SCIOPTA

High Performance
Real-Time Operating Systems

ARM
IPS
Internet Protocols
Applications

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. S06017RL1

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. For this IPS TCP/IP stack there is a number of specific TCP/IP applications available, such as Web Server, TFTP, DNS, DHCP, Telnet, SMTP etc.

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

For each application there is a separate chapter including each an introduction into the techniques and concepts and information about the interfaces to access the IPS application functionality. Furthermore you will find useful information about system design and configuration.

Please consult also the SCIOPTA ARM - Kernel, User’s Guide and the SCIOPTA IPS Internet protocols, User’s Guide.

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.

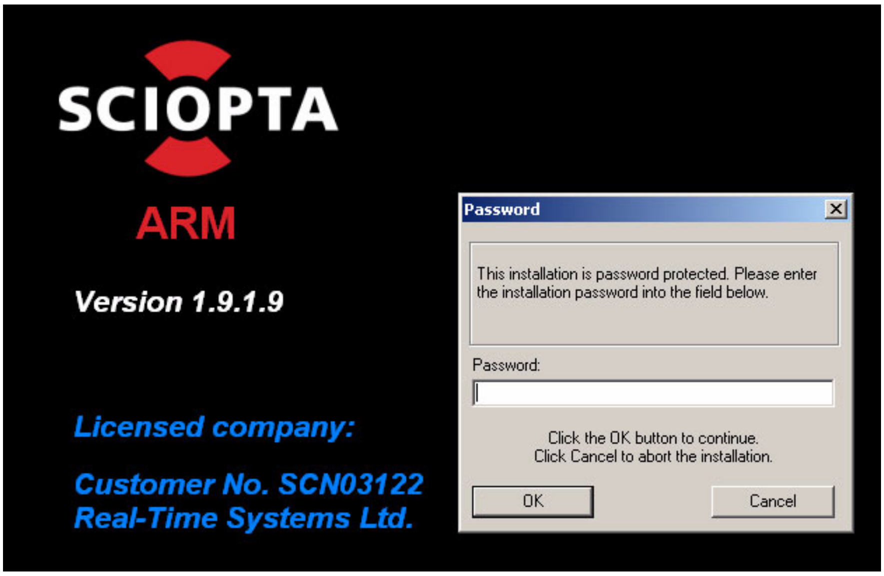

Figure 2-1: Main Installation Window

3 Getting Started

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.

You can find the detailed step-by-step getting started tutorials in the “Getting Started” chapters of the specific IPS Applications descriptions of this manual.

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, fec.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.

4 Web Server

4.1 Description

The SCIOPTA web server is used to supply data from embedded systems to host web browsers by using standardised protocols and languages. The web server can also be used to get set-up data from a browser and to configure the embedded system accordingly.

The web server is a repository for web pages in the target embedded system. It handles request from host web browsers and is responsible for the data transfer between browser and the web server.

The HTTP server process (**SCP_http**) will dynamically be created and killed by the HTTP daemon process (SCP_httpd). The HTTP daemon and the page processes are static processes. For every web page there is a page process.

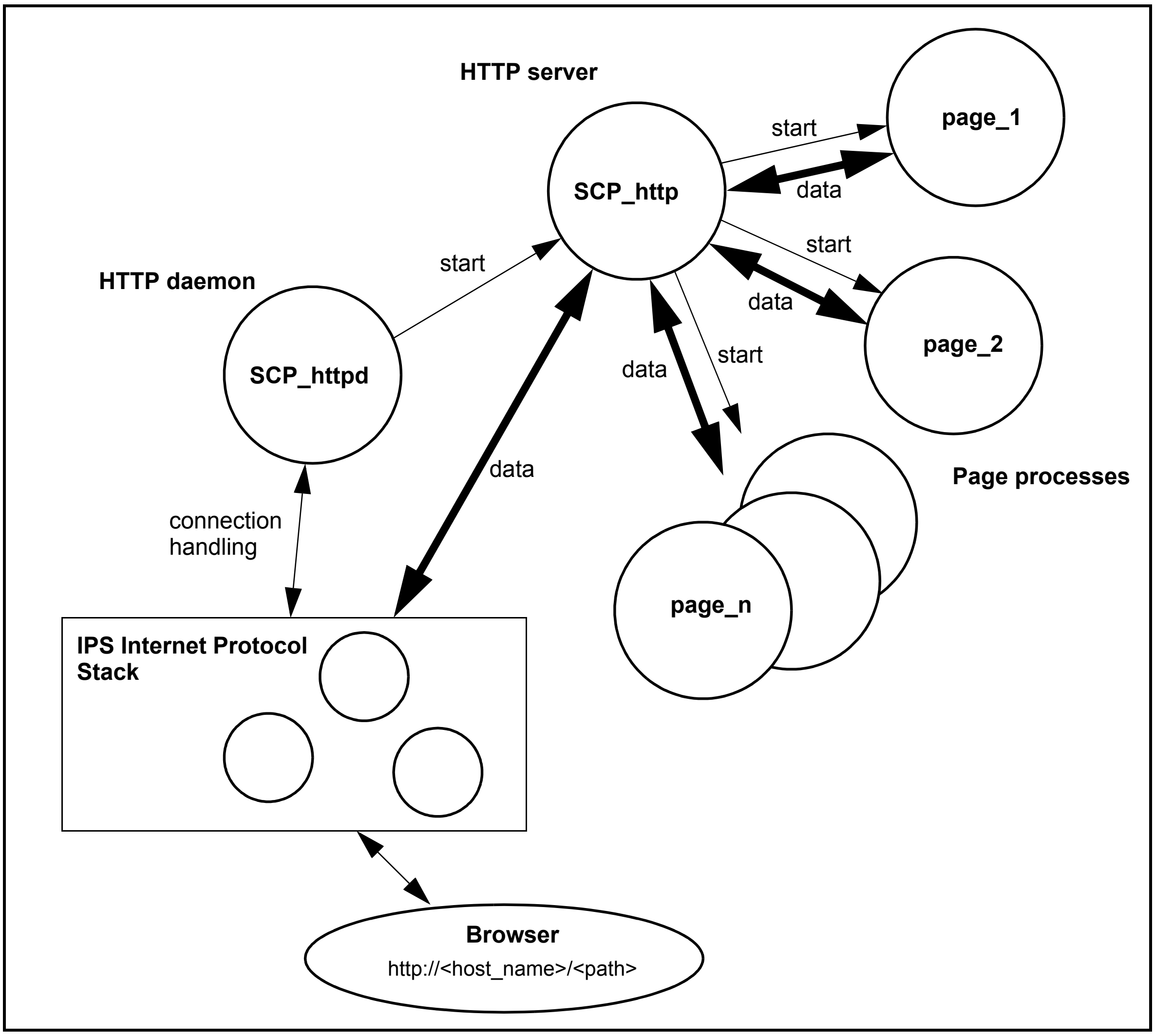

Figure 4-1: SCIOPTA IPS Web Server Structure

At start-up the HTTP daemon performs an open to the IPS stack and with the returned protocol descriptor will do an ips_bind, an ips_listen and waits on an ips_accept.

To start a web session the browser requests an URL in the form

http://<**host_name**>/<**path**>.

The **host_name** is the name or IP address where the IPS TCP/IP protocol stack is located.

The **path** is the full path to the process containing the page content. The HTTP server will break down **path** into a process path and a remainder path. The page process can again break down the remainder path to further selecting the target (e.g. the remainder path can be a file path). Additionally a **document-root** will be added to the **path**. The **document-root** is defined by the HTTP process at configuration. For instance a "/" starts the query in the system module while a "/user/" starts the query in the module user. Please consult the SCIOPTA Kernel User's Guide and Reference Manual for more information about SCIOPTA modules.

The browser tries to connect to the IPS stack and if it is successful the HTTP daemon returns from the ips_accept including the protocol descriptor. The HTTP daemon now creates the HTTP server process (SCP_https) and starts it by sending a start message containing the protocol descriptor.

The first action of the HTTP server process is to receive the browser request. It will retrieve the URL and depending on the specified page process name the HTTP server will send a start message to the corresponding page process. The page process sends back a reply message containing the next page date request (line query, path, version, etc.). The last reply message contains the information "**no more data**". The HTTP server receives data from the client (browser) and sends (partly) the first HTTP header.

The page process sends the HTML code to the HTTP server process. The HTTP server encapsulates the HTML code in the HTTP protocol and returns it via the IPS stack to the browser.

The communication will normally be closed by the browser and this will automatically kill the HTTP server process (SCP_https).

4.2 Getting Started Web Server Example

4.2.1 Description

This small web server example returns a web page including “hello world” messages.

To start a web session the browser needs to requests an URL in the form: **http://<host_name>/<path>**.

The **host_name** is the name or IP address where the IPS TCP/IP protocol stack is located. The **path** is the full path to the process containing the page content. Please consult the “SCIOPTA - IPS Applications” manual for more information.

The browser tries to connect to the IPS stack and if it is successful the HTTP daemon returns including the protocol descriptor. The HTTP daemon now creates the HTTP server process (**SCP_http**) and starts it by sending a start message containing the protocol descriptor.

The first action of the HTTP server process is to receive the browser request. It will retrieve the URL and depending on the specified CGI name the HTTP server will send a start message to the corresponding CGI process. The CGI process sends back a start reply message containing the size of the HTML page.The CGI process sends the HTML code to the HTTP server process. The HTTP server encapsulates the HTML code in the HTTP protocol and returns it via the IPS stack to the browser.

The communication will normally be closed by the browser and this will automatically kill the HTTP server process (**SCP_http**).

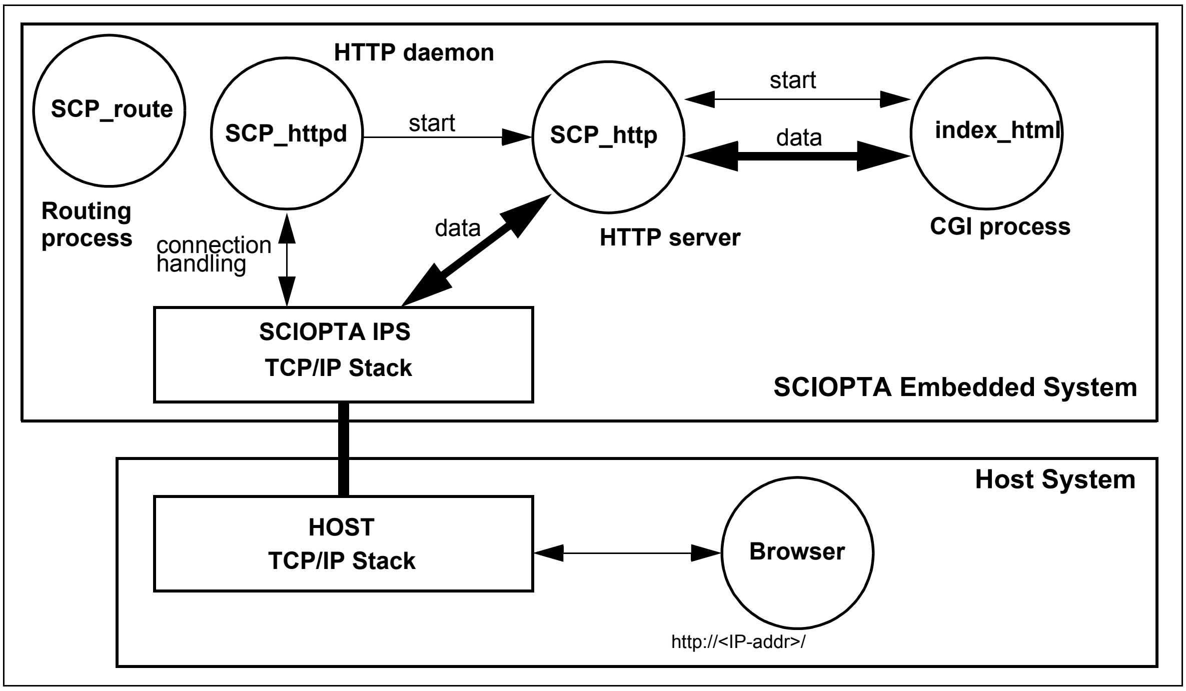

Figure 4-2: Getting Started Web Server Example

SCIOPTA

4.2.2 SCIOPTA IPS Web Server - Windows Host

4.2.2.1 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\http\
- 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
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

4.2.2.2 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_http**) 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_http).
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\http\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_http).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_http).
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_http).
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_http**) 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_http\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. Launch a web browser on the host computer.
31. Enter the target URL (http://<target IP address>) in the browser:
32. Look at the target web page displayed in your browser.
33. You can also set breakpoints in the web server application and watch the behaviour.

4.3 Web Server Configuration

4.3.1 Introduction

SCIOPTA Web Server is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use the Web Server. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User’s Guide for more information about the IPS stack and how to configure it.

Web Server 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 - Target Manual for your selected processor for more information about using sconf.exe and the static configuration process.

   The static object will be generated and started automatically at system start.
2. SCIOPTA Web Server configuration which configures the http daemon process (**SCP_httpd**).

4.3.2 Web Server Module Configuration

In our standard getting started examples 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.

The web server is placed in the “user” module. But you are free to split your application differently into other modules or put all processes in one module.

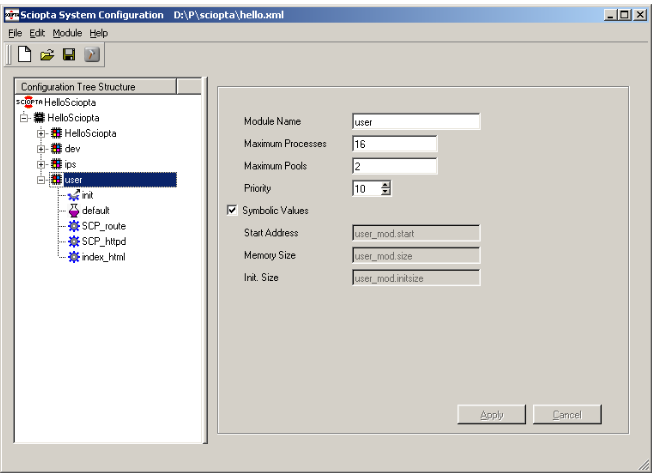

Figure 4-3: Web Server module “user”

4.3.2.1 Web Server 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 256 bytes. You could optimize this stack by doing a stack analysis using the DRUID system level debugger.

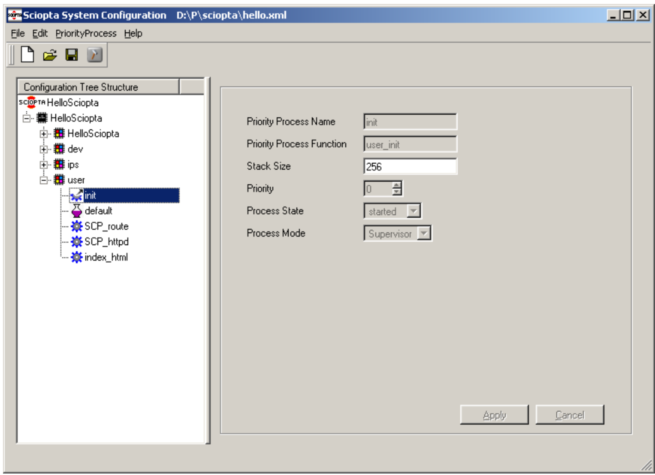

Figure 4-4: Web Server module init process

4.3.2.2 Web Server Module Default Pool

The pool parameters must be designed to fit the requirements of the web server messages buffer sizes.

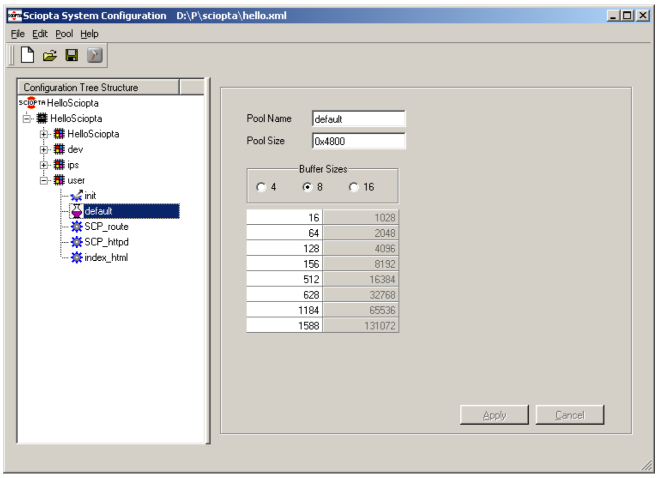

Figure 4-5: Web Server module default pool

4.3.2.3 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.

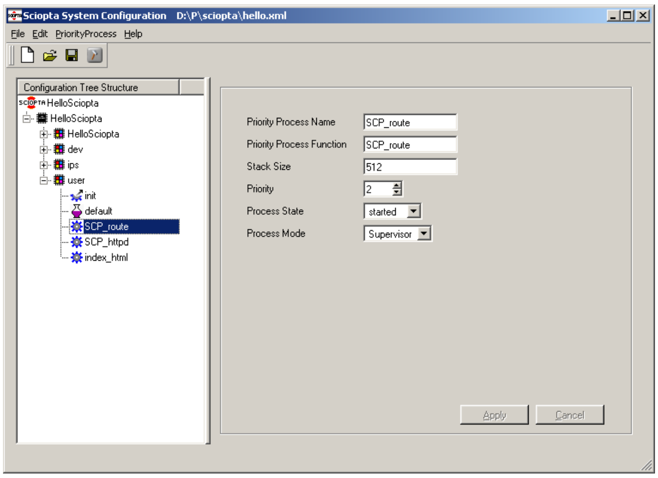

Figure 4-6: Routing process SCP_route

4.3.2.4 HTTP Daemon Process

The HTTP daemon is the main managing process in the web server. It handles the connection with the IPS stack and creates and starts the web server process **SCP_http**.

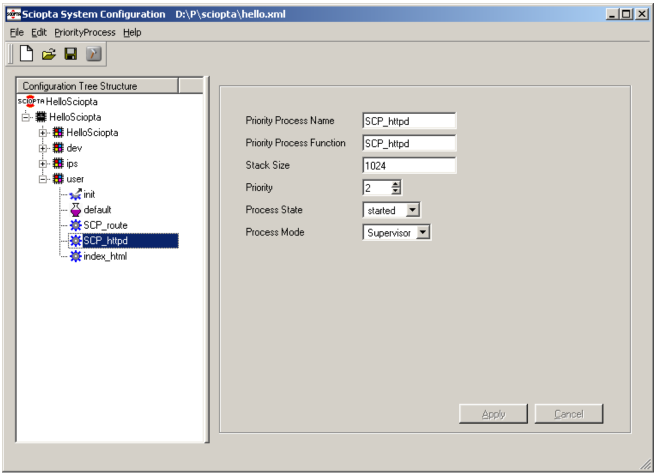

Figure 4-7: Web Server HTTP daemon process SCP_httpd

4.3.2.5 Page Processes

For each web page you need to declare a page process.

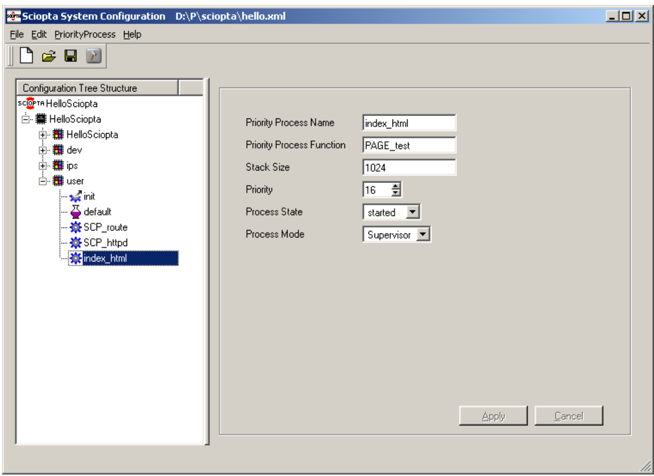

Figure 4-8: Web Server page process

4.3.3 Web Server Configuration

In the previous chapters we have described how to configure the static modules, pools and processes for SCIOPTA Web Server. 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 Web Server you need also to configure some web server parameters depending on the web server specified properties. These parameters are defined by calling the http daemon process at start-up (**SCP_httpd**).

In the SCIOPTA IPS Web Server example we have include the declaration of this processes in the file **httpsetup.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\common\

4.3.3.1 HTTP Daemon Configuration

You need to declare a http daemon initialization process **SCP_httpd**. The only function of this process is to call the internal http daemon function **http_daemon** including the web server parameter configuration parameter.

Syntax http_daemon

```
void http_daemon (const http_config_t *defaultConfig);
```

Parameters

defaultConfig	Default web server configuration. Please consult chapter **4.6.1 “HTTP Configuration Structure http_config_t” on page 4-17** for type information.

Example:

```
/**
 * Configuration for HTTP daemon process
 */
static const http_config_t conf = {
  /* Message id */           0,
  /* Bound to IP address */  { 0, { 0, } },
  /* Bound to Port */        80,
  /* Document root module */ "/user/",
  /* default html page */    "index_html",
  /* Max URI length */       300,
  /* Max Request length */   500,
  /* Max Connections */        3
};
/**
 * HTTP daemon process definition
 */
SC_PROCESS (SCP_httpd)
{
  http_daemon (&conf);
  sc_procKill (SC_CURRENT_PID, 0);
}
```

4.4 Web Server System Building

Please consult chapters “System Building” of the SCIOPTA ARM - Kernel, User’s Guide and SCIOPTA ARM - IPS Internet Protocols, User’s Guide for general information about the system building process.

You will find there information about:

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

4.4.1 Include Files

No specific include files search directories for IPS Web Server needs to be declared. Please consult chapter “Include Files” of the SCIOPTA ARM - Kernel, User’s Guide for all information about include files.

4.4.2 SCIOPTA ARM IPS Web Server Libraries

4.4.2.1 Delivered IPS Web Server Libraries

IPS Web Server	libhttp_**XY**.a	GCC
	http_**XY**.l	ARM RealView
	http_**XY**.r79	IAR Systems

4.4.2.2 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.

4.4.2.3 IPS Web Server Libraries “Y”

For SCIOPTA IPS Web Server 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.

Please consult chapter ‘IPS Internet Protocols Libraries “Y” ‘ of the SCIOPTA ARM - IPS Internet Protocols, User’s Guide for a table showing the included features for each of the three IPS levels.

4.5 Using Web Server

4.5.1 Page Processes

For each addressed web page you need to write a separate page process. No other processes needs to be written by the user. The HTTP daemon process (**SCP_httpd**) and the HTTP server process (SCP_https) are included in the delivery.

The page process needs to do the following:

1. Receive the start message **HTTP_PAGE_START**. This will start the procedure.
2. Reply to the HTTP server process by sending an **HTTP_PAGE_REPLY** message.
3. Get a new instance of an HTTP object by using the **http_new()** function.
4. Writing the web page content by using the **http_printf()** function.
5. Send an empty netbuf (eof) to the HTTP server to terminate the data transfer.
6. Release and free the HTTP object instance.

4.5.2 Example

```
#include <sciopta.h>
#include <string.h>
#include <sdd/sdd.msg>
#include <sdd/sdd.h>
#include <ips/connect.h>
#include <http/printf.h>
#include <http/page.msg>
#include <http/server.h>

union sc_msg {
  sc_msgid_t id;
  sdd_netbuf_t netbuf;
  ips_ack_t ack;
  http_pageData_t pageData;
  http_pageReply_t pageReply;
};

SC_PROCESS (PAGE_test)
{
  sdd_obj_t httpd;
  sc_msg_t msg, url, query, remoteAddr, scriptName;
  http_t *page;
  sc_pid_t to;
  int i, counter;

  counter = 0;
  static const sc_msgid_t select[2] = { HTTP_PAGE_START, 0 };
  static sc_msgid_t next[2] = { 0, 0 };

  memset((char *)&httpd,0,sizeof(httpd));
```

```
for (;;) {
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) select, SC_MSGRX_MSGID);
  to = sc_msgSndGet (&msg);
  switch (msg->id) {

  case HTTP_PAGE_START:

    /**
     * Start CGI
     */
    ++counter;
    httpd.sender = httpd.receiver = httpd.controller = to;
    sc_msgFree (&msg);

    /**
     * want path info
     */
    msg = sc_msgAllocClr (sizeof (http_pageReply_t), HTTP_PAGE_REPLY,
                          SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
    /* set more to path info */
    msg->pageReply.more = HTTP_PAGE_PATH_INFO;
    sc_msgTx (&msg, to, 0);
    /* only want to receive requested msg id */
    next[0] = HTTP_PAGE_PATH_INFO;
    url = sc_msgRx (SC_ENDLESS_TMO, (void *) next, SC_MSGRX_MSGID);

    /**
     * want query
     */
    msg = sc_msgAllocClr (sizeof (http_pageReply_t), HTTP_PAGE_REPLY,
                          SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
    /* set more to query string */
    msg->pageReply.more = HTTP_PAGE_QUERY_STRING;
    sc_msgTx (&msg, to, 0);
    /* only want to receive requested msg id */
    next[0] = HTTP_PAGE_QUERY_STRING;
    query = sc_msgRx (SC_ENDLESS_TMO, (void *) next, SC_MSGRX_MSGID);

    /**
     * want remote addr
     */
    msg = sc_msgAllocClr (sizeof (http_pageReply_t), HTTP_PAGE_REPLY,
                          SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
    /* set more to remote addr */
    msg->pageReply.more = HTTP_PAGE_REMOTE_ADDR;
    sc_msgTx (&msg, to, 0);
    /* only want to receive requested msg id */
    next[0] = HTTP_PAGE_REMOTE_ADDR;
    remoteAddr = sc_msgRx (SC_ENDLESS_TMO, (void *) next, SC_MSGRX_MSGID);

    /**
     * want script name
     */
    msg = sc_msgAllocClr (sizeof (http_pageReply_t), HTTP_PAGE_REPLY,
                          SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
    /* set more to script name */
    msg->pageReply.more = HTTP_PAGE_SCRIPT_NAME;
    sc_msgTx (&msg, to, 0);
    /* only want to receive requested msg id */
    next[0] = HTTP_PAGE_SCRIPT_NAME;
    scriptName = sc_msgRx (SC_ENDLESS_TMO, (void *) next, SC_MSGRX_MSGID);
```

```
/**
 * have engouh infos for my simple cgi
 */
msg = sc_msgAllocClr (sizeof (http_pageReply_t), HTTP_PAGE_REPLY,
                  SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
sc_msgTx (&msg, to, 0);

/**
 * check for data stage - terminates with a null buf
 */
next[0] = SDD_NET_RECEIVE;
msg = sc_msgRx (SC_ENDLESS_TMO, (void *) next, SC_MSGRX_MSGID);
while (SDD_NET_DATA_SIZE (&msg->netbuf)) {
  ips_ack (&httpd, &msg->netbuf);
  sc_msgFree (&msg);
  msg = sc_msgRx (SC_ENDLESS_TMO, (void *) next, SC_MSGRX_MSGID);
}
sc_msgFree (&msg);

/**
 * start with my page now
 */
page = http_new (&httpd, SC_DEFAULT_POOL);

/**
 * write http header parts
 */
http_printf (page, "HTTP/1.1 200 OK\r\n");
http_printf (page, "Content-Type: text/html\r\n");
http_eof (page);

/**
 * write your content
 */
http_printf (page, "<html><body><h1>Hello world</h1>\n");
http_printf (page, "visit: %d<br>\n", counter);
http_printf (page, "url: %s<br>\n", url->pageData.data);
http_printf (page, "query: %s<br>\n", query->pageData.data);
http_printf (page, "remote address: %s<br>\n", remoteAddr->pageData.data);
http_printf (page, "script name: %s<br>\n", scriptName->pageData.data);

/* ok do not need url any more */
sc_msgFree (&url);

/* ok do not need query any more */
sc_msgFree (&query);

/* ok do not need remoteAddr any more */
sc_msgFree (&remoteAddr);
```

```
      /* ok do not need scriptName any more */
      sc_msgFree (&scriptName);

      http_printf (page, "system time: %d<br>\n", sc_tickGet());
      for (i = 0; i < 100; i++) {
        http_printf (page,
             "Line %d: And again \"hello world\"<BR>\n",i);
      }
      http_printf (page, "</html></body>");

      /* send a eof */
      http_eof (page);
      http_free (&page);
      break;

    default:
      sc_miscError (1, 0);
      break;
    }
  }
}
```

4.6 Structures

4.6.1 HTTP Configuration Structure http_config_t

The HTTP configuration structure is used in a message to configure the HTTP daemon.

It contains the message ID as it is a standard SCIOPTA message.

```
typedef struct http_config_s {
   sc_msgid_t          id;
   ips_addr_t          addr;
   __u16               port;
   char                *documentRoot;
   int                 httpURIMax;
   int                 httpREQMax;
   __uint          connMax;
} http_config_t;
```

Members

id

Standard SCIOPTA message ID.

addr

Bound IP address. Please consult chapter 5 (Structures) of the SCIOPTA IPS Internet Protocols User's Guide for information about the isp_addr_t structure.

port

Bound port.

documentRoot

Pointer to the **document-root string**. A **document-root** will be added to the **path**. For instance a "/" starts the query in the system module while a "/user/" starts the query in the module user. Please consult the SCIOPTA Kernel User's Guide and Reference Manual for more information about SCIOPTA modules.

httpURIMax

Maximum number of Unified Resource Identifiers (URI).

httpREQMax

Maximum request length in bytes.

Header

<install_dir>\sciopta\<version>\include\http\server.h

4.6.2 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**).

4.7 Web Server Message Interface

4.7.1 HTTP_PAGE_AUTH_TYPE

This message is used to get the authorization type. See also section 11 of the HTTP/1.1 specification.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_AUTH_TYPE** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_AUTH_TYPE**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.2 HTTP_PAGE_CONTENT_LENGTH

This message is used to get the length of a HTTP request message body (e.g. POST, PUT).

The HTTP server (SCP_https) process sends an **HTTP_PAGE_CONTENT_LENGTH** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_CONTENT_LENGTH**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.3 HTTP_PAGE_CONTENT_TYPE

This message is used to get the media type. See also section 3.7 of the HTTP/1.1 specification.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_CONTENT_TYPE** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_CONTENT_TYPE**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t         id;
   char               data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.4 HTTP_PAGE_GATEWAY_INTERFACE

This message is used to get the CGI version. See also CGI/1.1.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_GATEWAY_INTERFACE** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_GATEWAY_INTERFACE**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.5 HTTP_PAGE_HTTP_ENTRY

This message is used to get a not specified entry (in the CGI) in the HTTP header.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_HTTP_ENTRY** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_HTTP_ENTRY**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data. In the request message data contains the requested entry name while in the reply message this member contains the result.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.6 HTTP_PAGE_HTTP_HEADER

This message is used to get the whole HTTP header.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_HTTP_HEADER** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_HTTP_HEADER**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.7 HTTP_PAGE_PATH_INFO

This message is used to get the rest of the URL after PATH to the CGI.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_PATH_INFO** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_PATH_INFO**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.8 HTTP_PAGE_QUERY_STRING

This message is used to get the request query (e.g. form data).

The HTTP server (SCP_https) process sends an **HTTP_PAGE_QUERY_STRING** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_QUERY_STRING**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.9 HTTP_PAGE_REMOTE_ADDR

This message is used to get the address of the client sending the request.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_REMOTE_ADDR** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_REMOTE_ADDR**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

SCIOPTA

4.7.10 HTTP_PAGE_REPLY

This message is used to reply to a **HTTP_PAGE_START** message. To save memory space this message should be sent as soon as possible before the first **http_printf()** function call.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_START** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message **HTTP_PAGE_REPLY**

http_pageReply_t Structure

```
typedef struct http_pageReply_s {
   sc_msgid_t         id;
   sc_msgid_t         more;
   char               data[1];
} http_pageReply_t;
```

Members

id

Standard SCIOPTA message ID.

more

SCIOPTA message ID representing the next HTTP meta variable message containing meta data. If **more** contains zero, no more meta data are requested.

data

Null terminated string with the additional data to get next information.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.11 HTTP_PAGE_REQUEST_METHOD

This message is used to get the request mode (GET, HEAD, POST etc.). See also section 5.1.1 of the HTTP/1.1 specification.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_REQUEST_METHOD** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_REQUEST_METHOD**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.12 HTTP_PAGE_SCRIPT_NAME

This message is used to get the script name (here the CGI page process name).

The HTTP server (SCP_https) process sends an **HTTP_PAGE_SCRIPT_NAME** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_SCRIPT_NAME**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.13 HTTP_PAGE_SERVER_NAME

This message is used to get the name of the server.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_SERVER_NAME** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_SERVER_NAME**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t        id;
   char              data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.14 HTTP_PAGE_SERVER_PORT

This message is used to get the port on which the server is bound (e.g. port 80).

The HTTP server (SCP_https) process sends an **HTTP_PAGE_SERVER_PORT** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_SERVER_PORT**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t         id;
   char               data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.15 HTTP_PAGE_SERVER_PROTOCOL

This message is used to get the server protocol (e.g. HTTP/1.1).

The HTTP server (SCP_https) process sends an **HTTP_PAGE_SERVER_PROTOCOL** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_SERVER_PROTOCOL**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.16 HTTP_PAGE_SERVER_SOFTWARE

This message is used to get the server software name (e.g. sciopta web server/1.0.0).

The HTTP server (SCP_https) process sends an **HTTP_PAGE_SERVER_SOFTWARE** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_SERVER_SOFTWARE**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.7.17 HTTP_PAGE_START

This message is used to start a page process. The page process represents one web page in the web server system.

The HTTP server (SCP_https) process sends an **HTTP_PAGE_START** message to a page process. The page process sends an **HTTP_PAGE_REPLY** reply message back

Message ID

Request message	**HTTP_PAGE_START**

http_pageDate_t Structure

```
typedef struct http_pageData_s {
   sc_msgid_t          id;
   char                data[1];
} http_pageData_t;
```

Members

id

Standard SCIOPTA message ID.

data

Null terminated string with the requested data.

Header

<install_dir>\sciopta\<version>\include\http\page.msg

4.8 Web Server Function Interface

4.8.1 http_eof

This method sends an empty netbuf to the HTTP server process. This informs the HTTP server process that the cgi has either terminated the HTTP header or terminated the data transfer.

```
void http_eof(
   http_t NEARPTR     self
);
```

Parameter

self

SDD object descriptor of the SDD http object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

Return Value

None.

Header

<install_dir>\sciopta\<version>\include\http\printf.h

4.8.2 http_free

This method is used to release and free an HTTP object.

```
void http_free(
   http_t NEARPTR NEARPTR     self
);
```

Parameter

self

SDD object descriptor of the SDD http object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information. Please note the **pointer to pointer** type.

Return Value

None.

Header

<install_dir>\sciopta\<version>\include\http\printf.h

4.8.3 http_new

This method is used to get a new instance of an HTTP object for a specific connection defined in an SDD object descriptor.

```
http_t http_new(
   sdd_obj_t NEARPTR    conn,
   sc_poolid_t          plid
);
```

Parameters

conn

SDD object (SDD connection descriptor). For cgi the sender must be the web server pid, all other entries are not needed. Please consult chapter 4.6.2 “NEARPTR and FARPTR” on page 4-18 for type information.

plid

Pool ID from where the memory buffer of the HTTP object will be allocated. This buffer contains also an network buffer (netbuf).

Return Value

Pointer to a new instance of an HTTP object.

Header

<install_dir>\sciopta\<version>\include\http\printf.h

4.8.4 http_printf

This method is used to perform a formatted print (printf) redirected to the HTTP server process. It is used inside a page process.

```
void http_printf(
   http_t NEARPTR        self,
   const char            *fmt,
   ...
);
```

Parameter

self

SDD object descriptor of the SDD http object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

fmt

Formatted string.

...

All parameters requested in the formatted string **fmt**.

Return Value

Void.

Header

<install_dir>\sciopta\<version>\include\http\printf.h

4.9 Requests for Comments RFC - HTTP

In this chapter we have listed the important RFCs which are important for web server HTTP implementation.

This list might not be complete and there is no guarantee that the SCIOPTA web server complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
1945	Hypertext Transfer Protocol -- HTTP/1.0	May 1996
2039	Applicability of Standards Track MIBs to Management of World Wide Web Servers	November 1996
2145	Use and Interpretation of HTTP Version Numbers	May 1997
2169	A Trivial Convention for using HTTP in URN Resolution	June 1997
2186	Internet Cache Protocol (ICP), version 2	September 1997
2227	Simple Hit-Metering and Usage-Limiting for HTTP	October 1997
2295	Transparent Content Negotiation in HTTP	March 1998
2296	HTTP Remote Variant Selection Algorithm -- RVSA/1.0	March 1998
2310	The Safe Response Header Field	April 1998
2518	HTTP Extensions for Distributed Authoring -- WEBDAV	February 1999
2585	Internet X.509 Public Key Infrastructure Protocols: FTP and HTTP	May 1999
2616	Hypertext Transfer Protocol -- HTTP/1.1	June 1999
2617	HTTP Authentication: Basic and Digest Access Authentication	June 1999
2660	The Secure HyperText Transfer Protocol	August 1999
2774	An HTTP Extension Framework	February 2000
2817	Upgrading to TLS Within HTTP/1.1	May 2000
2818	HTTP Over TLS	May 2000
2831	Using Digest Authentication as a SASL Mechanism	May 2000
2935	Internet Open Trading Protocol (IOTP) HTTP Supplement	September 2000
2936	HTTP MIME Type Handler Detection	September 2000
2964	Use of HTTP State Management	October 2000
2965	HTTP State Management Mechanism	October 2000
3143	Known HTTP Proxy/Caching Problems	June 2001
3205	On the use of HTTP as a Substrate	Februar 2002
3229	Delta encoding in HTTP	January 2002
3230	Instance Digests in HTTP	January 2002
3310	Hypertext Transfer Protocol (HTTP) Digest Authentication	September 2002
3507	Internet Content Adaptation Protocol (ICAP)	April 2003

4.10 Requests for Comments RFC - HTML

In this chapter we have listed the important RFCs which are important for web server HTML implementation.

This list might not be complete and there is no guarantee that the SCIOPTA web server complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
2557	MIME Encapsulation of Aggregate Documents, HTML (MHTML)	March 1999
2659	Security Extensions For HTML	August 1999
2731	Encoding Dublin Core Metadata in HTML	December 1999
2854	The 'text/html' Media Type	June 2000

4.11 Internet Draft

draft-coar-cgi-v11-04

The Common Gateway Interface (CGI) is a simple interface for running external programs, software or gateways under an information server in a platform-independent manner. Currently, the supported information servers are HTTP servers.

The interface has been in use by the World-Wide Web since 1993. This specification defines the 'current practice' parameters of the 'CGI/1.1' interface developed and documented at the U.S. National Centre for Supercomputing Applications. This document also defines the use of the CGI/1.1 interface on UNIX(R) and other, similar systems.

5 SMTP Simple Mail Transfer Protocols

5.1 Description

SMTP, Simple Mail Transfer Protocol, is a protocol for sending e-mail messages between servers.

Actually the SMTP client side is implemented. There is not much use of an SMTP **server** in an embedded system.

Usually SMTP client in an embedded system is used to send emails to a remote SMTP server. This can be implemented by designing single SCIOPTA process which is using the SMTP function interface.

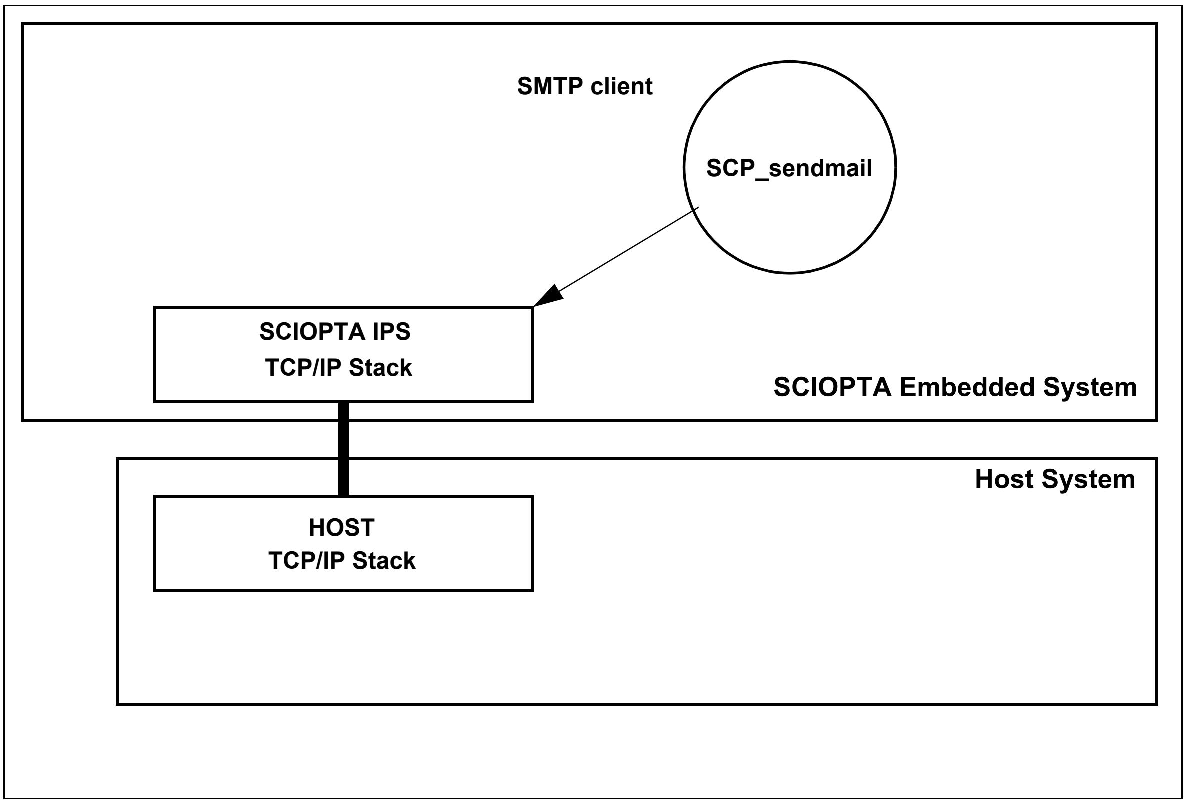

Figure 5-1: SCIOPTA IPS Simple Mail Transfer Protocol Structure

5.2 Getting Started - SMTP Example

5.2.1 Description

SMTP, Simple Mail Transfer Protocol, a protocol for sending e-mail messages between servers.

In this simple getting started example for the SCIOPTA SMTP there is a process (**SCP_sendmail**) which is sending an email to a server. The IP address of the host must be known.

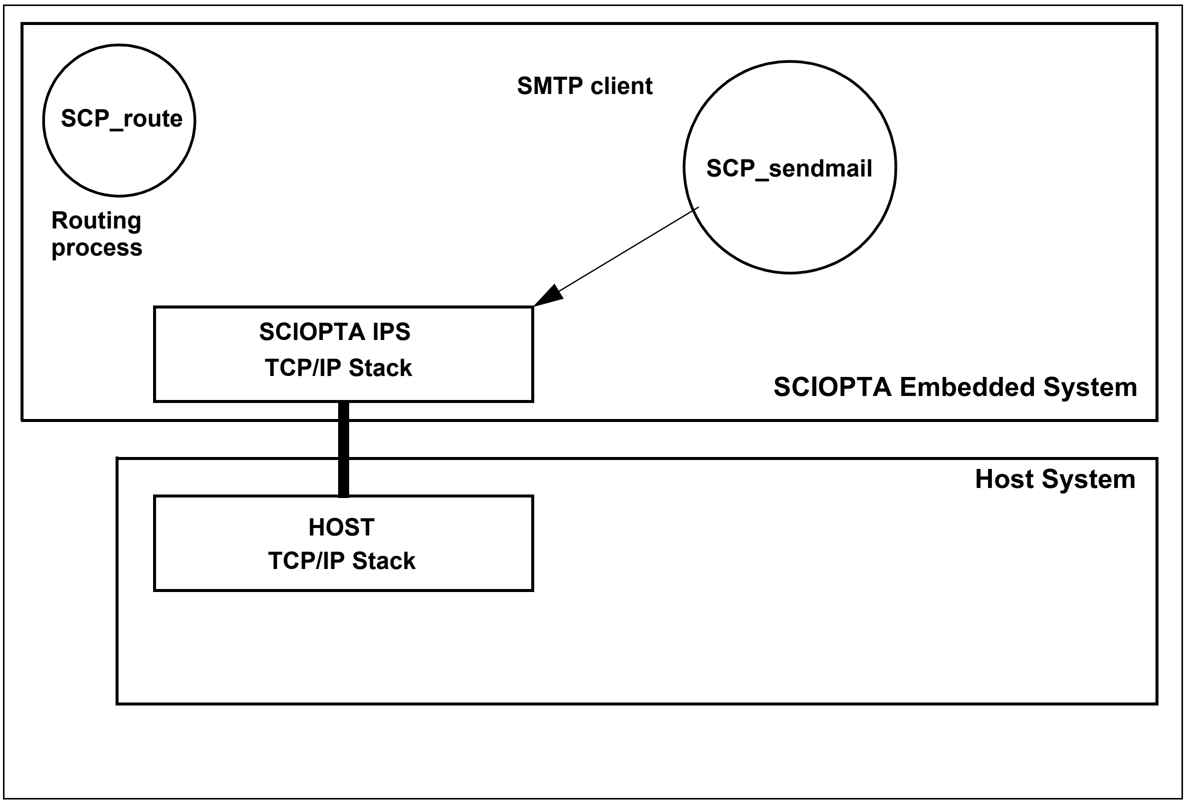

Figure 5-2: Getting Started SMTP Client Example

5.2.2 IPS SMTP Example - Windows Host

5.2.2.1 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\http\
- Network cable connected between your target board and your host workstation or to your network.
- Internet Email server with known IP address.
- Optionally a network protocol analyser running on your host computer such as Ethereal.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

5.2.2.2 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_smtp**) 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_smtp).
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\smtp\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_smtp).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_smtp).
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_smtp).
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_smtp**) 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_smtp\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. Check on your mail server if the mail defined in the file mailclient.c has arrived.
31. You can also set breakpoints in the target smtp application and watch the behaviour.
32. For have a closer look at the network traffic generated by this example you can run a network analyser such as Ethereal on your host computer and trace the network messages.

5.3 SMTP Configuration

SCIOPTA SMTP is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use SMTP. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User's Guide for more information about the IPS stack and how to configure it.

There is no configuration needed for setting up an SMTP client. No specific processes or daemons needs to be defined, configured and started.

If emails need to be sent from a SCIOPTA embedded system the SMTP function interface will be used by SCIOPTA processes. Therefore configuration is limited to setting up standard SCIOPTA processes.

5.4 SMTP System Building

Please consult chapters "System Building" of the SCIOPTA ARM - Kernel, User's Guide and SCIOPTA ARM - IPS Internet Protocols, User's Guide for general information about the system building process.

You will find there information about:

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

5.4.1 Include Files

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

5.4.2 SCIOPTA ARM IPS SMTP Libraries

5.4.2.1 Delivered IPS SMTP Libraries

IPS SMTP	libsmtp_**XY**.a	GCC
	smtp_**XY**.l	ARM RealView
	smpt_**XY**.r79	IAR Systems

5.4.2.2 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.

5.4.2.3 IPS SMTP Libraries “Y”

For SCIOPTA IPS SMTP 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.

Please consult chapter ‘IPS Internet Protocols Libraries “Y” ‘ of the SCIOPTA ARM - IPS Internet Protocols, User’s Guide for a table showing the included features for each of the three IPS levels.

5.5 Using SMTP

5.5.1 SMTP Process

A process sending emails using SMTP needs to do the following:

1. Wait for a mail host SMTP server route.
2. Connect to the remote SMTP server using the smtp_connect() function.
3. Start the email session by setting the sender and the receiver email address. This is done using the smtp_from() and smtp_to() function.
4. Define the start of the email content by using the smt_begin() function.
5. Write the email header by using smtp_printf() functions.
6. Write the email body by using smtp_printf() functions.
7. Define the end of the email content by using the smt_end() function.
8. Stop the email session and close the connection to the mailhost by using the smtp_close() function.

5.6 Example

```
#include <sciopta.h>
#include <ips/addr.h>
#include <ips/router.h>
#include <mail/smtp.h>

static const ips_addr_t mailhost = { 4, { 10, 0, 1, 135 } };
#define MAIL_PORT                  25
#define MAIL_TIMEOUT               12000 /* ms */

#define MAIL_FROM                  "device@esystem.com"
#define MAIL_TO                    "alarm@company.com"
```

```
SC_PROCESS (SCP_sendmail)
{
  smtp_t *smtp;

  /* wait for mail host route
   */
  ipv4_routeWait ((char *)mailhost.addr, SC_ENDLESS_TMO);

  /* connect to mailhost
   */
  smtp = smtp_connect ((ips_addr_t *)&mailhost, MAIL_PORT, "", SC_DEFAULT_POOL,
                       MAIL_TIMEOUT);

  if ( !smtp ){
    sc_miscError(0x11,0);
    sc_procKill(SC_CURRENT_PID,0);
  }
  /* start mail session
   */
  smtp_from (smtp, MAIL_FROM);
  smtp_to (smtp, MAIL_TO);
  /* start data
   */
  smtp_begin (smtp);
  /* mail header
   */
  smtp_printf (smtp, "From: " MAIL_FROM "\r\n");
  smtp_printf (smtp, "To: " MAIL_TO "\r\n");
  smtp_printf (smtp, "Subject: Test test\r\n");
  /* mail body
   */
  smtp_printf (smtp, "This is a test\r\n");
  smtp_printf (smtp, "A dot on a new line\r\n");
  smtp_printf (smtp, ".\r\n");
  smtp_printf (smtp, "A dotdot on a new line\r\n");
  smtp_printf (smtp, "..\r\n");
  smtp_printf (smtp, "A dot with following text\r\n");
  smtp_printf (smtp, ". This is a new text after a dot \r\n");
  /* end the mail
   */
  smtp_end (smtp);

  /* close the connection to the mailhost
   */
  smtp_close (&smtp);

  /* kill this process
   */
  sc_procKill (SC_CURRENT_PID, 0);
}
```

5.7 SMTP Function Interface

5.7.1 smtp_bcc

This method is used to specify the email address of an addressee to send a blind carbon copy.

If there are more than one addressee to send a blind carbon copy just call this function for every addressee.

```
int smtp_bcc(
   smtp_t NEARPTR        self,
   const char            *bcc
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

bcc

Email address of the blind carbon copy addressee.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.2 smtp_begin

This method is used to define the start of the email content.

```
int smtp_begin(
   smtp_t NEARPTR        self
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.3 smtp_cc

This method is used to specify the email address of an addressee to send a carbon copy.

If there are more than one addressee to send a carbon copy just call this function for every addressee.

```
int smtp_cc(
   smtp_t NEARPTR       self,
   const char           *cc
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

cc

Email address of the carbon copy addressee.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.4 smtp_close

This method is used to close the smtp session. This will close the connection to the email host and free the email descriptor (message).

```
void smtp_close(
   smtp_t NEARPTR NEARPTR       self
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information. Please note the **pointer to pointer** type.

Return Value

None

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.5 smtp_connect

This method is used to connect to an SMTP server.

```
smtp_t NEARPTR smtp_connect(
   ips_addr_t          *addr,
   __u16               port,
   const char          *domain,
   sc_poolid_t         plid,
   __u32               tmo
);
```

Parameter

addr

Address of the SMTP server. Please consult chapter 5 (Structures) of the SCIOPTA IPS Internet Protocols User's Guide and Reference Manual for information about the isp_addr_t structure. Please consult chapter **4.6.2 "NEARPTR and FARPTR" on page 4-18** for type information.

port

Port number (usually 25).

domain

Domain of the SMTP server.

plid

ID of the message pool from where the mail descriptor (message) will be allocated.

tmo

Timeout for waiting on a reply.

Return Value

Pointer to a new instance of an smtp descriptor for the email. Please consult chapter **4.6.2 "NEARPTR and FARPTR" on page 4-18** for type information.

A return value of **NULL** indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

SCIOPTA

5.7.6 smtp_end

This method is used to define the end of the email content.

```
int smtp_end(
   smtp_t NEARPTR        self
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.7 smtp_from

This method is used to define the sender address of the email.

If there are more than one sender to define just call this function for every sender.

```
int smtp_from(
   smtp_t NEARPTR        self,
   const char            *from
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

from

Address of the sender.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.8 smtp_printf

This method is used to add a line of text in the email by using the well known printf function.

If this function is used to send an SMTP command you have to get the SMTP reply code by using the smtp_waitOk() function. This is not needed if a dot (".") command is sent on a new line (will be escaped).

```
int smtp_printf(
   smtp_t NEARPTR          self,
   const char              *fmt,
   ...
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 "NEARPTR and FARPTR" on page 4-18** for type information.

fmt

Formatted string.

...

All parameters requested in the formatted string **fmt**.

Return Value

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Returns the number of character written on success.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.9 smtp_status

This method is used to get the status of the mail host in plain text.

```
char *smtp_status(
   smtp_t NEARPTR          self
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

Return Value

Returns a pointer to the character string which includes the description of the last status of the mail host.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.10 smtp_to

This method is used to define the email address of the email addressee.

If there are more than one addressee to define just call this function for every addressee.

```
int smtp_to(
   smtp_t NEARPTR          self,
   const char              *to
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 "NEARPTR and FARPTR" on page 4-18** for type information.

to

Address of the addressee.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 "SMTP Reply Codes" on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.11 smtp_waitOk

This method is used to wait on a response from the SMTP server.

If you have included a command in an smtp_printf function you need to wait on a reply from the SMTP server.

```
int smtp_waitOk(
   smtp_t NEARPTR          self
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.7.12 smtp_end

This method is used to define the end of the email content.

```
int smtp_end(
   smtp_t NEARPTR          self
);
```

Parameter

self

SDD object descriptor of the SDD smtp object. Please consult chapter **4.6.2 “NEARPTR and FARPTR” on page 4-18** for type information.

Return Value

A return value of 0 or higher indicates a successful operation. The return value corresponds to the SMPT reply code (see chapter **5.8 “SMTP Reply Codes” on page 5-21**).

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\mail\smtp.h

5.8 SMTP Reply Codes

This is just a short list of the SMTP reply codes. Please consult RFC2821 for more information.

.

Code	Description
211	System status, or system help reply
214	Help message
220	<domain> Service ready
221	<domain> Service closing transmission channel
250	Requested mail action okay, completed
251	User not local; will forward to <forward-path>
252	Cannot VRFY user, but will accept message and attempt delivery
354	Start mail input; end with <CRLF>.<CRLF>
421	<domain> Service not available, closing transmission channel
450	Requested mail action not taken: mailbox unavailable
451	Requested action aborted: local error in processing
452	Requested action not taken: insufficient system storage
500	Syntax error, command unrecognized
501	Syntax error in parameters or arguments
502	Command not implemented
503	Bad sequence of commands
504	Command parameter not implemented
550	Requested action not taken: mailbox unavailable
551	User not local; please try <forward-path>
552	Requested mail action aborted: exceeded storage allocation
553	Requested action not taken: mailbox name not allowed
554	Transaction failed (Or, in the case of a connection-opening response, "No SMTP service here")

5.9 Requests for Comments RFC - SMTP

In this chapter we have listed the important RFCs which are important for the SMTP implementation.

This list might not be complete and there is no guarantee that SCIOPTA SMTP complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
0876	Survey of SMTP implementations	September 1983
1047	Duplicate messages and SMTP	February 1988
1090	SMTP on X.25	February 1989
1428	Transition of Internet Mail from Just-Send-8 to 8bit-SMTP/MIME	February 1993
1652	SMTP Service Extension for 8bit-MIMEtransport	July 1994
1845	SMTP Service Extension for Checkpoint/Restart	September 1995
1846	SMTP 521 Reply Code	September 1995
1870	SMTP Service Extension for Message Size Declaration	November 1995
1985	SMTP Service Extension for Remote Message Queue Starting	August 1996
2033	Local Mail Transfer Protocol	October 1996
2034	SMTP Service Extension for Returning Enhanced Error Codes	October 1996
2442	The Batch SMTP Media Type	November 1998
2476	Message Submission	December 1998
2554	SMTP Service Extension for Authentication	March 1999
2645	ON-DEMAND MAIL RELAY (ODMR) SMTP with Dynamic IP Addresses	August 1999
2821	Simple Mail Transfer Protocol	April 2001
2852	Deliver By SMTP Service Extension	June 2000
2920	SMTP Service Extension for Command Pipelining	September 2000
3030	SMTP Service Extensions for Transmission of Large and Binary MIME Messages	December 2000
3207	SMTP Service Extension for Secure SMTP over Transport Layer Security	February 2002
3461	Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs)	January 2003
3463	Enhanced Mail System Status Codes	January 2003
3848	ESMTP and LMTP Transmission Types Registration	July 2004

6 TFTP Trivial File Transfer Protocols

6.1 Description

TFTP is a simple protocol to transfer files between machines on different networks implementing UDP.

TFTP client and TFTP server is implemented in SCIOPTA IPS.

Usually TFTP client in an embedded system is used to write a file or a memory content to a remote TFTP server or to receive a file from the remote TFTP server and to store it in the embedded system. This can be implemented by designing single SCIOPTA process which is using the TFTP function interface. The process waits on a route to the TFTP server and sends the file to the TFTP server on the host system.

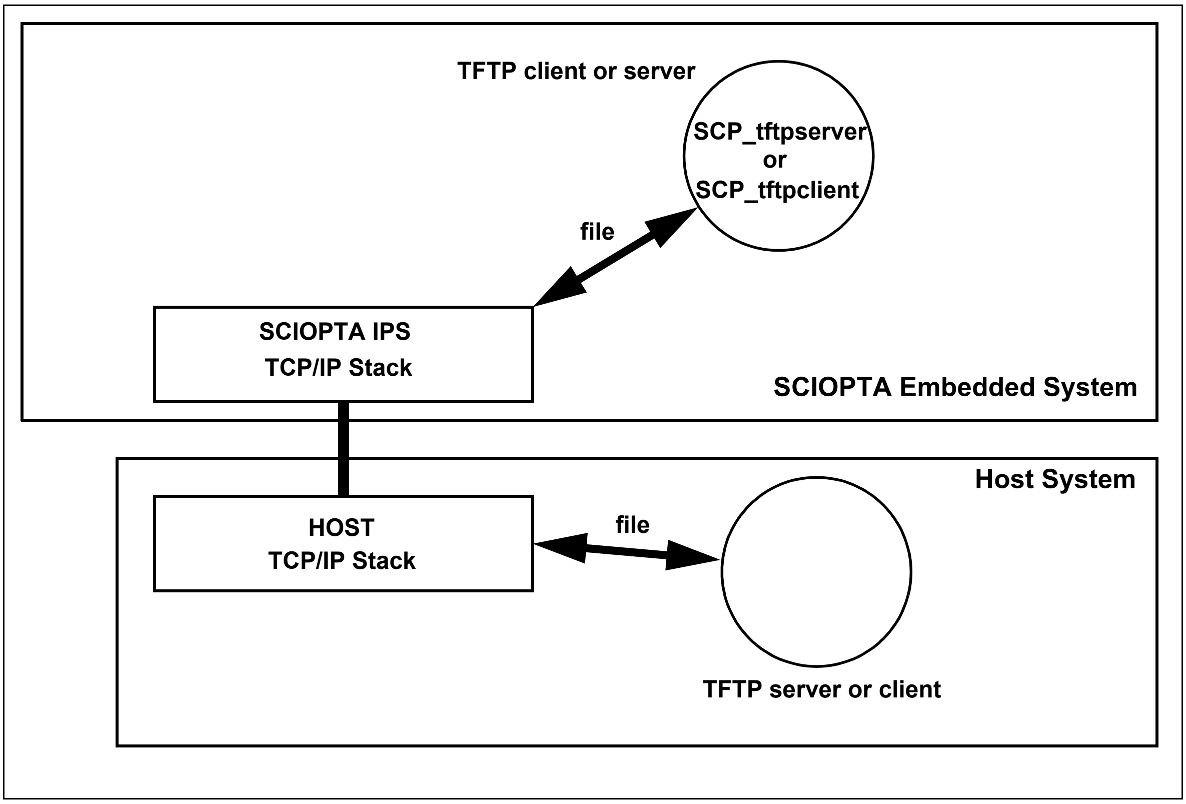

Figure 6-1: SCIOPTA IPS Trivial Files Transfer Protocol Structure

6.2 Getting Started - TFTP Client Example

6.2.1 Description

TFTP is a simple protocol to transfer files between machines on different networks implementing UDP.

In this simple getting started example for the SCIOPTA TFTP client product we will write a file to a remote server.

The TFTP client example consists of a single SCIOPTA process which is using the TFTP function interface. The process waits on a route to the TFTP server and sends the file **test.bin** to the TFTP server on the host system. After a successful transfer the process will kill itself.

The user can check a successful transfer by tracking down the file test.bin.

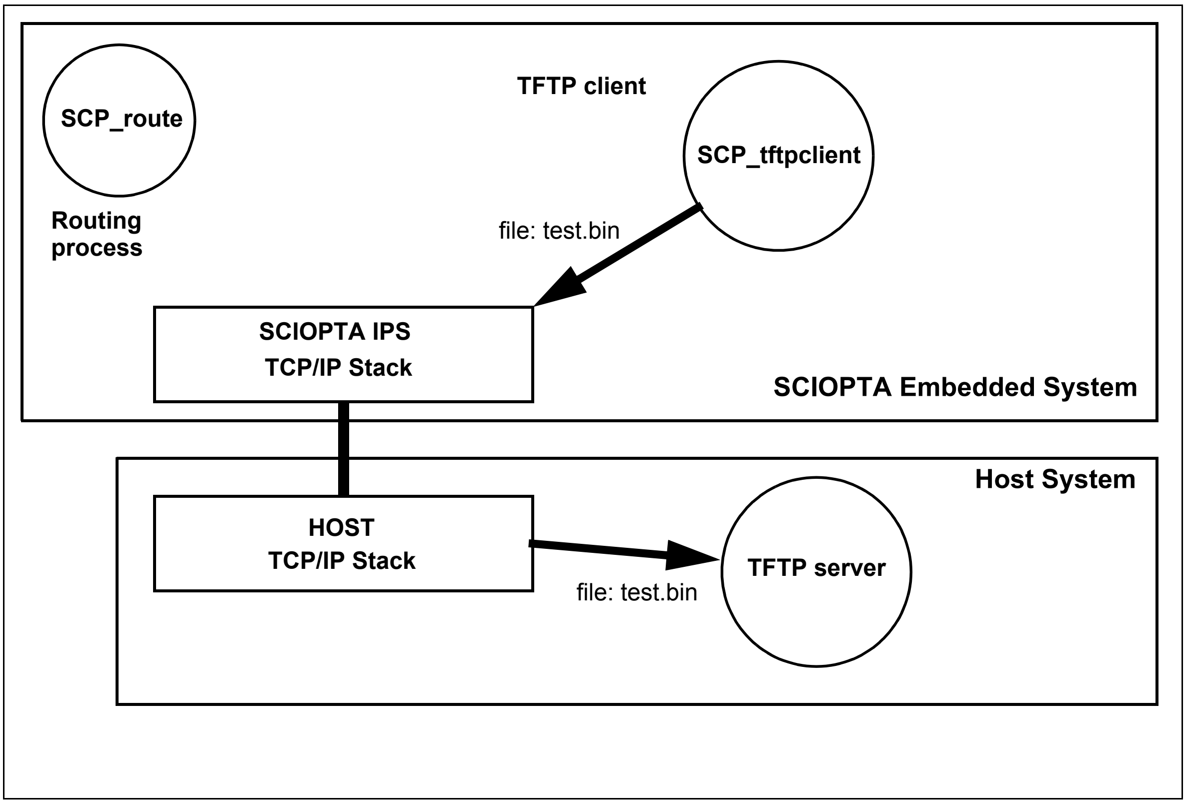

Figure 6-2: Getting Started TFTP Client Example

6.2.2 IPS TFTP Client Example - Windows Host

6.2.2.1 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\tftp\
- Network cable connected between your target board and your host workstation or to your network.
- A standard TFTP server program such as TFTP Daemon32 running on your host computer.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

6.2.2.2 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_tftp**) 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_tftp).
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\tftp\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_tftp).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_tftp).
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_tftp) 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_tftp).
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_tftp**) 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_tftp\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. Check that the file **test.bin** has been transferred and stored at the specified location.
31. You can also set breakpoints in the target tftp client application and watch the behaviour.

6.3 Getting Started - TFTP Server Example

6.3.1 Description

TFTP is a simple protocol to transfer files between machines on different networks implementing UDP.

In this simple getting started example for the SCIOPTA TFTP server product we will write a file to an embedded system server.

The TFTP server example consists of a single SCIOPTA process which is using the TFTP function interface. The process waits on a route to the TFTP server and receives the file **test.bin** from the TFTP client on the host system.

The user can check a successful transfer by tracking down the file test.bin.

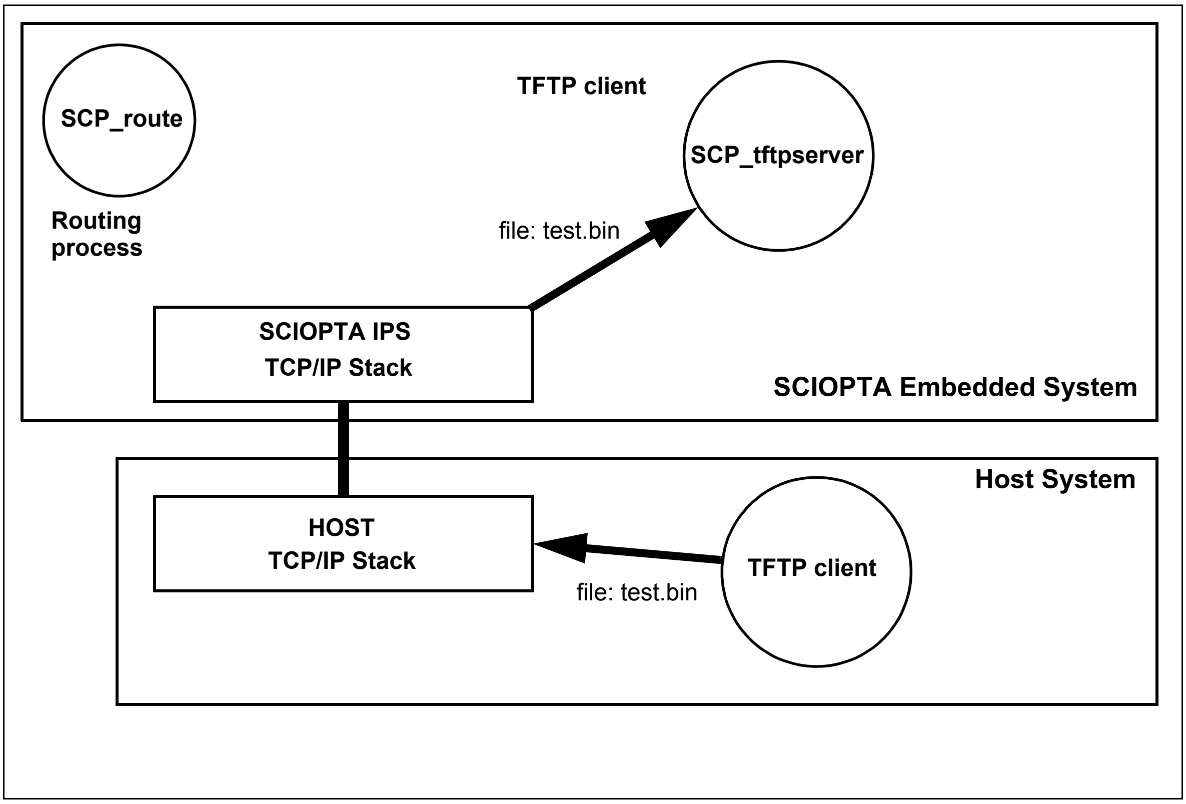

Figure 6-3: Getting Started TFTP Server Example

6.3.2 IPS TFTP Server Example - Windows Host

6.3.2.1 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\tftpserver\
- Network cable connected between your target board and your host workstation or to your network.
- A standard TFTP client program running on your host computer.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

6.3.2.2 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_tftpserver**) 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_tftpserver).
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\tftpserver\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_tftpserver).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_tftpserver).
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_tftpserver) 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_tftpserver).
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_tftpserver**) 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_tftpserver\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. Check that the file **test.bin** has been transferred and stored at the specified location.
30. You can also set breakpoints in the target tftp server application and watch the behaviour.

6.4 TFTP Configuration

SCIOPTA TFTP is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use TFTP. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User's Guide for more information about the IPS stack and how to configure it.

There is no configuration needed for setting up an TFTP client. No specific processes or daemons needs to be defined, configured and started.

If files or a memory content need to be sent or received in a SCIOPTA embedded system the TFTP function interface will be used by SCIOPTA processes. Therefore configuration is limited to setting up standard SCIOPTA processes. Please consult the SCIOPTA kernel and target manuals for more information about configuring processes.

6.5 TFTP System Building

Please consult chapters "System Building" of the SCIOPTA ARM - Kernel, User's Guide and SCIOPTA ARM - IPS Internet Protocols, User's Guide for general information about the system building process.

You will find there information about:

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

6.5.1 Include Files

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

6.5.2 SCIOPTA ARM IPS TFTP Libraries

6.5.2.1 Delivered IPS TFTP Libraries

IPS TFTP	libtftp_**XY**.a	GCC
	tftp_**XY**.l	ARM RealView
	tftp_**XY**.r79	IAR Systems

6.5.2.2 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.

6.5.2.3 IPS TFTP Libraries “Y”

For SCIOPTA IPS TFTP 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.

Please consult chapter ‘IPS Internet Protocols Libraries “Y” ‘ of the SCIOPTA ARM - IPS Internet Protocols, User’s Guide for a table showing the included features for each of the three IPS levels.

6.6 Using TFTP

6.6.1 TFTP Process

A process transferring a file (or memory content) from the embedded system using TFTP client to the host computer running a TFTP server needs to do the following:

1. Wait for a host TFTP server route.
2. Connect to the remote TFTP server using the tftp_connect() function.
3. Transfer the file or memory location by using the tftp_putMem() function.
4. Close the connection to the TFTP host server using the tftp_close() function.

A process receiving a file from the host computer running a TFTP server and store it in the embedded system using TFTP client needs to do the following:

1. Wait for a host TFTP server route.
2. Connect to the remote TFTP server using the tftp_connect() function.
3. Get the file and store it in the embedded system by using the tftp_get2Mem() function.
4. Close the connection to the TFTP host server using the tftp_close() function.

6.7 Example

```
#include <sciopta.h>
#include <ips/addr.h>
#include <ips/router.h>
#include <tftp/tftp.h>

#include <string.h> /* memset */

#define TFTP_PORT        69
#ifndef TFTP_SERVER
#define TFTP_SERVER      10,0,1,135
#endif

ips_addr_t tftp_server = { 4, { TFTP_SERVER } };

char test_bin[ 4049 ];
```

```
/* TFTP client setup */

SC_PROCESS (SCP_tftpclient)
{
  tftp_handle_t *h;
  int error;
  int i;

  for(i = 0; i < sizeof(test_bin) ; ++i){
    test_bin[i] = (i & 63)+32;
  }

  /* Wait route to TFTP server  */
  ipv4_routeWait (tftp_server.addr, SC_ENDLESS_TMO);

  /* connect on TFTP server */
  h = tftp_connect (&tftp_server, TFTP_PORT, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  if ( h ){
    /* put test.bin */
    error = tftp_putMem (h, test_bin, sizeof(test_bin), "test.bin");
    /* close connection */
    tftp_close(&h);

    if ( error < 0 ){
      /* ups, give IPS some time to finish ... */
      sc_sleep(100);
      sc_miscError(0x11,sc_miscErrnoGet());
    }
  }

  /* Clear memory so we can check if read was succesfull */
  memset(test_bin,0,sizeof(test_bin));

  /* connect on TFTP server */
  h = tftp_connect (&tftp_server, TFTP_PORT, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

  if ( h ){
    /* get test.bin */
    error = tftp_get2Mem (h, test_bin, sizeof(test_bin), "test.bin");

    /* close connection */
    tftp_close(&h);

    if ( error < 0 ){
      /* ups, give IPS some time to finish ... */
      sc_sleep(100);
      sc_miscError(0x12,sc_miscErrnoGet());
    }
  }

  /* kill this process */
  sc_procKill (SC_CURRENT_PID, 0);
}
```

6.8 TFTP Function Interface

6.8.1 tftp_accept

This method is used to accept a connection (the first packet arrived). You need to call **tftp_listen()** before calling **tftp_accept()**.

This function acts as a server.

```
tftp_handle_t NEARPTR tftp_accept(
  tftp_handle_t NEARPTR          self,
  __u8                           *opcode,
  char                           *file,
  int                            len
);
```

Parameter

self

SDD object descriptor of the SDD tftp object (requested tftp connection).

opcode

Request code of the TFTP server side.

This parameter can be one of the following values:

Value	Meaning
TFTP_RRQ	TFTP read request
TFTP_WRQ	TFTP write request

file

Requested file name.

len

Length of the file string to avoid range checks.

Return Value

Pointer to the tftp descriptor of a new instance of an accepted connection.

A return value of **NULL** indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.2 tftp_close

This method is used to close an TFTP connection and free the TFTP descriptor (message).

```
void tftp_close(
   tftp_handle_t NEARPTR NEARPTR      self
);
```

Parameter

self

SDD object descriptor of the SDD tftp object. Please note the **pointer to pointer** type.

Return Value

None

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.3 tftp_connect

This method is used to connect to a TFTP server.

This function acts as a client.

```
tftp_handle_t NEARPTR tftp_connect(
   ips_addr_t                    *addr,
   int                           port,
   sc_poolid_t                   plid,
   sc_ticks_t                    tmo
);
```

Parameter

addr

Address of the TFTP server. Please consult chapter 5 (Structures) of the SCIOPTA IPS Internet Protocols User's Guide and Reference Manual for information about the isp_addr_t structure.

port

Port of the TFTP server.

plid

ID of the message pool from where the TFTP descriptor (message) will be allocated.

tmo

Maximum time the function is waiting to get the TFTP descriptor from the defined message pool.

Return Value

Pointer to the tftp descriptor of a new instance of a connection.

A return value of **NULL** indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.4 tftp_get2File

This method is used to get a file from a TFTP server and store it in the SCIOPTA file system.

This function only acts as a client.

```
int tftp_get2File(
   tftp_handle_t NEARPTR        self,
   sdd_obj_t NEARPTR            dev,
   const char                   *from
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

dev

SDD network device descriptor of an opened device with write access.

from

Name of the file on the remote TFTP server.

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.5 tftp_get2Mem

This method is used to get a file from a TFTP server and store it in the memory of the local embedded system.

This function only acts as a client.

```
int tftp_get2Mem(
   tftp_handle_t NEARPTR         self,
   __u8                          *start,
   __u32                         len,
   const char                    *from
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

start

Start address of the memory where the file content will be stored.

len

Size of the memory.

from

Name of the file on the remote TFTP server.

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.6 tftp_listen

This method is used to listen on a specified port for a TFTP connection (first packet).

This function acts as a server.

```
tftp_handle_t NEARPTR tftp_listen(
   int                             port,
   sc_poolid_t                     plid,
   sc_ticks_t                      tmo
);
```

Parameter

port

Port on which TFTP should listen for a connection (first packet).

plid

ID of the message pool from where the TFTP descriptor (message) will be allocated.

tmo

Maximum time to wait for packets on the established connection.

Return Value

Pointer to the tftp descriptor of a new instance of a connection. Please consult chapter for type information.

A return value of **NULL** indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.7 tftp_mode

This method is used to set the mode of the TFTP client.

This function only acts as a client.

```
int tftp_mode(
   tftp_handle_t NEARPTR        self,
   const char                   *mode
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

mode

TFTP mode.

This parameter can be one of the following values:

Value	Meaning
“ascii”	ASCII mode
“binary”	Binary mode

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.8 tftp_putFile

This method is used to put a file from the SCIOPTA file system to the remote TFTP server.

This function only acts as a client.

```
int tftp_putFile(
   tftp_handle_t NEARPTR         self,
   sdd_obj_t NEARPTR             dev,
   const char                    *from
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

dev

SDD network device descriptor of an opened device with read access.

from

Name of the file on the remote TFTP server.

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.9 tftp_putMem

This method is used to put the content of a memory area of the local embedded system to the TFTP server.

This function only acts as a client.

```
int tftp_get2Mem(
   tftp_handle_t NEARPTR     self,
   __u8                      *start,
   __u32                     len,
   const char                *to
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

start

Start address of the memory.

len

Size of the memory.

to

Name of the file on the remote TFTP server.

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.10 tftp_recv2Mem

This method is used to get a file from a TFTP client and store it in a memory area.

This function only acts as a server.

```
int tftp_rec2Mem(
   tftp_handle_t NEARPTR        self,
   __u8                         *mem,
   __u32                        len
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

mem

Start of the memory area.

len

Length of the memory.

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.8.11 tftp_sendMem

This method is used to send the content of a memory area to the client.

This function only acts as a server.

```
int tftp_sendMem(
   tftp_handle_t NEARPTR        self,
   __u8                         *mem,
   __u32                        len
);
```

Parameter

self

SDD object descriptor of the SDD tftp object.

mem

Start of the memory area.

len

Length of the memory to send.

Return Value

A return value of 0 or higher indicates a successful operation.

A return value of -1 indicates an error condition. The error code can be read by using the **sc_miscErrnoGet()** function call.

Header

<install_dir>\sciopta\<version>\include\tftp\tftp.h

6.9 Requests for Comments RFC - TFTP

In this chapter we have listed the important RFCs which are important for the TFTP implementation.

This list might not be complete and there is no guarantee that SCIOPTA TFTP complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
0906	Bootstrap loading using TFTP	June 1984
1350	The TFTP Protocol (Revision 2)	July 1992
1785	TFTP Option Negotiation Analysis	March 1995
1986	Experiments with a Simple File Transfer Protocol for Radio Links using Enhanced Trivial File Transfer Protocol (ETFTP)	August 1996
2090	TFTP Multicast Option	February 1997
2347	TFTP Option Extension	May 1998
2348	TFTP Blocksize Option	May 1998
2349	TFTP Timeout Interval and Transfer Size Options	May 1998
3617	Uniform Resource Identifier (URI) Scheme and Applicability Statement for the Trivial File Transfer Protocol (TFTP)	October 2003

7 DNS Domain Name System

7.1 Description

DNS Domain Name System (or Service or Server) is a service in a TCP/IP network such as SCIOPTA IPS that translates domain names into IP addresses.

A typical DNS user process requiring the DNS service from the SCIOPTA resolver process is using specific DNS messages such as **DNS_IP_REQUEST_MSG** and **DNS_NAME_REQUEST_MSG**.

The DNS resolver process is replying to DNS request by scanning its own name-server list (res_nameServers[]).

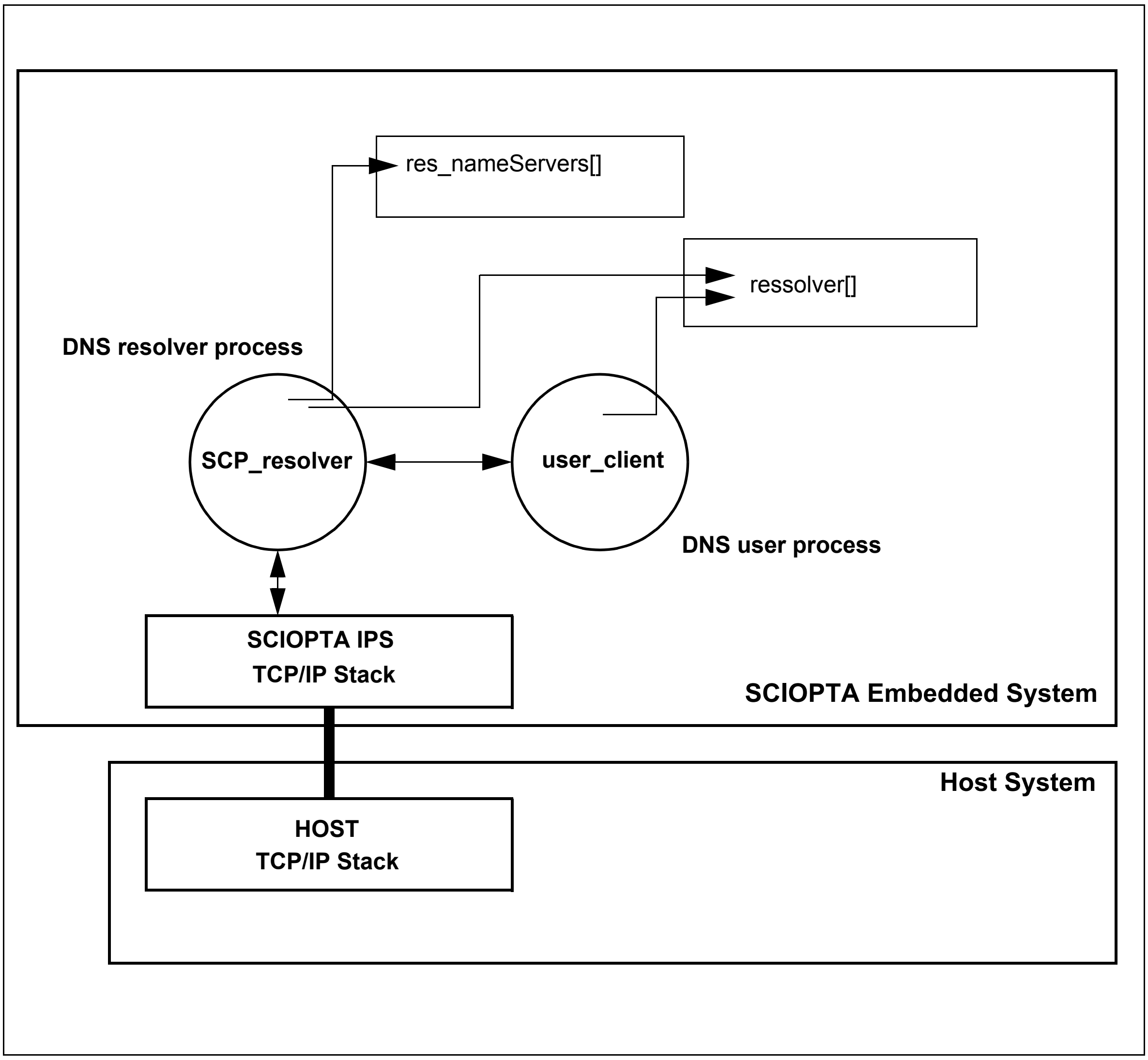

Figure 7-1: SCIOPTA IPS Domain Name System Structure

7.2 Getting Started - DNS Example

7.2.1 Description

DNS Domain Name System (or Service or Server) is a service in a TCP/IP network such as SCIOPTA IPS that translates domain names into IP addresses.

In this simple getting started example for the SCIOPTA DNS there is DNS test process which requiring the DNS service from the SCIOPTA resolver process.

The DNS resolver process is replying by scanning the res_nameServers[] list included in the resolv.c file.

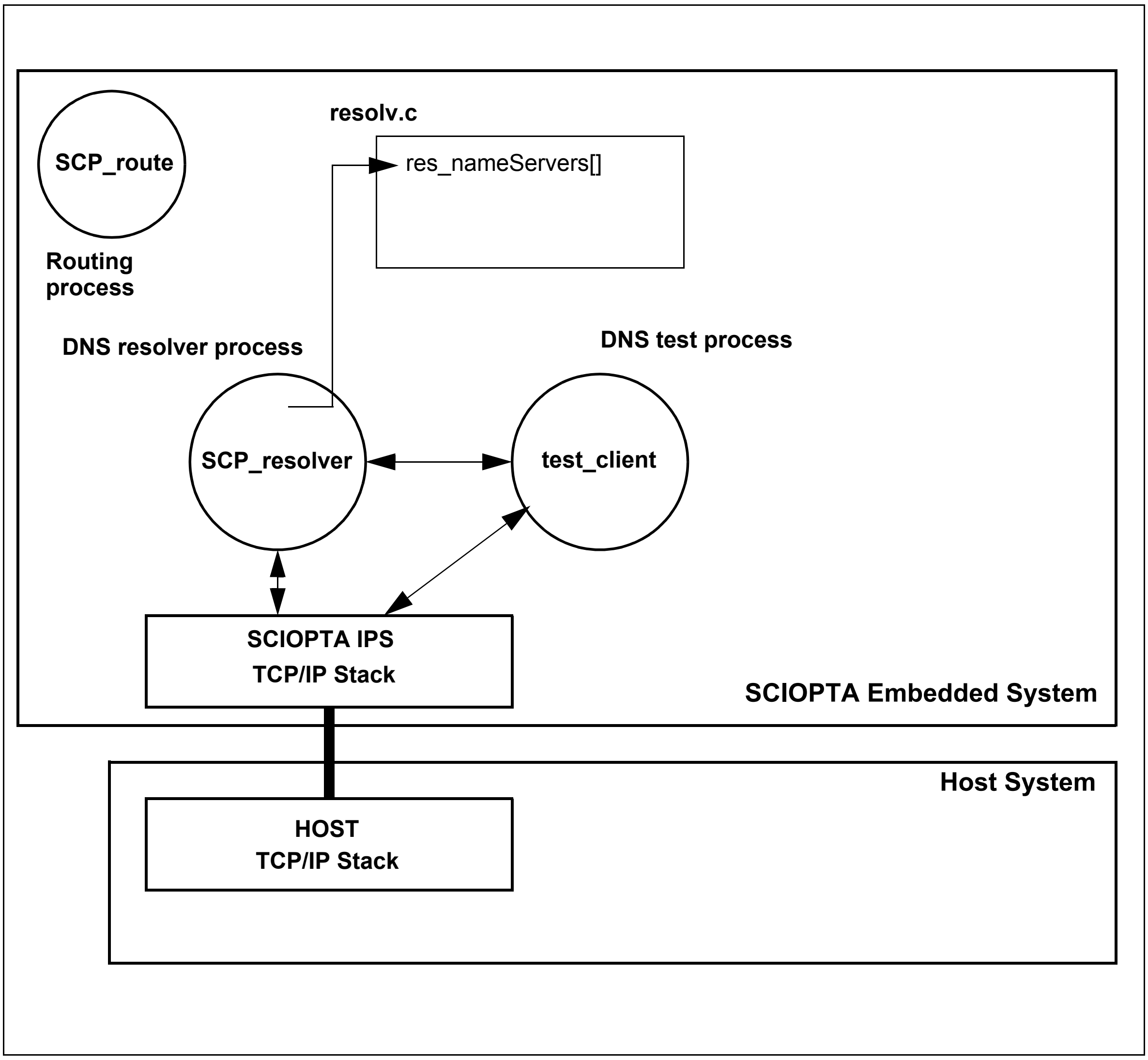

Figure 7-2: Getting Started DNS Client Example

7.2.2 IPS DNS Example - Windows Host

7.2.2.1 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\resolver\
- Network cable connected between your target board and your host workstation or to your network.
- Optionally a network protocol analyser running on your host computer such as Ethereal.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

7.2.2.2 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_resolver**) 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_resolver).
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\resolver\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_resolver).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_resolver).
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_resolver).
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_resolver**) 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_resolver\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. You can set breakpoints in the target dns (resolver) application and watch the behaviour.
31. For have a closer look at the network traffic generated by this example you can run a network analyser such as Ethereal on your host computer and trace the network messages.

7.3 DNS Configuration

7.3.1 Introduction

SCIOPTA DNS is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use DNS. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User's Guide for more information about the IPS stack and how to configure it.

DNS 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 - Target Manual for your selected processor for more information about using sconf.exe and the static configuration process.

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

2. SCIOPTA DNS Server configuration which configures the DNS resolver process (SCP_resolver).

   This process needs to write its name (and path) in the variable **resolver** and needs to call the DNS library function **resolver_process**.

   Furthermore a global variable **res_nameServers** holding the name servers and a global variable **res_noOfNameServers** holding the maximum number of name servers in the **res_nameServer** list must be defined and set-up.

7.3.2 DNS Module Configuration

In our standard getting started examples 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.

DNS is placed in the "ips" module. But you are free to split your application differently into other modules or put all processes in one module.

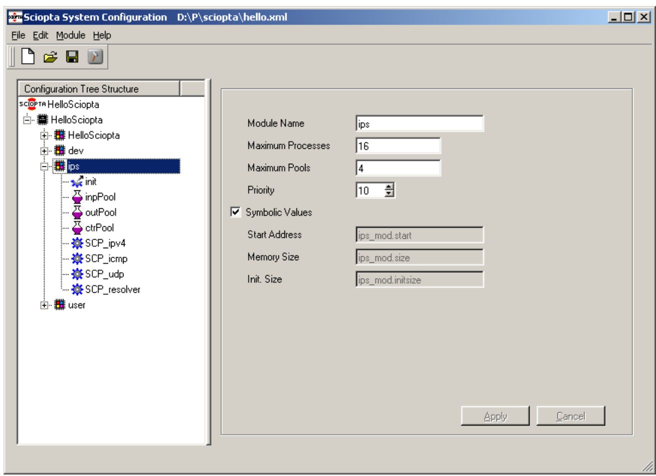

Figure 7-3: DNS module "ips"

7.3.2.1 Resolver Process

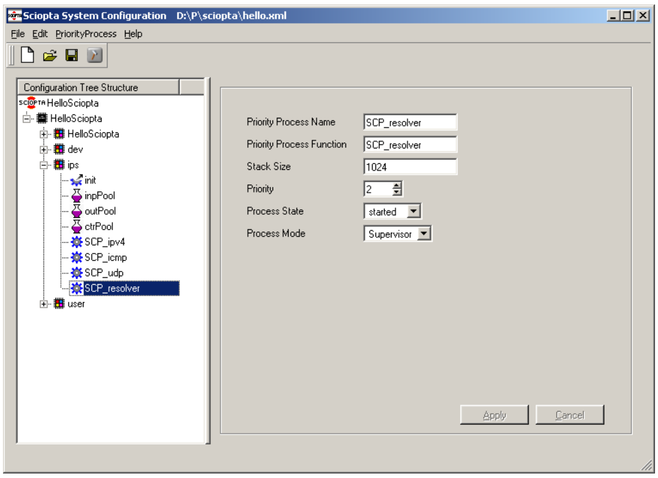

Figure 7-4: Resolver process SCP_resolver

7.3.3 DNS Configuration

In the previous chapters we have described how to configure the static modules, pools and processes for SCIOPTA DNS. 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 DNS you need also to configure some DNS parameters depending on the DNS specified properties. These parameters are defined by calling the resolver process at start-up (**SCP_resolver**).

In the SCIOPTA IPS DNS example we have include the declaration of this processes in the file **resolver.c** which can be found in the examples delivery.

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

7.3.3.1 Resolver Process Name

The global variable **resolver** holds the process name (and path) of the resolver process. It will be accessed by DNS user processes to search for the running resolver process.

Syntax

```
#define LOCAL_PATH_MAX (SC_PROC_NAME_SIZE*3+5)
char resolver[LOCAL_PATH_MAX] = {0};
```

Setup

The resolver process needs to write its own process name (and path) into this variable:

```
msg = sc_procPathGet(SC_CURRENT_PID,1);
strncpy(resolver,msg->path.path,LOCAL_PATH_MAX);
sc_msgFree(&msg);
```

7.3.3.2 Resolver Process Configuration

The DNS resolver process needs to call the resolver process function (**resolver_process()**) which is included in the DNS library.

Syntax

```
void resolver_process (__u16 port, __u32 tmo);
```

Parameters

port	UDP DNS port.
tmo	Maximum time to wait for a response.

Example

```
#include <sciopta.h>
#include <sciopta.msg>
#include <dns/parser.h>

#define RESOLVER_PORT    53
#define RESOLVER_TIMEOUT 4000 /* ms */

union sc_msg {
  sc_msgid_t id;
  sc_procPathGetMsgReply_t path;
};

#define LOCAL_PATH_MAX (SC_PROC_NAME_SIZE*3+5)
char resolver[LOCAL_PATH_MAX] = {0};

SC_PROCESS (SCP_resolver)
{
  union sc_msg *msg;

  msg = sc_procPathGet(SC_CURRENT_PID,1);
  strncpy(resolver,msg->path.path,LOCAL_PATH_MAX);
  sc_msgFree(&msg);

  resolver_process (RESOLVER_PORT, RESOLVER_TIMEOUT);
}
```

7.3.3.3 Name Server List

A global variable **res_nameServers** holding the name servers and a global variable **res_noOfNameServers** holding the maximum number of name servers in the **res_nameServer[]** list must be defined.

These variables will be accessed by the resolver process function **resolver_process()**.

In the SCIOPTA IPS DNS example we have include the declaration of this processes in the file **resolv.c** which can be found in the examples delivery.

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

Example

```
#include <ips/addr.h>

/* say how many name servers are in the search list */

int res_noOfNameServers = 3;

/* define your name servers */

ips_addr_t res_nameServers[] = {
  {4, {10, 0, 1, 11}},
  {4, {147, 86, 130, 1}},
  {4, {147, 86, 130, 2}},
};
```

7.4 DNS System Building

Please consult chapters "System Building" of the SCIOPTA ARM - Kernel, User's Guide and SCIOPTA ARM - IPS Internet Protocols, User's Guide for general information about the system building process.

You will find there information about:

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

7.4.1 Include Files

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

7.4.2 SCIOPTA ARM IPS DNS Libraries

7.4.2.1 Delivered IPS DNS Libraries

IPS DNS	libdns_**XY**.a	GCC
	dns_**XY**.l	ARM RealView
	dns_**XY**.r79	IAR Systems

7.4.2.2 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.

7.4.2.3 IPS DNS Libraries “Y”

For SCIOPTA IPS DNS 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.

Please consult chapter ‘IPS Internet Protocols Libraries “Y” ‘ of the SCIOPTA ARM - IPS Internet Protocols, User’s Guide for a table showing the included features for each of the three IPS levels.

7.5 Using DNS

7.5.1 DNS User Process

For using the SCIOPTA DNS you need to define and set-up a DNS user process.

The DNS user process requiring the DNS service from the SCIOPTA resolver process is using specific DNS messages such as **DNS_IP_REQUEST_MSG** and **DNS_NAME_REQUEST_MSG**.

The DNS resolver process is replying to DNS request by scanning its own name-server list (res_nameServers[]).

The DNS user process is doing usually the following:

1. Get the resolver process ID from the **resolver[]** variable.
2. Allocate a **DNS_IP_REQUEST_MSG** and send it to the resolver process if you need the IP address of a host by supplying its name.
3. Receive the **DNS_IP_REPLY_MSG** and retrieve the IP address.
4. Allocate a **DNS_NAME_REQUEST_MSG** and send it to the resolver process if you need the name of a host by supplying its IP address.
5. Receive the **DNS_NAME_REPLY_MSG** and retrieve the host name.

7.5.2 Example DNS User Process

```
#include <sciopta.h>
#include <string.h>

#include <dns/parser.h>
#include <dns/resolver.msg>

#define __TMO sc_tickTick2Ms(500) /*ms*/

  union sc_msg {
  sc_msgid_t id;
  dns_name_t ipRequest;
  dns_ip_t nameRequest;
  dns_ip_t ipReply;
  dns_name_t nameReply;
};

SC_PROCESS (test_client)
{
  extern const char resolver[];
  static const sc_msgid_t sel[3] = { DNS_IP_REPLY_MSG, DNS_NAME_REPLY_MSG, 0 };
  union sc_msg *msg;
  unsigned char *desiredHost[] = { "www.sciopta.com" };
  ips_addr_t tempIp = { 4, {147, 86, 130, 1} }; /*loki.cs.fh-aargau.ch*/
  int count = 0;
  sc_pid_t resolver_pid = SC_ILLEGAL_PID;
  int toggle;

  /* Wait to let IPS and resolver startup */
  sc_sleep ( __TMO );

  if ( resolver[0] ){
    resolver_pid = sc_procIdGet (resolver, SC_NO_TMO);
  }
```

```
  if ( resolver_pid == SC_ILLEGAL_PID ){
    /* no resolver process started ! */
    sc_miscError(0x11,0);
  }

  for (toggle = 0; /*for ever*/ ; toggle = 1 - toggle) {
    if (toggle) {
      msg = sc_msgAlloc (sizeof (dns_name_t),
                        DNS_IP_REQUEST_MSG, 0, SC_ENDLESS_TMO);

      msg->ipRequest.noOf = 1;
      strcpy (msg->ipRequest.entry[0].name, desiredHost[0]);
    }
    else {
      msg = sc_msgAlloc (sizeof (dns_ip_t),
              DNS_NAME_REQUEST_MSG, 0, SC_ENDLESS_TMO);

      msg->nameRequest.noOf = 1;
      memcpy (&msg->nameRequest.entry[0].ip, &tempIp, sizeof (ips_addr_t));
    }
    sc_msgTx (&msg, resolver_pid, 0);

    msg = sc_msgRx (SC_ENDLESS_TMO, (void *)sel, SC_MSGRX_MSGID);
    switch ( msg->id ) {
    case DNS_IP_REPLY_MSG:
      if (!msg->ipReply.parent.error) {
        ++count;
        kprintf (0, "%d: got a response: %d : %d.%d.%d.%d\n",
         count,
         msg->ipReply.noOf,
         msg->ipReply.entry[0].ip.addr[0],
         msg->ipReply.entry[0].ip.addr[1],
         msg->ipReply.entry[0].ip.addr[2],
         msg->ipReply.entry[0].ip.addr[3]);
      }
      else {
        kprintf (0, "got no ip response\n");
      }
      break;
    case DNS_NAME_REPLY_MSG:
      if (!msg->nameReply.parent.error) {
        ++count;
        kprintf (0, "%d: got a name: %s \n", count,
                 msg->nameReply.entry[0].name);
      }
      else {
        kprintf (0, "got no name reponse\n");
      }
      break;
    default:
      sc_miscError(0x11,1);
      break;
    }
    sc_msgFree (&msg);
    sc_sleep ( __TMO );
  }

  sc_procKill (SC_CURRENT_PID, 0);
}
```

7.6 DNS Message Interface

7.6.1 DNS_IP_REQUEST_MSG

This message is used to get the IP address of a host by supplying the host name.

The user process sends a **DNS_IP_REQUEST_MSG** message to the resolver process (**SCP_resolver**). The resolver process replies with a **DNS_IP_REPLY_MSG** including the host IP address.

Message ID

Request message	**DNS_IP_REQUEST_MSG**

dns_name_t Structure

```
typedef struct dns_name_s {
   sc_msgid_t                  id;
   dns_t                       parent;
   int                         noOf;
   struct dns_nameEntry {
      int                      ttl;
      __u8                     name[DNS_NAME_MAX + 1];
   } entry[1];
} dns_name_t;
```

Members

id

Standard SCIOPTA message ID.

parent

Not used

noOf

Number of inquiries.

entry[n].ttl

Not used.

entry[n].name[]

Host name

Header

<install_dir>\sciopta\<version>\include\dns\parser.h
<install_dir>\sciopta\<version>\include\dns\resover.msg

7.6.2 DNS_IP_REPLY_MSG

This message is used to reply to a IP address request including the IP address of a host.

The user process sends a **DNS_IP_REQUEST_MSG** message to the resolver process (**SCP_resolver**). The resolver process replies with a **DNS_IP_REPLY_MSG** including the host IP address.

Message ID

Reply message **DNS_IP_REPLY_MSG**

dns_ip_t Structure

```
typedef struct dns_ip {
   sc_msgid_t                id;
   dns_t                     parent;
   int                       noOf;
   struct dns_ipEntry {
      int                    ttl;
      ips_addr_t             ip;
   } entry[1];
} dns_ip_t;
```

Members

id

Standard SCIOPTA message ID.

parent

Not used

noOf

Number of replies.

entry[n].ttl

Time To Live value.

entry[n].ip

Host address. Please consult chapter 5 (Structures) of the SCIOPTA IPS Internet Protocols User's Guide and Reference Manual for information about the isp_addr_t structure.

Header

<install_dir>\sciopta\<version>\include\dns\parser.h
<install_dir>\sciopta\<version>\include\dns\resover.msg

7.6.3 DNS_NAME_REQUEST_MSG

This message is used to get the name of a host by supplying the IP address of the host.

The user process sends a **DNS_NAME_REQUEST_MSG** message to the resolver process (**SCP_resolver**). The resolver process replies with a **DNS_NAME_REPLY_MSG** including the host name.

Message ID

Reply message	**DNS_NAME_REQUEST_MSG**

dns_ip_t Structure

```
typedef struct dns_ip {
   sc_msgid_t             id;
   dns_t                  parent;
   int                    noOf;
   struct dns_ipEntry {
      int                 ttl;
      ips_addr_t          ip;
   } entry[1];
} dns_ip_t;
```

Members

id

Standard SCIOPTA message ID.

parent

Not used

noOf

Number of replies.

entry[n].ttl

Time To Live value.

entry[n].ip

Host address. Please consult chapter 5 (Structures) of the SCIOPTA IPS Internet Protocols User's Guide and Reference Manual for information about the isp_addr_t structure.

Header

<install_dir>\sciopta\<version>\include\dns\parser.h
<install_dir>\sciopta\<version>\include\dns\resover.msg

7.6.4 DNS_NAME_REPLY_MSG

This message is used to reply to a host name request including the name of a host.

The user process sends a **DNS_NAME_REQUEST_MSG** message to the resolver process (**SCP_resolver**). The resolver process replies with a **DNS_NAME_REPLY_MSG** including the host name.

Message ID

Request message	**DNS_NAME_REPLY_MSG**

dns_name_t Structure

```
typedef struct dns_name_s {
   sc_msgid_t               id;
   dns_t                    parent;
   int                      noOf;
   struct dns_nameEntry {
      int                   ttl;
      __u8                  name[DNS_NAME_MAX + 1];
   } entry[1];
} dns_name_t;
```

Members

id

Standard SCIOPTA message ID.

parent

Not used

noOf

Number of inquiries.

entry[n].ttl

Not used.

entry[n].name[]

Host name

Header

<install_dir>\sciopta\<version>\include\dns\parser.h
<install_dir>\sciopta\<version>\include\dns\resover.msg

7.7 Requests for Comments RFC - DNS

In this chapter we have listed the important RFCs which are important for the DNS implementation.

This list might not be complete and there is no guarantee that SCIOPTA DNS complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
1034	Domain names - concepts and facilities	November 1987
1035	Domain names - implementation and specification	November 1987
1101	DNS encoding of network names and other types	April 1989
1183	New DNS RR Definitions	October 1990
1383	An Experiment in DNS Based IP Routing	December 1992
1394	Relationship of Telex Answerback Codes to Internet Domains	January 1993
1401	Correspondence between the IAB and DISA on the use of DNS	January 1993
1464	Using the Domain Name System To Store Arbitrary String Attributes	May 1993
1480	The US Domain	June 1993
1535	A Security Problem and Proposed Correction With Widely Deployed DNS Software	October 1993
1536	Common DNS Implementation Errors and Suggested Fixes	October 1993
1591	Domain Name System Structure and Delegation	March 1994
1611	DNS Server MIB Extensions	May 1994
1612	DNS Resolver MIB Extensions	May 1994
1706	DNS NSAP Resource Records	October 1994
1712	DNS Encoding of Geographical Location	November 1994
1713	Tools for DNS debugging	November 1994
1788	ICMP Domain Name Messages	April 1995
1794	DNS Support for Load Balancing	April 1995
1876	A Means for Expressing Location Information in the Domain Name System	January 1996
1912	Common DNS Operational and Configuration Errors	February 1996
1982	Serial Number Arithmetic	August 1996
1995	Incremental Zone Transfer in DNS	August 1996
1996	A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)	August 1996
2136	Dynamic Updates in the Domain Name System (DNS UPDATE)	April 1997
2163	Using the Internet DNS to Distribute MIXER Conformant Global Address Mapping (MCGAM)	January 1998
2181	Clarifications to the DNS Specification	July 1997
2182	Selection and Operation of Secondary DNS Servers	July 1997
2219	Use of DNS Aliases for Network Services	October 1997
2230	Key Exchange Delegation Record for the DNS	November 1997
2247	Using Domains in LDAP/X.500 Distinguished Names	January 1998
2308	Negative Caching of DNS Queries (DNS NCACHE)	March 1998

RFC	Description	Date
2345	Domain Names and Company Name Retrieval	May 1998
2352	A Convention For Using Legal Names as Domain Names	May 1998
2517	Building Directories from DNS: Experiences from WWWSeeker	February 1999
2535	Domain Name System Security Extensions	March 1999
2536	DSA KEYs and SIGs in the Domain Name System (DNS)	March 1999
2538	Storing Certificates in the Domain Name System (DNS)	March 1999
2539	Storage of Diffie-Hellman Keys in the Domain Name System (DNS)	March 1999
2540	Detached Domain Name System (DNS) Information	March 1999
2541	DNS Security Operational Considerations	March 1999
2606	Reserved Top Level DNS Names	June 1999
2671	Extension Mechanisms for DNS (EDNS0)	August 1999
2672	Non-Terminal DNS Name Redirection	August 1999
2673	Binary Labels in the Domain Name System	August 1999
2694	DNS extensions to Network Address Translators (DNS_ALG)	September 1999
2782	A DNS RR for specifying the location of services (DNS SRV)	February 2000
2826	IAB Technical Comment on the Unique DNS Root	May 2000
2845	Secret Key Transaction Authentication for DNS (TSIG)	May 2000
2874	DNS Extensions to Support IPv6 Address Aggregation and Renumbering	July 2000
2929	Domain Name System (DNS) IANA Considerations	September 2000
2930	Secret Key Establishment for DNS (TKEY RR)	September 2000
2931	DNS Request and Transaction Signatures (SIG(0)s)	September 2000
3007	Secure Domain Name System (DNS) Dynamic Update	November 2000
3008	Domain Name System Security (DNSSEC) Signing Authority	November 2000
3026	Liaison to IETF/ISOC on ENUM	January 2001
3071	Reflections on the DNS, RFC 1591, and Categories of Domains J.	February 2001
3088	OpenLDAP Root Service An experimental LDAP referral service	April 2001
3090	DNS Security Extension Clarification on Zone Status	March 2001
3110	RSA/SHA-1 SIGs and RSA KEYs in the Domain Name System (DNS)	May 2001
3123	A DNS RR Type for Lists of Address Prefixes (APL RR)	June 2001
3130	Notes from the State-Of-The-Technology: DNSSEC	June 2001
3172	Management Guidelines & Operational Requirements for the Address and Routing Parameter Area Domain (arpa)	September 2001
3197	Applicability Statement for DNS MIB Extensions	November 2001
3225	Indicating Resolver Support of DNSSEC	December 2001
3226	DNSSEC and IPv6 A6 aware server/resolver message size requirements	December 2001
3258	Distributing Authoritative Name Servers via Shared Unicast Addresses	April 2002
3363	Representing Internet Protocol version 6 (IPv6) Addresses in the Domain Name System (DNS)	August 2002

RFC	Description	Date
3364	Tradeoffs in Domain Name System (DNS) Support for Internet Protocol version 6 (IPv6)	August 2002
3397	Dynamic Host Configuration Protocol (DHCP) Domain Search Option	November 2002
3403	Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database	October 2002
3425	Obsoleting IQUERY	November 2002
3445	Limiting the Scope of the KEY Resource Record (RR)	December 2002
3467	Role of the Domain Name System (DNS)	February 2003
3596	DNS Extensions to Support IP Version 6	October 2003
3597	Handling of Unknown DNS Resource Record (RR) Types	September 2003
3645	Generic Security Service Algorithm for Secret Key Transaction Authentication for DNS (GSS-TSIG)	October 2003
3646	DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)	December 2003
3655	Redefinition of DNS Authenticated Data (AD) bit	November 2003
3658	Delegation Signer (DS) Resource Record (RR)	December 2003
3681	Delegation of E.F.F.3.IP6.ARPA	January 2004
3755	Legacy Resolver Compatibility for Delegation Signer (DS)	May 2004
3757	Domain Name System KEY (DNSKEY) Resource Record (RR) Secure Entry Point (SEP) Flag	May 2004
3832	Remote Service Discovery in the Service Location Protocol (SLP) via DNS SRV	July 2004
3833	Threat Analysis of the Domain Name System (DNS)	August 2004
3845	DNS Security (DNSSEC) NextSECure (NSEC) RDATA Format	August 2004

8 Telnet Terminal Emulation Program

8.1 Description

Telnet is a terminal emulation program for TCP/IP networks such as SCIOPTA using the IPS stack.

The Telnet daemon process is managing and controlling the telnet device processes (**SCP_td_n**). The Telnet device processes are implemented as a standard SCIOPTA SDD devices which are registered at the standard device manager (**SCP_devman**). Please consult the SCIOPTA - Device Driver, User's Guide and Reference Manual for more information about the SCIOPTA device driver system. The telnet application processes include the telnet interactive functionality and can also implement as a shell.

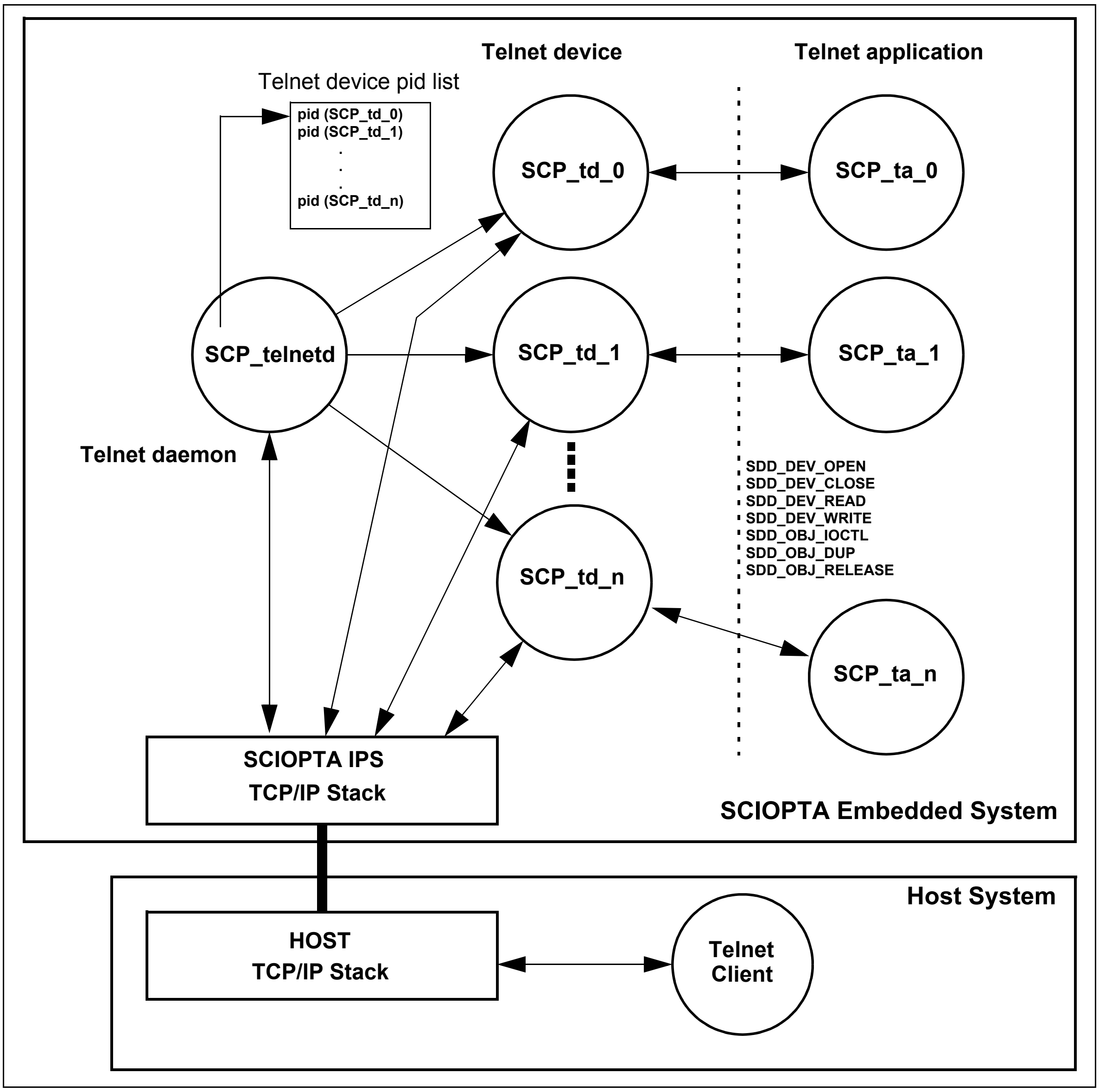

Figure 8-1: SCIOPTA Telnet System Structure

8.2 Getting Started - Telnet Example

8.2.1 Description

Telnet is a terminal emulation program for TCP/IP networks such as SCIOPTA using the IPS stack.

This simple getting started example for the SCIOPTA telnet server is setting up a telnet server which is just sending a welcome message to the telnet client on the host and then returns all typed characters.

The Telnet daemon process is managing and controlling the telnet device processes. In this example there is only one Telnet device (SCP_telnet0).

The Telnet device process (SCP_telnet0) is implemented as a standard SCIOPTA SDD device which is registered at the standard device manager (**SCP_devman**). Please consult the SCIOPTA - Device Driver, User's Guide and Reference Manual for more information about the SCIOPTA device driver system.

The bouncer is a process which is sending the message "Hello Sciopta" via the telnet device process (SCP_telnet0) when the Telnet client on the host connects to the SCIOPTA Telnet server. Then all typed lines from the Telnet client will be returned by the bouncer to it.

Typing a single quote (".") will terminate the session on the target side.

.

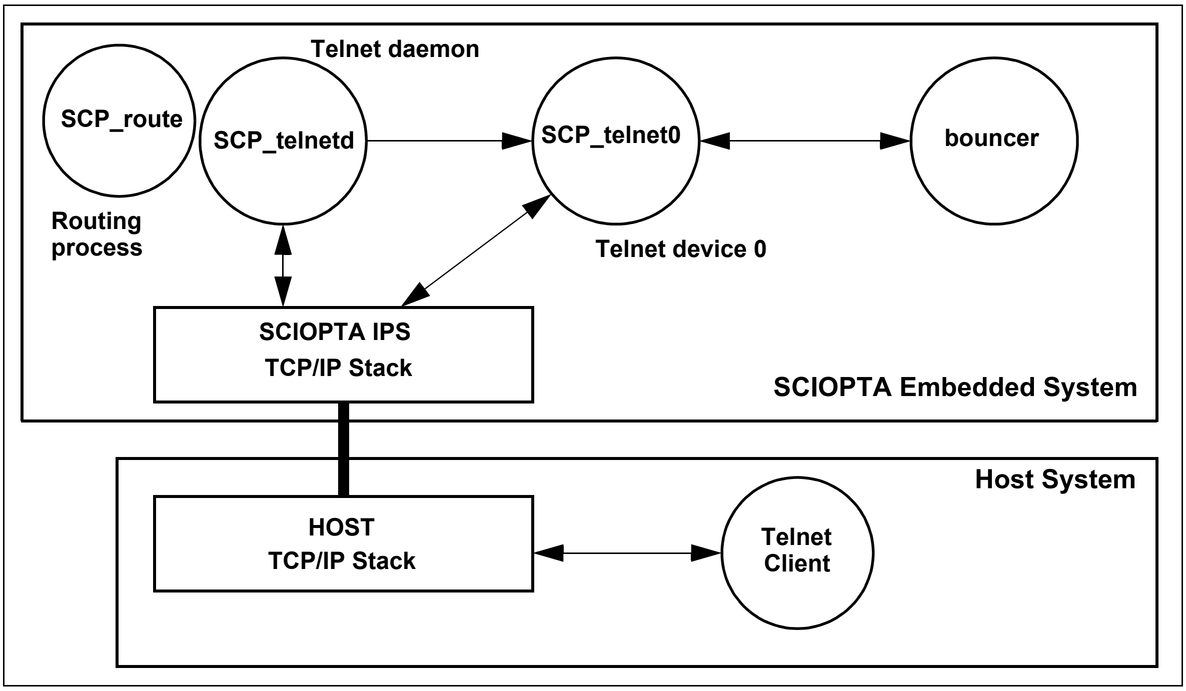

Figure 8-2: Getting Started Telnet Client Example

8.2.2 IPS Telnet Example - Windows Host

8.2.2.1 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\telnet\
- Network cable connected between your target board and your host workstation or to your network.
- A standard Telnet client program running on your host computer.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

8.2.2.2 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_telnet**) 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_telnet).
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\telnet\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_telnet).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_telnet).
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_telnet).
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_telnet**) 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_telnet\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. The message “Hello Sciopta” must be written in your telnet window. Each typed line will be returned by the target telnet server.
33. You can also set breakpoints in the target telnet application and watch the behaviour.

8.3 Telnet Configuration

8.3.1 Introduction

SCIOPTA Telnet is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use Telnet. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User's Guide for more information about the IPS stack and how to configure it.

Telnet 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 - Target Manual for your selected processor for more information about using sconf.exe and the static configuration process.

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

2. SCIOPTA Telnet configuration which configures the telnet daemon process (**SCP_telnetd**).

   For each concurrent telnet session a pair of a telnet device process (**SCP_td_n**) and telnet application process (**SCP_ta_n**) needs to be defined and created. The telnet device process (SCP_td_n) calls the Telnet library function **telnet_device()**.

8.3.2 Telnet Module Configuration

In our standard getting started examples 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.

Telnet is placed in the "user" module. But you are free to split your application differently into other modules or put all processes in one module.

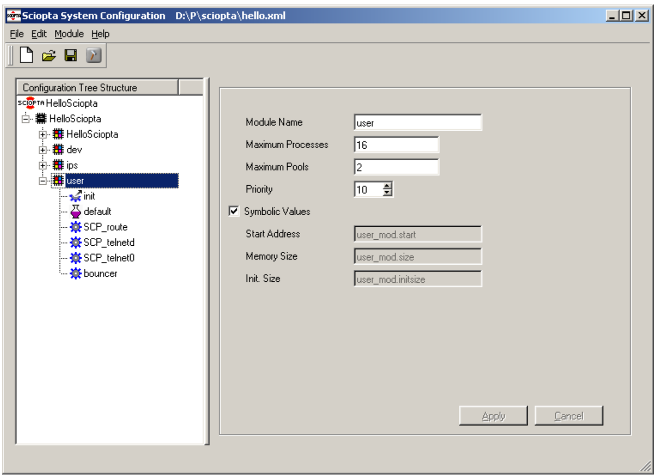

Figure 8-3: Telnet module "user"

8.3.2.1 Telnet 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 256 bytes. You could optimize this stack by doing a stack analysis using the DRUID system level debugger.

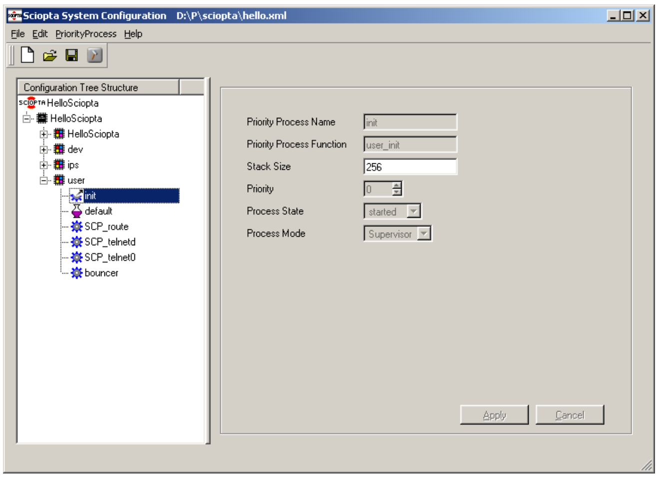

Figure 8-4: Telnet module init process

8.3.2.2 Telnet Module Default Pool

The pool parameters must be designed to fit the requirements of the telnet messages buffer sizes.

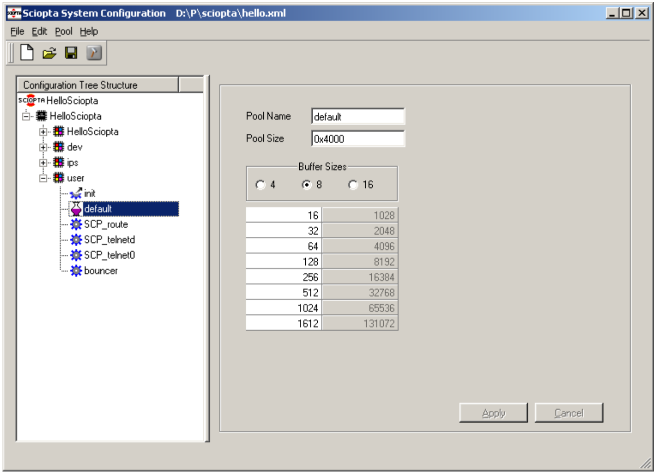

Figure 8-5: Telnet module default pool

8.3.2.3 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.

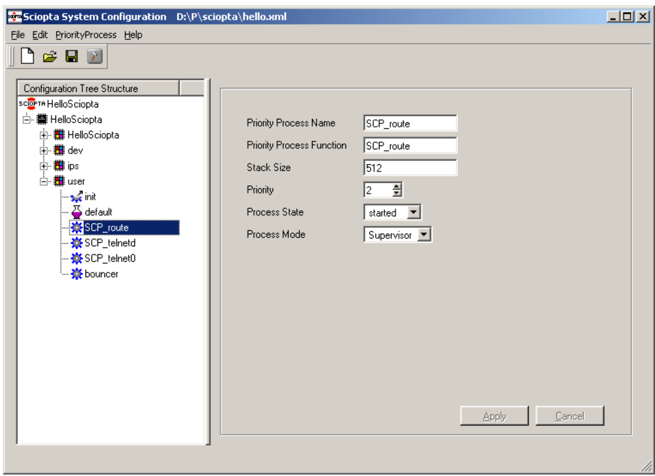

Figure 8-6: Routing process SCP_route

8.3.2.4 Telnet Daemon Process

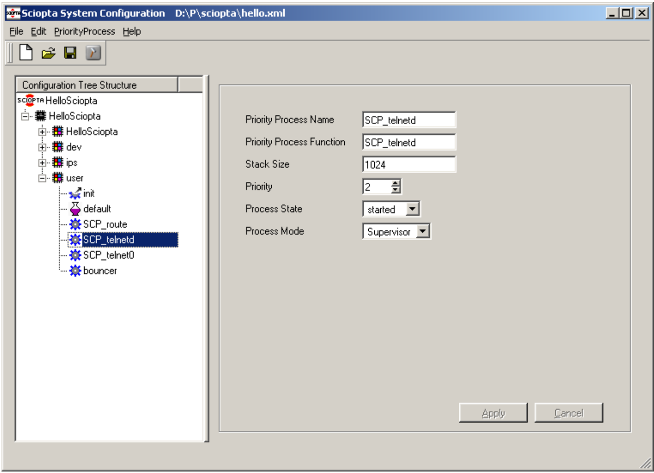

Figure 8-7: Telnet daemon process SCP_telnetd

8.3.2.5 Telnet Device Process

For each concurrent telnet session a pair of a telnet device process and telnet application process needs to be defined and created. The telnet device process is called SCP_telnetX() where X is a consecutive number for each telnet session.

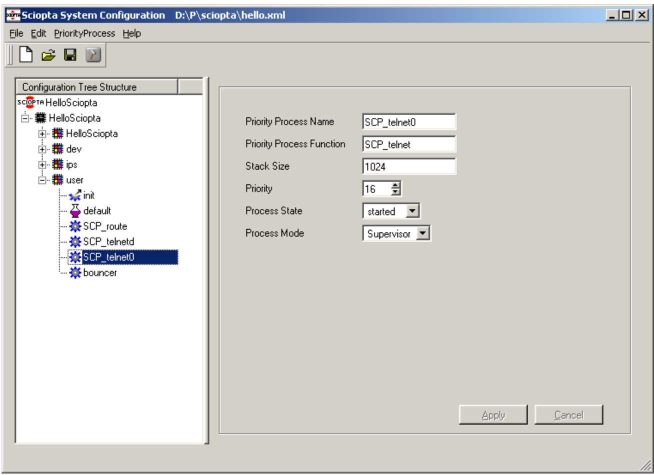

Figure 8-8: Telnet device process example SCP_telnet0

8.3.2.6 Telnet Application Process

For each concurrent telnet session a pair of a telnet device process and telnet application process needs to be defined and created. The telnet application process can have any name (here: bouncer).

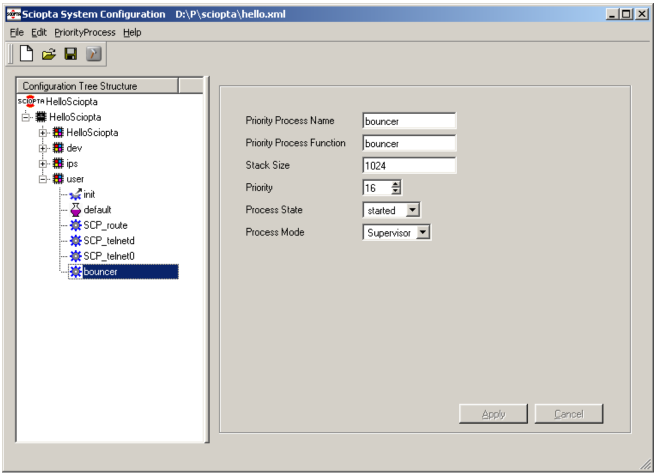

Figure 8-9: Telnet application process example (bouncer)

8.3.3 Telnet Configuration

In the previous chapters we have described how to configure the static modules, pools and processes for SCIOPTA Telnet. 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 Telnet you need also to configure some Telnet parameters depending on the Telnet specified properties. These parameters are defined by calling the telnet daemon process at start-up (**SCP_telnetd**). Additionally you need to configure the telnet device process (**SCP_telnet**) by passing the process name to the **telnet_device()** function.

In the SCIOPTA IPS Telnet example we have include the declaration of these processes in the file **telnetsetup.c** which can be found in the examples delivery.

File location: <install_folder>\sciopta\<version>\exp\ips\common\telnet\telnetsetup.c

8.3.3.1 Telnet Daemon Process Configuration

The Telnet Daemon needs to know what Telnet devices exist. Before calling **telnet_daemon** the list must be set-up. This can be done by using the **sc_procIdGet** system call which returns the PID for a given process name. Please consult the Telnet daemon process example.

The Telnet daemon process needs to call the telnet daemon process function (**telnet_daemon**) which is included in the Telnet library.

Syntax

```
void telnet_daemon(
   ips_addr_t            *ip,
   __u16                 port,
   sc_pid_t              *server
);
```

Parameters

ip

Pointer to the IP address. Usually any address (ip.len = 0). Please consult chapter 5 (Structures) of the SCIOPTA IPS Internet Protocols User's Guide and Reference Manual for information about the isp_addr_t structure.

port

Port number. Usually 23.

server

Pointer to the telnet pid device list. The telnet daemon needs to know what telnet devices exist.

Example

```
#include <sciopta.h>
#include <sciopta.msg>
#include <telnet/telnet.h>

#include <string.h>

union sc_msg {
  sc_msgid_t id;
  sc_procNameGetMsgReply_t nameget;
};


/* define telnet daemon
 */
SC_PROCESS (SCP_telnetd)
{
  ips_addr_t ip;
  sc_pid_t server[4];

  /* telnet daemon needs to know what telnet device exist, they are the
   * telnets worker process.
   */
  server[0] = sc_procIdGet ("/user/SCP_telnet0", IDGET_TMO);
  server[1] = sc_procIdGet ("/user/SCP_telnet1", IDGET_TMO);
  server[2] = sc_procIdGet ("/user/SCP_telnet2", IDGET_TMO);
  server[3] = 0;

  /* our telnet server should listen on ANY address and on port 23
   */
  ip.len = 0;
  telnet_daemon (&ip, 23, server);
}
```

8.3.3.2 Telnet Device Process Configuration

The Telnet device process needs to call the telnet device process function (**telnet_device()**) which is included in the Telnet library. The process name needs to be retrieved and passed to the **telnet_device()** function.

Syntax

```
void telnet_device(
   const char          *name
);
```

Parameter

name

Pointer to the name of the telnet device process.

Example

```
#include <sciopta.h>
#include <sciopta.msg>
#include <telnet/telnet.h>

#include <string.h>

union sc_msg {
  sc_msgid_t id;
  sc_procNameGetMsgReply_t nameget;
};

#define IDGET_TMO 100/*ticks*/

/* define telnet device
 */
SC_PROCESS (SCP_telnet)
{
  union sc_msg *msg;
  char dev[10];
  char *p;
  msg = sc_procNameGet(SC_CURRENT_PID);
  p = strstr(msg->nameget.process,"SCP_telnet");
  if ( p == 0 || p != msg->nameget.process ){
    sc_miscError(0x12,(sc_extra_t)&msg->nameget.process);
    sc_msgFree(&msg);
  }
  strncpy(dev,&p[4],9);
  dev[9]=0;
  sc_msgFree(&msg);

  telnet_device (dev);
}
```

8.4 Telnet System Building

Please consult chapters "System Building" of the SCIOPTA ARM - Kernel, User's Guide and SCIOPTA ARM - IPS Internet Protocols, User's Guide for general information about the system building process.

You will find there information about:

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

8.4.1 Include Files

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

8.4.2 SCIOPTA ARM IPS Telnet Libraries

8.4.2.1 Delivered IPS Telnet Libraries

IPS Telnet	libtelnet_**XY**.a	GCC
	telnet_**XY**.l	ARM RealView
	telnet_**XY**.r79	IAR Systems

8.4.2.2 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.

8.4.2.3 IPS Telnet Libraries “Y”

For SCIOPTA IPS Telnet 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.

Please consult chapter ‘IPS Internet Protocols Libraries “Y” ‘ of the SCIOPTA ARM - IPS Internet Protocols, User’s Guide for a table showing the included features for each of the three IPS levels.

8.5 Using Telnet

The telnet device processes must be started before the telnet daemon process as it needs the running telnet device process to set-up the telnet device pid list.

The telnet daemon sets-up the telnet device pid list by entering the process id for each existing telnet device process.

The telnet application opens the telnet device and blocks until the device is ready.

The telnet daemon executes an open (tcp), a bind (ip, port), a listen and within a while loop an accept. The daemon is waiting on a connection.

The telnet client builds up a connection to the telnet daemon which gets in its turn the accept back and the daemon receives the fully constructed SDD object descriptor.

The telnet daemon scans the telnet device pid list and searches for a free telnet device. The daemon sends the SDD object descriptor to the first found free telnet device. The telnet device is now fully functional and unlocks the telnet application.

The telnet application adjusts the telnet device by using **sdd_devIoctl** functions. The telnet application can now read from and write to the telnet client by using **sdd_devRead** and **sdd_devWrite** functions or by using the device driver message interface.

The telnet device can be used exactly the same way as standard SCIOPTA devices such as a serial device.

Do not forget to close the telnet device after use so that the telnet device can be used again for another telnet session.

8.5.1 Telnet Application Example

```
#include <sciopta.h>
#include <sciopta.msg>
#include <telnet/telnet.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <sdd/sdd.h>
#include <sdd/sdd.msg>

#define DEVICE_MANAGER "/SCP_devman"
#define DEVICE "telnet0"

SC_PROCESS(bouncer)
{
  int opt;
  sdd_obj_t *man, *dev;
  char buf[32];
  ssize_t n;

  man = sdd_manGetRoot (DEVICE_MANAGER, "/", SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
```

```
  for (;;) {

    /* open device for the shell, fd will be 0 == stdin
     */
    dev = sdd_manGetByName(man,DEVICE);
    if (!dev || ! SDD_IS_A (dev, SDD_DEV_TYPE)) {
      sc_msgFree ((union sc_msg **) &man);
      sc_miscError (1, 0);
    }

    if ( sdd_devOpen(dev,O_RDWR) == -1 ){
      sc_msgFree ((union sc_msg **) &man);
      sc_miscError (1, 0);
    }

    /* set echo and tty mode
     */
    opt = 1;
    sdd_devIoctl(dev,SERECHO,(unsigned long )&opt);

    opt = 1;
    sdd_devIoctl(dev,SERTTY,(unsigned long )&opt);

    sdd_devIoctl(dev,SERSETEOL,(unsigned long )"\n");

    n = sdd_devWrite(dev,"Hello Sciopta\n",14);

    for(; n > 0;){
      n = sdd_devRead(dev,buf,31);
      if ( n == 2 && buf[0] == '.' ){
    n = 0;
      } else if ( n > 0 ){
    buf[n] = 0;
    n = sdd_devWrite(dev,buf,n);
      }
    }
    sdd_devClose(dev);
    sdd_objFree(&dev);
  }
}
```

8.6 Telnet API

8.6.1 Introduction

There is no specific API to use SCIOPTA Telnet (Telnet application process). But there three specific IOCTL commands used in Telnet.

8.6.2 Telnet IOCTL Commands

Please consult the SCIOPTA Device Driver, User's Guide and Reference Manual for more information about the **ioctl** function.

8.6.2.1 SERECHO

SERECHO is per default not switched on. For instance to keep password secret from the user ECHO is switched off. But usually it is switched on. For extremely simple Telnet clients the status of SERECHO does not care. It is good design practice to switch on SERECHO and to switch it off only in specific cases.

Syntax

```
int opt = 1;                /* Switch SERECHO on */

ioctl (fd, SERECHO, &opt);
```

8.6.2.2 SERTTY

SERTTY is per default not switched on. If the Telnet application process is a shell this cannot be accepted because the read is only returned if it has received the number of bytes which it had requested and not already by receiving a newline. Therefore it is good practice to switch on SERTTY.

Syntax

```
int opt = 1;                /* Switch SERTTY on */

ioctl (fd, SERTTY, &opt);
```

8.6.2.3 SERSETEOL

SERSETEOL defines the newline character. The newline character is defined as '/n' per default. You need to set this option only if your Telnet application process is using another newline character as '/n'.

Syntax

```
ioctl (fd, SERSETEOL, '/n');
```

8.7 Requests for Comments RFC - Telnet

In this chapter we have listed the important RFCs which are important for the Telnet implementation.

This list might not be complete and there is no guarantee that SCIOPTA Telnet complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
0097	First Cut at a Proposed Telnet Protocol	February 1971
0137	Telnet Protocol - a proposed document	April 1971
0139	Discussion of Telnet Protocol	May 1971
0206	User Telnet - description of an initial implementation	August 1971
0215	NCP, ICP, and Telnet: The Terminal IMP implementation	August 1971
0216	Telnet access to UCSB's On-Line System	September 1971
0318	Telnet Protocols	April1972
0328	Suggested Telnet Protocol Changes	April 1972
0339	MLTNET: A Multi Telnet Subsystem for Tenex	May 1972
0340	Proposed Telnet Changes	May 1972
0393	Comments on Telnet Protocol Changes	October 1972
0435	Telnet issues	January 1973
0461	Telnet Protocol meeting announcement	February 1973
0466	Telnet logger/server for host LL-67	February 1973
0495	Telnet Protocol specifications	May 1973
0513	Comments on the new Telnet specifications	May 1973
0559	Comments on The New Telnet Protocol and its Implementation	August 1973
0560	Remote Controlled Transmission and Echoing Telnet option	August 1973
0562	Modifications to the Telnet specification	August 1973
0563	Comments on the RCTE Telnet option	August 1973
0581	Corrections to RFC 560: Remote Controlled Transmission and Echoing Telnet Option	November 1973
0587	Announcing new Telnet options	November 1973
0593	Telnet and FTP implementation schedule change	November 1973
0595	Second thoughts in defense of the Telnet Go-Ahea	December 1973
0596	Second thoughts on Telnet Go-Ahead	December 1973
0617	Note on socket number assignment	February 1974
0624	Comments on the File Transfer Protocol	February 1974
0652	Telnet output carriage-return disposition option	October 1974
0653	Telnet output horizontal tabstops option	October 1974
0654	Telnet output horizontal tab disposition option	October 1974
0655	Telnet output formfeed disposition option	October 1974
0656	Telnet output vertical tabstops option	October 1974

RFC	Description	Date
0657	Telnet output vertical tab disposition option	October 1974
0658	Telnet output linefeed disposition	October 1974
0659	Announcing additional Telnet options	October 1974
0688	Tentative schedule for the new Telnet implementation for the TIP	June 1975
0698	Telnet extended ASCII option	July 1975
0726	Remote Controlled Transmission and Echoing Telnet option	March 1977
0727	Telnet logout option	April 1977
0728	Minor pitfall in the Telnet Protocol	April 1977
0732	Telnet Data Entry Terminal option	September 1977
0735	Revised Telnet byte macro option	November 1977
0736	Telnet SUPDUP option	October 1977
0748	Telnet randomly-lose option	April 1978
0749	Telnet SUPDUP-Output option	September 1978
0779	Telnet send-location option	April 1981
0818	Remote User Telnet service	November 1982
0854	Telnet Protocol Specification	May 1983
0855	Telnet Option Specifications	May 1983
0856	Telnet Binary Transmission	May 1983
0857	Telnet Echo Option	May 1983
0858	Telnet Suppress Go Ahead Option	May 1983
0859	Telnet Status Option	May 1983
0860	Telnet Timing Mark Option	May 1983
0861	Telnet Extended Options: List Option	May 1983
0885	Telnet end of record option	December 1983
0927	TACACS user identification Telnet option	December 1984
0933	Output marking Telnet option	January 1985
0946	Telnet terminal location number option	May 1985
1041	Telnet 3270 regime option	January 1988
1043	Telnet Data Entry Terminal option: DODIIS implementation	February 1988
1053	Telnet X.3 PAD option	April 1988
1073	Telnet window size option	October 1988
1079	Telnet terminal speed option	December 1988
1091	Telnet terminal-type option	February 1989
1096	Telnet X display location option	May 1989
1097	Telnet subliminal-message option	April 1989
1143	The Q Method of Implementing TELNET Option Negotiation	February 1990
1184	Telnet Linemode Option	October 1990

RFC	Description	Date
1205	5250 Telnet interface	February 1991
1372	Telnet Remote Flow Control Option	October 1992
1408	Telnet Environment Option	January 1993
1411	Telnet Authentication: Kerberos Version	January 1993
1412	Telnet Authentication: SPX	January 1993
1571	Telnet Environment Option Interoperability Issues	January 1994
1572	Telnet Environment Option	January 1994
1576	TN3270 Current Practices	January 1994
1646	TN3270 Extensions for LUname and Printer Selection	July 1994
1922	Chinese Character Encoding for Internet Messages	March 1996
2066	TELNET CHARSET Option	January 1997
2217	Telnet Com Port Control Option	October 1997
2355	TN3270 Enhancements	June 1998
2561	Base Definitions of Managed Objects for TN3270E Using SMIv2	April 1999
2562	Definitions of Protocol and Managed Objects for TN3270E Response Time Collection Using SMIv2 (TN3270E-RT-MIB)	April 1999
2840	TELNET KERMIT OPTION	May 2000
2877	5250 Telnet Enhancements	July 2000
2941	Telnet Authentication Option	September 2000
2942	Telnet Authentication: Kerberos Version 5	September 2000
2943	TELNET Authentication Using DSA	September 2000
2944	Telnet Authentication: SRP	September 2000
2946	Telnet Data Encryption Option	September 2000
2947	Telnet Encryption: DES3 64 bit Cipher Feedback	September 2000
2948	Telnet Encryption: DES3 64 bit Output Feedback	September 2000
2949	Telnet Encryption: CAST-128 64 bit Output Feedback	September 2000
2950	Telnet Encryption: CAST-128 64 bit Cipher Feedback	September 2000
2951	TELNET Authentication Using KEA and SKIPJACK	September 2000
2952	Telnet Encryption: DES 64 bit Cipher Feedback	September 2000
2953	Telnet Encryption: DES 64 bit Output Feedback	September 2000

9 DHCP Dynamic Host Configuration Protocol

9.1 Description

DHCP Dynamic Host Configuration Protocol is a protocol for assigning dynamic IP addresses to devices on a TCP/IP network such as SCIOPTA IPS.

DHCP client and DHCP server is implemented in SCIOPTA IPS.

Using DHCP client in SCIOPTA is extremely simple. The user just needs to configure a DHCP client process (e.g. **SCP_dhcpclient**) which is calling the library function **dchp_client**. This function is doing the whole DHCP client work such as making the request, setting the routes, renewing the **Lease** and freeing the **Lease**.

For DHCP server there is a server daemon process (**SCP_dhcpd**) which performs all DCHP server function in the target system.

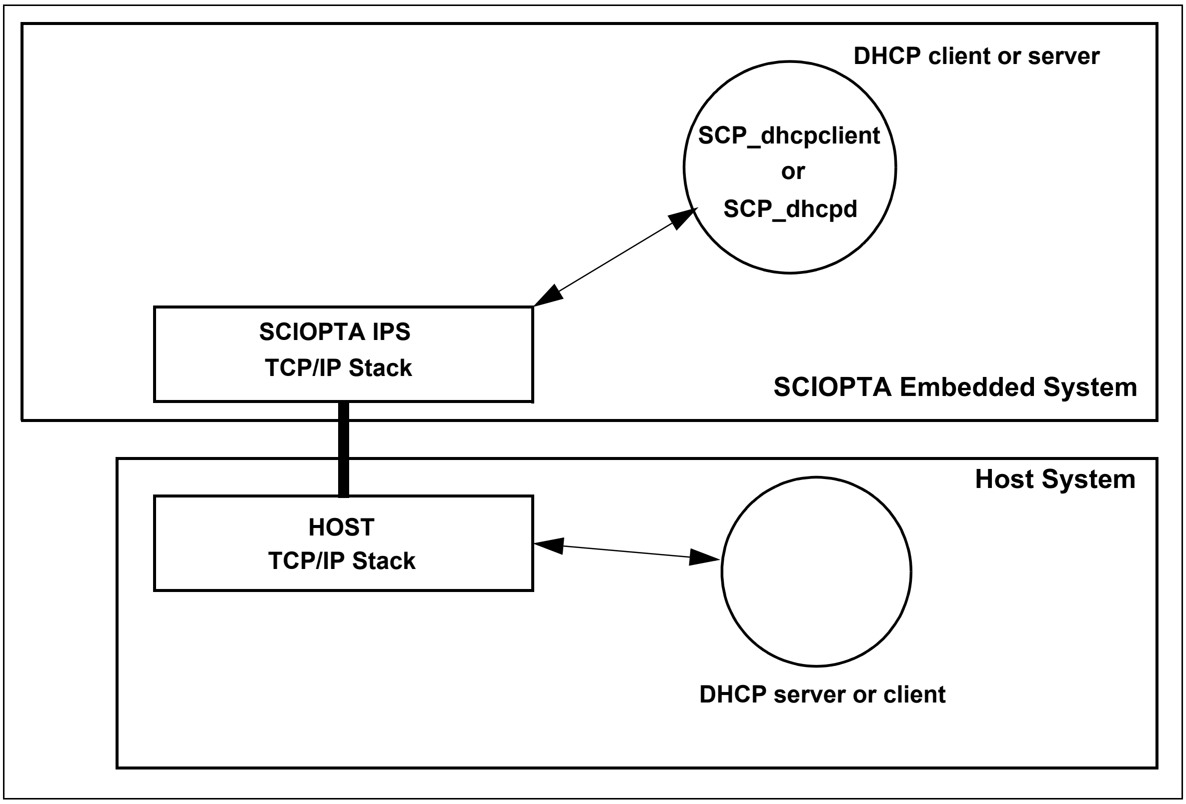

Figure 9-1: IPS Dynamic Host Configuration Protocol System Structure

9.2 Getting Started - DHCP Client Example

9.2.1 Description

DHCP Dynamic Host Configuration Protocol is a protocol for assigning dynamic IP addresses to devices on a TCP/IP network such as SCIOPTA IPS.

In this simple getting started example for the SCIOPTA DHCP client there is DHCP client process which is asking for a IP address from a DHCP server anywhere in the network.

The SCP_proxy process is setting routes which it gets from process SCP_dhcpclient. The SCP_proxy process was introduced to better follow the system behaviour by scanning messages.

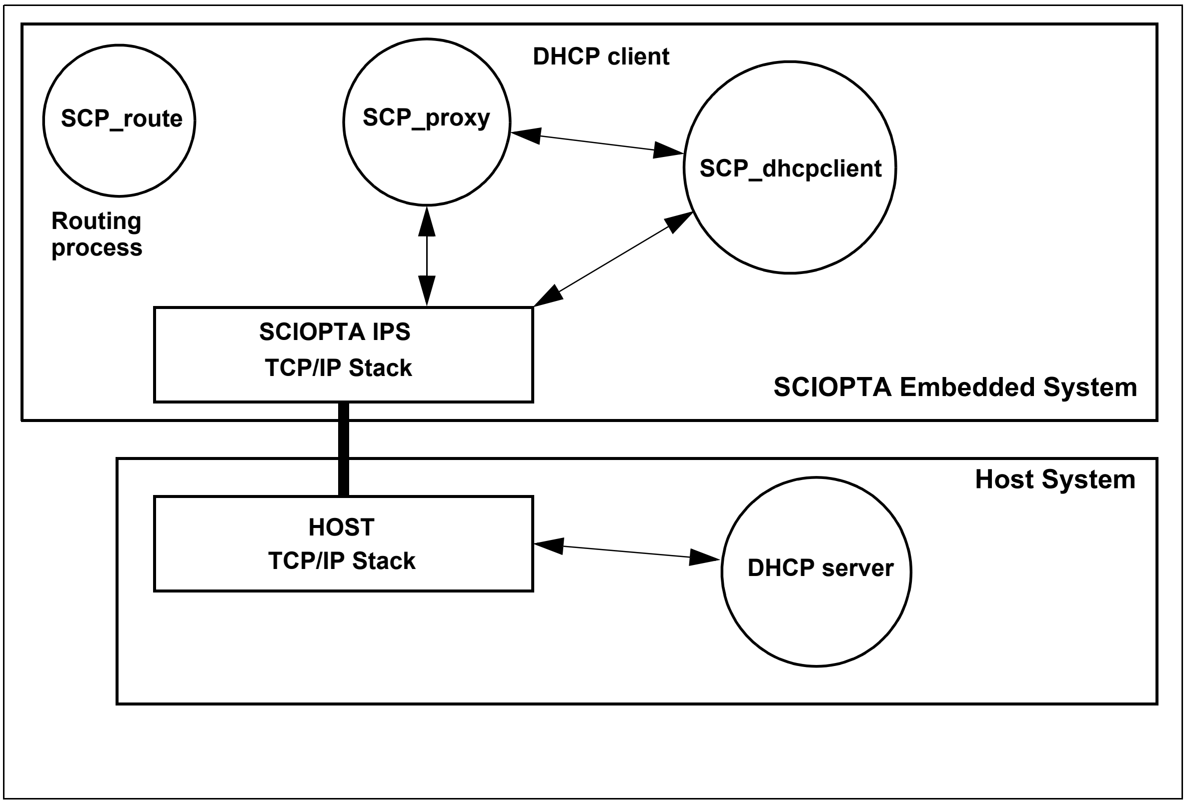

Figure 9-2: Getting Started DHCP Client Example

9.2.2 IPS DHCP Client Example - Windows Host

9.2.2.1 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\dhcp\
- Network cable connected between your target board and your host workstation or to your network.
- A DHCP server running in the connected network such as TFTPD32.
- Optionally a network protocol analyser running on your host computer such as Ethereal.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

9.2.2.2 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_dhcp**) 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_dhcp).
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\dhcp\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_dhcp).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_dhcp).
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_dhcp).
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_dhcp**) 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_dhcp\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. You can set breakpoints in the target dhcp client application and watch the behaviour.
31. For have a closer look at the network traffic generated by this example you can run a network analyser such as Ethereal on your host computer and trace the network messages.

9.3 Getting Started - DHCP Server Example

9.3.1 Description

DHCP Dynamic Host Configuration Protocol is a protocol for assigning dynamic IP addresses to devices on a TCP/IP network such as SCIOPTA IPS.

In this simple getting started example for the SCIOPTA DHCP server there is DHCP server process which is delivering IP addresses to DHCP clients anywhere in the network.

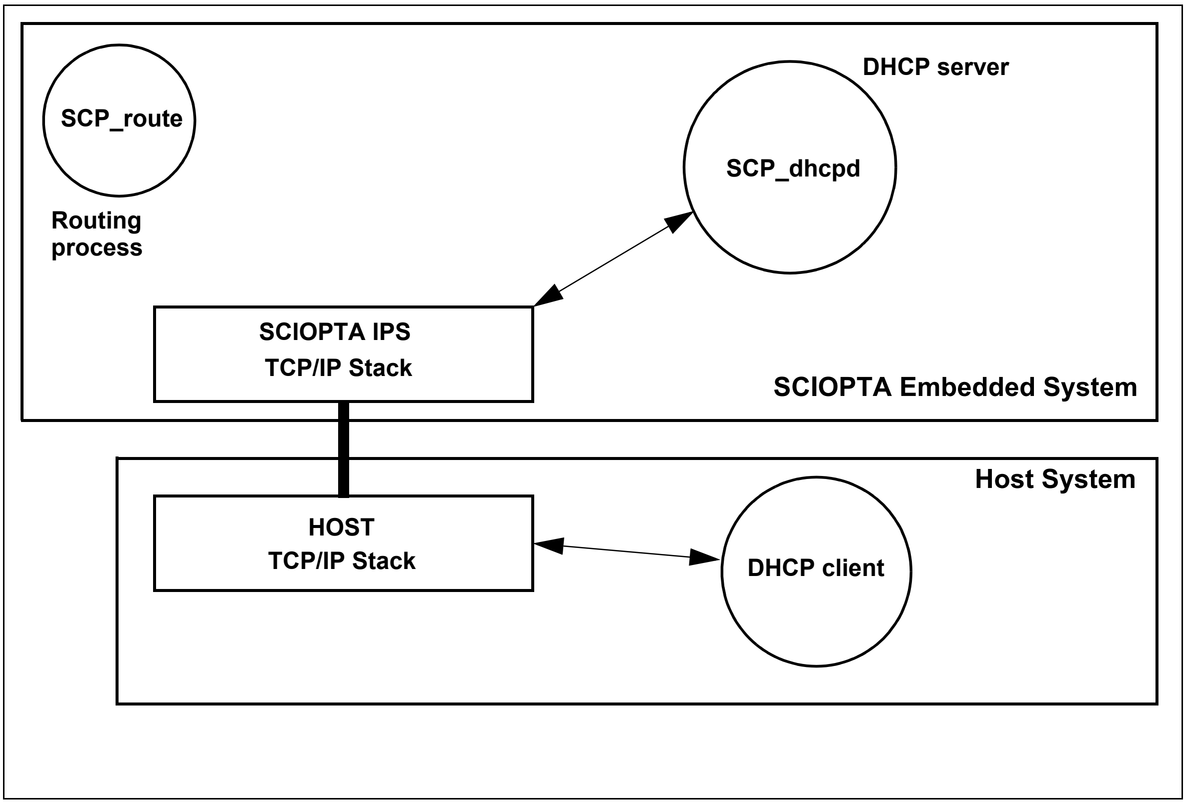

Figure 9-3: Getting Started DHCP Server Example

9.3.2 IPS DHCP Server Example - Windows Host

9.3.2.1 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\dhcpd\
- Network cable connected between your target board and your host workstation or to your network.
- A DHCP client running in the connected network such as TFTPD32.
- Optionally a network protocol analyser running on your host computer such as Ethereal.
- Source-level emulator/debugger for ARM connected to the target board.
- SCIOPTA ARM - Base Package.
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - IPS Internet Protocols
- SCIOPTA ARM - IPS Internet Protocols Applications
- 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.

9.3.2.2 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_dhcpd**) 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_dhcpd).
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\dhcpd\<board> to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_dhcpd).
14. Browse to the working directory of your Eclipse project (e.g. c:\myproject\sciopta\ips_dhcpd).
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_dhcpd).
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_dhcpd**) 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_dhcpd\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. You can set breakpoints in the target dhcp server application and watch the behaviour.
31. For have a closer look at the network traffic generated by this example you can run a network analyser such as Ethereal on your host computer and trace the network messages.

9.4 DHCP Client Configuration

9.4.1 Introduction

SCIOPTA DHCP is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use DHCP. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User's Guide for more information about the IPS stack and how to configure it.

DHCP Client 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 - Target Manual for your selected processor for more information about using sconf.exe and the static configuration process.

   The static object will be generated and started automatically at system start.
2. SCIOPTA DHCP Client configuration which configures the DHCP client process (SCP_dhcpclient).

9.4.2 DHCP Client Module Configuration

In our standard getting started examples 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.

Telnet is placed in the "user" module. But you are free to split your application differently into other modules or put all processes in one module.

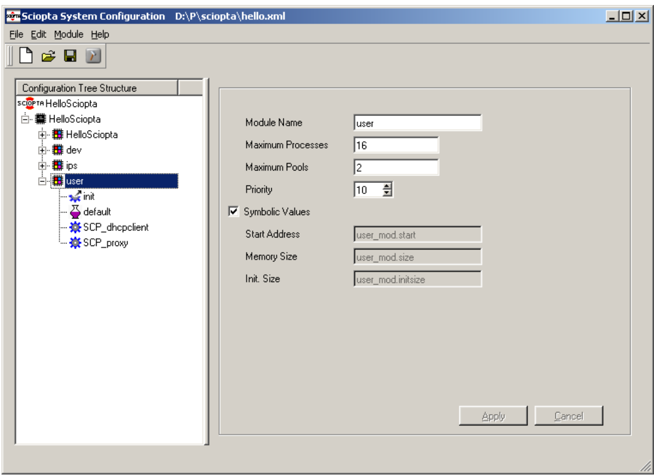

Figure 9-4: DHCP client module "user"

9.4.2.1 DHCP Client 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 256 bytes. You could optimize this stack by doing a stack analysis using the DRUID system level debugger.

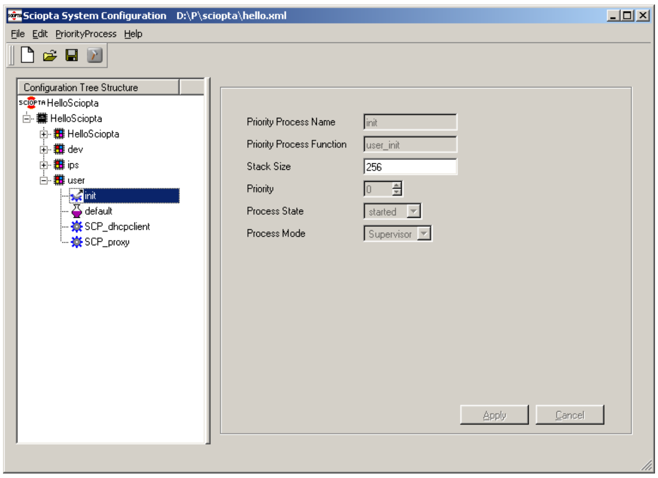

Figure 9-5: DHCP client module init process

9.4.2.2 DHCP Client Module Default Pool

The pool parameters must be designed to fit the requirements of the DHCP client messages buffer sizes.

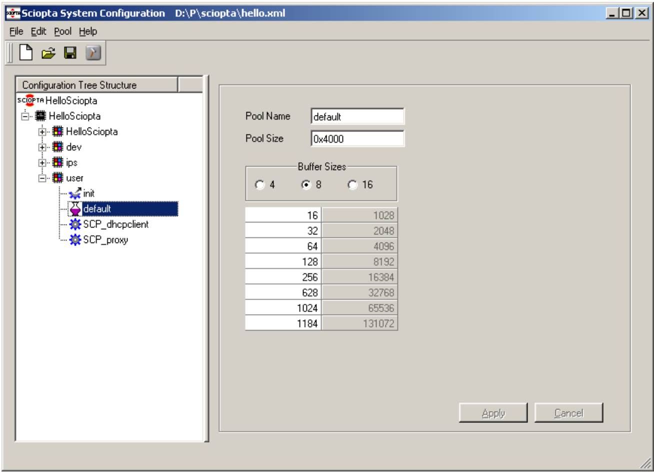

Figure 9-6: DHCP client module default pool

9.4.2.3 Proxy Process

The SCP_proxy process is setting routes which it gets from process SCP_dhcpclient. The SCP_proxy process was introduced to better follow the system behaviour by scanning messages.

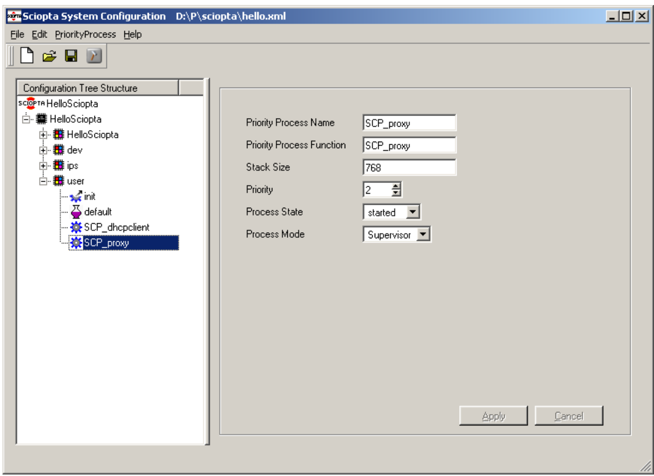

Figure 9-7: DHCP client proxy process SCP_proxy

9.4.2.4 DHCP Client Process

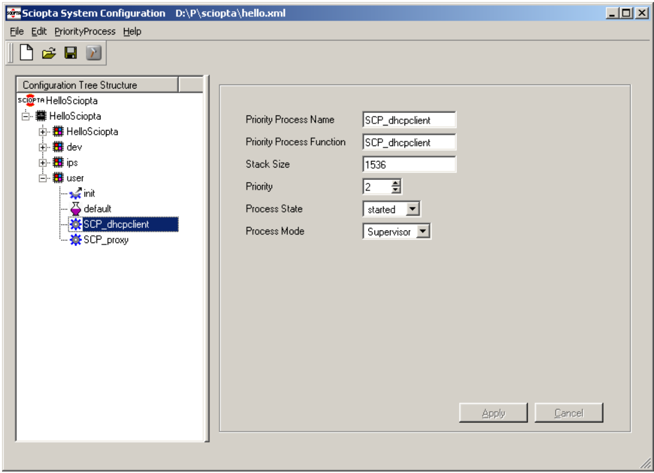

Figure 9-8: DHCP client process SCP_dhcpclient

9.4.3 DHCP Client Configuration

In the previous chapters we have described how to configure the static modules, pools and processes for SCIOPTA DHCP Client. 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 IPS DHCP Client you need also to configure some DHCP Client parameters depending on the DHCP specified properties. These parameters are defined by calling the DHCP Client process at start-up (SCP_dhcpclient).

In the SCIOPTA IPS DHCP Client example we have include the declaration of this process in the file **dhcpclient.c** which can be found in the examples delivery.

File location: <install_folder>\sciopta\<version>\exp\ips\common\dhcp\dhcpclient.c

9.4.3.1 DHCP Client Process Configuration

The DHCP client process needs to call the DHCP client process function (**dhcp_client**) which is included in the DHCP library.

Syntax

```
void dhcp_client(
   const char          *ethDev,
   const char          *serverIp,
   const char          *bootFile,
   __u16               maxMsgSize
);
```

Parameters

ethDev

Pointer to the name of the ethernet device on which the DHCP shall run.

serverIp

Pointer to the name of the DHCP server address.

bootFile

NULL. Reserved for later use.

maxMsgSize

The maximum message size used by **dhco_client**.

Example

```
#include <sciopta.h>
#include <bootp/dhcpclient.h>

#define DHCPC_DEV_NAME      "eth0"
#define DHCP_BOOTFILE_NAME  ""
#define DHCP_SERVER_NAME    ""
#define DHCP_PKT_SIZE       1000

SC_PROCESS (SCP_dhcpclient)
{
  dhcp_client ( DHCPC_DEV_NAME,
                DHCP_BOOTFILE_NAME,
                DHCP_SERVER_NAME,
                DHCP_PKT_SIZE);
}
```

9.4.3.2 DHCP Client Proxy Process

The SCP_proxy process is setting routes which it gets from process SCP_dhcpclient. The SCP_proxy process was introduced to better follow the system behaviour by scanning messages.

Example

```
#include <sciopta.h>
#include <sdd/sdd.h>
#include <sdd/sdd.msg>
#include <ips/ipv4.h>
#include <ips/router.h>
#include <logd/logd.h>

union sc_msg {
  sc_msgid_t id;
  ipv4_route_t route;
};

SC_PROCESS (SCP_proxy)
{
  sc_msg_t msg;
  sdd_obj_t *devman;
  sdd_obj_t *router;
  sc_pid_t from;
  char source[16];
  char netmask[16];
  char gateway[16];
  logd_t *logd;

  static const sc_msgid_t select[2] = { IPV4_ROUTE_ADD, 0 };
  static const sc_msgid_t forward[3] = { SDD_ERROR, IPV4_ROUTE_ADD_REPLY, 0 };
```

```
  devman = sdd_manGetRoot ("/SCP_devman", "devman", SC_DEFAULT_POOL,
                           SC_FATAL_IF_TMO);
  if (!devman) {
    sc_miscError (SC_ERR_FATAL, 1);
  }
  router = sdd_manGetByName (devman, "router");
  sdd_objFree (&devman);
  if (!router) {
    sc_miscError (SC_ERR_FATAL, 2);
  }

  logd = logd_new ("/SCP_logd", LOGD_INFO, "proxy", SC_DEFAULT_POOL,
                   SC_FATAL_IF_TMO);

  logd_printf (logd, LOGD_INFO, "DHCP route proxy started\n");

  for (;;) {
    msg = sc_msgRx (SC_ENDLESS_TMO, (const void *) select, SC_MSGRX_MSGID);
    switch (msg->id) {
    case IPV4_ROUTE_ADD:

      if (*(__u32 *) msg->route.router == 0) {
    ipv4_ntoa (msg->route.source, source, 16);
    ipv4_ntoa (msg->route.netmask, netmask, 16);

    logd_printf (logd, LOGD_INFO, "IP Address: %s  Netmask: %s\n", source,
                 netmask);
      }
      else if (*(__u32 *) msg->route.netmask == 0) {
    ipv4_ntoa (msg->route.router, gateway, 16);

    logd_printf (logd, LOGD_INFO,"Default Gateway: %s\n", gateway);
      }
      else {
    ipv4_ntoa (msg->route.router, gateway, 16);

    logd_printf (logd, LOGD_INFO,"Next Router: %s  Netmask: %s\n", gateway,
                 netmask);
      }

      from = sc_msgSndGet(&msg);
      msg->route.device.object.base.handle = router->base.handle;
      sc_msgTx (&msg, router->controller, 0);
      msg = sc_msgRx (SC_ENDLESS_TMO, (const void *) forward, SC_MSGRX_MSGID);
      sc_msgTx (&msg, from , 0);
      break;
    default:
      sc_miscError(SC_ERR_FATAL,(sc_extra_t)msg);
      break;
    }
  }
}
```

9.5 DHCP Server Configuration

9.5.1 Introduction

SCIOPTA DHCP is an application on top of the SCIOPTA IPS TCP/IP stack. The IPS stack needs to be configured before you can use DHCP. Please consult the configuration chapter of the SCIOPTA - IPS Internet Protocols, User's Guide for more information about the IPS stack and how to configure it.

DHCP Server 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 - Target Manual for your selected processor for more information about using sconf.exe and the static configuration process.

   The static object will be generated and started automatically at system start.
2. SCIOPTA DHCP Server configuration which configures the DHCP server daemon process (**SCP_dhcpd**).

9.5.2 DHCP Server Module Configuration

In our standard getting started examples 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.

Telnet is placed in the "user" module. But you are free to split your application differently into other modules or put all processes in one module.

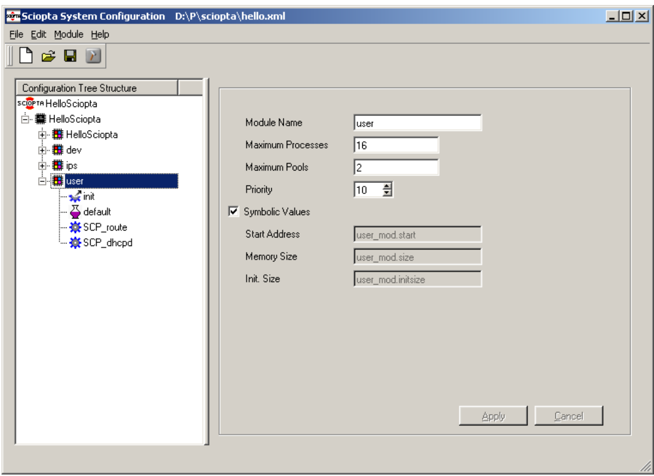

Figure 9-9: DHCP server module "user"

9.5.2.1 DHCP Server 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 256 bytes. You could optimize this stack by doing a stack analysis using the DRUID system level debugger.

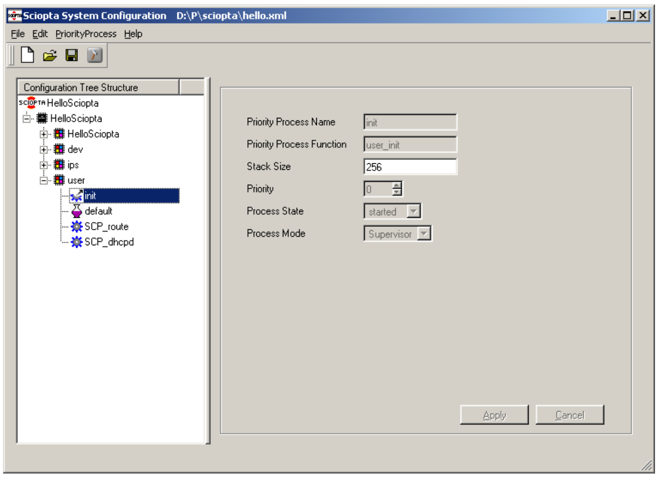

Figure 9-10: DHCP server module init process

9.5.2.2 DHCP Server Module Default Pool

The pool parameters must be designed to fit the requirements of the DHCP client messages buffer sizes.

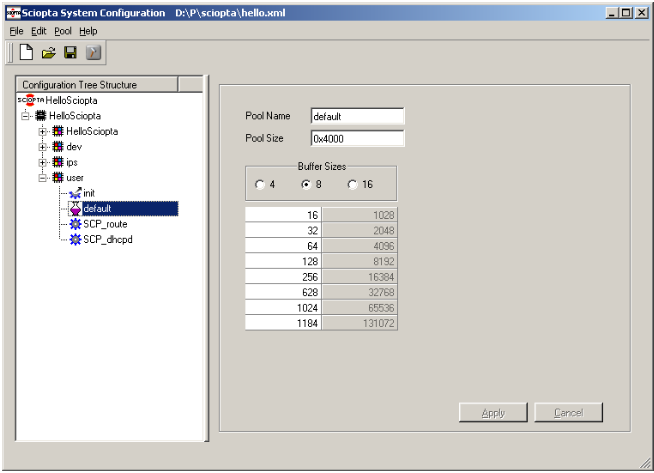

Figure 9-11: DHCP server module default pool

9.5.2.3 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.

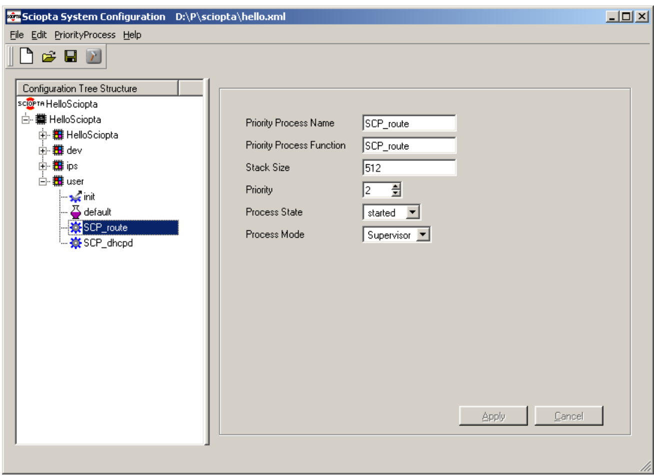

Figure 9-12: Routing process SCP_route

9.5.2.4 DHCP Server Daemon Process

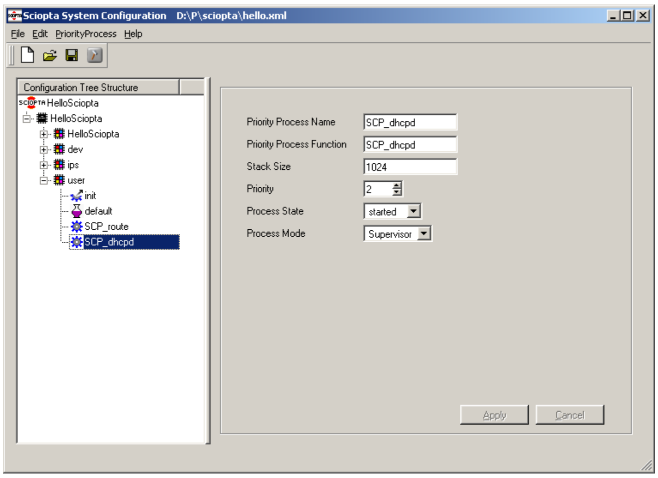

Figure 9-13: DHCP server daemon process SCP_dhcpd

9.5.3 DHCP Server Configuration

In the previous chapters we have described how to configure the static modules, pools and processes for SCIOPTA DHCP Server. 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 IPS DHCP Server you need also to configure some DHCP Server parameters depending on the DHCP specified properties. These parameters are defined by calling the DHCP Server daemon process at start-up (**SCP_dhcpd**).

In the SCIOPTA IPS DHCP Client example we have include the declaration of this process in the file **dhcpclient.c** which can be found in the examples delivery.

File location: <install_folder>\sciopta\<version>\exp\ips\common\dhcp\dhcpdaemon.c

9.5.3.1 DHCP Server Daemon Process Configuration

The DHCP server daemon process needs to call the DHCP server process function (**dhcp_server**) which is included in the DHCP library.

Syntax

```
void dhcp_server(
   dhcp_zone_t           *defaultConfig,
   sc_pool_id            plid
);
```

Parameters

defaultConfig

Pointer to the default configuration.

plid

Pool ID for DHCP server messages.

Example

```
#include <sciopta.h>

#include <ips/addr.h>
#include <bootp/dhcpserver.msg>
#include <bootp/dhcpserver.h>

/**
 * Configuration for DHCP daemon process
 */
dhcps_zone_t defaultZone = {
  /* id */        0,
  /* error */     0,
  /* subnet */    { 10,  0,   3,   0 },
  /* netmask */   { 255, 255, 255, 0 },
  /* router*/     { 10,  0,   3,   2 },
  /* leasetime */ 20,
  /* range */
  {
    { 10,  0,  3,   180 },
    { 10,  0,  3,   200 }
  },
  /* nameServer */
  {
    { 10,  0,  3,   11  },
    { 147, 86, 130, 1   },
    { 0,   0,  0,   0   }
  },
};

/**
 * DHCP daemon process definition
 */
SC_PROCESS (SCP_dhcpd)
{
  dhcp_server (&defaultZone, SC_DEFAULT_POOL);
  sc_procKill (SC_CURRENT_PID, 0);
}
```

9.6 DHCP System Building

Please consult chapters "System Building" of the SCIOPTA ARM - Kernel, User's Guide and SCIOPTA ARM - IPS Internet Protocols, User's Guide for general information about the system building process.

You will find there information about:

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

9.6.1 Include Files

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

9.6.2 SCIOPTA ARM IPS DHCP Libraries

9.6.2.1 Delivered IPS DHCP Libraries

IPS DHCP	libdhcp_**XY**.a	GCC
	dhcp_**XY**.l	ARM RealView
	dhcp_**XY**.r79	IAR Systems

9.6.2.2 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.

9.6.2.3 IPS DHCP Libraries “Y”

For SCIOPTA IPS DHCP 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.

Please consult chapter ‘IPS Internet Protocols Libraries “Y” ‘ of the SCIOPTA ARM - IPS Internet Protocols, User’s Guide for a table showing the included features for each of the three IPS levels.

9.7 DHCP Structures

9.7.1 DHCP Server Configuration Structure dhcp_zone_t

The HTTP configuration structure is used in a message to configure the DHCP server daemon.

It contains the message ID as it is a standard SCIOPTA message.

```
typedef struct dhcps_zone_s {
   sc_msgid_t          id;
   sc_errcode_t        error;
   __u8                subnet[4];
   __u8                netmask[4];
   __u8                router[4];
   __u32               leaseTime;
   __u8                range[2][4];
   __u8                nameServer[3][4];
} dhcps_zone_t;
```

Members

id

Standard SCIOPTA message ID.

error

DHCP server error code.

subnet

Subnet definition.

netmask

Network mask.

router

Router address.

leasetime

DHCP server lease time.

range

Range IP address array.

leasetime

Name servers IP address array.

Header

<install_dir>\sciopta\<version>\include\bootp\dhcpserver.msg

9.8 Requests for Comments RFC - DHCP

In this chapter we have listed the important RFCs which are important for the DHCP implementation.

This list might not be complete and there is no guarantee that SCIOPTA DHCP complies to all proposals, specifications, standards and information in these RFCs.

RFC	Description	Date
1534	Interoperation Between DHCP and BOOTP	October 1993
2131	Dynamic Host Configuration Protocol	March 1997
2132	DHCP Options and BOOTP Vendor Extensions	March 1997
2241	DHCP Options for Novell Directory Services	November 1997
2242	NetWare/IP Domain Name and Information	November 1997
2322	Management of IP numbers by peg-dhcp	April 1998
2485	DHCP Option for The Open Group's User Authentication Protocol	January 1999
2563	DHCP Option to Disable Stateless Auto-Configuration in IPv4 Clients	May 1999
2610	DHCP Options for Service Location Protocol	June 1999
2855	DHCP for IEEE 1394	June 2000
2937	The Name Service Search Option for DHCP	September 2000
2939	Procedures and IANA Guidelines for Definition of New DHCP Options and Message Types	September 2000
3004	The User Class Option for DHCP	November 2000
3011	The IPv4 Subnet Selection Option for DHCP	November 2000
3046	DHCP Relay Agent Information Option	January 2001
3118	Authentication for DHCP Messages	June 2001
3203	DHCP reconfigure extension	December 2001
3256	The DOCSIS (Data-Over-Cable Service Interface Specifications) Device Class DHCP (Dynamic Host Configuration Protocol) Relay Agent Information Sub-option	April 2002
3315	Dynamic Host Configuration Protocol for IPv6 (DHCPv6)	July 2003
3319	Dynamic Host Configuration Protocol (DHCPv6) Options for Session Initiation Protocol (SIP) Servers	July 2003
3361	Dynamic Host Configuration Protocol (DHCP-for-IPv4) Option for Session Initiation Protocol (SIP) Servers	August 2002
3396	Encoding Long Options in the Dynamic Host Configuration Protocol (DHCPv4)	November 2002
3397	Dynamic Host Configuration Protocol (DHCP) Domain Search Option	November 2002
3442	The Classless Static Route Option for Dynamic Host Configuration Protocol (DHCP) version 4	December 2002
3456	Dynamic Host Configuration Protocol (DHCPv4) Configuration of IPsec Tunnel Mode	January 2003
3495	Dynamic Host Configuration Protocol (DHCP) Option for CableLabs Client Configuration	March 2003
3527	Link Selection sub-option for the Relay Agent Information Option for DHCPv4	April 2003

RFC	Description	Date
3594	PacketCable Security Ticket Control Sub-Option for the DHCP CableLabs Client Configuration (CCC) Option	September 2003
3633	IPv6 Prefix Options for Dynamic Host Configuration Protocol (DHCP) version 6	December 2003
3634	Key Distribution Center (KDC) Server Address Sub-option for the Dynamic Host Configuration Protocol (DHCP) CableLabs Client Configuration (CCC) Option	December 2003
3646	DNS Configuration options for Dynamic Host Configuration Protocol for IPv6 (DHCPv6)	December 2003
3679	Unused Dynamic Host Configuration Protocol (DHCP) Option Codes	January 2004
3736	Stateless Dynamic Host Configuration Protocol (DHCP) Service for IPv6	April 2004
3825	Dynamic Host Configuration Protocol Option for Coordinate-based Location Configuration Information	July 2004

10 SCIOPTA ARM Releases

10.1 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.

10.2 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

11 Manual Versions

11.1 Manual Version 2.1

- Chapter 2 Installation, Main Installation Window screen shot changed.

11.2 Manual Version 2.0

- The following manuals have been combinded in this new SCIOPTA ARM - IPS Applications, User’s Guide:
    - SCIOPTA - IPS Applications, User’s Guide and Reference Manual Version 1.1
    - SCIOPTA - ARM Target Manual

11.3 Former SCIOPTA - IPS Applications, User’s Guide Versions

11.3.1 Manual Version 1.1

- Chapter 2 Web Server, Figure 2-1 line: “http://<host_name>/<module_name>/<page_process_name>” modified into: http://<host_name>/<path>.
- Chapter 2.1 Web Server Description, second page rewritten.
- Chapter 2.2.3 Web Server Page Module Variable, removed.
- Chapter 2.5 Web Server Message Interface. Whole chapter rewritten.
- Chapter 2.4 Web Server, Structures, new chapter.
- Chapter 2 Web Server, all **http_t *** changed to **http_t NEARPTR** to support SCIOPTA 16 Bit systems.
- All **sdd_obj_t *** changed to **sdd_obj_t NEARPTR** to support SCIOPTA 16 Bit systems.
- Chapter 3 SMTP, all **smtp_t *** changed to **smtp_t NEARPTR** to support SCIOPTA 16 Bit systems.
- Chapter 4 TFTP, all **tftp_handle_t *** changed to **tftp_handle_t NEARPTR** to support SCIOPTA 16 Bit systems.
- Chapter 4.1 TFTP Description, TFTP client **and server** are implemented.
- Chapter 4.5.10 tftp_rec2Mem, new chapter.
- Chapter 4.5.11 tftp_sendMem, new chapter.
- Chapter 4.6.4 tftp_get2File, parameter **file** replaced by **dev**.
- Chapter 4.6.8 tftp_putFile, parameter **file** replaced by **dev**.

11.3.2 Manual Version 1.0

Initial version.

11.4 Former SCIOPTA ARM - Target Manual Versions

11.4.1 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.

11.4.2 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.

11.4.3 Manual Version 2.0

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

11.4.4 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).

11.4.5 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.

12 Index

Symbols

A

B

C

D