SCIOPTA

High Performance
Real-Time Operating Systems

ARM
FAT File System

User’s Guide

Copyright

Disclaimer

Trademark

Headquarters

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

Document No. S04290RL1

Table of Contents

1 Introduction

1.1 SCIOPTA ARM Real-Time Operating System

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

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

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

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

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

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

The 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** FAT 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 FAT File System concept and gives an introduction how to use the **SCIOPTA** 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 FAT File System Overview

SCIOPTA FAT File System is a DOS compatible file system designed for embedded systems requiring the attachment of PC compatible media and in particular flash card memory devices.

1.3.1 Features

- FAT12, FAT16 and FAT32
- Long Filenames
- Robust
- Multiple Volumes
- Mix of media types
- Unicode16 support
- Media Error Handling
- Fully PC compatible
- Multiple files open for reading/writing
- Multiple simultaneous users of open file
- ANSI C compliant C source
- Standard API (fopen, fclose, fwrite, etc.)
- Efficient zero-copy read/write
- Caching Options
- CheckDisk Utility

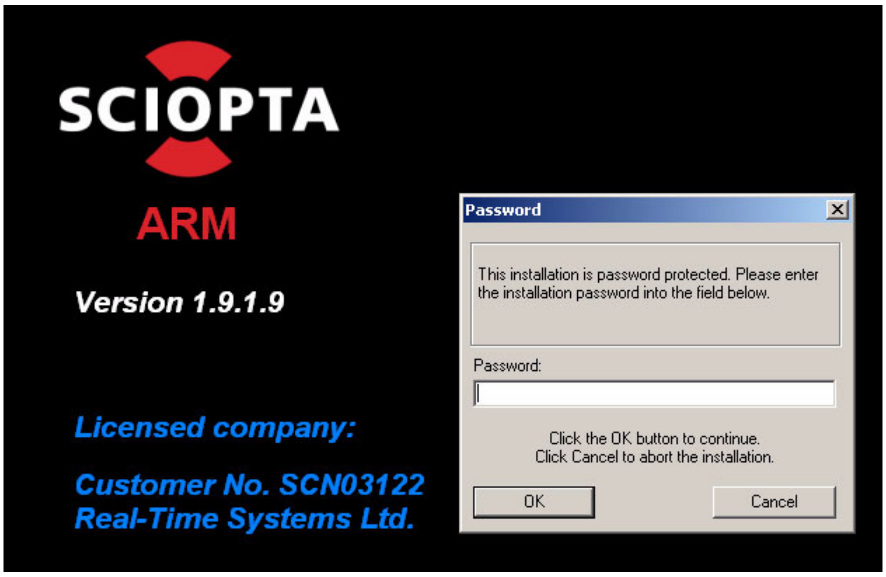
1.3.2 Sample Drivers

- Compact Flash Cards
- MultiMediaCards
- SD, SDv2 and SDHC Cards
- Hard Disk Drive
- RAM Drive
- USB Mass Storage
- USB MTP (Media Transfer Protocol)
- NAND FLash
- Atmel DataFlash

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.

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 - FAT File System RAM Disk Example

3.2.1 Description

In this SCIOPTA FAT 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 FAT 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 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_fatfs is included which starts the file system.
- The file fatfstest.c contains some user code to show file system functionality included in the process SCP_fatfsTest.

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

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

3.2.2 FAT File System RAM Disk Example Block Diagram

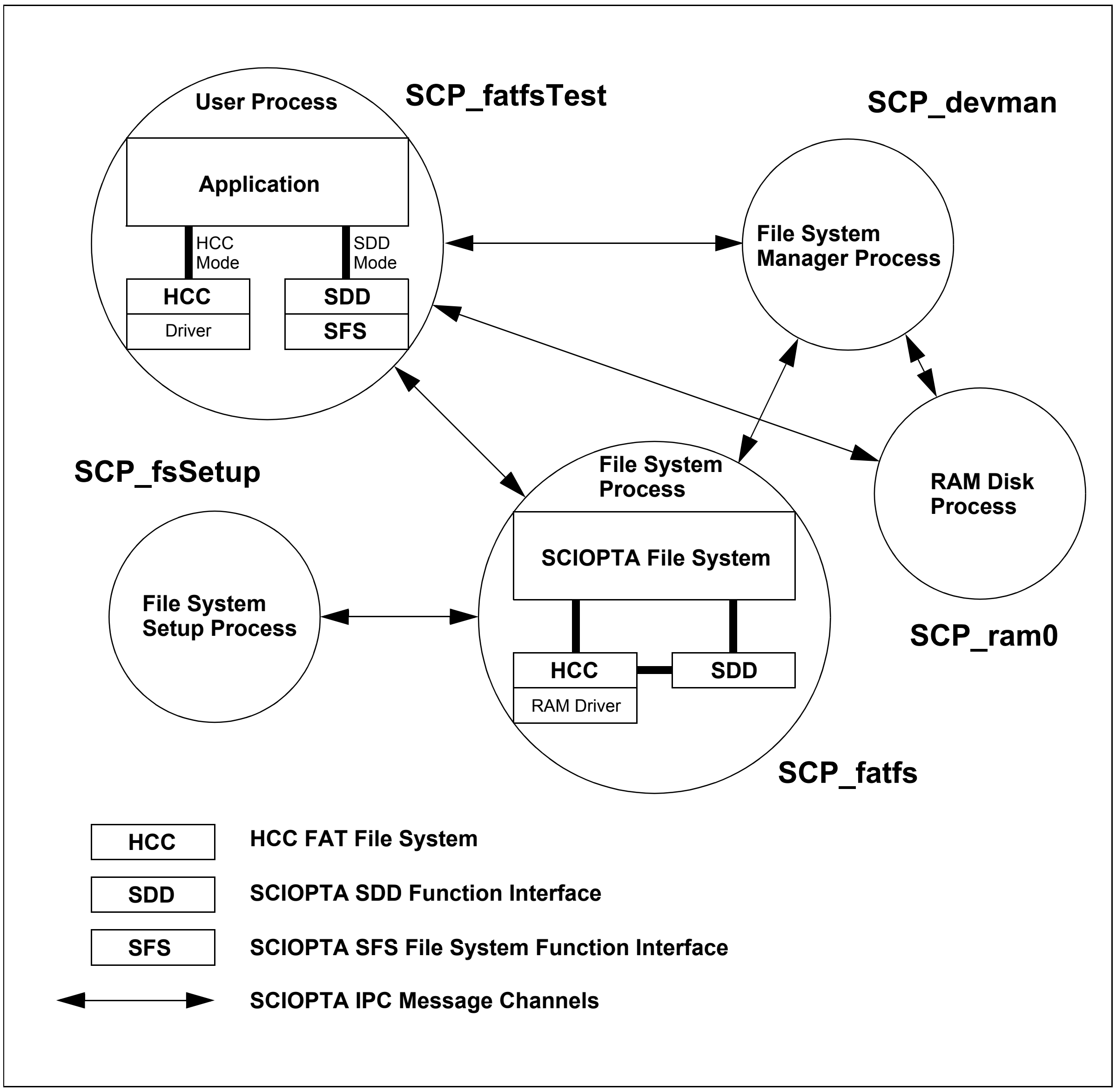


Figure 3-1: SCIOPTA FAT File System Block Diagram

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\fatfs_ram\. Actually supported are:
    - **Atmel AT91SAM7SE-EK**
    - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - FAT 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\effs_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\fatfs_ram\. Actually supported are:
    - **Atmel AT91SAM7SE-EK**
    - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - FAT 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_fatfsram** 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\tinyeffs\<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_fatfsram**) is highlighted and open the project (menu: **Project > Open Project**).
22. Expand the project by selecting the [+] button and make sure that the **sfs_fatfsram** 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_fatfsram**) 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\fatfs_ram\. Actually supported are:
  - **Atmel AT91SAM7SE-EK**
  - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - FAT 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\fatfs_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\fatfs_ram\. Actually supported are:
  - **Atmel AT91SAM7SE-EK**
  - **Atmel AT91SAM9261-EK**
- SCIOPTA ARM - Kernel.
- SCIOPTA ARM - FAT 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\fatfs_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 FAT File System Project Overview

4.1 Introduction

This is an introduction into a typical SCIOPTA ARM FAT 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 FAT 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 FAT 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 FAT File System Techniques and Concepts

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

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

4.6.1 HCC Mode

The SCIOPTA FAT File System uses and integrates the FAT 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 FAT File System API.

Please consult chapter 6 “HCC FAT 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 FAT File System function interface (API) can be found in chapter 8 “SCIOPTA FATFS Function Interface Reference” on page 8-1.

4.7 SCIOPTA FAT File System Files

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

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

Files

common.c	HCC common functions.
fat.c	HCC FAT short filename functions.
fat_lfn.c	HCC alternative source file to fat.c for long filenames.
fat_m.c	HCC FAT file system reentrancy wrapper.
port.c	Ported functions

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

SCIOPTA - HCC Integration Layer Files

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

sc2fatfs.c	HCC FAT FS 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 FAT 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 FAT File System application.

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

Files

fatfstest.c	Example file system process (SDD Mode)
hcc_fatfstest.c	Example file system process (HCC Mode)
fs_setup.c	Example file system setup process

File location<installation_folder>\sciopta\<version>\exp\sfs\common\fatfs_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 9.17 “Error Hook” on page 9-9 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 “fatfs_ram” SCIOPTA FAT 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\fatfs_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\fatfs_ram\<board>\

For the SCIOPTA FAT 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\fatfs_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 FAT File System we will use HCC FAT File System drivers. Please consult chapter 11.21 “FAT File System Drivers” on page 11-35 for more information about the HCC FAT File System drivers.

For the “fatfs_ram” SCIOPTA FAT File System getting started example we will use the RAM disk driver.

Files

ramdrv_f.c	RAM driver.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\ram\

Project Files

winIDEA_gnu.ind	iSYSTEM winIDEA indirection file for GNU GCC

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

4.12 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. 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 "fatfs_ram" SCIOPTA FAT 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\fatfs_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 FAT 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 "fatfs_ram" SCIOPTA FAT 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\fatfs_ram\

board.mk	Board dependent makefiles.

File location: <installation_folder>\sciopta\<version>\exp\krn\arm\fatfs_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 "fatfs_ram" SCIOPTA FAT 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\fatfs_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 “fatfs_ram” SCIOPTA FAT 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\fatfs_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 “fatfs_ram” SCIOPTA FAT 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\fatfs_ram\<board>

5 FAT File System Technology and Methods

5.1 SCIOPTA FAT File System Overview

SCIOPTA FAT File System is a DOS compatible file system designed for embedded systems requiring the attachment of PC compatible media and in particular flash card memory devices.

Some of the features include:

- FAT12, FAT16 and FAT32
- Long Filenames
- Robust
- Multiple Volumes
- Mix of media types
- Unicode16 support
- Media Error Handling
- Fully PC compatible
- Multiple files open for reading/writing
- Multiple simultaneous users of open file
- ANSI C compliant C source
- Standard API (fopen, fclose, fwrite, etc.)
- Efficient zero-copy read/write
- Caching Options
- CheckDisk Utility

Sample Drivers

- Compact Flash Cards
- MultiMediaCards
- SD, SDv2 and SDHC Cards
- Hard Disk Drive
- RAM Drive
- USB Mass Storage
- USB MTP (Media Transfer Protocol)
- NAND FLash
- Atmel DataFlash

5.2 System Features

5.2.1 Unicode Support

Support for 16-bit Unicode is provided.

Unicode 7/8 are supported by the file system transparently. The option described here is required only for Unicode 16 support. To support Unicode 16 character sets the developer must uncomment the line:

/* #define HCC_UNICODE */

This will force any build to include the Unicode 16 API. This build will also force Long Filename support (see next section), which is necessary for Unicode 16 support. With this build you may now use the **Unicode16** API calls. Section 5 describes the API functions that may be used with Unicode 16 strings.

Use of Unicode 16 implies that the host system has wchar ("wide character") support or an equivalent definition.

5.2.2 Drives Partitions and Volumes

FAT provides functions for creating and managing multiple drives, partitions and volumes.

A drive consists of a physical medium which is controlled by a single driver. Examples are an HDD or a Compact Flash Card.

All drives contain zero or more partitions. If the drive is not partitioned, then there is just a single volume on that drive. Removable media such as flash cards have no partitions or one partition on the card.

To each partition may be added a single volume. A volume can exist on a drive without partitions.

The file system operates on volumes. You can have one volume or a set of volumes; additional functions are provided to work with the set of volumes if it's greater than one (A:, B:, C:, etc.).

5.2.3 Sector Sizes

The system allows the developer to specify the maximum sector size of the attached media. Traditionally most FAT based devices have used a sector size of 512 bytes. However for devices whose native sector size is not 512 bytes (e.g. 2K page NAND flash based devices) it is more efficient to use other values. The system allows you to specify the maximum sector size allowed.

A variable sector size is normally only required in systems which have removable media attached. If a larger than necessary maximum sector size is set, then the file system uses more RAM. Therefore, in resource constrained systems, it may be necessary to restrict the allowed sector size.

5.2.4 Long File Names

The system includes two main source files:

- **fat.c** is the file system without long filename support. If long filenames exist on the media the system will ignore the long name part and use only the short name.
- **fat_lfn.c** is the file system with complete long filename support.

Because of the increase in system resources required to handle long filenames, they should be used only when necessary. This will avoid increasing the stack sizes of applications that call the file system, as well as an increase in the amount of checking that's required.

5.2.5 Maximum Number of Volumes

The maximum number of volumes allowed by your system should be set in a specific file. This value must be set to the maximum number of volumes that will be available on the target system.

If only a RAM drive is used, the value should be 1; if a RAM drive and a CF card drive, then this value is 2, and so on.

5.2.6 Maximum Open Files

The maximum number of simultaneously open files are also defined in a specific file. It is the total number of files that may be simultaneously open across all volumes.

5.2.7 Cache

The system includes two caching mechanisms to enhance performance: FAT caching and write data caching.

5.2.7.1 FAT Caching

FAT caching enables the file system to read several sectors from the FAT in one access so that when accessing the files the file system does not have to read new FAT sectors so frequently. FAT caching is arranged in blocks so that each block can cover different areas of the FAT. The number of sectors that each block contains and the number of blocks are configurable.

5.2.7.2 Write Caching

The write cache defines the maximum number of sectors that can be written in one operation from the caller's data buffer. This is also depends on the availability of contiguous space on the target drive. The write cache requires an F_POS structure (24 bytes) for each entry in the write cache. The main purpose of these structures is to be able to wind back a write in the event of an error in writing.

5.2.7.3 Directory Cache

This can be enabled only if long file names are used.

If Directory Cache is enabled you must specify the number of sectors to read ahead with DIR_CACHESIZE. This will allocate this number of sectors of memory for directory caching (e.g., if set to 32 and F_MAX_SECTOR_SIZE is 512 then 16Kbytes of memory will be allocated).

The system will never read more than the size of a cluster into this cache; therefore, there is no value in having a DIR_CACHESIZE greater than the sectors per cluster of the target device.

5.2.8 FAT Free Cluster Bit Field

If FAT Bit Field is enabled then the system will attempt to allocate a block to contain a bit table of free clusters.

This table is maintained by the file system and is used to accelerate searches for free clusters. This makes a large difference to the write performance when writing to a large and full disk.

5.2.9 Separator Character

A separator character can be defined that allows the selection of the forward slash or back slash as a separator in file paths.

5.2.10 Fast Seeking

The developer can define a number of points in a file to use as markers to allow fast seeking in a file.

A number of points can be defined to be stored with every file descriptor. Seeking which will only work from the current position or the beginning of the file can also be selected.

5.3 Drive Format

This document does not describe a FAT file system in detail. There are many reference works to choose from for this purpose. This file system handles the majority of the features of a FAT file system with no need for the developer to understand further. However, there are some areas where an understanding may help.

There are three different forms in which your removable media may be formatted:

- Completely unformatted media
- Master Boot Record
- Boot sector information only

Please consult chapter for more information about drive format.

5.4 FAT File System Block Diagram

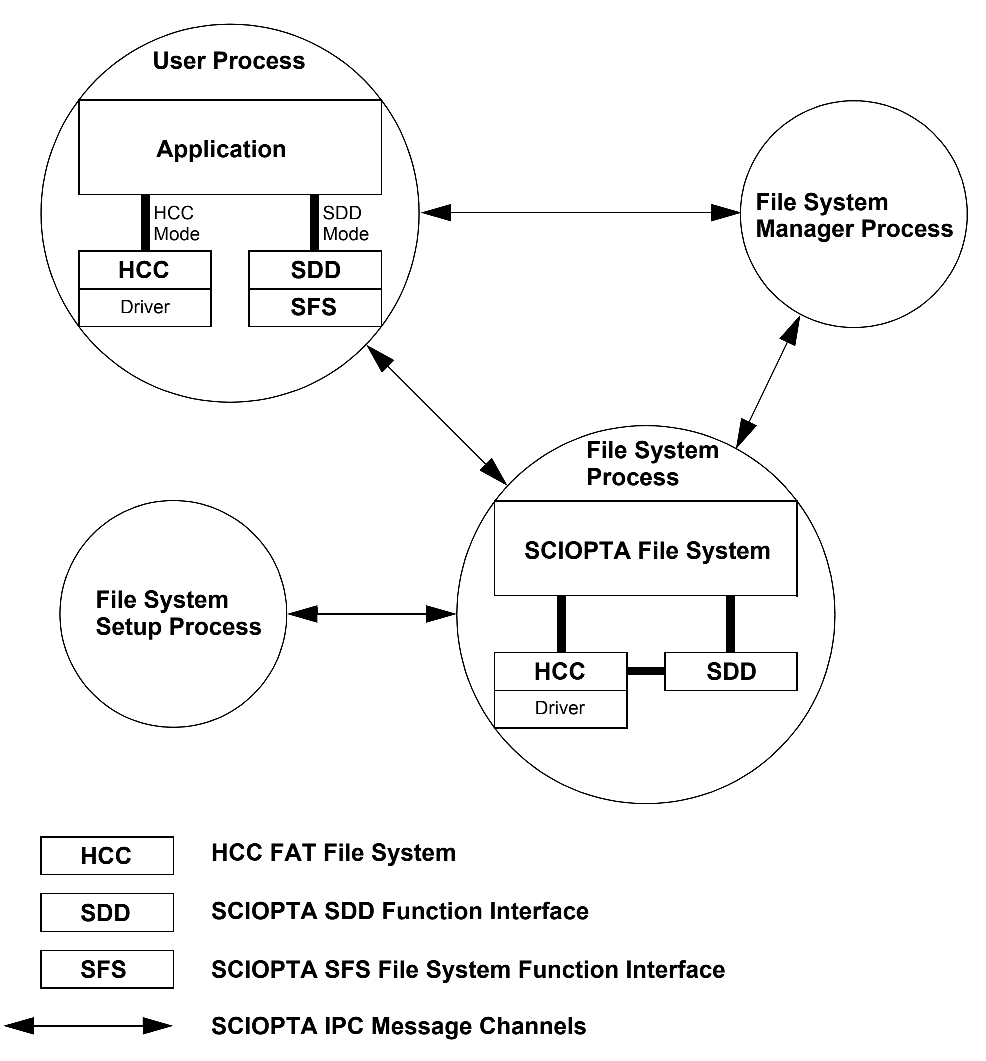

User Process
Application
HCC Mode
SDD Mode
HCC
Driver
SDD
SFS
File System Manager Process
File System Process
SCIOPTA File System
HCC
Driver
SDD
File System Setup Process
HCC
HCC FAT File System
SDD
SCIOPTA SDD Function Interface
SFS
SCIOPTA SFS File System Function Interface
SCIOPTA IPC Message Channels

Figure 5-1: SCIOPTA FAT File System Block Diagram

5.5 HCC FAT File System

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

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

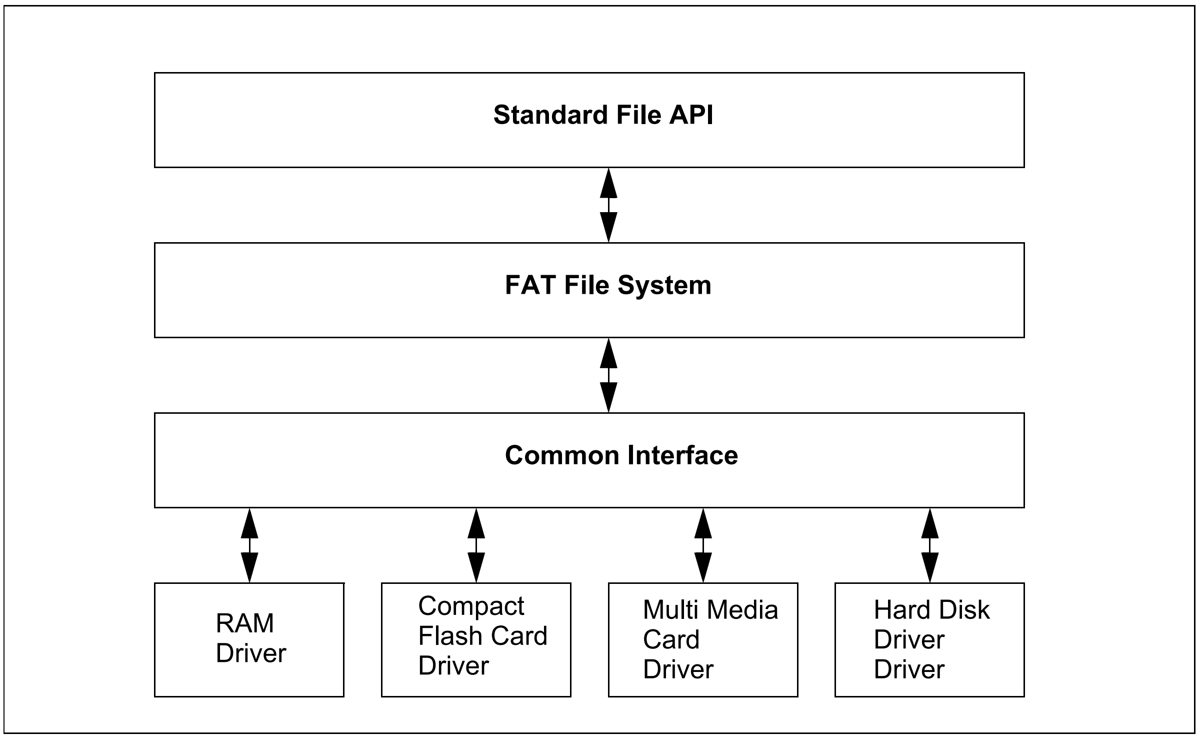


Figure 5-2: HCC FAT File System Structure

5.6 Processes

5.6.1 File System Process

The file system process includes the main SCIOPTA FAT File System initialization functions. It is using the HCC FAT File System (HCC FAT API), the HCC FAT 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 FAT 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 SCIOPTA FAT File System. The HCC FAT File System functions are called directly by the user. The HCC FAT File System runs in the context of the user process.

5.6.4.2 SDD Mode

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

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 FAT File System API

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

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

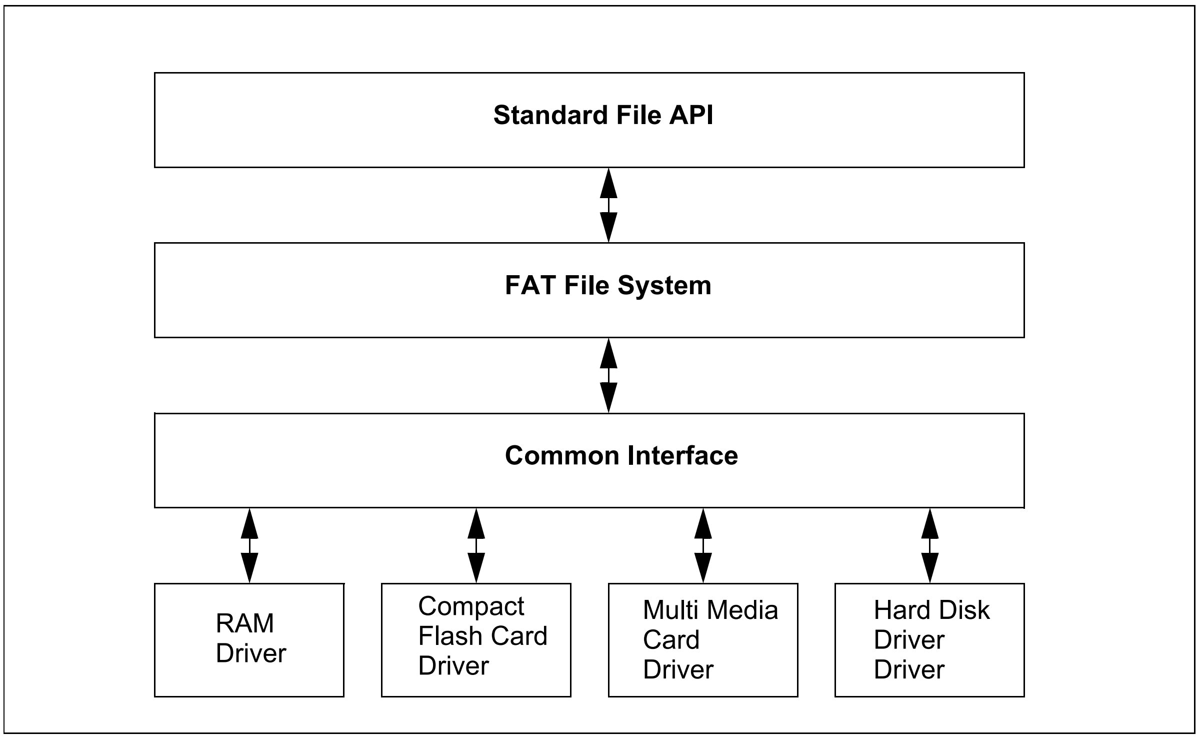


Figure 6-1: HCC FAT File System Structure

6.1 Overview

In chapter 6.2 “HCC FAT 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:

6.2 HCC FAT File System Functions List

Function	Short Description	Page
f_chdir	Change current working directory.	6-36
f_chdrive	Change current drive.	6-29
f_checkvolume	Check the status of a drive that has been initialized.	6-13
f_close	Close a previously opened file.	6-66
f_createdriver	Initialize a driver	6-17
f_createpartition	Create one or more partitions on a drive.	6-24
f_delete	Delete a file.	6-44
f_delvolume	Delete an existing volume.	6-12
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.	6-71
f_filelength	Get the length of a file.	6-46
f_findfirst	Find first file or subdirectory in specified directory.	6-48
f_findnext	Find the next file or subdirectory in a specified directory after a previous call to f_findfirst or f_findnext.	6-50
f_flush	Flush data to the media.	6-78
f_format	Format a drive.	6-16
f_ftruncate	If a file is opened for writing, then this function truncates it to the specified length.	6-79
f_getattr	Get the attributes of a specified file.	6-56
f_getc	Read a character from the current position in the open target file.	6-75
f_getcwd	Get current working folder on current drive.	6-30
f_getdcwd	Get current working folder on selected drive.	6-32
f_getdrive	Get current drive number.	6-28

Function	Short Description	Page
f_getfreespace	Fill a user allocated structure with information about the usage of the volume specified.	6-20
f_getlabel	Get the volume label of a specified file.	6-22
f_getlasterror	Returns with the last error code.	6-80
f_getpartition	Get the used sectors and system indication byte from a partitioned medium.	6-26
f_gettimedate	Get time and date information from a file or directory.	6-54
f_getversion	Retrieve file system version information.	6-5
f_get_oem	Return the OEM name “HCC_SAFE_FS”.	6-23
f_get_volume_count	Returns the number of volumes currently available to the user.	6-14
f_get_volume_list	Returns a list of volumes currently available to the user.	6-15
f_init	Initialize the file system	6-6
f_initvolume	Initialize a volume.	6-9
f_initvolumepartition	Initialize a volume on an existing partition.	6-11
f_mkdir	Make a new directory.	6-34
f_move	Moves a file or directory; the original is lost.	6-42
f_open	Opens a file.	6-62
f_putc	Write a character to the open target file at the current file position.	6-74
f_read	Read data from the current file position.	6-68
f_releasedriver	Release a driver when it is no longer required.	6-19
f_releaseFS	Release a previously assigned unique task ID.	6-8
f_rename	Rename a file or directory.	6-40
f_rewind	Set the current file position in the open target file to the beginning.	6-73
f_rmdir	Remove directory.	6-38
f_seek	Move read/write position in the file.	6-69
f_setattr	Set the attributes of a specified file.	6-58
f_seteof	Move the end of file to the current file position.	6-72
f_setlabel	Set a volume label.	6-21
f_settimedate	Set time and date on a file or on a directory.	6-52
f_stat	Get information about a file.	6-60
f_tell	Tell the current file position in the target file.	6-70
f_truncate	Opens a file for writing and truncates it to the specified length.	6-76
f_wchdir	Change current working directory with **Unicode16** name.	6-37
f_wdelete	Delete a file with **Unicode16** name.	6-45
f_wfilelength	Get the length of a file with **Unicode16** name.	6-47
f_wfindfirst	Find first file or subdirectory in specified directory with **Unicode16** name.	6-49

Function	Short Description	Page
f_wfindnext	Find the next file or subdirectory in a specified directory with **Unicode16** name after a previous call to f_wfindfirst or f_wfindnext.	6-51
f_wgetattr	Set the attributes of a specified **Unicode16** file.	6-57
f_wgetcwd	Get current working folder on current drive.	6-31
f_wgetdcwd	Get current working folder on selected drive.	6-33
f_wgettimedate	Get time and date information from a file or directory with a **Unicode16** name.	6-55
f_wmkdir	Make a new directory with a **Unicode16** name.	6-35
f_wmove	Moves a file or directory with a **Unicode16** name.	6-43
f_wopen	Opens a file with **Unicode16** file name.	6-64
f_wrename	Rename a file or directory with **Unicode16** name.	6-41
f_wrmdir	Remove **Unicode16** directory.	6-39
f_write	Write data into file at current position.	6-67
f_wsetattr	Set the attributes of a specified **Unicode16** file.	6-59
f_wsettimedate	Set time and date on a file or on a directory with **Unicode16** name.	6-53
f_wstat	Get information about a file with a **Unicode16** name.	6-61
f_wtruncate	Opens a file with a **Unicode16** name for writing and truncates it to the specified length.	6-77

6.3 General 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.

The developer can insert code into this function if there are any special requirements for a particular target system. Function initiates internal variables.

```
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(); /* initialize 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_f.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_f.c**.

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

Parameter	Description
ID	Unique identifier for calling task

Return Value	Condition
None	

6.4 Volume Functions

6.4.1 f_initvolume

This function is used to initialize a volume. The function is called with a pointer to the driver function that must be called to retrieve drive configuration information from the relevant driver. This function works independently of the status of the hardware i.e. it does not matter if a card is inserted or not.

Function **f_initvolume** always initiates the first partition on the media. To use multiple partitions, use the f_initvolumepartition function.

```
int f_initvolume(
             int            drivenum,
             F_DRIVERINIT   *driver_init,
             unsigned long  driver_param
)
```

Parameter	Description
drivenum	Drive to be initialized (0:A, 1:B...).
driver_init	Pointer to initialization function for driver.
driver_param	The **driver_param** can be used to pass information to the low-level driver. When the **xxx_initfunc** of the driver is called this parameter will be passed to the driver. The use of this parameter is optional and driver dependent. One use is to specify which device associated with the specified driver will be initialized. For convenience a definition F_AUTO_ASSIGN has been predefined to mean that the driver should assign devices as it wishes. This convention is optional and has no affect on the file system. For more information about its usage please see the Driver Interface section.

Return Value	Condition
F_NO_ERROR	Drive successfully initialized.
Error Code	Error condition.

Example

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

   f_init();

   /* Make a RAM volume on Drive A */
   f_initvolume(0, f_ramdrvinit, F_AUTO_ASSIGN);

   /*Make a Compact Flash Volume on Drive B */
   f_initvolume(1, f_cfcinit, F_AUTO_ASSIGN);

   /*Make an MMC Volume on Drive C */
   f_initvolume(2, f_mmcinit, F_AUTO_ASSIGN);

         .
         .
         .
}
```

6.4.2 f_initvolumepartition

This function is used to initialize a volume on an existing partition. The function is called with a pointer to the function that must be called to retrieve drive configuration information. This function requires the target drive to be connected.

If only the first partition is used on a medium then f_initvolume should be used.

```
int f_initvolumepartition(
              int             drivenum,
              F_DRIVER        *driver,
              int             partition
)
```

Parameter	Description
drivenum	Drive to be initialized (0:A, 1:B...).
driver	Initialized driver (get from f_createdriver).
partition	Which partition is requested to be built.

Return Value	Condition
F_NO_ERROR	Volume successfully initialized.
Error Code	Error condition.

Example

```
F_DRIVER *hdd;

int myinitfs(void)
{
   int ret;

   ret=f_createdriver(&hdd,f_hdddrvinit,0);
   if (ret) return ret;

   ret=f_initvolumepartition(0,hdd,0);
   if (ret) return ret;

   ret=f_initvolumepartition(1,hdd,1);

   return ret;
}
```

6.4.3 f_delvolume

This function is used to delete an existing volume. The link between the file system and the driver will be broken; i.e., an xxx_release call will be made to the driver. 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. If the volume's driver was created independently with f_createdriver, then this function deletes only the volume, and the f_releasedriver function is needed to be called for calling xxx_release driver functions.

This function works independently of the status of the hardware; i.e., it does not matter if a card is or is not inserted.

```
int f_delvolume(
             int            drivenum
)
```

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

Return Value	Condition
F_NO_ERROR	Drive successfully deleted.
Error Code	Error condition.

Example

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

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

6.4.4 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);
   }
      .
      .
}
```

6.4.5 f_get_volume_count

This function returns the number of volumes currently available to the user.

```
int f_get_volume_count(void)
```

Parameter	Description
none	

Return Value	Condition
Number of active volumes	

Example

```
void mygetvols(void)
{
   printf("there are %d active volumes\n",
      f_get_volume_count());
         .
         .
}
```

6.4.6 f_get_volume_list

This function returns a list of volumes currently available to the user.

```
int f_get_volume_list(
              int     *buffer
)
```

Parameter	Description
buffer	Where to store the volume list.

Return Value	Condition
number of active volumes	

Example

```
void mygetvols(void)
{
   int i,j;
   int buffer[F_MAXVOLUME];

   if (i=f_get_volume_list(buffer))
   {
      for (j=0;j<i;j++)
      {
         printf("Volume %d is active\n", buffer[j]);
      }
   }
}
```

6.4.7 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:
}
```

6.4.8 f_createdriver

This function is used to initialize a driver. The function is called with a pointer to the driver function that must be called to retrieve drive configuration information from the relevant driver.

This function works independently of the status of the hardware; i.e., it does not matter if a card is inserted or not.

This function is only necessary if multiple partitions on a drive are used.

If f_initvolume is used to initiate a volume, then f_createdriver is not required as it is called automatically.

On a drive which was created directly with the f_createdriver function then f_releasedriver must be called to release the driver.

```
int f_createdriver(
             F_DRIVER        **driver,
             F_DRIVERINIT    *driver_init,
             unsigned long   driver_param
)
```

Parameter	Description
driver	Driver ptr, where to set up driver pointer
driver_init	Pointer to initialization function for driver.
driver_param	The **driver_param** can be used to pass information to the low-level driver. When the **xxx_initfunc** of the driver is called this parameter will be passed to the driver. The use of this parameter is optional and driver dependent. One use is to specify which device associated with the specified driver will be initialized. For convenience a definition F_AUTO_ASSIGN has been predefined to mean that the driver should assign devices as it wishes. This convention is optional and has no affect on the file system. For more information about its usage please see the Driver Interface section.

Return Value	Condition
F_NO_ERROR	Drive successfully initialized.
Error Code	Error condition.

Example

```
F_DRIVER *hdd;

int myinitfs(void)
{
   int ret;

   ret=f_createdriver(&hdd,f_hdddrvinit,0);
   if (ret) return ret;

   ret=f_initvolumepartition(0,hdd,0);
   if (ret) return ret;

   ret=f_initvolumepartition(1,hdd,1);

   return ret;
}
```

6.4.9 f_releasedriver

This function is used to release a driver when it is no longer required. **f_initvolume()** or **f_createdriver()** can be called again after this.

If the driver was created by f_initvolume then this function should not be called; f_delvolume will release the driver automatically.

If the driver was created by f_createdriver then, after f_delvolume has been called for each volume on this drive, then f_releasedriver should be called to release the driver.

If the driver was created by f_createdriver and f_releasedriver is called then f_delvolume will be called automatically for each volume on this drive.

```
int f_releasedriver(
             F_DRIVER        *driver,
             int             partition
)
```

Parameter	Description
driver	Initialized driver (get from f_createdriver).

Return Value	Condition
F_NO_ERROR	Drive successfully released.
Error Code	Error condition.

Example

```
F_DRIVER *hdd;

int myinitfs(void)
{
   int ret;

   ret=f_createdriver(&hdd,f_hdddrvinit,0);
   if (ret) return ret;
   ret=f_initvolumepartition(0,hdd,0);
   if (ret) return ret;
   ret=f_initvolumepartition(1,hdd,1);
   return ret;
}

int myclose(void)
{
   return f_releasedriver(hdd);
}
```

6.4.10 f_getfreespace

This function fills a structure with information about the drive space usage - total space, free space, used space and bad (damaged) size.

If a drive is greater than 4GB, then the high elements of the returned structure (e.g., pspace.total_high) should also be read to get the upper 32 bits of each of the numbers.

The first call to this function after a drive is mounted may take some time depending on the size and format of the disk being used. After the initial call changes to the volume are counted - the function then returns immediately with this data.

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

Parameter	Description
drvenum	Number of drive.
pSpace	Pointer to F_SPACE structure

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

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

   /* get free space on current drive */
   int ret = f_getfreespace(f_getcurrdrive(),&space);

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

6.4.11 f_setlabel

This function sets a volume label. The volume label should be an ASCII string with a maximum length of 11 characters. Non-printable characters will be padded out as space characters.

```
int f_setlabel(
                int          drivenum,
                const char   *pLabel
)
```

Parameter	Description
drvenum	Number of drive.
pLabel	pointer to null terminated string to use.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void setlabel(void)
{
   int result = f_setlabel(f_getcurrdrive(),"DRIVE 1");

   if (result)
   printf("Error on Drive");
}
```

6.4.12 f_getlabel

This get the volume label of a specified file. The pointer passed for storage should be capable of holding an 11 character string.

```
int f_getlabel(
                int          drivenum,
                const char   *pLabel,
                long         len
)
```

Parameter	Description
drvenum	Number of drive.
pLabel	pointer to copy label to.
len	length of storage area.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void getlabel(void)
{
   char label[12];
   int result;

   result = f_getlabel(f_getcurrdrive(),label,12);

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

6.4.13 f_get_oem

This returns the OEM name in the disk boot record. The pointer passed for storage should be capable of holding a 8-character string.

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

Parameter	Description
drivenum	Drive number.
str	Pointer to copy label to.
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[9];
   int result;

   oem_name[8]=0;/* zero terminate string */
   result = f_get_oem(f_getcurrdrive(),oem_name,8);

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

6.4.14 f_createpartition

This function is used to create one or more partitions on a drive. It is called with a pointer to the function that must be called to retrieve drive configuration information.

This function may also be used to remove partitions by overwriting the current partition table.

If only a single volume is required then it is simpler not to use a partition table and use f_initvolume to format.

Calling this function will logically destroy all data on the drive.

The number of sectors on the target drive can be found by calling the

driver->getphy(driver,&phy). This information can be used to build the F_PARTITION structure before f_createpartition is called.

F_PARTITION structure is defined as

```
typedef struct
{
   unsigned long secnum;              /* number of sectors in this partition */
   unsigned char system_indicator;    /* use F_SYSIND_XX values*/
} F_PARTITION;
```

In this descriptor **secnum** is the number of sectors in the partition, system_indicator value depends on the format that will be used in the partition. See **F_SYSIND_xx** values in **fat.h**.

This function works similarly to the MS-DOS Fdisk function.

```
int f_createpartition(
                F_DRIVER        *driver,
                int             parnum,
                F_PARTITION     *par
)
```

Parameter	Description
driver	Initialized driver (get from f_createdriver).
parnum	Number of partition is in par ptr.
par	Partition pointer points to partition descriptor.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
static F_PARTITION par2[2]=
{
   {1000, F_SYSIND_DOSFAT16UPTO32MB},
   {2000, F_SYSIND_DOSFAT16UPTO32MB}
};

F_DRIVER *hdd;

int mypartitiondrive()
{
   int ret;

   ret=f_createdriver(&hdd,f_hdddrvinit,0);
   if (ret) return ret;

   ret=f_createpartition(driver,2,par2);
   if (ret) return ret;

   return ret;
}
```

6.4.15 f_getpartition

This function gets the used sectors and system indication byte from a partitioned medium.

For drives which do not contain a partition table, this function returns with the number of sectors and 0 in the system indication byte.

If there is a partition table, then it collects information from the partition table entries. If space in the **F_PARTITION** table is insufficient, then f_getpartition signals F_ERR_MEDIATOOLARGE error. In this case the medium has more partition table entries than the number of entries passed by the **F_PARTITION** table structure, so the caller should increase the number of entries in this table.

F_PARTITION structure is defined as

```
typedef struct
{
   unsigned long secnum;             /* number of sectors in this partition */
   unsigned char system_indicator;   /* use F_SYSIND_XX values*/
} F_PARTITION;
int f_getpartition(
                  F_DRIVER         *driver,
                  int              parnum,
                  F_PARTITION      *par
)
```

Parameter	Description
driver	Initialized driver (get from f_createdriver).
parnum	Number of entry in the par parameter.
par	Partition pointer to retrieve information inside.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
static F_PARTITION par10[10];

int mypartitionlist(F_DRIVER *driver)
{
   int par;
   int ret=f_getpartition(driver,10 ,par10);
   if (ret) return ret; /* error */
   for (par=0; par<10; par++)
   {
      printf ("%d par - %d sys_ind %d sectors\n",
         par, par[10].secnum, par10[par].system_indicator);
   }
   return 0;
}
```

6.5 Drive/Directory Handler Functions

6.5.1 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();
      .
      .
}
```

6.5.2 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.
Error Code	Error condition.

Example

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

6.5.3 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.
Error Code	Error condition.

Example

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

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

6.5.4 f_wgetcwd

Get current **Unicode16** 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.
Error Code	Error condition.

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”)
   }
}
```

6.5.5 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.
Error Code	Error condition.

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”)
   }
}
```

6.5.6 f_wgetdcwd

Get current **Unicode16** 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.
Error Code	Error condition.

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”)
   }
}
```

6.5.7 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.
Error Code	Error condition.

Example

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

6.5.8 f_wmkdir

Make a new directory with a **Unicode16** 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.
Error Code	Error condition.

Example

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

6.5.9 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.
Error Code	Error condition.

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 */
      .
      .
}
```

6.5.10 f_wchdir

Change current working directory with **Unicode16** name.

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

Parameter	Description
dirname	New **Unicode16** directory name to change.

Return Value	Condition
F_NO_ERROR	Directory has been changed successfully.
Error Code	Error condition.

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 */
      .
      .
}
```

6.5.11 f_rmdir

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

If a directory is read-only then this function returns an error code.

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

Parameter	Description
dirname	Directory name to remove.

Return Value	Condition
F_NO_ERROR	Directory is removed successfully.
Error Code	Error condition.

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 */
      .
      .
}
```

6.5.12 f_wrmdir

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

If a directory is read-only then this function returns an error code.

```
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.
Error Code	Error condition.

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 */
      .
      .
}
```

6.6 File Functions

6.6.1 f_rename

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

If a file or directory is read-only it cannot be renamed. If a file is already open it cannot be renamed.

```
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.
Error Code	Error condition.

Example

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

6.6.2 f_wrename

Rename a file or directory with **Unicode16** name. This function has been obsoleted by f_wmove.

If a file or directory is read-only it cannot be renamed. If a file is already open it cannot be renamed.

```
int f_wrename(
             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.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

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

6.6.3 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.
Error Code	Error condition.

Example

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

6.6.4 f_wmove

Moves a file or directory with a **Unicode16** 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.
Error Code	Error condition.

Example

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

6.6.5 f_delete

Delete a file.

A read-only or open file cannot be deleted.

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

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

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

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

6.6.6 f_wdelete

Delete a file with **Unicode16** name.

A read-only or open file cannot be deleted.

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

Parameter	Description
filename	**Unicode16** file name with/without path to be deleted

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

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

6.6.7 f_filelength

Get the length of a file.

This function can also return with the opened file's size when **_f_findopensize** is allowed to search for it. If **_f_findopensize** returns always with zero, then this feature is disabled.

```
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;
}
```

6.6.8 f_wfilelength

Get the length of a file with **Unicode16** name.

This function can also return with the opened file's size when **_f_findopensize** is allowed to search for it. If **_f_findopensize** returns always with zero, then this feature is disabled.

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

Parameter	Description
filename	**Unicode16** 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;
}
```

6.6.9 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.

Files with the system attribute set will be ignored.

If this is called with "*.*" and this is not the root directory, the first entry found will be "." - the current directory.

```
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.
Error Code	Error condition.

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));
   }
}
```

6.6.10 f_wfindfirst

Find first file or subdirectory in specified directory with **Unicode16** name. First call f_wfindfirst function and if file was found get the next file with f_wfindnext function.

Files with the system attribute set will be ignored.

If this is called with "*.*" and this is not the root directory, the first entry found will be "." - the current directory.

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

Parameter	Description
filename	**Unicode16** name of file to find.
find	Where to store find information.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

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));
   }
}
```

6.6.11 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.

Files with the system attribute set will be ignored.

If this is called with "*.*" and this is not the root directory, the first entry found will be "." - the current directory.

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

Parameter	Description
find	Find structure (from f_findfirst).

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

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));
   }
}
```

6.6.12 f_wfindnext

Find the next file or subdirectory in a specified directory with **Unicode16** 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.

Files with the system attribute set will be ignored.

If this is called with "*.*" and this is not the root directory, the first entry found will be "." - the current directory.

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

Parameter	Description
find	Find structure (from f_findfirst).

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

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));
   }
}
```

6.6.13 f_settimedate

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

```
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.
Error Code	Error condition.

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);
}
```

6.6.14 f_wsettimedate

Set time and date on a file or on a directory with **Unicode16** name.

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

Parameter	Description
filename	**Unicode16** name of file.
ctime	Creation time of file or directory.
cdate	Creation date of file or directory.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

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);
}
```

6.6.15 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.

```
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.
Error Code	Error condition.

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!"
   }
}
```

6.6.16 f_wgettimedate

Get time and date information from a file or directory with a **Unicode16** name. This field is automatically set by the system when a file or directory is created and when a file is closed.

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

Parameter	Description
filename	**Unicode16** 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.
Error Code	Error condition.

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!"
  }
}
```

6.6.17 f_getattr

This routine is used to get the attributes of a specified file. Possible file attribute settings are defined by the FAT file system:

```
   F_ATTR_ARC           Archive
   F_ATTR_DIR           Directory
   F_ATTR_VOLUME        Volume
   F_ATTR_SYSTEM        System
   F_ATTR_HIDDEN        Hidden
   F_ATTR_READONLY      Read Only
int f_gettattr(
             const char      *filename,
             unsigned char   *attr
)
```

Parameter	Description
filename	Name of target file or directory.
attr	Pointer to place attribute setting .

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void myfunc(void)
{
   unsigned char attr;

   /* find if myfile is read only */

   if(!f_getattr("myfile.txt",&attr)
   {
      if(attr & F_ATTR_READONLY)
         printf("myfile.txt is read only");
      else
         printf("myfile.txt is writable");
   }
   else
   {
      printf("file not found");
   }
}
```

6.6.18 f_wgetattr

This routine is used to get the attributes of a specified **Unicode16** file. Possible file attribute settings are defined by the FAT file system:

```
   F_ATTR_ARC           Archive
   F_ATTR_DIR           Directory
   F_ATTR_VOLUME        Volume
   F_ATTR_SYSTEM        System
   F_ATTR_HIDDEN        Hidden
   F_ATTR_READONLY      Read Only
int f_gettattr(
             const char      *filename,
             unsigned char   *attr
)
```

Parameter	Description
filename	**Unicode16** name of target file or directory.
attr	Pointer to place attribute setting .

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void myfunc(void)
{
   unsigned char attr;

   /* find if myfile is read only */

   if(!f_getattr("myfile.txt",&attr)
   {
      if(attr & F_ATTR_READONLY)
         printf("myfile.txt is read only");
      else
         printf("myfile.txt is writable");
   }
   else
   {
      printf("file not found");
   }
}
```

6.6.19 f_setattr

This routine is used to set the attributes of a specified file. Possible file attribute settings are defined by the FAT file system:

```
   F_ATTR_ARC           Archive
   F_ATTR_DIR           Directory
   F_ATTR_VOLUME        Volume
   F_ATTR_SYSTEM        System
   F_ATTR_HIDDEN        Hidden
   F_ATTR_READONLY      Read Only
int f_settattr(
             const char      *filename,
             unsigned char   *attr
)
```

Parameter	Description
filename	Name of target file or directory.
attr	New attribute setting.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void myfunc(void)
{

   /* make myfile read only and hidden */

   f_setattr("myfile.txt", F_ATTR_READONLY | F_ATTR_HIDDEN);
}
```

6.6.20 f_wsetattr

This routine is used to set the attributes of a specified **Unicode16** file. Possible file attribute settings are defined by the FAT file system:

```
   F_ATTR_ARC            Archive
   F_ATTR_DIR            Directory
   F_ATTR_VOLUME         Volume
   F_ATTR_SYSTEM         System
   F_ATTR_HIDDEN         Hidden
   F_ATTR_READONLY       Read Only
int f_wsettattr(
             const W_CHAR    *filename,
             unsigned char   *attr
)
```

Parameter	Description
filename	**Unicode16** name of target file or directory.
attr	New attribute setting .

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

Example

```
void myfunc(void)
{

   /* make myfile read only and hidden */

   f_setattr("myfile.txt", F_ATTR_READONLY | F_ATTR_HIDDEN);
}
```

6.6.21 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.

This function can also return with the opened file's current size when **_f_findopensize** is allowed to search through all open file descriptors for its modified size. If this feature is disabled then **_f_findopensize** returns always zero.

```
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);
}
```

6.6.22 f_wstat

Get information about a file with a **Unicode16** 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 **Unicode16** 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);
}
```

6.7 Read/Write Functions

6.7.1 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);
}
```

6.7.2 f_wopen

Opens a file with **Unicode16** file name.

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

Parameter	Description	
filename	**Unicode16** 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);
}
```

6.7.3 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.
Error Code	Error condition.

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”);
}
```

6.7.4 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.

```
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);
}
```

6.7.5 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;
}
```

6.7.6 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.
Error Code	Error condition.

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;
}
```

6.7.7 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;
}
```

6.7.8 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;
}
```

6.7.9 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
F_NO_ERROR	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;
}
```

6.7.10 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
F_NO_ERROR	Success.
Error Code	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;
}
```

6.7.11 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;
}
```

6.7.12 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;
}
```

6.7.13 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;
}
```

6.7.14 f_wtruncate

Opens a file with a **Unicode16** 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	**Unicode16** 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;
}
```

6.7.15 f_flush

Flushes an open file to disk. This is logically equivalent to doing a close and open on a file to ensure the data changed before the flush is committed to the disk.

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

Parameter	Description
filehandle	File handle of target.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

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”);
}
```

6.7.16 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;
}
```

6.7.17 f_getlasterror

It returns with the last error code. Last error code is cleared/changed when any API function is called.

```
int f_getlasterror()
```

Parameter	Description
none	

Return Value	Condition
Last error code	

Example

```
int myopen()
{
   F_FILE *file;
   file=f_open("nofile.tst","rb");
   if (!file)
   {
      int rc=f_getlasterror();
      printf ("f_open failed, errorcode:%d\n",rc);
      return rc;
   }

   return F_NO_ERROR;
}
```

6.8 HCC Function Error Codes

Error	Value	Description
F_NO_ERROR	0	Success
F_ERR_INVALIDDRIVE	1	The specified drive does not exist
F_ERR_NOTFORMATTED	2	The specified volume has not been formatted
F_ERR_INVALIDDIR	3	The specified directory is invalid
F_ERR_INVALIDNAME	4	The specified file name is invalid
F_ERR_NOTFOUND	5	The file or directory could not be found
F_ERR_DUPLICATED	6	The file or directory already exists
F_ERR_NOMOREENTRY	7	The volume is full
F_ERR_NOTOPEN	8	The file access function requires the file to be open.
F_ERR_EOF	9	End of file
F_ERR_RESERVED	10	Not used
F_ERR_NOTUSEABLE	11	Invalid parameters for **f_seek**
F_ERR_LOCKED	12	The file has already been opened for writing/appending.
F_ERR_ACCESSDENIED	13	The necessary physical read and/or write functions are not present for this volume
F_ERR_NOTEMPTY	14	The directory to be renamed or deleted is not empty.
F_ERR_INITFUNC	15	If no init function available for a driv0er or the function generates an error.
F_ERR_CARDREMOVED	16	The card has been removed.
F_ERR_ONDRIVE	17	Non-recoverable error on drive
F_ERR_INVALIDSECTOR	18	A sector has developed an error.
F_ERR_READ	19	Error reading the volume
F_ERR_WRITE	20	Error writing file to volume
F_ERR_INVALIDMEDIA	21	Media not recognized
F_ERR_BUSY	22	The caller could not obtain the semaphore within the expiry time
F_ERR_WRITEPROTECT	23	The physical medium is write protected
F_ERR_INVFATTYPE	24	The type of FAT is not recognized
F_ERR_MEDIATOOSMALL	25	Media is too small for the format type requested
F_ERR_MEDIATOOLARGE	26	Media is too large for the format type requested
F_ERR_NOTSUPPSECTORSIZE	27	The sector size is not supported. The only supported sector size is 512 bytes.
F_ERR_UNKNOWN	28	Unspecified error has occurred
F_ERR_DRVALREADYMNT	29	The drive is already mounted
F_ERR_TOOLONGNAME	30	The name is too long
F_ERR_RESERVED_1	31	Reserved
F_ERR_DELFUNC	32	The delete drive driver function failed

Error	Value	Description
F_ERR_ALLOCATION	33	Malloc failed to allocate required memory
F_ERR_INVALIDPOS	34	An invalid position is selected
F_ERR_NOMORETASK	35	All task entries are exhausted
F_ERR_NOTAVAILABLE	36	The called function is not supported by the target volume
F_ERR_TASKNOTFOUND	37	The caller's task identifier was not registered - normally because the **f_enterfs()** function has not been called.
F_ERR_UNUSABLE	38	The file system has become unusable - normally as a result of excessive error rates on the underlying media.

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 FAT 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 FATFS 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 FAT 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
);
```

<table>
<tr><th>Parameter</th><th colspan="2">Description</th></tr>
<tr><td><b>dir</b></td><td colspan="2">SDD directory descriptor.</td></tr>
<tr><td><b>name</b></td><td colspan="2">Path of the new file or directory relative to the parameter <b>dir</b>.</td></tr>
<tr><td rowspan="4"><b>type</b></td><td colspan="2">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:</td></tr>
<tr><td><b>Value</b></td><td><b>Description</b></td></tr>
<tr><td>SFS_ATTR_DIR</td><td>Defines an SDD directory object.

SDD_OBJ_TYPE | SDD_MAN_TYPE | SFS_DIR_TYPE</td></tr>
<tr><td>SFS_ATTR_FILE</td><td>Defines an SDD file object.

SDD_OBJ_TYPE | SDD_DEV_TYPE | SDD_FILE_TYPE</td></tr>
<tr><td><b>mode</b></td><td colspan="2">0 (reserved for later use)</td></tr>
</table>

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 FAT File System Block Diagram

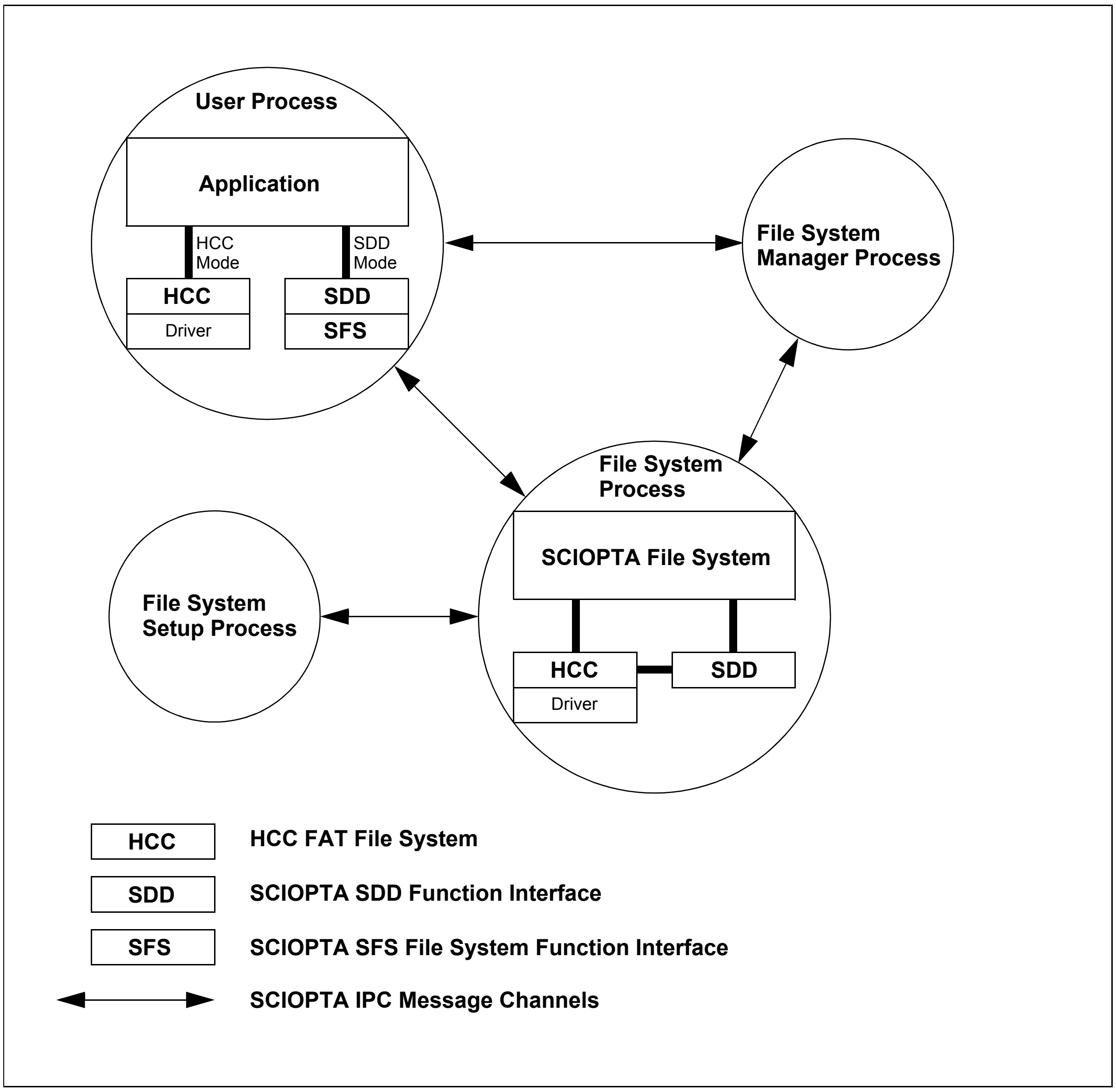

User Process
Application
HCC Mode
SDD Mode
HCC
Driver
SDD
SFS
File System Manager Process
File System Process
SCIOPTA File System
HCC
Driver
SDD
File System Setup Process
HCC
HCC FAT File System
SDD
SCIOPTA SDD Function Interface
SFS
SCIOPTA SFS File System Function Interface
SCIOPTA IPC Message Channels

Figure 9-1: SCIOPTA FAT File System Block Diagram

9.4 File System Process

The file system process includes the main SCIOPTA FAT File System initialization functions. It is using the HCC FAT File System (HCC FAT API), the HCC FAT 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 FAT File System you need to include some HCC FAT Flash File System source files and SCIOPTA integration files in your project.

9.4.1 HCC FAT File System

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

common.c	HCC common functions.
fat.c	HCC FAT short filename functions.
fat_lfn.c	HCC alternative source file to fat.c for long filenames.
fat_m.c	HCC FAT file system reentrancy wrapper.
port.c	Ported functions

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

The HCC FAT File System files can be found at:

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

9.4.2 SCIOPTA - HCC Integration

These files include the SCIOPTA integration in the HCC FAT File System.

sc2fatfs.c	HCC FAT FS 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 FATFS Function Interface Reference” on page 8-1.

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

For the “fatfs_ram” SCIOPTA FAT File System getting started example the following file contains the flash file system setup:

Files

fs_setup.c	Example file system setup process

File location<installation_folder>\sciopta\<version>\exp\sfs\common\fatfs_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 FAT File System.

9.7.1 HCC Mode

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

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

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

For the “fatfs_ram” SCIOPTA FAT File System getting started example the following file contains the HCC Mode FAT File System user process:

Files

hcc_fatfstest.c	Example FAT file system process (HCC Mode)

File location<installation_folder>\sciopta\<version>\exp\sfs\common\fatfs_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 File System) functions. These functions send and receive messages to and from the file system process for accessing the HCC FAT 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 FATFS Function Interface Reference” on page 8-1).

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

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

Files

fatfstest.c	Example file system process (SDD Mode)

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

9.8 Implementing FAT File System Drivers

Please consult chapter 11.21 "FAT File System Drivers" on page 11-35 for information about SCIOPTA FAT 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 Requirements

9.9.1 System Requirements

The SCIOPTA FAT 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 FAT 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.

9.9.2 Stack Requirements

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. However, if long filenames are used then the stack size should be increased to 4KB; see the Long Filenames section below.

9.9.3 Real-Time Requirements

The bulk of the file system is code that executes without delay. There are exceptions at the driver level, where delays in reading and writing from/to the physical media, and in the communication itself, cause the system to wait on external events. The points at which delays occur are documented in the applicable driver sections. The developer should modify the drivers to meet the system's requirements, either by implementing interrupt control of the relevant events or scheduling other parts of the system that can proceed without completion of the events. Read the relevant driver sections for details.

9.10 Unicode Support

Please consult the configuration chapter 10.4.5 “Unicode” on page 10-7 for information about unicode support.

9.11 Drives, Partitions and Volumes

FAT provides functions for creating and managing multiple drives, partitions and volumes.

Some definitions:

- A drive consists of a physical medium which is controlled by a single driver. Examples are an HDD or a Compact Flash Card.
- All drives contain zero or more partitions. If the drive is not partitioned, then there is just a single volume on that drive. Removable media such as flash cards have no partitions or one partition on the card.
- To each partition may be added a single volume. A volume can exist on a drive without partitions.

The file system operates on volumes. You can have one volume or a set of volumes; additional functions are provided to work with the set of volumes if it's greater than one (A:, B:, C:, etc.)

The API function calls f_getdrive, f_chdrive and f_getcwd, refer to drives by name, because this is the convention, but the names are really references to volumes. If there's no requirement to create or delete partitions, then these functions should not be included in the system:

- f_initvolumepartition
- f_createdriver
- f_releasedriver
- f_createpartition

If multiple partitions are to be used then these four functions are used to create drivers for partitioned drives and to create partitions on those drives. Partitions are created on a single volume, such as an HDD, and so a single driver is used to access the volume even though there are multiple partitions on it. These volumes need to be controlled by a single lock.

Some operating systems will not recognize multiple partitions on removable media. It is therefore "normal" to restrict the use of multiple partitions to fixed drives. FAT-created partitions are Windows XP compatible.

9.12 Long File names

Please consult the configuration chapter 10.4.1 “Long File Names” on page 10-6 for information about long file names.

9.13 Maximum Tasks and CWD

SCIOPTA has integrated the HCC FAT File Systems such that more than a single task is allowed to access the file system. Reentrancy and maintenance of the current working directory (CWD) are handled by SCIOPTA.

9.14 Get Time

For the system to be compatible with other systems, it is necessary to provide a real-time function so that files can be time-stamped.

An empty function (**f_gettime**) is provided in **port.c**. It should be modified by the developer to provide the time in standard format.

The required format for the time for PC compatibility is a short integer 't' (16 bit) such that:

```
2-second increments (0-30 valid)   (t & 0x001f)
minute              (0-59 valid)   ((t & 0x07e0) >> 5)
hour                (0-23 valid)   ((t & 0xf800) >> 11)
```

9.15 Get Date

For the system to be compatible with other systems it is necessary to provide a real time function so that files can be date-stamped.

An empty function (**f_getdate**) is provided in **port.c**. It should be modified by the developer to provide the date in standard format.

The required format for the date for PC compatibility is a short integer 'd' (16 bit) such that:

```
day               (0-31)          (d & 0x001f)
month             (1-12 valid)    ((d & 0x01e0) >> 5)
years since 1980  (0-119 valid)   ((d & 0xfe00) >> 9)
```

9.16 Random Number

The **port.c** file contains a function (**f_getrand**) that the file system uses to get a pseudo-random number to use as the volume serial number. This function is required only if a hard-format of devices is required.

It is recommended that the developer replace this routine with a random function from the base system, or alternatively generate a random number based on a combination of the system time/date and a system constant such as a MAC address.

9.17 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\

9.18 FAT Drive Format

This document does not describe a FAT file system in detail. There are many reference works to choose from for this purpose. This file system handles the majority of the features of a FAT file system with no need for the developer to understand further. However, there are some areas where an understanding may help. This section describes these features and provides additional information about FAT formats.

There are three different forms in which your removable media may be formatted:

- Completely unformatted media
- Master Boot Record
- Boot sector information only

The sections below describe how the system handles these three situations.

9.18.1 Completely unformatted

An unformatted drive is not useable until it has been formatted. Most flash cards are preformatted whereas hard disk drives tend to be unformatted when delivered.

The format of the card is determined by the number of sectors on it. Information about the connected device is given to the system from the **xxx_getphy** call to the driver from which the number of available clusters on the device is calculated.

When the **f_format** function is called the drive will be formatted with Boot Record In-formation or **xxx_getphy**.

If more partitions are required, then use the **f_createpartition**, **f_initvolumepartition** and **f_format** functions for formatting.

9.18.2 Master Boot Record

If a card contains a Master Boot Record (MBR), then it is formatted as in the tables be-low. Function **f_createpartition** also can create an MBR.

When a device is inserted with an MBR it will be treated as if it has just one partition (the first in the partition table if **f_initvolume** is used. Multiple partitions can be initialized by **f_initvolumepartition** function.

Offset	Bytes	Entry Description	Value/Range
`0x0`	446	Consistency check routine	
`0x1be`	16	Partition table entry.	(table below)
`0x1ce`	16	Partition table entry.	(table below)
`0x1de`	16	Partition table entry.	(table below)
`0x1ee`	16	Partition table entry.	(table below)
`0x1fe`	1	Signature.	0x55
`0x1fe`	1	Signature.	0xaa

Table 10: Master Boot Record

Offset	Bytes	Entry Description	Value/Range
`0x0`	1	Boot descriptor.	0x00 (non-bootable device) 0x80 (bootable device)
`0x1`	3	First partition sector.	Address of first sector
`0x4`	1	File system descriptor.	0 = empty 1 = FAT12 4 = FAT16 < 32MB 5 = Extended DOS 6 = FAT16 >= 32MB 0xB=FAT32 0x10-0xff free
`0x5`	3	Last partition sector.	Address of last sector
`0x8`	4	First sector position relative to device start.	First sector number
`0xc`	4	Number of sectors in partition.	Between 1 and max number on device

Table 11: Partition Entry Description

9.18.3 Boot Sector Information

This is arrangement is used as standard by the file system. The first 36 bytes of the boot sector are given in the first table. This group is the same for FAT12/16/32. The second table shows the format for the rest of the boot sector for FAT12/16. The third table shows the format of the boot sector for FAT32.

Offset	Bytes	Entry Description	Value/Range
`0x0`	3	Jump Command	0xeb 0xXX 0x90
`0x3`	8	OEM name	XXX: specify in udefs.h
`0xb`	2	Bytes/Sector	512
`0xd`	1	Sectors/Cluster	XXX(1-64)
`0xe`	2	Reserved sectors	1
`0x10`	1	Number of FATs	2
`0x11`	2	Number of root directory entries	512
`0x13`	2	Number of sectors on media	XXX (depends on card size; if greater than 65535 then 0 and the total number of sectors is used)
`0x15`	1	Media Descriptor	0xf8 (hard disk), 0xf0 (removable media)
`0x16`	2	Sectors/FAT16	XXX (normally 2). This must be zero for FAT32.
`0x18`	2	Sectors/Track	32 (not relevant)
`0x1a`	2	Number of heads	2 (not relevant)
`0x1c`	4	Number of hidden sectors	0 or if MBR present number relative sector offset of this sector.
`0x20`	4	Number of total sectors	XXX (depends on card size) or 0

Table 12: Boot Sector Information, Table First 36 bytes

Offset	Bytes	Entry Description	Value/Range
`0x24`	1	Drive Number	0
`0x25`	1	Reserved	0
`0x26`	1	Extended boot signature	0x29
`0x27`	4	Volume ID or Serial Number	Random number generated at format
`0x2b`	11	Volume Label	"NO LABEL" is put here by a format
`0x36`	8	File System type	"FAT16" or "FAT12"
`0x3e`	448	Load Program Code	Filled with zeroes.
`0x1fe`	1	Signature	0x55
`0x1ff`	1	Signature	0xaa

Table 13: Boot Sector Information Table, FAT12/16 after byte 36

The serial number field is generated by the random number function - see porting section for information about its generation.

Offset	Bytes	Entry Description	Value/Range
`0x24`	4	Sectors/FAT32	The number of sectors in one FAT
`0x28`	2	ExtFlags	Always zero.
`0x2a`	2	File System Version	0
`0x2c`	4	Root Cluster	Cluster number of the first cluster of the root directory
`0x30`	2	File System Info	Sector number of FSINFO structure in the re-served area of the FAT32. Usually 1.
`0x32`	2	Backup Boot Sector	If non-zero it indicates the sector number in the reserved area of the volume of a copy of the boot record. Usually 6.
`0x34`	12	Reserved	All bytes always zero
`0x40`	1	Drive Number	0
`0x41`	1	Reserved	0
`0x42`	1	Boot Signature	0x29
`0x43`	4	Volume ID	Random number generated at format.
`0x47`	11	Volume Label	"NO LABEL" is put here by a format
`0x52`	8	File System Type	Always set to string "FAT32".

Table 14: Boot Sector Information, FAT32 after byte 36

9.19 Using CheckDisk

This section describes the usage of the f_checkdisk utility.

FAT file systems were not designed to be failsafe; i.e., they were not designed in such a way that if power is lost unexpectedly they will always be reconstructed in a clean state. Several types of errors may occur, such as loss of chains, or lost directory entries. This utility is designed to correct all errors that can occur from unexpected power loss when using FAT. Note that if the media are used in a device with a different FAT implementation, then it may not be possible to correct all errors.

This utility must be used as stand-alone; i.e., no other application should be accessing the file system while it is running.

Often a check-disk operation can be performed by more powerful devices such as desktop computers and in this case it is normal to omit the check-disk files from the build. How-ever, if there are non-removable media then the f_checkdisk utility should be included in the build.

To include the f_checkdisk utility in your project add the following files to your build:

Include Files

chkdsk.h	Check disk defines.

File location: <install_folder>\sciopta\<version>\sfs\hcc\fatfs\

Source Files

chkdsk.c	Check disk source file.

File location: <install_folder>\sciopta\<version>\sfs\hcc\fatfs\

To use check disk the system must have defined **USE_MALLOC**. This is necessary because with removable media the size of the table required for check disk can vary a lot. This memory is required only for the duration of the check disk process.

9.19.1 Build Options

For CheckDisk operation these settings needed to be revised:

CHKDSK_LOG_ENABLE

This option should be enabled in **chkdsk.h** if you want to generate a log file for the actions of **f_checkdisk**. This is recommended.

CHKDSK_LOG_SIZE

This specifies the maximum size in RAM to be used for storing CheckDisk log information.

9.19.2 f_checkdisk

This function checks the state of the attached media and automatically fixes errors that have been detected. It can create a log file of its findings.

```
int f_checkdisk(
             int            drivenum,
             int            param
)
```

Parameter	Description	
drivenum	Number of drive to be checked.	
param		
	CHKDSK_ERASE_BAD_CHAIN	The function will automatically erase all bad chains found. Otherwise the file with the bad chain will be terminated at the last good cluster.
	CHKDSK_ERASE_LOST_CHAIN	The function will automatically erase all lost chains found. Otherwise a LOSTxxxx file will be created with the files contents.
	CHKDSK_ERASE_LOST_BAD_CHAIN	The function will automatically erase all bad lost chains. Otherwise a LOSTxxxx file will be created and this file will be terminated at the last good cluster.

Return Value	Condition
F_NO_ERROR	Completed successfully.
FC_WRITE_ERROR	Unable to write a sector.
FC_READ_ERROR	Unable to read a sector.
FC_CLUSTER_ERROR	Unable to access a cluster in the FAT.
FC_ALLOCATION_ERROR	Memory allocation failed.

Example

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

   /* check drive 0 ("A") */
   if(ret=f_checkdisk(0, 0)
   {
      printf("Check Disk Failed: error %d\n",ret);
   }
   else
   {
      printf("Check Disk Finished\n");
   }
   .
   .
   .
}
```

9.19.3 Memory Requirements

The f_checkdisk utility requires memory to run. This is typically 1K of static memory (0.5K if logging is disabled) and 1.5K of stack.

Additionally two blocks must be allocated dynamically (using malloc). The sizes of the blocks are approximately:

```
   (NUMBER_OF_CLUSTERS+4096) / 8

               and

      512 + CHKDSK_LOG_SIZE
```

The second of these is not required if logging is not enabled. **CHKDSK_LOG_SIZE** is defined in **chkdsk.h**. The number of clusters on a device can be very large. It depends on how the device is formatted (number of sectors per cluster) and the size of the device. The number of clusters on a device can be approximated as:

```
   (SIZE_OF_MEDIA) / (512 * SECTORS_PER_CLUSTER)
```

The number of sectors per cluster is always in the range 2^n where 0 <= n < 7.

9.19.4 Log File Entries

Each time the f_checkdisk utility is run a log file is generated if enabled. The following messages may appear in the log file:

`Directory: <directory_path>`

Displays directory where error messages below have been found.

`Directory entry deleted: <name>`

Either a file entry or a directory entry has been deleted from this directory

`Lost entry deleted (found in a subdirectory):/ <LOSTxxxx>`

The named lost directory or file entry has been recovered.

`Entry deleted (reserved/bad cluster): <name>`

The first cluster in a directory entry is unusable or there is a bad element in the chain and **CHKDSK_ERASE_BAD_CHAIN** is set.

`File size changed: <name> < old_size> <new_size>`

A file was found whose size is smaller than the minimum number of clusters needed to store that file, or the file size is greater than that which can be stored in the cluster chain. The file size has been changed to the maximum for the clusters allocated to that file. The user should analyze this file to find the correct termination point.

`Start cluster changed: <name> (either "." or "..")`

An invalid cluster has been found in a directory entry for either "." or "..". This has been fixed.

`Entry deleted (cross linked chain): <name>`

If the start cluster of the named file is cross-linked or if any subsequent cluster is cross-linked and **CHKDSK_ERASE_BAD_CHAIN** is set then this message will give the name of the removed file.

`Lost directory chain saved: <LOSTxxxx>`

A directory chain with no references has been found. It has been recreated with the name LOSTxxxx.

`Lost file chain saved: <LOSTxxxx>`

A file chain with no references has been found. It has been recreated in the root directory with the name LOSTxxxx.

`Lost chain removed (first cluster/cnt): <cluster> <count>`

A lost chain has been discovered and removed. This will appear only if **CHKDSK_ERASE_LOST_CHAIN** or **CHKDSK_ERASE_LOST_BAD_CHAIN** is enabled. If not, a LOSTxxxx file will be created.

`Last cluster changed (bad next cluster value): <name>`

In checking the file chain, an invalid cluster was discovered. The cluster prior to the bad cluster is changed to end of file and the file size is adjusted to the maxi-mum for the new size of cluster chain.

`Moving lost directory: /<LOSTxxxx>`

A lost directory has been recovered.

`'..' changed to root: <LOSTxxxx>`

A lost directory entry has been placed in the root so its '..' entry has been changed to point to the root.

`FAT2 updated according to FAT1.`

FAT1 and FAT2 were found to be different and FAT1 is used as the correct version. This can appear only once at the beginning of the log file.

`Long filename entry/entries removed. Count=`

This appears at the end of the log file and is a count of the number of long file-name entries that were invalid and unrecoverable.

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\fatsf_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 FAT File System Setup and Configuration

You need to write a process **SCP_fatfs** (declared with the process name **SCP_fatfs)**. The only function of this process is to call the flash file system function **fatfs_process** including the configuration parameter.

```
void fatfs_process (
             const char      *FSNAME
);
```

Parameter	Description
FSNAME	Name of file system to register at file system manager.

Return Value	Condition
None	

Example

```
SC_PROCESS(SCP_fatfs)
{
  sc_poolid_t pl;

  pl = sc_poolIdGet("fs_pool");
  if ( pl != SC_ILLEGAL_POOLID ){
    sc_poolDefault(pl);
  }

  fatfs_process(FS_NAME);
}
```

10.4 HCC FAT File System User Configuration

The following configuration definitions and settings are included in the file **udefs.h**.

Include Files

udefs.h	HCC FAT File System definitions.

File location: <install_folder>\sciopta\<version>\include\hcc\fatfs\

10.4.1 Long File Names

The system includes two main source files:

fat.c is the file system without long filename support. If long filenames exist on the media the system will ignore the long name part and use only the short name.

fat_lfn.c is the file system with complete long filename support.

Because of the increase in system resources required to handle long filenames, they should be used only when necessary. This will avoid increasing the stack sizes of applications that call the file system, as well as an increase in the amount of checking that's required.

To choose between using the long filename version and the short use the

`F_LONGFILENAME definition in` **`udefs.h.`**

The maximum long filename space required by the standard is 260 bytes. As a consequence each time a long filename is processed, large areas of memory must be available. The developer may, depending on the application, reduce the size of **F_MAXPATH** and **F_MAXLNAME** (in **udefs_f.h**) to reduce the resources used by the system. The structure **F_LFNINT** must NOT be modified as this is used to process the files on the media which may be created by other systems.

The most critical function for long filenames is **fn_rename**, which must keep two long filenames on the stack, with additional structures for handling it. If this function is not required for your application it is sensible to comment it out. This can reduce the stack requirements significantly (by approximately 1K).

10.4.2 Maximum Number of Volumes

The maximum number of volumes allowed by your system should be set in the **F_MAXVOLUME** definition in **udefs.h**. Set this value to the maximum number of volumes that will be available on the target system. If only a RAM drive is used, set the value to 1; if a RAM drive and a CF card drive, then set this value to 2, and so on.

Volumes are given drive letters as specified in the f_initvolume function.

The system is designed so that access to each volume is entirely independent; i.e., if an operation is being performed on a volume, then it does not block access to other volumes.

10.4.3 Maximum Open Files

The maximum number of simultaneously open files must be specified in the **udefs.h** file. This is set in the **F_MAXFILES** definition. It is the total number of files that may be simultaneously open across all volumes.

Please consult also chapter 11.21 “FAT File System Drivers” on page 11-35.

10.4.4 Separator Character

The **udefs_f.h** file contains a definition **F_SEPARATORCHAR** that allows the selection of the forward slash or back slash as a separator in file paths.

10.4.5 Unicode

Support for 16-bit Unicode is provided. Unicode 7/8 are supported by the file system transparently. The option described here is required only for **Unicode16** support.

To support **Unicode16** character sets the developer must uncomment the line in the file **udefs.h**:

```
/* #define HCC_UNICODE */
```

This will force any build to include the **Unicode16** API. This build will also force Long Filename support (see next section), which is necessary for **Unicode16** support. With this build you may now use the **Unocode16** API calls.

Use of **Unicode16** implies that the host system has wchar ("wide character") support or an equivalent definition.

Using **Unicode16** creates additional resource requirements because all string and path accesses effectively use twice the space. Therefore it is recommended that this option be used only if it is a requirement of your system to use **Unicode16**.

To allow the file system to generate consistent short filenames the user may want to include character set conversion tables in the code. There are two points in the code where you must insert this conversion if required - these are in the **_f_createlfn()** function in the **fat_lfn.c** module and marked with the comment:

```
/* here we can add ...
```

10.4.6 Sector Sizes

The system allows the developer to specify the maximum sector size of the attached media. Traditionally most FAT based devices have used a sector size of 512 bytes. However for devices whose native sector size is not 512 bytes (e.g. 2K page NAND flash based devices) it is more efficient to use other values. The system allows you to specify the maximum sector size allowed - this is **F_MAX_SECTOR_SIZE** in **udefs_f.h**.

A variable sector size is normally only required in systems which have removable media attached. If a larger than necessary maximum sector size is set, then the file system uses more RAM. Therefore, in resource constrained systems, it may be necessary to restrict the allowed sector size.

If a device is attached with a sector size which is not supported then the error **F_ERR_NOTSUPPSECTORSIZE** is returned.

The cache options (**FATCACHE** and **DIRCACHE**) always allocate buffers of size **F_MAX_SECTOR_SIZE**. If the attached media has a smaller sector size then it will fill the buffer anyway.

Example

If **FATCACHE_READAHEAD** is set to 4 and **F_MAX_SECTOR_SIZE** is 2048; then if a 512 byte sector media is connected the allocated fatcache block will be 4*2048=8192 bytes. The read-ahead size will therefore be 16 sectors of 512 bytes each.

10.4.7 Cache Setup and Options

The system includes two caching mechanisms to enhance performance: FAT caching and write data caching.

10.4.7.1 FAT Caching

FAT caching enables the file system to read several sectors from the FAT in one access so that when accessing the files the file system does not have to read new FAT sectors so frequently. FAT caching is arranged in blocks so that each block can cover different areas of the FAT. The number of sectors that each block contains and the number of blocks are configurable.

FAT caching requires 512 additional bytes of RAM per sector.

The following definitions are provided in **udefs.h**

```
#define FATCACHE_ENABLE

#ifdef FATCACHE_ENABLE
#define FATCACHE_BLOCKS 4       /*number of different FAT cache blocks*/
#define FATCACHE_READAHEAD 8    /* number of FAT sectors to read */
                                /* to a block */
#define FATCACHE_SIZE (FATCACHE_BLOCKS*FATCACHE_READAHEAD)
#endif
```

Note: The additional RAM required for FAT caching is:

```
FATCACHE_BLOCKS*FATCACHE_READAHEAD*F_MAX_SECTOR_SIZE
```

This default setting requires 16K of additional RAM.

10.4.7.2 Write Caching

The write cache defines the maximum number of sectors that can be written in one op-eration from the caller's data buffer. This is also depends on the availability of contiguous space on the target drive. The write cache requires an F_POS structure (24 bytes) for each entry in the write cache. The main purpose of these structures is to be able to wind back a write in the event of an error in writing.

The default setting for the write caching in **udefs.h** is:

```
#define WR_DATACACHE_SIZE 32
```

This will require 768 additional bytes of RAM.

10.4.7.3 Directory Cache

This can be enabled only if **F_LONGFILENAME** is defined. This can be enabled by de-fining **DIRCACHE_ENABLE** in udefs_f.h. If this is enabled you must specify the number of sectors to read ahead with **DIR_CACHESIZE**. This will allocate this number of sectors of memory for directory caching (e.g., if set to 32 and **F_MAX_SECTOR_SIZE** is 512 then 16Kbytes of memory will be allocated). Note also that the system will never read more than the size of a cluster into this cache; therefore, there is no value in having a **DIR_CACHESIZE** greater than the sectors per cluster of the target device.

10.4.8 FAT Free Cluster Bit Field

In **udefs.h** there is a **FATBITFIELD_ENABLE** definition. If this is enabled then the system will attempt to malloc a block to contain a bit table of free clusters. This table is maintained by the file system and is used to accelerate searches for free clusters. This makes a large difference to the write performance when writing to a large and full disk.

10.4.9 Memcpy and Memset

Supplied with the system are memcpy and memset functions.

It is recommended that you re-define these so that they call versions that are optimized for your target system. As with all embedded systems, these routines are used frequently, and use lots of time; therefore, a good **memcpy** routine can have a large impact on the overall performance of your system.

The following has been defined in **udefs.h** and should be modified to call target-opti-mized versions of these functions:

```
#ifdef INTERNAL_MEMFN
#define _memcpy(d,s,l) _f_memcpy(d,s,l)
#define _memset(d,c,l) _f_memset(d,c,l)
#else
#include <string.h>
#define _memcpy(d,s,l) memcpy(d,s,l)
#define _memset(d,c,l) memset(d,c,l)
#endif
```

10.4.10 Last accessed date

In **udefs.h** there is a #define for automatic updating of the last-accessed time field in the directory entry of read file. Set **F_UPDATELASTACCESSDATE** to 1 if you want to allow this option. In this case whenever you open a file for read ("r"), then a sector write will happen on directory entry. This updates the last accessed date (date is checked before updating to ensure it needs updating). To avoid this option (which saves unnecessary sector writes) set **F_UPDATELASTACCESSDATE** to 0. In this case only other file ma-nipulations ("r+","w","w+","a","a+") change this date entry.

10.4.11 Fast Seeking

The developer can define a number of points in a file to use as markers to allow fast seeking in a file. The **F_MAXSEEKPOS** definition in **udefs.h** sets a number of points to be stored with every file descriptor. Setting this to zero will mean that seeking will al-ways work from the current position or the beginning of the file only. **F_MAXSEEKPOS** should be equal to a power of 2 or equal to zero.

The memory usage of the system is increased by:

```
F_MAXSEEKPOS*F_MAXFILES*sizeof(long)
```

11 Board Support Packages

11.1 Introduction

Only the device drivers which are needed in a typical SCIOPTA ARM - FAT 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 FAT File System we will use HCC FAT Flash File System drivers. Please consult chapter 11.21 "FAT File System Drivers" on page 11-35 for more information about the HCC FAT 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 SCIOPTA FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT 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 FAT File System Drivers

In the SCIOPTA FAT File System the HCC FAT File System driver can be directly used.

They can be found at: File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\

The following diagram illustrates the structure of the HCC FAT File System device driver.

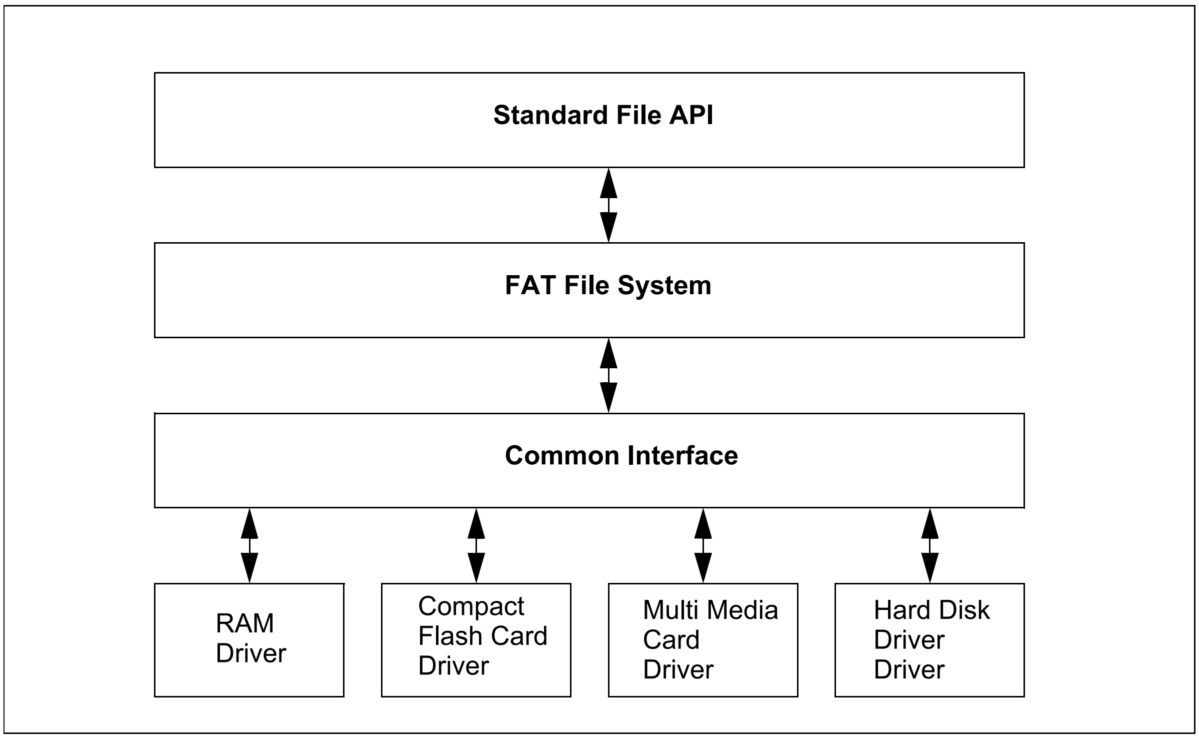


Figure 11-1: HCC FAT File System Device Driver Structure

11.21.1 HCC Driver Interface

This section documents the required interface functions to provide a media driver for the file system.

Reference should also be made to the sample device drivers supplied with the code when developing a new driver. The easiest starting point is the RAM driver.

11.21.1.1 Driver Interface Functions

```
xxx_initfunc
xxx_getphy
xxx_readsector
xxx_readmultiplesector
xxx_writesector
xxx_writemultiplesector
xxx_getstatus
xxx_release
```

These are the routines that may be supplied by any driver.

The **xxx** is a reference to the particular driver being developed e.g. **xxx=cfc** for compact flash card driver.

The **xxx_initfunc** routine is mandatory and is passed to the f_initvolume routine to initialize a volume. This passes a set of pointers to the driver interface functions below to the file system.

The xxx_getphy routine is mandatory and is called by the file system to find out the physical properties of the device, such as the number of sectors.

The xxx_readsector routine is mandatory and is used to read a sector from the target device.

The xxx_readmultiplesector routine is optional and is used to read a series of sectors from the target device. If not available xxx_readsector will be used.

The xxx_writesector routine is optional and is required to write a sector to the target device. It is mandatory if format is required.

The xxx_writemultiplesector routine is optional and is used to write a series of sectors to the target device. If not available xxx_writesector will be used.

The xxx_getstatus routine is optional and is used only for removable media to discover whether a card has been removed or changed.

The xxx_release routine is optional and can be used to release any resources associated with a drive when it is removed.

11.21.1.2xxx_initfunc

Passed to the f_initvolume and/or routine to create the driver. The routine passes to the file system a set of function pointers to access the volume and also the driver pointer itself. These function pointers are in addition to the other functions documented in this section.

xxx_initfunc should allocate or use a static structure. It must return with the filled **F_DRIVER** structure and its pointer value. The **F_DRIVER** structure is defined as:

```
typedef struct F_DRIVER
{
   FN_MUTEX_TYPE mutex;/* mutex for the driver*/
   char separated;/* signal if the driver is separated */

   unsigned long user_data;/* user defined data */
   void *user_ptr;/* user define pointer */

   /* driver functions */
   F_WRITESECTOR          writesector;
   F_WRITEMULTIPLESECTOR  writemultiplesector;
   F_READSECTOR           readsector;
   F_READMULTIPLESECTOR   readmultiplesector;
   F_GETPHY               getphy;
   F_GETSTATUS            getstatus;
   F_RELEASE              release;
} _F_DRIVER;
```

All function pointers to inform the file system which functions to call.

user_ptr and **user_data** are assigned by the driver. The values stored in user_ptr and user_data are included in the **F_DRIVER** structure and all driver function calls for that volume. The uses of these fields are determined by the driver; typically they are used to identify one of a set of attached interfaces, such as one of a number of Compact Flash card slots being controlled by a single driver. A call to f_delvolume will cause the file system to call the driver xxx_release with **F_DRIVER** structure pointer, where the as-signed user_ptr, which will then be removed when the driver function returns.

```
F_DRIVER *xxx_initfunc(
             unsigned long     driver_param
)
```

Parameter	Description
driver_param	Driver parameter. The driver_param value passed to the xxx_initfunc is determined by the f_initvolume or f_createdriver call. The driver may use this value in the user_ptr or user_data field of the returned structure or assign another value as the driver requires. The file system will make all subsequent calls to driver functions with the assigned value in the F_DRIVER structure.

Return Value	Condition
Driver pointer (*F_DRIVER)	Success.
NULL	Error condition.

11.21.1.3 xxx_getphy

This function is called by the file system to discover the physical properties of the drive. The routine will set the number of cylinders, heads and tracks and the number of sectors per track.

```
int xxx_getphy(
             F_DRIVER     *driver,
             F_PHY        *pPhy
)
```

Parameter	Description
driver	Driver structure.

pPhy	Pointer to physical control structure.
	The **F_PHY** structure is defined as follows:
	
	`typedef struct`
	`{`
	`   unsigned short number_of_cylinders; /* number of cylinders */`
	`   unsigned short sector_per_track;   /* sectors per track */`
	`   unsigned short number_of_heads;   /* number of heads */`
	`   unsigned long number_of_sectors;   /* number of sectors */`
	`   unsigned char media_descriptor;     /* fix or removable */`
	`                                 /* use _MEDIADESC_xxx */`
	`} F_PHY;`
	
	The number of cylinders is not required by the system. All other parameters must be set correctly by the **xxx_getphy** function.

Return Value	Condition
F_NO_ERROR	Success.
Error Code	Error condition.

11.21.1.4xxx_readsector

This function is called by the file system to read a complete sector.

```
int xxx_readsector(
             F_DRIVER        *driver,
             void            *data,
             unsigned long   sector
)
```

Parameter	Description
driver	Driver structure.
data	Pointer to write data to from specified sector.
sector	Number of sector to be read.

Return Value	Condition
0	Success.
!=0	Sector out of range.

11.21.1.5xxx_readmultiplesector

This function is called by the file system to read a series of consecutive sectors. This function is optional - its inclusion will enhance performance on most devices and is par-ticularly important with hard disk drives.

```
int xxx_readmultiplesector(
             F_DRIVER        *driver,
             void            *data,
             unsigned long   sector,
             int             cnt
)
```

Parameter	Description
driver	Driver structure.
data	Pointer to write data to from specified sector.
sector	Number of first sector to be written.
cnt	number of sectors to write.

Return Value	Condition
0	Success.
!=0	Sector out of range.

11.21.1.6 xxx_writesector

This function is called by the file system to write a complete sector.

```
int xxx_writesector(
             F_DRIVER        *driver,
             void            *data,
             unsigned long   sector
)
```

Parameter	Description
driver	Driver structure.
data	Pointer to data to write data to specified sector.
sector	Number of sectors to be written.

Return Value	Condition
0	Success.
!=0	Sector out of range.

11.21.1.7 xxx_writemultiplesector

This function is called by the file system to write a series of consecutive sectors. This function is optional - its inclusion will enhance performance on most devices and is particularly important with hard disk drives.

```
int xxx_writemultiplesector(
             F_DRIVER        *driver,
             void            *data,
             unsigned long   sector,
             int             cnt
)
```

Parameter	Description
driver	Driver structure.
data	Pointer to data to write to specified sector.
sector	Number of first sector to be written.
cnt	number of sectors to write.

Return Value	Condition
0	Success.
!=0	Sector out of range.

11.21.1.8 xxx_getstatus

This function is called by the file system to check the status of the media. It used with removable media to check that a card has not been removed or swapped. The function returns a bit field of new status information.

If the drive is a permanent medium (e.g., hard disk or RAM drive), this function may be omitted.

```
int xxx_getstatus(
             F_DRIVER        *driver
)
```

Parameter	Description
driver	Driver structure.

Return Value	Condition
0	Success.
F_ST_MISSING	Card has been removed (Bit field).
F_ST_CHANGED	The card has been removed and replaced (Bit field).
F_ST_WRITEPROTECT	The card is write protected (Bit field).

11.21.1.9 xxx_release

This function is called by the file system to remove a drive. The drive can use this call to free any resources associated to that drive. Use of this routine in the driver is optional.

This function is called is an f_delvolume API call is made if volume was created by f_initvolume or this function is called when f_releasedriver is called. After this is completed the file system removes all record of this volume.

```
void xxx_release(
             F_DRIVER        *driver
)
```

Parameter	Description
driver	Driver structure.

Return Value	Condition
none	

11.21.2 Compact Flash Card

11.21.2.1 Overview

The Compact Flash Card (CFC) driver is designed to operate with all standard type 1 and 2 compact flash cards.

There are three methods for interfacing with a Compact Flash Card:

- True IDE Mode
- PC Memory Mode
- PC I/O Mode

The package contains a sample driver for all three modes.

Throughout the code the areas which are target specific have been put within an HCC_HW definition e.g.

```
#ifdef HCC_HW
Target specific hardware parts
#endif
```

Within these areas the parts listed in this section must be provided for the driver to function.

Include Files

cfc_ide.h	Header file for IDE source file.
cfc_io.h	Header file for I/O.
cfc_mem.h	Header file for MEM.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\cfc\arm\

Source Files

cfc_ide.c	IDE source file.
cfc_io.c	I/O source.
cfc_mem.c	MEM source.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\cfc\arm\

11.21.2.2 Hardware Porting

The following are the header file definitions which must be modified

CFC_TOVALUE - this value is hardware dependent and is a counter for loop expiry. The developer may replace this with a host OS timeout function.

CFC_CSO - this is for accessing a chip select register and is hardware dependent. The code assumes a chip select is used to access the card and is removed after access. The developer must modify this and all accesses to meet the host system design. It should also be noted that the chip select needs to be set for a relatively long access time (>300ns). Developers should check the timing in the CFC Specification.

The following definitions are used to access the compact flash registers:

```
CFC_BASE        - Base address of the compact flash card
CFC_DATA        - Macro to access the data register
CFC_SECTORCOU   - Macro to access the sector count register
CFC_SECTORNO    - Macro to access the sector number register
CFC_CYLINDERLO  - Macro to access the cylinder low word register
CFC_CYLINDERHI  - Macro to access the cylinder high word register
CFC_SELC        - Macro to access the select card register
CFC_COMMAND     - Macro to access the command register
CFC_STATE       - Macro to access the state register (same address as command)
```

CPLD Logic:

HCC uses CPLD logic in most of its reference designs for CFCards. The following definitions are used to read from HCC CPLD logic state changes in the card.

```
CFC_CPLDSTATE        - MACRO for reading the state
CFC_CPLDSTATE_CDCH   - State bit for card has changed
CFC_CPLDSTATE_CFCD   - State bit for card removed
```

The developer must provide code to reflect this functionality. Contact support@hcc-embedded.com for reference design information.

11.21.2.3 Setting IDE Mode

A special sequence needs to be executed to force the compact flash card into IDE mode. This is done in HCC hardware by a sequence executed by the CPLD which:

1. switches off power to the card
2. OE signal is grounded
3. switches power on

Please refer to the CFC specification.

11.21.3 MultiMediaCard/Secure Digital Card Driver

11.21.3.1 Overview

The sample drivers provided support MMC cards, SD cards Version 1 and 2 and SDHC cards. Various other derivative types are also supported, such as mini-SD cards and Transflash.

Secure Digital cards are a super-set of MultiMediaCards. They can be used exactly in the same manner as MMCs but have additional functionality available. In particular they have two additional interface pins.

When used in Secure Digital mode there are 3 methods of communicating with the card:

1. SPI mode

   This is available on both MMC and SD cards primarily because of its wide availability and ease of use. Because many standard CPUs support an SPI interface it reduces the load on the host system compared to other interface methods. When SPI is implemented by software control this benefit is lost.

2. MultiMediaCard Mode

   This is a special mode for communicating with MultiMediaCards requiring very few IO pins. It has the disadvantage that generally software has to control every bit transfer and clock.

3. Secure Digital Mode

   This is not compatible with MultiMediaCards. It has the basic advantage that it uses four data lines and thus the potential transfer speeds are higher (up to 10MBytes/sec) but un-less there is specific UART hardware on the host system the load on the host is generally much higher than in SPI mode (with hardware support).

11.21.3.2 Implementation

FAT provides two generic MMC/SD card drivers - one for handling a single MMC card interface, the other for handling multiple MMC interfaces through a single driver. These drivers can be found in the **mmc/multi** and **mmc/single** directories. These drivers do not normally require modification.

Sample SPI drivers are included in sub-directories. These must be ported for particular targets.

Source Files

drv.c	Multi Media Card driver mmc/multi.
drvs.c	Multi Media Card SPI driver mmc/multi.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\mmc\multi\arm\

drv.c	Multi Media Card driver mmc/single.
drvs.c	Multi Media Card SPI driver mmc/single.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\mmc\single\arm\

11.21.3.3 Porting the SPI Driver

The sample drivers are included to provide an easy porting reference. There are no standards for SPI implementations so each target is different, though generally this functionality is easy to realize.

The SPI driver must include the following functions:

`void spi_tx1 (unsigned char data8)`

Transmits a single byte through the SPI port.

`void spi_tx2 (unsigned short data16)`

Transmits two bytes through he SPI port. This may simply call spi_tx1() twice.

`void spi_tx4 (unsigned long data32)`

Transmits four bytes through he SPI port. This may simply call spi_tx1() four times.

`void spi_tx512 (unsigned char *buf)`

Transmits two bytes through the SPI port. This may simply call spi_tx1() twice.

`unsigned char spi_rx1 (void)`

Receives a single byte.

`void spi_rx512 (unsigned char *buf)`

Receives 512 bytes.

`void spi_cs_lo (void)`

Set the SPI chip select to low (active) state.

`void spi_cs_hi (void)`

Set the SPI chip select to high (inactive) state.

`int spi_init (void)`

Does any required SPI port initialization.

`void spi_set_baudrate (unsigned long br)`

Sets the baud rate of the SPI port.

`unsigned long spi_get_baudrate (void)`

Gets the current baud rate of the SPI port.

`int get_cd (void)`

Gets the state of the Card Detect signal.

`int get_wp (void)`

Gets the state of the Write Protect signal.

`t_mmc_dsc *get_mmc_dsc (void)`

Gets the MMC parameter structure; may be used at higher level for information about the connected card.

The following functions are required only when the driver supports multiple MMC/SD card interfaces simultaneously.

`F_DRIVER *spi_add_device (unsigned long driver_param)`

This function adds a new sub-device or interface to the driver.

`int spi_del_device (void *user_ptr)`

This function removes a sub-device or interface from the driver.

`int spi_check_device (void *user_ptr)`

This function ensures that the interface pointed to by the user_ptr is the active interface.

11.21.4 Hard Disk Drive

11.21.4.1 Overview

The Hard Disk Drive (HDD) driver is designed to operate with a standard IDE HDD. The sample driver is designed to handle two HDDs simultaneously.

Include Files

hdd_ide.h	Header file for IDE source file.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\hdd\arm\

Source Files

hdd_ide.c	IDE source file.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\hdd\arm\

11.21.4.2 Hardware Porting

Throughout the code the areas that are target-specific have been put within an HCC_HW definition. An example is

```
#ifdef HCC_HW
Target specific hardware parts
#endif
```

Within these areas the parts listed in this section must be provided for the driver to function.

The following are the header file definitions that must be modified:

#define	Description
HDD_TOVALUE	This value is hardware dependent; it is a counter for loop expiry. The developer may replace it with a host OS timeout function.
HDD_INIT_TO	This value is hardware dependent; it is a counter for loop expiry. The developer may replace it with a host OS timeout function.
HDD_BASE0	Base address of the first HDD.
HDD_CSBASE0	Chip select base register for first HDD.
HDD_CSOPT0	Chip select option register for first HDD.
HDD_CONTROL0	Control register in CPLD control logic for HDD.

Hard Disk Drive Registers:

The following definitions are used to access the hard disk drive registers:

#define	Description
`HDD_DATA`	Macro to access the data register.
`HDD_FEATURE`	Macro to access the feature register.
`HDD_SECTORCOU`	Macro to access the sector count register.
`HDD_SECTORNO`	Macro to access the sector number register.
`HDD_CYLINDERLO`	Macro to access the cylinder low word register.
`HDD_CYLINDERHI`	Macro to access the cylinder high word register.
`HDD_SELC`	Macro to access the select card register.
`HDD_COMMAND`	Macro to access the command register.
`HDD_STATE`	Macro to access the state register (same address as command).

11.21.5 RAM Driver

The RAM driver is a good starting point for implementing a new driver. The sample RAM driver is written to support two independent drives.

The RAM driver does not include a ram_getstatus routine because there is no concept of removing and replacing the drive; it is always present once initialized.

Follow these steps to build a RAM drive:

1. Include the ramdrv.c and ramdrv.h files in your file system build. This ensures that the drive can be mounted.
2. Modify the **RAMDRIVE_SIZE** define to the size of block of RAM you wish to use for this drive. NB: This is statically assigned; if you require it to be malloc'd this is a minor change. Also note: there are minimum sizes for FAT16 and FAT32. To build a FAT16 file system you must assign 2.8MB of RAM and 32MB for a FAT32. Because of this, it is normal to run FAT12 in RAM. About 50K is the minimum required to run a RAM drive.
3. Call f_initvolume with the number of the volume and also with a pointer to the **f_ramdrvinit** function.
4. Call f_format to format the drive.

```
void main(void)
{
   /* Initialize File System */
   f_init();

   /* mount RAM drive as drive A: */
   f_initvolume(0, f_ramdrvinit, F_AUTO_ASSIGN);

   /* format the drive */
   /* creates boot sector information and volume */

   f_format(0, F_FAT12_MEDIA);  create FAT12 in RAM */

   /* now free to use the drive */
           .
           .
           .
}
```

The RAM drive may now be accessed as a standard drive using the API calls.

When running the test suite with the RAM drive, certain tests will fail be-cause the drive is destroyed through the simulated power on/off.

Building a RAM driver requires a quantity of RAM. The typical minimum size of RAM we recommend using for a FAT12 RAM drive is 32K. The actual minimum size of a FAT12 RAM drive is 36 sectors (18K) which allows just one sector (512 bytes) for file storage!

Include Files

ramdrv_f.h	RAM driver header.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\ram\

Source Files

ramdrv_f.c	RAM driver.

File location: <install_folder>\sciopta\<version>\bsp\hcc\fatfs\ram\

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.

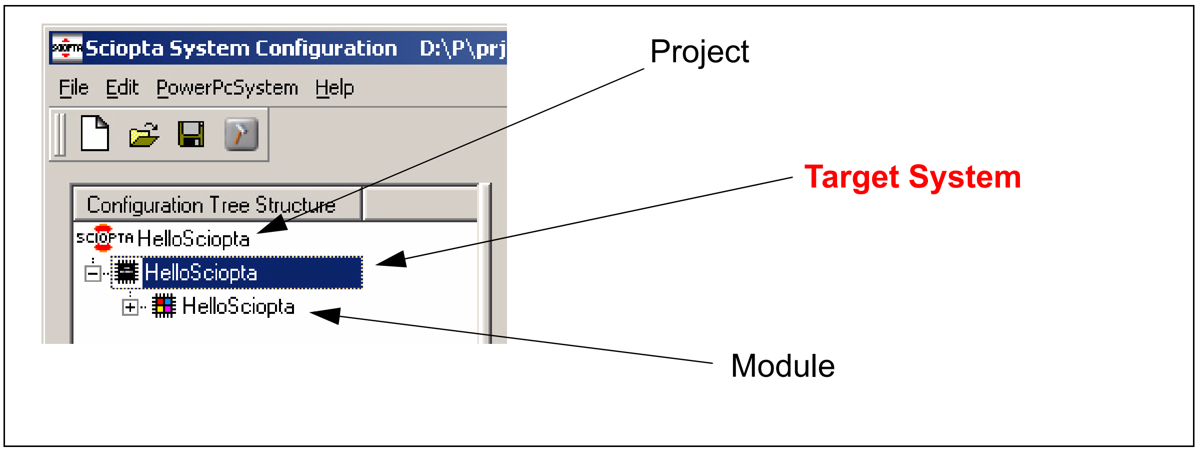

Sciopta System Configuration D:\P\prj
File Edit PowerPcSystem Help
Configuration Tree Structure
HelloSciopta
HelloSciopta
HelloSciopta
Project
Target System
Module

Figure 12-1: SCIOPTA System

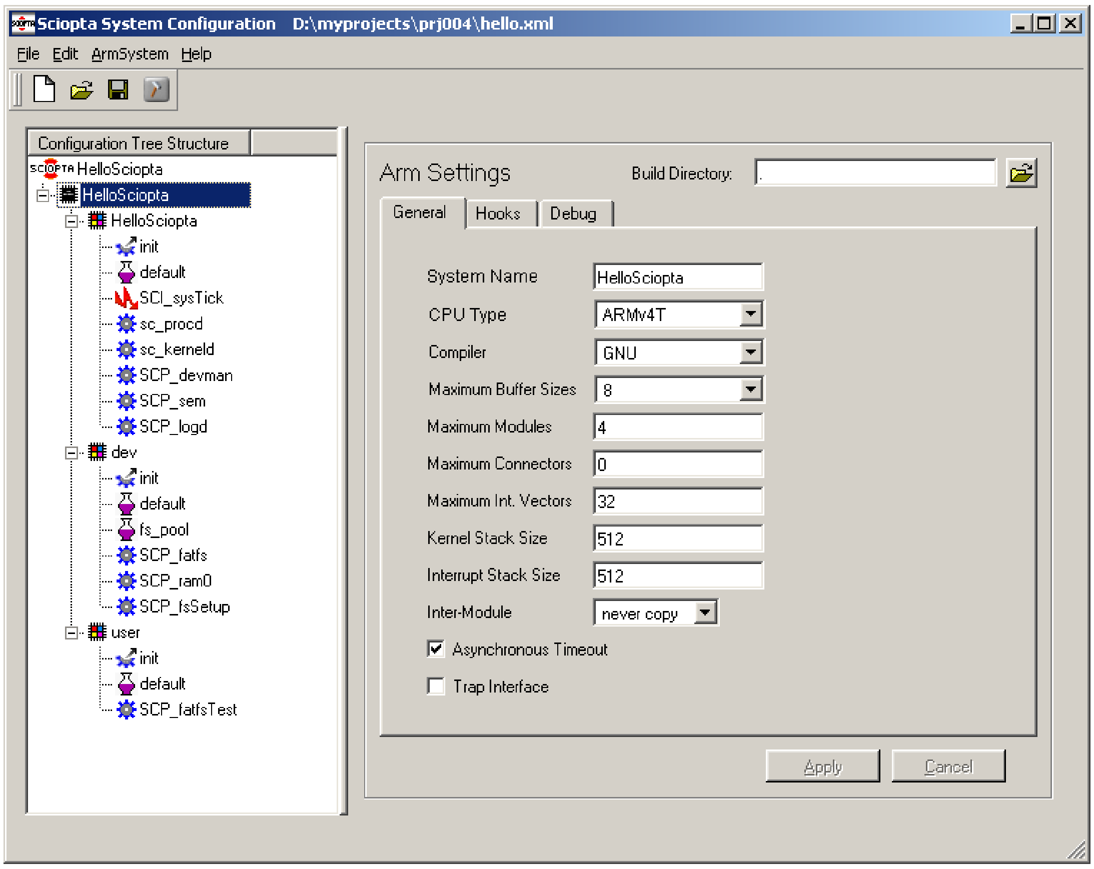
Please consult the chapter “Kernel Configuration” of the SCIOPTA ARM - Kernel, User’s Guide for a detailed description of the SCONF target system, module, process and pool configuration and the configuration parameters.

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 FATFS 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: FAT File System SCONF Configuration Example

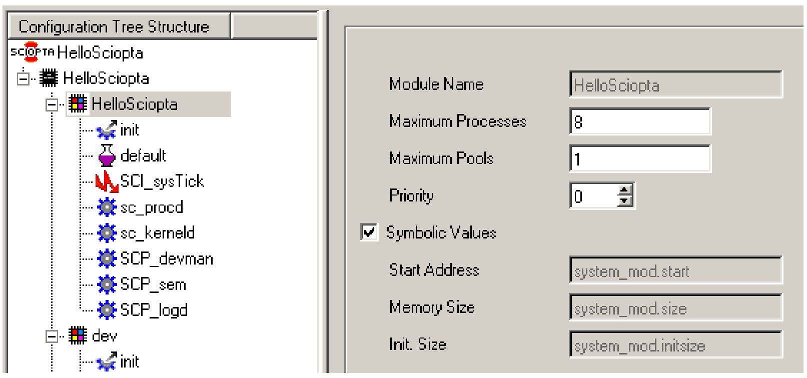
Above example system consists of four modules.

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.

SCIOPTA

12.4 System Module

The system module gets automatically the name HelloSciopta which is the system name.

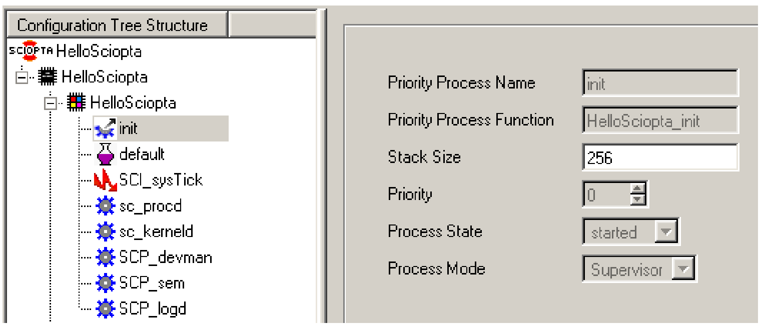


Configuration Tree Structure
- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd
    - dev
      - init

Module Name	HelloSciopta
Maximum Processes	8
Maximum Pools	1
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

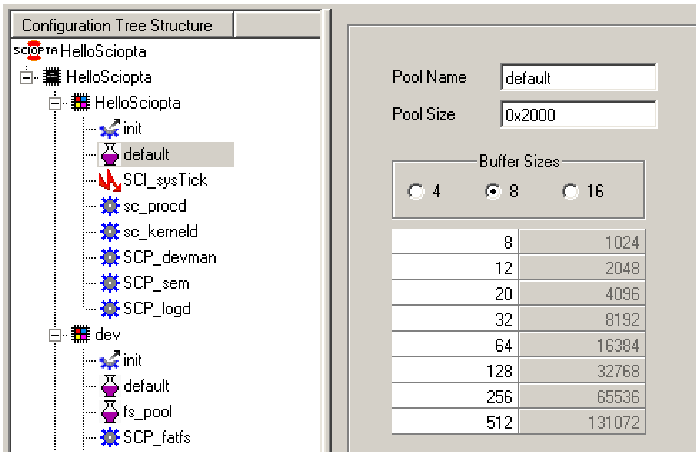
The init process is automatically created and gets the name **init** and the function **HelloSciopta_init**. Only the stack size need to be configured.



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



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_fatfs

Pool Name: default

Pool Size: 0x2000

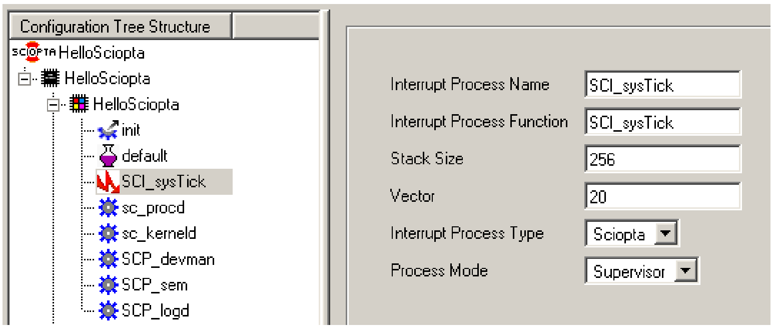
Buffer Sizes: ( ) 4 [✓ 8] ( ) 16

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.



Configuration Tree Structure

- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd

Interrupt Process Name: SCI_sysTick

Interrupt Process Function: SCI_sysTick

Stack Size: 256

Vector: 20

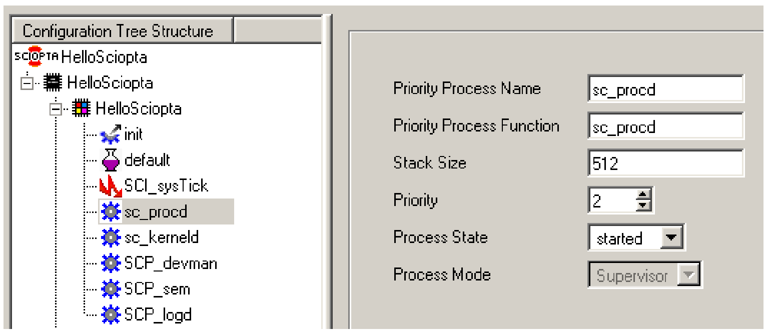
Interrupt Process Type: Sciopta

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.



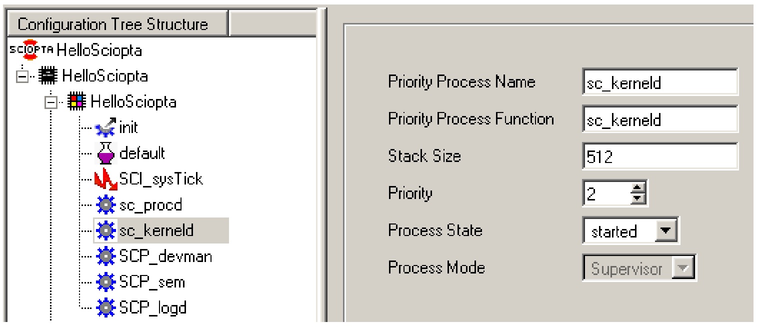
Priority Process Name: sc_procd
Priority Process Function: sc_procd
Stack Size: 512
Priority: 2
Process State: started
Process Mode: Supervisor

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

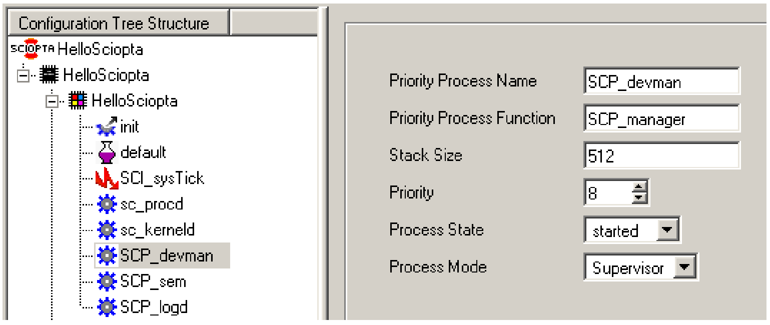
12.4.6 Flash File System Manager Process

SCIOPTA FAT 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.



Configuration Tree Structure
- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd

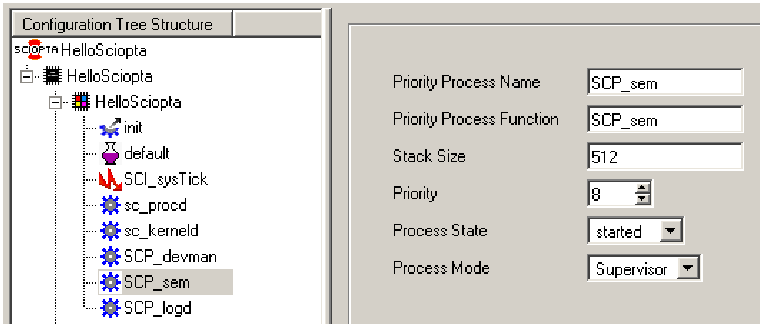
Priority Process Name	SCP_devman
Priority Process Function	SCP_manager
Stack Size	512
Priority	8
Process State	started
Process Mode	Supervisor

12.4.7 SCIOPTA Semaphore Emulation Process

The SCIOPTA - HCC 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.



Configuration Tree Structure
- HelloSciopta
  - HelloSciopta
    - HelloSciopta
      - init
      - default
      - SCI_sysTick
      - sc_procd
      - sc_kerneld
      - SCP_devman
      - SCP_sem
      - SCP_logd

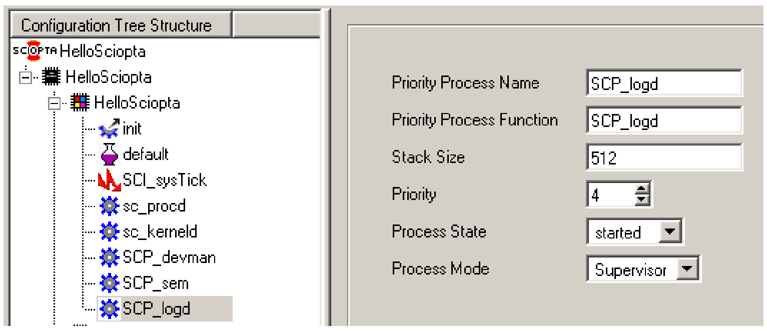
Priority Process Name	SCP_sem
Priority Process Function	SCP_sem
Stack Size	512
Priority	8
Process State	started
Process Mode	Supervisor

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.

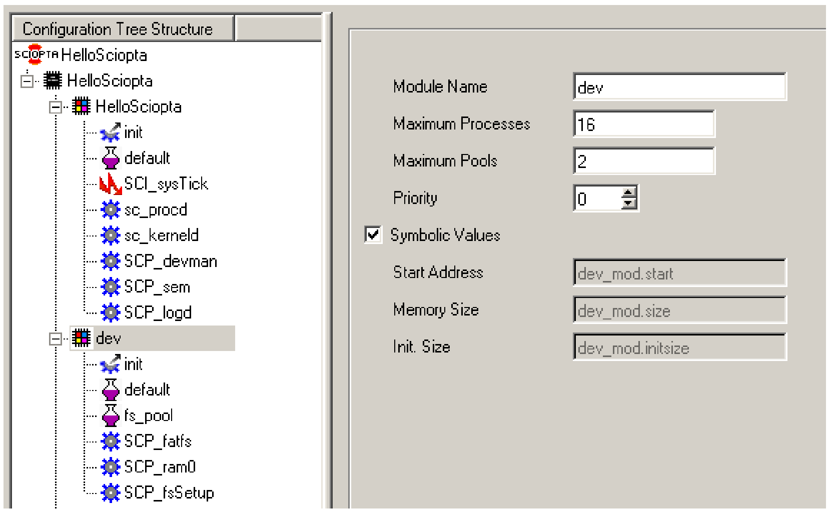
The user just need to define a prioritized static process with the process name SCP_logd and the process function SCP_logd.



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

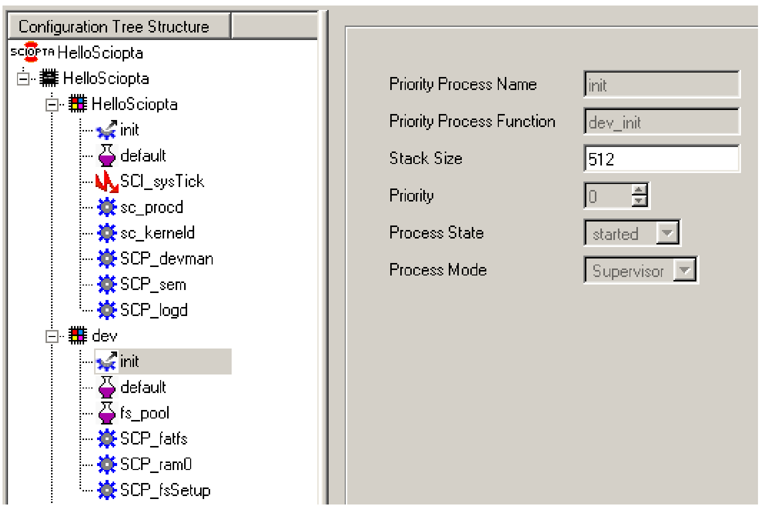
The flash file system processes are placed in the dev module.



Module Name	dev
Maximum Processes	16
Maximum Pools	2
Priority	0
✓ Symbolic Values	
Start Address	dev_mod.start
Memory Size	dev_mod.size
Init. Size	dev_mod.initsize

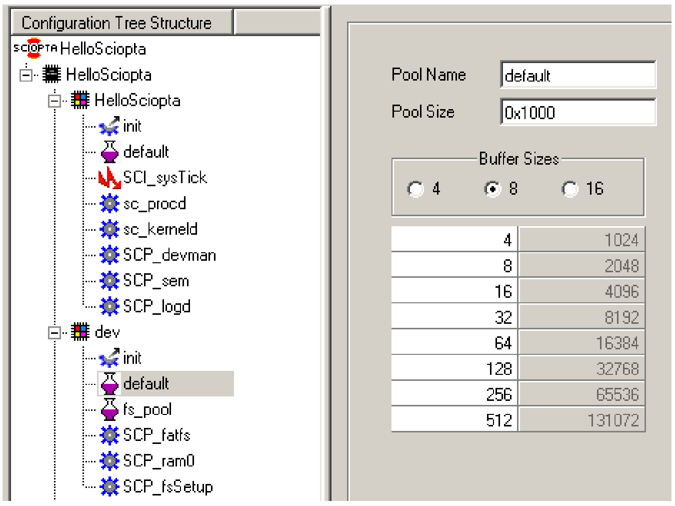
12.5.1 dev Module Init Process Configuration

The init process is automatically created and gets the name **init** and the function **dev_init**. Only the stack size need to be configured.



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



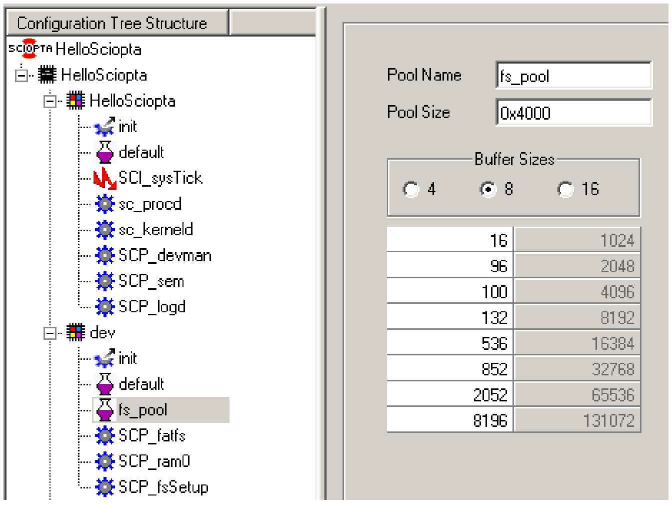
Pool Name: default

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



Pool Name: fs_pool

Pool Size: 0x4000

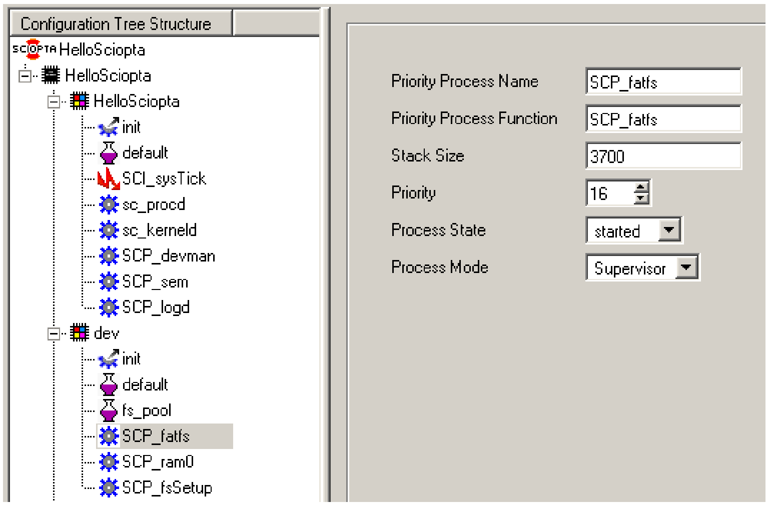
Buffer Sizes: ( ) 4 [✓ 8] ( ) 16

16	1024
96	2048
100	4096
132	8192
536	16384
852	32768
2052	65536
8196	131072

12.5.4 File System Process

The file system process includes the main SCIOPTA FAT File System initialization functions. It is using the HCC FAT File System (HCC FAT API), the HCC FAT 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_fatfs** and the process function **SCP_fatfs**.

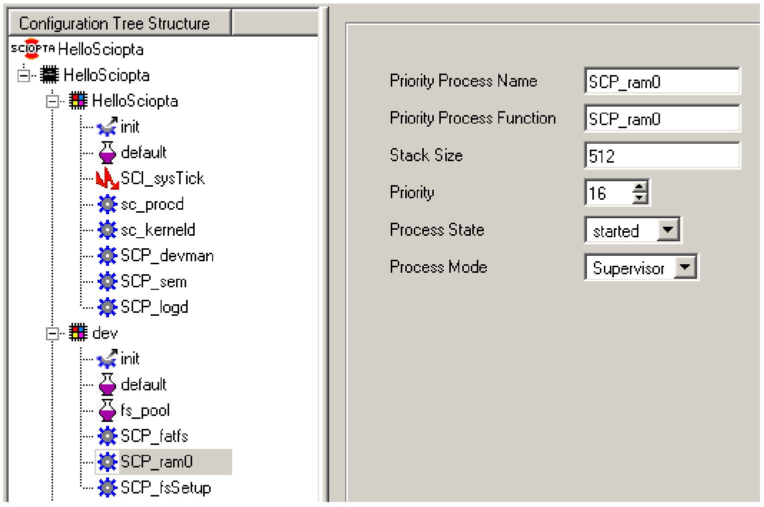


Priority Process Name	SCP_fatfs
Priority Process Function	SCP_fatfs
Stack Size	3700
Priority	16
Process State	started
Process Mode	Supervisor

12.5.5 SCIOPTA RAM Disk Driver Process

If the RAM disk driver is used and implemented as a SCIOPTA device driver the RAM disk driver process must be declared. Other file device processes must be declared similarly.

The user needs to define a prioritized static process with the process name **SCP_ram0** and the process function **SCP_ram0**.

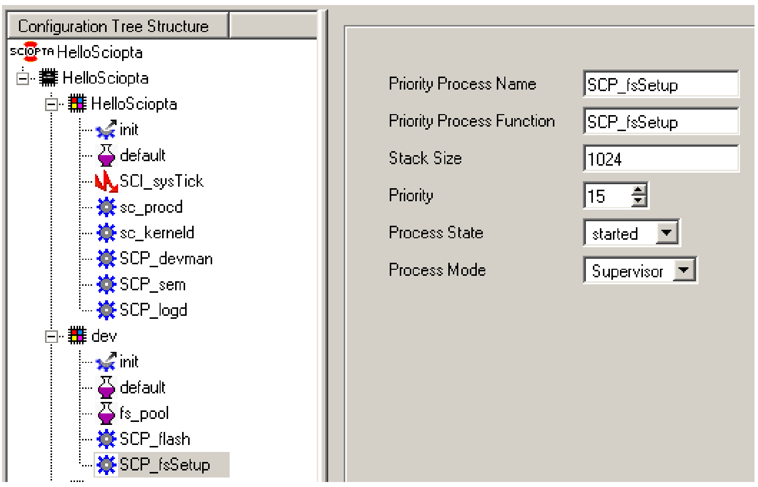


Priority Process Name	SCP_ram0
Priority Process Function	SCP_ram0
Stack Size	512
Priority	16
Process State	started
Process Mode	Supervisor

12.5.6 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.

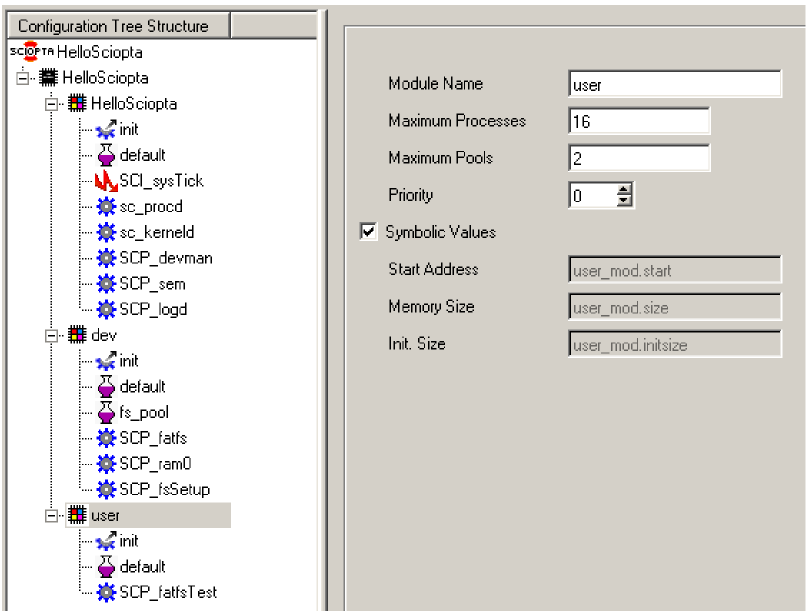
The user needs to define a prioritized static process with the process name **SCP_fsSetup** and the process function **SCP_fsSetup**.



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

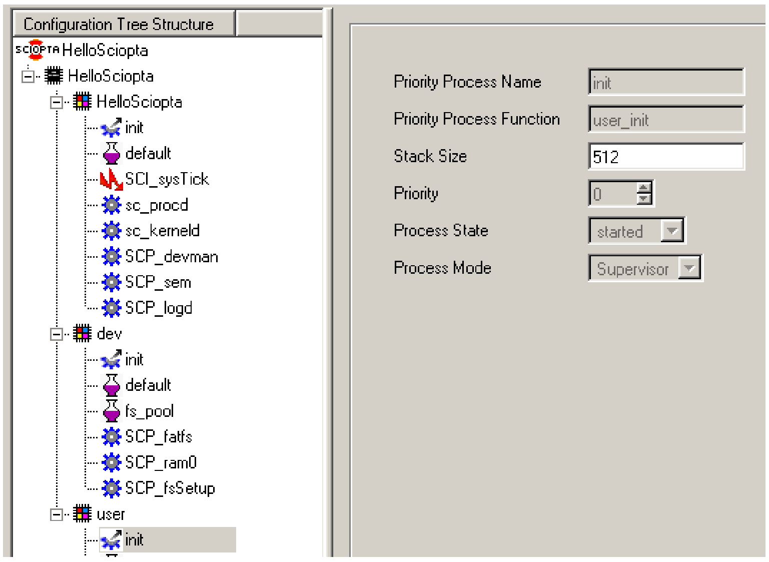
The user Module contains the FAT file system application.



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

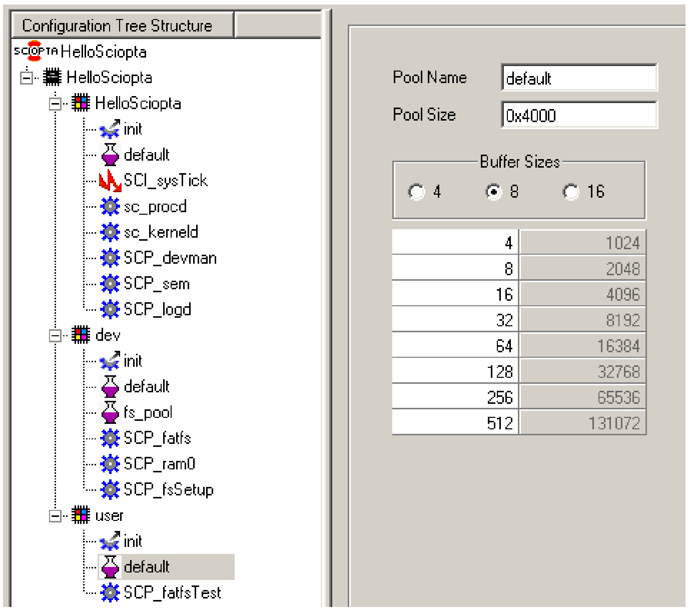
12.6.1 user Module Init Process Configuration

The init process is automatically created and gets the name **init** and the function **user_init**. Only the stack size need to be configured.



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



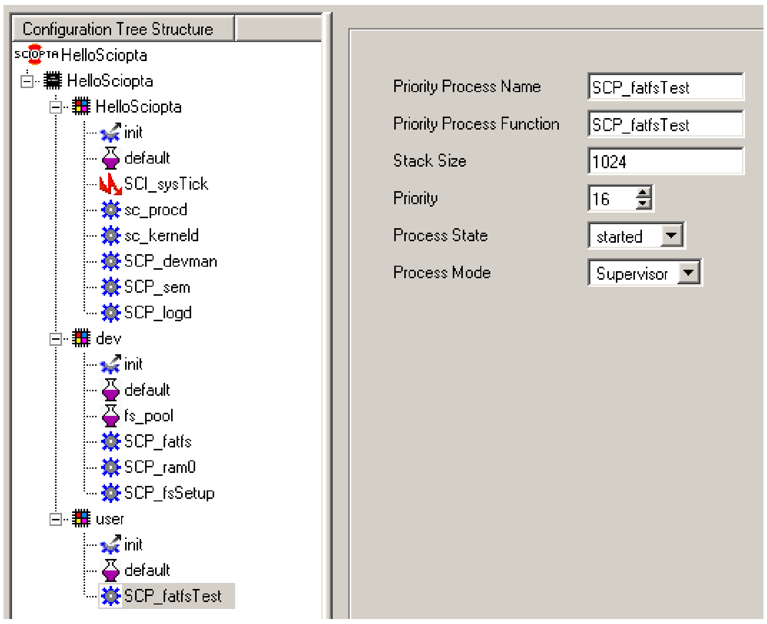
Pool Name: default
Pool Size: 0x4000

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 FAT File System User Process

This is the example process of the getting started example.



Priority Process Name: SCP_fatfsTest
Priority Process Function: SCP_fatfsTest
Stack Size: 1024
Priority: 16
Process State: started
Process Mode: Supervisor

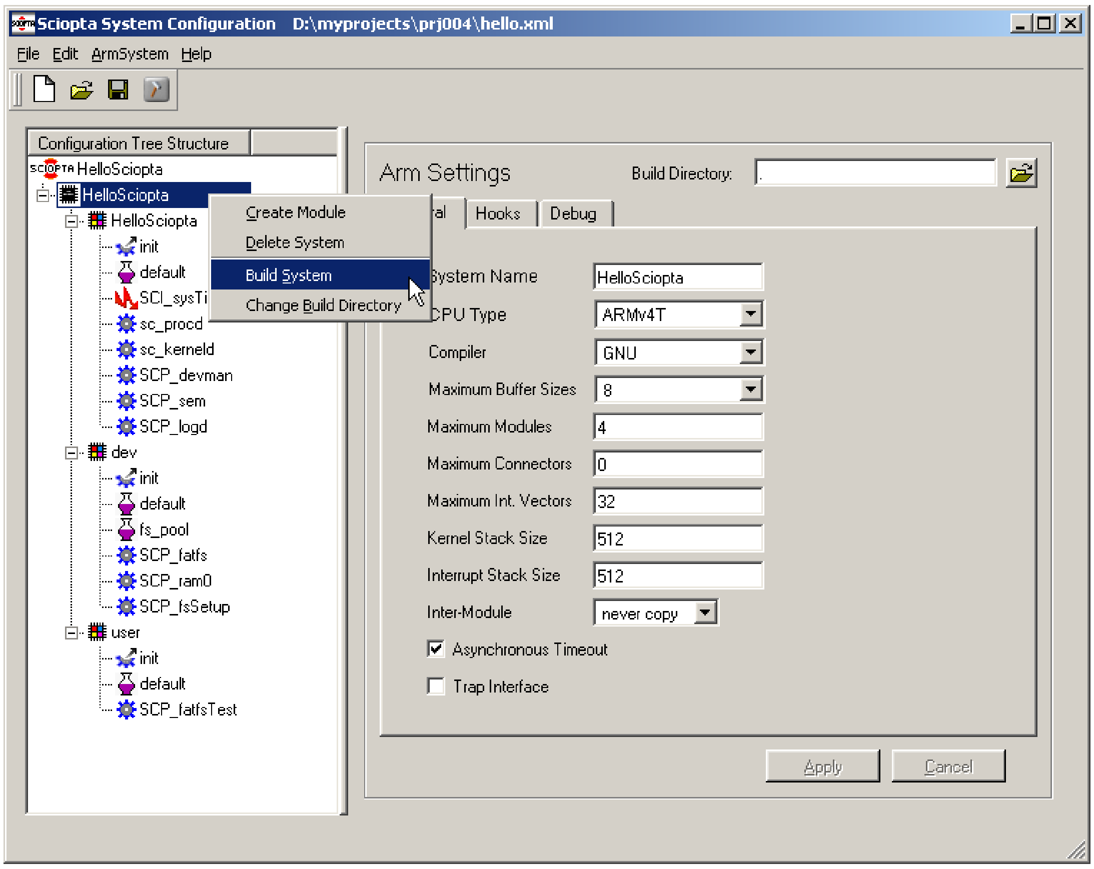
12.7 Generating the Configuration Files

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 FATFS Function Interface Reference” on page 8-1).

You need to include this library for all SCIOPTA FAT File System projects.

13.3.1 GNU GCC File System Libraries

The file name of the libraries have the following format:

lib<name>_Xt.a

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\<CPU>\include
%(SCIOPTA_HOME)\bsp\arm\<CPU>\<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 2.1

- 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, 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 2.0

- Manual completely rewritten.

16.3 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**.

16.4 Manual Version 1.0

- Initial version.

17 Index