTx (&netmsg, tcpmsg->netTCP.sender, 0);
```

5.6.5 TCP Receiving Using the IPS Message Interface

The device driver will send a SDD_NET_RECEIVE message to the SCP_link process of the IPS stack if the network data came in. The user process waits on a SDD_NET_RECEIVE message coming from the IPS stack.

Before you can receive TCP data you should get the TCP SDD protocol descriptor and open the TCP protocol driver as described in chapter 5.6.4 "TCP Sending Using the IPS Message Interface" on page 5-14.

1. **Receive Mode** Setting

   IPS is configured for synchronous communication by default. As we are working with messages we will set IPS to asynchronous mode. We will send an IPS_SET_OPTION message to the TCP protocol driver.

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   ips_option_t       netOpt;
};

sc_msg_t      optmsg;
```

   Allocate an IPS_SET_OPTION message of type ips_option_t.

```
optmsg = sc_msgAlloc (sizeof (ips_option_t)+optlen,
                         IPS_SET_OPTION,
                         SC_DEFAULT_POOL,
                         SC_FATAL_IF_TMO);
```

   Fill the message body.

optmsg->netOpt.base.id	IPS_SET_OPTION (Filled by sc_msgAlloc())
optmsg->netOpt.base.error	0
optmsg->netOpt.base.handle	Access handle: openmsg->netOpen.base.handle
optmsg->netOpt.level	SOL_SOCKET
optmsg->netOpt.optname	SO_SC_ASYNC (The process will now receive all messages from the TCP protocol driver
optmsg->netOpt.optlen	Size of char as optval is a char. Could also be an int or long but at the minimum size of char.
optmsg->netOpt.optval[0]	1 (async mode will be switched on)

   Send the message to the controller process of the TCP protocol driver.

```
sc_msgTx (&optmsg, tcpmsg->netTCP.controller, 0);
```

   Receive the IPS_SET_OPTION_REPLY message from the TCP protocol driver.

```
static const sc_msgid_t select[2] = { IPS_SET_OPTION_REPLY, 0 };

optmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

   Check optmsg->netOpt.base.error for a returned error condition.

2. Network buffer message definition:

```
union sc_msg {
   sc_msgid_t         id;
   sdd_netbuf_t       netBuf;
};
sc_msg_t      netmsg;
```

3. Receive the SDD_NET_RECEIVE message from the IPS stack.

```
static const sc_msgid_t select[2] = { SDD_NET_RECEIVE, 0 };

netmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

4. You can retrieve the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

5. Sending the IPS_ACK message.

   Message definition:

```
union sc_msg {
   sc_msgid_t         id;
   ips_ack_t          netAck;
};

sc_msg_t      ackmsg;
```

   Allocate an IPS_ACK message of type ips_ack_t.

```
ackmsg = sc_msgAlloc (sizeof (ips_option_t),
                      IPS_ACK,
                      SC_DEFAULT_POOL,
                      SC_FATAL_IF_TMO);
```

   Fill the message body.

ackmsg->netAck.base.id	IPS_ACK (Filled by sc_msgAlloc())
ackmsg->netAck.base.error	0
ackmsg->netAck.base.handle	Access handle: openmsg->netOpen.base.handle
ackmsg->netAck.size	Size of the received network buffer.

   Send the message to the controller process of the TCP protocol driver.

```
sc_msgTx (&ackmsg, tcpmsg->netTCP.controller, 0);
```

5.7 Using the IPS Function Interface

5.7.1 Introduction

The IPS Function Interface was introduced to simplify the implementation of the BSD Socket Interface.

But the user can also access this IPS Function Interface directly to encapsulate the message passing between the client and the IPS TCP/IP stack.

As the set-up of a communication uses normally always the same message passing procedures it is obvious to use the IPS Function Interface.

Another reason to use the function interface is that the probability of having modifications in the message interface (on the lowest level) is higher than to have it for the function interface.

While the IPS Function Interface are very useful for configuring and setting-up the IPS stack it is recommended to use the IPS Message Interface for receiving and sending the network packages. This will result in systems with higher performance and give the user more flexibility. The Message Interface for sending and receiving will not be modified over time or the modification will be backwards compatible.

Before you can send and receive network data with UDP and TCP you need to setup the routes (see chapter 4.8 “Routing Configuration” on page 4-21).

5.7.2 UDP Sending Using the IPS Function Interface

1. Opening the UDP protocol driver

   To be able to do a connect, bind, send etc. we need to open the UDP protocol driver.

```
sdd_obj_t NEARPTR udp = ips_open ("ipv4", "udp", 0);
```

2. **Binding** (server side)

   Binding is used to define a specific IP address and port number where UDP receives network packages.

```
__u16   srcPort = <source port>;
struct ips_addr_s {
   size_t  len = 4;
   __u8    addr[16] = <source IP address>
} srcAddr;

ret = ips_bind (udp, &srcAddr, htons(srcPort));
```

3. **Connecting** (server and client side)

   Connecting is used to set the default destination peer.

```
__u16   dstPort = <destination port>;
struct ips_addr_s {
   size_t  len = 4;
   __u8    addr[16] = <destination IP address>
} dstAddr;

ret = ips_connect (udp, &dstAddr, dstPort);
```

4. Setting-Up the **Network Buffer**

   Use the ips_alloc function to allocate a network buffer.

```
sdd_netbuf_t *netBuf = ips_alloc (netbufsize,
                                  SC_DEFAULT_POOL,
                                  SC_FATAL_IF_TMO)
```

   Fill the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

5. **Sending** the Network Buffer

   Send the message to the sender process of the UDP protocol driver

```
ips_send (udp, &netBuf);
```

5.7.3 UDP Receiving Using the IPS Function Interface

The device driver will send a SDD_NET_RECEIVE message to the SCP_link process of the IPS stack if the network data came in. The user process waits on a SDD_NET_RECEIVE message coming from the IPS stack. Receiving UDP Data with the IPS Function Interface is done in the same way as with the IPS Message Interface.

Before you can receive UDP data you should get the UDP SDD protocol descriptor and open the UDP protocol driver as described in chapter 5.7.2 “UDP Sending Using the IPS Function Interface” on page 5-22.

1. **Receive Mode** Setting

   IPS is configured for synchronous communication by default. As we are working with messages we will set IPS to asynchronous mode.

```
__u8 opt=1;  /* Also int opt=1 can be used */

ret = ips_setOption (udp, SOL_SOCKET, SO_SC_ASYNC, &opt, sizeof(opt));
```

2. Network buffer message definition:

```
union sc_msg {
   sc_msgid_t          id;
   sdd_netbuf_t        netBuf;
};
sc_msg_t      netmsg;
```

3. Receive the SDD_NET_RECEIVE message from the IPS stack.

```
static const sc_msgid_t select[2] = { SDD_NET_RECEIVE, 0 };

netmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
```

4. You can retrieve the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

5. Acknowledge the received network buffer.

```
ips_netbufAck (udp, netmsg);
```

5.7.4 TCP Sending Using the IPS Function Interface

1. Opening the TCP protocol driver

   To be able to do a connect, bind, send etc. we need to open the TCP protocol driver.

```
sdd_obj_t NEARPTR tcp = ips_open ("ipv4", "tcp", 0);
```

2. **Binding** (server and client side)

   Binding is used to define a specific IP address and port number where TCP receives network packages.

```
__u16   srcPort = <source port>;
struct ips_addr_s  {
   size_t  len = 4;
   __u8    addr[16] = <source IP address>
} srcAddr;

ret = ips_bind (tcp, &srcAddr, htons(srcPort));
```

3. **Listen** (server side)

   Listen is waiting on connection requests of destination peers on the binded source port and IP address.

```
int backlog =1;

ret = ips_listen (tcp, backlog);
```

4. **Accepting** (server side)

   Accepting is used to accept a connection request of a destination peer and to establish the connection.

```
__u16   dstPort = <destination port>;
struct ips_addr_s  {
   size_t  len = 4;
   __u8    addr[16] = <destination IP address>
} dstAddr;

ret = ips_accept (tcp, &dstAddr, dstPort);
```

5. **Connecting** (client side)

   Connecting is used to set the default destination peer.

```
__u16   dstPort = <destination port>;
struct ips_addr_s  {
   size_t  len = 4;
   __u8    addr[16] = <destination IP address>
} dstAddr;

ret = ips_connect (tcp, &dstAddr, dstPort);
sc_msgTx (&netmsg, tcp->sender, 0);
```

6. Setting-Up the **Network Buffer**

   Use the ips_alloc function to allocate a network buffer.

```
sdd_netbuf_t *netBuf = ips_alloc (netbufsize,
                                  SC_DEFAULT_POOL,
                                  SC_FATAL_IF_TMO)
```

   Fill the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

```
SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
```

7. **Sending** the Network Buffer

   Send the message to the sender process of the TCP protocol driver

```
ips_send (tcp, &netBuf);
```

5.7.5 TCP Receiving Using the IPS Function Interface

The device driver will send a SDD_NET_RECEIVE message to the SCP_link process of the IPS stack if the network data came in. The user process waits on a SDD_NET_RECEIVE message coming from the IPS stack. Receiving TCP Data with the IPS Function Interface is done in the same way as with the IPS Message Interface.

Before you can receive TCP data you should get the TCP SDD protocol descriptor and open the TCP protocol driver as described in chapter 5.7.4 “TCP Sending Using the IPS Function Interface” on page 5-24.

1. **Receive Mode** Setting

   IPS is configured for synchronous communication by default. As we are working with messages we will set IPS to asynchronous mode.

   ```
   char opt=1;

   ret = ips_setOption (tcp, SOL_SOCKET, SO_SC_ASYNC, &opt, sizeof(opt));
   ```

2. Network buffer message definition:

   ```
   union sc_msg {
      sc_msgid_t         id;
      sdd_netbuf_t       netBuf;
   };
   sc_msg_t      netmsg;
   ```

3. Receive the SDD_NET_RECEIVE message from the IPS stack.

   ```
   static const sc_msgid_t select[2] = { SDD_NET_RECEIVE, 0 };

   netmsg = sc_msgRx ( SC_ENDLESS_TMO, (void *)select, SC_MSGRX_MSGID);
   ```

4. You can retrieve the netbuffer (network message) data by using the macro SDD_NET_DATA. This macro ist defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h

   ```
   SDD_NET_DATA (netmsg) [i]; /* i=0 .. (netbufsize-1) */
   ```

5. Acknowledge the received network buffer.

   ```
   ips_netbufAck (tcp, netmsg);
   ```

5.8 Using The IPS BSD Socket Interface

5.8.1 Introduction

SCIOPTA Internet Protocols IPS provides also a BSD Socket Interface. This allows network programmers to use the well known socket function calls to access the IPS Stack and the network.

Before you can send and receive network data with UDP and TCP you need to setup the routes (see chapter **4.8 “Routing Configuration” on page 4-21**).

5.8.2 BSD Descriptors

The BSD Socket Interface is using **BSD File Descriptors** or **BSD Socket Descriptors** to specifying and access sockets. Please note that BSD file descriptors and BSD socket descriptors are exactly identical.

The BSD socket Interface user needs to build-up a **File Descriptor Table**. In SCIOPTA this table is organized as a process variable array and must be initialized as follows:

1. Define and initialize a process variable by allocating a message buffer with enough memory and initializing the process variable by the sc_procVarInit() system call.

   Please consult the SCIOPTA - Kernel, User’s Guide for more information about process variables.

2. Initialize the BSD descriptor table

   SCIO_INIT (“<root manager>”,<any message pool>, <max number of BSD descriptors>).

   SCIO_INIT is defined in the file: <install_dir>\sciopta\<version>\include\scio.h.

Example

```
msg = sc_msgAlloc (SC_TAG_SIZE * 2, 0, SC_DEFAULT_POOL, SC_ENDLESS_TMO);
sc_procVarInit (&msg, 2);

SCIO_INIT ("SCP_rcsman", SC_DEFAULT_POOL, 8);
```

From this moment all socket system call can be used even inside external functions which are called from the process.

The BSD descriptor is just an index in the file descriptor table. The file descriptor table contains the SDD protocol descriptors of the SCIOPTA IPS Stack. Each BSD socket call which generates a socket will return the BSD descriptor (Index) and fill the corresponding SDD protocol descriptor in the table.

5.8.3 Socket Addressing

Types use in sockets address definitions:

```
typedef unsigned int socklen_t;
typedef unsigned short sa_family_t;
```

Defined in the include file: <install_dir>\sciopta\<version>\include\sys\socket.h

5.8.3.1 Generic Socket Address

The generic socket address is defined as follows:

```
struct sockaddr {
  sa_family_t sa_family;
  char sa_data[14];
};
```

All BSD socket function calls are using this generic address.

5.8.3.2 Specific Socket Address

For IP version 4 addresses the following structure is defined:

```
struct sockaddr_in {
   sa_family_t          sin_family;          /* Address family */
   unsigned short int   sin_port;            /* Port number */
   struct in_addr       sin_addr;            /* Internet address*/

/* Pad to size of `struct sockaddr'. */

   unsigned char        __pad[__SOCK_SIZE__ - sizeof (short int) -
                              sizeof (unsigned short int) -
                              sizeof (struct in_addr)];
};
```

The padding is used to fill the remaining parts of the generic address and should not be done by the user.

To avoid compiler warning you need to cast the struct sockaddr_in to struct sockaddr for every BSD system call.

5.8.4 Macros

To handle the little-endian and big-endian issue all address and ports are transformed into the netbyte order:

```
htons    for    __u16
htonl    for    __u32
```

To be able to read the received __u16 and __u32 data packages they must be converted back into the host byte order by using **ntohs** and **ntohl**.

These macros are defined the include file: <install_dir>\sciopta\<version>\include\sys\socket.h

5.8.5 UDP Sending Using the IPS BSD Socket Interface

A connection in the true sense of the word does not exist in UDP. UDP is a connectionless communication. Nevertheless the BSD calls bind and connect are available in IPS and must actually be used for the asynchronous connection as the sendto() and the recvfrom() calls are not yet implemented.

1. Setting-Up the **File Descriptor Table**

   In order to use the BSD socket system calls you need to set-up the file descriptor table

```
union sc_msg {
   sc_msgid_t        id;
};
sc_msg_t     fdmsg;

fdmsg = sc_msgAlloc (SC_TAG_SIZE * 2, 0, SC_DEFAULT_POOL, SC_ENDLESS_TMO);
sc_procVarInit (&fdmsg, 2);

SCIO_INIT ("SCP_rcsman", SC_DEFAULT_POOL, 8);
```

2. Getting the **UDP BSD Socket Descriptor**

```
int sd = socket(PF_INET, SOCK_DGRAM, 0);
```

3. **Sending** the data.

```
struct sockaddr_in  toaddr;

toaddr.sin_family =       AF_INET
toaddr.sin_port =         htons (<destination port number>)
ret =                     inet_aton ("<IP address>", &toaddr.sin_addr)

sendto (sd, <any_data>, len, &toaddr, sizeof(toaddr));
```

SCIOPTA

5.8.6 UDP Receiving Using the IPS BSD Socket Interface

The device driver will send a SDD_NET_RECEIVE message to the SCP_link process of the IPS stack if the network data came in. The user process waits on a SDD_NET_RECEIVE message coming from the IPS stack.

Before you can receive UDP data you should get the UDP BSD socket descriptor as described in chapter 5.8.5 “UDP Sending Using the IPS BSD Socket Interface” on page 5-29.

1. **Binding** (server side)

   Binding is used to define a specific IP address and port number where UDP receives network packages.

```
struct sockaddr_in  sinsrc;

sinsrc.sin_family =       AF_INET
sinsrc.sin_port =         htons (<source port number>)
ret =                     inet_aton ("<IP address>", &sinsrc.sin_addr)

ret = bind (sd, (struct sockaddr *) &sinsrc, sizeof (sinsrc));
```

2. **Receiving** the data.

```
struct sockaddr_in  fromaddr;

fromaddr.sin_family =     AF_INET
fromaddr.sin_port =       htons (<destination port number>)
ret =                     inet_aton ("<IP address>", &fromaddr.sin_addr)

recvfrom (sd, <any_data>, len, &fromaddr, sizeof(fromaddr));
```

5.8.7 TCP Sending Using the IPS BSD Socket Interface

1. Setting-Up the **File Descriptor Table**

   In order to use the BSD socket system calls you need to set-up the file descriptor table

```
union sc_msg {
   sc_msgid_t        id;
};
sc_msg_t     fdmsg;

fdmsg = sc_msgAlloc (SC_TAG_SIZE * 2, 0, SC_DEFAULT_POOL, SC_ENDLESS_TMO);
sc_procVarInit (&fdmsg, 2);

SCIO_INIT ("SCP_rcsman", SC_DEFAULT_POOL, 8);
```

2. Getting the **TCP BSD Socket Descriptor**

```
int sd = socket(AF_INET, SOCK_STREAM, 0);
```

3. **Binding** (server side)

   Binding is used to define a specific IP address and port number where UDP receives network packages.

```
struct sockaddr_in  sinsrc;

sinsrc.sin_family =       AF_INET
sinsrc.sin_port =         htons (<source port number>)
ret =                     inet_aton ("<IP address>", &sinsrc.sin_addr)

ret = bind (sd, (struct sockaddr *) &sinsrc, sizeof (sinsrc));
```

4. **Listening** (server side)

   Listen is waiting on connection requests of destination peers on the binded source port and IP address.

```
int backlog =1;

ret = listen (sd, backlog);
```

5. **Connecting** (client side)

   Connecting is used to set the default destination peer.

```
struct sockaddr_in  sindst;

sindst.sin_family =       AF_INET
sindst.sin_port =         htons (<destination port number>)
ret =                     inet_aton ("<IP address>", &sindst.sin_addr)

ret = connect (sd, (struct sockaddr *) &sindst, sizeof (sindst));
```

6. **Accepting** (server side)

   Accepting is used to accept a connection request of a destination peer and to establish the connection.

```
struct sockaddr_in  sindst;

sindst.sin_family =       AF_INET
sindst.sin_port =         htons (<destination port number>)
ret =                     inet_aton ("<IP address>", &sindst.sin_addr)

nd = accept (sd, (struct sockaddr *) &sindst, sizeof (sindst));
```

   nd is the new accepted TCP BSD socket descriptor.

7. **Sending** the network data (client)

```
ret = ips_send (sd, <data>, len);
```

8. **Sending** the network data (server)

```
ret = ips_send (nd, <data>, len);
```

5.8.8 TCP Receiving Using the IPS BSD Socket Interface

The device driver will send a SDD_NET_RECEIVE message to the SCP_link process of the IPS stack if the network data came in. The user process waits on a SDD_NET_RECEIVE message coming from the IPS stack.

Before you can receive TCP data you should get the TCP BSD socket descriptor as described in chapter **5.8.7 "TCP Sending Using the IPS BSD Socket Interface" on page 5-31**.

1. **Receiving** the network data

```
recv (sd, <data>, len);
```

5.9 BSD Descriptor and SDD Descriptor

Sometimes it is useful to get the SDD descriptor from a BSD descriptor or on the other hand to hook an SDD descriptor in a socket. This might be used if you want to make an SDD descriptor available to another process (e.g. in a multithreaded server).

For instance, in one process there is a BSD descriptor **nd** (which e.g. was obtained by an accept call). Now you need to get the SDD descriptor to duplicate it and to send it to another process which can work on it:

```
sdd_obj_t NEARPTR obj, *cpy;
fcntl (nd, F_SC_GETBIOSHDL, &obj);
cpy =  sdd_objDup (obj)
```

The object **cpy** can now be send to another process. The receiving process can save this SDD descriptor into its file descriptor table and can continue to use the BSD socket calls. There is a clean way to do this:

Mostly the BSD descriptors 0,1 and 2 are reserved for standard in, out and err (implementation dependent). We want to store the received instance in descriptor 3. The name of the SDD descriptor is **myobj** and it is of type sdd_obj_t.

1. First we duplicate and save BSD descriptor 3 (if we don not know if descriptor 3 is available).

```
new3 = dup (3);
```

2. BSD descriptor 3 will be closed.

```
close (3);
```

3. The received SDD descriptor **myobj** will be stored in BSD descriptor 3.

```
if (fcntl (3, F_SC_SETBIOSHDL, myobj) == -1) {
   error ();
}
```

An

```
ips_send (3, &netbuf);
```

will send to the installed socket.

6 System Building

6.1 Introduction

In this chapter we will give you some information about the building process for a ARM target system using SCIOPTA IPS Internet Protocols.

Please consult chapter “System Building” of the SCIOPTA ARM - Kernel, User’s Guide for general information about system building. You will find there information about:

- System Files
- Data Types
- System Building Diagramm
- Assembling, Compiling and Linking
- Kernel and Utilities Libraries
- Include Files
- Linker Scripts
- Memory Map

6.2 System Design

First you have 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.

6.3 System Files

For initializing and setting up your target you need some specific system files such as board setup files, files to initialize the C/C++ environment, device driver and interrupt handler files. You will find SCIOPTA system files for many popular boards in the Board Support Package (BSP) delivery. If you are using a different board or you are designing your own hardware these file represent a good model and starting point for writing your own system files.

Please consult chapter **7 “Board Support Packages” on page 7-1** and the SCIOPTA - Device Driver, User’s Guide.

6.4 Include Files

No specific include files search directories for IPS internet protocols needs to be declared. Please consult chapter "Include Files" of the SCIOPTA ARM - Kernel, User's Guide for all information about include files.

6.5 SCIOPTA ARM IPS Libraries

6.5.1 Delivered IPS Libraries

IPS Internet Protocols (TCP/IP)	libips_**XY**.a	GCC
	ips_**XY**.l	ARM RealView
	ips_**XY**.r79	IAR Systems

6.5.2 Optimization "X"

The libraries are delivered for three different compiler optimization. The letter **X** defines one of three compiler optimization levels.

"X" can have a value of 0,1 and 2 and defines the optimization.

0 No Optimization.

1 Optimization for size.

2 Optimization for speed.

6.5.3 IPS Internet Protocol Libraries "Y"

For the SCIOPTA IPS Internet Protocols there are libraries for three different level of network system complexity included.

"Y" can be not present or can have a value of **s** or **f**.

<none> No letter. This for standard systems needing usual network functionality.

s The letter "s". This is for **small** systems needing just limited network functionality.

f The letter "f". This is for systems needing **full** featured networking support.

The following tables shows the included features for each of the three IPS levels:

IPS Feature	<none> standard	"s" small	"f" full
Logd (Log Daemon)	not included	not included	not included
Raw Protocol	not included	not included	included
IP4 Cache Size	4 packets	2 packets	2 packets
IP4 Fragmentation	not included	not included	included

IPS Feature	<none> standard	“s” small	“f” full
IP4 Forwarding	included	not included	included
IP4 ARP	included	included	included
IP4 ARP Device	included	not included	included
IP4 ARP Cache	4 entries	2 entries	4 entries
IP4 ARP Max Probes	2 probes	2 probes	2 probes
IP4 Router	included	not included	included
IP4 Router Device	included	not included	included
IP4 Route Get	included	not included	included
IP4 Route Wait	included	not included	included
IP4 MIB Statistics	not included	not included	included
IP4 Raw	not included	not included	included
ICMP4 MIB Statistics	not included	not included	included
ICMP4 Device	included	not included	included
UDP Checksum	included	included	included
UDP Socket Enumeration	not included	not included	included
UDP MIB Statistics	not included	not included	included
TCP Multiple Pools	included	not included	included
TCP Cork Algorithm	included	included	included
TCP Piggy Pack Algorithm	included	not included	included
TCP Nagle Algorithm	included	not included	included
TCP Silly Window Algorithm	included	included	included
TCP Time Stamps	not included	not included	not included
TCP MIB Statistics	not included	not included	included
TCP Socket Enumeration	not included	not included	included

6.6 Linking the SCIOPTA ARM System

Please consult chapter “Linking the SCIOPTA ARM System” of the SCIOPTA ARM - Kernel Manual for general information about linking.

In addition you need to link the corresponding IPS library (libips_XY.a).

7 Board Support Packages

Only the device driver for SCIOPTA ARM - IPS Internet protocols are listed here. Please consult chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for all other information about SCIOPTA BSPs.

7.1 CPU and Board Independent Drivers

7.1.1 SMSC91C111 Ethernet Controller Low-Level Driver

This is a low-level driver for the SMSC91C111 ethernet controller. This driver includes low-level functions which are either CPU nor board dependent and can be used for all boards using the SMSC91C111 chip.

7.1.1.1 SMSC91C111 Ethernet Controller Include Files

smsc91c111.h	Defines for SMSC91C111.

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

7.1.1.2 SMSC91C111 Ethernet Controller Source Files

smsc91c111.c	Lowlevel part of the driver for SMSC91C111.

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

7.2 Atmel AT91SAM7X Boards and Drivers

7.2.1 Atmel AT91SAM7X Ethernet Driver

This is a driver for the Atmel AT91SAM7X chip using the on-chip ethernet controller. This driver is not board dependent and can be used for all boards using the Atmel AT91SAM7X chip.

7.2.1.1 Atmel AT91SAM7X Ethernet Driver Include Files

eth.h	Ethernet driver include file for AT91SAM7x.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\include\

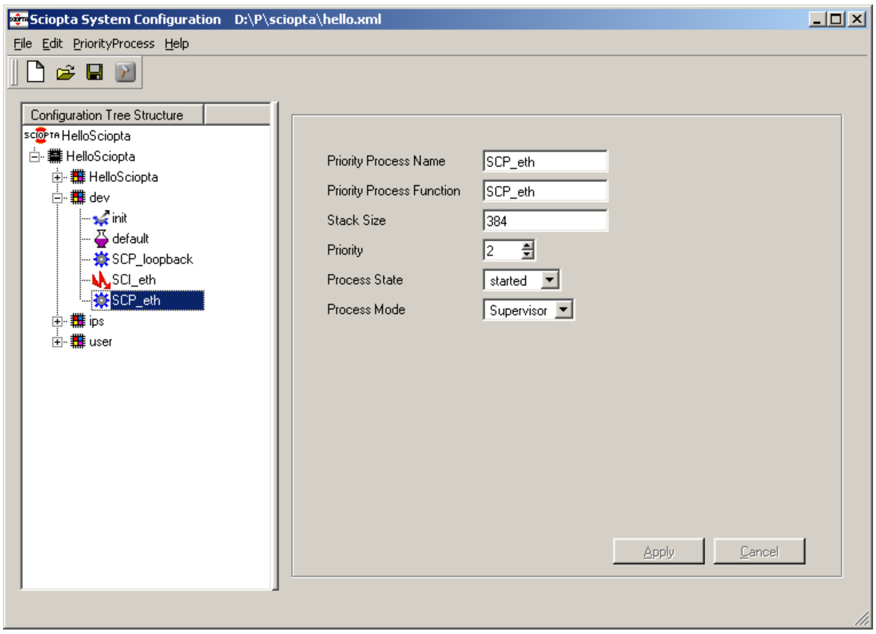
7.2.1.2 Atmel AT91SAM7X Ethernet Driver Source Files

eth.c	Ethernet driver for AT91SAM7x.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\src\

7.2.1.3 Atmel AT91SAM7X Ethernet Driver Prioritized Processes

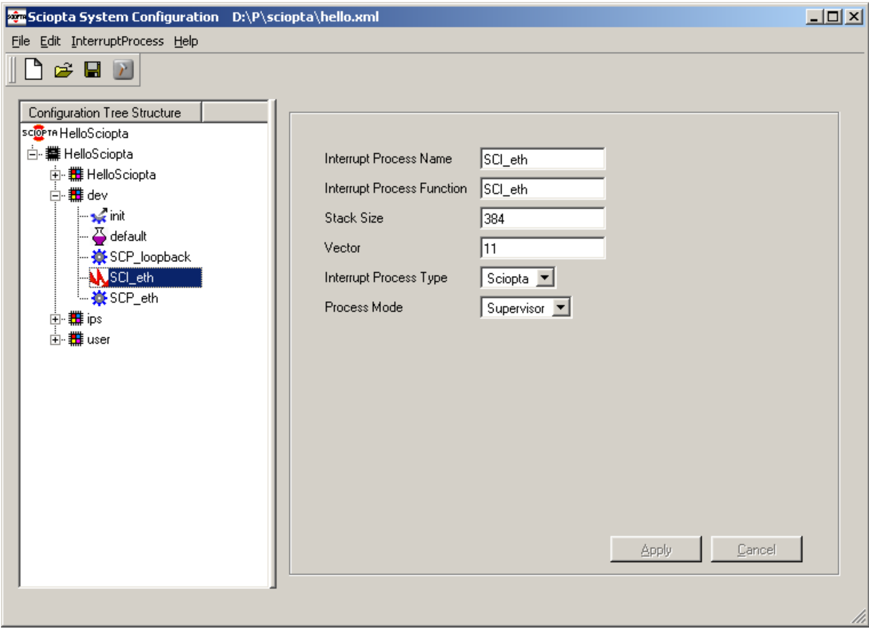
You need to declare the process **SCP_eth** in the SCIOPTA SCONF Utility:



The process is included in the file eth.c.

7.2.1.4 Atmel AT91SAM7X Ethernet Driver Interrupt Processes

You need to declare the interrupt process **SCI_eth** in the SCIOPTA SCONF Utility:



The process is included in the file eth.c.

7.3 STMicroelectronics STR9 Boards and Drivers

7.3.1 STMicroelectronics STR9 Ethernet Driver

This is a driver for the STMicroelectronics STR9 chip using the on-chip ethernet controller. This driver is not board dependent and can be used for all boards using the STMicroelectronics STR9 chip.

7.3.1.1 STMicroelectronics STR9 Ethernet Driver Include Files

eth.h	Ethernet driver include file for AT91SAM7x.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\include\

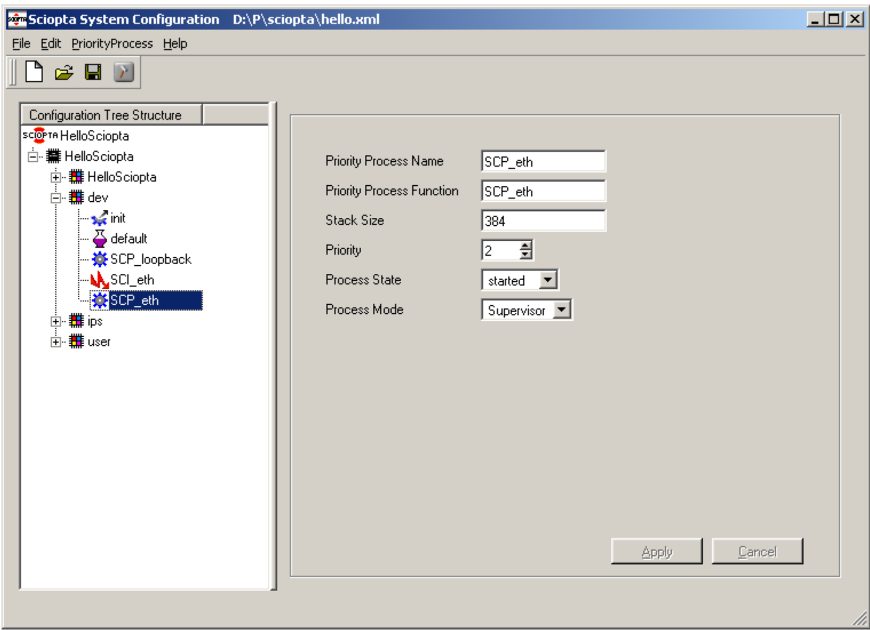
7.3.1.2 STMicroelectronics STR9 Ethernet Driver Source Files

eth.c	Ethernet driver for AT91SAM7x.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\src\

7.3.1.3 STMicroelectronics STR9 Ethernet Driver Prioritized Processes

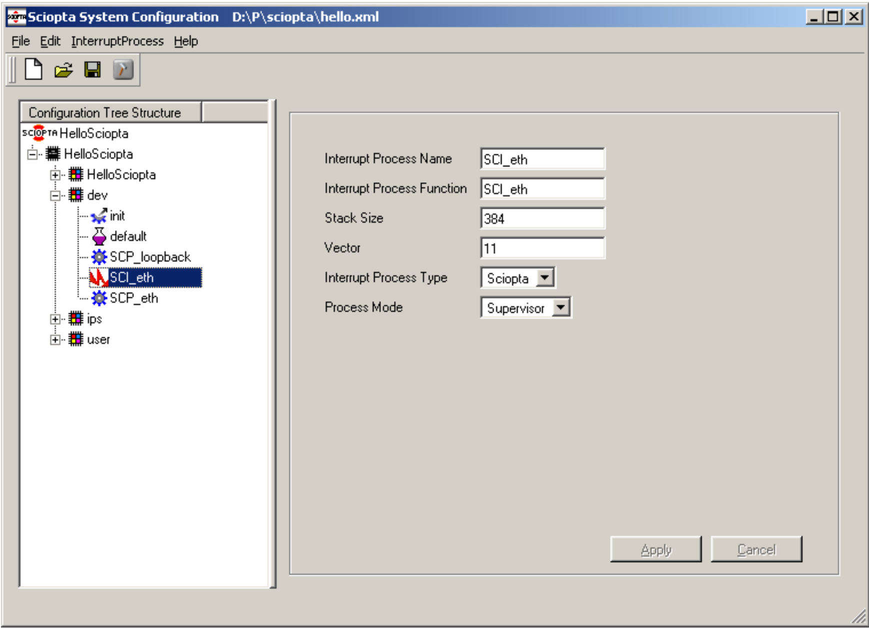
You need to declare the process **SCP_eth** in the SCIOPTA SCONF Utility:



The process is included in the file eth.c.

7.3.1.4 STMicroelectronics STR9 Ethernet Driver Interrupt Processes

You need to declare the interrupt process **SCI_eth** in the SCIOPTA SCONF Utility:



The process is included in the file eth.c.

7.4 phyCORE-LPC2294 Board

7.4.1 SMSC91C111 Ethernet Controller Driver

This is a driver for the SMSC91C111 ethernet controller on the phyCORE-LPC2294 board.

7.4.1.1 SMSC91C111 Ethernet Driver Include Files

smsc91c111_p.h Hardware dependent part of the SMSC91C111 driver.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc21xx\phyCore2294\include\

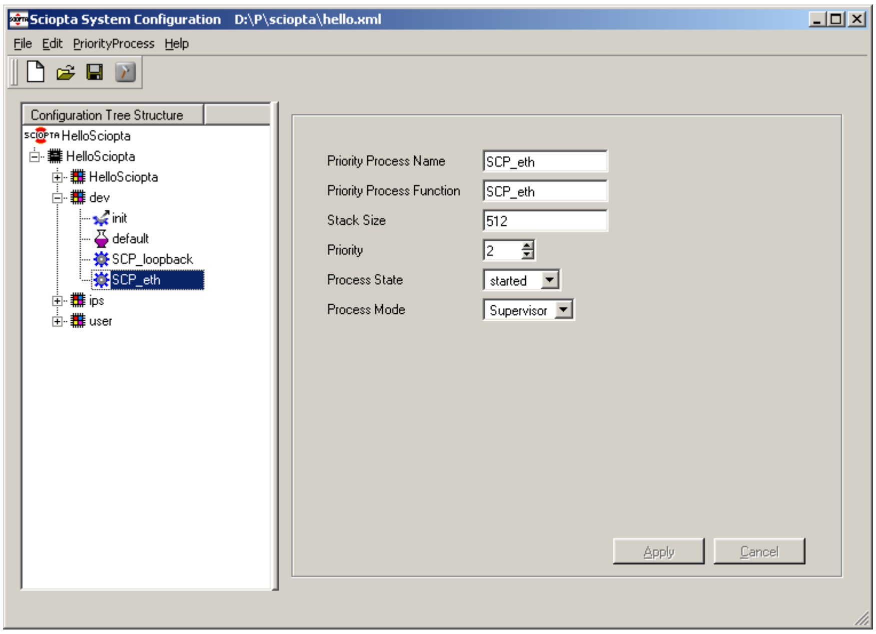
7.4.1.2 SMSC91C111 Ethernet Driver Source Files

eth.c Ethernet driver for SMCS SMSC91C111.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc21xx\phyCore2294\src\

7.4.1.3 SMSC91C111 Ethernet Driver Prioritized Processes

You need to declare the process **SCP_eth** in the SCIOPTA SCONF Utility. The ethernet driver interrupt processes are dynamically created.



The process is included in the file eth.c.

8 Structures

8.1 Base SDD Object Descriptor Structure sdd_baseMessage_t

The base SDD object descriptor structure is the basic component of all SDD object descriptors. It is inherited by all other specific SDD object descriptors and represents the smallest common denominator.

It contains the message ID (SDD object descriptors are SCIOPTA messages), an error variable and the handle of the SDD object.

```
typedef struct sdd_baseMessage_s {
   sc_msgid_t            id;
   sc_errorcode_t        error;
   void                  *handle;
} sdd_baseMessage_t;
```

Members

id

Standard SCIOPTA message ID.

error

Error code.

handle

Handle of the SDD object. This is usually a pointer to a structure which further specifies the SDD object.

The user of a device object which is opening and closing the device, reading from the device and writing to the device does not need to know the handle and the handle structure. The user will usually get the SDD device descriptor by using the **sdd_manGetByName** function call. The SDD device manager will return the SDD device descriptor including the handle.

Only processes inside the SDD object (the device driver) may access and use the handle.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

SCIOPTA

8.2 Standard SDD Object Descriptor Structure sdd_obj_t

This structure contains more specific information about SDD objects such as types, names and process IDs. It is an extension of the base SDD object descriptor structure **sdd_baseMessage_t**.

```
typedef struct sdd_obj_s {
   sdd_baseMessage_t     base;
   void                  *manager;
   sc_msgid_t            type;
   unsigned char         name[SC_NAME_MAX + 1];
   sc_pid_t              controller;
   sc_pid_t              sender;
   sc_pid_t              receiver;
} sdd_obj_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (see chapter **8.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 8-1**).

manager

Contains a manager access handle. It is a pointer to a structure which further specifies the manager.

This is only used if the SDD object descriptor describes an SDD manager and is only used in SDD manager messages (**SDD_MAN_XXX**).

For SDD file managers a 0 defines an SDD root manager.

You do not need to write anything in the manager handle if you are using the function interface as this is done in the interface layer.

type

Type of the SDD object. More than one value can be defined and must be separated by OR instructions. The values determine the type of messages which are handled by the SDD object.

This member can be one or more of the following values:

Value	Meaning
SDD_OBJ_TYPE	General SDD object type. Handles the following messages:
	SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY
	SDD_OBJ_DUPLICATE / SDD_OBJ_DUPLICATE_REPLY
	SDD_OBJ_INFO / SDD_OBJ_INFO_REPLY
SDD_MAN_TYPE	The SDD object is an SDD manager. It handles the following manager messages:
	SDD_MAN_ADD / SDD_MAN_ADD_REPLY
	SDD_MAN_RM / SDD_MAN_RM_REPLY
	SDD_MAN_GET / SDD_MAN_GET_REPLY
	SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY
	SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY

SDD_DEV_TYPE	The SDD object is an SDD device. It handles the following device messages: SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY SDD_DEV_DUALOPEN / SDD_DEV_DUALOPEN_REPLY SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY SDD_DEV_READ / SDD_DEV_READ_REPLY SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY
SDD_FILE_TYPE	The SDD object is an SDD file. It handles the following file messages: SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY
SDD_NET_TYPE	The SDD object is an SDD protocol or network device. It handles the following network messages: SDD_NET_RECEIVE / SDD_NET_RECEIVE_REPLY SDD_NET_RECEIVE_2 / SDD_NET_RECEIVE_2_REPLY SDD_NET_RECEIVE_URGENT / SDD_NET_RECEIVE_URGENT_REPLY SDD_NET_SEND / SDD_NET_SEND_REPLY

name

Contains the name of the SDD object. The name must be unique within a domain. A manager corresponds to a domain.

controller

The controller process ID of the SDD object.

sender

The sender process ID of the SDD object. If the SDD object is a device driver, the sender process sends the data to the physical layer. It usually receives SDD_DEV_WRITE or SDD_NET_SEND messages and can reply with the corresponding reply messages.

receiver

The receiver process ID of the SDD object. If the SDD object is a device driver, the receiver process receives the data from the physical layer. In passive synchronous mode the receiver process receives the SDD_DEV_READ messages and replies with the SDD_DEV_READ_REPLY message. In active asynchronous mode (used by network devices) the receiver process sends a SDD_NET_RECEIVE, SDD_NET_RECEIVE_2 or SDD_NET_RECEIVE_URGENT message.

Remarks

For specific or simple SDD objects the process IDs for **controller**, **sender** and **receiver** can be the same. These SDD objects contain therefore just one process.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

8.3 Base SDD Object Info Structure sdd_objInfo_t

The base SDD object info structure is used in message for getting generic information about SDD objects. It is an extension of the base SDD object descriptor structure **sdd_baseMessage_t**.

```
typedef struct sdd_objInfo_s {
   sdd_baseMessage_t    base;
   int                  ref;
} sdd_objInfo_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (see chapter 8.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 8-1).

ref

Number of references. Number of processes which accesses this device.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

8.4 SDD Network Buffer Structure sdd_netbuf_t

To send data over a network a specific data buffer is needed. The network buffer is used by protocol drivers or network device drivers. It is an extension of the base SDD object descriptor structure **sdd_baseMessage_t**.

Network buffers normally travels through some protocol drivers (layers) before it is sent to the network device driver.

```
typedef struct sdd_netbuf_s {
   sdd_baseMessage_t    base;
   void                 (*doBeforeSend)(sdd_netbuf_t * netbuf);
   sc_pid_t             sendItBackTo;
   int                  pktype;
   int                  protocol;
   size_t               size;
   size_t               cur;
   size_t               head;
   size_t               data;
   size_t               tail;
   size_t               end;
   unsigned char        inlineBuf[1];
} sdd_netbuf_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (see chapter **8.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 8-1**).

doBeforeSend

Was introduced for protocols which ignore layer structures. With **doBeforeSend** a higher protocol layer could still do some work in the buffer. The detail actions depend on the protocol.

You do not need to touch this variable for normal network application programming.

sendItBackTo

Client process ID to send back an acknowledge by using an SDD_NET_SEND_REPLY message (the received message ID can just be incremented by one).

pkttype

Network packet types.

This member can be one of the following values:

Value	**Meaning**
SDD_BROADCAST_PKT	Broadcast packet.
SDD_MULTICAST_PKT	Multicast packet.
SDD_HOST_PKT	Packet exactly for this host.
SDD_OTHERHOST_PKT	Packet actually for another host which is used in promiscuous mode.

protocol

Protocol types.

This member can be one of the following values (actually supported protocols):

Value	Meaning
IPS_P_IP	IP version 4 packet.
IPS_P_ARP	ARP packet.
IPS_P_ALL	Specific protocol identification which will usually not be defined by the device.

size

Size of the whole data range (**tail** minus **data**).

cur

Index which will be defined by the network stack system. In IP version 4 **cur** will normally point to the protocol header of the lower layer. At transmission the **cur** index always points to the protocol header of the higher protocol. See also **doBeforeSend**.

head

Index which points to the head data.

data

Index which points to the data of the actual protocol layer. The device driver should transmit the data starting from **data** to **tail**.

tail

Index which points to the end of the actual valid data.

end

Index which points to the end of the usable data range. It is not allowed to write beyond this marker.

inlineBuf[]

Start of network data of the network buffer.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

8.5 IPS Network Address Structure ips_addr_t

The IPS network address structure is used to define IPS addresses. The address format was defined to be flexible enough to be used also for IPv6.

```
typedef struct ips_addr_s {
      size_t                    len;
      __u8                      addr[16]
} ips_addr_t;
```

Members

len

Length of the network address.

This member might have one of the following values:

Value	Meaning
0	Address is 0. In UDP/TCP this corresponds to a wildcard (accepts connections to all local network devices).
4	Length of IPv4 address.
6	Length of hardware MAC address
16	Length of IPv6 address.

addr

Binary array which contains the network address.

Header

<install_dir>\sciopta\<version>\include\ips\addr.h

8.6 IPS Network Device Structure ips_dev_t

This structure is used to describe a SDD network device. It extends the standard object descriptor structure and adds specific information about the network device.

```
typedef struct ips_dev_s {
   sdd_obj_t                   object;
   int                         type;
   __u32                       mtu;
   __u32                       mru;
   ips_addr_t                  hwaddr;
   ips_addr_t                  broadcast
} ips_dev_t;
```

Members

object

Specifies the standard SDD object descriptor structure of an SDD object (see chapter **8.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 8-2**).

type

IPS type.

This member can be one of the following values:

Value	Meaning
IPS_DEV_TYPE_NETROM	
IPS_DEV_TYPE_ETHER	
IPS_DEV_TYPE_EETHER	
IPS_DEV_TYPE_AX25	
IPS_DEV_TYPE_PRONET	
IPS_DEV_TYPE_CHAOS	
IPS_DEV_TYPE_IEEE802	
IPS_DEV_TYPE_ARCNET	
IPS_DEV_TYPE_APPLETLK	
IPS_DEV_TYPE_DLCI	
IPS_DEV_TYPE_ATM	
IPS_DEV_TYPE_SLIP	
IPS_DEV_TYPE_CSLIP	
IPS_DEV_TYPE_PPP	
IPS_DEV_TYPE_TUNNEL	
IPS_DEV_TYPE_LOOPBACK	
IPS_DEV_TYPE_IRDA	

mtu

Maximum transmit unit. May not be higher than defined in the device.

mru

Maximum receive unit. May not be higher than defined in the device.

hwaddr

Hardware address (e.g. MAC address).

broadcast

Not yet implemented.

Header

<install_dir>\sciopta\<version>\include\ips\device.msg

8.7 IPV4 ARP Address Structure ipv4_arp_t

Structure of the network ARP entry.

```
typedef struct ipv4_arp_s {
   sc_msgid_t            id;
   sc_errcode_t          error;
   __u8                  ipaddr[4];
   __u8                  hwaddr[6];
} ipv4_arp_t;
```

Members

id

Actually not used.

error

Must be set to zero.

ipaddr

IP address of the ARP entry.

hwaddr

Hardware MAC address of the ARP entry.

Header

<install_dir>\sciopta\<version>\include\ips\arp.msg

8.8 IPV4 Route Structure ipv4_route_t

Structure of the IPS route.

```
typedef struct ipv4_route_s {
   ips_dev_t      device;
   __u8           source[4];
   __u8           netmask[4];
   __u8           router[4];
   __u16          metric;
} ipv4_route_t;
```

Members

device

Specifies the standard SDD object descriptor structure of an SDD network device (see chapter **8.6 “IPS Network Device Structure ips_dev_t” on page 8-9**).

source

Source address (user´s IP address)

netmask

Network mask.

router

Router address to reach the destination if routing is needed. If not set to 0.

metric

Quality of the connection.

Header

<install_dir>\sciopta\<version>\include\ips\router.msg

8.9 NEARPTR and FARPTR

Some 16-bit kernels need near and far pointer defines.

In 32-bit kernels this is just defined as a pointer type (*):

```
#define FARPTR    *
#define NEARPTR   *
```

This mainly to avoid cluttering up sources with #if/#endif.

These target processor specific data types are defined in the file **types.h** located in **sciopta\<cpu>\arch**.

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

This file will be included by the main type file **(types.h** located in **ossys**).

9 IPS Message Interface Reference

9.1 Introduction

The specific SCIOPTA IPS internet protocols messages in addition to the standard SCIOPTA device driver messages are listed.

Please consult chapter **5.6 "Using the IPS Message Interface" on page 5-7** for information how to use the IPS Message Interface.

The messages are listed in alphabetical order. The request and reply message are described together.

Please consult the SCIOPTA device driver, user's guide and reference manual for information about the following standard SCIOPTA device driver messages which are also used in the SCIOPTA IPS internet protocols:

- **SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY**
- **SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY**
- **SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY**
- **SDD_DEV_READ / SDD_DEV_READ_REPLY**
- **SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY**
- **SDD_ERROR**
- **SDD_MAN_ADD / SDD_MAN_ADD_REPLY**
- **SDD_MAN_GET / SDD_MAN_GET_REPLY**
- **SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY**
- **SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY**
- **SDD_MAN_RM / SDD_MAN_RM_REPLY**
- **SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY**
- **SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY**
- **SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY**
- **SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY**
- **SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY**

9.2 IPS_ACCEPT / IPS_ACCEPT_REPLY

This message is used to establish a TCP client-server connection on the server side. Before you can use accept you need to do the following steps:

- Get the TCP SDD protocol descriptor.
- Open the TCP protocol driver described by the above descriptor.
- Bind the TCP protocol driver by sending an **IPS_BIND** message.
- Establish a listen queue by sending an **IPS_LISTEN** message.

The server user process sends an **IPS_ACCEPT** message to the controller process of the TCP protocol driver. The ID of the controller process of the TCP protocol driver is included in the TCP SDD protocol descriptor.

The TCP protocol driver replies with an **IPS_ACCEPT_REPLY** message which includes the accepted access handle. This handle must be used for sending network data.

Message IDs

Request Message	**IPS_ACCEPT**
Reply Message	**IPS_ACCEPT_REPLY**

ips_accept_t Structure

```
typedef struct ips_accept_s {
   ips_connect_t              connect
} ips_accept_t;
```

ips_connect_t Structure

```
typedef struct ips_connect_s {
   sdd_baseMessage_t    base;
   __u16                dstPort;
   ips_addr_t           dstAddr
} ips_connect_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter **8.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 8-1** for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure contains the accepted access handle. This handle must be included in the network buffer message to be sent to the protocol driver.

dstPort

Used in the request message and contains the destination port. Not modified by the protocol driver and therefore contains the same value in the request message.

dstAddr

Used in the request message and contains the destination IP address of type ips_addr_t. Please consult chapter **8.5 "IPS Network Address Structure ips_addr_t" on page 8-8** for type information. Not modified by the protocol driver and therefore contains the same value in the request message.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOMEM	Not enough memory to execute this call.
SC_ENOTSUPP	This request is not supported.

Header

<install_dir>\sciopta\<version>\include\ips\connect.msg

9.3 IPS_ACK

This message is used to acknowledge a network buffer. This message was implemented to allow a flow control in the **IPS** stack. Overrunning will therefore be avoided. This message is only used in the asynchronous mode.

Message ID

Request message	**IPS_ACK**

ips_ack_t Structure

```
typedef struct ips_ack_s {
   sdd_baseMessage_t       base;
   size_t                  size;
} ips_ack_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter **8.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 8-1** for type information.

The **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor.

size

Size of the acknowledged data.

Header

<install_dir>\sciopta\<version>\include\ips\connect.msg

9.4 IPS_BIND / IPS_BIND_REPLY

This message is used to define a specific IP address and port number where the UDP or TCP protocol driver receives network packages.

The client or server user process sends an **IPS_BIND** message to the controller process of the UDP or TCP protocol driver. The ID of the controller process of the protocol driver is included in the SDD protocol descriptor.

The protocol driver replies with an **IPS_BIND_REPLY** message which is only used for error reporting.

Message IDs

Request Message	**IPS_BIND**
Reply Message	**IPS_BIND_REPLY**

ips_bind_t Structure

```
typedef struct ips_bind_s {
   sdd_baseMessage_t     base;
   __u16                 srcPort;
   ips_addr_t            srcAddr
} ips_bind_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter **8.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 8-1** for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

srcPort

Used in the request message and contains the local port to receive network data. Not modified by the protocol driver and therefore contains the same value in the request message.

srcAddr

Used in the request message and contains the local IP address to receive network data of type ips_addr_t. Please consult chapter **8.5 "IPS Network Address Structure ips_addr_t" on page 8-8** for type information. Not modified by the protocol driver and therefore contains the same value in the request message.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOMEM	Not enough memory to execute this call.
SC_ENOTSUPP	This request is not supported.

Header

<install_dir>\sciopta\<version>\include\ips\connect.msg

9.5 IPS_CONNECT / IPS_CONNECT_REPLY

The **IPS_BIND** message only allows specification of a local address. To specify the remote side of an address connection the **IPS_CONNECT** message is used. This is a fully qualified connection as there no wildcard (0) allowed for port and address.

The client or server user process sends an **IPS_CONNECT** message to the controller process of the UDP or TCP protocol driver. The ID of the controller process of the protocol driver is included in the SDD protocol descriptor.

The protocol driver replies with an **IPS_CONNECT_REPLY** message which is only used for error reporting.

Message IDs

Request Message	**IPS_CONNECT**
Reply Message	**IPS_CONNECT_REPLY**

ips_connect_t Structure

```
typedef struct ips_connect_s {
   sdd_baseMessage_t     base;
   __u16                 dstPort;
   ips_addr_t            dtsAddr
} ips_connect_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter 8.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 8-1 for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

dstPort

Used in the request message and contains the remote port of the connection. The wildcard value 0 is not allowed. Not modified by the protocol driver and therefore contains the same value in the request message.

dstAddr

Used in the request message and contains the remote IP address of the connection.of type ips_addr_t. Please consult chapter 8.5 “IPS Network Address Structure ips_addr_t” on page 8-8 for type information. The wildcard value 0 is not allowed. Not modified by the protocol driver and therefore contains the same value in the request message.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOMEM	Not enough memory to execute this call.
SC_ENOTSUPP	This request is not supported.

Header

<install_dir>\sciopta\<version>\include\ips\connect.msg

9.6 IPS_LISTEN / IPS_LISTEN_REPLY

This message is used to activate a listen which is waiting on connection requests of destination peers on the binded source port and IP address.

The server user process sends an **IPS_LISTEN** message to the controller process of the TCP protocol driver. The ID of the controller process of the TCP protocol driver is included in the TCP SDD protocol descriptor.

The TCP protocol driver replies with an **IPS_LISTEN_REPLY** message which is only used for error reporting.

Message IDs

Request Message	**IPS_LISTEN**
Reply Message	**IPS_LISTEN_REPLY**

ips_listen_t Structure

```
typedef struct ips_listen_s {
   sdd_baseMessage_t       base;
   int                     backlog
} ips_listen_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter 8.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 8-1 for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

backlog

Used in the request message and contains the maximum number of connection request by the peer which will be queued . Not modified by the protocol driver and therefore contains the same value in the request message.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure is not valid.
ENOMEM	Not enough memory to execute this call.
EISCON	The socket is already connected.
EADDRINUSE	The local address is already in use.
SC_ENOTSUPP	This request is not supported.

Header

<install_dir>\sciopta\<version>\include\ips\connect.msg

9.7 IPS_ROUTER_ADD / IPS_ROUTER_ADD_REPLY

This message is used to add a route in IPS.

The user sends an **IPS_ROUTER_ADD** message to the controller process of the router. The ID of the controller process of the router is included in the SDD object descriptor of the router.

The router replies with an **IPS_ROUTER_ADD_REPLY** message which is only used for error reporting.

The SDD object descriptor of the router is registered at the SCP_devman manager and must be requested by sending a **SDD_MAN_GET** message to process SCP_devman including the name “router” in the message.

The SDD device descriptor must be requested from a device manager. Network devices are registered in the SCP_netman manager and must be requested by sending a **SDD_MAN_GET** message to process SCP_netman including the name of the network device in the message. The returned SDD network device descriptor can then be copied into the **IPS_ROUTER_ADD** message.

Message IDs

Request Message	**IPS_ROUTER_ADD**
Reply Message	**IPS_ROUTER_ADD_REPLY**

ips_ipv4_route_t Structure

```
typedef struct ips_ipv4_route_s {
   ips_dev_t                device;
   __u8                     source[4];
   __u8                     destination[4];
   __u8                     netmask[4];
   __u8                     router[4];
   __u16                    metric
} ips_ipv4_route_t;
```

Members

device

Specifies the SDD object descriptor structure of an SDD object (here: the IPS network device descriptor). Please consult chapter 8.6 “IPS Network Device Structure ips_dev_t” on page 8-9 for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

The SDD network device descriptor parameters (**device**) are received from the network device manager (SCP_netman).

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

source

Used in the request message and contains the source address (the user's IP address). Not modified by the protocol driver and therefore contains the same value in the request message.

destination

Used in the request message and contains the destination address (point-to-point or subnet address). Not modified by the protocol driver and therefore contains the same value in the request message.

netmask

Used in the request message and contains the network mask. Not modified by the protocol driver and therefore contains the same value in the request message.

router

Used in the request message and contains the router address to reach the destination if routing is needed. If not set to 0. Not modified by the protocol driver and therefore contains the same value in the request message.

metric

Used in the request message and contains information about the quality of the connection. Not modified by the protocol driver and therefore contains the same value in the request message.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EINVAL	The netmask is not valid.
EEXIST	The route already exists.

System Errors

The following system errors can occur during handling of this message. The error will be sent to the kernel and is fatal. The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_devman** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	The process **SCP_ipv4** is not (yet) available.

Header

<install_dir>\sciopta\<version>\include\ips\router.msg

9.8 IPS_ROUTER_RM / IPS_ROUTER_RM_REPLY

Description

This message is used to remove a route in IPS.

The user sends an **IPS_ROUTER_RM** message to the controller process of the router. The ID of the controller process of the router is included in the SDD object descriptor of the router.

The router replies with an **IPS_ROUTER_RM_REPLY** message which is only used for error reporting.

The SDD object descriptor of the router is registered at the SCP_devman manager and must be requested by sending a **SDD_MAN_GET** message to process SCP_devman including the name “router” in the message.

Message IDs

Request Message	**IPS_ROUTER_RM**
Reply Message	**IPS_ROUTER_RM_REPLY**

ips_ipv4_route_t Structure

```
typedef struct ips_ipv4_route_s {
   ips_dev_t              device;
   __u8                   source[4];
   __u8                   destination[4];
   __u8                   netmask[4];
   __u8                   router[4];
   __u16                  metric
} ips_ipv4_route_t;
```

Members

device

Specifies the SDD object descriptor structure of an SDD object (here: the IPS network device descriptor). Please consult chapter 8.6 “IPS Network Device Structure ips_dev_t” on page 8-9 for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

The SDD network device descriptor parameters (**device**) are received from the network device manager (SCP_netman).

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

source

Used in the request message and contains the source address (the user's IP address). Not modified by the protocol driver and therefore contains the same value in the request message.

destination

Not used in the request message and can contain any value. Not modified by the protocol driver and therefore contains the same value in the request message.

netmask

Used in the request message and contains the network mask. Not modified by the protocol driver and therefore contains the same value in the request message.

router

Not used in the request message and can contain any value. Not modified by the protocol driver and therefore contains the same value in the request message.

metric

Used in the request message and contains information about the quality of the connection. Not modified by the protocol driver and therefore contains the same value in the request message.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EINVAL	The netmask is not valid.
EEXIST	The route already exists.

System Errors

The following system errors can occur during handling of this message. The error will be sent to the kernel and is fatal. The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_devman** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	The process **SCP_ipv4** is not (yet) available.

Header

<install_dir>\sciopta\<version>\include\ips\router.msg

9.9 IPS_SET_OPTION / IPS_SET_OPTION_REPLY

This message is used to set options associated with a protocol driver.

When setting options the level at which the option resides and the name of the option must be specified. To set options at generic level, it is specified as **SOL_SOCKET**.

The user sends an **IPS_SET_OPTION** message to the controller process of the protocol driver. The ID of the controller process of the protocol driver is included in the SDD protocol descriptor.

The protocol driver replies with an **IPS_SET_OPTION_REPLY** message which is only used for error reporting.

Message IDs

Request Message	**IPS_SET_OPTION**
Reply Message	**IPS_SET_OPTION_REPLY**

ips_listen_t Structure

```
typedef struct ips_set_option_s {
   sdd_baseMessage_t        base;
   int                      level;
   int                      optname;
   int                      optlen;
   char                     optval[1];
} ips_set_option_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter **8.1 "Base SDD Object Descriptor Structure sdd_baseMessage_t" on page 8-1** for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

level

This member can be one of the following values:

Value	Meaning
SOL_SOCKET	Generic settings. Defined option names: **SO_SC_ASYNC** **SO_SC_SNDACK**
SOL_TCP	TCP specific settings. Defined option names: **TCP_NODELAY**
SOL_UDP	UDP specific settings.

optname

This member can be one of the following values:

Value	Meaning
SO_SC_ASYNC	Asynchronous package delivery (message queue). The value (**optval**) can be **ON (1)** or **OFF (0)**.
SO_SC_SND_ACK	All sent packages are returned to the sender as **IPS_SEND_REPLY** message. The value (**optval**) can be **ON (1)** or **OFF (0)**.
TCP_NODELAY	Nagle algorithm switching. disabled. The value (**optval**) can be **disable (1)** or **enable (0)**.

optlen

size of the option value (optval) in bytes.

optval

Option value.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure is not valid
EPERM	No permission to set this option.
ENOPROTOOPT	The option for this level is not supported.
SC_ENOTSUPP	This request is not supported.

Header

<install_dir>\sciopta\<version>\include\ips\connect.msg

9.10 SDD_MAN_ADD / SDD_MAN_ADD_REPLY

This message is used to add a new network device in the device driver system. It extends the standard **SDD_MAN_ADD** message of type sdd_manAdd_t (see SCIOPTA - Device Driver, User´s Guide and Reference Manual) with this message of type ips_dev_t.

The network device driver sends a **SDD_MAN_ADD** message (which is the network device descriptor) to the network device manager SCP_netman.

The network device manager replies with an **SDD_MAN_ADD_REPLY** message which is only used for error reporting.

Message IDs

Request Message	**IPS_MAN_ADD**
Reply Message	**IPS_MAN_ADD_REPLY**

ips_dev_t Structure

```
typedef struct ips_dev_s {
   sdd_obj_t                    object;
   int                          type;
   __u32                        mtu;
   __u32                        mru;
   ips_addr_t                   hwaddr;
   ips_addr_t                   broadcast
} ips_dev_t;
```

Members

object

Specifies the standard SDD object descriptor structure of an SDD object. Please consult chapter **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** for type information.

type

Used in the request message and contains the IPS type. Not modified by the network device manager and therefore contains the same value in the reply message.

This member can be one of the following values:

Value	**Meaning**
IPS_DEV_TYPE_NETROM	
IPS_DEV_TYPE_ETHER	
IPS_DEV_TYPE_EETHER	
IPS_DEV_TYPE_AX25	
IPS_DEV_TYPE_PRONET	
IPS_DEV_TYPE_CHAOS	
IPS_DEV_TYPE_IEEE802	
IPS_DEV_TYPE_ARCNET	
IPS_DEV_TYPE_APPLETLK	

IPS_DEV_TYPE_DLCI

IPS_DEV_TYPE_ATM

IPS_DEV_TYPE_SLIP

IPS_DEV_TYPE_CSLIP

IPS_DEV_TYPE_PPP

IPS_DEV_TYPE_TUNNEL

IPS_DEV_TYPE_LOOPBACK

IPS_DEV_TYPE_IRDA

mtu

Used in the request message and contains the maximum transmit unit. May not be higher than defined in the device. Not modified by the network device manager and therefore contains the same value in the reply message.

mru

Used in the request message and contains the maximum receive unit. May not be higher than defined in the device. Not modified by the network device manager and therefore contains the same value in the reply message.

hwaddr

Used in the request message and contains the hardware address (e.g. MAC address). May not be higher than defined in the device. Not modified by the network device manager and therefore contains the same value in the reply message.

broadcast

Not yet implemented.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EEXIST	The route already exists.

Header

<install_dir>\sciopta\<version>\include\ips\device.msg

9.11 SDD_MAN_GET / SDD_MAN_GET_REPLY

This message is used to get the SDD network device descriptor of a registered network device. It extends the standard **SDD_MAN_GET** message of type sdd_manGet_t (see SCIOPTA - BSP and Device Driver, User´s Guide and Reference Manual) with this message of type ips_dev_t.

The user sends a **SDD_MAN_GET** message to the network device manager SCP_netman.

The network device manager replies with an **SDD_MAN_GET_REPLY** message which includes the SDD network device descriptor of the registered network device.

Message IDs

Request Message	**IPS_MAN_GET**
Reply Message	**IPS_MAN_GET_REPLY**

ips_dev_t Structure

```
typedef struct ips_dev_s {
   sdd_obj_t                   object;
   int                         type;
   __u32                       mtu;
   __u32                       mru;
   ips_addr_t                  hwaddr;
   ips_addr_t                  broadcast
} ips_dev_t;
```

Members

object

Specifies the standard SDD object descriptor structure of an SDD object. Please consult chapter **8.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 8-2** for type information.

type

Used in the reply message and contains the IPS type. Not used in the request message.

This member can be one of the following values:

Value	**Meaning**
IPS_DEV_TYPE_NETROM	
IPS_DEV_TYPE_ETHER	
IPS_DEV_TYPE_EETHER	
IPS_DEV_TYPE_AX25	
IPS_DEV_TYPE_PRONET	
IPS_DEV_TYPE_CHAOS	
IPS_DEV_TYPE_IEEE802	
IPS_DEV_TYPE_ARCNET	
IPS_DEV_TYPE_APPLETLK	
IPS_DEV_TYPE_DLCI	
IPS_DEV_TYPE_ATM	

IPS_DEV_TYPE_SLIP

IPS_DEV_TYPE_CSLIP

IPS_DEV_TYPE_PPP

IPS_DEV_TYPE_TUNNEL

IPS_DEV_TYPE_LOOPBACK

IPS_DEV_TYPE_IRDA

mtu

Used in the reply message and contains the maximum transmit unit. May not be higher than defined in the device. Not used in the request message.

mru

Used in the reply message and contains the maximum receive unit. May not be higher than defined in the device. Not used in the request message.

hwaddr

Used in the reply message and contains the hardware address (e.g. MAC address). May not be higher than defined in the device. Not used in the request message.

broadcast

Not yet implemented.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
ENOENT	Device not found.

Header

<install_dir>\sciopta\<version>\include\ips\device.msg

9.12 SDD_MAN_RM / SDD_MAN_RM_REPLY

This message is used to remove a registered network device in the device driver system. It extends the standard **SDD_MAN_RM** message of type sdd_manRm_t (see SCIOPTA - BSP and Device Driver, User´s Guide and Reference Manual) with this message of type ips_dev_t.

The user sends a **SDD_MAN_RM** message to the network device manager SCP_netman.

The network device manager replies with an **SDD_MAN_RM_REPLY** message which is only used for error reporting.

Message IDs

Request Message	**IPS_MAN_RM**
Reply Message	**IPS_MAN_RM_REPLY**

ips_dev_t Structure

```
typedef struct ips_dev_s {
   sdd_obj_t                  object;
   int                        type;
   __u32                      mtu;
   __u32                      mru;
   ips_addr_t                 hwaddr;
   ips_addr_t                 broadcast
} ips_dev_t;
```

Members

object

Specifies the standard SDD object descriptor structure of an SDD object. Please consult chapter **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** for type information.

type

Not used.

mtu

Not used.

mru

Not used.

hwaddr

Not used.

broadcast

Not yet implemented.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. In the request message **error** must be set to zero.

Value of error	Meaning
EEXIST	The network device is not registered.

Header

<install_dir>\sciopta\<version>\include\ips\device.msg

9.13 SDD_NET_CLOSE / SDD_NET_CLOSE_REPLY

This message is used to close a network protocol driver.

The user process sends a **SDD_NET_CLOSE** message to the network protocol driver process. The network protocol driver process (and the network device driver handle) must be retrieved from the SDD network device descriptor.

The protocol driver process must reply with the **SDD_NET_CLOSE_REPLY** reply message.

Message IDs

Request Message	**SDD_NET_CLOSE**
Reply Message	**SDD_NET_CLOSE_REPLY**

ips_listen_t Structure

```
typedef struct sdd_netClose_s {
    sdd_baseMessage_t       base;
} sdd_netClose_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter **8.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 8-1** for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

Errors

A device driver dependent error can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. If the network protocol driver was successfully closed, **error** contains the value of zero. In the request message **error** must be set to zero.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

9.14 SDD_NET_OPEN / SDD_NET_OPEN_REPLY

This message is used to open a network protocol driver.

The user process sends a **SDD_NET_OPEN** message to the network protocol driver process. The network protocol driver process (and the network device driver handle) must be retrieved from the SDD network device descriptor.

The protocol driver process must reply with the **SDD_NET_OPEN_REPLY** reply message.

Message IDs

Request Message	**SDD_NET_OPEN**
Reply Message	**SDD_NET_OPEN_REPLY**

ips_listen_t Structure

```
typedef struct sdd_netOpen_s {
   sdd_baseMessage_t         base;
} sdd_netOpen_t;
```

Members

base

Specifies the base SDD object descriptor structure of an SDD object (here: the SDD protocol descriptor). Please consult chapter **8.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 8-1** for type information.

In the request message the **handle** member of the **sdd_baseMessage_t** structure contains the access handle which is included in the SDD protocol descriptor. The SDD TCP protocol descriptor was previously returned in the **MAN_GET_REPLY** message of the IPv4 protocol driver.

In the reply message the **handle** member of the **sdd_baseMessage_t** structure is not modified by the protocol driver.

Errors

A device driver dependent error can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. If the network protocol driver was successfully opened, **error** contains the value of zero. In the request message **error** must be set to zero.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

9.15 SDD_NET_RECEIVE / SDD_NET_RECEIVE_REPLY

This message is used to receive network data from a network device driver.

The network device driver receiver process sends a **SDD_NET_RECEIVE** message to the IPv4 process of IPS. The IPv4 process can reply with the **SDD_NET_RECEIVE_REPLY** reply message but this is normally not used.

The user process waits for an **SDD_NET_RECEIVE** message by calling an sc_msgRx() system call.

The macro **SDD_NET_DATA** (netbuf) is available to simplify the network buffer accesses. The macro is defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h.

Message IDs

Request Message	**SDD_NET_RECEIVE**
Reply Message	**SDD_NET_RECEIVE_REPLY**

sdd_netReceive_t Structure

```
typedef struct sdd_netReceive_s {
   struct sdd_netBuffer_s       buffer
} sdd_netReceive_t;
```

Members

buffer

Specifies the network buffer. Please consult chapter **8.4 "SDD Network Buffer Structure sdd_netbuf_t" on page 8-6** for type information.

Errors

A device driver dependent error can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. If the network data was successfully received, **error** contains the value of zero. In the request message **error** must be set to zero.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

9.16 SDD_NET_RECEIVE_URGENT / SDD_NET_RECEIVE_URGENT_REPLY

Description

This message is used to receive very urgent network data from a network device driver. A user process which expects to receive urgent data from a device driver should also always test the **SDD_NET_RECEIVE_URGENT** message when receiving messages from IPS.

The device driver sender process sends a **SDD_NET_RECEIVE_URGENT** message to the user process. The user process can reply with the **SDD_NET_RECEIVE_URGENT_REPLY** reply message but this is normally not used.

Message IDs

Request Message	**SDD_NET_RECEIVE_URGENT**
Reply Message	**SDD_NET_RECEIVE_URGENT_REPLY**

sdd_netReceive_t Structure

```
typedef struct sdd_netReceive_s {
   struct sdd_netBuffer_s      buffer
} sdd_netReceive_t;
```

Members

buffer

Specifies the network buffer. Please consult chapter **8.4 “SDD Network Buffer Structure sdd_netbuf_t” on page 8-6** for type information.

Errors

A device driver dependent error can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. If the network data was successfully received, **error** contains the value of zero. In the request message **error** must be set to zero.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

9.17 SDD_NET_SEND / SDD_NET_SEND_REPLY

This message is used to send network data to a network device driver.

The user process sends a **SDD_NET_SEND** message to the network device driver sender process. The network device driver sender process (and the network device driver handle) must be retrieved from the SDD network device descriptor.

The device driver sender can reply with the **SDD_NET_SEND_REPLY** reply message but this is normally not used.

The macro **SDD_NET_DATA** (netbuf) is available to simplify the network buffer accesses. The macro is defined in the file <install_dir>\sciopta\<version>\include\sdd\sdd.h.

Message IDs

Request Message	**SDD_NET_SEND**
Reply Message	**SDD_NET_SEND_REPLY**

sdd_netReceive_t Structure

```
typedef struct sdd_netSend_s {
   struct sdd_netBuffer_s       buffer
} sdd_netSend_t;
```

Members

buffer

Specifies the network buffer. Please consult chapter **8.4 “SDD Network Buffer Structure sdd_netbuf_t” on page 8-6** for type information.

Errors

A device driver dependent error can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. If the network data was successfully sent, **error** contains the value of zero. In the request message **error** must be set to zero.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

9.18 SDD_OBJ_INFO / SDD_OBJ_INFO_REPLY

This message is used to get information from a network device driver. It extends the standard **SDD_OBJ_INFO** message of type sdd_objInfo_t (see SCIOPTA - Device Driver, User´s Guide and Reference Manual) with this message of type ips_info_t.

The user process sends a **SDD_OBJ_INFO** message to the network device driver controller process. The network device driver controller process (and the network device driver handle) must be retrieved from the SDD network device descriptor.

The network device driver controller process replies with an **SDD_OBJ_INFO_REPLY** message which includes the requested information.

To get meaningful values a connection must exist (bind, connect, listen and accept).

Message IDs

Request Message	**SDD_OBJ_INFO**
Reply Message	**SDD_OBJ_INFO_REPLY**

ips_dev_t Structure

```
typedef struct ips_info_s {
   sdd_objInfo_t             objectInfo;
   int                       family;
   __u16                     srcPort;
   __u16                     dstPort;
   ips_addr_t                srcAddr;
   ips_addr_t                dstAddr
} ips_info_t;
```

Members

objectInfo

Specifies the SDD object info structure of an SDD object. Please consult chapter **8.3 "Base SDD Object Info Structure sdd_objInfo_t" on page 8-5** for type information.

family

Used in the reply message and contains the socket address family. Not used in the request message.

This member can be one of the following values:

Value	**Meaning**
AF_INET	

srcPort

Used in the reply message and contains the source port of the connection. Not used in the request message.

dstPort

Used in the reply message and contains the destination port of the connection. Not used in the request message.

srcAddr

Used in the reply message and contains the source address of the connection.Please consult chapter **8.5 “IPS Network Address Structure ips_addr_t” on page 8-8** for type information. Not used in the request message.

dstAddr

Used in the reply message and contains the source address of the connection. Please consult chapter **8.5 “IPS Network Address Structure ips_addr_t” on page 8-8** for type information. Not used in the request message.

Errors

A device driver dependent error can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure and is used in the reply message. If the network data was successfully sent, **error** contains the value of zero. In the request message **error** must be set to zero.

Header

<install_dir>\sciopta\<version>\include\sdd\sdd.msg

10 IPS Function Interface Reference

10.1 Introduction

Only the specific SCIOPTA IPS Internet Protocol functions in addition to the standard SCIOPTA device driver functions are listed.

The functions are listed in alphabetical order.

Please consult the SCIOPTA device driver, user's guide and reference manual for information about the following standard SCIOPTA device driver functions which are also used in the SCIOPTA IPS internet protocols:

- **sdd_devClose**
- **sdd_devIoctl**
- **sdd_devOpen**
- **sdd_devRead**
- **sdd_devWrite**
- **sdd_manAdd**
- **sdd_manGetByName**
- **sdd_manGetByPath**
- **sdd_manGetFirst**
- **sdd_manGetNext**
- **sdd_manGetNoOfItems**
- **sdd_manGetRoot**
- **sdd_manRm**
- **sdd_objDup**
- **sdd_objFree**
- **sdd_objResolve**
- **sdd_objSizeGet**
- **sdd_objTimeGet**
- **sdd_objTimeSet**

10.2 ips_accept

This function is used to establish a TCP client-server connection on the server side. Before you can use ips_accept() you need to do the following steps:

- Get the TCP SDD protocol descriptor and open the TCP protocol driver described by the descriptor and get the access handle by using the **ips_open** function.
- Bind the access handle by using an **ips_bind** function.
- Establish a listen queue by using an **ips_listen** function.

The **ips_accept** function extracts the first connection request on the queue of pending connections, creates a new SDD protocol descriptor with the same properties of the SDD protocol descriptor **self**, and allocates and returns a new SDD protocol descriptor including the new access handle. The new connection does not remain in the listening state. The listening state of the original handle will not be modified.

If there are no pending connections present on the queue **ips_accept** blocks the caller until a connection is present. The peer address of the connection will be returned in dstAddr and dstPort.

```
sdd_obj_t NEARPTR ips_accept (
   sdd_obj_t NEARPTR    self,
   ips_addr_t           *dstAddr,
   __u16                dstPort
);
```

Parameter

self

SDD protocol descriptor. Please consult chapters 8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2 and 8.9 “NEARPTR and FARPTR” on page 8-13 for type information.

dstAddr

When **ips_accept** returns, dstAddr contains the pointer to the destination address of the accepted connection. Please consult chapter 8.5 “IPS Network Address Structure ips_addr_t” on page 8-8 for type information.

dstPort

Destination port.

Return Value

If the functions succeeds the return value is the pointer to the SDD protocol descriptor which includes a new accepted handle. Please consult chapters 8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2 and 8.9 “NEARPTR and FARPTR” on page 8-13 for type information.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
EINVAL	The parameter **self** is NULL.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to execute this call.
EOPNOTSUPP	The ips_accept operation is not supported by this access handle.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  ips_accept_t accept;
};
sdd_obj_t NEARPTR
ips_accept (sdd_obj_t NEARPTR self, ips_addr_t * dstAddr, __u16 * dstPort)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPS_ACCEPT_REPLY, 0
  };
  sc_msg_t msg;
  sdd_obj_t NEARPTR b;
  if (!self || !self->controller) {
    sc_miscErrnoSet (ENOTSOCK);
    return NULL_OBJ;
  }
  /* cause, an accept just have a changed handle and the same pids like
     self!! */
  b = (sdd_obj_t NEARPTR)
    sc_msgAlloc (sc_msgSizeGet ((sc_msgptr_t) &self), 0,
                 sc_msgPoolIdGet ((sc_msgptr_t) &self),
                 SC_FATAL_IF_TMO);
  memcpy (b, self, sc_msgSizeGet ((sc_msgptr_t) &self));
  msg =
    sc_msgAlloc (sizeof (ips_accept_t), IPS_ACCEPT, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  msg->accept.connect.base.error = 0;
  msg->accept.connect.base.handle = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->id == SDD_ERROR) {
    sc_miscErrnoSet (msg->accept.connect.base.error);
    sc_msgFree (&msg);
    sc_msgFree ((sc_msgptr_t) &b);
    return NULL_OBJ;
  }
  else if (msg->accept.connect.base.error) {
    b->base.handle = msg->accept.connect.base.handle;
    ips_close (&b);
    sc_miscErrnoSet (msg->accept.connect.base.error);
    sc_msgFree (&msg);
    return NULL_OBJ;
  }
  else {
    b->base.handle = msg->accept.connect.base.handle;
    memcpy (dstAddr, &msg->accept.connect.dstAddr, sizeof (ips_addr_t));
    *dstPort = msg->accept.connect.dstPort;
    sc_msgFree (&msg);
    return b;
  }
}
```

10.3 ips_ack

This function is used to acknowledge a netbuffer.

This is used to implement a flow control to avoid overrunning the client message queue.

```
void ips_ack(
   sdd_obj_tNEARPTR     self,
   sdd_netbuf_t NEARPTR netbuf
);
```

Parameter

self

SDD protocol descriptor which includes the access handle. Please consult chapters **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

netbuf

Network buffer of the incoming netbuf message of type sdd_netbuf_t. Please consult chapters **8.4 “SDD Network Buffer Structure sdd_netbuf_t” on page 8-6** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

Return Value

None.

Errors

None.

Include Files

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  ips_ack_t ack;
};
void
ips_ack (sdd_obj_t NEARPTR self, sdd_netbuf_t NEARPTR netbuf)
{
  sc_msg_t msg;
  if (!netbuf || !self || !self->controller) {
    return;
  }
  msg =
    sc_msgAlloc (sizeof (ips_ack_t), IPS_ACK, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  msg->ack.base.error = 0;
  msg->ack.base.handle = self->base.handle;
  msg->ack.size = SDD_NET_DATA_SIZE (netbuf);
  sc_msgTx (&msg, self->controller, 0);
}
```

10.4 ips_alloc

This function is used to allocate a netbuffer.

```
sdd_netbuf_t NEARPTR ips_alloc(
   size_t           size,
   sc_poolid_t      plid,
   sc_ticks_t       tmo);
```

Parameter

size

The requested size of the network buffer.

id

Message ID which will be placed at the beginning of the data buffer of the message.

plid

Pool ID from where the netbuffer will be allocated.

This parameter can be one of the following values:

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

tmo

Allocation timing parameter:

This parameter can be one of the following values:

Value	Meaning
SC_ENDLESS_TMO	Time-out is not used. Blocks and waits endless until a buffer is available from the message pool.
SC_NO_TMO	A NIL pointer will be returned if there is memory shortage in the message pool.
SC_FATAL_IF_TMO	A (fatal) kernel error will be generated if a message buffer of the requested size is not available.
0 < tmo < SC_TMO_MAX	Time-out value in system ticks. Alloc with time-out. Blocks and waits the specified number of ticks to get a message buffer.

Return Value

The return value depends on the used **tmo** parameter when **sc_msgAlloc** was called.

If the **tmo** parameter was SC_ENDLESS_TMO the pointer to the allocated netbuffer is returned.

If the **tmo** parameter was SC_NO_TMO and if a buffer of the requested size is available the pointer to the allocated netbuffer is returned.

If the **tmo** parameter was SC_NO_TMO and if a buffer of the requested size is **not** available a NIL pointer is returned.

If the **tmo** parameter was a positive value (>0) and the system responds within the time-out period the pointer to the allocated netbuffer is returned.

If the **tmo** parameter was a positive value (>0) and the system responds **not** within the time-out period (time-out expired) a NIL pointer is returned.

Please consult chapters **8.4 "SDD Network Buffer Structure sdd_netbuf_t" on page 8-6** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information.

Errors

None.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
sdd_netbuf_t NEARPTR ips_alloc (size_t size, sc_poolid_t plid, sc_ticks_t tmo)
{
  ssize_t diff = 512 - size;
  if (diff > 0) {
    return sdd_netbufAlloc (68, size, diff, plid, tmo);
  }
  else {
    return sdd_netbufAlloc (68, size, 0, plid, tmo);
  }
}
```

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
};
sdd_netbuf_t NEARPTR
sdd_netbufAlloc (size_t head, size_t data, size_t tail,
            sc_poolid_t plid, sc_ticks_t tmo)
{
  sdd_netbuf_t NEARPTR tmp = NULL_NETBUF;
  PRINTF ("sdd_netbufAlloc\n");
  tmp = (sdd_netbuf_t NEARPTR) sc_msgAlloc(sizeof (sdd_netbuf_t)+head+data+tail,
                      SDD_NETBUF,
                      plid,
                      tmo);
  if (tmp) {
#ifdef SC_DEBUG
    memset (tmp, 0xe, sizeof (sdd_netbuf_t) + head + data + tail);
#endif
    tmp->base.error = 0;
    tmp->base.handle = NULL_HANDLE;
    tmp->doBeforeSend = NULL;
    tmp->returnTo = SC_ILLEGAL_PID;
    tmp->pkttype = SDD_HOST_PKT;
    tmp->head = 0;
    tmp->cur = tmp->data = tmp->head + head;
    tmp->tail = tmp->data + data;
    tmp->end = tmp->tail + tail;
  }
  return tmp;
}
```

10.5 ips_bind

This function is used to define a specific IP address and port number where the UDP or TCP protocol driver receives network packages.

The bind call gives the handle in the SDD protocol descriptor **self** the local address **srcAddr**.

Before a **TCP** handle is put into the **LISTEN** state to receive connections, you usually need to first assign a local address using **ips_bind** to make the handle visible.

```
int ips_bind (
   sdd_obj_t NEARPTR  self,
   ips_addr_t         *srcAddr,
   __u16              srcPort
);
```

Parameter

self

SDD protocol descriptor. Please consult chapters **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

srcAddr

Source address. Please consult chapter **8.5 “IPS Network Address Structure ips_addr_t” on page 8-8** for type information.

srcPort

Source port.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EINVAL	The parameter **self** is NULL.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to execute this call.
EOPNOTSUPP	The ips_bind operation is not supported by this access handle.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  ips_bind_t bind;
};
int
ips_bind (sdd_obj_t NEARPTR self, ips_addr_t * srcAddr, __u16 srcPort)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPS_BIND_REPLY, 0
  };
  sc_msg_t msg;
  int error = 0;
  if (!self || !self->controller) {
    sc_miscErrnoSet (ENOTSOCK);
    return -1;
  }
  msg =
    sc_msgAlloc (sizeof (ips_bind_t), IPS_BIND, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  msg->bind.base.error = 0;
  msg->bind.base.handle = self->base.handle;
  if (srcAddr) {
    memcpy (&msg->bind.srcAddr, srcAddr, sizeof (ips_addr_t));
  }
  else {
    memset (&msg->bind.srcAddr, 0, sizeof (ips_addr_t));
  }
  msg->bind.srcPort = srcPort;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  error = (int)msg->bind.base.error;
  sc_msgFree (&msg);
  if (error) {
    sc_miscErrnoSet (error);
    return -1;
  }
  return 0;
}
```

10.6 ips_close

This function is used to close a protocol driver and to close an open connection.

```
int ips_close (
   sdd_obj_t NEARPTR NEARPTR    self
);
```

Parameter

self

SDD protocol descriptor which includes the access handle. This parameter will be set to zero after returning and the SDD protocol descriptor (message) will be freed. Please consult chapters **8.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 8-2** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information. Please note the **pointer to a pointer** type.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EINVAL	The parameter **self** is NULL.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to execute this call.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
};
int
ips_close (sdd_obj_t NEARPTR NEARPTR self)
{
  if (self && *self) {
    sdd_netClose ((*self));
    sdd_objFree (self);
    return 0;
  }
  else {
    sc_miscErrnoSet (EINVAL);
    return -1;
  }
}

union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  sdd_netClose_t close;
};
int sdd_netClose (sdd_obj_t NEARPTR self)
{
  sc_msg_t msg, msg2;
  static const sc_msgid_t closeRpl[2] = {
    SDD_NET_CLOSE_REPLY, 0
  };
  static const sc_msgid_t errorRpl[2] = {
    SDD_ERROR, 0
  };
  msg = sc_msgAlloc (sizeof (sdd_netClose_t), SDD_NET_CLOSE,
           sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->close.base.error = 0;
  msg->close.base.handle = self->base.handle;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) closeRpl, SC_MSGRX_MSGID);
  msg2 = sc_msgRx (SC_NO_TMO, (void *) errorRpl, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    if (msg2 && msg2->base.handle == self->base.handle) {
      sc_msgFree (&msg2);
    }
    else if (msg2) {
      sc_msgTx (&msg2, SC_CURRENT_PID, 0);
    }
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

10.7 ips_connect

This function is used to initiate a connection on a access handle.

The SDD protocol descriptor **self** includes the access handle. If it is a UDP access handle the address **dstAddr** will be used as default for receiving and transmitting. If it is a TCP access handle this call attempts to make a connection to a peer.

Generally, TCP access handles may successfully connect only once; UDP access handles may use connect multiple times to change their association.

```
int ips_connect (
   sdd_obj_t NEARPTR    self,
   ips_addr_t           *dstAddr,
   __u16                dstPort
);
```

Parameter

self

SDD protocol descriptor. Please consult chapters **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

dstAddr

Destination address. Please consult chapter **8.5 “IPS Network Address Structure ips_addr_t” on page 8-8** for type information.

dstPort

Destination port.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EINVAL	The parameter **self** is NULL.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to execute this call.
EISCON	The socket is already connected.
ECONNREFUSED	The connection has been refused by the peer.
ETIMEDOUT	A time out occurred at connection.
ENETUNREACH	The network cannot be reached.
EADDRINUSE	The local address is already in use.
EOPNOTSUPP	The ips_connect operation is not supported by this access handle.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  ips_connect_t connect;
};
int
ips_connect (sdd_obj_t NEARPTR self, ips_addr_t * dstAddr, __u16 dstPort)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPS_CONNECT_REPLY, 0
  };
  sc_msg_t msg;
  int error = 0;
  if (!self || !self->controller) {
    sc_miscErrnoSet (ENOTSOCK);
    return -1;
  }
  msg =
    sc_msgAlloc (sizeof (ips_connect_t), IPS_CONNECT, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  msg->connect.base.error = 0;
  msg->connect.base.handle = self->base.handle;
  if (dstAddr) {
    memcpy (&msg->connect.dstAddr, dstAddr, sizeof (ips_addr_t));
  }
  else {
    memset (&msg->connect.dstAddr, 0, sizeof (ips_addr_t));
  }
  msg->connect.dstPort = dstPort;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  error = (int)msg->connect.base.error;
  sc_msgFree (&msg);
  if (error) {
    sc_miscErrnoSet (error);
    return -1;
  }
  return 0;
}
```

10.8 ips_devGetByHwAddr

This function is used to get an SDD network device descriptor for the given hardware MAC address.

```
ips_dev_t NEARPTR ips_devGetByHwAddr (
   ips_addr_t              *addr
);
```

Parameter

addr

MAC address. Please consult chapter **8.5 “IPS Network Address Structure ips_addr_t” on page 8-8** for type information.

Return Value

If the functions succeeds the return value is the pointer to the SDD network device descriptor. Please consult chapters **8.6 “IPS Network Device Structure ips_dev_t” on page 8-9** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
ENOENT	Device not found.

System Errors

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_netman** which normally should reside in the system module does not exist.

Header

<install_dir>\sciopta\<version>\include\ips\device.h

10.9 ips_devGetByName

This function is used to get an SDD network device descriptor for the given name.

```
ips_dev_t NEARPTR ips_devGetByName(
   const char              *name
);
```

Parameter

name

Name of the network device.

Return Value

If the functions succeeds the return value is the pointer to the SDD network device descriptor. Please consult chapters **8.6 "IPS Network Device Structure ips_dev_t" on page 8-9** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
ENOENT	Device not found.

System Errors

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_netman** which normally should reside in the system module does not exist.

Header

<install_dir>\sciopta\<version>\include\ips\device.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  ips_dev_t device;
  sdd_obj_t object;
};
ips_dev_t NEARPTR
ips_devGetByName (const char *name)
{
  sdd_obj_t NEARPTR netman;
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, SDD_MAN_GET_REPLY, 0
  };
  netman = sdd_manGetRoot ("/SCP_netman", "netman", SC_DEFAULT_POOL,
                           SC_ENDLESS_TMO);
  if (!netman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_netman");
  }
  msg =
    sc_msgAlloc (sizeof (ips_dev_t), SDD_MAN_GET, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  msg->object.base.error = 0;
  msg->object.manager = netman->base.handle;
  strncpy (msg->object.name, name, SC_NAME_MAX);
  sc_msgTx (&msg, netman->controller, 0);
  sc_msgFree ((sc_msgptr_t) &netman);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->object.base.error) {
    sc_miscErrnoSet (msg->object.base.error);
    sc_msgFree (&msg);
    return NULL;
  }
  else {
    return &msg->device;
  }
}
```

10.10 ips_devRegister

This function is used to register an SDD network device descriptor at the network device manager process **SCP_netman**.

This function call is used by network device drivers.

```
int ips_devRegister(
   ips_dev_t NEARPTR NEARPTRdev
);
```

Parameter

dev

SDD network device descriptor to register. Please consult chapters **8.6 "IPS Network Device Structure ips_dev_t" on page 8-9** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information. Please note the **pointer to a pointer** type.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
EEXIST	Device already exists.

System Error

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_netman** which normally should reside in the system module does not exist.

Include Files

<install_dir>\sciopta\<version>\include\ips\device.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  ips_dev_t device;
  sdd_obj_t object;
};
/** Interface implementations
*/
int
ips_devRegister (ips_dev_t NEARPTR NEARPTR dev)
{
  sdd_obj_t NEARPTR netman;
  int ret;
  netman = sdd_manGetRoot ("/SCP_netman", "netman", SC_DEFAULT_POOL,
                           SC_ENDLESS_TMO);
  if (!netman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_netman");
  }
  ret = sdd_manAdd (netman, (sdd_obj_t NEARPTR NEARPTR) dev);
  sc_msgFree ((sc_msgptr_t) &netman);
  return ret;
}
```

10.11 ips_devUnregister

This function is used to remove an SDD network device descriptor from the network device manager process **SCP_netman**.

This function call is used by device drivers.

```
int ips_devUnregister(
   ips_dev_tNEARPTR      dev
);
```

Parameter

dev

SDD network device descriptor to remove. Please consult chapters **8.6 "IPS Network Device Structure ips_dev_t" on page 8-9** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
ENOENT	Device not found.

System Error

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_netman** which normally should reside in the system module does not exist.

Include Files

<install_dir>\sciopta\<version>\include\ips\device.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  ips_dev_t device;
  sdd_obj_t object;
};
int
ips_devUnregister (ips_dev_t NEARPTR dev)
{
  sdd_obj_t NEARPTR netman;
  int ret;
  netman = sdd_manGetRoot ("/SCP_netman", "netman", SC_DEFAULT_POOL,
                           SC_ENDLESS_TMO);
  if (!netman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_netman");
  }
  ret = sdd_manRm (netman, (sdd_obj_t NEARPTR) dev, sizeof (ips_dev_t));
  sc_msgFree ((sc_msgptr_t) &netman);
  return ret;
}
```

10.12 ips_getOption

This function is used to get the options associated with a access handle of a SDD protocol descriptor.

```
int ips_getOption(
   sdd_obj_t NEARPTR  self,
   int                level,
   int                optname,
   void               *optval,
   socklen_t          *optlen
);
```

Parameter

self

SDD protocol descriptor. Please consult chapters **8.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 8-2** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information.

level

This parameter can be one of the following values:

Value	Meaning
SOL_SOCKET	Generic settings. Defined option names: **SO_SC_ASYNC** **SO_SC_RET_ACK** **SO_SC_RET_BUF**
SOL_TCP	TCP specific settings. Defined option names: **TCP_NODELAY**
SOL_UDP	UDP specific settings.

optname

This parameter can be one of the following values:

Value	Meaning
SO_SC_ASYNC	Asynchronous package delivery (message queue). The value (**optval**) can be **ON (1)** or **OFF (0)**.
SO_SC_RET_ACK	On every sent package an **IPS_ACK** message will be returned. The value (**optval**) can be **ON (1)** or **OFF (0)**.
SO_SC_RET_BUF	All sent packages are returned to the sender as **IPS_SEND_REPLY** message. The value (**optval**) can be **ON (1)** or **OFF (0)**.
TCP_NODELAY	Nagle algorithm switching. disabled. The value (**optval**) can be **disable (1)** or **enable (0)**.

optval

Pointer to the option value.

optlen

Pointer to the size of the option value (optval) in bytes.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EOPNOTSUPP	The ips_getOption operation is not supported by this access handle.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  ips_option_t option;
};
int
ips_getOption (sdd_obj_t NEARPTR self, int level, int optname, void *optval,
           size_t * optlen)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPS_GET_OPTION_REPLY, 0
  };
  if (!self || !self->controller) {
    sc_miscErrnoSet (ENOTSOCK);
    return -1;
  }
  if (!optlen || !*optlen) {
    sc_miscErrnoSet (EINVAL);
    return -1;
  }
  msg = sc_msgAllocClr (sizeof (ips_option_t) + (*optlen), IPS_GET_OPTION,
                sc_msgPoolIdGet ((sc_msgptr_t) &self),
            SC_FATAL_IF_TMO);
  msg->base.handle = self->base.handle;
  msg->option.level = level;
  msg->option.optname = optname;
  msg->option.optlen = *optlen;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    if (msg->option.optlen < *optlen) {
      *optlen = msg->option.optlen;
    }
    memcpy (optval, msg->option.optval, *optlen);
    sc_msgFree (&msg);
    return 0;
  }
}
```

10.13 ips_listen

This message is used to activate a listen which is waiting on connection requests of destination peers on the binded source port and IP address.

This used for **TCP** access handles. To accept connections, an access handle is first created, a willingness to accept incoming connections and a queue limit for incoming connections are specified with **ips_listen**, and then the connections are accepted with **ips_accept**. An **ips_bind** needs to be performed before using **listen**.

```
int ips_listen(
   sdd_obj_t NEARPTR        self,
   int                      backlog
);
```

Parameter

self

SDD protocol descriptor which includes the access handle. Please consult chapters **8.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 8-2** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information.

backlog

The backlog parameter defines the maximum length the queue of pending connections may grow to.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EINVAL	The parameter **self** is NULL.
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
ENOMEM	Not enough memory to execute this call.
EISCON	The socket is already connected.
EADDRINUSE	The local address is already in use.
EOPNOTSUPP	The ips_listen operation is not supported by this access handle.

Include Files

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
  ips_listen_t listen;
  ips_accept_t accept;
};
int
ips_listen (sdd_obj_t NEARPTR self, int backlog)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPS_LISTEN_REPLY, 0
  };
  sc_msg_t msg;
  int error = 0, i;
  if (!self || !self->controller) {
    sc_miscErrnoSet (ENOTSOCK);
    return -1;
  }
  msg =
    sc_msgAlloc (sizeof (ips_listen_t), IPS_LISTEN, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  msg->listen.base.error = 0;
  msg->listen.base.handle = self->base.handle;
  msg->listen.backlog = backlog;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  error = (int)msg->listen.base.error;
  sc_msgFree (&msg);
  if (error) {
    sc_miscErrnoSet (error);
    return -1;
  }
  else {
    /* send backlog accept to tcp */
    for (i = 0; i < backlog; i++) {
      msg =
        sc_msgAlloc (sizeof (ips_accept_t), IPS_ACCEPT, SC_DEFAULT_POOL,
                     SC_FATAL_IF_TMO);
      msg->accept.connect.base.error = 0;
      msg->accept.connect.base.handle = self->base.handle;
      sc_msgTx (&msg, self->controller, 0);
    }
    return 0;
  }
}
```

10.14 ips_open

This function is used to get a protocol descriptor and to open the protocol driver.

```
sdd_obj_t NEARPTR ips_open (
   const char           *family,
   const char           *protocol,
   int                  subProtocol
);
```

Parameter

family

Name of the protocol family.

This parameter can be one of the following values:

Value	Meaning
"ipv4"	IP version 4

protocol

Name of the protocol.

This parameter can be one of the following values:

Value	Meaning
"tcp"	TCP
"udp"	UDP
"icmp"	ICMP

subProtocol

This parameter can be one of the following values:

Value	Meaning
0	For TCP and UDP
PROTO_ICMP_ECHO	For ICMP
PROTO_ICMP_TIMESTAMP	For ICMP
PROTO_ICMP_INFORMATION	For ICMP
PROTO_ICMP_ADDR_MASK	For ICMP
PROTO_ICMP_MOBILE_REG	For ICMP
PROTO_ICMP_DOMAIN_NAME	For ICMP

Return Value

If the functions succeeds the return value is the pointer to the SDD protocol descriptor. Please consult chapters **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
ENOENT	Either ipv4 family or the protocol driver does not exist or has not been started yet.
NOMEM	Not enough memory to execute this call.

Include Files

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
};
sdd_obj_t NEARPTR
ips_open (const char *family, const char *protocol, int subProtocol)
{
  unsigned int tmo;
  sdd_obj_t NEARPTR link, *ipv4, *ret;
  link = sdd_manGetRoot ("/SCP_link", "link", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  if (!link) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_link");
  }
  tmo = 100;
  while (tmo < 16000 && !(ipv4 = sdd_manGetByName (link, family))) {
    sc_sleep ((sc_ticks_t)sc_tickMs2Tick (tmo));
    tmo *= 2;
  }
  sc_msgFree ((sc_msgptr_t) &link);
  if (!ipv4) {
    return NULL_OBJ;
  }
  tmo = 100;
  while (tmo < 16000 && !(ret = sdd_manGetByName (ipv4, protocol))) {
    sc_sleep ((sc_ticks_t)sc_tickMs2Tick (tmo));
    tmo *= 2;
  }
  sc_msgFree ((sc_msgptr_t) &ipv4);
  if (ret) {
    if (sdd_netOpen (ret, subProtocol, SC_CURRENT_PID) == -1) {
      sc_msgFree ((sc_msgptr_t) &ret);
    }
    return ret;
  }
  return NULL_OBJ;
}
```

10.15 ips_send

This function is used to send a netbuffer.

```
void ips_send(
   sdd_obj_tNEARPTR                self,
   sdd_netbuf_t NEARPTR NEARPTR    netbuf
);
```

Parameter

self

SDD protocol descriptor which includes the access handle. Please consult chapters **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

netbuf

Network buffer to send. Please consult chapters **8.4 “SDD Network Buffer Structure sdd_netbuf_t” on page 8-6** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information. Please note the **pointer to a pointer** type.

Return Value

None.

Errors

None.

Include Files

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
union sc_msg {
  sc_msgid_t id;
  sdd_baseMessage_t base;
};
/** Interface implementations
*/
void ips_send (sdd_obj_t NEARPTR self, sdd_netbuf_t NEARPTR NEARPTR netbuf)
{
  if (!self || !self->sender || self->sender == SC_ILLEGAL_PID) {
    return;
  }
  (*netbuf)->base.id = SDD_NET_SEND;
  (*netbuf)->base.handle = self->base.handle;
  sc_msgTx ((sc_msgptr_t) netbuf, self->sender, 0);
}
```

10.16 ips_setOption

This function is used to set the options associated with a access handle of an SDD protocol descriptor.

```
int ips_setOption(
   sdd_obj_t NEARPTR  self,
   int                level,
   int                optname,
   void               *optval,
   socklen_t          *optlen
);
```

Parameter

self

SDD protocol descriptor. Please consult chapters **8.2 "Standard SDD Object Descriptor Structure sdd_obj_t" on page 8-2** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information.

level

This parameter can be one of the following values:

Value	Meaning
SOL_SOCKET	Generic settings. Defined option names: **SO_SC_ASYNC** **SO_SC_RET_ACK** **SO_SC_RET_BUF**
SOL_TCP	TCP specific settings. Defined option names: **TCP_NODELAY**
SOL_UDP	UDP specific settings.

optname

This parameter can be one of the following values:

Value	Meaning
SO_SC_ASYNC	Asynchronous package delivery (message queue). The value (**optval**) can be **ON (1)** or **OFF (0)**.
SO_SC_RET_ACK	On every sent package an **IPS_ACK** message will be returned. The value (**optval**) can be **ON (1)** or **OFF (0)**.
SO_SC_RET_BUF	All sent packages are returned to the sender as **IPS_SEND_REPLY** message. The value (**optval**) can be **ON (1)** or **OFF (0)**.
TCP_NODELAY	Nagle algorithm switching. disabled. The value (**optval**) can be **disable (1)** or **enable (0)**.

optval

Option value.

optlen

Size of the option value (optval) in bytes.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
EBADF	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is NULL.
ENOTSOCK	The member **handle** of the **sdd_baseMessage_t** structure (in parameter **self**) is not valid.
EPERM	No permission to set this option.
EOPNOTSUPP	The level is not available.
ENOPROTOOPT	Option not supported by this level.

Header

<install_dir>\sciopta\<version>\include\ips\connect.h

Function Code

```
int
ips_setOption (sdd_obj_t NEARPTR self, int level, int optname,
           const void *optval, size_t optlen)
{
  sc_msg_t msg;
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPS_SET_OPTION_REPLY, 0
  };
  if (!self || !self->controller) {
    sc_miscErrnoSet (ENOTSOCK);
    return -1;
  }
  msg =
    sc_msgAlloc (sizeof (ips_option_t) + optlen, IPS_SET_OPTION,
         sc_msgPoolIdGet ((sc_msgptr_t) &self), SC_FATAL_IF_TMO);
  msg->base.error = 0;
  msg->base.handle = self->base.handle;
  msg->option.level = level;
  msg->option.optname = optname;
  if (optval && optlen) {
    memcpy (msg->option.optval, optval, optlen);
  }
  msg->option.optlen = optlen;
  sc_msgTx (&msg, self->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

10.17 ipv4_aton

This function is used to transform a network ascii address string into the network address in binary format.

```
int ipv4_aton(
   const char      *str,
   __u8            *addr
);
```

Parameters

str

Address in ASCII.

addr

Binary network address. The size must be at least 4 bytes.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EINVAL	Invalid Parameter.

Header

<install_dir>\sciopta\<version>\include\ips\ipv4.h

Example

```
ips_addr_t a;

a.len = 4;
if (ipv4_aton ("10.0.1.135", a.addr) == -1) {
   kprintf ("Error: %d\n", sc_miscErrnoGet ());
}
   else {
      kprintf ("Address: %d.%d.%d.%d\n", a.addr[0], a.addr[1], a.addr[2], a.addr[3]);
}
```

10.18 ipv4_arpAdd

This function is used to add a static ARP entry.

This entry once accepted will never be modified or removed by the IPS system. To remove an ARP entry you need to use the function call 10.19 “ipv4_arpRm” on page 10-39.

If the ARP entry already exists it will be overwritten by the new ARP entry.

```
int ipv4_arpAdd(
   ipv4_arp_t NEARPTR NEARPTRarp
);
```

Parameter

arp

ARP entry to add. Please consult chapters 8.7 “IPV4 ARP Address Structure ipv4_arp_t” on page 8-11 and 8.9 “NEARPTR and FARPTR” on page 8-13 for type information. Please note the **pointer to a pointer** type.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
ENOMEM	Not enough memory to execute this call.

System Error

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_devman** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	IPv4 not available or not yet up.

Include Files

<install_dir>\sciopta\<version>\include\ips\arp.h
<install_dir>\sciopta\<version>\include\ips\arp.msg

Function Code

```
int
ipv4_arpAdd (ipv4_arp_t NEARPTR NEARPTR arp)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPV4_ARP_ADD_REPLY, 0
  };
  sc_msg_t msg;
  sdd_obj_t NEARPTR devman, *arpdev;
  devman =
    sdd_manGetRoot ("/SCP_devman", "devman", SC_DEFAULT_POOL,
                    SC_FATAL_IF_TMO);
  if (!devman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_devman");
  }
  arpdev = sdd_manGetByName (devman, "arp");
  sdd_objFree (&devman);
  if (!arpdev) {
    sc_miscError (IPS_ERR_BASE + SC_ENOENT, (sc_extra_t) "arp");
  }
  (*arp)->id = IPV4_ARP_ADD;
  /* cause of the unique msg id we do not need the arpdev->base.handle! */
  sc_msgTx ((sc_msgptr_t) arp, arpdev->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  sdd_objFree (&arpdev);
  if (msg->arp.error) {
    sc_miscErrnoSet (msg->arp.error);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

10.19 ipv4_arpRm

This function is used to remove a static ARP entry.

```
int ipv4_arpRm(
   ipv4_arp_t NEARPTRarp
);
```

Parameter

arp

ARP entry to add. Please consult chapters **8.7 “IPV4 ARP Address Structure ipv4_arp_t” on page 8-11** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information. The member **hwaddr** is not used.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
ENOENT	The ARP entry does not exist.

System Error

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_devman** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	IPv4 not available or not yet up.

Include Files

<install_dir>\sciopta\<version>\include\ips\arp.h
<install_dir>\sciopta\<version>\include\ips\arp.msg

Function Code

```
int
ipv4_arpRm (ipv4_arp_t NEARPTR arp)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPV4_ARP_RM_REPLY, 0
  };
  sc_msg_t msg;
  sdd_obj_t NEARPTR devman, *arpdev;
  devman =
    sdd_manGetRoot ("/SCP_devman", "devman", SC_DEFAULT_POOL,
                    SC_FATAL_IF_TMO);
  if (!devman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_devman");
  }
  arpdev = sdd_manGetByName (devman, "arp");
  sdd_objFree (&devman);
  if (!arpdev) {
    sc_miscError (IPS_ERR_BASE + SC_ENOENT, (sc_extra_t) "arp");
  }
  msg =
    sc_msgAlloc (sizeof (ipv4_arp_t), 0, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  memcpy (msg, arp, sizeof (ipv4_arp_t));
  msg->id = IPV4_ARP_RM;
  /* cause of the unique msg id we do not need the arpdev->base.handle! */
  sc_msgTx (&msg, arpdev->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  sdd_objFree (&arpdev);
  if (msg->arp.error) {
    sc_miscErrnoSet (msg->arp.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

10.20 ipv4_getProtocol

This function is used to get a SDD protocol descriptor.

```
sdd_obj_t NEARPTR ipv4_getProtocol(
   const char                *name
);
```

Parameters

name

Name of the SDD protocol descriptor. Valid names are: “udp”, “tcp” and “icmp”.

Return Value

If the functions succeeds the return value is the pointer to the SDD protocol descriptor. Please consult chapters **8.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 8-2** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

If the function fails the return value is NULL. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
ENOENT	The protocol is not (yet) registered or not existing.

System Error

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal.The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_link** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	The process **SCP_ipv4** is not available or not yet up.

Include Files

<install_dir>\sciopta\<version>\include\ips\ipv4.h

Function Code

```
sdd_obj_t NEARPTR
ipv4_getProtocol (const char *name)
{
  sdd_obj_t NEARPTR link;
  sdd_obj_t NEARPTR ipv4;
  sdd_obj_t NEARPTR ret;
  link =
    sdd_manGetRoot ("/SCP_link", "link", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
  if (!link) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_link");
  }
  ipv4 = sdd_manGetByName (link, "ipv4");
  sc_msgFree ((sc_msgptr_t) &link);
  if (!ipv4) {
    sc_miscError (IPS_ERR_BASE + SC_ENOENT, __LINE__);
  }
  ret = sdd_manGetByName (ipv4, name);
  sc_msgFree ((sc_msgptr_t) &ipv4);
  return ret;
}
```

10.21 ipv4_ntoa

This function is used to transform a network address in binary format into a network ascii address string.

```
int ipv4_ntoa(
   __u8          *addr,
   char          *str,
   int           len
);
```

Parameter

addr

Binary network address. The size must be at least 4 bytes.

str

Address in ascii.

len

Size of the ascii address string.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
EINVAL	Invalid Parameter.

Include Files

<install_dir>\sciopta\<version>\include\ips\ipv4.h
<install_dir>\sciopta\<version>\include\ips\addr.h

Example

```
#include <sciopta.h>
#include <ips/ipv4.h>
#include <ips/addr.h>

char address[16];
ips_addr_t a;

a.len = 4;
a.addr[0] = 10;
a.addr[1] = 0;
a.addr[2] = 1;
a.addr[3] = 135;

if (ipv4_ntoa (a.addr, address, 16) == -1) {
   kprintf ("Error: %d\n", sc_miscErrnoGet ());
   }
else {
   kprintf ("Address: %s\n", address);
}
```

10.22 ipv4_routeAdd

This function is used to add a route in IPS.

The SDD network device descriptor can be requested from process SCP_netman by using the function **10.9 "ips_devGetByName" on page 10-18** or the function **10.8 "ips_devGetByHwAddr" on page 10-17**.

The device specific parameters (isp_dev_s) can then be copied into the **ipv4_route_t** structure.

```
int ipv4_routeAdd(
   ipv4_route_t NEARPTR NEARPTRroute
);
```

Parameter

route

SDD object descriptor of a route to add. Please consult chapters **8.8 "IPV4 Route Structure ipv4_route_t" on page 8-12** and **8.9 "NEARPTR and FARPTR" on page 8-13** for type information. Please note the **pointer to a pointer** type.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	Meaning
EINVAL	The netmask is not valid.
EEXIST	The route already exists.

System Errors

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal. The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_devman** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	The process **SCP_ipv4** is not (yet) available.

Include Files

<install_dir>\sciopta\<version>\include\ips\router.h
<install_dir>\sciopta\<version>\include\ips\router.msg

Function Code

```
int
ipv4_routeAdd (ipv4_route_t NEARPTR NEARPTR route)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPV4_ROUTE_ADD_REPLY, 0
  };
  sc_msg_t msg;
  sdd_obj_t NEARPTR devman;
  sdd_obj_t NEARPTR router;
  devman =
    sdd_manGetRoot ("/SCP_devman", "devman", SC_DEFAULT_POOL,
                    SC_FATAL_IF_TMO);
  if (!devman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_devman");
  }
  router = sdd_manGetByName (devman, "router");
  sdd_objFree (&devman);
  if (!router) {
    sc_miscError (IPS_ERR_BASE + SC_ENOENT, (sc_extra_t) "ipv4");
  }
  (*route)->device.object.base.id = IPV4_ROUTE_ADD;
  /* (*route)->device.object.manager = router->base.handle; */
  /* because of the unique msg id we do not need the router->base.handle! */
  sc_msgTx ((sc_msgptr_t) route, router->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  sdd_objFree (&router);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

10.23 ipv4_routeRm

This function is used to remove a route in IPS.

```
int ipv4_routeRm(
   ipv4_route_t NEARPTRroute
);
```

Parameter

route

SDD object descriptor of a route to remove. Please consult chapters **8.8 “IPV4 Route Structure ipv4_route_t” on page 8-12** and **8.9 “NEARPTR and FARPTR” on page 8-13** for type information.

Return Value

If the functions succeeds the return value is zero or positive.

If the function fails the return value is -1. To get the error information call **sc_miscErrnoGet**.

Errors

The following error codes are defined by a standard device driver for the **sc_miscErrnoGet** system call return value:

Value	**Meaning**
ENOENT	Route entry does not exist.

System Errors

The following system errors can occur during handling of this function. The error will be sent to the kernel and is fatal. The kernel will call the error hook if it exists. The error hook is user written and can be handled by the user.

IPS_ERR_BASE+SC_ENOPROC	The process **SCP_devman** which normally should reside in the system module does not exist.
IPS_ERR_BASE+SC_ENOENT	The process **SCP_ipv4** is not (yet) available.

Include Files

<install_dir>\sciopta\<version>\include\ips\router.h
<install_dir>\sciopta\<version>\include\ips\router.msg

Function Code

```
int
ipv4_routeRm (ipv4_route_t NEARPTR route)
{
  static const sc_msgid_t select[3] = {
    SDD_ERROR, IPV4_ROUTE_RM_REPLY, 0
  };
  sc_msg_t msg;
  sdd_obj_t NEARPTR devman;
  sdd_obj_t NEARPTR router;
  devman =
    sdd_manGetRoot ("/SCP_devman", "devman", SC_DEFAULT_POOL,
                    SC_FATAL_IF_TMO);
  if (!devman) {
    sc_miscError (IPS_ERR_BASE + SC_ENOPROC, (sc_extra_t) "/SCP_devman");
  }
  router = sdd_manGetByName (devman, "router");
  sdd_objFree (&devman);
  if (!router) {
    sc_miscError (IPS_ERR_BASE + SC_ENOENT, (sc_extra_t) "ipv4");
  }
  msg =
    sc_msgAlloc (sizeof (ipv4_route_t), 0, SC_DEFAULT_POOL,
                 SC_FATAL_IF_TMO);
  memcpy (msg, route, sizeof (ipv4_route_t));
  msg->id = IPV4_ROUTE_RM;
  /* msg->route.device.object.manager = router->base.handle; */
  /* because of the unique msg id we do not need the router->base.handle! */
  sc_msgTx (&msg, router->controller, 0);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  sdd_objFree (&router);
  if (msg->base.error) {
    sc_miscErrnoSet (msg->base.error);
    sc_msgFree (&msg);
    return -1;
  }
  else {
    sc_msgFree (&msg);
    return 0;
  }
}
```

11 BSD Socket Interface Reference

11.1 Introduction

This chapter is the reference of the SCIOPTA BSD Socket Interface. The functions are listed in alphabetical order.

11.2 accept

Description

This function is used to accept a connection on a socket.

The argument sd is a BSD socket descriptor that has been created with **socket()**, bound to an address with **bind()**, and is listening for connections after a **listen()**. The **accept()** function extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of sd, and allocates and returns a new BSD descriptor for the socket. The new socket does not remain in the listening state. The listening state of the original socket will not be modified.

The **addr** argument is a pointer to the local address which is defined by the socket family. The argument **addrlen** must contain the size of the structure of **addr**. If **accept** returns, **addrlen** contains the real size of the address of the accepted connection and **addr** contains the address of the other side.

If there are no pending connections present on the queue **accept** blocks the caller until a connection is present.

Syntax

```
int accept (int sd, struct sockaddr *addr, socklen_t *addrlen);
```

Parameters

sd	BSD socket descriptor.
addr	Pointer to the local address. The structure is defined by the socket family.
addrlen	Size of the structure of parameter **addr**.

Return Value

Returns a new socket (>=0) if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad BSD socket descriptor.
EINVAL	Wrong parameter (struct socketaddr, socklen_t).
ENOTSOCK	Descriptor is not a socket but a file descriptor.
ENOMEM	Not enough memory to execute this call.
EOPNOTSUPP	The referenced socket is not of type **SOCK_STREAM**.

EMFILE Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int accept (int sd, struct sockaddr *addr, socklen_t * addrlen)
{
   static const sc_msgid_t select[2] = { SCIO_SOCKET_REPLY, 0 };
   fd_tab_t *fd_tab;
   int fd = 0;
   sdd_obj_t *obj;
   ips_addr_t ip;
   struct sockaddr_in *s_in;
   sc_pid_t socket;
   void *handle;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                          sd <= fd_tab->max_fd && fd_tab->bf[sd]) {
         s_in = (void *) addr;
         obj = ips_accept (fd_tab->bf[sd], &ip, &s_in->sin_port);
         if (!obj) {
            return -1;
         }
         else {
               /* store the handle form accept */
            handle = obj->base.handle;
               /* get scio socket from listener socket */
            obj->base.id = SCIO_SOCKET;
            sc_msgTx ((union sc_msg **) &obj, obj->controller, 0);
            obj = (sdd_obj_t *) sc_msgRx (SC_ENDLESS_TMO, (void *) select,
                                                                SC_MSGRX_MSGID);
               /* replace the listener handle with the accepted handle */
            obj->base.handle = handle;

               /* start a socket process */
            if ((socket =sc_procTmpCreate (SCP_socket, "socket", 512,
                                             SC_DEFAULT_POOL)) == SC_ILLEGAL_PID) {
               sc_miscErrnoSet (ENOMEM);
               return -1;
            }

            obj->base.id = SCIO_SOCKET;
            sc_msgTx ((union sc_msg **) &obj, socket, 0);
```

```
            obj = (sdd_obj_t *) sc_msgRx (SC_ENDLESS_TMO, (void *) select,
                                                            SC_MSGRX_MSGID);

            while (fd < fd_tab->max_fd && fd_tab->bf[fd]) {
               ++fd;
            }
            if (fd != fd_tab->max_fd) {
               fd_tab->bf[fd] = obj;
               return fd;
            }
            else {
               sc_miscErrnoSet (EMFILE);
               return -1;
            }
         }
      }
      else {
         sc_miscErrnoSet (EBADF);
         return -1;
      }
   }

   sc_miscErrnoSet (EMFILE);
   return -1;
}
```

11.3 bind

Description

This function is used to bind a name to the socket.

The bind call gives the socket **sd** the local address **addr**. This is also called "assigning a name to a socket". When a socket is created with socket(), it exists in a address family but has no name assigned.

Before a **SOCK_STREAM** socket is put into the **LISTEN** state to receive connections, you usually need to first assign a local address using **bind** to make the socket visible.

Syntax

```
int bind (int sd, struct sockaddr *addr, socklen_t len);
```

Parameters

sd	BSD socket descriptor.
addr	Pointer to the local address. The structure is defined by the socket family.
len	Size of the structure of parameter **addr**.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad BSD socket descriptor.
EINVAL	Wrong parameter (struct socketaddr, socklen_t).
ENOTSOCK	Descriptor is not a socket but a file descriptor.
ENOMEM	Not enough memory to execute this call.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int bind (int sd, struct sockaddr *addr, socklen_t addrlen)
{
   fd_tab_t *fd_tab;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                          sd <= fd_tab->max_fd && fd_tab->bf[sd]) {
         if (addr && addr->sa_family == AF_INET && addrlen == 16) {
            ips_addr_t ip;
            struct sockaddr_in *s_in;
            s_in = (void *) addr;
            if (s_in->sin_addr.s_addr) {
               ip.len = 4;
               memcpy (&ip.addr, &s_in->sin_addr.s_addr, 4);
            }
            else {
               ip.len = 0;
            }
            return ips_bind (fd_tab->bf[sd], &ip, s_in->sin_port);
         }
         else {
            sc_miscErrnoSet (EINVAL);
            return -1;
         }
      }
      else {
         sc_miscErrnoSet (EBADF);
         return -1;
      }
   }

   sc_miscErrnoSet (EMFILE);
   return -1;
}
```

11.4 close

Description

This function is used to close a BSD descriptor.

The BSD descriptor is closed, so that it no longer refers to any socket and may be reused. An actual connection from the host to peer will be removed while a possible connection from peer to host will be maintained. It will only be removed if the peer also closes the connection.

Syntax

```
int close (int fd);
```

Parameters

fd	BSD descriptor.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad BSD descriptor.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\unistd.h

Function Code

```
int close (int fd)
{
   fd_tab_t *fd_tab;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                            fd <= fd_tab->max_fd && fd_tab->bf[fd]) {
         if ((sdd_devClose (fd_tab->bf[fd])) != -1) {
            sc_msgFree ((union sc_msg **) &fd_tab->bf[fd]);
            fd_tab->bf[fd] = NULL;
            return 0;
         }
         else {
            sc_msgFree ((union sc_msg **) &fd_tab->bf[fd]);
         fd_tab->bf[fd] = NULL;
         return -1;
         }
      }
   }

   sc_miscErrnoSet (EBADF);
   return -1;
}
```

11.5 connect

Description

This function is used to initiate a connection on a socket.

The parameter **sockfd** is a BSD socket descriptor returned by the call **socket()**. If the socket is of type **SOCK_DGRAM**, the address **serv_addr** will be used as default for transmitting. If the socket is of type **SOCK_STREAM**, this call attempts to make a connection to a peer. The parameter **addrlen** specifies the size of the structure of **serv_addr** (usually sizeof (struct sockaddr_in)).

Generally, stream sockets may successfully connect only once; datagram sockets may use connect multiple times to change their association.

Syntax

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

Parameters

sockfd	BSD socket descriptor.
serv_addr	Pointer to the local address. The structure is defined by the socket family.
addrlen	Size of the structure of parameter **serv_addr**.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad descriptor.
EINVAL	Wrong parameter (struct socketaddr, socklen_t).
ENOTSOCK	Descriptor is not a socket but a file descriptor.
ENOMEM	Not enough memory to execute this call.
EISCON	The socket is already connected.
ECONNREFUSED	The connection has been refused by the peer.
ETIMEDOUT	A time out occurred at connection.
ENETUNREACH	The network cannot be reached.
EADDRINUSE	The local address is already in use.
EAFNOSUPPORT	Wrong address family.
EMFILE	Descriptor table not installed.

SCIOPTA

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int connect (int sd, const struct sockaddr *addr, socklen_t addrlen)
{
   fd_tab_t *fd_tab;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                          sd <= fd_tab->max_fd && fd_tab->bf[sd]) {
         if (addr && addr->sa_family == AF_INET && addrlen == 16) {
            ips_addr_t ip;
            struct sockaddr_in *s_in;
            s_in = (void *) addr;
            ip.len = 4;
            memcpy (&ip.addr, &s_in->sin_addr.s_addr, 4);
            return (ips_connect (fd_tab->bf[sd], &ip, s_in->sin_port));
         }
         else {
            sc_miscErrnoSet (EINVAL);
            return -1;
         }
      }
      else {
         sc_miscErrnoSet (EBADF);
         return -1;
      }
   }

   sc_miscErrnoSet (EMFILE);
   return -1;
}
```

11.6 dup

Description

This function is used to duplicate a BSD descriptor.

A copy of the BSD descriptor **oldfd** will be created.

Syntax

```
int dup (int oldfd);
```

Parameters

oldfd	Original BSD descriptor.

Return Value

Returns a new BSD descriptor (>=0) if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad descriptor.
EMFILE	Descriptor table not installed

Include Files

<install_dir>\sciopta\<version>\include\sys\unistd.h

Function Code

```
int dup (int oldfd)
{
   fd_tab_t *fd_tab;

   sc_miscErrnoSet (0);
   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC) {
         int fd;

         if (oldfd < 0 || oldfd >= fd_tab->max_fd || !fd_tab->bf[oldfd]) {
            sc_miscErrnoSet (EBADF);
            return -1;
         }

         fd = 0;
         while (fd < fd_tab->max_fd && fd_tab->bf[fd]) {
            ++fd;
         }
         if (fd >= fd_tab->max_fd) {
            sc_miscErrnoSet (EMFILE);
            return -1;
         }
         if ((fd_tab->bf[fd] = sdd_objDup (fd_tab->bf[oldfd]))) {
            return fd;
         }
         else {
            return -1;
         }
      }
   }
   else {
      sc_miscErrnoSet (EMFILE);
   }
   return -1;
}
```

11.7 dup2

Description

This function is used to duplicate a BSD descriptor.

A copy of the BSD descriptor **oldfd** will be created. **dup2** makes newfd be the copy of oldfd, closing newfd first if necessary.

Syntax

```
int dup2 (int oldfd, int newfd);
```

Parameters

oldfd	Original BSD descriptor.
newfd	New BSD descriptor.

Return Value

Returns a new BSD descriptor (>=0) if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad BSD descriptor (**oldfd**).
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\unistd.h

Function Code

```
int dup2 (int oldfd, int newfd)
{
   fd_tab_t *fd_tab;

   sc_miscErrnoSet (0);
   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC) {
         if (newfd < 0 || newfd >= fd_tab->max_fd) {
            sc_miscErrnoSet (EINVAL);
            return -1;
         }

         if (fd_tab->bf[newfd]) {
            sc_miscErrnoSet (EEXIST);
            return -1;
         }

         if (oldfd < 0 || oldfd >= fd_tab->max_fd || !fd_tab->bf[oldfd]) {
            sc_miscErrnoSet (EBADF);
            return -1;
         }

         if ((fd_tab->bf[newfd] = sdd_objDup (fd_tab->bf[oldfd]))) {
            return newfd;
         }
         else {
            return -1;
         }
      }
   }
   else {
      sc_miscErrnoSet (EMFILE);
   }
   return -1;
}
```

11.8 gethostbyaddr

Description

This function is used to return the host names in struct hostent for the given host address.

Syntax

```
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Parameters

addr	Pointer to a socket address (sockaddr_in, castet to const * char).	
len	Length of **addr** in bytes.	
type	Socket type.	
	AF_INET	Internet IP protocol.

Return Value

Returns a pointer to struct hostent if the operation was successful. A return value of **NULL** indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

A pointer to static data may be returned which will be overwritten by subsequent calls. In this case a copy is required.

```
struct hostent {
   char    *h_name;
   char    **h_aliases;
   int     h_addrtype;
   int     h_length;
   char    **h_addr_list;
}
#define h_addr  h_addr_list[0]   /* for backward compatibility */
```

h_name	Pointer to the official name of the host.	
h_aliases	Pointer to a pointer to a name list. This is a zero terminated list of alternative host names.	
h_addrtype	Host address type.	
	AF_INET	Internet IP protocol.
h_length	The length of the address in bytes.	
h_addr_list	Pointer to a pointer to an address list. This is a zero terminated list of network addresses in netbyte order.	
h_addr	First element in **h_addr_list** for backward compatibility.	

Errors

ENOENT No entry

Include Files

<install_dir>\sciopta\<version>\include\netdb.h
<install_dir>\sciopta\<version>\include\sys\socket.h

Function Code

```
struct hostent * gethostbyaddr (const char *addr, int len, int type)
{
   struct hostent *hostent;
   union sc_msg *msg;
   int i;
   static const sc_msgid_t select[3] = {SDD_ERROR, DNS_NAME_REPLY_MSG, 0};
   sc_pid_t to = sc_procIdGet (resolver, 0);

   if (len <= 16 && type == AF_INET && to) {
      int error;

      msg = sc_msgAlloc (sizeof (dns_pcapIp_t),
                         DNS_NAME_REQUEST_MSG,
                         SC_DEFAULT_POOL,
                         SC_ENDLESS_TMO);
      msg->nameRequest.noOf = 1;
      msg->nameRequest.entry[0].ip.len = 4;
      for (i = 0; i < 4; i++) {
         msg->nameRequest.entry[0].ip.addr[i] = atoi (addr);
         addr = strstr (addr, ".") + 1;
      }
      sc_msgTx (&msg, to, 0);
      msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
      error = msg->ipReply.parent.error;
      if (!error && msg->nameReply.noOf) {
         hostent = getHostent ();

         if (hostent) {
            hostent->h_length = len;
            strncpy ((char *) hostent->h_name,
                     (const char *) msg->nameReply.entry[0].name,
                     DNS_NAME_MAX);
            hostent->h_name[DNS_NAME_MAX] = 0;
         }

         return hostent;
      }
```

```
      else {
         sc_msgFree (&msg);
         if (error) {
            sc_miscErrnoSet (error);

         else {
            sc_miscErrnoSet (ENOENT);
         }
         return NULL;
      }
   }
   else {
      return NULL;
   }
}
```

11.9 gethostbyname

Description

This function is used to return a structure of type hostent for the given host name.

Syntax

```
struct hostent *gethostbyname (const char *name);
```

Parameters

name Either a host name or an IPv4 address

Return Value

Returns a pointer to struct hostent if the operation was successful. A return value of **NULL** indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

A pointer to static data may be returned which will be overwritten by subsequent calls. In this case a copy is required.

```
struct hostent {
   char    *h_name;
   char    **h_aliases;
   int     h_addrtype;
   int     h_length;
   char    **h_addr_list;
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

h_name	Pointer to the official name of the host.	
h_aliases	Zero terminated array of alternative names for the host..	
h_addrtype	Host address type.	
	AF_INET	Internet IP protocol.
h_length	The length of the address in bytes.	
	4	IPv4
h_addr_list	Pointer to a pointer to an address list. This is a zero terminated list of network addresses in netbyte order.	
h_addr	First element in **h_addr_list** for backward compatibility.	

Errors

ENOENT No entry

SCIOPTA

Include Files

<install_dir>\sciopta\<version>\include\netdb.h
<install_dir>\sciopta\<version>\include\sys\socket.h

Function Code

```
struct hostent *gethostbyname (const char *name)
{
   struct hostent *hostent;
   union sc_msg *msg;
   static const sc_msgid_t select[3] = {SDD_ERROR, DNS_IP_REPLY_MSG, 0};
  sc_pid_t to = sc_procIdGet (resolver, 0);

   if (to && to != SC_ILLEGAL_PID) {
      int error;

      msg = sc_msgAlloc (sizeof (dns_pcapName_t),
                         DNS_IP_REQUEST_MSG,
                         SC_DEFAULT_POOL,
                         SC_ENDLESS_TMO);
      msg->ipRequest.noOf = 1;
      strncpy ((char *) msg->ipRequest.entry[0].name, name, DNS_NAME_MAX);
      sc_msgTx (&msg, to, 0);
      msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
      error = msg->ipReply.parent.error;
      if (!error && msg->ipReply.noOf) {
         int i;
         hostent = getHostent ();
         if (hostent) {
            hostent->h_length = msg->ipReply.entry[0].ip.len;
            for (i = 0; i < msg->ipReply.noOf; i++) {
               memcpy (hostent->h_addr_list[i],
                       msg->ipReply.entry[0].ip.addr,
                       hostent->h_length);
            }
         }
         return hostent;
      }
      else {
         sc_msgFree (&msg);
         if (error) {
            sc_miscErrnoSet (error);
         }
         else {
            sc_miscErrnoSet (ENOENT);
         }
         return NULL;
      }
   }
   else {
      sc_miscErrnoSet (ENOENT);
      return NULL;
   }
}
```

11.10 getpeername

Description

This function is used to return the name of a connected peer.

Syntax

```
int getpeername(int sd, struct sockaddr *addr, socklen_t *len);
```

Parameters

sd	BSD socket descriptor.
addr	Pointer to the local address. The structure is defined by the socket family.
len	Pointer to the size of the structure of parameter **addr**.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad descriptor.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int getpeername (int fd, struct sockaddr *addr, socklen_t * len)
{
   fd_tab_t *fd_tab;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                            fd <= fd_tab->max_fd && fd_tab->bf[fd]) {
         ips_info_t *info = (ips_info_t *) sdd_objInfo (fd_tab->bf[fd],
                         sizeof (ips_info_t));
         if (!info) {
            return -1;
         }
         else {
            if (info->family == AF_INET && *len >= sizeof (struct sockaddr_in)) {
               struct sockaddr_in *s_in;
               s_in = (void *) addr;
               s_in->sin_family = info->family;
               memcpy (&s_in->sin_addr.s_addr, info->dstAddr.addr, 4);
               s_in->sin_port = info->dstPort;
               *len = sizeof (struct sockaddr_in);
               sc_miscErrnoSet (0);
               return 0;
            }
            else {
               sc_miscErrnoSet (EINVAL);
               return -1;
            }
         }
      }
   }
   sc_miscErrnoSet (EBADF);
   return -1;
}
```

11.11 getsockname

Description

This function is used to return the socket name.

Syntax

```
int getsockname(int sd, struct sockaddr *addr, socklen_t *len);
```

Parameters

sd	BSD socket descriptor.
addr	Pointer to the local address. The structure is defined by the socket family.
len	Pointer to the size of the structure of parameter **addr**.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad descriptor.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int getsockname (int fd, struct sockaddr *addr, socklen_t * len)
{
   fd_tab_t *fd_tab;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                          fd <= fd_tab->max_fd && fd_tab->bf[fd]) {
         ips_info_t *info =(ips_info_t *) sdd_objInfo (fd_tab->bf[fd],
                               sizeof (ips_info_t));
         if (info->objectInfo.base.error) {
            sc_miscErrnoSet (info->objectInfo.base.error);
            sc_msgFree ((union sc_msg **) &info);
            return -1;
         }
         else {
            if (info->family == AF_INET && *len >= sizeof (struct sockaddr_in)) {
               struct sockaddr_in *s_in;
               s_in = (void *) addr;
               s_in->sin_family = info->family;
               memcpy (&s_in->sin_addr.s_addr, info->srcAddr.addr, 4);
               s_in->sin_port = info->srcPort;
               *len = sizeof (struct sockaddr_in);
               sc_miscErrnoSet (0);
               return 0;
            }
         }
      }
   }

   sc_miscErrnoSet (EBADF);
   return -1;
}
```

11.12 getsockopt

Description

This function is used to get the options associated with a socket.

Syntax

```
int getsockopt(int sd, int level, int optname, void *optval, socklen_t *optlen);
```

Parameters

sd	BSD socket descriptor.	
level	**SOL_SOCKET**	Generic socket settings.
	SOL_TCP	TCP specific settings.
	SOL_UDP	UDP specific settings.
optname	**SO_SC_ASYNC**	Asynchronous package delivery (message queue).
	SO_SC_RET_ACK	On every sent package an **IPS_ACK** message will be returned.
	SO_SC_RET_BUF	All sent packages are returned to the sender as **IPS_SEND_REPLY** message.
	TCP_NODELAY	Nagle algorithm disabled.
optval	Pointer to option value.	
optlen	Pointer to size of option value (optval) in bytes.	

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad BSD descriptor.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int getsockopt (int fd, int level, int optname, void *optval, socklen_t * optlen)
{
   fd_tab_t *fd_tab;
   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                          fd <= fd_tab->max_fd && fd_tab->bf[fd]) {
         return ips_getOption (fd_tab->bf[fd], level, optname, optval, optlen);
      }
   }

   sc_miscErrnoSet (EBADF);
   return -1;
}
```

11.13 inet_aton

Description

This function is used to convert the Internet host address **cp** from the standard numbers-and-dots notation into binary data in net byte order and stores it in the structure that **inp** points to.

Syntax

```
int inet_aton(const char *cp, struct in_addr *inp);
```

Parameters

cp Pointer to a internet host address.

inp Pointer to an internet address.

```
typedef __u32 in_addr_t;
   struct in_addr {
   in_addr_t s_addr;
};
```

Return Value

Returns **=! 0** if the operation was successful. A return value of 0 indicates an error condition.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\netinet\in.h
<install_dir>\sciopta\<version>\include\arpa\inet.h

11.14 inet_ntoa

Description

This function is used to convert the Internet host address **in** given in network byte order to a string in standard numbers-and-dots notation.

The functions returns a string, which subsequent calls will overwrite.

Syntax

```
char *inet_ntoa(struct in_addr in);
```

Parameters

in Internet address.

```
typedef __u32 in_addr_t;
   struct in_addr {
   in_addr_t s_addr;
};
```

Return Value

Pointer to a string. The pointer could point to static data which will be overwritten by subsequent calls. In this case a copy is required.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\netinet\in.h
<install_dir>\sciopta\<version>\include\arpa\inet.h

11.15 listen

Description

This function is used to listen for connections on a socket.

This used for sockets of type **SOCK_STREAM**. To accept connections, a socket is first created with **socket**, a willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**, and then the connections are accepted with **accept**. A **bind** needs to be performed before using **listen**.

Syntax

```
int listen(int sd, int n);
```

Parameters

sd	BSD socket descriptor.
n	The backlog parameter n defines the maximum length the queue of pending connections may grow to.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Descriptor in **sd** is not valid.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EOPNOTSUPP	Not of type **SOCK_STREAM**.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int listen (int sd, int backlog)
{
   fd_tab_t *fd_tab;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                            sd <= fd_tab->max_fd && fd_tab->bf[sd]) {
         return ips_listen (fd_tab->bf[sd], backlog);
      }
      else {
         sc_miscErrnoSet (EINVAL);
         return -1;
      }
   }
   sc_miscErrnoSet (EMFILE);
   return -1;
}
```

11.16 recv

Description

This function is used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

Syntax

```
ssize_t recv(int sd, const void *buf, size_t n, int flags);
```

Parameters

sd	BSD socket descriptor.	
buf	Pointer to received data.	
n	Number of bytes which you want to receive.	
flags	**0**	No flags.
	MSG_DONTWAIT	Enables non-blocking operation. **EAGAIN** will be returned if no data available.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Descriptor in **sd** is not valid.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EOPNOTSUPP	Not of type **SOCK_STREAM**.
EMFILE	Descriptor table not installed.
EIO	I/O error.
EAGAIN	If non-blocking operation and no data are received.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

11.17 recvfrom

Description

This function is used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented. If **addr** is not NULL, and the socket is not connection-oriented, the source address of the message is filled in.

Syntax

```
ssize_t recvfrom(int sd, void *buf, size_t n, int flags,
                                     sockaddr *from, socklen_t *fromlen);
```

Parameters

sd	BSD socket descriptor.	
buf	Pointer to received data.	
n	Number of bytes which you want to receive.	
flags	**0**	No flags.
	MSG_DONTWAIT	Enables non-blocking operation. **EAGAIN** will be returned if no data available.
from	Pointer to the source address. The structure is defined by the socket family.	
fromlen	Pointer to the size of the structure of parameter **from**.	

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Descriptor in **sd** is not valid.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EOPNOTSUPP	Not of type **SOCK_STREAM**.
EMFILE	Descriptor table not installed.
EIO	I/O error.
EAGAIN	If non-blocking operation and no data are received.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

11.18 send

Description

This function is used to transmitt a message to another socket. Send may be used only when the socket is on a connected state.

Syntax

```
ssize_t send(int sd, const void *buf, size_t n, int flags);
```

Parameters

sd	BSD socket descriptor.	
buf	Pointer to data.	
n	Length of the data.	
flags	**0**	No flags.
	MSG_DONTWAIT	Enables non-blocking operation. **EAGAIN** will be returned if no data available.

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Descriptor in **sd** is not valid.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EOPNOTSUPP	Not of type **SOCK_STREAM**.
EMFILE	Descriptor table not installed.
EIO	I/O error.
EAGAIN	If non-blocking operation and no data are received.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

11.19 sendto

Description

This function is used to transmitt a message to another socket. Send may be used only when the socket is on a connected state , while sendto may be used at any time.

Syntax

```
ssize_t sendto(int sd, const void *buf, size_t n, int flags,
                                          sockaddr *addr, socklen_t *len);
```

Parameters

sd	BSD socket descriptor.	
buf	Pointer to data.	
n	Length of the data.	
flags	**0**	No flags.
	MSG_DONTWAIT	Enables non-blocking operation. **EAGAIN** will be returned if no data available.
addr	Pointer to the destination address. The structure is defined by the socket family.	
len	Pointer to the size of the structure of parameter **addr**.	

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Descriptor in **sd** is not valid.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EOPNOTSUPP	Not of type **SOCK_STREAM**.
EMFILE	Descriptor table not installed.
EIO	I/O error.
EAGAIN	If non-blocking operation and no data are received.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

11.20 setsockopt

Description

This function is used to set the options associated with a socket.

Syntax

```
int setsockopt(int sd, int level, int optname, const void *optval, socklen_t optlen);
```

Parameters

sd	BSD socket descriptor.	
level	**SOL_SOCKET**	Generic socket settings.
	SOL_TCP	TCP specific settings.
	SOL_UDP	UDP specific settings.
optname	**SO_SC_ASYNC**	Asynchronous package delivery (message queue).
	SO_SC_RET_ACK	On every sent package an **IPS_ACK** message will be returned.
	SO_SC_RET_BUF	All sent packages are returned to the sender as **IPS_SEND_REPLY** message.
	TCP_NODELAY	Nagle algorithm disabled.
optval	Pointer to option value.	
optlen	Size of option value (optval) in bytes.	

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EBADF	Bad descriptor.
ENOTPROTOOPT	The option is unknown at the level indicated.
ENOTSOCK	Descriptor is not a socket but a file descriptor.
EMFILE	Descriptor table not installed.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h

<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int setsockopt (int fd, int level,
                int optname,
                const void *optval,
                socklen_t optlen)
{
   fd_tab_t *fd_tab;
   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC &&
                                       fd <= fd_tab->max_fd && fd_tab->bf[fd]) {
         return ips_setOption (fd_tab->bf[fd], level, optname, optval, optlen);
      }
   }

   sc_miscErrnoSet (EBADF);
   return -1;
}
```

11.21 socket

Description

This function is used to create an endpoint for communication.

It returns a BSD socket descriptor for the specified domain, type and protocol.

Syntax

```
int socket(int domain, int type, int protocol);
```

Parameters

domain	Communication domain within which communication will take place. This selects the protocol family which will be used.	
	PF_INET	IPv4 Internet protocols.
type	Socket type which specifies the semantics of communication.	
	SOCK_STREAM	Provides sequenced, reliable, two-way connection based byte streams (TCP/IP).
	SOCK_DGRAM	Supports datagrams, connectionless, unreliable messages of a fixed maximum length (UDP/IP).
protocol	**SOCK_ICMP**	ICMP socket support.
		PROTO_ICMP_ECHO
		PROTO_ICMP_TIMESTAMP
		PROTO_ICMP_INFORMATION
		PROTO_ICMP_ADDR_MASK
		PROTO_ICMP_MOBILE_REG
		PROTO_ICMP_DOMAIN_NAME

Return Value

Returns a zero if the operation was successful. A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** system call.

Errors

EINVAL	Wrong parameter (struct socketaddr, socklen_t).
EMFILE	Descriptor table not installed.
EPROTONOSUPPORT	Protocol not supported.

Include Files

<install_dir>\sciopta\<version>\include\sys\socket.h
<install_dir>\sciopta\<version>\include\sys\types.h

Function Code

```
int socket (int domain, int type, int proto)
{
   static const sc_msgid_t select[2] = { SCIO_SOCKET_REPLY, 0 };
   int tmo = 100;
   fd_tab_t *fd_tab;
   int i;
   sc_pid_t socket;

   if (sc_procVarGet (SCIO_PROCVAR_ID, (sc_var_t *) & fd_tab)) {
      if (fd_tab && fd_tab->magic == SCIO_MAGIC) {
         int fd;
         sdd_obj_t *protocol = NULL;

         if (domain == PF_INET && type == SOCK_DGRAM) {
            while (tmo < 10000 && !(protocol = ipv4_getProtocol ("udp"))) {
               sc_sleep (sc_tickMs2Tick (tmo));
               tmo *= 2;
            }
            if (!protocol) {
               sc_miscErrnoSet (EPROTONOSUPPORT);
               return -1;
            }
         }
         else if (domain == PF_INET && type == SOCK_STREAM) {
            i = 0;
            while (tmo < 10000 && !(protocol = ipv4_getProtocol ("tcp"))) {
               sc_sleep (sc_tickMs2Tick (tmo));
               tmo *= 2;
            }
            if (!protocol) {
               sc_miscErrnoSet (EPROTONOSUPPORT);
               return -1;
            }
         }
         else if (domain == PF_INET && type == SOCK_ICMP) {
            i = 0;
            while (tmo < 10000 && !(protocol = ipv4_getProtocol ("icmp"))) {
               sc_sleep (sc_tickMs2Tick (tmo));
               tmo *= 2;
            }
            if (!protocol) {
               sc_miscErrnoSet (EPROTONOSUPPORT);
               return -1;
            }
         }
```

```
        else {
           if (domain == PF_INET) {
              sc_miscErrnoSet (EPROTONOSUPPORT);
           }
           else {
              sc_miscErrnoSet (EINVAL);
           }
           return -1;
        }
        if (SDD_IS_A (protocol, SDD_DEV_TYPE)) {
           fd = 0;
           while (fd < fd_tab->max_fd && fd_tab->bf[fd]) {
              ++fd;
           }
           if (fd != fd_tab->max_fd) {
                 /* start a socket process */
              if ((socket =
                 sc_procTmpCreate (SCP_socket, "listener", 512,
                                              SC_DEFAULT_POOL)) == SC_ILLEGAL_PID) {
                   sc_miscErrnoSet (ENOMEM);
                   return -1;
              }
              if ((sdd_devOpen (protocol, proto)) != -1) {
                   /* send protocol to socket */
                 protocol->base.id = SCIO_SOCKET;
                 sc_msgTx ((union sc_msg **) &protocol, socket, 0);
                 protocol =(sdd_obj_t *) sc_msgRx (SC_ENDLESS_TMO, (void *) select,
                                                                SC_MSGRX_MSGID);
                 fd_tab->bf[fd] = protocol;
                 return fd;
              }
              else {
                 return -1;
              }
           }
           else {
              sc_miscErrnoSet (EMFILE);
              return -1;
           }
        }
        else {
           sc_miscErrnoSet (EINVAL);
           return -1;
        }
     }
  }
  sc_miscErrnoSet (EMFILE);
  return -1;
}
```

12 Request For Comments RFC in SCIOPTA IPS

In this chapter we have listed the important RFCs which are important for the SCIOPTA IPS implementation.

This list might not be complete and there is no guarantee that SCIOPTA IPS complies to all proposals, specifications, standards and information in these RFCs

12.1 Requests for Comments RFC - UDP

RFC	Description	Date
0768	User Datagram Protocol	August 1980

12.2 Requests for Comments RFC - ICMP

RFC	Description	Date
0792	Internet Control Message Protocol	September 1981
0950	Internet Standard Subnetting Procedure	August 1985

12.3 Requests for Comments RFC - IPv4

RFC	Description	Date
0791	Internet Protocol	September 1981
2474	Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers	December 1998
3168	The Addition of Explicit Congestion Notification (ECN) to IP	September 2001
3260	New Terminology and Clarifications for Diffserv	April 2002

12.4 Requests for Comments RFC - TCP

RFC	Description	Date
0793	Transmission Control Protocol	September 1981
0813	Window and Acknowledgement Strategy in TCP	July 1982
1323	TCP Extensions for High Performance	May 1992
3168	The Addition of Explicit Congestion Notification (ECN) to IP	September 2001
2581	TCP Congestion Control	April 1999
3390	Increasing TCP's Initial Window	October 2002

13 SCIOPTA ARM Releases

13.1 SCIOPTA ARM Release Notes

On the **SCIOPTA ARM** CD in the subfolder **doc** you will find a text file named **RelNotes_ARM.txt**

This file 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.

13.2 Installed Files

On the **SCIOPTA ARM** CD in the subfolder **doc** you will find a text file named **revisions.txt**

This file contains a list of all installed files including the following information for each file:

- File name
- SCIOPTA document number
- File version
- File description

14 Manual Versions

14.1 Manual Version 2.1

- Chapter 2 Installation, Main Installation Window screen shot changed.

14.2 Manual Version 2.0

- The following manuals have been combinded in this new SCIOPTA ARM - IPS Internet Protocols, User's Guide:
  - SCIOPTA - IPS Internet Protocols, User's Guide Version 1.4
  - SCIOPTA - IPS Internet Protocols, Function Interface Manual Version 1.5
  - SCIOPTA - ARM Target Manual

14.3 Former SCIOPTA - IPS Internet Protocols, User's Guide Versions

14.3.1 Manual Version 1.4

- All union sc_msg * changed to sc_msg_t to support SCIOPTA 16 Bit systems.
- All sdd_obj_t * changed to sdd_obj_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 sdd_netbuf_t * changed to sdd_netbuf_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.
- Chapter 4 Using SCIOPTA IPS, detailed description of SDD descriptors moved to chapter 5 Structures.
- Chapter 5 Structures, new chapter.
- Manual split into a User's Guide and a Function Interface part.

14.3.2 Manual Version 1.3

- Chapter 7.18 ipv4_aton, new function call.
- Chapter 7.22 ipv4_ntoa, new function call.

14.3.3 Manual Version 1.2

- Former chapter 4 "System Design" and chapter 7 IPS Function Interface Reference, functions and definitions starting with ips_ipv4Xxx changed to ipv4_xxx.
- Chapter 4.6.2 UDP Sending Using the IPS Message Interface, and chapter 4.6.4 TCP Sending Using the IPS Message Interface, message SDD_DEV_OPEN replaced by SDD_NET_OPEN.
- Chapter 6.13 SDD_NET_CLOSE/SDD_NET_CLOSE_REPLY, new message.

- Chapter 6.14 SDD_NET_OPEN/SDD_NET_OPEN_REPLY, new message.
- Chapter 7.6 ops_close, new function call.
- Chapter 7.14, ips_open, new function call.
- Chapter 7.17 ips_setOption, optname **SO_SC_SND_ACK** replace by **SO_SC_RET_BUF**. New optname **SO_SC_RET_ACK**.
- Chapter 8 BSD Socket Interface Reference, propriatary BSD calls ips_ack, ips_alloc, ips_getpeername and ips_send removed and socket calls recv, recvfrom, send and sendto now documented.
- Chapter 7.4 ips_alloc, new function call.
- Chapter 7.16 ips_send, new function call.
- Chapter 4.7.2 UDP Sending Using the IPS Function Interface, and chapter 4.7.4 TCP Sending Using the IPS Function Interface, ips_open() and ips_send() now used.
- Chapter 7.2 ips_accep, description: ips_open() now used.

14.3.4 Manual Version 1.1

- Former chapter 4.6.1 “Introduction” fifth paragraph rewritten.
- Chapter 8.2 accept, new function code.
- Chapter 8.21 socket, new function code.

14.3.5 Manual Version 1.0

Initial version.

14.4 Former SCIOPTA - IPS Internet Protocols, Function Interface Versions

14.4.1 Manual Version 1.5

- Chapter 3.12 ips_getOption and chapter 3.16 ips_setOption, optname **SO_SC_SND_ACK** replace by **SO_SC_RET_BUF**, new optname **SO_SC_RET_ACK**.
- Chapter 4.12 getsockopt and chapter 4.20 setsockopt, optname **SO_SC_SND_ACK** replace by **SO_SC_RET_BUF**, new optname **SO_SC_RET_ACK**.

14.4.2 Manual Version 1.4

- All union sc_msg * changed to sc_msg_t to support SCIOPTA 16 Bit systems.
- All sdd_obj_t * changed to sdd_obj_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 sdd_netbuf_t * changed to sdd_netbuf_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.
- Chapter 4 Using SCIOPTA IPS, detailed description of SDD descriptors moved to chapter 5 Structures.
- Chapter 5 Structures, new chapter.
- Manual split into a User's Guide and a Function Interface Manual part.

14.4.3 Manual Version 1.3

- Chapter 7.18 ipv4_aton, new function call.
- Chapter 7.22 ipv4_ntoa, new function call.

14.4.4 Manual Version 1.2

- Former chapter 4 "System Design" and chapter 7 IPS Function Interface Reference, functions and definitions starting with ips_ipv4Xxx changed to ipv4_xxx.
- Chapter 4.6.2 UDP Sending Using the IPS Message Interface, and chapter 4.6.4 TCP Sending Using the IPS Message Interface, message SDD_DEV_OPEN replaced by SDD_NET_OPEN.
- Chapter 6.13 SDD_NET_CLOSE/SDD_NET_CLOSE_REPLY, new message.
- Chapter 6.14 SDD_NET_OPEN/SDD_NET_OPEN_REPLY, new message.
- Chapter 7.6 ops_close, new function call.
- Chapter 7.14, ips_open, new function call.
- Chapter 7.17 ips_setOption, optname **SO_SC_SND_ACK** replace by **SO_SC_RET_BUF**. New optname **SO_SC_RET_ACK**.
- Chapter 8 BSD Socket Interface Reference, propriatary BSD calls ips_ack, ips_alloc, ips_getpeername and ips_send removed and socket calls recv, recvfrom, send and sendto now documented.

- Chapter 7.4 ips_alloc, new function call.
- Chapter 7.16 ips_send, new function call.
- Chapter 4.7.2 UDP Sending Using the IPS Function Interface, and chapter 4.7.4 TCP Sending Using the IPS Function Interface, ips_open() and ips_send() now used.
- Chapter 7.2 ips_accep, description: ips_open() now used.

14.4.5 Manual Version 1.1

- Former chapter 4.6.1 “Introduction” fifth paragraph rewritten.
- Chapter 8.2 accept, new function code.
- Chapter 8.21 socket, new function code.

14.4.6 Manual Version 1.0

Initial version.

14.5 Former SCIOPTA ARM - Target Manual Versions

14.5.1 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.

14.5.2 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.

14.5.3 Manual Version 2.0

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

14.5.4 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).

14.5.5 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.

15 Index