SCIOPTA

High Performance
Real-Time Operating Systems

ARM
Flash File System

User's Guide

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. S04129RL1

Table of Contents

SCIOPTA

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 full featured operating system environment includes:

- KRN - Pre-emptive Multi-Tasking Real-Time Kernel
- BSP - Board Support Packages
- IPS - Internet Protocols (TCP/IP)
- IPS Applications - Internet Protocols Applications (Web Server, TFTP, DNS, DHCP, Telnet, SMTP etc.)
- SFATFS - FAT File System
- SFFS - FLASH File System, NOR
- SFFSN - FLASH File System, NAND support
- USBD - Universal Serial Bus, Device
- USBH - Universal Serial Bus, Host
- DRUID - System Level Debugger
- SCIOPTA PEG - 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** Flash File System is designed for high flexibility allowing the user to adapt the file handling for various embedded applications and follows closely the **SCIOPTA** device driver concept. This manual includes a description of the SCIOPTA Flash File System concept and gives an introduction how to use the **SCIOPTA** Flash File System. Detailed descriptions of the file system structure and interfaces are included.

In the reference part all specific file system messages and functions are listed.

Please consult also the SCIOPTA ARM - Kernel, User's Guide and the SCIOPTA ARM - SDD Device Driver, User's Guide.

1.3 SCIOPTA Flash File System Overview

SCIOPTA SFFS is an extremely robust embedded flash file system guaranteed to be 100% safe against power-failure. It is easy-to-use, scalable and can easy be ported to all SCIOPTA ARM target systems. SFFS has minimal requirements from the embedded system.

Some of the features include:

- API
    - Standard API
    - SCIOPTA SDD Function Interface
    - Multi-user interface
    - Long file names
    - Unicode 16 support
- NOR Flash Support
    - Wear-leveling
    - Bad block handling
    - Easy porting for all known device types
    - Sample driver with porting description
- Atmel DataFlash Support
    - Wear-leveling
    - All devices supported
    - Manages the 10K writes/sector limitation
    - Failsafe implementation of DataFlash interface
- NAND Flash Support
    - Wear-leveling
    - Bad-block management
    - ECC algorithm
    - Easy porting for all known device types
    - Sample driver with porting description

1.4 System Features

1.4.1 Power Fail Safety

The SCIOPTA Flash File System is entirely power fail safe. The system may be stopped at any point and restarted and no data will be lost; the previous completed state of the file system will be restored.

1.4.2 Long File Names

The SCIOPTA Flash File System supports file names of almost unlimited length. File name handling is efficient - it is built from a chain of small fragments taken from the descriptor block.

1.4.3 Multiple Volumes

The SCIOPTA Flash File System supports multiple volumes. Each volume must have its own driver routine, which normally has its own physical handler (except for the RAM drive).

1.4.4 Multiple Open Files in a Volume

The SCIOPTA Flash File System allows multiple files to be opened simultaneously on a volume or on different volumes.

1.4.5 Case Sensitive File Names

By default the file system uses case insensitive names.

1.4.6 Separator Character

You can define the file separator character. By default this is a backslash.

1.4.7 Unicode

It is possible to enable the use of API functions for Unicode.

1.4.8 Static Wear

Flash devices are usually manufactured to a specification which includes a guaranteed number of write-erase cycles that can be performed on each block before it may develop a fault. Because of this it is important to use the blocks in a device "evenly" if the device is to be used for its maximum lifetime.

The SCIOPTA Flash File System uses a process called dynamic wear to allocate the least use blocks from those available. However, in systems where there are large areas of static data (e.g., the executable binary for the system) then the areas where this is stored may be written only once leaving a relatively small section of the device to handle the much more heavily used files.

For this reason a process called static wear is introduced.

1.4.9 Free Block Allocation

The system includes two different algorithms for allocating blocks in the file system. One finds the first block it finds with an available sector; the second (default) allocates the block with the most free sectors.

1.5 HCC Safe Flash File System

The SCIOPTA Flash File System uses and integrates the Safe Flash File System from HCC Embedded. The user can directly access the HCC Safe Flash File System API (HCC-Mode). This is the preferred standard method when using the file system without an MMU and memory protection.

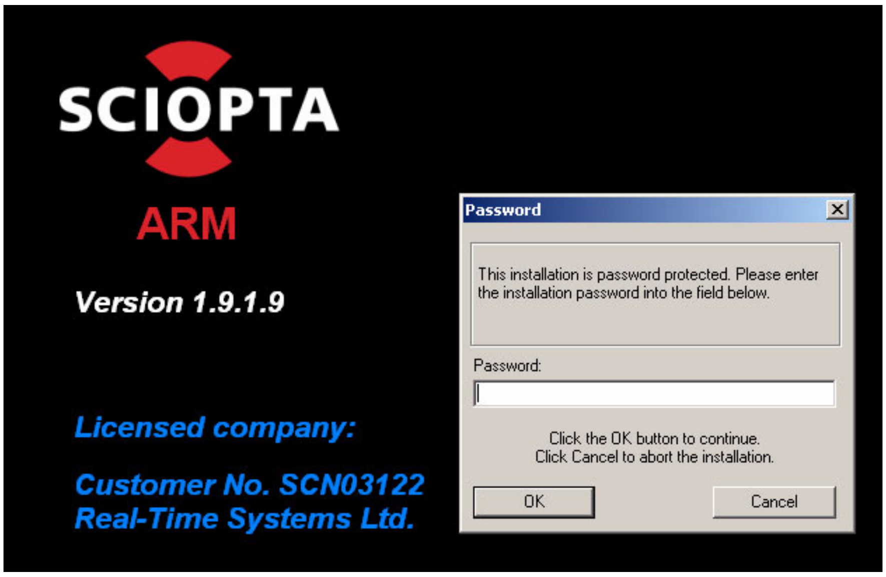
1.6 SCIOPTA Flash File System

If an MMU with memory protection is used, the SCIOPTA Flash File System must be accessed by using the SCIOPTA device driver method (SDD-Mode). You can directly use predefined SDD and SFS messages or the SDD and SFS function interface.

2 Installation

2.1 Introduction

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



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

Password:

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

Figure 2-1: Main Installation Window

3 Getting Started

3.1 Introduction

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

Please Note:

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

3.2 Getting Started - Flash File System RAM Disk Example

3.2.1 Description

In this SCIOPTA Flash File System getting started example we will create a file system in the RAM of the embedded system using a RAM-Disk driver. RAM disk is used to show how to use the SCIOPTA Flash File System on application level and to have an example which will run on many boards equipped with any kind of RAM. Examples using real flash devices are available from SCIOPTA.

The example application consists of two files.

- The file fs_setup.c contains the file system setup code included in the process SCP_fsSetup. In the same file the process SCP_flash is included which starts the file system.
- The file effstest.c contains some user code to show file system functionality included in the process SCP_effsTest.

The process SCP_fsSetup gets first the root manager and the file system from the file system manager (SCP_devman). It then mounts the drive and formats it if needed.

The process SCP_flash is the basic file system (file system process) and is initialized in the file fs_setup.c

The user process SCP_effsTest is doing different file system calls to show the usage of the SCIOPTA Flash File System.

3.2.2 Flash File System RAM Disk Example Block Diagram

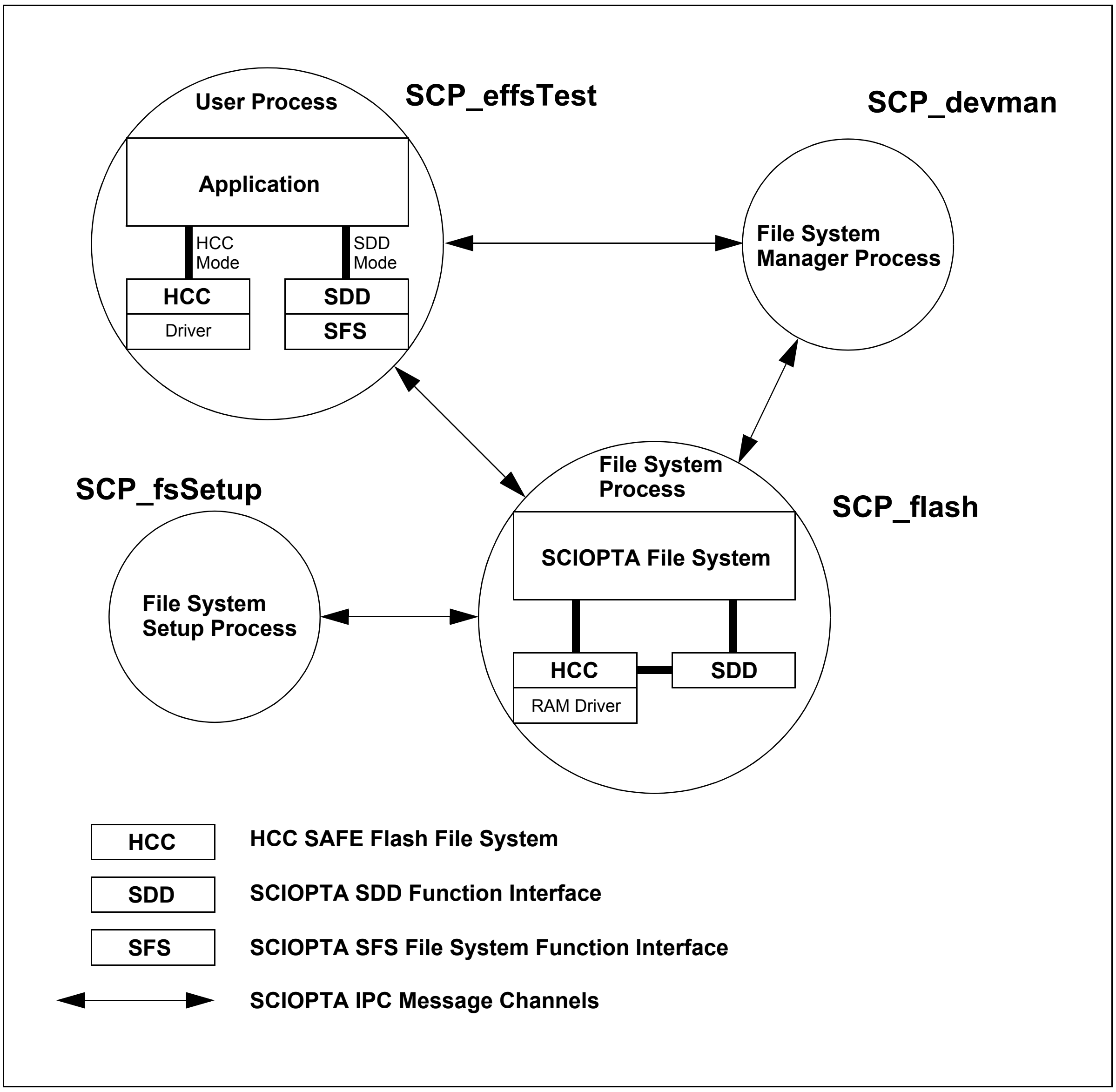

SCP_effsTest
User Process
Application
HCC Mode
SDD Mode
HCC
Driver
SDD
SFS
SCP_devman
File System Manager Process
SCP_fsSetup
File System Setup Process
SCP_flash
File System Process
SCIOPTA File System
HCC
RAM Driver
SDD
HCC
HCC SAFE Flash File System
SDD
SCIOPTA SDD Function Interface
SFS
SCIOPTA SFS File System Function Interface
SCIOPTA IPC Message Channels

Figure 3-1: SCIOPTA Flash File System Block Diagram

3.2.3 Makefile and GNU GCC

3.2.3.1 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.
- 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 separate SCIOPTA ARM tools CD delivery.
- A debugger/emulator for ARM which supports GNU GCC such as the iSYSTEM winIDEA Emulator/Debugger for ARM or the Lauterbach TRACE32 debugger for ARM.
- Target board which is included in the SCIOPTA examples. For each board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\. Actually supported are:
  - **Atmel AT91SAM7SE-EK**
  - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - Flash File System
- This getting started example is sending some specific log messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board.
  - For the **Atmel AT91SAM7SE-EK** board COM1 is used.
  - For the **Atmel AT91SAM9261-EK** board COM1 is used.

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

3.2.3.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide.
2. Be sure that the GNU GCC compiler bin directory is included in the PATH environment variable as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide.
3. Be sure that the %SCIOPTA_HOME%\bin\win32 directory is included in the PATH environment variable as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide. This will give access to the GNU make utility.
4. Create an example working directory at a suitable place.
5. Copy the script **copy_files.bat** (for running the SDD-Mode example) or the **copy_files_hcc.bat** (for running the HCC-Mode example) from the example directory for your selected target boards:

   <install_folder>\sciopta\<version>\exp\sfs\arm\flashfs_ram\<board>\

   to your project folder.

6. Open a Windows Explorer or a Command Prompt window.
7. Go to the working directory of your project.
8. Type **copy_files.bat** (for the SDD-Mode example) or **copy_files_hcc.bat** (for the HCC-Mode example) to execute the batch file and the file copy process.
9. Launch the SCIOPTA configuration utility **SCONF** from the desktop or the Star menu.
10. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF** (menu: **File** > **Open**).
11. 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
    - sconf.h.
12. Execute the makefile for your selected target board:

    - For the **Atmel AT91SAM7SE-EK** board: **gnu-make BOARD_SEL=15**
    - For the **Atmel AT91SAM9261-EK** board: **gnu-make BOARD_SEL=11**

    The executable example file sciopta.elf is created.
13. Launch your ARM source-level emulator/debugger and load the resulting sciopta.elf.
14. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. The speed must be set to 115200 baud.
15. For some emulators/debuggers specific project and board initialization files can be found in the created project folder or in other example directories.
16. Now you can start the system and check the log messages on your host terminal window.
17. You can also set breakpoints anywhere in the example system and watch the behaviour.

3.2.4 Eclipse IDE and GNU GCC

3.2.4.1 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.
- 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 separate SCIOPTA ARM tools CD delivery.
- A debugger/emulator for ARM which supports GNU GCC such as the iSYSTEM winIDEA Emulator/Debugger for ARM or the Lauterbach TRACE32 debugger for ARM.
- Target board which is included in the SCIOPTA examples. For each board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\. Actually supported are:
  - **Atmel AT91SAM7SE-EK**
  - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - Flash File System.
- Eclipse Version 3.3.1.1 (special distribution including SCIOPTA plug-ins), included on the separate SCIOPTA PowerPC Tools CD.
- In order to run the Eclipse Platform you also need the Sun Java 2 SDK, Standard Edition for Microsoft Windows.
- This getting started example is sending some specific log messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board.
  - For the **Atmel AT91SAM7SE-EK** board COM1 is used.
  - For the **Atmel AT91SAM9261-EK** board COM1 is used.

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

3.2.4.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide.
2. Be sure that the GNU GCC compiler bin directory is included in the PATH environment variable as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide.
3. Be sure that the %SCIOPTA_HOME%\bin\win32 directory is included in the PATH environment variable as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide. This will give access to the GNU make utility.
4. Create a project folder to hold all project files (e.g. d:\myprojects\sciopta) if you have not already done it for other getting-started projects.
5. Launch Eclipse. The Workspace Launcher window opens.
6. Select your created project folder (e.g. c:\myproject\sciopta) as your workspace (by using the Browse button).

7. Click the **OK** button. The workbench windows opens. Close the Welcome Tab.
8. Maximize the workbench and deselect **“Build Automatically”** in the **Project** menu.
9. Open the **New Project** window (menu: **File -> New -> C Project**).
10. Select arm-gcc ELF (Gnu)from the menu in the left window.
11. The New Project window opens. Enter the new project name: **sfs_effsram** and click on the **Finish** button.
12. Select the new created project in the browser window and close the project (menu: **Project -> Close Project**).
13. The next steps we will execute outside Eclipse.
14. Copy the script **copy_files.bat** from the example directory for your selected target boards:

    <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

    to your project folder.
15. Open a Command Prompt window.
16. Go to the example working directory of your project.
17. Type **copy_files.bat** (for the SDD-Mode example) or **copy_files_hcc.bat** (for the HCC-Mode example) to execute the batch file and the file copy process.
18. Launch the SCIOPTA configuration utility **SCONF** from the desktop or the Star menu.
19. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF**.
    File > Open
20. 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
    - sconf.h
21. Swap back to the Eclipse workbench. Make sure that the kernel hello project (**sfs_effsram**) is highlighted and open the project (menu: **Project > Open Project**).
22. Expand the project by selecting the [+] button and make sure that the **sfs_effsram** project is highlighted.
23. Type the F5 key (or menu: **File > Refresh**).
24. Now you can see all files in the Eclipse Navigator window.
25. Select the Console window at the bottom of the Eclipse workbench to see the project building output.
26. Be sure that the project (**sfs_effsram**) is high-lighted and build the project (menu: **Project > Build Project**).
27. The executable (sciopta.elf) will be created in the debug folder of the project.
28. Launch your ARM source-level emulator/debugger and load the resulting sciopta.elf.
29. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. The speed must be set to 115200 baud.
30. For some emulators/debuggers specific project and board initialization files can be found in the created project folder or in other example directories.
31. Now you can start the system and check the log messages on your host terminal window.
32. You can also set breakpoints anywhere in the example system and watch the behaviour.

3.2.5 iSYSTEM winIDEA and GNU GCC

3.2.5.1 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.
- 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 separate SCIOPTA ARM tools CD delivery.
- Target board which is included in the SCIOPTA examples. For each board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\. Actually supported are:
    - **Atmel AT91SAM7SE-EK**
    - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - Flash File System.
- iSYSTEM iC3000 - winIDEA Emulator/Debugger for ARM.
- This getting started example is sending some specific log messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board.
    - For the **Atmel AT91SAM7SE-EK** board COM1 is used.
    - For the **Atmel AT91SAM9261-EK** board COM1 is used.

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

3.2.5.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in the chapter "Installation" of the SCIOPTA ARM - Kernel, User's Guide.
2. Be sure that the GNU GCC compiler bin directory is included in the PATH environment variable as described in the chapter "Installation" of the SCIOPTA ARM - Kernel, User's Guide.
3. Create an example working directory at a suitable place.
4. Copy the script **copy_files.bat** (for running the SDD-Mode example) or the **copy_files_hcc.bat** (for running the HCC-Mode example) from the example directory for your selected target boards:

   <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

   to your project folder.
5. Open a Command Prompt window.
6. Go to the working directory of your project.
7. Type **copy_files.bat** (for the SDD-Mode example) or **copy_files_hcc.bat** (for the HCC-Mode example) to execute the batch file and the file copy process.

8. Launch the SCIOPTA configuration utility **SCONF** from the desktop or the Star menu.
9. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF**.
   File > Open
10. 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
    - sconf.h.
11. Launch the iSYSTEM iC3000 - winIDEA Emulator/Debugger for ARM.
12. Open the example workspace (menu: **File > Workspace > Open Workspace...**). Browse to your example project directory and select the workspace file iC3000.jrf.
13. Make the project (menu: **Project > Make**) or type the **F7** button.
14. The executable (sciopta.elf) will be created in the debug folder of the project.
15. Download the sciopta.elf file into the target system (menu: **Debug > Download**) or type the **Ctrl+F3** button.
16. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. The speed must be set to 115200 baud.
17. Run the system (menu: **Debug > Run**) or type the **F5** button.
18. Now you can check the log messages on your host terminal window.
19. You can also set breakpoints anywhere in the example system and watch the behaviour.

3.2.6 IAR Embedded Workbench

3.2.6.1 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.
- IAR Embedded Workbench for ARM including the following main components:
  - IAR Assembler for ARM 4.42A
  - IAR C/C++ Compiler for ARM 4.42A
  - IAR Embedded Workbench IDE 4.8
  - IAR XLINK 4.60I
  - IAR C-SPY including suitable target connection
- Target board which is included in the SCIOPTA examples. For each board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\. Actually supported are:
  - **Atmel AT91SAM7SE-EK**
  - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - Flash File System
- This getting started example is sending some specific log messages to a selected UART of the board. To display these messages on your host PC you can optionally connect a serial line from a COM port of your host PC to an UART port of your selected target board.
  - For the **Atmel AT91SAM7SE-EK** board COM1 is used.
  - For the **Atmel AT91SAM9261-EK** board COM1 is used.

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

3.2.6.2 Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide.
2. Be sure that the %SCIOPTA_HOME%\bin\win32 directory is included in the PATH environment variable as described in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide. This will give access to the sconf.exe utility. Some IAREW examples might call sconf.exe directly from the workbench to do the SCIOPTA configuration.
3. Be sure that the IAR Embedded Workbench is installed as described in the IAREW user manuals.
4. Create an example working directory at a suitable place.

5. Copy the script **copy_files_iar.bat** (for running the SDD-Mode example) or the **copy_files_iar_hcc.bat** (for running the HCC-Mode example) from the example directory for your selected target boards:

   <install_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

   to your project folder.
6. Open a Command Prompt window.
7. Go to the working directory of your project.
8. Type **copy_files_iar.bat** (for the SDD-Mode example) or **copy_files_iar_hcc.bat** (for the HCC-Mode example) to execute the batch file and the file copy process.
9. Launch the SCIOPTA configuration utility **SCONF** from the desktop or the Star menu.
10. Load the SCIOPTA example project file hello.xml from your project folder into **SCONF**.
    File > Open
11. 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
    - sconf.h.
12. Launch the IAR Embedded Workbench.
13. Select the **Open existing workbench** button in the Embedded Workbench Startup window.
14. Browse to your example project directory and select the IAR Embedded Workbench file for your selected board: **<board>.eww**.
15. Select the project in the Workspace and Make the project (menu: **Project > Make**) or type the **F7** button.
16. The executable (sciopta.elf) will be created in the Output folder of the project.
17. Download and debug the sciopta.elf file into the target system (menu: **Project > Debug**) or type the **Ctrl+D** button.
18. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. The speed must be set to 115200 baud.
19. Run the system (menu: **Debug > Go**) or type the **Go** button.
20. Now you can check the log messages on your host terminal window.

You can also set breakpoints anywhere in the example system and watch the behaviour.

4 SCIOPTA Flash File System Project Overview

4.1 Introduction

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

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

4.2 Files

List of typical files needed from the SCIOPTA distribution for a small SCIOPTA FAT File System example.

File	Description	Compiler	Link
common.c	HCC common functions.	All	4.7 “SCIOPTA Flash File System Files” on page 4-4
fat.c	HCC FAT short filenames.	All	
fat_lfn.c	HCC long filenames.	All	
fat_m.c	HCC FAT wrapper.	All	
port.c	Ported functions	All	
sc2fatfs.c	HCC FAT FS integration	All	
sc2hcc.c	SCIOPTA - HCC integration	All	
fatfstest.c	Example file (SDD Mode)	All	4.8 “Writing your File System Application” on page 4-5
hcc_fatfstest.c	Example file (HCC Mode)	All	
fs_setup.c	Example FS setup process	All	
error.c	Error hook	All	4.9 “Error Handling” on page 4-5
system.c	System module functions	All	4.10 “System Configuration and Initialization” on page 4-6
map.c	Module mapping	IAR EW	
module_hook.c	Module hook functions.	All	
ramdrv_f.c	RAM driver.	All	4.11 “General System Functions and Drivers” on page 4-7
winIDEA_gnu.ind	winIDEA indirection file	GNU	
module.ld	GNU modules linker script	GNU	4.12 “ARM Family System Functions and Drivers” on page 4-7
cstartup.S	C Startup file	GNU	
cstartup.s	C startup for ARM RealView.	ARM RealView	
exception.S	Exception handler	GNU	
exception.s	Exception handler	ARM RealView	
exception.s79	Exception handler	IAR	

File	Description	Compiler	Link
systick.c	System tick interrupt process	All	4.13 “ARM CPUs System Functions and Drivers” on page 4-9
simple_uart.c	UART routines	All	
led.c	Board led function	All	
irq_handler.S	Interrupt wrapper	GNU	
irq_handler.s	Interrupt wrapper	ARM RealView	
irq_handler.s79	Interrupt wrapper	IAR EW	
resethook.S	Board setup	GNU	4.14 “Board System Functions and Drivers” on page 4-10
resethook.s	Board setup	ARM RealView	
resethook.s79	Board setup	IAR EW	
<board>.ld	Main linker script	GNU	
<board>.sct	Linker script	ARM RealView	
<board>.xcl	Linker script	IAR EW	
config.h	Board settings	All	
<board>.ini	winIDEA board initialization	GNU	
<board>.mac	IAR CSpy board initialization	IAR EW	
<board>.cmm	Trace32 board initialization	All	
hello.xml	SCIOPTA configuration file	All	4.15 “Kernel Configuration” on page 4-12
sciopta.S	Kernel source	GNU	4.16 “Kernel” on page 4-12
sciopta.s	Kernel source	ARM RealView	
sciopta.s79	Kernel source	IAR EW	
Makefile	Example makefile	GNU	4.18.1 “Makefiles” on page 4-13
board.mk	Board makefile	GNU	
.cproject	Eclipse project file	GNU	4.18.2 “Eclipse” on page 4-13
iC3000.xjrf	winIDEA project file	GNU	4.18.3 “iSYSTEM winIDEA” on page 4-14
iC3000.xqrf	winIDEA project file	GNU	
<board>.ewd	IAR project file	IAR EW	4.18.4 “IAR Embedded Workbench” on page 4-14
<board>.eww	IAR project file	IAR EW	

4.3 SCIOPTA Techniques and Concepts

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

Please read the chapter "SCIOPTA Technology and Methods" of the ARM - Kernel, User's Guide to get an introduction into the SCIOPTA technology.

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

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

4.4 SCIOPTA Flash File System Techniques and Concepts

Please consult chapter 5 "Flash File System Technology and Methods" on page 5-1 to get an introduction into the SCIOPTA Flash File System technology.

A SCIOPTA Flash File System application consists of specific user and system processes and file system specific messages and functions. Please consult chapter 5.4 "Flash File System Block Diagram" on page 5-6 of an overview block diagram of a typical SCIOPTA Flash File System application.

4.5 SCIOPTA Kernel System Calls

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

Please consult the chapter "Application Programming Interface" of the ARM - Kernel User's Guide for a detailed description of all SCIOPTA system calls.

4.6 SCIOPTA Flash File System API

4.6.1 HCC Mode

The SCIOPTA Flash File System uses and integrates the Safe Flash File System from HCC Embedded.

For systems without MMU (Memory Management Unit) controlled by the SCIOPTA SMMS Memory Management System it is highly recommended to use this HCC Mode and to work directly with the HCC Safe File System API.

Please consult chapter 6 “HCC Safe Flash File System API” on page 6-1 for a detailed description of the HCC API.

4.6.2 SDD Mode

In SDD Mode the programmer uses the SCIOPTA Device Driver (SDD) structures, messages and functions. SDD messages and functions are extended by specific file system messages and functions (SFS).

While the SDD and SFS messages can well be used we recommend to use the SCIOPTA SDD and SFS functions.

A detailed description of the message interface (API) can be found in chapter 7 “Message Interface Reference” on page 7-1.

A detailed description of the SCIOPTA Flash File System function interface (API) can be found in chapter 8 “SCIOPTA SFFS Function Interface Reference” on page 8-1.

4.7 SCIOPTA Flash File System Files

In order to use the SCIOPTA Flash File System you need to include some HCC Safe Flash File System source files and SCIOPTA integration files in your project.

Please consult chapter 9.4.1 “HCC Safe File System” on page 9-3 for more information about the HCC Safe File System files.

Files

fsm.c	HCC SAFE intermediate layer
fsf.c	HCC SAFE standard API
fsmf.c	SAFE Standard API multi-thread wrapper.
port_s.c	Ported functions

File location<installation_folder>\sciopta\<version>\sfs\hcc\effs\common\

SCIOPTA - HCC Integration Layer Files

Please consult chapter 9.4.2 “SCIOPTA - HCC Integration” on page 9-3 for more information.

sc2effs.c	HCC EFFS integration layer
sc2hcc.c	SCIOPTA - HCC integration layer

File location<installation_folder>\sciopta\<version>\sfs\hcc\src\

4.8 Writing your File System Application

Now you are ready to write your SCIOPTA Flash File System application. You need to design your application and system which is divided into modules, processes and the interprocess communication and coordination.

When you are analysing a real-time system you need to partition a large and complex real-time system into smaller components and you need to design system structures and interfaces carefully. This will not only help in the analysing and design phase, it will also help you maintaining and upgrading the system.

In chapter 9 “Application Programming” on page 9-1 you will find information how to design a SCIOPTA Flash File System application.

For the “effs_ram” SCIOPTA Flash File System getting started example the following files contain the application processes and messages:

Files

effstest.c	Example file system process (SDD Mode)
hcc_effstest.c	Example file system process (HCC Mode)
fs_setup.c	Example file system setup process

File location<installation_folder>\sciopta\<version>\exp\sfs\common\effs_ram\

4.9 Error Handling

SCIOPTA uses a centralized mechanism for error reporting called Error Hooks. Error Hooks must be registered and written by the user. Please consult chapter 9.11 “Error Hook” on page 9-11 for more information about error hooks.

Files

error.c	Error hook example.

File location: <installation_folder>\sciopta\<version>\exp\common\

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

Please consult chapter 10.2.4 “Start Hook” on page 10-3 for more information.

4.10 System Configuration and Initialization

System and application configuration consists basically to write the start hook (see chapter 10.2.4 “Start Hook” on page 10-3), the system module hook (see chapter 10.2.6.1 “System Module Hook” on page 10-4) and other system initialization functions. This is usually done in a specific file called system.c which can be found in the SCIOPTA examples deliveries.

For the “effs_ram” SCIOPTA Flash File System getting started example the file system.c contains the system and application configuration for the example system module:

Files

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

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

System and application configuration functions for other modules (user module hooks) is usually done in specific files having the same name as the module (**dev.c**, **ips.c** etc.) which can also be found in the SCIOPTA examples deliveries. Please consult also chapter 10.2.6.2 “User Modules Hooks” on page 10-4.

For IAR you need to define the free RAM of the modules in a separate file. In this area there are no initialized data. Module Control Block (ModuleCB), Process Control Blocks (PCBs), Stacks and Message Pools are placed in this free RAM. Please consult also chapter chapter 14.3 “IAR Embedded Workbench Linker Script” on page 14-2.

map.c	Module mapping for IAR

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

For the SCIOPTA Flash File System getting started example the file module_hook.c contains the user module hooks including system and application configuration for the example modules:

Files

module_hook.c	Configuration file including module hooks and other setup code.

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\

4.11 General System Functions and Drivers

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

For the SCIOPTA Flash File System we will use HCC Safe Flash File System drivers. Please consult chapter 11.21 "Flash File System Drivers" on page 11-35 for more information about the HCC Safe Flash File System drivers.

For the "effs_ram" SCIOPTA Flash File System getting started example we will use the RAM disk driver.

Files

ramdrv_s.c	RAM driver.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\ram\

4.12 ARM Family System Functions and Drivers

For a basic system some general ARM CPU family functions and drivers which are not board and CPU-derivative dependent are needed.

Please consult chapter 11.3 "ARM Family System Functions and Drivers" on page 11-1 for more information about ARM functions and drivers.

Project Files

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

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

Source Files

To setup and initialize the C-environment you need to include C startup functions. C startup functions are compiler specific.

Please consult for more information about C-startup.

For IAR Embedded Workbench there is no cstartup file needed as we are using the IAR C initialization library function.

cstartup.S	C startup assembler source for GNU GCC.
exception.S	Exception handler for GNU GCC.

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

cstartup.s	C startup assembler source for ARM RealView.
exception.s	Exception handler for ARM RealView

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

exception.s79	Exception handler for IAR EW.

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

4.13 ARM CPUs System Functions and Drivers

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

Please consult chapters

11.4 "AT91SAM7 System Functions and Drivers" on page 11-2
11.5 "AT91SAM9 System Functions and Drivers" on page 11-4
11.6 "LPC21xx System Functions and Drivers" on page 11-6
11.7 "LPC23xx and LPC24xx System Functions and Drivers" on page 11-8
11.8 "STR7 System Functions and Drivers" on page 11-10
11.9 "STR9 System Functions and Drivers" on page 11-12

Files

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

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

irq_handler.S	Interrupt wrapper for GNU GCC.

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

irq_handler.s	Interrupt wrapper for ARM RealView.

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

irq_handler.s79	Interrupt wrapper for IAR Embedded Workbench.

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

4.14 Board System Functions and Drivers

For your board you need to implement board functions and drivers. Usually at least a reset hook containing early board setup code is needed. Reset hooks are compiler manufacturer and board specific. Reset hook examples can be found in the SCIOPTA Board Support Package deliveries.

Please consult also chapters

11.10 “Atmel AT91SAM7A3-EK Board” on page 11-14
11.11 “Atmel AT91SAM7SE-EK Board” on page 11-16
11.12 “Atmel AT91SAM7S-EK Board” on page 11-18
11.13 “Atmel AT91SAM7X-EK Board” on page 11-20
11.14 “Atmel AT91SAM9260-EK Board” on page 11-22
11.15 “Atmel AT91SAM9261-EK Board” on page 11-24
11.16 “Atmel AT91SAM9263-EK Board” on page 11-26
11.17 “Phytec phyCORE-LPC2294 Board” on page 11-28
11.18 “Embedded Artists LPC2468 OEM Board” on page 11-30
11.19 “STR711-SK Board” on page 11-31
11.20 “STR912-SK Board” on page 11-33

Files

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

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

resethook.S	Board setup for GNU GCC.
mmu.S	MMU setup for GNU GCC.
led.S	Routines to access the board LEDs if available on the board for GNU GCC.

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

resethook.s	Board setup for ARM RealView.
mmu.s	MMU setup for ARM RealView.
led.s	Routines to access the board LEDs if available on the board for ARM RealView.

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

resethook.s79	Board setup for IAR EW.
mmu.s79	MMU setup for IAR EW.
led.s79	Routines to access the board LEDs if available on the board for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\src\iar\

Linker Scripts

Please consult the chapter "Linker Script and Memory Map" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of linker scripts.

A link is usually controlled by a linker script or linker control file. Linker scripts are compiler and board specific. Linker script examples can be found in the SCIOPTA Board Support Package deliveries. Please consult also chapter 14 "Linker Scripts and Memory Map" on page 14-1 for more information about linker scripts.

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

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

Board Configuration

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

config.h	Board configuration defines.

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

Debugger Board Setup

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

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

4.15 Kernel Configuration

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

Please consult chapter 12 “Kernel Configuration” on page 12-1 for more information about kernel configuration.

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

For the “effs_ram” SCIOPTA Flash File System getting started example the configuration file is delivered in the example projects delivery:

SCONF Configuration File

hello.xml	SCIOPTA kernel configuration file.

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

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

Generated Kernel Configuration Files

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

4.16 Kernel

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

Source File

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

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

4.17 Libraries and Include Files

Make sure that you have included the kernel libraries as described in chapter 13.2 “Kernel Libraries” on page 13-1 and the SCIOPTA Flash File System library as described in chapter 13.3 “SCIOPTA File System Library” on page 13-5.

Make sure that the include search directories are defined as described in chapter 13.4 “Include Files” on page 13-8. Please consult this chapter for more information about SCIOPTA includes.

4.18 Building the Project

4.18.1 Makefiles

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

Please consult a getting started example (e.g. chapter 3.2.3 “Makefile and GNU GCC” on page 3-3) for a description how to build a system with makefiles.

For the “effs_ram” SCIOPTA Flash File System getting started example the makefiles are delivered in the example projects delivery:

Files

Makefile	Example makefile.

File location: <installation_folder>\sciopta\<version>\exp\krn\arm\effs_ram\

board.mk	Board dependent makefiles.

File location: <installation_folder>\sciopta\<version>\exp\krn\arm\effs_ram\<board>\

4.18.2 Eclipse

SCIOPTA supplies Eclipse project files if you want to use Eclipse as an IDE for editing and building SCIOPTA applications.

Use the Eclipse project file as described in the getting started examples (e.g. chapter 3.2.4 “Eclipse IDE and GNU GCC” on page 3-5). Please consult also chapter 15.3 “Eclipse IDE and GNU GCC” on page 15-3.

For the “effs_ram” SCIOPTA Flash File System getting started example the Eclipse project file can be found in the example projects delivery:

Files

.cproject	Eclipse project file

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

4.18.3 iSYSTEM winIDEA

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

Use the iSYSTEM winIDEA as described in the getting started examples (e.g. chapter 3.2.5 “iSYSTEM winIDEA and GNU GCC” on page 3-7). Please consult also chapter 15.4 “iSYSTEM© winIDEA” on page 15-5.

For the “effs_ram” SCIOPTA Flash File System getting started example the iSYSTEM winIDEA project file can be found in the example projects delivery:

Files

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

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>

4.18.4 IAR Embedded Workbench

If the IAR Embedded Workbench (EW) is your preferred build and debug environment you will find IAR EW project files in the SCIOPTA examples deliveries.

Use the IAR Embedded Workbench as described in the getting started examples (e.g. chapter 3.2.6 “IAR Embedded Workbench” on page 3-9). Please consult also chapter 15.5 “IAR Embedded Workbench for ARM” on page 15-6.

For the “effs_ram” SCIOPTA Flash File System getting started example the IAR EW project file can be found in the example projects delivery:

Files

<board>.ewd	IAR Embedded Workbench project file.
<board>.eww	IAR Embedded Workbench project file.

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>

5 Flash File System Technology and Methods

5.1 SCIOPTA Flash File System Overview

SCIOPTA SFFS is an extremely robust embedded flash file system guaranteed to be 100% safe against power-failure. It is easy-to-use, scalable and can easy be ported to all SCIOPTA ARM target systems. SFFS has minimal requirements from the embedded system.

Some of the features include:

- API
  - Standard API
  - SCIOPTA SDD Function Interface
  - Multi-user interface
  - Long file names
  - Unicode 16 support
- NOR Flash Support
  - Wear-leveling
  - Bad block handling
  - Easy porting for all known device types
  - Sample driver with porting description
- Atmel DataFlash Support
  - Wear-leveling
  - All devices supported
  - Manages the 10K writes/sector limitation
  - Failsafe implementation of DataFlash interface
- NAND Flash Support
  - Wear-leveling
  - Bad-block management
  - ECC algorithm
  - Easy porting for all known device types
  - Sample driver with porting description

5.2 NAND and NOR Flash

The SCIOPTA Flash File System has been designed to allow the easy integration of all standard flash devices. But what are these devices?

Flash devices have certain basic properties in common:

- They are designed for the non-volatile storage of code and data.
- To write to an area it must first be erased, which changes all erased bits to 1. Programming consists of changing 1s to 0s. To change a 0 to a 1 an erase operation must be performed.
- Flash devices are all divided into erase units (blocks) such that to erase any part a whole block must be erased.
- Data areas all wear out after a number of erase cycles. The guarantee for the number of successful erase cycles varies among the chip types. Therefore, it is important for any file system that uses flash to manage the wearing of the flash. This is done by avoiding the overuse of any one block.

There are two basic types of Flash chips generally available today. They have quite distinct physical characteristics and thus require quite different handling.

5.2.1 NOR Flash

NOR flash has been the cornerstone of non-volatile memory in embedded systems for many years. NOR has two basic characteristics: it stores data in a non-volatile way and it can be accessed directly from an address bus ("random access") and thus can be used to run code.

NOR flash has some drawbacks. Firstly the erase/write time is very long; even if quite small mounts of data are written an erase may cause a delay of as much as 2 seconds. Careful design of the SCIOPTA Flash File System has ensured that this kind of behaviour is minimized, but in certain cases it isn't avoidable.

5.2.2 NAND/AND Flash

NAND flash (also AND) is a newer type of flash chip technology whose primary difference is:

- NAND can store approximately four times as much data as NOR technology for about the same price.
- NAND has much faster erase and write times, and thus is a superior choice for applications that require regular data storage.

There is a price to be paid for the improved performance:

- Data cannot be accessed via a standard address/data bus; instead commands must be sent to set the address and then the data can be read/written sequentially.
- Chips come from the factory with a number of bad blocks that can never be used.
- Bits may flip unexpectedly (don't panic! - see below).

Because of these complications NAND chips are designed with some additional features:

- Each block is divided into a number of read/write pages (typically 512 or 2048 bytes in size).
- Each page has an associated "spare" area that is used to store error correction and block management information. By using this area effectively the general performance and reliability of the devices is very high.

The NAND flash driver is contains the necessary spare area management and fast ECC algorithm.

5.2.3 NOR/NAND Summary

The following table summarizes the differences between NOR and NAND flash types. The given data are indicative and subject to change as the technologies evolve.

Property	NOR	NAND
Price	4x/MB	x/MB
Size	64KB - 64MB	16MB - 2 GB
Bootable	Yes	No
Random Access	Yes	No
Guaranteed Erase Cycles	10,000 - 100,000	1,000,000 with ECC
Block Erase Time (1)	2 s	2 ms
Write Time (2)	10 us/word	200 us/page
Read Time (2)	100 ns/word	50 us/page

1. Blocks on NAND flash devices are normally smaller than blocks on NOR flash devices. Since an erase must precede writing, the smaller block size is generally beneficial to a file system's performance.
2. The page size for NAND flash devices is typically 512 or 2048 bytes. Because file system access to the physical device is only in sectors the page access times are the most important when looking at the performance of the file system.

Please Note

New devices with new features are being produced all the time. The above table should be used as an indication. For any particular chip type the specific datasheet for that device must be consulted.

5.3 System Features

5.3.1 Power Fail Safety

The SCIOPTA Flash File System is entirely power fail safe. The system may be stopped at any point and restarted and no data will be lost; the previous completed state of the file system will be restored.

When a file is closed its data are automatically flushed from the file system. Until this close takes place the file is preserved. The user may also use the f_flush command to write the current state of the file to the medium and thus update its failsafe state.

5.3.2 Long File Names

The SCIOPTA Flash File System supports file names of almost unlimited length. File name handling is efficient - it is built from a chain of small fragments taken from the descriptor block. If a file name is longer than FS_MAXDENAME (default 13) an additional FS_MAXLFN (default 11) byte block is allocated to store the longer name. These additional blocks are added by the file system automatically.

5.3.3 Multiple Volumes

The SCIOPTA Flash File System supports multiple volumes. Each volume must have its own driver routine, which normally has its own physical handler (except for the RAM drive).

5.3.4 Multiple Open Files in a Volume

The SCIOPTA Flash File System allows multiple files to be opened simultaneously on a volume or on different volumes.

5.3.5 Case Sensitive File Names

By default the file system uses case insensitive names.

5.3.6 Separator Character

You can freely define the file separator character. By default this is a backslash.

5.3.7 Unicode

The SCIOPTA Flash File System supports Unicode.

5.3.8 Static Wear

Flash devices are usually manufactured to a specification which includes a guaranteed number of write-erase cycles that can be performed on each block before it may develop a fault. Because of this it is important to use the blocks in a device "evenly" if the device is to be used for its maximum lifetime.

The SCIOPTA Flash File System uses a process called dynamic wear to allocate the least use blocks from those available. However, in systems where there are large areas of static data (e.g., the executable binary for the system) then the areas where this is stored may be written only once leaving a relatively small section of the device to handle the much more heavily used files.

For this reason a process called static wear is introduced. When the fs_staticwear function is called it searches for blocks that have been used much less than the most used blocks in the system and if this difference is greater than a defined threshold (FS_STATIC_DISTANCE), then these two blocks will be exchanged in the system.

5.4 Flash File System Block Diagram

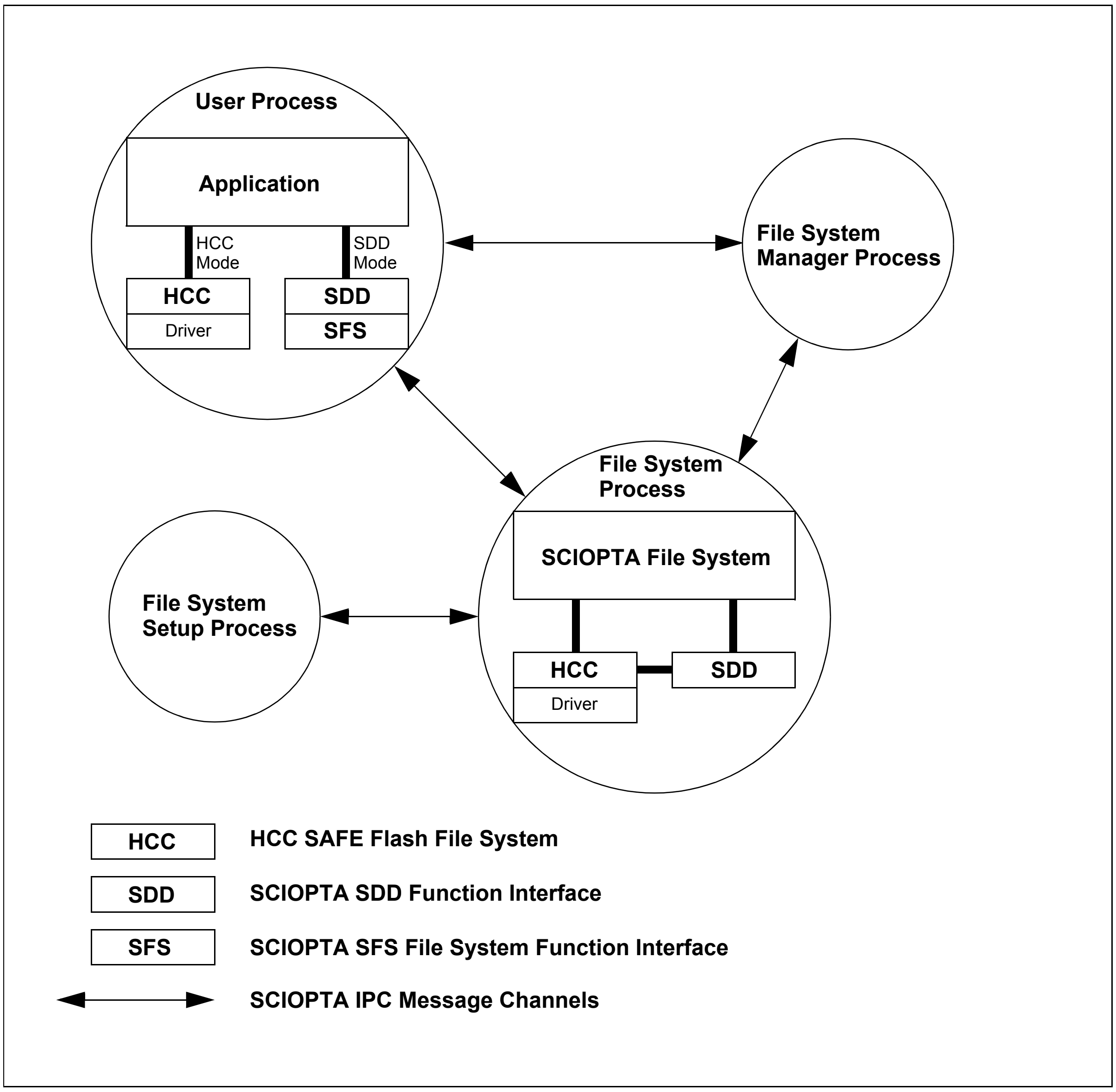

User Process
Application
HCC Mode
SDD Mode
HCC
Driver
SDD
SFS
File System Manager Process
File System Process
SCIOPTA File System
HCC
Driver
SDD
File System Setup Process
HCC
HCC SAFE Flash File System
SDD
SCIOPTA SDD Function Interface
SFS
SCIOPTA SFS File System Function Interface
SCIOPTA IPC Message Channels

Figure 5-1: SCIOPTA Flash File System Block Diagram

5.5 HCC Safe Flash File System

The SCIOPTA Flash File System uses and integrates the Safe Flash File System from HCC Embedded.

The following diagram illustrates the structure of the file system software.

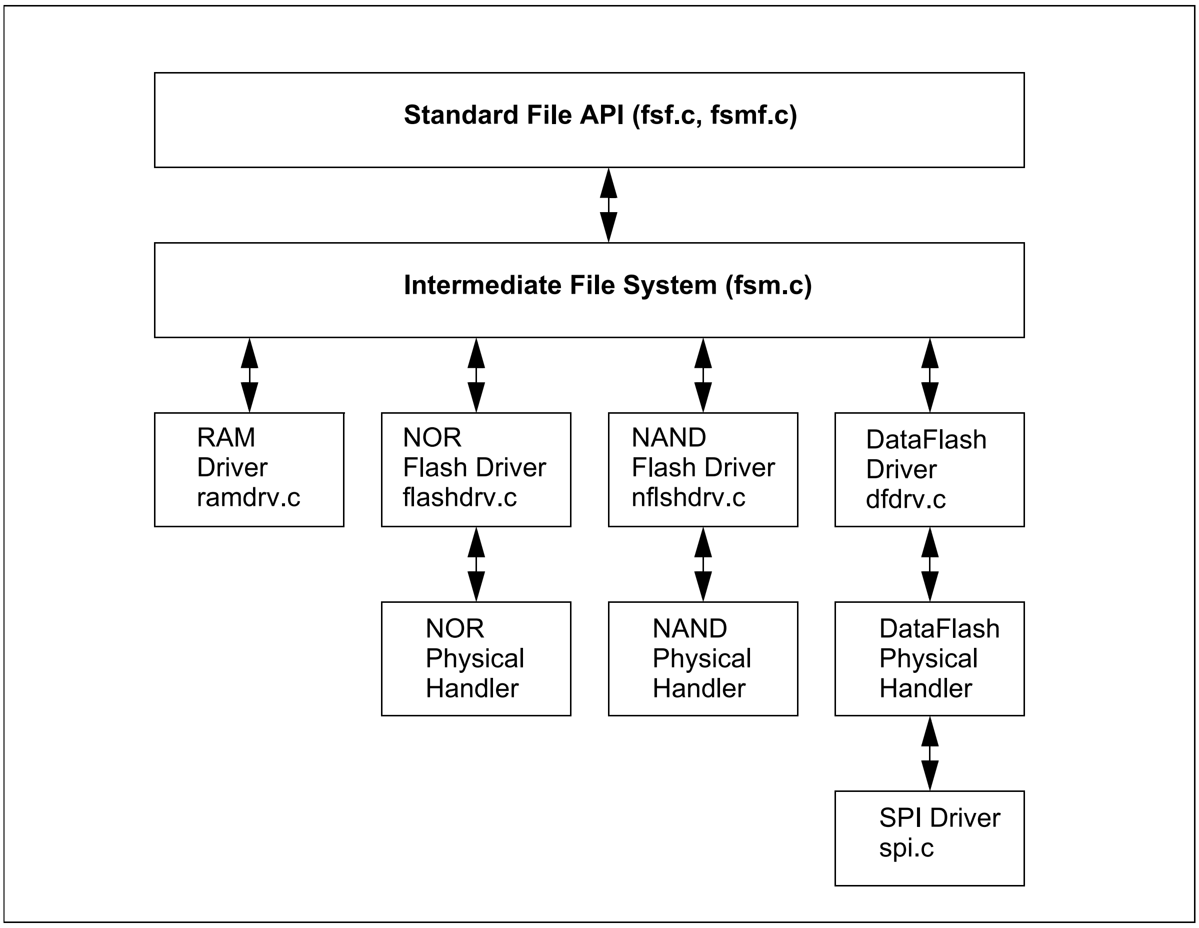

Standard File API (fsf.c, fsmf.c)
Intermediate File System (fsm.c)
RAM Driver ramdrv.c
NOR Flash Driver flashdrv.c
NAND Flash Driver nflshdrv.c
DataFlash Driver dfdrv.c
NOR Physical Handler
NAND Physical Handler
DataFlash Physical Handler
SPI Driver spi.c

Figure 5-2: HCC Safe Flash File System Structure

5.6 Processes

5.6.1 File System Process

The file system process includes the main sciopta file system initialization functions. It is using the HCC Safe File System (HCC Safe API), the HCC Safe File System drivers and the SDD (SCIOPTA Device Driver) interface library.

The file system process registers the file system at the manager process.

For each physical device you need a file system process.

5.6.2 File System Manager Process

SCIOPTA Flash File Systems are basically organized the same way as SCIOPTA devices. Therefore there is also a manager process which maintains the file system. The files system process registers the file system at the file system manager process. The file system manager process holds a list of registered file systems.

For each physical device you need a file system manager process.

5.6.3 File System Setup Process

The file system setup process is mainly used for mounting file systems and to wait for file systems to be initialized and up-and-running.

5.6.4 User Process

The user process contains the user code to access the file system. There are two ways of using the SCIOPTA file system.

5.6.4.1 HCC Mode

This is the most common way to use the file system. The HCC Safe File System functions are called directly by the user. The HCC Safe File System runs in the context of the user process.

5.6.4.2 SDD Mode

In SDD mode the user accesses the file system by using the SDD (SCIOPTA Device Driver) and the SFS (SCIOPTA Flash File System) functions. These functions send and receive messages to and from the file system process for accessing the HCC Safe File System.

This must be used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

5.7 SCIOPTA File System Descriptors

As the SCIOPTA Falsh File System is similar to the standard sciopta device driver concept, the file system uses the same kind of SDD objects.

5.7.1 SDD Objects

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

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

5.7.2 SDD Object Descriptors

SDD object descriptors are data structures in SCIOPTA containing information about **SDD objects**.

SDD object descriptors are stored in standard SCIOPTA messages. Therefore, SDD object descriptors contain a message ID structure element.

5.7.2.1 File System SDD Object Descriptors

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

5.8 Messages

SCIOPTA is a message based real-time operating system. Messages are the preferred method for interprocess communication (IPC). You can directly use predefined SDD and SFS messages to use the file system.

5.8.1 SDD Messages

SDD messages are predefined standardized messages to be used with SCIOPTA devices. As the SCIOPTA File System is the same way organized as a SCIOPTA device these messages will be used to access some standard file system methods.

Please consult the SCIOPTA ARM - SDD Device Driver, User's Guide for a detailed description of the SDD messages.

Some standard SDD messages:

Message	Description
SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY	Closes a device.
SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY	Sets or gets specific parameters to/from device drivers on the hardware layer.
SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY	Opens a device for read, write or read/write.
SDD_DEV_READ / SDD_DEV_READ_REPLY	Reads data from a device driver.
SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY	Writes data to a device driver.
SDD_ERROR	Used by device driver and other processes which do not use reply messages as answer of request messages for returning error codes.
SDD_MAN_ADD / SDD_MAN_ADD_REPLY	Adds a new device in the device driver system.
SDD_MAN_GET / SDD_MAN_GET_REPLY	Gets the SDD device descriptor (including the process IDs and handle) of a registered device.
SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY	Gets the device descriptor (including the process IDs and handle) of the first registered device from the SDD manager's device list.
SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY	Gets the device descriptor (including the process IDs and handle) of the next registered device from the SDD manager's device list.
SDD_MAN_RM / SDD_MAN_RM_REPLY	Removes a device from the device driver system.
SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY	Creates a copy of a device with identical data structures.
SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY	Releases an on-the-fly object.
SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY	Gets the size of an SDD object.
SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY	Gets the time from device drivers.
SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY	Sets the time of device drivers.

5.8.2 SFS SCIOPTA File System Messages

The SFS Messages extend the SDD Messages to support file system functionality.

These messages are described in this manual.

Some standard SDD messages:

Message	Description
SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY	Increases or decreases the size of an existing file.
SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY	Positioning write and read pointers in a file.
SFS_ASSIGN / SFS_ASSIGN_REPLY	Assigns the specified SDD file device which holds the file system.
SFS_MOUNT / SFS_MOUNT_REPLY	Mounts the object to the specified directory.
SFS_SYNC / SFS_SYNC_REPLY	Synchronizes the data residing in a cache with the data residing in the assigned SDD file device.
SFS_UNMOUNT / SFS_UNMOUNT_REPLY	Unmounts the specified directory and return the mounted object.

5.9 SDD Sciopta Device Driver Function Interface

In SDD Mode a user process accesses the file system by using the SDD device driver function interface. This must be used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

As the SCIOPTA File System is the same way organized as a SCIOPTA device these functions will be used to access some standard file system methods.

Please consult the SCIOPTA ARM - SDD Device Driver, User's Guide for a detailed description of the SDD functions.

Some standard SDD functions:

Function	Description
sdd_devAread	Reads data in an asynchronous mode from a device driver.
sdd_devClose	Closes an open device.
sdd_devIoctl	Gets and sets specific parameters in device drivers on the hardware layer.
sdd_devOpen	Opens device for read, write or read/write, or to create a device.
sdd_devRead	Reads data from a device driver.
sdd_devWrite	Writes data to a device driver.
sdd_manAdd	Adds a new device in the device driver system by registering it at a device manager process.
sdd_manGetByName	Gets the SDD device descriptor of a registered device from the manager's device list by giving the name as parameter.
sdd_manGetByPath	Gets the SDD device descriptor of a registered device from the manager's device list by giving the path as parameter.
sdd_manGetFirst	Gets the SDD device descriptor of the first registered device from the manager's device list.
sdd_manGetNext	Gets the SDD device descriptor of the next registered device from the manager's device list.
sdd_manNoOfItems	Gets the number of registered devices of the manager device list.
sdd_manGetRoot	Gets (creates) an SDD object descriptor of a root manager process.
sdd_manRm	Removes registered device, files and directories.
sdd_objDup	Creates a copy of the SDD device descriptor of a device by adopting the same state and context as the original device.
sdd_objFree	Releasse and frees an SDD object mainly an on-the-fly object.
sdd_objResolve	Returns the last struct manager in a given path for hierarchical organized managers.
sdd_objSizeGet	Gets the size of an SDD object.
sdd_objTimeGet	Gets the time of an SDD device.
sdd_objTimeSet	Sets the time of an SDD device.

5.10 SFS SCIOPTA File System Function Interface

The SFS Functions extend the SDD Functions to support file system functionality. This must be used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

These functions are described in this manual.

Some standard SFS functions:

Function	Description
sfs_assign	Assigns a SDD file device to an SDD file manager.
sfs_close	Closes a file object.
sfs_copy	Copies the file or the directory from the original location oldpath to the new location newpath within the SDD file manager.
sfs_create	Creates in the specified directory (dir) a new file or a new directory with the specified name, type and mode.
sfs_delete	Deletes a file or directory.
sfs_get	Gets a directory or file object.
sfs_mount	Mounts an SDD file manager or SDD device manager.
sfs_move	Moves the file or the directory from the original location oldpath to the new location newpath within the SDD file manager.
sfs_open	Gets and opens a directory or file object.
sfs_read	Reads data from a file object.
sfs_resize	Increases or decreases the size of an existing file object.
sfs_seek	Positions write and read pointers in file object.
sfs_sync	Synchronizes the data cache with the physical SDD file device.
sfs_unmount	Unmounts a mounted SDD file manager of a SDD directory.
sfs_write	Writes data to a file object.

6 HCC Safe Flash File System API

The SCIOPTA Flash File System uses and integrates the Safe Flash File System from HCC Embedded. The user can directly access the HCC Safe Flash File System API (HCC-Mode). This is the preferred standard method when using the flash file system without an MMU and memory protection.

The following diagram illustrates the structure of the file system software.

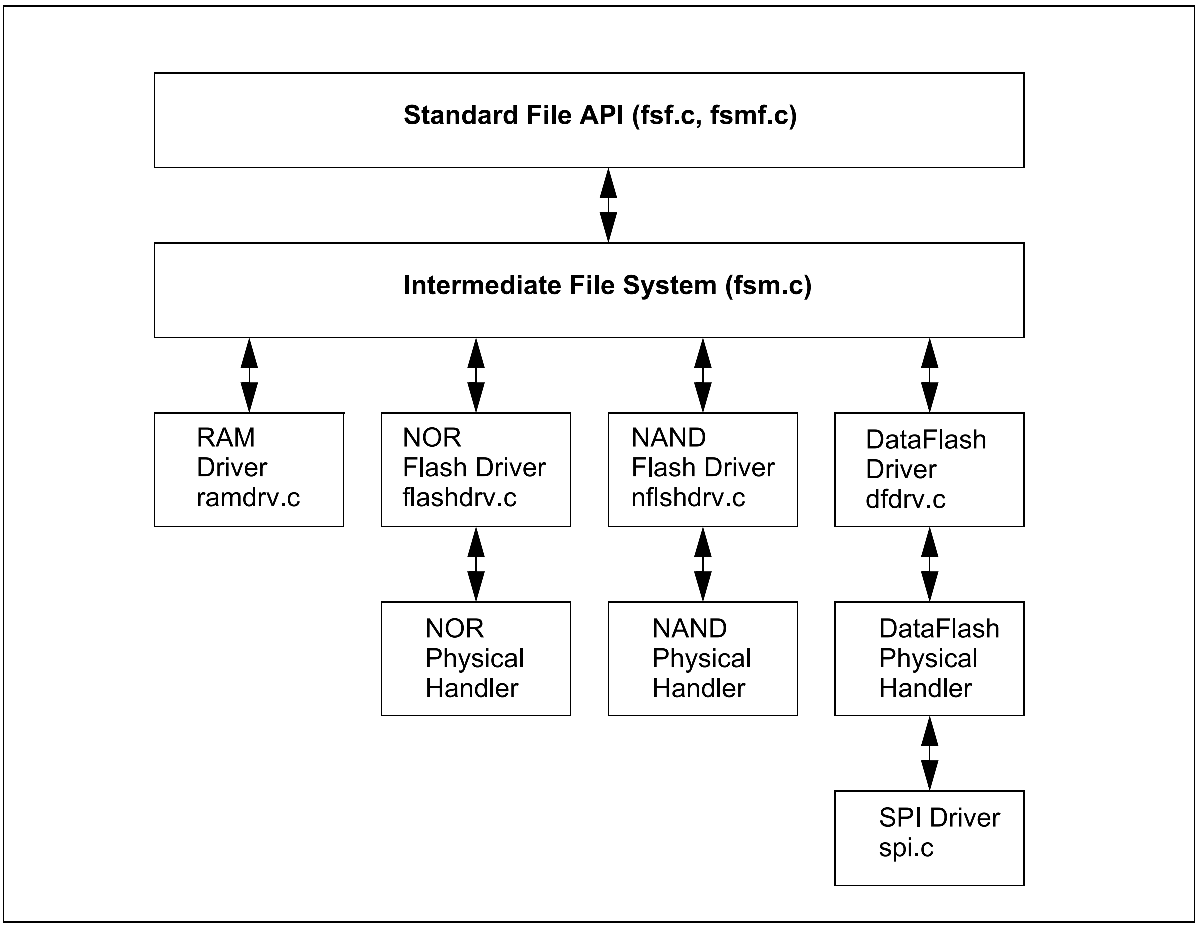

Standard File API (fsf.c, fsmf.c)
Intermediate File System (fsm.c)
RAM Driver ramdrv.c
NOR Flash Driver flashdrv.c
NAND Flash Driver nflshdrv.c
DataFlash Driver dfdrv.c
NOR Physical Handler
NAND Physical Handler
DataFlash Physical Handler
SPI Driver spi.c

Figure 6-1: HCC Safe Flash File System Structure

6.1 Overview

In chapter 6.2 “HCC File System Functions List” on page 6-2 the system calls are listed in alphabetical order while in the remaining chapters the system calls are listed in functional groups as follows:

Function Group	Page
Common Functions	6-5
Drive/Directory Handler Functions	6-17
File Functions	6-30
Read/Write Functions	6-55

6.2 HCC File System Functions List

Function	Short Description	Page
fs_staticwear	Even the wear of blocks that are rarely used.	6-14
f_chdir	Change current working directory.	6-25
f_chdrive	Change current drive.	6-18
f_checkvolume	Check the status of a drive that has been initialized.	6-29
f_close	Close a previously opened file.	6-59
f_delete	Delete a file.	6-34
f_enterFS	Create resources for the calling task in the file system and allocates a current working directory for that task.	6-7
f_eof	Check whether the current position in the opened target file is the end of the file.	6-64
f_filelength	Get the length of a file.	6-36
f_findfirst	Find first file or subdirectory in specified directory.	6-38
f_findnext	Find the next file or subdirectory in a specified directory after a previous call to f_findfirst or f_findnext.	6-40
f_flush	Flush data to the media.	6-60
f_format	Format a drive.	6-12
f_ftruncate	If a file is opened for writing, then this function truncates it to the specified length.	6-52
f_getc	Read a character from the current position in the open target file.	6-67
f_getcwd	Get current working folder on current drive.	6-19
f_getdcwd	Get current working folder on selected drive.	6-21
f_getdrive	Get current drive number.	6-17
f_getfreespace	Fill a user allocated structure with information about the usage of the volume specified.	6-13
f_getpermission	Retrieves file or directory permission field associated with a file.	6-48
f_gettimedate	Get time and date information from a file or directory.	6-44
f_getversion	Retrieve file system version information.	6-5

Function	Short Description	Page
f_get_oem	Return the OEM name "HCC_SAFE_FS".	6-16
f_init	Initialize the file system	6-6
f_mkdir	Make a new directory.	6-23
f_move	Moves a file or directory; the original is lost.	6-32
f_open	Opens a file.	6-55
f_putc	Write a character to the open target file at the current file position.	6-66
f_read	Read data from the current file position.	6-62
f_releaseFS	Release a previously assigned unique task ID.	6-8
f_rename	Rename a file or directory.	6-30
f_rewind	Set the current file position in the open target file to the beginning.	6-65
f_rmdir	Remove directory.	6-27
f_seek	Move read/write position in the file.	6-63
f_seteof	Move the end of file to the current file position.	6-68
f_setpermission	Set the file or directory permission field associated with a file.	6-46
f_settimedate	Set time and date on a file or on a directory.	6-42
f_stat	Get information about a file.	6-53
f_tell	Tell the current file position in the target file.	6-69
f_truncate	Opens a file for writing and truncates it to the specified length.	6-50
f_unmountdrive	Unmount an existing volume.	6-15
f_wchdir	Change current working directory with Unicode 16 name.	6-26
f_wdelete	Delete a file with unicode16 name.	6-35
f_wfilelength	Get the length of a file with Unicode 16 name.	6-37
f_wfindfirst	Find first file or subdirectory in specified directory with Unicode 16 name.	6-39
f_wfindnext	Find the next file or subdirectory in a specified directory with unicode 16 name after a previous call to f_wfindfirst or f_wfindnext.	6-41
f_wgetcwd	Get current working folder on current drive.	6-20
f_wgetdcwd	Get current working folder on selected drive.	6-22
f_wgetpermission	Retrieves file or directory permission field associated with a file with a unicode 16 name.	6-49
f_wgettimedate	Get time and date information from a file or directory with a Unicode 16 name.	6-45
f_wmkdir	Make a new directory with a Unicode 16 name.	6-24
f_wmove	Moves a file or directory with a Unicode 16 name.	6-33
f_wopen	Opens a file with Unicode 16 file name.	6-57
f_wrename	Rename a file or directory with unicode16 name.	6-31
f_wrmdir	Remove Unicode16 directory.	6-28
f_write	Write data into file at current position.	6-61

Function	Short Description	Page
f_wsetpermission	This sets the file or directory permission field associated with a file with unicode 16 name.	6-47
f_wsettimedate	Set time and date on a file or on a directory with unicode16 name.	6-43
f_wstat	Get information about a file with a unicode 16 name.	6-54
f_wtruncate	Opens a file with a unicode 16 name for writing and truncates it to the specified length.	6-51

6.3 Common Functions

6.3.1 f_getversion

This function is used to retrieve file system version information.

```
char * f_getversion(void)
```

Parameter	Description
None	

Return Value	Condition
Pointer to null terminated ASCII string	

Example

```
void display_fs_version(void)
{
   printf("File System Version: %s",f_getversion());
}
```

6.3.2 f_init

This function initializes the file system. It must be called once to initialize file system before using any other file system function.

This function sets all critical initialized data to zero. This feature is required only if your compiler does not set un-initialized static data to zero – a feature of TI's Code Composer.

```
int f_init(void)
```

Parameter	Description
None	

Return Value	Condition
F_NO_ERROR	Drive successfully initialized.
Error Code	Drive not successfully initialized.

Example

```
void main(void)
{
   f_init(); /* init file system */
      .
      .
}
```

6.3.3 f_enterFS

If the target system allows multiple tasks to use the file system, then this function must be called by a task before using any other file API functions. This function creates resources for the calling task in the file system and allocates a current working directory for that task.

The f_releaseFS call must be made to release the task from the file system and free the allocated resource.

The correct operation of this function also requires that fn_gettaskID() in port_s.c has been ported to give a unique identifier for each task.

```
int f_enterFS(void)
```

Parameter	Description
None	

Return Value	Condition
0	Successful call.
Error Code	Error condition.

6.3.4 f_releaseFS

This function is called to release a previously assigned unique task ID that is used to track the calling task’s current working directory.

The unique task identifier is generated by fn_gettaskID() in port_s.c.

```
void f_releaseFS(long ID)
```

Parameter	Description
ID	Unique identifier for calling task

Return Value	Condition
None	

6.3.5 f_mountdrive

This is used to mount and map a new drive.

```
int f_mountdrive(
                   int          drivenum,
                   void         *buffer,
                   long         buffsize,
                   FS_DRVMOUNT  mountfunc,
                   FS_PHYGETID  phyfunc
)
```

Parameter	Description	
drivenum	Number of the drive to be mounted where 0 is drive 'A', 1 is drive 'B' etc. This has the maximum value of (FS_MAXVOLUME-1) in fsm.h.	
buffer	This is a pointer for a buffer area to be used by the generic driver. Its size depends on the specific devices and configuration used. For a **RAM** drive a buffer of the size required for the whole **RAM** file system should be allocated as shown in the example below. For a **NOR** drive the generic NOR flash function **fs_getmem_norflashdrive** must be called with a pointer to the get physical function of the specific physical chip driver to be mounted (e.g. fs_phy_nor_29lvxxx). This function then calculates and returns the amount of memory that must be allocated for this physical driver. The caller must then allocate this amount of memory and pass its pointer and size to the f_mountdrive function. See example code below. For a **NAND** drive the generic **NAND** flash function **fs_getmem_nandflashdrive** must be called with a pointer to the get physical function of the specific physical chip driver to be mounted (e.g. fs_phy_nand_K9F2816X0C). This function then calculates and returns the amount of memory that must be allocated for this physical driver. The caller must then allocate this amount of memory and pass its pointer and size to the f_mountdrive function. See example code below.	
buffsize	This is the size of the allocated buffer being passed to the mount function.	
mountfunc	This is a pointer to the generic mount function for the media type required. The mountfunc is a driver function that describes which drive needs to be mounted. This calls the physical driver function to be associated with it. Standard examples are:	
	Value	**Description**
	fs_mount_ramdrive	For using drive as RAM drive.
	fs_mount_flashdrive	For using drive as NOR flash drive.
	fs_mount_nandflashdrive	For using drive as NAND flash drive.

<table>
<tr><th>Parameter</th><th colspan="2">Description</th></tr>
<tr><td rowspan="6"><b>phyfunc</b></td><td colspan="2">This is a pointer to a physical driver function for the desired device that is called by the generic mount function to get information about how to use the device. For a RAM drive this function is NULL.

Standard examples are:</td></tr>
<tr><th>Value</th><th>Description</th></tr>
<tr><td>fs_phy_nor_sim</td><td>For PC emulation of NOR physical</td></tr>
<tr><td>fs_phy_nor_29lvxxx</td><td>For AMD flash</td></tr>
<tr><td>fs_phy_nand_sim</td><td>For PC emulation of NAND physical</td></tr>
<tr><td>fs_phy_nand_ K9F2816X0C</td><td>For Samsung NAND flash</td></tr>
</table>

Return Value	Condition
FS_VOL_OK	Successfully mounted.
FS_VOL_NOTMOUNT	Not mounted.
FS_VOL_NOTFORMATTED	Drive is mounted but drive is not formatted
FS_VOL_NOMEMORY	Not enough memory, drive is not mounted
FS_VOL_NOMORE	No more drive available (FS_MAXVOLUME)
FS_VOL_DRVERROR	Mount driver error, not mounted

Example

```
/* this example shows how to mount Ramdrive, */
/* FLASH drive and NANDFLASHdrive */

char p0buffer[0x100000]; /* 1M */

void main(void)
{
   char *p1buffer, *p2buffer;
   long memsize;

   f_init();

   f_mountdrive(0,p0buffer,sizeof(p0buffer),fs_mount_ramdrive, 0);
   /* Drive A will be RAM drive */

   memsize=fs_getmem_flashdrive(fs_phy_nor_29lvxxx);
   if (!memsize)
   {
      /* flash is not identified */
   }

   p1buffer=(char*)malloc(memsize);
   if (!p1buffer)
   {
      /* Not enough memory to allocate */
   }

   f_mountdrive(1,p1buffer,memsize,fs_mount_flashdrive,
      fs_phy_nor_29lvxxx);
      /* Drive B will be NOR flash drive, with */
      /* AMD physical driver */

   memsize=fs_getmem_nandflashdrive(fs_phy_nand_K9F2816X0C);
   if (!memsize)
   {
      /* nand flash is not identified, */
   }
   p2buffer=(char*)malloc(memsize);
   if (!p2buffer)
   {
      /* Not enough memory to allocate */
   }

f_mountdrive(2,p2buffer,memsize,fs_mount_nandflashdrive,
   fs_phy_nand_K9F2816X0C);
      /* Drive C will be NAND flash drive with */
      /* Samsung physical */
}
```

6.3.6 f_format

Format a drive. All data will be destroyed on the drive with the exception of the wear-level information on a FLASH device.

```
int f_format(int drivenum)
```

Parameter	Description
drivenum	Which drive needs to be formatted.

Return Value	Condition
F_NO_ERROR	Drive successfully formatted.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_BUSY	There is any file open.
F_ERR_NOTFORMATTED	Drive cannot be formatted

Example

```
char buffer[0x30000];

void myinitfs(void)
{
   int ret;

   f_init();
   ret=f_mountdrive(0,buffer,sizeof(buffer),fs_mount_flashdrive,
    fs_phy_nor_29lvxxx);
      /* Drive A will be NOR flash drive */

   if (ret==FS_VOL_OK) return; /* initialized */
   if (ret==FS_VOL_NOTFORMATTED)
   {
      ret=f_format(0); /* format drive A */
      if (ret==F_ERR_NOTERR) return; /* formatted */
   }
initializationfailed:
}
```

6.3.7 f_getfreespace

This function fills a user allocated structure with information about the usage of the volume specified. The information returned is the total size of the drive, the free space on the drive, the used space on the drive and the bad space on the drive.

```
int f_getfreespace(
                   int        drvnum,
                   F_SPACE    *pSpace
)
```

Parameter	Description
drvenum	Number of drive.
pSpace	Pointer to user’s free space structure

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void info(void)
{
   int ret;
   F_SPACE space;

   ret = f_getfreespace(f_getdrive(),&space);

   if(!ret)
   {
      printf("There are %d total bytes,\
         %d free bytes,\
         %d used bytes,\
         %d bad bytes.\n",
         space.total,space.free,\
         space.used,space.bad);
   }
   else
      printf(“Error %d\n”,ret);
}
```

6.3.8 fs_staticwear

This function is called to even the wear of blocks that are rarely used.

Read the “Static Wear” part of Section 1 of this manual for information about when and how to use this function.

```
int fs_staticwear(int drvnum)
```

Parameter	Description
drvenum	Number of target drive.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void idle(void)
{
   int ret;

   /* try static wear on Drive A */

   ret = fs_staticwear(0);

   if(!ret)
   {
      printf("Static Wear Done\n”);
   }
   Else
   {
      printf(“Error in static wear!!\n”,ret);
   }
}
```

6.3.9 f_unmountdrive

This function is used to unmount an existing volume. Any open files on the media will be marked as closed so that subsequent API accesses to a previously opened file handle will return with an error.

This function works independently of the status of the hardware.

```
int f_unmountdrive(int drivenum)
```

Parameter	Description
drvenum	Drive to be unmounted (0:A, 1:B...)

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void mydelfs(int num)
{
   int ret;

   /*Delete volume 1 */
   if(f_unmountdrive (num))
      printf("Unable to delete volume %d", num);
         .
         .
         .
}
```

6.3.10 f_get_oem

This returns the OEM name "HCC_SAFE_FS". The pointer passed for storage should be capable of holding a 12-character string.

```
int f_get_oem(
                int       drivenum,
                char      *str,
                long      len
)
```

Parameter	Description
drivenum	Drive number.
str	Pointer to copy label.
len	Length of storage area

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void get_disk_oem(void)
{
   char oem_name[12];
   int result;

   oem_name[11]=0;/* zero terminate string */
   result = f_get_oem(f_getcurrdrive(),oem_name,12);

   if (result)
      printf("Error on Drive");
   else
      printf("Drive OEM is %s",oem_name);
}
```

6.4 Drive/Directory Handler Functions

6.4.1 f_getdrive

Gets current drive number.

```
int f_getdrive(void)
```

Parameter	Description
None	

Return Value	Condition
Current Drive. 0-A, 1-B, 2-C etc.	

Example

```
void myfunc(void)
{
   int currentdrive;
      .
   currentdrive=f_getdrive();
      .
      .
}
```

6.4.2 f_chdrive

Change current drive.

```
int f_chdrive(int drivenum)
```

Parameter	Description
drivenum	Drive number to be current drive (0-A,1-B,2-C,…)

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Drive number is invalid

Example

```
void myfunc(void)
{
      .
      .
   f_chdrive(0); /* select drive A */
      .
      .
}
```

6.4.3 f_getcwd

Get current working folder on current drive.

```
int f_getcwd(
              char    *buffer,
              int     maxlen
)
```

Parameter	Description
buffer	Where to store current working directory string.
maxlen	Length of the buffer.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Current drive is invalid.

Example

```
void myfunc(void)
{
   char buffer[F_MAXPATH];

   if (!f_getcwd(buffer, F_MAXPATH))
   {
      printf ("current directory is %s",buffer);
   }
   else
   {
      printf ("Drive Error")
   }
}
```

6.4.4 f_wgetcwd

Get current working folder on current drive.

```
int f_wgetcwd(
             W_CHAR  *buffer,
             int     maxlen
)
```

Parameter	Description
buffer	Where to store current working directory string.
maxlen	Length of the buffer.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Current drive is invalid.

Example

```
void myfunc(void)
{
   W_CHAR buffer[F_MAXPATH];

   if (!f_wgetcwd(buffer, F_MAXPATH))
   {
      wprintf (“current directory is %s”,buffer);
   }
   else
   {
      wprintf (“Drive Error”)
   }
}
```

6.4.5 f_getdcwd

Get current working folder on selected drive.

```
int f_getdcwd(
             int     drivenum,
             char    *buffer,
             int     maxlen
)
```

Parameter	Description
drivenum	Specify drive (0-A, 1-B, 2-C)
buffer	Where to store selected working directory string.
maxlen	Length of the buffer.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Selected drive is invalid.

Example

```
void myfunc(int drivenum)
{
  char buffer[F_MAXPATH];

  if (!f_getcwd(drivenum,buffer, F_MAXPATH))
  {
    printf ("current directory is %s",buffer);
    printf ("on drive %c",drivenum+'A');
  }
  else
  {
    printf ("Drive Error")
  }
}
```

6.4.6 f_wgetdcwd

Get current working folder on selected drive.

```
int f_getdcwd(
             int     drivenum,
             W_CHAR  *buffer,
             int     maxlen
)
```

Parameter	Description
drivenum	Specify drive (0-A, 1-B, 2-C)
buffer	Where to store selected working directory string.
maxlen	Length of the buffer.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Selected drive is invalid.

Example

```
void myfunc(int drivenum)
{
   W_CHAR buffer[F_MAXPATH];

   if (!f_wgetcwd(drivenum,buffer, F_MAXPATH))
   {
      wprintf (“current directory is %s”,buffer);
      wprintf (“on drive %c”,drivenum+’A’);
   }
   else
   {
      wprintf (“Drive Error”)
   }
}
```

6.4.7 f_mkdir

Make a new directory.

```
int f_mkdir(
             const char *dirname
)
```

Parameter	Description
dirname	New directory name to create.

Return Value	Condition
F_NO_ERROR	New directory name created successfully.
F_ERR_INVALIDNAME	Directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_INVALIDDIR	Invalid path.
F_ERR_DUPLICATED	Entry already exists.
F_ERR_NOMOREENTRY	Directory is full.

Example

```
void myfunc(void)
{
     .
     .
  f_mkdir(“subfolder”);/* creating directory */
  f_mkdir(“subfolder/sub1”);
  f_mkdir(“subfolder/sub2”);
  f_mkdir(“a:/subfolder/sub3”
     .
     .
}
```

6.4.8 f_wmkdir

Make a new directory with a Unicode 16 name.

```
int f_wmkdir(
             const WCHAR  *dirname
)
```

Parameter	Description
dirname	New Unicode16 directory name to create.

Return Value	Condition
F_NO_ERROR	New directory name created successfully.
F_ERR_INVALIDNAME	Directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_INVALIDDIR	Invalid path.
F_ERR_DUPLICATED	Entry already exists.
F_ERR_NOMOREENTRY	Directory is full.

Example

```
void myfunc(void)
{
      .
      .
   f_wmkdir(“subfolder”);/* creating directory */
   f_wmkdir(“subfolder/sub1”);
   f_wmkdir(“subfolder/sub2”);
   f_wmkdir(“a:/subfolder/sub3”
      .
      .
}
```

6.4.9 f_chdir

Change current working directory.

```
int f_chdir(
             const char    *dirname
)
```

Parameter	Description
dirname	New directory name to change.

Return Value	Condition
F_NO_ERROR	Directory has been changed successfully.
F_ERR_INVALIDNAME	Directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	Path not found.

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir("subfolder");
   f_chdir("subfolder");/* change directory */
   f_mkdir("sub2");
   f_chdir("..");/* go to upward */
   f_chdir("subfolder/sub2");
   /* goto into sub2 dir */
      .
      .
}
```

6.4.10 f_wchdir

Change current working directory with Unicode 16 name.

```
int f_wchdir(
             const WCHAR  *dirname
)
```

Parameter	Description
dirname	New Unicode 16 directory name to change.

Return Value	Condition
F_NO_ERROR	Directory has been changed successfully.
F_ERR_INVALIDNAME	Directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	Path not found.

Example

```
void myfunc(void)
{
     .
     .
  f_wmkdir(“subfolder”);
  f_wchdir(“subfolder”);/* change directory */
  f_wmkdir(“sub2”);
  f_wchdir(“..”);/* go to upward */
  f_wchdir(“subfolder/sub2”);
  /* goto into sub2 dir */
     .
     .
}
```

6.4.11 f_rmdir

Remove directory. Directory has to be empty when it is removed, otherwise returns with error code without removing.

```
int f_rmdir(
             const char   *dirname
)
```

Parameter	Description
dirname	Directory name to remove.

Return Value	Condition
F_NO_ERROR	Directory is removed successfully.
F_ERR_INVALIDNAME	Directory name contains invalid characters.
F_ERR_NOTFOUND	Directory not found.
F_ERR_INVALIDDIR	Directory name is not a directory.
F_ERR_NOTEMPTY	Directory not empty.

Example

```
void myfunc(void)
{
      .
      .
   f_mkdir("subfolder"); /* creating directories */
   f_mkdir("subfolder/sub1");
      .
      . doing some work
      .
   f_rmdir("subfolder/sub1");
   f_rmdir("subfolder"); /* removes directory */
      .
      .
}
```

6.4.12 f_wrmdir

Remove Unicode16 directory. Directory has to be empty when it is removed, otherwise returns with error code without removing.

```
int f_rmdir(
             const W_CHAR *dirname
)
```

Parameter	Description
dirname	Unicode16 directory name to remove.

Return Value	Condition
F_NO_ERROR	Directory is removed successfully.
F_ERR_INVALIDNAME	Directory name contains invalid characters.
F_ERR_NOTFOUND	Directory not found.
F_ERR_INVALIDDIR	Directory name is not a directory.
F_ERR_NOTEMPTY	Directory not empty.

Example

```
void myfunc(void)
{
     .
     .
  f_wmkdir(“subfolder”); /* creating directories */
  f_wmkdir(“subfolder/sub1”);
     .
     . doing some work
     .
  f_wrmdir(“subfolder/sub1”);
  f_wrmdir(“subfolder”); /* removes directory */
     .
     .
}
```

6.4.13 f_checkvolume

This function is used to check the status of a drive that has been initialized.

```
int f_checkvolume(
             int     drivenum
)
```

Parameter	Description
drivenum	Drive to be checked (0:A, 1:B...).

Return Value	Condition
F_NO_ERROR	Drive is working.
else	There is an error on the drive; e.g., card is missing

Example

```
void mychkfs(int num)
{
   int ret;

   /*Delete volume 1 */
   if(f_checkvolume(num))
   {
      printf(“Volume %d is not usable, Error %d”, num, ret);
   }
   else
   {
      printf((“Volume %d is working”, num);
   }
      .
      .
}
```

6.5 File Functions

6.5.1 f_rename

Rename a file or directory. This function has been obsoleted by **f_move**.

```
int f_rename(
             const char    *filename,
             const char    *newname
)
```

Parameter	Description
filename	File or directory name with/without path.
newname	New name of file or directory.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.
F_ERR_BUSY	File is open for read or write.
F_ERR_DUPLICATED	Name already exists.

Example

```
void myfunc(void)
{
      .
      .
   f_rename (“oldfile.txt”,”newfile.txt”);
   f_rename (“A:/subdir/oldfile.txt”,”newfile.txt”);
      .
      .
}
```

6.5.2 f_wrename

Rename a file or directory with unicode16 name. This function has been obsoleted by **f_wmove**.

```
int f_wrename(
             const W_CHAR *filename,
             const W_CHAR *newname
)
```

Parameter	Description
filename	Unicode 16 file or directory name with/without path.
newname	New Unicode 16 name of file or directory.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.
F_ERR_BUSY	File is open for read or write.
F_ERR_DUPLICATED	Name already exists.

Example

```
void myfunc(void)
{
     .
     .
  f_wrename (“oldfile.txt”,”newfile.txt”);
  f_wrename (“A:/subdir/oldfile.txt”,”newfile.txt”);
     .
     .
}
```

6.5.3 f_move

Moves a file or directory; the original is lost. This function obsoletes **f_rename()**. The source and target must be in the same volume.

```
int f_move(
             const char    *filename,
             const char    *newname
)
```

Parameter	Description
filename	File or directory name with/without path.
newname	New name of file or directory with/without path.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.
F_ERR_BUSY	File is open for read or write.
F_ERR_DUPLICATED	Name already exists.

Example

```
void myfunc(void)
{
     .
     .
   f_move ("oldfile.txt","newfile.txt");
   f_move ("A:/subdir/oldfile.txt","A:/newdir/oldfile.txt");
     .
     .
}
```

6.5.4 f_wmove

Moves a file or directory with a Unicode 16 name. The original is lost. This function obsoletes **f_wrename()**. The source and target must be in the same volume.

```
int f_wmove(
             const W_CHAR *filename,
             const W_CHAR *newname
)
```

Parameter	Description
filename	Unicode16 file or directory name with/without path.
newname	New unicode16 name of file or directory with/without path.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.
F_ERR_BUSY	File is open for read or write.
F_ERR_DUPLICATED	Name already exists.

Example

```
void myfunc(void)
{
      .
      .
   f_wmove ("oldfile.txt","newfile.txt");
   f_wmove ("A:/subdir/oldfile.txt","A:/newdir/oldfile.txt");
      .
      .
}
```

6.5.5 f_delete

Delete a file.

```
int f_delete(
             const char    *filename
)
```

Parameter	Description
filename	File name with/without path to be deleted

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File not found.
F_ERR_BUSY	File is open for read or write.
F_ERR_INVALIDDIR	File name is a directory name.

Example

```
void myfunc(void)
{
     .
     .
  f_delete ("oldfile.txt");
  f_delete ("A:/subdir/oldfile.txt");
     .
     .
}
```

6.5.6 f_wdelete

Delete a file with unicode16 name.

```
int f_wdelete(
             const W_CHAR *filename
)
```

Parameter	Description
filename	Unicode 16 file name with/without path to be deleted

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File not found.
F_ERR_BUSY	File is open for read or write.
F_ERR_INVALIDDIR	File name is a directory name.

Example

```
void myfunc(void)
{
     .
     .
  f_wdelete ("oldfile.txt");
  f_wdelete ("A:/subdir/oldfile.txt");
     .
     .
}
```

6.5.7 f_filelength

Get the length of a file.

```
long f_filelength(
             char       *filename
)
```

Parameter	Description
filename	File name with/without path

Return Value	Condition
Length of file	

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,”r”);
   long size=f_filelength(filename);

   if (!file)
   {
      printf (“%s Cannot be opened!”,filename);
      return 1;
   }

   if (size>buffsize)
   {
      printf (“Not enough memory!”);
      return 2;
   }

   f_read(buffer,size,1,file);
   f_close(file);

   return 0;
}
```

6.5.8 f_wfilelength

Get the length of a file with Unicode 16 name.

```
long f_wfilelength(
             W_CHAR     *filename
)
```

Parameter	Description
filename	Unicode 16 file name with/without path

Return Value	Condition
Length of file	

Example

```
int myreadfunc(W_CHAR *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_wopen(filename,”r”);
   long size=f_wfilelength(filename);

   if (!file)
   {
      printf (“%s Cannot be opened!”,filename);
      return 1;
   }

   if (size>buffsize)
   {
      printf (“Not enough memory!”);
      return 2;
   }

   f_read(buffer,size,1,file);
   f_close(file);

   return 0;
}
```

6.5.9 f_findfirst

Find first file or subdirectory in specified directory. First call **f_findfirst** function and if file was found get the next file with **f_findnext** function.

```
int f_findfirst(
             const char    *filename,
             F_FIND        *find
)
```

Parameter	Description
filename	Name of file to find.
find	Where to store find information

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_INVALIDDIR	Invalid path.
F_ERR_NOTFOUND	File not found.

Example

```
void mydir(void)
{
   F_FIND find;

   if (!f_findfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf (“filename:%s”,find.filename);
       if (find.attr&F_ATTR_DIR)
         {
            printf (“ directory\n”);
          }
          else
         {
            printf (“ size %d\n”,find.len);
      }
      } while (!f_findnext(&find));
   }
}
```

6.5.10 f_wfindfirst

Find first file or subdirectory in specified directory with Unicode 16 name. First call **f_wfindfirst** function and if file was found get the next file with **f_wfindnext** function.

```
int f_wfindfirst(
             const W_CHAR *filename,
             F_WFIND      *find
)
```

Parameter	Description
filename	Unicode 15 name of file to find.
find	Where to store find information.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_INVALIDDIR	Invalid path.
F_ERR_NOTFOUND	File not found.

Example

```
void mydir(void)
{
   F_WFIND find;
   if (!f_wfindfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
          }
          else
         {
            printf (" size %d\n",find.len);
      }
      } while (!f_wfindnext(&find));
   }
}
```

6.5.11 f_findnext

Find the next file or subdirectory in a specified directory after a previous call to **f_findfirst** or **f_findnext**. First call **f_findfirst**; if file was found get the rest of the matching files by repeated calls to **f_findnext**.

```
int f_findnext(
             F_FIND       *find
)
```

Parameter	Description
find	Find structure (from f_findfirst).

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File not found.

Example

```
void mydir(void)
{
   F_FIND find;
   if (!f_findfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
         if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
          }
          else
         {
            printf (" size %d\n",find.len);
      }
      } while (!f_findnext(&find));
   }
}
```

6.5.12 f_wfindnext

Find the next file or subdirectory in a specified directory with unicode 16 name after a previous call to **f_wfindfirst** or **f_wfindnext**. First call **f_wfindfirst**; if file was found get the rest of the matching files by repeated calls to **f_wfindnext**.

```
int f_wfindnext(
             F_WFIND      *find
)
```

Parameter	Description
find	Find structure (from f_findfirst).

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File not found.

Example

```
void mydir(void)
{
   F_WFIND find;
   if (!f_wfindfirst("A:/subdir/*.*",&find))
   {
      do
      {
         printf ("filename:%s",find.filename);
          if (find.attr&F_ATTR_DIR)
         {
            printf (" directory\n");
          }
          else
         {
            printf (" size %d\n",find.len);
         }
      } while (!f_wfindnext(&find));
   }
}
```

6.5.13 f_settimedate

Set time and date on a file or on a directory.

A recommended format for the use of the time date fields is given in.

Note:

The time/date data is simply two 16-bit numbers associated with the specified file that the developer is free to use as desired.

```
int f_settimedate(
             const char      *filename,
             unsigned short  ctime,
             unsigned short  cdate
)
```

Parameter	Description
filename	Name of file.
ctime	Creation time of file or directory.
cdate	Creation date of file or directory.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   unsigned short ctime,cdate;

   ctime = (15<<11)+(30<<5)+(23>>1);
   /* 15:30:22 */
   cdate = ((2002-1980)<<9)+(11<<5)+(3);
   /* 2002.11.03. */

   f_mkdir("subfolder");/* creating directory */
   f_settimedate("subfolder",ctime,cdate);
}
```

6.5.14 f_wsettimedate

Set time and date on a file or on a directory with unicode16 name.

A recommended format for the use of the time date fields is given in.

Note:

The time/date data is simply two 16-bit numbers associated with the specified file that the developer is free to use as desired.

```
int f_wsettimedate(
             const W_CHAR    *filename,
             unsigned short  ctime,
             unsigned short  cdate
)
```

Parameter	Description
filename	Unicode 16 name of file.
ctime	Creation time of file or directory.
cdate	Creation date of file or directory.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   unsigned short ctime,cdate;

   ctime = (15<<11)+(30<<5)+(23>>1);
   /* 15:30:22 */
   cdate = ((2002-1980)<<9)+(11<<5)+(3);
   /* 2002.11.03. */

   f_wmkdir("subfolder");/* creating directory */
   f_wsettimedate("subfolder",ctime,cdate);
}
```

6.5.15 f_gettimedate

Get time and date information from a file or directory. This field is automatically set by the system when a file or directory is created and when a file is closed.

Note:

The time/date data are simply two 16-bit numbers associated with the specified file that the developer is free to use as desired.

```
int f_gettimedate(
             const char      *filename,
             unsigned short  *pctime,
             unsigned short  *pcdate)
```

Parameter	Description
filename	Target file or directory.
pctime	Pointer where to store the time.
pcdate	Pointer where to store the date.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   unsigned short t,d;
   if (!f_gettimedate("subfolder",&t,&d))
   {
      unsigned short sec=(t & 0x001f) << 1;
      unsigned short minute=((t & 0x07e0) >> 5);
      unsigned short hour=((t & 0x0f800) >> 11);
      unsigned short day= (d & 0x001f);
      unsigned short month= ((d & 0x01e0) >> 5);
      unsigned short year=1980+((d & 0xfe00) >> 9);
      printf ("Time: %d:%d:%d",hour,minute,sec);
      printf ("Date: %d.%d.%d",year,month,day);
   }
   else
   {
      printf ("File time cannot retrieved!"
   }
}
```

6.5.16 f_wgettimedate

Get time and date information from a file or directory with a Unicode 16 name. This field is automatically set by the system when a file or directory is created and when a file is closed.

Note:

The time/date data are simply two 16-bit numbers associated with the specified file that the developer is free to use as desired.

```
int f_wgettimedate(
             const W_CHAR    *filename,
             unsigned short  *pctime,
             unsigned short  *pcdate)
```

Parameter	Description
filename	Unicode 16 name of target file or directory.
pctime	Pointer where to store the time.
pcdate	Pointer where to store the date.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   unsigned short t,d;
   if (!f_wgettimedate("subfolder",&t,&d))
   {
      unsigned short sec=(t & 0x001f) << 1;
      unsigned short minute=((t & 0x07e0) >> 5);
      unsigned short hour=((t & 0x0f800) >> 11);
      unsigned short day= (d & 0x001f);
      unsigned short month= ((d & 0x01e0) >> 5);
      unsigned short year=1980+((d & 0xfe00) >> 9);
      wprintf ("Time: %d:%d:%d",hour,minute,sec);
      wprintf ("Date: %d.%d.%d",year,month,day);
   }
   else
   {
      wprintf ("File time cannot retrieved!"
   }
}
```

6.5.17 f_setpermission

This sets the file or directory permission field associated with a file.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system

```
int f_setpermission(
          const char      *filename,
          unsigned long   secure)
```

Parameter	Description
filename	Target file.

secure	32 bit number to associate with file name
	The first six bits are reserved for use by the system, as follows:
	`#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))`
	`#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))`
	`#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))`
	`#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))`
	`#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))`
	`#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))`

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   f_mkdir("subfolder");/* creating directory */
   f_setpermission ("subfolder",0x00ff0000);
}
```

6.5.18 f_wsetpermission

This sets the file or directory permission field associated with a file with unicode 16 name.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system

```
int f_wsetpermission(
              const W_CHAR    *filename,
              unsigned long   secure)
```

Parameter	Description
filename	Unicode16 name of target file.

secure	32 bit number to associate with file name
	The first six bits are reserved for use by the system, as follows:
	`#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))`
	`#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))`
	`#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))`
	`#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))`
	`#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))`
	`#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))`

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_INVALIDDRIVE	Drive does not exist.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   f_mkdir(“subfolder”);/* creating directory */
   f_wsetpermission (“subfolder”,0x00ff0000);
}
```

6.5.19 f_getpermission

Retrieves file or directory permission field associated with a file.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system

```
int f_getpermission(
             const char      *filename,
             unsigned long   *psecure)
```

Parameter	Description
filename	Target file.

psecure	Pointer to where to store permission 32 bit number associated with file name.
	The first six bits are reserved for use by the system, as follows:
	`#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))`
	`#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))`
	`#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))`
	`#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))`
	`#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))`
	`#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))`

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   unsigned long secure;

   if (!f_getpermission (“subfolder”,&secure))
   {
      printf (“permission is: %d”,secure);
   }
   else
      printf (“Permission cannot be retrieved!”);
}
```

6.5.20 f_wgetpermission

Retrieves file or directory permission field associated with a file with a unicode 16 name.

Every file/directory in the file system has an associated 32-bit field; it is known as the permission setting. Except for the top 6 bits, this field is freely programmable by the developer and could, for instance, be used to create a user access system

```
int f_wgetpermission(
             const W_CHAR    *filename,
             unsigned long   *psecure)
```

Parameter	Description
filename	Unicode16 name of target file.

psecure	Pointer to where to store permission 32 bit number associated with file name.
	The first six bits are reserved for use by the system, as follows:
	`#define FSSEC_ATTR_ARC      (0x20UL<<(31-6))`
	`#define FSSEC_ATTR_DIR      (0x10UL<<(31-6))`
	`#define FSSEC_ATTR_VOLUME   (0x08UL<<(31-6))`
	`#define FSSEC_ATTR_SYSTEM   (0x04UL<<(31-6))`
	`#define FSSEC_ATTR_HIDDEN   (0x02UL<<(31-6))`
	`#define FSSEC_ATTR_READONLY (0x01UL<<(31-6))`

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_INVALIDNAME	File or directory name contains invalid characters.
F_ERR_NOTFOUND	File or directory not found.

Example

```
void myfunc(void)
{
   unsigned long secure;
   if (!f_wgetpermission (“subfolder”,&secure))
   {
      wprintf (“permission is: %d”,secure);
   }
   else
      wprintf (“Permission cannot be retrieved!”);
}
```

6.5.21 f_truncate

Opens a file for writing and truncates it to the specified length. If the length is greater than the length of the existing file, then the file is padded with zeroes to the truncated length.

```
F_FILE *f_truncate(
             const char      *filename,
             unsigned long   length
)
```

Parameter	Description
filename	File to be opened.
length	New length of file

Return Value	Condition
Pointer to the associated opened file handle.	File successfully opened.
0	File could not be opened.

Example

```
int mytruncatefunc(char *filename, unsigned long length)
{
   F_FILE *file=f_truncate(filename,length);

   if(!file)
      printf("File not found");
   else
   {
      printf("File %s truncated to %d bytes,
      filename, length);
      f_close(file);
   }
   return 0;
}
```

6.5.22 f_wtruncate

Opens a file with a unicode 16 name for writing and truncates it to the specified length. If the length is greater than the length of the existing file, then the file is padded with zeroes to the truncated length.

```
F_FILE *f_wtruncate(
             const W_CHAR    *filename,
             unsigned long   length
)
```

Parameter	Description
filename	Unicode 16 file to be opened.
length	New length of file

Return Value	Condition
Pointer to the associated opened file handle.	File successfully opened.
0	File could not be opened.

Example

```
int mywtruncatefunc(W_CHAR *filename, unsigned long length)
{
  F_FILE *file=f_wtruncate(filename,length);
  if(!file)
    printf(“File not found”);
  else
  {
    printf(“File %s truncated to %d bytes,
    filename, length);
    f_close(file);
  }

  return 0;
}
```

6.5.23 f_ftruncate

If a file is opened for writing, then this function truncates it to the specified length. If the length is greater than the length of the existing file, then the file is padded with zeroes to the truncated length.

```
int f_ftruncate(
             F_FILE          *filehandle,
             unsigned long   length
)
```

Parameter	Description
filehandle	Open file handle.
length	New length of file.

Return Value	Condition
F_NO_ERROR	Success.
Error code	Error condition.

Example

```
int mytruncatefunc(F_FILE *file, unsigned long length)
{
   int ret=f_ftruncate(filename,length);

   if (ret)
   {
      printf(“error:%d\n”,ret);
   }
   else
   {
      printf(“File is truncated to %d bytes”, length);
   }

   return ret;
}
```

6.5.24 f_stat

Get information about a file. This function retrieves information by filling the F_STAT structure passed to it. It sets file size, creation time/date, the drive number where the file is located, and secure attributes.

```
int f_stat(
             const char      *filename,
             F_STAT          *stat
)
```

Parameter	Description
filename	File.
stat	Pointer to F_STAT structure to be filled.

Return Value	Condition
F_NO_ERROR	Success.
Error code	Error condition.

Example

```
void myfunc(void)
{
   F_STAT stat;
   if (f_stat("myfile.txt",&stat))
   {
      printf ("error");
      return;
   }
   printf ("filesize:%d",stat.filesize);
}
```

6.5.25 f_wstat

Get information about a file with a unicode 16 name. This function retrieves information by filling the F_STAT structure passed to it. It sets file size, creation time/date, the drive number where the file is located, and secure attributes.

```
int f_stat(
             const W_CHAR    *filename,
             F_STAT          *stat
)
```

Parameter	Description
filename	File with a unicode 16 name.
stat	Pointer to F_STAT structure to be filled.

Return Value	Condition
F_NO_ERROR	Success.
Error code	Error condition.

Example

```
void myfunc(void)
{
   F_STAT stat;
   if (f_wstat("myfile.txt",&stat))
   {
      printf ("error");
      return;
   }
   printf ("filesize:%d",stat.filesize);
}
```

6.6 Read/Write Functions

6.6.1 f_open

Opens a file.

```
F_FILE *f_open(
             const char      *filename,
             const char      *mode
)
```

Parameter	Description	
filename	File to be opened.	
mode	Pointer to open mode string.	
	Mode	**Description**
	"r"	Open an existing file for reading. The stream is positioned at the beginning of the file.
	"r+"	Open an existing file for reading and writing. The stream is positioned at the beginning of the file.
	"w"	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
	"w+"	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
	"a"	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
	"a+"	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Return Value	Condition
Pointer to the associated opened file handle.	File successfully opened.
0	File could not be opened.

Example

```
void myfunc(void)
{
   F_FILE *file;
   char c;

   file=f_open("myfile.bin","r");
   if (!file)
   {
      printf ("File cannot be opened!");
      return;
   }

   f_read(&c,1,1,file); /* read 1byte */
   printf ("'%c' is read from file",c);
   f_close(file);
}
```

6.6.2 f_wopen

Opens a file with Unicode 16 file name.

```
F_FILE *f_wopen(
             const W_CHAR    *filename,
             const char      *mode
)
```

Parameter	Description	
filename	Unicode 16 name of target file.	
mode	Pointer to open mode string.	
	Mode	**Description**
	"r"	Open an existing file for reading. The stream is positioned at the beginning of the file.
	"r+"	Open an existing file for reading and writing. The stream is positioned at the beginning of the file.
	"w"	Truncate file to zero length or create file for writing. The stream is positioned at the beginning of the file.
	"w+"	Open for reading and writing. The file is created if it does not exist; otherwise it is truncated. The stream is positioned at the beginning of the file.
	"a"	Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
	"a+"	Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

Return Value	Condition
Pointer to the associated opened file handle.	File successfully opened.
0	File could not be opened.

Example

```
void myfunc(void)
{
   F_FILE *file;
   char c;

   file=f_wopen(“myfile.bin”,”r”);
   if (!file)
   {
      wprintf (“File cannot be opened!”);
      return;
   }

   f_read(&c,1,1,file); /* read 1byte */
   wprintf (“’%c’ is read from file”,c);
   f_close(file);
}
```

6.6.3 f_close

Closes a previously opened file.

```
int f_close(
             F_FILE       *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_NOTOPEN	File not open.
F_ERR_INVALIDDRIVE	File handle points to invalid drive.
F_ERR_ONDRIVE	Cannot be written into device

Example

```
void myfunc(void)
{
   F_FILE *file;
   char *string=”ABC”;

   file=f_open(“myfile.bin”,”w”);

   if (!file)
   {
      printf (“File cannot be opened!”);
      return;
   }

   f_write(string,3,1,file); /* write 3byte */
   if (!f_close(file))
   {
      printf (“File stored”);
   }
   else printf (“file close error”);
}
```

6.6.4 f_flush

Flush data to the media. This command allows the user to update the file on the media and therefore update the failsafe state of the file without again closing and opening the file. Once this command has completed, the new state of the file will be restored after system restarting or a system failure.

```
int f_flush(
             F_FILE       *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_NOTOPEN	File not open.
F_ERR_INVALIDDRIVE	File handle points to invalid drive.
F_ERR_ONDRIVE	Cannot be written into device

Example

```
void myfunc(void)
{
   F_FILE *file;
   char *string=”ABC”;

   file=f_open(“myfile.bin”,”w”);
   if (!file)
   {
      printf (“File cannot be opened!”);
      return;
   }

   f_write(string,3,1,file); /* write 3byte */

   if (!f_flush(file))
   {
      printf (“New data is now failsafe”);
   }
   else printf (“file flush error”);
}
```

6.6.5 f_write

Write data into file at current position. File has to be opened with "r+", “w”, “w+”, "a+" or “a”. The file pointer is moved forward by the number of bytes successfully written.

Note

Data is NOT permanently stored to the media until either an **f_flush** or **f_close** has been done on the file.

```
int f_write(
             const void   *buf,
             long         size,
             long         size_st,
             F_FILE       *filehandle
)
```

Parameter	Description
buf	Pointer to data buffer.
size	Size of items to be written.
size_st	Number of items to be written.
filehandle	File handle to write to.

Return Value	Condition
Number of items written	

Example

```
void myfunc(void)
{
   F_FILE *file;
   char *string=”ABC”;

   file=f_open(“myfile.bin”,”w”);
   if (!file)
   {
      printf (“File cannot be opened!”);
      return;
   }
   if (f_write(string,1,3,file)!=3)
   {  /* write 3bytes */
      printf (“not all items written”);
   }
   f_close(file);
}
```

6.6.6 f_read

Read data from the current file position. File has to be opened with “r”, "r+", "w+" or "a+". The file pointer is moved forward by the number of bytes read.

```
int f_read(
             void         *buf,
             long         size,
             long         size_st,
             F_FILE       *filehandle
)
```

Parameter	Description
buf	Pointer to data buffer.
size	Size of each of items to be read.
size_st	Number of items to be read.
filehandle	File handle to read from.

Return Value	Condition
Number of items read.	

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,”r”);

   long size=f_filelength(filename);
   if (!file)
   {
      printf (“%s Cannot be opened!”,filename);
      return 1;
   }
   if (f_read(buffer,1,size,file)!=size)
   {
      printf (“not all items read”);
   }
   f_close(file);
   return 0;
}
```

6.6.7 f_seek

Move read/write position in the file.

```
int f_read(
             F_FILE        *filehandle,
             long          offset,
             long          whence
)
```

Parameter	Description	
filehandle	File handle to read from.	
offset	Relative byte position according to whence	
whence	Where to calculate offset from. Whence parameter could be one of:	
	F_SEEK_CUR	Current position of file pointer.
	F_SEEK_END	End of file.
	F_SEEK_SET	Beginning of file

Return Value	Condition
F_NO_ERROR	Success.
F_ERR_NOTFORREAD	File not open for reading.
F_ERR_NOTUSEABLE	whence parameter is invalid.
F_ERR_INVALIDDRIVE	File handle points to invalid drive.
F_ERR_ONDRIVE	Drive is not readable.

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,”r”);
   f_read(buffer,1,1,file); /* read 1 byte */
   f_seek(file,0,F_SEEK_SET);
   f_read(buffer,1,1,file);/*read the same 1 byte */
   f_seek(file,-1,F_SEEK_END);
   f_read(buffer,1,1,file); /* read last 1 byte */
   f_close(file);
   return 0;
}
```

6.6.8 f_eof

Check whether the current position in the opened target file is the end of the file.

```
int f_eof(
             F_FILE        *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
0	Not at end of file.
!=0	End of file or invalid file handle

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");

   while (!f_eof())
   {
      if (!buffsize) break;
      buffsize--;
      f_read(buffer++,1,1,file);
   }

   f_close(file);
   return 0;
}
```

6.6.9 f_rewind

Set the current file position in the open target file to the beginning.

```
int f_rewind(
             F_FILE       *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
0	Success.
!=0	Failed: invalid file handle.

Example

```
void myfunc(void)
{
   char buffer[4];
   char buffer2[4];

   F_FILE *file=f_open("myfile.bin",”r”);

   if (file)
   {
      f_read(buffer,4,1,file);
      f_rewind(file); /* rewind file pointer */
      f_read(buffer2,4,1,file);
      /* read from beginning */
      f_close(file);
   }
   return 0;
}
```

6.6.10 f_putc

Write a character to the open target file at the current file position. The current file position is incremented.

```
int f_putc(
             int          ch,
             F_FILE       *filehandle
)
```

Parameter	Description
ch	Character to be written
filehandle	File handle of target.

Return Value	Condition
Written character	Success.
-1	Write Failed.

Example

```
void myfunc (char *filename, long num)
{
   F_FILE *file=f_open(filename,"w");

   while (num--)
   {
      int ch='A';
      if(ch!=(f_putc(ch))
      {
         printf("f_putc error!");
         break;
      }
   }
   f_close(file);
   return 0;
}
```

6.6.11 f_getc

Read a character from the current position in the open target file.

```
int f_getc(
              F_FILE       *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
Character read from file	Success.
-1	Read Failed.

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,”r”);
   while (buffsize--)
   {
      int ch;
      if((ch=f_getc(file))== -1)
         break;
      *buffer++=(char)ch;
      buffsize--;
   }

   f_close(file);
   return 0;
}
```

6.6.12 f_seteof

Move the end of file to the current file position. All data after the new EOF position are lost.

```
int f_seteof(
             F_FILE       *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
0	Success.
error code	Failed.

Example

```
int mytruncatefunc(char *filename, int position)
{
   F_FILE *file=f_open(filename,”r”);

   f_seek(file,position,F_SEEK_SET);

   if(f_seteof(file))
   {
      printf(“Truncate Failed\n”);
      return 1;
   }

   f_close(file);
   return 0;
}
```

6.6.13 f_tell

Tell the current file position in the target file.

```
long f_tell(
            F_FILE       *filehandle
)
```

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
Current read or write file position	

Example

```
int myreadfunc(char *filename, char *buffer, long buffsize)
{
   F_FILE *file=f_open(filename,"r");

   printf ("Current position %d",f_tell(file));
   f_read(buffer,1,1,file); /* read 1 byte */
   printf ("Current position %d",f_tell(file));
   f_read(buffer,1,1,file); /* read 1 byte */
   printf ("Current position %d",f_tell(file));

   f_close(file);
   return 0;
}
```

7 Message Interface Reference

SCIOPTA is a message based real-time operating system. Messages are the preferred method for interprocess communication (IPC). You can directly use predefined SDD and SFS messages to use the file system.

Directly using SCIOPTA SDD and SFS messages makes only sense in SDD Mode which is used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

7.1 SDD Messages

SDD messages are predefined standardized messages to be used with SCIOPTA devices. As the SCIOPTA Flash File System is the same way organized as a SCIOPTA device these messages will be used to access some standard file system methods.

Please consult the SCIOPTA ARM - SDD Device Driver, User's Guide for a detailed description of the SDD messages.

Some standard SDD messages:

Message	Description
SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY	Closes a device.
SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY	Sets or gets specific parameters to/from device drivers on the hardware layer.
SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY	Opens a device for read, write or read/write.
SDD_DEV_READ / SDD_DEV_READ_REPLY	Reads data from a device driver.
SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY	Writes data to a device driver.
SDD_ERROR	Used by device driver and other processes which do not use reply messages as answer of request messages for returning error codes.
SDD_MAN_ADD / SDD_MAN_ADD_REPLY	Adds a new device in the device driver system.
SDD_MAN_GET / SDD_MAN_GET_REPLY	Gets the SDD device descriptor (including the process IDs and handle) of a registered device.
SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY	Gets the device descriptor (including the process IDs and handle) of the first registered device from the SDD manager's device list.
SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY	Gets the device descriptor (including the process IDs and handle) of the next registered device from the SDD manager's device list.
SDD_MAN_RM / SDD_MAN_RM_REPLY	Removes a device from the device driver system.
SDD_OBJ_DUP / SDD_OBJ_DUP_REPLY	Creates a copy of a device with identical data structures.
SDD_OBJ_RELEASE / SDD_OBJ_RELEASE_REPLY	Releases an on-the-fly object.
SDD_OBJ_SIZE_GET / SDD_OBJ_SIZE_GET_REPLY	Gets the size of an SDD object.
SDD_OBJ_TIME_GET / SDD_OBJ_TIME_GET_REPLY	Gets the time from device drivers.
SDD_OBJ_TIME_SET / SDD_OBJ_TIME_SET_REPLY	Sets the time of device drivers.

7.2 Structures

The SCIOPTA device driver and file system messages and function interface are using some standard message structures which are described in this chapter.

7.2.1 Base SDD Object Descriptor Structure sdd_baseMessage_t

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

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

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

Members	Description
id	Standard SCIOPTA message ID.
error	Error code.
handle	Handle of the SDD object. In the file system the handle is a pointer to a structure which further specifies the file. The user of a file which is opening and closing the file, reading from the file and writing to the file does not need to know the handle and the handle structure. The user will usually get the SDD device descriptor by using the sdd_manGetByName function call. The SDD file manager will return the SDD device descriptor including the handle. Only processes inside the SDD object may access and use the handle.

Header

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

7.2.2 Standard SDD Object Descriptor Structure sdd_obj_t

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

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

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

Members	Description
base	Specifies the base SDD object descriptor structure of an SDD object (see chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2).
manager	Contains a manager access handle. It is a pointer to a structure which further specifies the manager. This is only used if the SDD object descriptor describes an SDD manager and is only used in SDD manager messages (SDD_MAN_XXX). For SDD file managers a 0 defines an SDD root manager. You do not need to write anything in the opaque manager handle if you are using the function interface as this is done in the interface layer.
type	Type of the SDD object. More than one value can be defined and must be separated by OR instructions. The values determine the type of messages which are handled by the SDD object. This member can be one or more of the following values:

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

Members	Description	
type(cont.)	SDD_DEV_TYPE	The SDD object is an SDD device. It handles the following device messages: SDD_DEV_OPEN / SDD_DEV_OPEN_REPLY SDD_DEV_DUALOPEN / SDD_DEV_DUALOPEN_REPLY SDD_DEV_CLOSE / SDD_DEV_CLOSE_REPLY SDD_DEV_READ / SDD_DEV_READ_REPLY SDD_DEV_WRITE / SDD_DEV_WRITE_REPLY SDD_DEV_IOCTL / SDD_DEV_IOCTL_REPLY
	SDD_FILE_TYPE	The SDD object is an SDD file. It handles the following file messages: SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY
	SDD_NET_TYPE	The SDD object is an SDD protocol or network device. It handles the following network messages: SDD_NET_RECEIVE / SDD_NET_RECEIVE_REPLY SDD_NET_RECEIVE_2 / SDD_NET_RECEIVE_2_REPLY SDD_NET_RECEIVE_URGENT / SDD_NET_RECEIVE_URGENT_REPLY SDD_NET_SEND / SDD_NET_SEND_REPLY
	SFS_DIR_TYPE	The SDD object is an SFS directory. It handles the following file messages: SDD_MAN_ADD / SDD_MAN_ADD_REPLY SDD_MAN_RM / SDD_MAN_RM_REPLY SDD_MAN_GET / SDD_MAN_GET_REPLY SDD_MAN_GET_FIRST / SDD_MAN_GET_FIRST_REPLY SDD_MAN_GET_NEXT / SDD_MAN_GET_NEXT_REPLY
name	Contains the name of the SDD object. The name must be unique within a domain. A manager corresponds to a domain.	
controller	The controller process ID of the SDD object.	
sender	The sender process ID of the SDD object. If the SDD object is a device driver, the sender process sends the data to the physical layer. It usually receives SDD_DEV_WRITE or SDD_NET_SEND messages and can reply with the corresponding reply messages.	
receiver	The receiver process ID of the SDD object. If the SDD object is a device driver, the receiver process receives the data from the physical layer. In passive synchronous mode the receiver process receives the SDD_DEV_READ messages and replies with the SDD_DEV_READ_REPLY message. In active asynchronous mode (used by network devices) the receiver process sends a SDD_NET_RECEIVE, SDD_NET_RECEIVE_2 or SDD_NET_RECEIVE_URGENT message.	

Header

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

7.2.3 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**).

7.3 SCIOPTA Flash File System Messages

7.3.1 SDD_FILE_RESIZE / SDD_FILE_RESIZE_REPLY

This message is used to increase or decrease the size of an existing file.

The user sends an **SDD_FILE_RESIZE** request message to the device driver sender process. The device driver sender process replies with the **SDD_FILE_RESIZE_REPLY** reply message.

If the file encounters an error, the error code will be included in the reply message.

Message ID	Description
SDD_FILE_RESIZE	Request message.
SDD_FILE_RESIZE_REPLY	Reply message.

sdd_fileResize_t Structure

```
typedef struct sdd_fileResize_s {
   sdd_baseMessage_t           base;
   ssize_t                     size;
} sdd_fileResize_t;
```

Members	Description
base	Specifies the base SDD object descriptor structure of an SDD object. Please consult chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2.
size	New size of the file.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure.

Error	Description
EBADF	The member handle of the sdd_baseMessage_t structure is not valid.

Header

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

7.3.2 SDD_FILE_SEEK / SDD_FILE_SEEK_REPLY

This message is used to positioning write and read pointers in a file.

The user sends an **SDD_FILE_SEEK** request message to the file process. The file process replies with an **SDD_FILE_SEEK_REPLY** message.

If the file encounters an error, the error code will be included in the reply message.

Message ID	Description
SDD_FILE_SEEK	Request message.
SDD_FILE_SEEK_REPLY	Reply message.

sdd_fileSeek_t Structure

```
typedef struct sdd_fileSeek_s {
   sdd_baseMessage_t             base;
   off_t                         offset;
   int                           whence;
} sdd_fileSeek_t;
```

Members	Description	
base	Specifies the base SDD object descriptor structure of an SDD object (see chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2).	
offset	Number of bytes to shift the read/write pointer.	
whence	Origin from where the read/write pointer will be shifted. This member can be one of the following values:	
	Value	**Description**
	SEEK_SET	Absolute position
	SEEK_CUR	Calculated from the actual position (negative shifts are also supported)
	SEEK_END	Calculated from the end.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure.

Error	Description
EBADF	The member handle of the sdd_baseMessage_t structure is not valid.

Header

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

7.3.3 SFS_ASSIGN / SFS_ASSIGN_REPLY

This message is received from a user process. The SDD file manager will assign the specified SDD file device which holds the file system.

The user (set-up) process sends a SFS_ASSIGN message to the file manager process. The file manager process replies with an SFS_ASSIGN_REPLY message.

If the file manager encounters an error, the error code will be included in the reply message.

Message ID	Description
SFS_ASSIGN	Request message.
SFS_ASSIGN_REPLY	Reply message.

sfs_assign_t Structure

```
typedef struct sfs_assign_s {
   sdd_obj_t                     object;
} sfs_assign_t;
```

Members	Description
object	Specifies the base SDD object descriptor structure of an SDD object (see chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2). The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure.

Error	Description
EBADF	The member handle of the sdd_baseMessage_t structure inside the sdd_obj_t structure is not valid.

Header

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

7.3.4 SFS_MOUNT / SFS_MOUNT_REPLY

This message is received from a user process. The SDD file manager will mount the object to the specified directory. The directory is specified in the manager handle (member **manager** of the sdd_obj_t structure, see chapter 7.2.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 7-3)

The user process sends an **SFS_MOUNT** message and replies with an **SFS_MOUNT_REPLY** message.

If the file manager encounters an error, the error code will be included in the reply message.

Message ID	Description
SFS_MOUNT	Request message.
SFS_MOUNT_REPLY	Reply message.

sfs_mount_t Structure

```
typedef struct sfs_mount_s {
   sdd_obj_t                    obj;
} sfs_mount_t;
```

Members	Description
obj	Specifies the base SDD object descriptor structure of an SDD object (see chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2). The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure.

Error	Description
EBADF	The member handle of the sdd_baseMessage_t structure inside the sdd_obj_t structure is not valid.

Header

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

7.3.5 SFS_SYNC / SFS_SYNC_REPLY

This message is received from a user process. The SDD file manager will synchronize the data residing in a cache with the data residing in the assigned SDD file device.

The user process sends an **SFS_SYNC** message and the file manager process replies with an **SFS_SYNC_REPLY** message.

If the file manager encounters an error, the error code will be included in the reply message.

Message ID	Description
SFS_SYNC	Request message.
SFS_SYNC_REPLY	Reply message.

sfs_sync_t Structure

```
typedef struct sfs_sync_s {
   sdd_baseMessage_t            base;
} sfs_sync_t;
```

Members	Description
base	Specifies the base SDD object descriptor structure of an SDD object (see chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2).

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure.

Error	Description
EBADF	The member handle of the sdd_baseMessage_t structure is not valid.

Header

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

7.3.6 SFS_UNMOUNT / SFS_UNMOUNT_REPLY

This message is received from a user process. The SDD file manager will unmount the specified directory and return the mounted object.

The user process sends an **SFS_UNMOUNT** message and replies with an **SFS_UNMOUNT_REPLY** message. The same **sfs_mount_t** structure will be used as for the **SFS_MOUNT** message.

If the file manager encounters an error, the error code will be included in the reply message.

Message ID	Description
SFS_UNMOUNT	Request message.
SFS_UNMOUNT_REPLY	Reply message.

sfs_mount_t Structure

```
typedef struct sfs_unmount_s {
   sdd_obj_t                obj;
} sfs_unmount_t;
```

Members	Description
obj	Specifies the base SDD object descriptor structure of an SDD object (see chapter 7.2.1 “Base SDD Object Descriptor Structure sdd_baseMessage_t” on page 7-2). The SDD object is usually a device but it can also be a file, a directory, a network protocol or another SCIOPTA object.

Errors

The following errors can occur. The error code is included in the **error** member of the **sdd_baseMessage_t** structure.

Error	Description
EBADF	The member handle of the sdd_baseMessage_t structure inside the sdd_obj_t structure is not valid.

Header

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

8 SCIOPTA SFFS Function Interface Reference

8.1 SDD Sciopta Device Driver Function Interface

In SDD Mode a user process accesses the file system by using the SDD device driver function interface. This must be used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

As the SCIOPTA Flash File System is the same way organized as a SCIOPTA device these functions will be used to access some standard file system methods.

Please consult the SCIOPTA ARM - SDD Device Driver, User's Guide for a detailed description of the SDD functions.

Some standard SDD functions:

Function	Description
sdd_devAread	Reads data in an asynchronous mode from a device driver.
sdd_devClose	Closes an open device.
sdd_devIoctl	Gets and sets specific parameters in device drivers on the hardware layer.
sdd_devOpen	Opens device for read, write or read/write, or to create a device.
sdd_devRead	Reads data from a device driver.
sdd_devWrite	Writes data to a device driver.
sdd_manAdd	Adds a new device in the device driver system by registering it at a device manager process.
sdd_manGetByName	Gets the SDD device descriptor of a registered device from the manager's device list by giving the name as parameter.
sdd_manGetByPath	Gets the SDD device descriptor of a registered device from the manager's device list by giving the path as parameter.
sdd_manGetFirst	Gets the SDD device descriptor of the first registered device from the manager's device list.
sdd_manGetNext	Gets the SDD device descriptor of the next registered device from the manager's device list.
sdd_manNoOfItems	Gets the number of registered devices of the manager device list.
sdd_manGetRoot	Gets (creates) an SDD object descriptor of a root manager process.
sdd_manRm	Removes registered device, files and directories.
sdd_objDup	Creates a copy of the SDD device descriptor of a device by adopting the same state and context as the original device.
sdd_objFree	Releases and frees an SDD object mainly an on-the-fly object.
sdd_objResolve	Returns the last struct manager in a given path for hierarchical organized managers.
sdd_objSizeGet	Gets the size of an SDD object.
sdd_objTimeGet	Gets the time of an SDD device.
sdd_objTimeSet	Sets the time of an SDD device.

8.2 SFS SCIOPTA File System Function Interface

The SFS Functions extend the SDD Functions to support file system functionality. This must be used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

8.2.1 SCIOPTA SDD Structures

The SCIOPTA device driver and file system messages and function interface are using some standard message structures

Please consult chapter 7.2.2 “Standard SDD Object Descriptor Structure sdd_obj_t” on page 7-3 for more information about the sdd_obj_t type and chapter 7.2.3 “NEARPTR and FARPTR” on page 7-5 for more information about theNEARPTR define.

8.2.2 sfs_assign

The **sfs_assign** function assigns a SDD file device to an SDD file manager. The SDD file device must be open.

```
int sfs_assign (
   sdd_obj_t NEARPTR          fs,
   sdd_obj_t NEARPTR NEARPTR  dev
);
```

Parameter	Description
fs	SDD file manager descriptor.
dev	SDD file device descriptor.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.3 sfs_close

The **sfs_close** function is used to close a file object. This a basically only a wrapper for the **sdd_devClose** function.

```
int NEARPTR sfs_close (
   sdd_obj_t NEARPTR NEARPTR    file
);
```

Parameter	Description
file	SDD file descriptor of the file to be closed.

Return Value	Condition	
>= 0	Success.	
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:	
	Value	**Description**
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.
	EIO	An input/output error occurred.
	SC_ENOTSUPP	This request is not supported.

Header

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

8.2.4 sfs_copy

The **sfs_copy** function copies the file or the directory from the original location **oldpath** to the new location **newpath** within the SDD file manager. If you want to copy between two SDD file managers, these must reside (mount) in the same file system tree.

```
int sfs_copy (
   sdd_obj_t NEARPTR  dir,
   const char         *oldpath,
   const char         *newpath
);
```

Parameter	Description
dir	SDD directory descriptor.
oldpath	Original file or directory (source) relative to parameter **dir**.
newpath	New file or directory (destination) relative to parameter **dir**.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.5 sfs_create

The **sfs_create** function creates in the specified directory (**dir**) a new file or a new directory with the specified name, type and mode.

```
int sfs_create (
   sdd_obj_t NEARPTR  dir,
   const char         *name,
   sc_msgid_t         type,
   mode_t             mode
);
```

Parameter	Description	
dir	SDD directory descriptor.	
name	Path of the new file or directory relative to the parameter **dir**.	
type	Type of the SDD file object. More than one value can be defined and must be separated by OR instructions. This parameter can be one of the following values:	
	Value	**Description**
	SFS_ATTR_DIR	Defines an SDD directory object. SDD_OBJ_TYPE | SDD_MAN_TYPE | SFS_DIR_TYPE
	SFS_ATTR_FILE	Defines an SDD file object. SDD_OBJ_TYPE | SDD_DEV_TYPE | SDD_FILE_TYPE
mode	0 (reserved for later use)	

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling **sc_miscErrnoGet**

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

8.2.6 sfs_delete

The **sfs_delete** function deletes a file or directory. If a directory needs to be deleted it must be empty.

```
int sfs_delete (
   sdd_obj_t NEARPTR  dir,
   const char         *path
);
```

Parameter	Description
dir	SDD directory descriptor.
path	Path of the starting point of the directory or the file relative to parameter **dir**.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.7 sfs_get

The **sfs_get** function is used to get a directory or file object.

```
sdd_obj_t NEARPTR sfs_get (
   sdd_obj_t NEARPTR        dir,
   const char               *name,
   sc_msgid_t               type
);
```

Parameter	Description
dir	SDD directory descriptor. Please consult chapters
name	Path of the file or directory relative to parameter **dir**.
type	Same as the type member of the sdd_obj_t structure (see chapter)

Return Value	Condition	
Pointer to the SDD descriptor of the directory or file	Success.	
NULL	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:	
	Value	**Description**
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.
	ENOENT	Device does not exists.
	ENOMEM	Not enough memory.
	SC_ENOTSUPP	This request is not supported.

Header

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

8.2.8 sfs_mount

The **sfs_mount** function mounts an SDD file manager or SDD device manager to the directory mount point (type: SFS_DIR_TYPE). Only SDD file managers can be unmount. If you want to unmount device manager they need to be killed.

```
int sfs_mount (
   sdd_obj_t NEARPTR         mountpoint,
   sdd_obj_t NEARPTR NEARPTR dir
);
```

Parameter	Description
mountpoint	SDD file manager descriptor of the mount point.
dir	SDD file manager descriptor or SDD device manager descriptor to be mounted.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.9 sfs_move

The **sfs_move** function moves the file or the directory from the original location **oldpath** to the new location **newpath** within the SDD file manager.

This can also be used to rename files.

If you want to move between two SDD file managers, they must reside (mount) in the same file system tree.

```
int sfs_move (
   sdd_obj_t NEARPTR  dir,
   const char         *oldpath,
   const char         *newpath
);
```

Parameter	Description
dir	SDD directory descriptor.
oldpath	Original file or directory (source) relative to parameter **dir**.
newpath	New file or directory (destination) relative to parameter **dir**.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.10 sfs_open

The **sfs_open** function is used to get and open a directory or file object.

```
sdd_obj_t NEARPTR sfs_open (
   sdd_obj_t NEARPTR       dir,
   const char              *name,
   flags_t                 flags
);
```

Parameter	Description	
dir	SDD directory descriptor.	
name	Path of the new file or directory relative to the parameter **dir**.	
flags	Contains BSD conform flags. This parameter can be one of the following values:	
	Value	**Description**
	O_RDONLY	Opens the device for read only.
	O_WRONLY	Opens the device for write only.
	O_RDWR	Opens the device for read and write.
	O_TRUNC	Decrease a file to length zero.
	O_APPEND	Sets the read/write pointer to the end of the file.
	O_TRUNC and O_APPEND can be ored with O_RDONLY and O_WRONLY. O_RDONLY cannot be ored with O_WRONLY (as it is not equal to O_RDWR!).	

Return Value	Condition		
Pointer to the SDD descriptor of the directory or file	Success.		
NULL	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:		
	Value	**Description**	
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.	
	EINVAL	Invalid parameter.	
	SC_ENOTSUPP	This request is not supported.	

Header

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

8.2.11 sfs_read

The **sfs_read** function is used to read data from a file object. This a basically only a wrapper for the sdd_devRead function.

```
ssize_t NEARPTR sfs_read (
   sdd_obj_t NEARPTR       file
   char                    *buf,
   ssize_t                 size
);
```

Parameter	Description
file	SDD file or directory descriptor.
buf	Buffer where the read data is stored.
size	Size of the data buffer.

Return Value	Condition	
Number of read bytes	Success.	
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:	
	Value	**Description**
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.
	EIO	An input/output error occurred.
	EINVAL	Invalid parameter.
	SC_ENOTSUPP	This request is not supported.

Header

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

8.2.12 sfs_resize

The **sfs_resize** function is used to increase or decrease the size of an existing file object.

Please note: This function is planned by HCC and therefore not yet implemented.

```
ssize_t NEARPTR sfs_resize (
   sdd_obj_t NEARPTR       file
   ssize_t                 size
);
```

Parameter	Description
file	SDD file descriptor.
buf	Buffer where the read data is stored.
size	Size of the data buffer.

Return Value	Condition	
New size of the file	Success.	
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:	
	Value	**Description**
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.
	EINVAL	Invalid parameter.
	ENOMEM	Not enough memory to resize.
	SC_ENOTSUPP	This request is not supported.

Header

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

8.2.13 sfs_seek

The **sfs_seek** function is used to positioning write and read pointers in file object.

```
off_t NEARPTR sfs_seek (
   sdd_obj_t NEARPTR       file,
   off_t                   off,
   flags_t                 whence
);
```

Parameter	Description	
file	SDD file descriptor.	
off	Number of bytes to shift the read/write pointer.	
whence	Origin from where the read/write pointer will be shifted. This parameter can be one of the following values:	
	SEEK_SET	Writes absolute.
	SEEK_CUR	Calculated from the actual position (negative shifts are also supported).
	SEEK_END	Calculated from the end.

Return Value	Condition	
number of bytes the read/ write pointer was shifted.	Success.	
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:	
	Value	**Description**
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.
	EINVAL	Invalid parameter.
	ENOMEM	Not enough memory to resize.
	SC_ENOTSUPP	This request is not supported.

Header

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

8.2.14 sfs_sync

The **sfs_sync** function synchronize the data cache with the physical SDD file device. This forces the system to write the data.

Please note: This function is not implemented in the HCC file system and is therefore meaningless.

```
int sfs_sync (
   sdd_obj_t NEARPTR  fs
);
```

Parameter	Description
fs	SDD file manager descriptor.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.15 sfs_unmount

The **sfs_unmount** function unmounts a mounted SDD file manager of a SDD directory (**mountpoint**).

Only SDD file managers can be unmount. If you want to unmount device manager they need to be killed.

```
int sfs_unmount (
   sdd_obj_t NEARPTR  mountpoint
);
```

Parameter	Description
mountpoint	SDD directory descriptor.

Return Value	Condition
>= 0	Success.
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet

Header

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

8.2.16 sfs_write

The **sfs_write** function is used to write data to a file object. This a basically only a wrapper for the sdd_devWrite function.

```
ssize_t NEARPTR sfs_write (
   sdd_obj_t NEARPTR      file
   char                   *buf,
   ssize_t                size
);
```

Parameter	Description
file	SDD file descriptor.
buf	Buffer where the data to be written is stored.
size	Size of the data buffer.

Return Value	Condition		
Number of written bytes	Success.		
-1	The function failed. The error information can be retrieved by calling sc_miscErrnoGet. The following codes are defined by a standard device drive for the sc_miscErrnoGet return value:		
	Value	**Description**	
	EBADF	The member handle of the sdd_baseMessage_t structure (in parameter self) is not valid.	
	EIO	An input/output error occurred.	
	EINVAL	Invalid parameter.	
	EFBIG	Size of data to be written to big.	
	SC_ENOTSUPP	This request is not supported.	

Header

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

9 Application Programming

9.1 Introduction

System design includes all phase from system analysis, through specification to system design, coding and testing. In this chapter we will give you some useful information of what methods, techniques and structures are available in SCIOPTA to fulfil your real-time requirements for your embedded system.

9.2 System Partition

When you are analysing a real-time system you need to partition a large and complex real-time system into smaller components and you need to design system structures and interfaces carefully. This will not only help in the analysing and design phase, it will also help you maintaining and upgrading the system.

In a SCIOPTA controlled real-time system you have different structure elements available to decompose the whole system into smaller parts.

Please consult the chapter "Application Programming" of the ARM - Kernel, User's Guide for more information how to use

- modules
- processes
- interprocess communications
- messages
- message pools
- triggers
- daemons
- trap interface
- error handling
- interrupt handling.

9.3 Flash File System Block Diagram

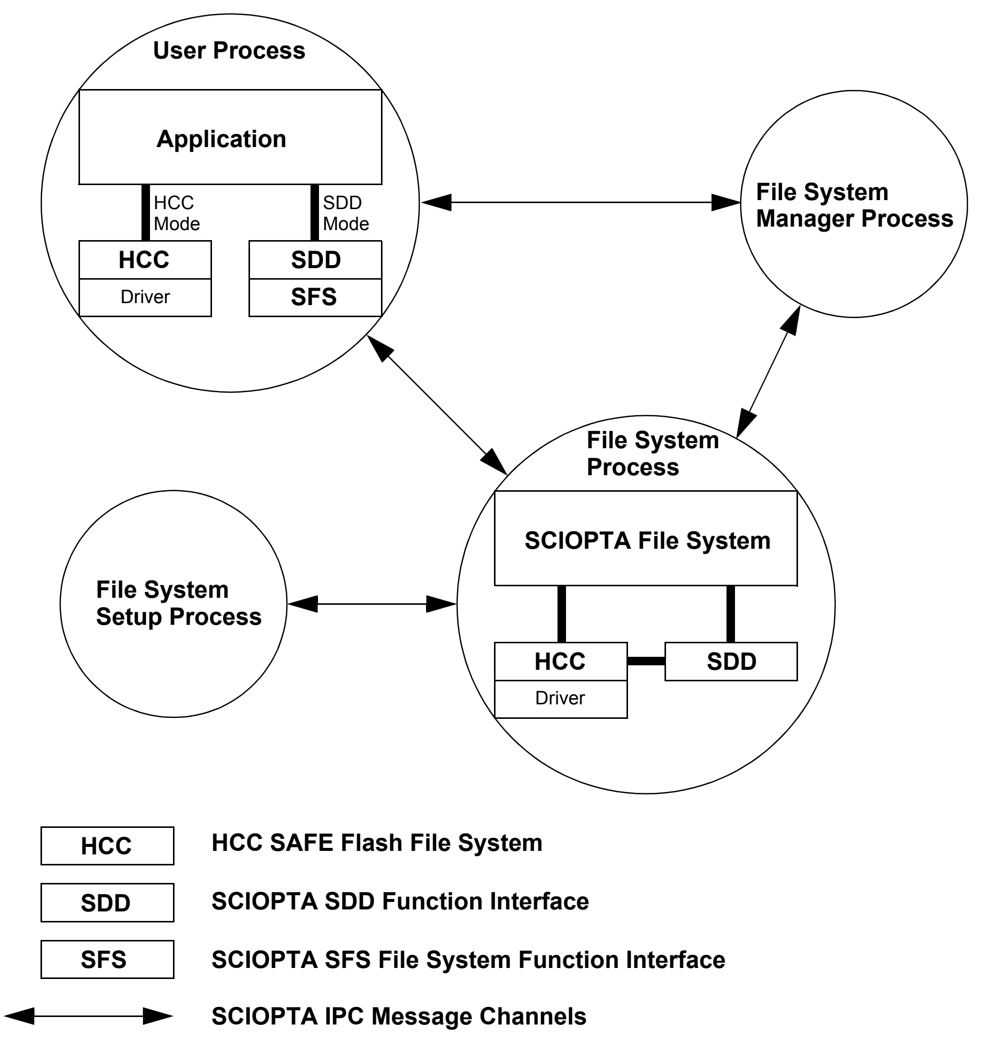

User Process
Application
HCC Mode
SDD Mode
HCC
Driver
SDD
SFS
File System Manager Process
File System Process
SCIOPTA File System
HCC
Driver
SDD
File System Setup Process
HCC
HCC SAFE Flash File System
SDD
SCIOPTA SDD Function Interface
SFS
SCIOPTA SFS File System Function Interface
SCIOPTA IPC Message Channels

Figure 9-1: SCIOPTA Flash File System Block Diagram

9.4 File System Process

The file system process includes the main sciopta file system initialization functions. It is using the HCC Safe File System (HCC Safe API), the HCC Safe File System drivers and the SDD (SCIOPTA Device Driver) interface library.

The file system process registers the file system at the file system manager process.

For each physical device you need a file system process.

In order to use the SCIOPTA Flash File System you need to include some HCC SAFE Flash File System source files and SCIOPTA integration files in your project.

9.4.1 HCC Safe File System

The SCIOPTA Flash File System uses and integrates the Safe Flash File System from HCC Embedded. The HCC Safe File System consist of the following files:

fsm.c	HCC SAFE intermediate layer
fsf.c	HCC SAFE standard API
fsmf.c	SAFE Standard API multi-thread wrapper.
port_s.c	Ported functions

The user does not need to modify these files. They must be included in the application build process.

The HCC Safe File System files can be found at:

File location<installation_folder>\sciopta\<version>\sfs\hcc\effs\common\

9.4.2 SCIOPTA - HCC Integration

These files include the SCIOPTA integration in the HCC Safe Flash File System.

sc2effs.c	HCC EFFS integration layer
sc2hcc.c	SCIOPTA HCC integration layer

The user does not need to modify these files. They must be included in the application build process.

The SCIOPTA - HCC integration files can be found at:

File location<installation_folder>\sciopta\<version>\sfs\hcc\src\

9.5 File System Manager Process

The file system manager process is a standard manager process included in the SCIOPTA gdd library (SCP_manager).

The user just need to define a prioritized static process with the process name SCP_devman and the process function SCP_manager.

9.6 File System Setup Process

This is a user written process and is mainly used for mounting file systems and to wait for file systems to be initialized and up-and-running.

The file system setup process is using SDD messages and SDD functions. Please consult chapter 7 “Message Interface Reference” on page 7-1 and chapter 8 “SCIOPTA SFFS Function Interface Reference” on page 8-1.

In the example directory of SCIOPTA Flash File System delivery you will find example of file system setup processes.

For the “effs_ram” SCIOPTA Flash File System getting started example the following file contains the file system setup:

Files

fs_setup.c	Example file system setup process

File location<installation_folder>\sciopta\<version>\exp\sfs\common\effs_ram\

9.7 File System User Process

The user process contains the user code to access the file system. There are two ways of using the SCIOPTA Flash File System.

9.7.1 HCC Mode

This is the most common way to use the file system. The HCC Safe File System functions are called directly by the user. The HCC Safe File System runs in the context of the user process.

In this mode the HCC Safe Flash File System API must be used. Please consult chapter 6 “HCC Safe Flash File System API” on page 6-1 .

In the example directory of SCIOPTA Flash File System delivery you will find example of file system user processes.

For the “effs_ram” SCIOPTA Flash File System getting started example the following file contains the HCC Mode flash file system user process:

Files

hcc_effstest.c	Example file system process (HCC Mode)

File location<installation_folder>\sciopta\<version>\exp\sfs\common\effs_ram\

9.7.2 SDD Mode

In SDD mode the user accesses the file system by using the SDD (SCIOPTA Device Driver) and the SFS (SCIOPTA F;ash File System) functions. These functions send and receive messages to and from the file system process for accessing the HCC Safe File System.

SDD Mode must be used for systems having the SCIOPTA Memory Management System (SMMS) and a CPU MMU.

In this mode you can use the SCIOPTA message interface (see chapter 7 “Message Interface Reference” on page 7-1) or the SCIOPTA SDD and Files System function interface (8 “SCIOPTA SFFS Function Interface Reference” on page 8-1).

In the example directory of SCIOPTA Flash File System delivery you will find example of file system user processes.

For the “effs_ram” SCIOPTA Flash File System getting started example the following file contains the SDD Mode flash file system user process:

Files

effstest.c	Example file system process (SDD Mode)

File location<installation_folder>\sciopta\<version>\exp\sfs\common\effs_ram\

9.8 Implementing Flash File System Drivers

Please consult chapter 11.21 “Flash File System Drivers” on page 11-35 for information about flash file system drivers.

The driver design achieves a high level of portability while still maintaining excellent performance of the system. The basic device architecture includes a high level driver for each general media type that shares some common properties. This driver handles issues of FAT maintenance, wear-leveling, etc. Below this lies a physical device handler that does the translation between the driver and the physical flash hardware.

Generally only the physical handler needs to be modified when the hardware configuration changes (different chip type, 1/2/4 devices in parallel etc.). There is a range of physical handlers available to make the porting process as simple as possible.

9.9 System Requirements

The SCIOPTA Flash File System is designed to be as open and portable as possible. No assumptions are made about the functionality or behaviour of the underlying operating system. For the SCIOPTA Flash File System to work at its best, certain porting work should be done as outlined below. This is a very straightforward task for an experienced engineer.

9.9.1 Timeouts

Flash devices are normally controlled by hardware control signals. As a result there is no explicit need for any time-out to control exception conditions. However, some operations on flash devices are relatively slow and it is often worthwhile to schedule other operations while waiting for them to complete (e.g., a NOR flash erase typically takes 2 seconds and a NAND flash erase takes 2 milliseconds).

For NOR flash in the 29lvxxx.c sample driver the DataPoll function is used to check for the completion of operations. This routine could be modified to force scheduling of the system or to use the event generation mechanism of the host system so that other operations can be performed while waiting.

For NAND flash in the K9F2816X0C sample driver the nandwaitrb function is used to check for the completion of operations. This routine could be modified to force scheduling of the system or to use the event generation mechanism of the host system so that other operations can be performed while waiting.

9.9.2 Real Time Clock

Whenever a file is created or closed (for writing) the system sets a date/time field associated with the file. To do this the following functions in port_s.c are called:

```
unsigned short fs_gettime(void)
unsigned short fs_getdate(void)
```

These functions by default enter zeroes into these fields. When porting to a system with a real-time clock, the functions should be modified to set the correct current time and date from your system. A recommended format for how this can be done is given by the following shift and mask definitions in the fsm.h file:

```
/* definitions for time */

#define FS_CTIME_SEC_SH      0
#define FS_CTIME_SEC_MASK    0x001f/* 0-30 in 2seconds */
#define FS_CTIME_MIN_SH      5
#define FS_CTIME_MIN_MASK    0x07e0/* 0-59 minutes */
#define FS_CTIME_HOUR_SH     11
#define FS_CTIME_HOUR_MASK   0xf800/* 0-23 hours */

/* definition for date */

#define FS_CDATE_DAY_SH      0
#define FS_CDATE_DAY_MASK    0x001f/* 0-31 days */
#define FS_CDATE_MONTH_SH    5
#define FS_CDATE_MONTH_MASK  0x01e0/* 1-12 months */
#define FS_CDATE_YEAR_SH     9
#define FS_CDATE_YEAR_MASK   0xfe00/* 0-119 (year 1980+value) */
```

Please Note:

Although this format is recommended, the developer may use these two 16 bit fields as they require - they will simply be updated according to the developer’s replacement function each time a file is created or closed.

9.9.3 Memory Allocation

There are some larger buffers required by the file system to handle FATs in RAM and also to buffer write processes.

There is a call to each driver to get the specific size of memory required for that drive. It is then up to the user to allocate this memory from the system.

Buffer sizes depend on the particular chips being used and their configurations. For further information see description of **f_mountdrive** function and the **fs_getmem_xxx** functions in the relevant driver sections.

9.9.4 Stack Requirements

The file system functions are always called in the context of the calling thread or task. Naturally the functions require stack space and the developer should allow for this in applications that call file system functions. Typically calls to the file system will use <2KBytes of stack.

9.9.5 Memcpy and Memset

The system includes memcpy and memset functions, which are provided as simple byte copy routines. To get best performance from the target platform these routines should be replaced with routines developed specifically for the target system. As in all embedded systems the copy routines are time consuming but optimized versions can yield excellent performance benefits.

9.10 Specific Features

9.10.1 Power Fail Safety

The flash file system is entirely power fail safe. The system may be stopped at any point and restarted and no data will be lost; the previous completed state of the file system will be restored.

When a file is closed its data are automatically flushed from the file system. Until this close takes place the file is preserved. The user may also use the f_flush command to write the current state of the file to the medium and thus update its failsafe state.

9.10.2 Long File Names

The file system supports file names of almost unlimited length. File name handling is efficient – it is built from a chain of small fragments taken from the descriptor block. If a file name is longer than FS_MAXDENAME (default 13) an additional FS_MAXLFN (default 11) byte block is allocated to store the longer name. These additional blocks are added by the file system automatically.

In **fsm.h** there is a FS_MAXLNAME define which sets the maximum allowed name length. By default this is set to FS_MAXDENAME+4*FS_MAXLFN (57 bytes). The developer may increase (or decrease) this by multiples of FS_MAXLFN bytes by changing the multiplier of FS_MAXLFN in the FS_MAXLNAME definition. This sets the number of these structures that may be used for a single name.

Long file names use memory from the descriptor blocks in the file system. The system uses an efficient algorithm for allocating additional blocks in units of FS_MAXLFN. However, the use of long file names reduces the number of file and directory entries that can be stored.

9.10.3 Multiple Volumes

The file system supports multiple volumes. Each volume must have its own driver routine, which normally has its own physical handler (except for the RAM drive).

The maximum number of volumes allowed by your system should be set in the FS_MAXVOLUME definition in **udefs.h**. Set this value to the maximum volume number used. If only a RAM drive is used, set the value to 1; if you use a RAM drive and NOR flash, then set this value to 2, etc. Volume letters are assigned by passing a parameter in the f_mountdrive function.

9.10.4 Multiple Open Files in a Volume

The file system allows multiple files to be opened simultaneously on a volume or on different volumes. Within each driver (ramdrv_s.c, flashdrv.c, nflshdrv.c) there is a MAXFILE definition that determines the number of files that the file system allows to be opened simultaneously on that volume at any particular time.

For each file that may be allowed to be opened simultaneously an array must be allocated that contains a sector size buffer. Thus, increasing MAXFILE for a particular volume increases the RAM required by the system.

9.10.5 Case Sensitive File Names

By default the file system uses case insensitive names. To enable case sensitive names set FS_EFFS_CASE_SENSITIVE in **udefs.h** to 1.

9.10.6 Separator Character

You can define the file separator character using the FS_SEPARATORCHAR definition in **udefs.h**. By default this is a backslash.

9.10.7 Unicode

To enable the use of API functions for Unicode, in the **udefs_s.h** file comment "#define HCC_UNICODE" in. Consult the API sections for the functions with prefixed w's for usage information; an example is **f_wopen** instead of f_open.

9.10.8 Static Wear

Flash devices are usually manufactured to a specification which includes a guaranteed number of write-erase cycles that can be performed on each block before it may develop a fault. Because of this it is important to use the blocks in a device "evenly" if the device is to be used for its maximum lifetime.

The file system uses a process called dynamic wear to allocate the least use blocks from those available. However, in systems where there are large areas of static data (e.g., the executable binary for the system) then the areas where this is stored may be written only once leaving a relatively small section of the device to handle the much more heavily used files.

For this reason a process called static wear is introduced. When the fs_staticwear function is called it searches for blocks that have been used much less than the most used blocks in the system and if this difference is greater than a defined threshold (FS_STATIC_DISTANCE), then these two blocks will be exchanged in the system.

To use the static functionality the files fstaticw.c and fstaticw.h must be included in your project. In the header file two defines should be set:

> FS_STATIC_DISTANCE - this specifies the minimum difference between a heavily used block and a lightly used block before a static swap is allowed. This number should not be so small that it causes unnecessary swapping. A reasonable figure to choose is between 1% and 10% of the guaranteed erase/write cycles of the target chip.

FS_STATIC_PERIOD - this specifies how often this function will actually attempt to do a wear. This may be used in systems where fs_staticwear is called very frequently to reduce the number of times that the function will be executed. This reduces unnecessary checking of the system. If you always know that the system is going to be idle when fs_staticwear is called then this may be set to 1 so that it is always executed - for instance if you just do a few calls to fs_staticwear at startup. If it is called at every available opportunity then you may want to execute this less frequently.

When the static wear function is executing, the file system is not accessible. The length of time the static wear function takes depends on the specification of the target chips being used and in particular the time required to erase a block and the time required to copy one block to another.

For static wear to function an additional driver function, BlockCopy, must also be provided. See driver sections for information as to how to implement this function. It is important to provide a highly optimized version of this, preferably using any special copy functions specific to the target chip, to achieve the best system performance and least disruption.

9.10.8.1 Do I need static wear?

In many cases this is an unnecessary overhead. You can assess its importance by looking at how your product is to be used and considering the specification of your target devices. Many devices have up to 1 million guaranteed erase/write cycles per block and in many applications this number will not be reached in the lifetime of the product.

9.10.8.2 When should I do static wear?

Because wear involves swapping blocks in the file system all access is excluded for the duration of the process. Thus, if there are time-critical features in your flash access applications then it is preferable to do static wear during idle moments. One useful time is during system boot where several static wears could be done without having a major impact on the boot time of the system.

9.10.9 Free Block Allocation

The system includes two different algorithms for allocating blocks in the file system. One finds the first block it finds with an available sector; the second (default) allocates the block with the most free sectors.

The algorithm used can be selected with the FSF_MOST_FREE_ALLOC define in **fsf.h**. The default setting is recommended.

9.11 Error Hook

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

There are two error hooks available:

A) Module Error Hook

B) Global Error Hook

If the kernel detect an error condition it will first call the module error hook and if it is not available call the global error hook. Error hooks are normal error handling functions and must be written by the user. Depending on the type of error (fatal or non-fatal) it will not be possible to return from an error hook.

If there are no error hooks present the kernel will enter an infinite loop (at label **SC_ERROR**) and all interrupts are disabled.

Please consult the chapter “Application Programming>Error Hook” of the SCIOPTA ARM - Kernel, User’s Guide for more information about Error Information, Error Hook Registering, Error Hook Declaration Syntax and Error Hook Example.

In the example directory of SCIOPTA delivery you will find an example of an error hook.

File

error.c	Error hook example.

File location: <installation_folder>\sciopta\<version>\exp\common\

10 System and Application Configuration

10.1 Introduction

Besides the kernel configuration described in chapter 12 "Kernel Configuration" on page 12-1 which defines system characteristics, modules, processes and message pools, you need to setup and initialize your board and your application.

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

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

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

10.2 System Start

10.2.1 Start Sequence

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

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

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

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

10.2.2 Reset Hook

Description

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

The reset hook must be written by the user.

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

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

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

Please consult also chapter 11 “Board Support Packages” on page 11-1 for more information.

Source File

resethook.S	Board setup.

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

Syntax

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

Parameter	Description
none	

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

10.2.3 C Startup

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

Source File

cstartup.S	C startup assembler source for GNU GCC

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

10.2.4 Start Hook

Description

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

The start hook is mainly used to do chip, board and system initialization. As the C environment is initialized it can be written in C. The start hook would also be the right place to include the registration of the system error hook and other kernel hooks.

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

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

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

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

Prototype

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

10.2.5 INIT Process

The INIT process is the first process in a module. Each module has at least one process and this is the init process. At module start the init process gets automatically the highest priority (0). After the init process has done some important work it will change its priority to the lowest level (32) and enter an endless loop.

Priority 32 is only allowed for the init process. All other processes are using priority 0 - 31. The INIT process acts therefore also as idle process which will run when all other processes of a module are in the waiting state.

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

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

10.2.6 Module Hooks

10.2.6.1 System Module Hook

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

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

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

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\effs_ram\<board>\

10.2.6.2 User Modules Hooks

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

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

10.3 Flash File System Setup and Configuration

You need to write a process **SCP_flashFs** (declared with the process name **SCP_flash**). The only function of this process is to call the flash file system function **effs_process** including the configuration parameter.

```
void effs_process (
             const char      *fsName
);
```

Parameter	Description
fsname	Name of file system to register at file system manager.

Return Value	Condition
None	

Example

```
#define FS_NAME "effs0"

SC_PROCESS(SCP_flashFs)
{
  sc_poolid_t pl;

  pl = sc_poolIdGet("fs_pool");
  if ( pl != SC_ILLEGAL_POOLID ){
    sc_poolDefault(pl);
  }

  effs_process(FS_NAME);
}
```

10.4 HCC Safe File System Configuration

10.4.1 Long File Names

The file system supports file names of almost unlimited length. File name handling is efficient – it is built from a chain of small fragments taken from the descriptor block. If a file name is longer than FS_MAXDENAME (default 13) an additional FS_MAXLFN (default 11) byte block is allocated to store the longer name. These additional blocks are added by the file system automatically.

In **fsm.h** there is a FS_MAXLNAME define which sets the maximum allowed name length. By default this is set to FS_MAXDENAME+4*FS_MAXLFN (57 bytes). The developer may increase (or decrease) this by multiples of FS_MAXLFN bytes by changing the multiplier of FS_MAXLFN in the FS_MAXLNAME definition. This sets the number of these structures that may be used for a single name.

Long file names use memory from the descriptor blocks in the file system. The system uses an efficient algorithm for allocating additional blocks in units of FS_MAXLFN. However, the use of long file names reduces the number of file and directory entries that can be stored.

Include Files

fsm.h	Intermediate file system header.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\common\

10.4.2 Multiple Volumes

The file system supports multiple volumes. Each volume must have its own driver routine, which normally has its own physical handler (except for the RAM drive).

The maximum number of volumes allowed by your system should be set in the FS_MAXVOLUME definition in **udefs.h**. Set this value to the maximum volume number used. If only a RAM drive is used, set the value to 1; if you use a RAM drive and NOR flash, then set this value to 2, etc. Volume letters are assigned by passing a parameter in the f_mountdrive function.

Include Files

udefs.h	HCC Safe File System definitions.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\

10.4.3 Multiple Open Files in a Volume

The file system allows multiple files to be opened simultaneously on a volume or on different volumes. Within each driver (ramdrv_s.c, flashdrv.c, nflshdrv.c) there is a MAXFILE definition that determines the number of files that the file system allows to be opened simultaneously on that volume at any particular time.

For each file that may be allowed to be opened simultaneously an array must be allocated that contains a sector size buffer. Thus, increasing MAXFILES for a particular volume increases the RAM required by the system.

Please consult also chapter 11.21 “Flash File System Drivers” on page 11-35.

10.4.4 Case Sensitive File Names

By default the file system uses case insensitive names. To enable case sensitive names set FS_EFFS_CASE_SENSITIVE in **udefs.h** to 1.

Include Files

udefs.h	HCC Safe File System definitions.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\

10.4.5 Separator Character

You can define the file separator character using the FS_SEPARATORCHAR definition in **udefs.h**. By default this is a backslash.

Include Files

udefs.h	HCC Safe File System definitions.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\

10.4.6 Unicode

To enable the use of API functions for Unicode, in the udefs_s.h file comment "#define HCC_UNICODE" in. Consult the API sections for the functions with prefixed w's for usage information; an example is f_wopen() instead of f_open().

Include Files

udefs_s.h	HCC Safe File System definitions.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\common\

10.4.7 Static Wear

Please consult chapter 9.10.8 “Static Wear” on page 9-9 for information about static wear.

A process called static wear may be used. When the fs_staticwear function is called it searches for blocks that have been used much less than the most used blocks in the system and if this difference is greater than a defined threshold (FS_STATIC_DISTANCE), then these two blocks will be exchanged in the system.

To use the static functionality the files fstaticw.c and fstaticw.h must be included in your project. In the header file two defines should be set:

FS_STATIC_DISTANCE - this specifies the minimum difference between a heavily used block and a lightly used block before a static swap is allowed. This number should not be so small that it causes unnecessary swapping. A reasonable figure to choose is between 1% and 10% of the guaranteed erase/write cycles of the target chip.

FS_STATIC_PERIOD - this specifies how often this function will actually attempt to do a wear. This may be used in systems where fs_staticwear is called very frequently to reduce the number of times that the function will be executed. This reduces unnecessary checking of the system. If you always know that the system is going to be idle when fs_staticwear is called then this may be set to 1 so that it is always executed - for instance if you just do a few calls to fs_staticwear at startup. If it is called at every available opportunity then you may want to execute this less frequently.

When the static wear function is executing, the file system is not accessible. The length of time the static wear function takes depends on the specification of the target chips being used and in particular the time required to erase a block and the time required to copy one block to another.

For static wear to function an additional driver function, BlockCopy, must also be provided. See driver sections for information as to how to implement this function. It is important to provide a highly optimized version of this, preferably using any special copy functions specific to the target chip, to achieve the best system performance and least disruption.

10.4.8 Free Block Allocation

The system includes two different algorithms for allocating blocks in the file system. One finds the first block it finds with an available sector; the second (default) allocates the block with the most free sectors.

The algorithm used can be selected with the FSF_MOST_FREE_ALLOC define in **fsf.h**. The default setting is recommended.

Include Files

fsf.h	Standard file system API header.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\common\

11 Board Support Packages

11.1 Introduction

Only the device drivers which are needed in a typical SCIOPTA ARM - Flash File System are listed here.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for all other information about SCIOPTA BSPs. For other SCIOPTA products please consult the BSP chapter of the respective manual.

11.2 General System Functions and Drivers

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

For the SCIOPTA Flash File System we will use HCC Safe Flash File System drivers. Please consult chapter 11.21 "Flash File System Drivers" on page 11-35 for more information about the HCC Safe Flash File System drivers.

Project Files

winIDEA_gnu.ind	iSYSTEM winIDEA indirection file for GNU GCC

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

11.3 ARM Family System Functions and Drivers

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

Only the basic drivers are listed here which are needed in a typical flash file system application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of kernel drivers.

Project Files

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

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

Source Files

cstartup.S	C startup assembler source for GNU GCC.
exception.S	Exception handler for GNU GCC.

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

cstartup.s	C startup assembler source for ARM RealView.
exception.s	Exception handler for ARM RealView.

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

exception.s79	Exception handler for IAR Embedded Workbench.

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

11.4 AT91SAM7 System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with AT91SAM7 based processors.

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a complete list of kernel drivers.

11.4.1 General AT91SAM7 Functions and Definitions

Include Files

at91sam7.h	AT91SAM7xxx definitions.
at91sam7a3.h, at91sam7a3_inc.h	AT91SAM7A3 definitions.
at91sam7s64.h, at91sam7s64_inc.h	AT91SAM7S64 definitions.
at91sam7s.h, at91sam7s_inc.h	AT91SAM7Sxx definitions.
at91sam7se512.h, at91sam7se512_inc.h	AT91SAM7SE512 definitions.
at91sam7x256.h, at91sam7x256_inc.h	AT91SAM7X256 definitions.
at91sam7x.h, at91sam7x_inc.h	AT91SAM7X definitions.
sys_irq.h	SysIRQ definitions.

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

Source Files

irq_handler.S	Interrupt handler for AT91SAM7 and GNU GCC.

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

irq_handler.s	Interrupt handler for AT91SAM7 and ARM RealView.

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

irq_handler.s79	Interrupt handler for AT91SAM7 and IAR EW.

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

11.4.2 AT91SAM7 System Tick Driver

11.4.2.1 AT91SAM7 System Tick Configuration

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the system tick driver configuration.

11.4.2.2 AT91SAM7 System Tick Interrupt Process

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the system tick interrupt process.

Source Files

systick.c	System timer setup.

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

11.4.3 AT91SAM7 UART Functions

This is not a full featured serial driver. Just some basic UART routines for the AT91SAM7 controllers are supplied to be used for system logging and printf debugging.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for detailed descriptions of serial drivers.

Include Files

simple_uart.h	Simple UART routines.

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

Source Files

simple_uart.c	Simple polling uart function for printf debugging or logging.

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

11.5 AT91SAM9 System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with AT91SAM9 based processors.

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of kernel drivers.

11.5.1 General AT91SAM9 Functions and Definitions

Include Files

at91sam9.h, at91sam9_inc.h	Common header for all sam9.
at91sam9260.h, at91sam9260_inc.h	AT91SAM9260 definitions.
at91sam9261.h, at91sam9261_inc.h	AT91SAM9261 definitions.
at91sam9261_inc.inc	AT91SAM9261 definitions.
lib_at91sam9260.h	AT91SAM9260 inlined functions.
lib_at91sam9261.h	AT91SAM9261 inlined functions.
sys_irq.h	SysIRQ definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\include\

Source Files

irq_handler.S	Interrupt handler for AT91SAM9 and GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\src\gnu\

irq_handler.s	Interrupt handler for AT91SAM9 and ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\src\arm\

irq_handler.s79	Interrupt handler for AT91SAM9 and IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\src\iar\

11.5.2 AT91SAM9 System Tick Driver

11.5.2.1 AT91SAM9 System Tick Configuration

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the system tick driver configuration.

11.5.2.2 AT91SAM9 System Tick Interrupt Process

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the system tick interrupt process.

Source Files

systick.c	System timer setup.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\src\

11.5.3 AT91SAM9 UART Functions

This is not a full featured serial driver. Just some basic UART routines for the AT91SAM9 controllers are supplied to be used for system logging and printf debugging.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for detailed descriptions of serial drivers.

Include Files

simple_uart.h	Simple UART routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\include\

Source Files

simple_uart.c	Simple polling uart function for printf debugging or logging.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\src\

11.6 LPC21xx System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with LPC21xx based processors.

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of kernel drivers.

11.6.1 General LPC21xx Functions and Definitions

Include Files

lpc21xx.h	LPC2xxx defines.
lpc21xx.iar	LPC2xxx defines.
lpc21xx.inc	LPC2xxx defines.

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

Source Files

eintx_irq.S	Interrupt wrapper for eint0..3 for LPC21xx and GNU GCC.
irq_handler.S	Interrupt handler for LPC21xx and GNU GCC.
spi_irq.S	Interrupt wrapper for spi0 and spi1 for LPC21xx and GNU GCC.
systick_irq.S	Interrupt wrapper for systick for LPC21xx and GNU GCC.

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

eintx_irq.s	Interrupt wrapper for eint0..3 for LPC21xx and ARM RealView.
irq_handler.s	Interrupt handler for LPC21xx and ARM RealView.
spi_irq.s	Interrupt wrapper for spi0 and spi1 for LPC21xx and ARM RealView.
systick_irq.s	Interrupt wrapper for systick for LPC21xx and ARM RealView.

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

eintx_irq.s79	Interrupt wrapper for eint0..3 for LPC21xx and IAR EW.
irq_handler.s79	Interrupt handler for LPC21xx and IAR EW.
spi_irq.s79	Interrupt wrapper for spi0 and spi1 for LPC21xx and IAR EW.
systick_irq.s79	Interrupt wrapper for systick for LPC21xx and IAR EW.

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

11.6.2 LPC21xx System Tick Driver

11.6.2.1 LPC21xx System Tick Configuration

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick driver configuration.

11.6.2.2 LPC21xx System Tick Interrupt Process

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick interrupt process.

Source Files

systick.c	System timer setup.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpx21xx\src\

11.6.3 LPC21xx UART Functions

This is not a full featured serial driver. Just some basic UART routines for the LPC21xx controllers are supplied to be used for system logging and printf debugging.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for detailed descriptions of serial drivers.

Include Files

simple_uart.h	Simple UART routines.

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

Source Files

simple_uart.c	Simple polling uart function for printf debugging or logging.

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

11.7 LPC23xx and LPC24xx System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with LPC23xx and LPC24xx based processors.

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of kernel drivers.

11.7.1 General LPC23xx and LPC24xx Functions and Definitions

Include Files

lpc23xx.h	HW register for LPC23xx/LPC24xx.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\include\

Source Files

irq_handler.S	Interrupt handler for LPC23xx/LPC24xx and GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\src\gnu\

irq_handler.s	Interrupt handler for LPC21xx/LPC24xx and ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\src\arm\

irq_handler.s79	Interrupt handler for LPC21xx/LPC24xx and IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\src\iar\

11.7.2 LPC23xx and LPC24xx System Tick Driver

11.7.2.1 LPC23xx and LPC24xx System Tick Configuration

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick driver configuration.

11.7.2.2 LPC23xx and LPC24xx System Tick Interrupt Process

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick interrupt process.

Source Files

systick.c	System timer setup.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpx24xx_lpc23xx\src\

11.7.3 LPC23xx and LPC24xx UART Functions

This is not a full featured serial driver. Just some basic UART routines for the LPC23xx and LPC24xx controllers are supplied to be used for system logging and printf debugging.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for detailed descriptions of serial drivers.

Include Files

simple_uart.h	Simple UART routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\include\

Source Files

simple_uart.c	Simple polling uart function for printf debugging or logging.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\src\

11.8 STR7 System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with STR7 based processors.

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a complete list of kernel drivers.

11.8.1 General STR7 Functions and Definitions

Include Files

gpio.h	GBIO functions.
rccu.h	RCCU functions and definitions.
str71x.h	STR71x memory map.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\include\

Source Files

irq_handler.S	Interrupt handler for STR7 and GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\src\gnu\

irq_handler.s79	Interrupt handler for STR7 and IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\src\iar\

11.8.2 STR7 System Tick Driver

11.8.2.1 STR7 System Tick Configuration

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick driver configuration.

11.8.2.2 STR7 System Tick Interrupt Process

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick interrupt process.

Source Files

systick.c	System timer setup.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\src\

11.8.3 STR7 UART Functions

This is not a full featured serial driver. Just some basic UART routines for the STR7 controllers are supplied to be used for system logging and printf debugging.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for detailed descriptions of serial drivers.

Include Files

simple_uart.h	Simple UART routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\include\

Source Files

simple_uart.c	Simple polling uart function for printf debugging or logging.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\src\

11.9 STR9 System Functions and Drivers

System functions which do not depend on a specific board and are common for all boards with STR9 based processors.

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a complete list of kernel drivers.

11.9.1 General STR9 Functions and Definitions

Include Files

str91x.h	STR91x memory map.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\include\

91x_*.h	STR91x definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\include\str91x\

Source Files

irq_handler.S	Interrupt handler for STR9 and GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\src\gnu\

irq_handler.s79	Interrupt handler for STR9 and IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\src\iar\

11.9.2 STR9 System Tick Driver

11.9.2.1 STR9 System Tick Configuration

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick driver configuration.

11.9.2.2 STR9 System Tick Interrupt Process

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the system tick interrupt process.

Source Files

systick.c	System timer setup.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\src\

11.9.3 STR9 UART Functions

This is not a full featured serial driver. Just some basic UART routines for the STR9 controllers are supplied to be used for system logging and printf debugging.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for detailed descriptions of serial drivers.

Include Files

simple_uart.h	Simple UART routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\include\

Source Files

simple_uart.c	Simple polling uart function for printf debugging or logging.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\src\

11.10 Atmel AT91SAM7A3-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.10.1 Atmel AT91SAM7A3-EK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Atmel AT91SAM7A3-EK board.

11.10.2 General Board Functions and Definitions

Project Files

at91sam7a3-ek.ld	GCC linker script example.
at91sam7a3-ek.sct	ARM RealView linker script example.
at91sam7a3-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\src\iar\

11.10.3 LED Driver

Simple functions to access the Atmel AT91SAM7A3-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM7A3-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM7A3-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7a3-ek\src\

11.11 Atmel AT91SAM7SE-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.11.1 Atmel AT91SAM7SE-EK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Atmel AT91SAM7SE-EK board.

11.11.2 General Board Functions and Definitions

Project Files

at91sam7se-ek.ld	GCC linker script example.
at91sam7se-ek.sct	ARM RealView linker script example.
at91sam7se-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\src\iar\

11.11.3 LED Driver

Simple functions to access the Atmel AT91SAM7SE-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM7SE-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM7SE-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7se-ek\src\

11.12 Atmel AT91SAM7S-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.12.1 Atmel AT91SAM7S-EK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Atmel AT91SAM7S-EK board.

11.12.2 General Board Functions and Definitions

Project Files

at91sam7s-ek.ld	GCC linker script example.
at91sam7s-ek.sct	ARM RealView linker script example.
at91sam7s-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\src\iar\

11.12.3 LED Driver

Simple functions to access the Atmel AT91SAM7S-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM7S-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM7S-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7s-ek\src\

11.13 Atmel AT91SAM7X-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.13.1 Atmel AT91SAM7X-EK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Atmel AT91SAM7X-EK board.

11.13.2 General Board Functions and Definitions

Project Files

at91sam7x-ek.ld	GCC linker script example.
at91sam7x-ek.sct	ARM RealView linker script example.
at91sam7x-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\src\iar\

11.13.3 LED Driver

Simple functions to access the Atmel AT91SAM7X-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM7X-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM7X-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam7\at91sam7x-ek\src\

11.14 Atmel AT91SAM9260-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a complete list of board drivers.

11.14.1 Atmel AT91SAM9260-EK Board Description

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the Atmel AT91SAM9260-EK board.

11.14.2 General Board Functions and Definitions

Project Files

at91sam9260-ek.ld	GCC linker script example.
at91sam9260-ek.sct	ARM RealView linker script example.
at91sam9260-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\include\

Include Files

config.h	Board configuration definitions.
config.inc	Board configuration definitions.
spi_cnf.h	SPI configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\include\

Source Files

sdram.c	SDRAM initialization.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\src\

pre_reset.S	Reset initialisation done after loading by romboot for GNU GCC.
resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\src\gnu\

pre_reset.s	Reset initialisation done after loading by romboot for ARM RealView.
resethook.s	Board setup for RAM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\src\iar\

11.14.3 LED Driver

Simple functions to access the Atmel AT91SAM9260-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM9260-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM9260-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9260-ek\src\

11.15 Atmel AT91SAM9261-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.15.1 Atmel AT91SAM9261-EK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Atmel AT91SAM9261-EK board.

11.15.2 General Board Functions and Definitions

Project Files

at91sam9261-ek.ld	GCC linker script example.
at91sam9261-ek.sct	ARM RealView linker script example.
at91sam9261-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\include\

Include Files

config.h	Board configuration definitions.
config.inc	Board configuration definitions.
spi_cnf.h	SPI configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\include\

Source Files

sdram.c	SDRAM initialization.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\src\

pre_reset.S	Reset initialisation done after loading by romboot for GNU GCC.
resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\src\gnu\

pre_reset.s	Reset initialisation done after loading by romboot for ARM RealView.
resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\src\iar\

11.15.3 LED Driver

Simple functions to access the Atmel AT91SAM9261-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM9261-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM9261-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9261-ek\src\

11.16 Atmel AT91SAM9263-EK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.16.1 Atmel AT91SAM9263-EK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Atmel AT91SAM9263-EK board.

11.16.2 General Board Functions and Definitions

Project Files

at91sam9263-ek.ld	GCC linker script example.
at91sam9263-ek.sct	ARM RealView linker script example.
at91sam9263-ek.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\include\

Include Files

config.h	Board configuration definitions.
config.inc	Board configuration definitions.
spi_cnf.h	SPI configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\include\

Source Files

sdram.c	SDRAM initialization.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\src\

pre_reset.S	Reset initialisation done after loading by romboot for GNU GCC.
resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\src\gnu\

pre_reset.s	Reset initialisation done after loading by romboot for ARM RealView.
resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\src\iar\

11.16.3 LED Driver

Simple functions to access the Atmel AT91SAM9263-EK board LEDs.

Include Files

led.h	Defines for the Atmel AT91SAM9263-EK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\include\

Source Files

led.c	Routines to access the LEDs on the Atmel AT91SAM9263-EK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\at91sam9\at91sam9263-ek\src\

11.17 Phytec phyCORE-LPC2294 Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.17.1 Phytec phyCORE-LPC2294 Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the Phytec phyCORE-LPC2294 board.

11.17.2 General Board Functions and Definitions

Project Files

phyCore2294.ld	GCC linker script example.
phyCore2294.sct	ARM RealView linker script example.
phyCore2294.xcl	IAR EW linker script example.

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

Include Files

config.h	Board configuration definitions.

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

Source Files

resethook.S	Board setup for GNU GCC.

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

resethook.s	Board setup for ARM RealView.

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

resethook.s79	Board setup for IAR EW.

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

11.17.3 LED Driver

Simple functions to access the Phytec phyCORE-LPC2294 board LEDs.

Include Files

led.h	Defines for the Phytec phyCORE-LPC2294 board's LED routines.

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

Source Files

led.c	Routines to access the LEDs on the Phytec phyCORE-LPC2294 board.

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

11.18 Embedded Artists LPC2468 OEM Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a complete list of board drivers.

11.18.1 Embedded Artists LPC2468 OEM Board Description

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the Embedded Artists LPC2468 OEM board.

11.18.2 General Board Functions and Definitions

Project Files

EA_LPC2468_16_OEM.ld	GCC linker script example.
EA_LPC2468_16_OEM.sct	ARM RealView linker script example.
EA_LPC2468_16_OEM.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\EA_LPC2468_16_OEM\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\EA_LPC2468_16_OEM\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\EA_LPC2468_16_OEM\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\EA_LPC2468_16_OEM\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\lpc24xx_lpc23xx\EA_LPC2468_16_OEM\src\iar\

11.19 STR711-SK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a complete list of board drivers.

11.19.1 STR711-SK Board Description

Please consult the chapter "Board Support Packages" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the STR711-SK board.

11.19.2 General Board Functions and Definitions

Project Files

str711-sk.ld	GCC linker script example.
str711-sk.sct	ARM RealView linker script example.
str711-sk.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\src\iar\

11.19.3 LED Driver

Simple functions to access the STR711-SK board LEDs.

Include Files

led.h	Defines for the STR711-SK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\include\

Source Files

led.c	Routines to access the LEDs on the STR711-SK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\str7\str711-sk\src\

11.20 STR912-SK Board

Only the basic drivers are listed here which are needed in a typical SCIOPTA Flash File System application.

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a complete list of board drivers.

11.20.1 STR912-SK Board Description

Please consult the chapter “Board Support Packages” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the STR912-SK board.

11.20.2 General Board Functions and Definitions

Project Files

str912-sk.ld	GCC linker script example.
str912-sk.sct	ARM RealView linker script example.
str912-sk.xcl	IAR EW linker script example.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\include\

Include Files

config.h	Board configuration definitions.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\include\

Source Files

resethook.S	Board setup for GNU GCC.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\src\gnu\

resethook.s	Board setup for ARM RealView.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\src\arm\

resethook.s79	Board setup for IAR EW.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\src\iar\

11.20.3 LED Driver

Simple functions to access the STR912-SK board LEDs.

Include Files

led.h	Defines for the STR912-SK board's LED routines.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\include\

Source Files

led.c	Routines to access the LEDs on the STR912-SK board.

File location: <install_folder>\sciopta\<version>\bsp\arm\str9\str912-sk\src\

11.21 Flash File System Drivers

In the SCIOPTA Flash File System the HCC Safe Flash File System driver can be directly used.

They can be found at: File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\

The following diagram illustrates the structure of the HCC Flash File System device driver.

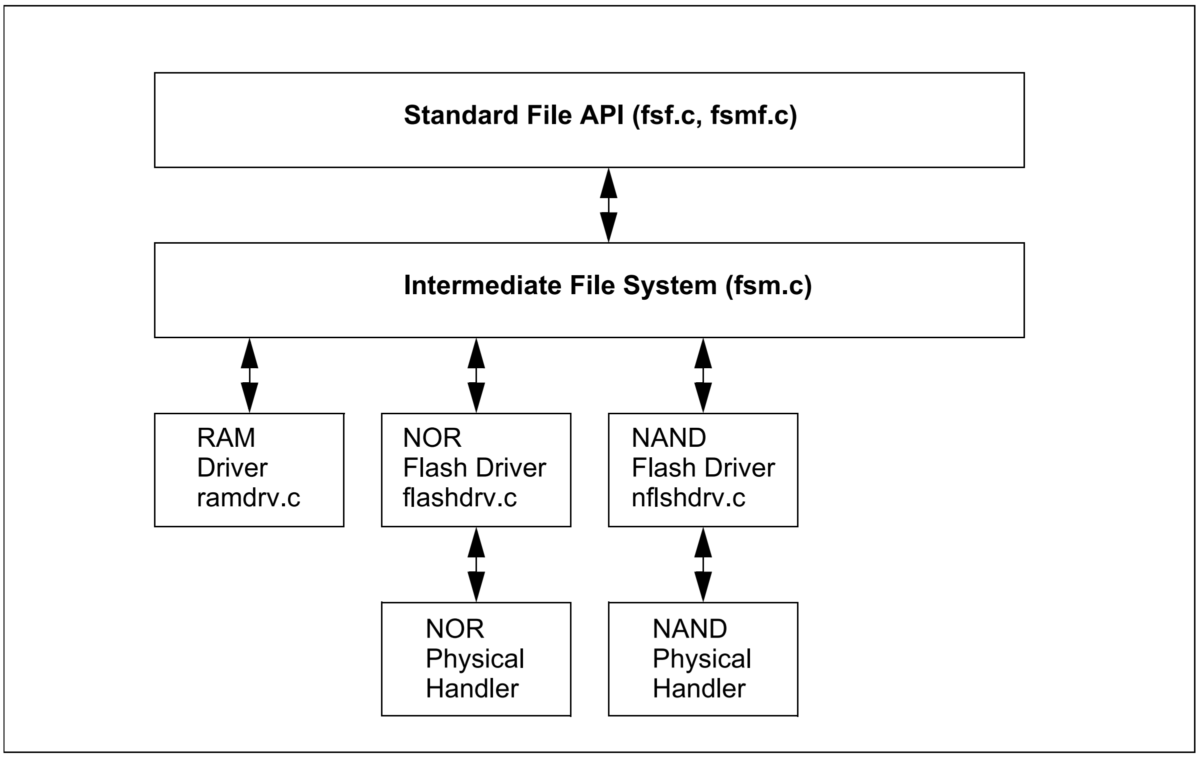

Standard File API (fsf.c, fsmf.c)
Intermediate File System (fsm.c)
RAM Driver ramdrv.c
NOR Flash Driver flashdrv.c
NAND Flash Driver nflshdrv.c
NOR Physical Handler
NAND Physical Handler

Figure 11-1: HCC Safe Flash File System Device Driver Structure

11.21.1 RAM Driver

Include Files

ramdrv_s.h	RAM driver header.
ramdrv_ti.h	Special extension of ramdrive header.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\ram\

Source Files

ramdrv_s.c	RAM driver.
ramdrv_ti.c	Special extension of ramdrive includes. This helps on 64K boundary alignments on DSP-s.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\ram\

11.21.2 NOR Flash Driver

Include Files

flashdrv.h	NOR flash driver header.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nor\

Source Files

flashdrv.c	NOR flash driver.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nor\

11.21.3 NOR Physical Handler for AMD

Include Files

29lv160b.h	Header for NOR flash physical handler for AMD 29lv160b.
29lv640d.h	Header for NOR flash physical handler for AMD 29lv640d.h.
29lv2562m.h	Header for NOR flash physical handler for AMD 29lv2562m.
29lv_8bit.h	Header for NOR flash physical handler for AMD 29lv_8bit.
29pl160cb.h	Header for NOR flash physical handler for AMD 29pl160cb.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nor\amd\

Source Files

29lv160b.c	NOR flash physical handler for AMD 29lv160b.
29lv640d.c	NOR flash physical handler for AMD 29lv640d.h.
29lv2562m.c	NOR flash physical handler for AMD 29lv2562m.
29lv_8bit.c	NOR flash physical handler for AMD 29lv_8bit.
29pl160cb.c	NOR flash physical handler for AMD 29pl160cb.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nor\amd\

11.21.4 NOR Physical Handler for Atmel

Include Files

at49bn6416t.h	Header for NOR flash physical handler for Atmel 49bn6416t.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nor\atmel\

Source Files

at49bn6416t.h	NOR flash physical handler for Atmel 49bn6416t.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nor\atmel\

11.21.5 NOR Physical Handler for Intel

Include Files

28f128j3.h	Header for NOR flash physical handler for Intel 28fl28j3.h.
28f128j3pre.h	Header for NOR flash physical handler for Intel 28fl28j3pre.
28f320j3.h	Header for NOR flash physical handler for Intel 28f320j3.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nor\intel\

Source Files

28f128j3.c	NOR flash physical handler for Intel 28fl28j3.h.
28f128j3pre.c	NOR flash physical handler for Intel 28fl28j3pre.
28f320j3.c	NOR flash physical handler for Intel 28f320j3.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nor\intel\

11.21.6 NAND Flash Driver

The NAND flash driver and the physical handlers are only included if the SCIOPTA Flash File System NAND option was purchased.

Include Files

nflshdrv.h	NAND flash driver header.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nand\

Source Files

nflshdrv.c	NAND flash driver.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nand\

11.21.7 NAND Physical Handler for Micron

Include Files

mt29f2g08aab.h	Header for NAND flash physical handler for Micron mt29f2g08aab.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nand\micron\

Source Files

mt29f2g08aab.c	NAND flash physical handler for Micron mt29f2g08aab.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nand\micron\

11.21.8 NAND Physical Handler for Samsung

Include Files

ecc.h	ECC functions header.
k9f1g08u0m.h	Header for NAND flash physical handler for Samsung k9flg08u0m.
k9f2816x0c.h	Header for NAND flash physical handler for Samsung k9f2816x0c.
k9f2816x0c_16bit.h	Header for NAND flash physical handler for Samsung k9f2816x0c 16 bit.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nand\samsung\

Source Files

ecc.c	ECC functions.
k9f1g08u0m.c	NAND flash physical handler for Samsung k9flg08u0m.
k9f2816x0c.c	NAND flash physical handler for Samsung k9f2816x0c.
k9f2816x0c_16bit.c	NAND flash physical handler for Samsung k9f2816x0c 16 bit.
k9f2816x0c_16bit_no_ecc.c	NAND flash physical handler for Samsung k9f2816x0c 16 bit no ECC.
k9f2816x0c_16bit_old_ecc.c	NAND flash physical handler for Samsung k9f2816x0c 16 bit old ECC.
k9f2816x0c_no_ecc.c	NAND flash physical handler for Samsung k9f2816x0c no ECC.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nand\samsung\

11.21.9 NAND Physical Handler for STMicroelectronics

Include Files

nand128w3a.h	Header for NAND flash physical handler for STMicroelectronics nand128w3a.

File location: <install_folder>\sciopta\<version>\include\hcc\effs\nand\stmicro\

Source Files

nand128w3a.c	NAND flash physical handler for STMicroelectronics nand128w3a.

File location: <install_folder>\sciopta\<version>\sfs\hcc\effs\nand\stmicro\

12 Kernel Configuration

12.1 Introduction

The SCIOPTA ARM kernel needs to be configured before you can build and download your application. In the SCIOPTA configuration utility **SCONF** (sconf.exe) you will define the parameters for SCIOPTA systems such as name of systems, static modules, processes and pools.

For a detailed description of the SCONF configuration utility, please consult the chapter "SCONF Kernel Configuration Utility" of the SCIOPTA ARM - Kernel, User's Guide.

12.2 System

For a SCIOPTA project it would be possible to define more than one system. You could configure a SCIOPTA ARM target system and other SCIOPTA target systems from within the same **SCONF** configuration window. But usually you will define just one target system.

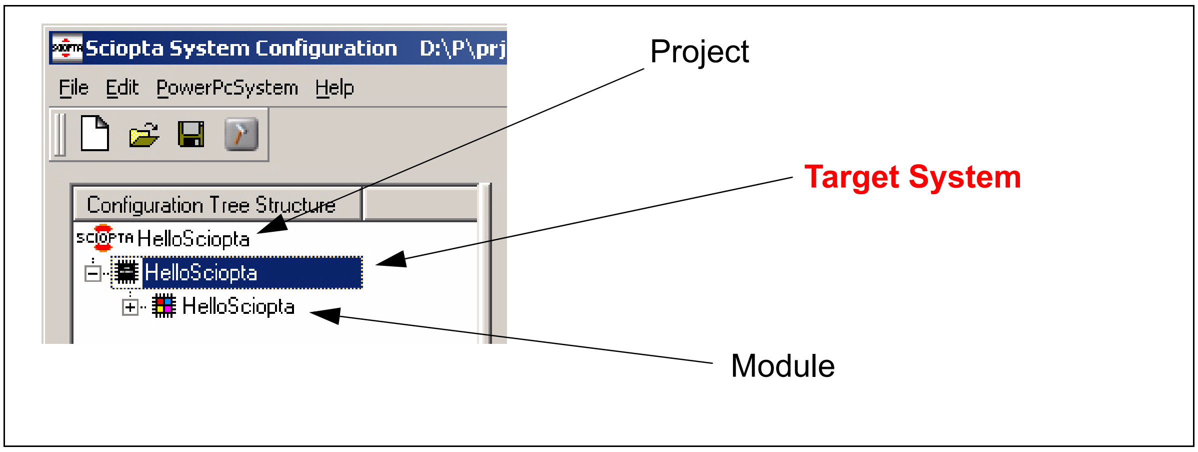

Sciopta System Configuration D:\P\prj
File Edit PowerPcSystem Help
Configuration Tree Structure
HelloSciopta
HelloSciopta
HelloSciopta
Project
Target System
Module

Figure 12-1: SCIOPTA System

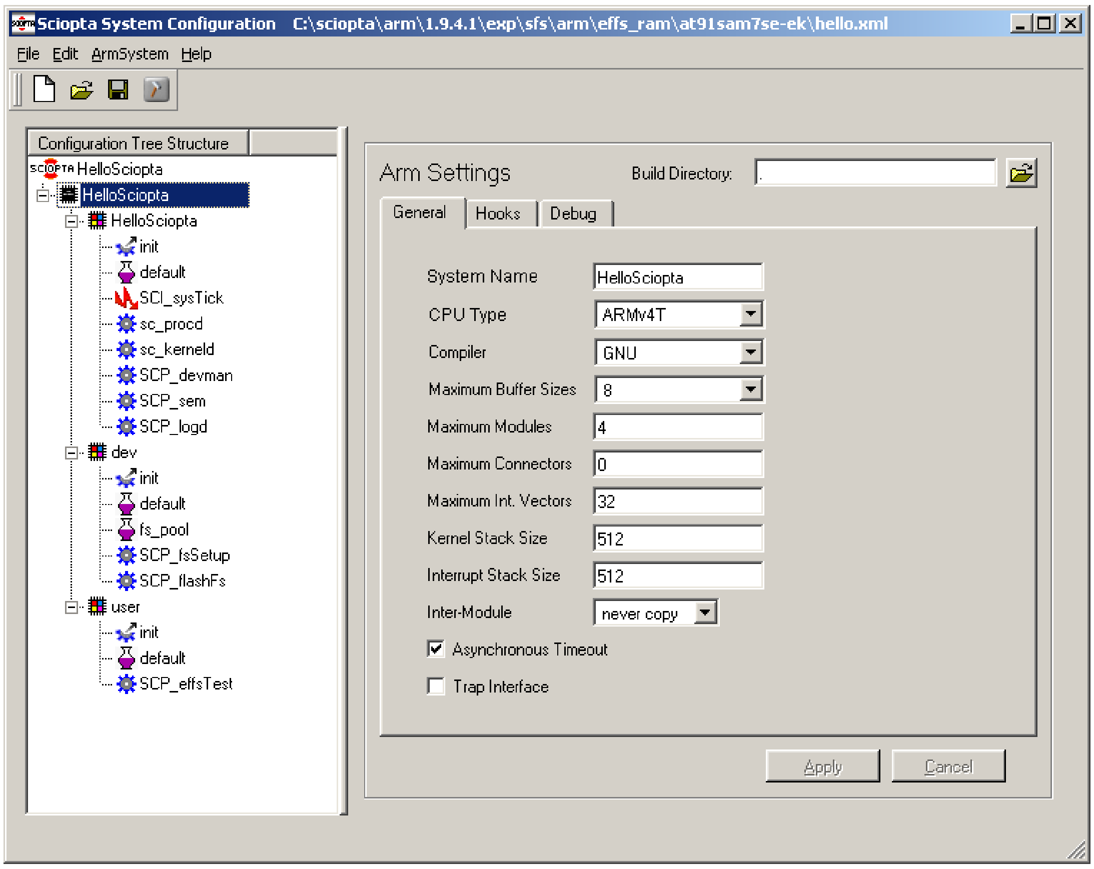
Please consult the chapter "Kernel Configuration" of the SCIOPTA ARM - Kernel, User's Guide for a detailed description of the SCONF target system, module, process and pool configuration and the configuration parameters.

12.3 Modules

Processes can be grouped into modules to improve system structure. A SCIOPTA system must have at least one module, also called module 0 or system module. The system module gets automatically the name of the system.

12.3.1 Modules in a Typical SFFS Application

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



System Name: HelloSciopta
CPU Type: ARMv4T
Compiler: GNU
Maximum Buffer Sizes: 8
Maximum Modules: 4
Maximum Connectors: 0
Maximum Int. Vectors: 32
Kernel Stack Size: 512
Interrupt Stack Size: 512
Inter-Module: never copy
[✓] Asynchronous Timeout
[ ] Trap Interface

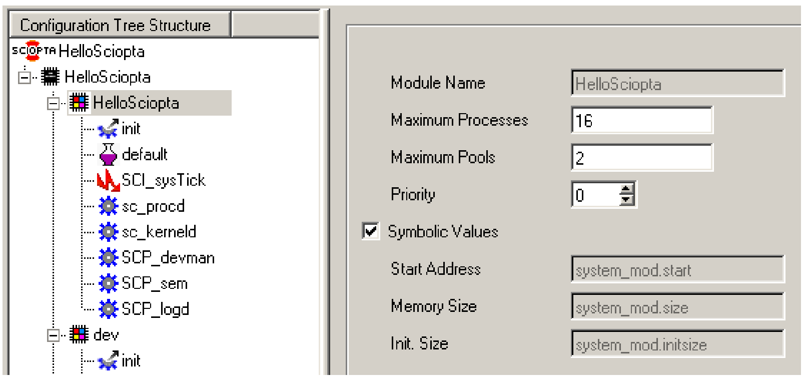
Figure 12-2: Flash File System SCONF Configuration Example

Above example system consists of four modules.

HelloSciopta	This is the system module and contains the daemons (kernel daemon sc_kerneld, process daemon sc_procd and log daemon SCP_logd), the file system manager (device manager SCP_devman) and other system pools and system processes. The system module gets automatically the name of the system.
dev	This module holds the file system processes and pools.
user	In this user module the file system application process and pool are placed.

12.4 System Module

The system module gets automatically the name HelloSciopta which is the system name.



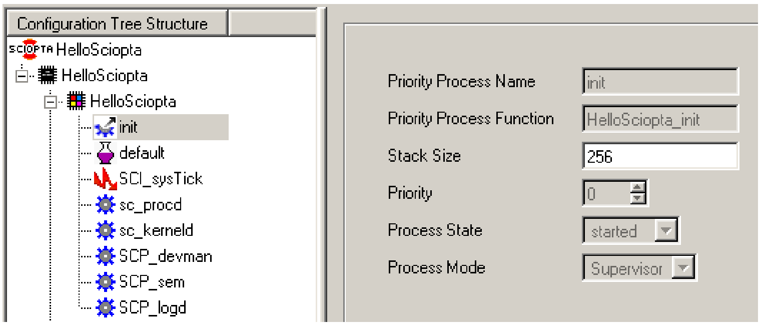
Configuration Tree Structure

- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd
    - dev
      - init

Module Name	HelloSciopta
Maximum Processes	16
Maximum Pools	2
Priority	0
☑ Symbolic Values	
Start Address	system_mod.start
Memory Size	system_mod.size
Init. Size	system_mod.initsize

12.4.1 System Module Init Process Configuration

The init process is automatically created and gets the name **init** and the function **HelloSciopta_init**. Only the stack size need to be configured.

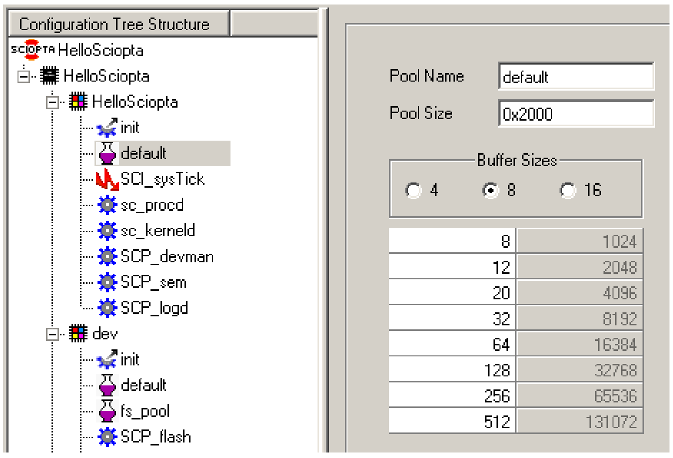


Configuration Tree Structure

- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd

Priority Process Name	init
Priority Process Function	HelloSciopta_init
Stack Size	256
Priority	0
Process State	started
Process Mode	Supervisor

12.4.2 Default Pool of the System Module



Pool Name: default

Pool Size: 0x2000

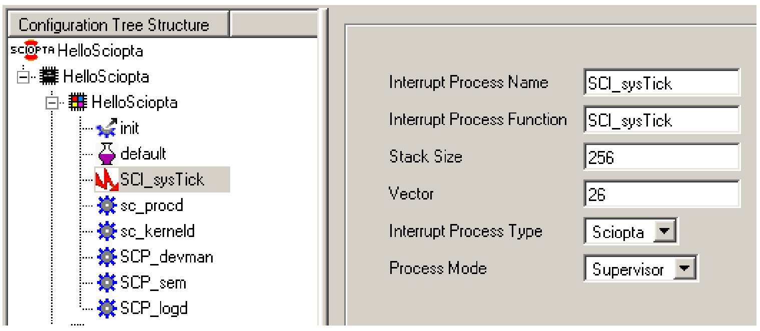
Buffer Sizes: 4 [✓ 8] 16

8	1024
12	2048
20	4096
32	8192
64	16384
128	32768
256	65536
512	131072

12.4.3 System Tick Interrupt Process

This is an interrupt process included in the SCIOPTA ARM Board Support Package.

Please consult the SCIOPTA ARM - Kernel, User's Guide for more information about the system tick and system tick interrupt process.

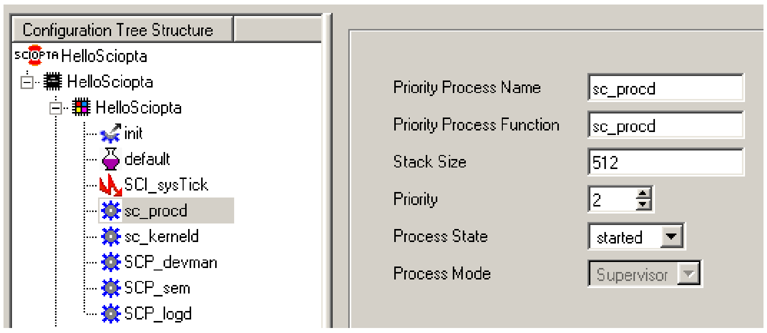


Interrupt Process Name	SCI_sysTick
Interrupt Process Function	SCI_sysTick
Stack Size	256
Vector	26
Interrupt Process Type	Sciopta
Process Mode	Supervisor

12.4.4 SCIOPTA Process Daemon

The Process Daemon (sc_procd) is identifying processes by name and supervises created and killed processes. The sc_procIdGet system call need a running process daemon.

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



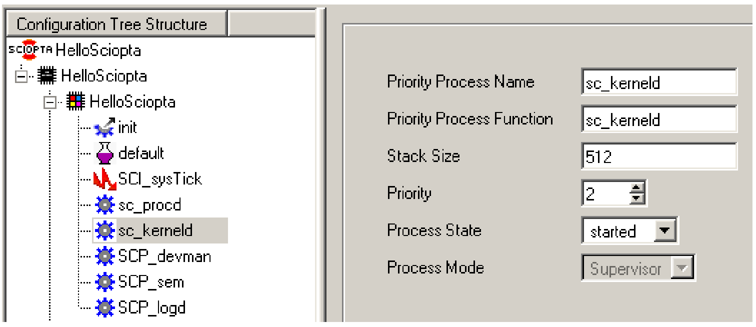
Priority Process Name	sc_procd
Priority Process Function	sc_procd
Stack Size	512
Priority	2
Process State	started
Process Mode	Supervisor

12.4.5 SCIOPTA Kernel Daemon

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

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

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



Priority Process Name	sc_kerneld
Priority Process Function	sc_kerneld
Stack Size	512
Priority	2
Process State	started
Process Mode	Supervisor

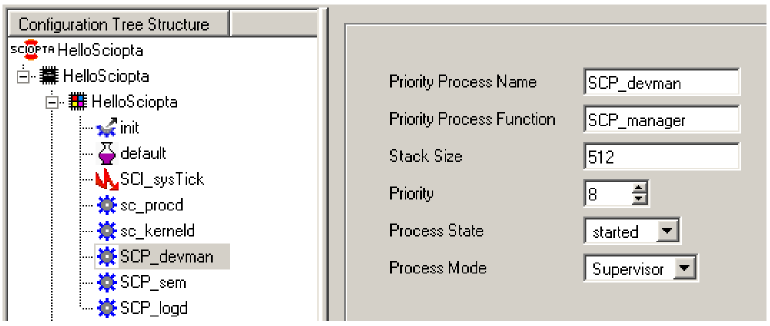
12.4.6 Flash File System Manager Process

SCIOPTA Flash File Systems are basically organized the same way as SCIOPTA devices. Therefore there is also a manager process which maintains the file system. The files system process registers the file system at the file system manager process. The file system manager process holds a list of registered file systems.

For each physical device you need a file system manager process.

The file system manager process is a standard manager process included in the SCIOPTA gdd library (SCP_manager).

The user just need to define a prioritized static process with the process name SCP_devman and the process function SCP_manager.



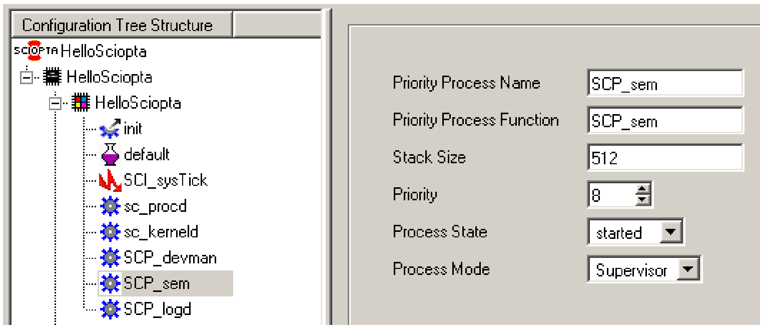
Priority Process Name	SCP_devman
Priority Process Function	SCP_manager
Stack Size	512
Priority	8
Process State	started
Process Mode	Supervisor

12.4.7 SCIOPTA Semaphore Emulation Process

The SCIOPTA - HCC Safe File System integration need the semaphore emulation process.

The semaphore emulation process is a process included in the SCIOPTA util library.

The user just need to define a prioritized static process with the process name SCP_sem and the process function SCP_sem.



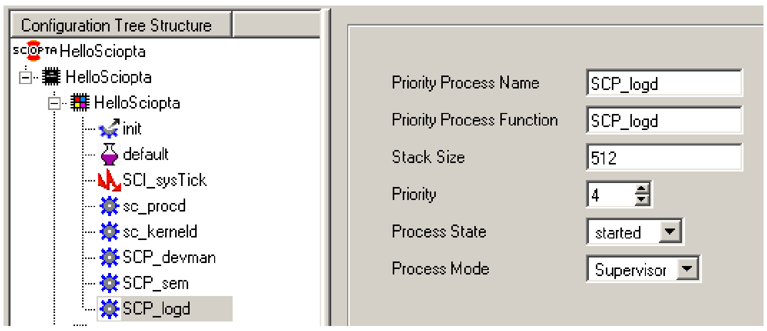
Priority Process Name	SCP_sem
Priority Process Function	SCP_sem
Stack Size	512
Priority	8
Process State	started
Process Mode	Supervisor

12.4.8 SCIOPTA Log Daemon

Log Daemon is a process which receives log messages from a user and sends it to an output device. Therefore, the user can generate debug- and log-messages to test the program flow. By setting an adequate (low) priority of the SCP_logd process the influence on the system timing can be minimized. The LogDaemon decouples occurrence and logging of events.

The LogDaemon process can be found in the utility library (e.g. for GNU GCC: libutil_x.a) of the SCIOPTA delivery.

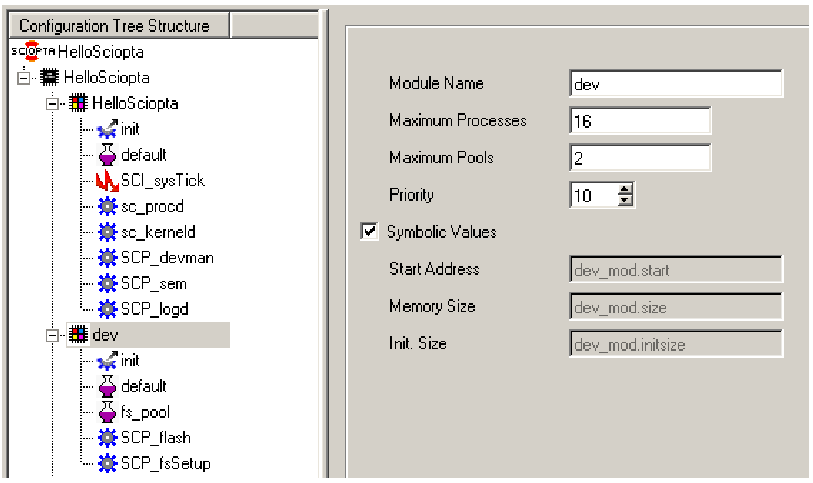
The user just need to define a prioritized static process with the process name SCP_logd and the process function SCP_logd.



Priority Process Name	SCP_logd
Priority Process Function	SCP_logd
Stack Size	512
Priority	4
Process State	started
Process Mode	Supervisor

12.5 dev Module

The flash file system processes are placed in the dev module.

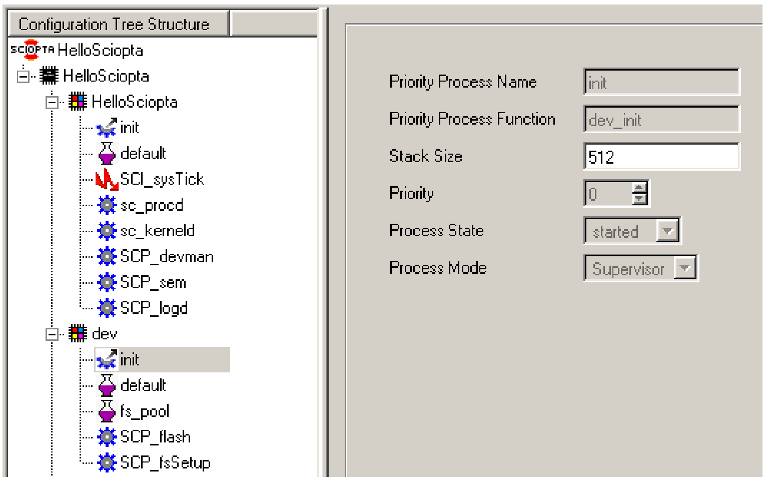


Configuration Tree Structure
- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd
    - dev
      - init
      - default
      - fs_pool
      - SCP_flash
      - SCP_fsSetup

Module Name	dev
Maximum Processes	16
Maximum Pools	2
Priority	10
✓ Symbolic Values	
Start Address	dev_mod.start
Memory Size	dev_mod.size
Init. Size	dev_mod.initsize

12.5.1 dev Module Init Process Configuration

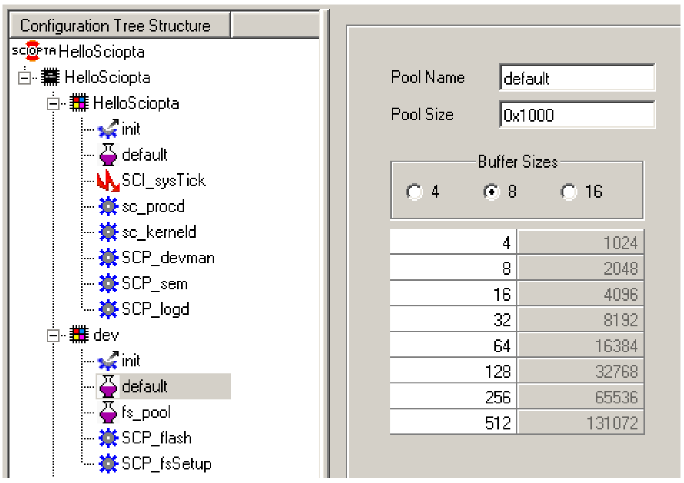
The init process is automatically created and gets the name **init** and the function **dev_init**. Only the stack size need to be configured.



Configuration Tree Structure
- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd
    - dev
      - init
      - default
      - fs_pool
      - SCP_flash
      - SCP_fsSetup

Priority Process Name	init
Priority Process Function	dev_init
Stack Size	512
Priority	0
Process State	started
Process Mode	Supervisor

12.5.2 Default Pool of the dev Module



Configuration Tree Structure

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      SCI_sysTick
      sc_procd
      sc_kerneld
      SCP_devman
      SCP_sem
      SCP_logd
    dev
      init
      default
      fs_pool
      SCP_flash
      SCP_fsSetup
```

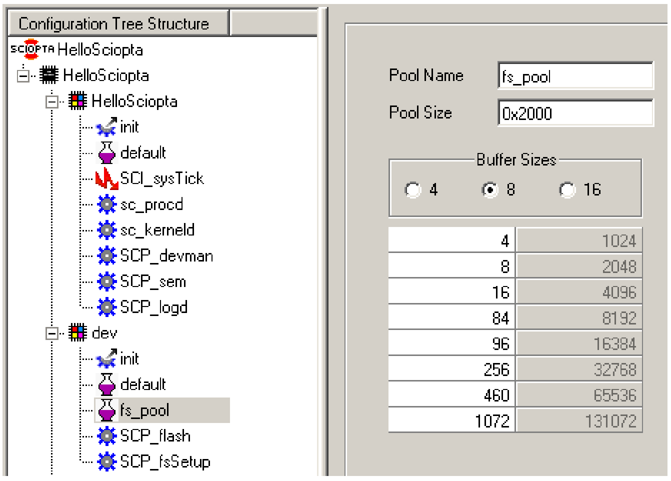
Pool Name: default

Pool Size: 0x1000

Buffer Sizes: 4 [✓ 8] 16

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

12.5.3 File System Pool



Configuration Tree Structure

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      SCI_sysTick
      sc_procd
      sc_kerneld
      SCP_devman
      SCP_sem
      SCP_logd
    dev
      init
      default
      fs_pool
      SCP_flash
      SCP_fsSetup
```

Pool Name: fs_pool

Pool Size: 0x2000

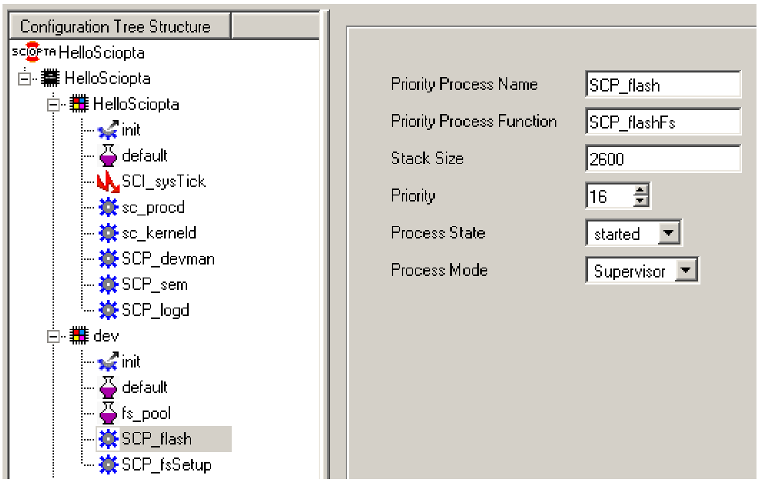
Buffer Sizes: 4 [✓ 8] 16

4	1024
8	2048
16	4096
84	8192
96	16384
256	32768
460	65536
1072	131072

12.5.4 File System Process

The file system process includes the main SCIOPTA Flash File System initialization functions. It is using the HCC Safe File System (HCC Safe API), the HCC Safe File System drivers and the SDD (SCIOPTA Device Driver) interface library. The file system process registers the file system at the file system manager process. For each physical device you need a file system process.

The user needs to define a prioritized static process with the process name **SCP_flash** and the process function **SCP_flashFs**.

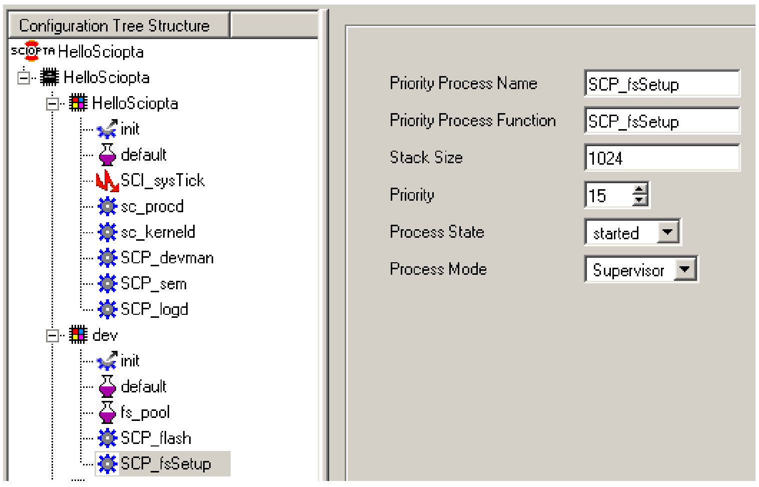


Priority Process Name	SCP_flash
Priority Process Function	SCP_flashFs
Stack Size	2600
Priority	16
Process State	started
Process Mode	Supervisor

12.5.5 File System Setup Process

This is a user written process and is mainly used for mounting file systems and to wait for file systems to be initialized and up-and-running.

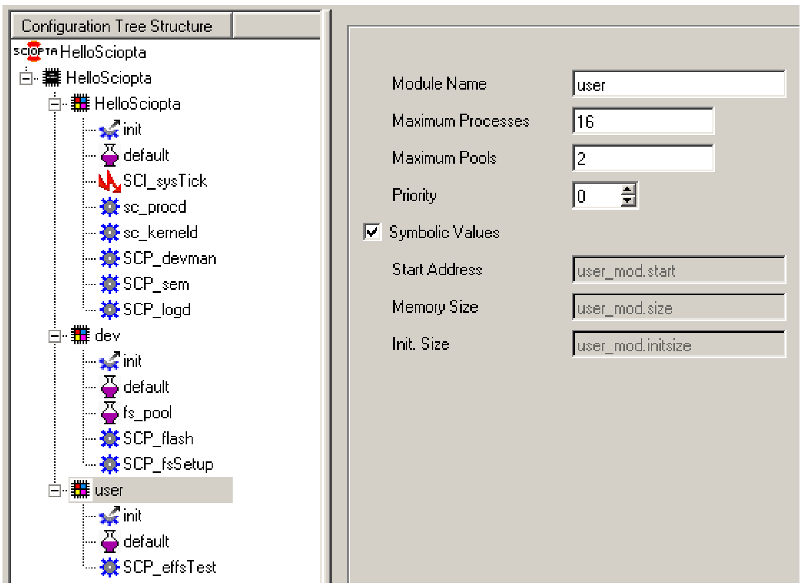
The user needs to define a prioritized static process with the process name **SCP_fsSetup** and the process function **SCP_fsSetup**.



Priority Process Name	SCP_fsSetup
Priority Process Function	SCP_fsSetup
Stack Size	1024
Priority	15
Process State	started
Process Mode	Supervisor

12.6 user Module

The user Module contains the flash file system application.



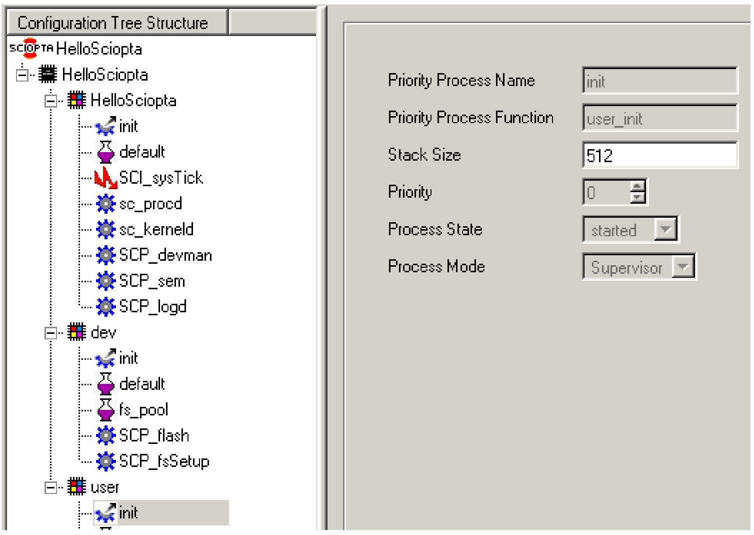
Configuration Tree Structure

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      SCI_sysTick
      sc_procd
      sc_kerneld
      SCP_devman
      SCP_sem
      SCP_logd
    dev
      init
      default
      fs_pool
      SCP_flash
      SCP_fsSetup
    user
      init
      default
      SCP_effsTest
```

Module Name	user
Maximum Processes	16
Maximum Pools	2
Priority	0
Symbolic Values	✓
Start Address	user_mod.start
Memory Size	user_mod.size
Init. Size	user_mod.initsize

12.6.1 user Module Init Process Configuration

The init process is automatically created and gets the name **init** and the function **user_init**. Only the stack size need to be configured.

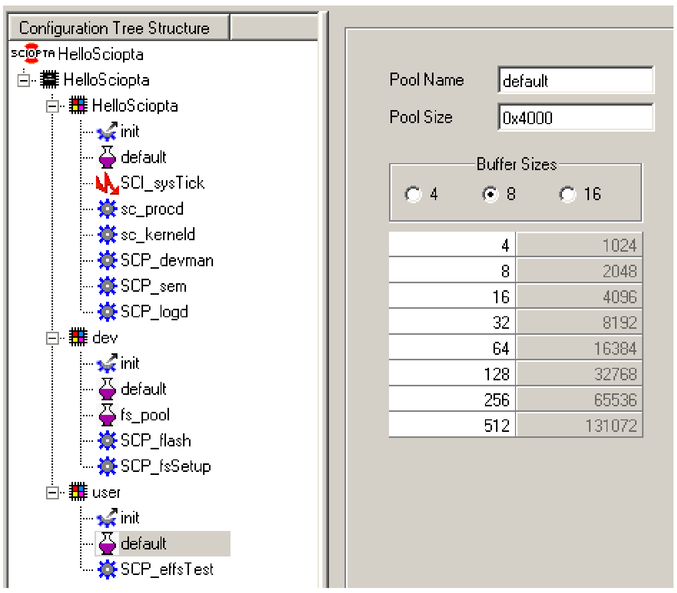


Configuration Tree Structure

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      SCI_sysTick
      sc_procd
      sc_kerneld
      SCP_devman
      SCP_sem
      SCP_logd
    dev
      init
      default
      fs_pool
      SCP_flash
      SCP_fsSetup
    user
      init
```

Priority Process Name	init
Priority Process Function	user_init
Stack Size	512
Priority	0
Process State	started
Process Mode	Supervisor

12.6.2 Default Pool of the user Module


Configuration Tree Structure

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      SCI_sysTick
      sc_procd
      sc_kerneld
      SCP_devman
      SCP_sem
      SCP_logd
    dev
      init
      default
      fs_pool
      SCP_flash
      SCP_fsSetup
    user
      init
      default
      SCP_effsTest
```

Pool Name: default

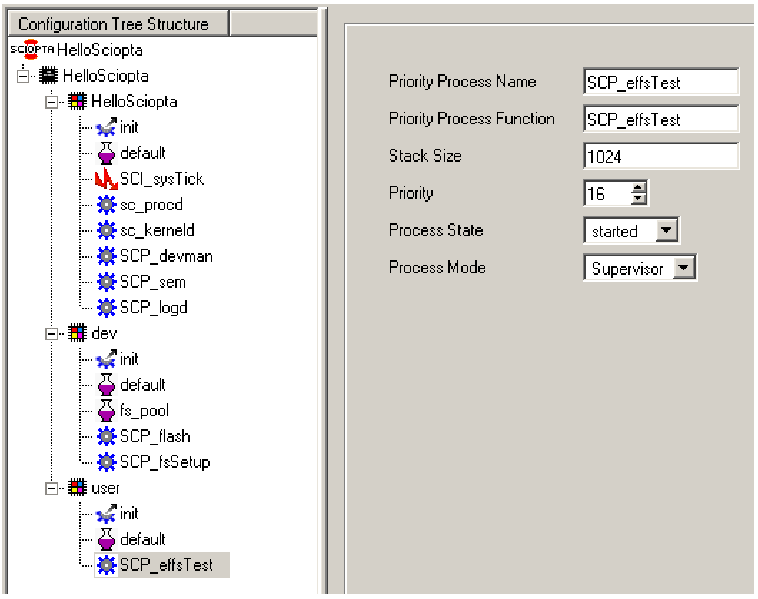
Pool Size: 0x4000

Buffer Sizes: ( ) 4 [✓ 8] ( ) 16

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

12.6.3 SCIOPTA Flash File System User Process

This is the example process of the getting started example.


Configuration Tree Structure

```
HelloSciopta
  HelloSciopta
    HelloSciopta
      init
      default
      SCI_sysTick
      sc_procd
      sc_kerneld
      SCP_devman
      SCP_sem
      SCP_logd
    dev
      init
      default
      fs_pool
      SCP_flash
      SCP_fsSetup
    user
      init
      default
      SCP_effsTest
```

Priority Process Name	SCP_effsTest
Priority Process Function	SCP_effsTest
Stack Size	1024
Priority	16
Process State	started
Process Mode	Supervisor

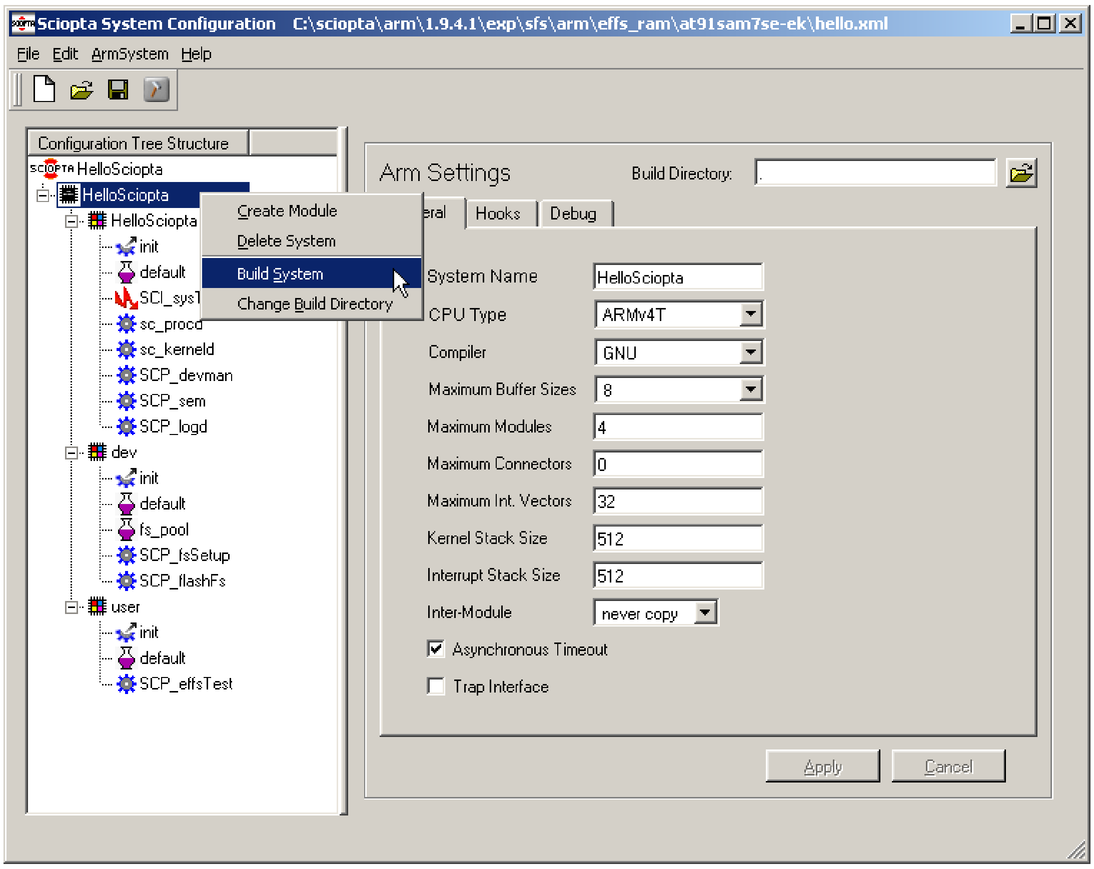
12.7 Generating the Configuration Files

After your configuration is correct the **SCONF** utility will generate three files sciopta.cnf, sconf.h and sconf.c which need to be included into your SCIOPTA project.

12.7.1 Generated Kernel Configuration Files

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

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



System Name: HelloSciopta
CPU Type: ARMv4T
Compiler: GNU
Maximum Buffer Sizes: 8
Maximum Modules: 4
Maximum Connectors: 0
Maximum Int. Vectors: 32
Kernel Stack Size: 512
Interrupt Stack Size: 512
Inter-Module: never copy
[✓] Asynchronous Timeout
[ ] Trap Interface

Figure 12-3: Build System

13 Libraries and Include Files

13.1 Kernel

The kernel is delivered as a stripped and undocumented assembler source file for each supported compiler.

SCIOPTA Kernel

sciopta.S	SCIOPTA kernel for GNU GCC.
sciopta.s	SCIOPTA kernel for ARM RealView
sciopta.s79	SCIOPTA kernel for IAR EW

File location: <installation_folder>\sciopta\<version>\lib\arm\krn\

13.2 Kernel Libraries

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

13.2.1 GNU GCC Kernel Libraries

The file name of the libraries have the following format:

lib<name>_Xt.lib

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

13.2.1.1 Optimization “X”

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

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

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

13.2.1.2 “t”

If the SCIOPTA Trap Interface is used, the libraries with the letter “t” after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

13.2.1.3 Included SCIOPTA Kernel Libraries

Utilities	libutil_**X**.a
Generic device driver	libgdd_**X**.a
Shell	libsh_**X**.a

13.2.1.4 Building Kernel Libraries for GCC

Please consult the chapter “Libraries and Include Files” of the SCIOPTA ARM - Kernel, User’s Guide for information how to build the kernel libraries.

13.2.2 ARM RealView Kernel Libraries

The file name of the libraries have the following format:

<name>_Xt.l

File location: <installation_folder>\sciopta\<version>\lib\arm\am\<version>

13.2.2.1 Optimization "X"

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

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

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

13.2.2.2 "t"

If the SCIOPTA Trap Interface is used, the libraries with the letter "t" after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

13.2.2.3 Included SCIOPTA GNU GCC Kernel Libraries

Utilities	util_**X**.l
Generic device driver	gdd_**X**.l
Shell	sh_**X**.l

13.2.2.4 Building Kernel Libraries for ARM RealView

Please consult the chapter "Libraries and Include Files" of the SCIOPTA ARM - Kernel, User's Guide for information how to build the kernel libraries.

13.2.3 IAR EW Kernel Libraries

The file name of the libraries have the following format:

<name>_Xt.r79

File location: <installation_folder>\sciopta\<version>\lib\arm\iar\

13.2.3.1 Optimization "X"

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

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

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

13.2.3.2 "t"

If the SCIOPTA Trap Interface is used, the libraries with the letter "t" after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

13.2.3.3 Included SCIOPTA GNU GCC Kernel Libraries

Utilities	util_**X**.s79
Generic device driver	gdd_**X**.s79
Shell	sh_**X**.s79

13.2.3.4 Building Kernel Libraries for IAR EW

Please consult the chapter "Libraries and Include Files" of the SCIOPTA ARM - Kernel, User's Guide for information how to build the kernel libraries.

13.3 SCIOPTA File System Library

The file system library contains the file system function interface (see chapter 8 “SCIOPTA SFFS Function Interface Reference” on page 8-1).

You need to include this library for all SCIOPTA Flash File System projects.

13.3.1 GNU GCC File System Libraries

The file name of the libraries have the following format:

lib<name>_Xt.lib

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

13.3.1.1 Optimization “X”

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

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

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

13.3.1.2 “t”

If the SCIOPTA Trap Interface is used, the libraries with the letter “t” after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

13.3.1.3 Included SCIOPTA File System Libraries

SCIOPTA File System Function Interface	libsfs_**Xt**.a

13.3.2 ARM RealView File System Libraries

The file name of the libraries have the following format:

<name>_Xt.a

File location: <installation_folder>\sciopta\<version>\lib\arm\arm\<version>

13.3.2.1 Optimization "X"

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

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

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

13.3.2.2 "t"

If the SCIOPTA Trap Interface is used, the libraries with the letter "t" after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

13.3.2.3 Included SCIOPTA File System Libraries

SCIOPTA File System Function Interface	sfs_**Xt**.a

13.3.3 IAR EW File System Libraries

The file name of the libraries have the following format:

<name>_Xt.r79

File location: <installation_folder>\sciopta\<version>\lib\arm\iar\

13.3.3.1 Optimization "X"

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

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

0	No Optimization.
1	Optimization for size.
2	Optimization for speed.

13.3.3.2 "t"

If the SCIOPTA Trap Interface is used, the libraries with the letter "t" after the Optimization letter X must be included. Systems using a Memory Management Unit (MMU) and the SCIOPTA MMU support module (SMMS, SCIOPTA Memory Management System) need to link these libraries.

13.3.3.3 Included SCIOPTA File System Libraries

SCIOPTA File System Function Interface	sfs_**Xt**.r79

13.4 Include Files

13.4.1 Include Files Search Directories

Please make sure that the environment variable **SCIOPTA_HOME** is defined as explained in the chapter “Installation” of the SCIOPTA ARM - Kernel, User’s Guide.

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

```
.
%(SCIOPTA_HOME)\include
%(SCIOPTA_HOME)\include\sciopta\arm
```

Depending on the CPU and board you are using:

```
%(SCIOPTA_HOME)\bsp\arm\include
%(SCIOPTA_HOME)\bsp\arm\pxa270\include
%(SCIOPTA_HOME)\bsp\arm\pxa270\<BOARD>\include
```

13.4.2 Main Include File sciopta.h

This file contains some main definitions and the SCIOPTA Application Programming Interface.

Each module or file which is using SCIOPTA system calls and definitions must include the file **sciopta.h**.

sciopta.h	Main include file.

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

The file sciopta.h includes all specific API header files.

File location: <installation_folder>\sciopta\<version>\include\kernel

13.4.3 Configuration Definitions sconf.h

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

The file **sconf.h** needs to be included if for instance you want to know the maximum number of modules allowed in a system. This information is stored in **SC_MAX_MODULES** in the file **sconf.h**. Please remember that **sconf.h** is automatically generated by the **sconf** configuration tool.

13.4.4 Main Data Types types.h

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

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

types.h	Processor independent data types.

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

13.4.5 ARM Data Types types.h

The ARM specific data types are defined in the file types.h located in \arm\arch.

types.h	ARM data types.

File location: <installation_folder>\sciopta\<version>\include\sciopta\arm\arch\

13.4.6 Global System Definitions defines.h

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

defines.h	System wide constant definitions.

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

14 Linker Scripts and Memory Map

14.1 Introduction

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

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

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

14.2 GCC Linker Script

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

<board>.ld	Linker script for GNU GCC

File location: <installation_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\inlcude\

This linker script file includes another linker script which is usually located at:

module.ld	Linker script for GNU GCC

File location: <installation_folder>\sciopta\<version>\bsp\arm\inlcude\

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

Please consult the chapter “Linker Scripts and Memory Map” of the ARM - Kernel User’s Guide for more information.

14.3 IAR Embedded Workbench Linker Script

You can find examples of linker scripts in the SCIOPTA examples and getting started projects. The linker scripts are board dependent and can be found at:

<board>.xcl	Linker script for IAR

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

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

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

Source File

map.c	Module mapping definitions for IAR

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\<example>\<board>\

Please consult the chapter "Linker Scripts and Memory Map" of the ARM - Kernel User's Guide for more information.

14.4 ARM RealView Linker Script

You can find examples of linker scripts in the SCIOPTA examples and getting started projects. The linker scripts are board dependent and can be found at:

<board>.sct	Linker script for ARM RealView

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

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

Please consult the chapter "Linker Scripts and Memory Map" of the ARM - Kernel User's Guide for more information.

15 System Building

15.1 Introduction

A SCIOPTA System can be built by using many different compilers and integrated development environments.

You can only use compilers which are officially supported by SCIOPTA.

In a typical SCIOPTA delivery you will find project files for different integrated development environments. An integrated development environment (IDE) is computer software to help computer programmers develop software.

They normally consist of a source code editor, a compiler and/or interpreter, build-automation tools, and (usually) a debugger. Sometimes a version control system and various tools to simplify the construction of a GUI are integrated as well. Many modern IDEs also integrate a class browser, an object inspector and a class hierarchy diagram, for use with object oriented software development. Although some multiple-language IDEs are in use, such as the Eclipse IDE or Microsoft Visual Studio, typically an IDE is devoted to a specific programming language.

15.2 Makefile and GNU GCC

15.2.1 Tools

You will need

- GNU GCC compiler version 3.4.4
- GNU Binutils version 2.16.1

These products can be found on the separate SCIOPTA ARM tools CD.

To run the makefile we are using the GNU Make utility which we have included in the \sciopta\bin\win32 directory. The GNU make consists of the following files:

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

15.2.2 Environment Variables

Please consult the chapter "System Building" of the ARM - Kernel User's Guide for information about environment variables.

15.2.3 Makefile Location

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

Makefile	Example makefile.

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\<example>\

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

board.mk	Board dependent makefiles.

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\<example>\<board>\

15.2.4 Project Setting

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

15.3 Eclipse IDE and GNU GCC

15.3.1 Introduction

Eclipse is an open source community whose projects are focused on providing an extensible development platform and application frameworks for building software. Eclipse provides extensible tools and frameworks that span the software development life cycle, including support for modelling, language development environments for Java, C/C++ and others, testing and performance, business intelligence, rich client applications and embedded development. A large, vibrant ecosystem of major technology vendors, innovative start-ups, universities and research institutions and individuals extend, complement and support the Eclipse Platform.

Please consult http://www.eclipse.org/ for more information.

15.3.2 Tools

You will need

- GNU GCC compiler version 3.4.4
- GNU Binutils version 2.16.1
- Eclipse Version 3.3.1.1 (special distribution including SCIOPTA plug-ins)

These products can be found on the separate SCIOPTA ARM tools CD.

In order to run the Eclipse Platform you also need the Sun Java 2 SDK, Standard Edition for Microsoft Windows.

To run the Eclipse build and make we are using the GNU Make utility which we have included in the \sciopta\bin\win32 directory. The GNU make consists of the following files:

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

15.3.3 Environment Variables

Please consult the chapter "System Building" of the ARM - Kernel User's Guide for information about environment variables.

15.3.4 Eclipse Project Files Location

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

.cproject	Eclipse project file

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\<example>\<board>

Please consult chapter 3.2.4 “Eclipse IDE and GNU GCC” on page 3-5 to get a detailed description how to create a SCIOPTA project inside Eclipse.

15.3.5 Eclipse Project Settings

Selecting **File > Properties** from the menu (or press **Alt+Enter**) opens the **Properties** window.

After expanding the **C/C++ Build** entry you can select **Settings** to see the project specific settings on the right side.

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

15.4 iSYSTEM© winIDEA

15.4.1 Introduction

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

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

15.4.2 Tools

You will need

- GNU GCC compiler version 3.4.4
- GNU Binutils version 2.16.1

These products can be found on the separate SCIOPTA ARM tools CD.

- iSYSTEM winIDEA

15.4.3 Environment Variables

Please consult the chapter "System Building" of the ARM - Kernel User's Guide for information about environment variables.

15.4.4 winIDEA Project Files Location

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

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

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\<example>\<board>

Please consult chapter 3.2.5 "iSYSTEM winIDEA and GNU GCC" on page 3-7 to get a detailed description how to create a SCIOPTA project inside Eclipse.

15.4.5 winIDEA Project Settings

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

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

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

15.5 IAR Embedded Workbench for ARM

15.5.1 Introduction

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

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

15.5.2 Tools

You will need

- IAR Embedded Workbench for ARM including the following main components:
    - IAR Assembler for ARM
    - IAR C/C++ Compiler for ARM
    - IAR Embedded Workbench IDE
    - IAR XLINK
    - IAR C-SPY including suitable target connection

15.5.3 Environment Variables

Please consult the chapter “System Building” of the ARM - Kernel User’s Guide for information about environment variables.

15.5.4 IAR EW Project Files Location

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

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

File location: <installation_folder>\sciopta\<version>\exp\sfs\arm\<example>\<board>

Please consult chapter 3.2.6 “IAR Embedded Workbench” on page 3-9 to get a detailed description how to create a SCIOPTA project inside Eclipse.

15.5.5 IAR EW Project Settings

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

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

16 Manual Versions

16.1 Manual Version 2.2

- Chapter 4.2 Files, added.
- Chapter 4.10 System Configuration and Initialization, file: **map.c** added.
- Chapter 4.11 General System Functions and Drivers, file: **winIDEA_gnu.ind** added.
- Chapter 4.12 ARM System Functions and Drivers, former chapter "4.15 C/C++ Language Setup" here included as **Source File**.
- Chapter 4.12 ARM System Functions and Drivers, file: **module.ld** moved from former chapter "4.16 Linker Scripts and Memory Map" to here as **Project Files**.
- Chapter 4.14 Board System Functions and Drivers, **Files** for IAR EW included.
- Chapter 4.14 Board System Functions and Drivers, former chapter "4.16 Linker Scripts and Memory Map" here included as **Linker Scripts**.
- Chapter 4.14 Board System Functions and Drivers, **Debugger Board Setup** files added.
- Chapter 4.18.2 Eclipse, project file name corrected, file .settings removed.
- Chapter 11.2 General System Functions and Drivers, file: **winIDEA_gnu.ind** added.
- Chapter 13.1 Kernel, ARM RealView and IAR EW added.
- Chapter 13.2.1 GNU GCC Kernel Libraries, file location added.
- Chapter 13.2.2 ARM RealView Kernel Libraries, added.
- Chapter 13.2.3 IAR EW Kernel Libraries, added.
- Chapter 13.3.2 ARM RealView File System Libraries, added.
- Chapter 13.3.3 IAR EW File System Libraries, added.
- Chapter 13.4.6 Global System Definitions defines.h, file location corrected.
- Chapter 14 Linker Scripts and Memory Map, new chapter added.
- Chapter 15 System Building, new chapter added.

16.2 Manual Version 2.1

- Chapter 3 Getting Started, some minor corrections.
- Chapter 3.2.3.1 and 3.2.4.1, Equipment, MSYS utility removed.
- Chapter 12.5.6 File System Setup Process, wrong process name and function corrected.

16.3 Manual Version 2.0

- Manual completely rewritten.

16.4 Manual Version 1.1

- All sdd_obj_t * changed to sdd_obj_t NEARPTR to support SCIOPTA 16 Bit systems.
- Chapter 2.1 Diagram, process File System Factory removed.

- Chapter 3.3 File System Factory replace by Files System Process. Whole chapter rewritten.
- Chapter 3.6 Setting Up the File System, 5. file system factory process replace by file system process.
- Chapter 3.6 Setting Up the File System, paragraph 6. removed.
- Chapter 3.7.1 Example, replaced by new version.
- Chapter 4.2 Standard SDD Object Descriptor Structure sdd_obj_t, **type** SFS_DIR_TYPE added.
- Former chapter 4.3 Base Object Info Structure sdd_objInfo_t, removed.
- Chapter 4.3 NEARPTR and FARPTR, added.
- Chapter 5.3 SDD_OBJ_INFO / SDD_OBJ_INFO_REPLY, removed.
- Chapter 6 Function Interface Reference, function codes added.
- Chapter 6.1 hccfatfs_dev, parameter **userData** description modified.
- Chapter 6.2 hccfatfs_format, parameter fattype named into mediatype, new parameter flags.
- Chapter 6.4, sfs_close, chapter 6.7, sfs_delete, chapter 6.8, sfs_get, chapter 6.11, sfs_open, chapter 6.12, sfs_read, chapter 6.13, sfs_resize, chapter 6.14, sfs_seek and chapter 6.17, sfs_write, added.
- Chapter 6.5 sfs_copy and chapter 6.10 sfs_move, parameter **fs** renamed into **dir**.
- Chapter 6.7 sfs_delete, parameter **root** rename into **dir**.

16.5 Manual Version 1.0

- Initial version.

17 Index