SCIOPTA

SCIOPTA Getting Start Manual for ARM

v1.1

Abstract

This document describes the **SCIOPTA Getting Start Manual for ARM** for the SCIOPTA Kernels.

Copyright

Disclaimer

Trademark

Contact

Corporate Headquarters
SCIOPTA Systems GmbH
Hauptstrasse 293
79576 Weil am Rhein
Germany
Tel. +49 7621 940 919 0
Fax +49 7621 940 919 19
email: sales@sciopta.com
www.sciopta.com

Contents

1. Introduction

1.1. SCIOPTA Real-Time Operating System

SCIOPTA is a high performance fully pre-emptive real-time operating system for hard real-time application available for many target platforms.

Available modules:
• Pre-emptive Multitasking Real-Time Kernel
• Board Support Packages
• IPS - Internet Protocols v4/v6(TCP/IP)
• IPS Applications - Internet Protocols Applications (Web Server, TFTP, FTP, DNS, DHCP, Telnet, SMTP etc.)
• FAT File System
• (fail) SAFE FAT File System
• Flash File System, NOR and NAND
• Universal Serial Bus, USB Device
• Universal Serial Bus, USB Host
• DRUID - System Level Debugger
• SCIOPTA PEG - Embedded GUI
• CONNECTOR - support for distributed multi-CPU systems
• SCIOPTA Memory Management System - Support for MMU
• SCAPI - SCIOPTA API for Windows or LINUX host
• SCSIM - SCIOPTA Simulator

SCIOPTA Real-Time Operating System contains design objects such as SCIOPTA modules, processes, messages and message pools. SCIOPTA is designed on a message based architecture allowing direct message passing between processes. Messages are mainly used for interprocess communication and synchronization. SCIOPTA messages are stored and maintained in memory pools. The memory pool manager is designed for high performance and memory fragmentation is avoided. Processes can be grouped in SCIOPTA modules, which allows you to design a very modular system. Modules can be static or created and killed during runtime as a whole. SCIOPTA modules can be used to encapsulate whole system blocks (such as a communication stack) and protect them from other modules in the system.

The SCIOPTA Real-Time Kernel has a very high performance. The SCIOPTA architecture is specifically designed to provide excellent real-time performance and small size. Internal data structures, memory management, interprocess communication and time management are highly optimized. SCIOPTA Real-Time kernels will also run on small single-chip devices without MMU.

1.2. SCIOPTA Safety Kernels

SCIOPTA provides a Real-Time Operating System for some CPU families which is certified according to IEC 61508 up to SIL3, CENELEC EN 50128 up to SIL3/4 and ISO 26262 up to ASIL D:

`IEC INTERNATIONAL STANDARD 61508`
Edition 2.0 2010-04
Functional safety of electrical/electronic/programmable electronic safety-related systems
Part 1: General requirements
Part 2: Requirements for electrical/electronic/programmable electronic safetyrelated systems (in addition for INT Kernel)
Part 3: Software requirements
Part 4: Definitions and abbreviations

`CENELEC European Committee for Electrotechnical Standardization FprEN 50128:2011`
Railway applications
Communication, signalling and processing systems

Software for railway control and protection systems

`ISO International Organization for Standardization 26262`
First Edition 2018-12
Road vehicles – Functional safety
Part 2: Management of functional safety
Part 6: Product development at the software level
Part 8: Supporting processes

Please consult the SCIOPTA Kernel Manuals for more information (Ref. SCIOPTA Kernel Reference Manual and SCIOPTA Architecture Manual).

1.3. CPU Family

SCIOPTA is delivered for a specific CPU Family such as: ARM®7/9, ARM®11, ARM® Cortex-M™, ARM® Cortex™-R, ARM® Cortex™-A, Renesas RX, Freescale™ PowerPC, apm PowerPC, Freescale™ ColdFire and Marvell Xscale.

Please consult the latest version of the SCIOPTA Price List for the complete list.

1.4. About This Manual

The purpose of this SCIOPTA Kernel – ARM CPU Family, User´s Manual is to give all needed information how to use SCIOPTA Real-Time Kernel in an embedded project for the ARM CPU family. SCIOPTA is actually supporting the following ARM CPU families: ARM®7/9, ARM®11, ARM® Cortex-M™, ARM® Cortex™-R, ARM® Cortex™-A.

Chapter Getting Started Examples describes how to install SCIOPTA. Topics such as system requirements, installation procedure and removing are covered.

Chapter Typical Kernel Project describes the procedures and tasks for designing and implementing a typical small SCIOPTA kernel project.

Chapter Manual Versions contains the manual version history.

Please consult also the **SCIOPTA Kernel Reference Manual** which contains a complete description of all system calls and error messages and the **SCIOPTA GDD and Utilities User's and Reference Manual** for a detailed description of the GDD interface and SCIOPTA utilities.

2. Getting Started Examples

2.1. Introduction

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

2.2. Example Description

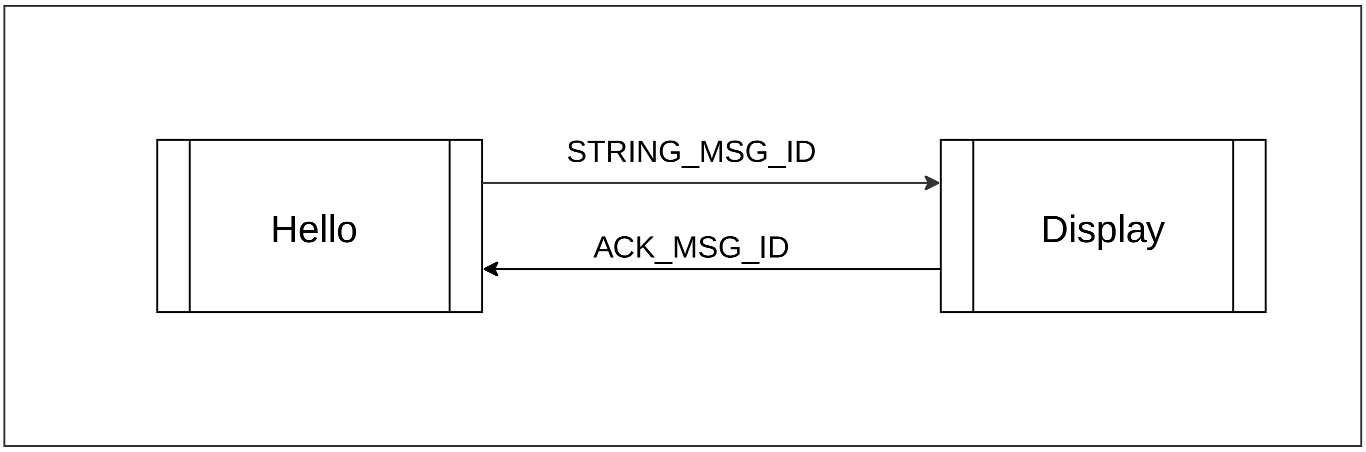


Figure 1. Getting Started Hello Example

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

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

2.3. Getting Started with IAR Embedded Workbench

This is a getting started step-by-step tutorial to build and run a small SCIOPTA real-time kernel example project for ARM by using the IAR Systems embedded workbench for ARM.

2.3.1. Equipment

- Personal computer or workstation running Microsoft® Windows.
- IAR Embedded Workbench C/C++ compiler and debugger tool suite for ARM.
- Target board which is supported by SCIOPTA examples. For each supported board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\.
- Serial line connected from a COM port of your host PC to the console UART of the selected board.
  The console port is board specific. If you want to change the port you may modify the files system.c, simple_uart.c, simple_uart.h and config.h.

2.3.2. Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in chapter SCIOPTA_HOME Environment Variable.
2. Be sure that the IAR Embedded Workbench is installed.
3. Create an example working directory at a suitable place.

4. Copy the script **copy_files_iar.bat** from the example directory for your selected target boards:

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

to your project folder.

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

2.4. Getting Started Makefile.

This is a getting started step-by-step tutorial to build and run a small SCIOPTA real-time kernel example project by using Makefile and GNU GCC.

2.4.1. Equipment

For architectures ARM, PowerPC and ColdFire the following equipment is used to run this getting started example:

- Personal computer or workstation running Microsoft® Windows.
- GNU make utility.
- Compiler for your Target architecture:
  - GCC: arm-none-eabi-gcc
  - TI: armcl
  - ARM v5: armcc
  - ARM v6: armclang
- Target board which is supported by SCIOPTA examples. For each supported board there is a directory in the example folder: <install_folder>\sciopta\<version>\exp\krn\<arch>\hello\.
- SCIOPTA Real-Time Kernel for your selected CPU-family.
- Serial line connected from a COM port of your host PC to the console UART of the selected board.
  The console port is board specific. If you want to change the port you may modify the files system.c, simple_uart.c, simple_uart.h and config.h.

2.4.2. Step-By-Step Tutorial

1. Check that the environment variable SCIOPTA_HOME is defined as described in chapter SCIOPTA_HOME Environment Variable.
2. Be sure that the SCIOPTA \win32\bin directory is included in the PATH environment variable as described in chapter SCIOPTA_HOME Environment Variable. 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 GCC compiler bin directory is included in the PATH environment variable.
4. Create an example working directory at a suitable place.
5. For each supported board, you then find below this board batch files, in general copy_files.bat (GCC example), copy_files_ti.bat (TI Makefile example), copy_files_arm.bat (ARM example).
6. Copy the script **copy_files.bat** from the example directory for your selected target boards:

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

to your project folder.

7. Double click **copy_files*.bat** to execute the batch file. All needed project files will be copied from the delivery to your project folder.
8. Check the Makefile for your board number. E.g. the ZedBoard has number 54. Then call make with the board number: make BOARD_SEL=54
9. This will build the example resulting in a sciopta.elf file which you now can download and run on the target board.

10. If you have connected a serial line from the COM port of your host PC to the UART of your target board, open a terminal window on your PC and connect it to your selected PC COM port. Parameters are 115200Bd, 8 Bit, no parity, 1 stop bit, no flow-control.
11. Now you can check the log messages on your host terminal window.

3. Typical Kernel Project

3.1. Introduction

This chapter describes the procedures and tasks for designing and implementing a typical small SCIOPTA kernel project.

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 realtime operating system therefore specific care needs to be taken to follow the design rules for such systems. Data should always be maintained in SCIOPTA messages and shared resources should be encapsulated within SCIOPTA processes.

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

The SCIOPTA kernel has over 120 system calls. Some of these calls are very specific and are only used in particular situations. Thus many system calls are only needed if you are designing dynamic applications for creating and killing SCIOPTA objects. Other calls are exclusively foreseen to be used in CONNECTOR processes which are needed in distributed applications.

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

Please consult the SCIOPTA - Kernel, Reference Manual for detailed description of the SCIOPTA system calls. The SCIOPTA Architecture Manual gives a detailed introduction into SCIOPTA concepts and architecture.

The SCIOPTA building procedure consists of the following steps:

- Configuring the kernel by using the **SCONF** configuration tool. The **SCONF** tool generates the C source file sconf.c (system defines and start) and the include file sconf.h.
- Locate the include files and define the include paths.
- Assemble the kernel.
- Locate and get all assembler source files and assemble it.
- Locate and get all C source files and compile them.
- Design the linker script to map your system into the target memory.
- Select and define the correct libraries for the SCIOPTA Generic Device Driver (gdd) and Utilities (util) functions.
- Link the system.

The Getting Started project (see Sciopta Installation Manual chapter Getting Started Examples) is a good example for the needed files of a SCIOPTA application. This example project as a good starting point for your system design.

3.2. SCIOPTA Board Support Packages Structure

3.2.1. Introduction

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

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

1. General Device Drivers
2. Architecture Device Drivers
3. CPU Family Device Drivers
4. Board Device Drivers

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

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

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

3.2.2. General Device Drivers

General System Functions are functions which are common to all architectures, all CPUs and all boards.
It contains mainly include and source device drivers files for external (not on-chip) controllers.
Generic debugger files might also be placed here.

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

3.2.3. Architecture Device Drivers

Architecture System Functions are functions which are architecture (**arm** for Cortex-M and Cortex-R) specific and are common to all CPUs and all boards.
It contains generic linker script include files (**module.ld**), C startup files (**cstartup.S**) and other architecture files.

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

3.2.4. CPU Family Device Drivers

CPU Family System Functions are functions which are architecture (**arm**) and CPU family specific and are common to all boards.
It contains mainly include and source device drivers files for on-chip controllers.

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

Device drivers for specific SCIOPTA middleware products (IPS internet protocols, file systems, USB and embedded GUI) are not described here. Please consult the specific SCIOPTA middleware manuals.

3.2.5. Board System Functions

Board System Functions are functions which are architecture (**arm**), CPU family and board specific.
It contains mainly include, source and project files for board setup and debugger initialization files and linker scripts.

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

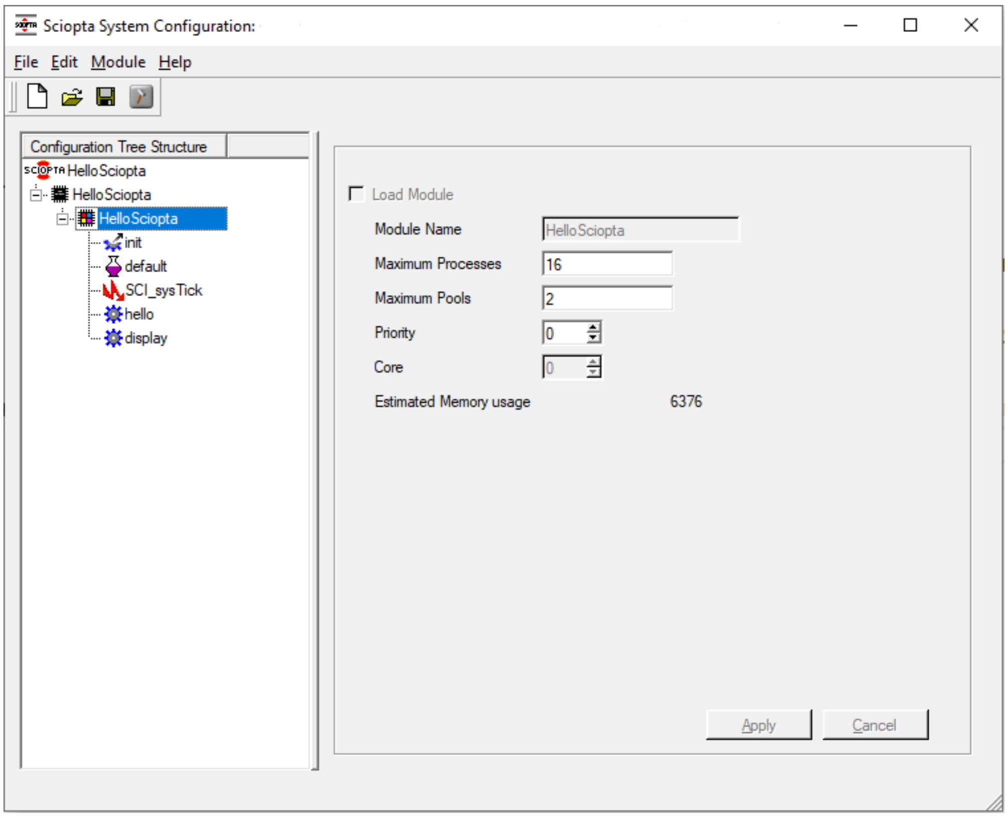
3.3. Kernel Configuration

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

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

For typical SCIOPTA examples and getting started projects kernel configuration files are included in the SCIOPTA deliveries.

For a small minimum system kernel example with one module containing one message pool (default), one interrupt process (SCI_sysTick needed for the SCIOPTA system tick) and two prioritized processes (hello and display) the SCONF main window looks as follows:



Load Module
Module Name: HelloSciopta
Maximum Processes: 16
Maximum Pools: 2
Priority: 0
Core: 0
Estimated Memory usage: 6376

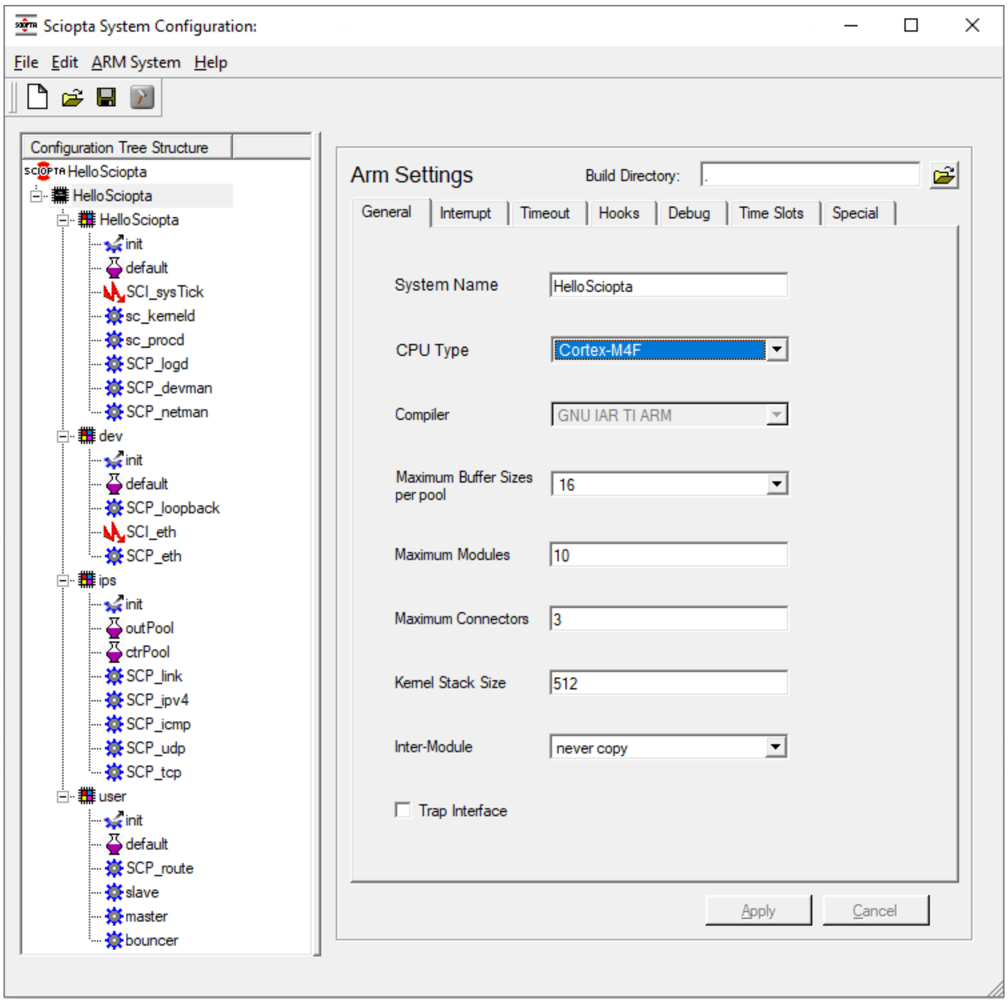
Figure 2. One-Module System

3.3.1. Files

sconf.c	SCIOPTA configuration file generated by **SCONF**. Location: <your_project_folder>
sconf.h	Kernel configuration include file generated by **SCONF**. Location: <your_project_folder>
your_project.xml	Kernel configuration settings of **SCONF**. Location: <your_project_folder> Typical kernel configuration files for small example projects (**hello.xml**) can be found in the SCIOPTA example delivery: Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

The kernel configuration (**sconf.c**) must be compiled and linked with the application.

Example of the SCONF main window of a multiple-module system.



System Name: HelloSciopta
CPU Type: Cortex-M4F
Compiler: GNU IAR TI ARM
Maximum Buffer Sizes per pool: 16
Maximum Modules: 10
Maximum Connectors: 3
Kernel Stack Size: 512
Inter-Module: never copy
Trap Interface: ☐

Figure 3. Multiple-Module System

3.4. SCIOPTA Real-Time Kernel

The SCIOPTA real-time kernels are provided in assembler source files. The files are specific for the different compiler environments. The kernel configuration file sconf.h will be included to adapt the kernel for the chosen configuration.

3.4.1. Files

sciopta.S	SCIOPTA kernel for GCC. Location: <install_folder>\sciopta\<version>\lib\arm\krn\
sciopta_iar.s	SCIOPTA kernel for IAR. Location: <install_folder>\sciopta\<version>\lib\arm\krn\
sciopta_ads.s	SCIOPTA kernel for ARM DS-5. Location: <install_folder>\sciopta\<version>\lib\arm\krn\
sciopta.asm	SCIOPTA kernel for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\lib\arm\krn\

The SCIOPTA real-time kernel must be assembled and linked with the application.

3.5. Exception Handler

3.5.1. Description

Exceptions are taken whenever the normal flow of a program must temporarily halt, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the processor preserves the critical parts of the current processor state so that the original program can resume when the handler routine has finished.

The exception handler handles all exceptions which are not specifically treated in SCIOPTA. Theses vectors contain the entry address of the exception handler (**exception_handler**).

All other exceptions are handled by specific SCIOPTA functions. The vectors contain the specific function addresses (such as kernel start, software interrupt handling and others). Please look into the exception handler source file for more information.

The maximum number of external interrupt vectors is defined by **SC_MAX_INT_VECTORS** which is configured by SCONF (see chapter General System Configuration TAB).
The exception vector table is located in the exception vector table file. See chapter Interrupt Vector Table.

3.5.2. Files

The following files contain the exception handler for ARM architectures. Do not modify these files.

cortexm0_exception.S	Cortex-M0 Exception handler for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
cortexm3_exception.S	Cortex-M3/M4 Exception handler for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
exception.S	All ARM except Cortex-M exception handler for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
cortexm0_exception.s	Cortex-M0 Exception handler for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\src\iar\
cortexm3_exception.s	Cortex-M3/4 Exception handler for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\src\iar\
exception.s	All ARM except Cortex-M exception handler for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\src\iar\
cortexm0_exception.s	Cortex-M0 Exception handler for ARM DS-5. Location: <install_folder>\sciopta\<version>\bsp\arm\src\arm\
exception.s	All ARM except Cortex-M exception handler for ARM DS-5. Location: <install_folder>\sciopta\<version>\bsp\arm\src\arm\
cortexm3_exception.asm	Cortex-M3/M4 Exception handler for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\bsp\arm\src\ccs\
exception.asm	All ARM except Cortex-M exception handler for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\bsp\arm\src\ccs\

3.5.3. Syntax

```
void exception_handler(void);
```

3.5.4. Exception Handler User Errors

The exception handler is generating specific error by calling **sc_miscError** for the defined exception.
Please consult the SCIOPTA Kernel Reference Manual for more information about user errors (**sc_miscError**) system call and SCIOPTA error hook.

Error Word	Extra Value
KERNEL_EILL_EXCEPTION | SC_ERR_FATAL | 0x01000000 A not defined exception was detected.	Register save area.

Please see file **error.c** (location: <install_folder>\sciopta\<version>\exp\arm\common\) for more information of the extra value (register save area).

3.6. Interrupt Vector Table

This file contains the Cortex-M interrupt vector table. Please consult the source files for detailed description of the vector table.

3.6.1. Files

cortexm3_vector.S	Interrupt vector table for Cortex-M3/M4 for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
cortexm0_vector.s	Interrupt vector table for Cortex-M0 for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\src\iar\
cortexm3_vector.s	Interrupt vector table for Cortex-M3/M4 for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\src\iar\
cortexm0_vector.s	Interrupt vector table for Cortex-M0 for ARM DS-5. Location: <install_folder>\sciopta\<version>\bsp\arm\src\arm\

For some Cortex-M microcontrollers a vector table is available in C:

vectors.c	Interrupt vector table written in C. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>src\

3.7. Interrupt Handler

All ARM processors except Cortex-M need an interrupt handler which is called at each interrupt.

The interrupt handler will do the following:

1. Execute an interrupt prologue
2. Get the interrupt number
3. Call the interrupt dispatcher of the kernel (**sc_sysIrqDispatcher**)
4. Execute an interrupt epilogue

Some interrupt handlers may contain additional early startup code and the vector table. This might also be placed in a separate boot file (boot.*).
The interrupt prologue/epilogue macros are included by the interrupt handler.

3.7.1. Files

irq_handler.S	Interrupt handler for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\src\gnu\
boot.S	Interrupt handler boot code for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\src\gnu\
irq.S	Interrupt prologue/epilogue macros for GCC. Location: <install_folder>\sciopta\<version>\include\machine\arm\
irq_handler.s	Interrupt handler for for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\src\iar\
boot.s	Interrupt handler boot code for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\src\iar\
irq_handler.asm	Interrupt handler for for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\src\ccs\
boot.asm	Interrupt handler boot code for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\src\ccs\
irq_ccs.inc	Interrupt prologue/epilogue macros for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\include\machine\arm\

3.8. System Tick Driver

3.8.1. Introduction

A SCIOPTA systems needs always to setup a system tick. The kernel contains a tick function (**sc_tick()**) which maintains a counter to control the timing functions. This function should be called at regular intervals by an external function. At each call the kernel internal tick counter is incremented by 1.

The system tick driver is usually an interrupt process which calls **sc_tick()**.

For ARM the included system tick driver is based on the SysTick of the specific CPU.

The drivers might include definition and source files of CPU manufacturer delivered firmware. If this is the case, these files must be included in the build process. Please consult the driver source and include files.

3.8.2. Files

systick.c	System tick driver. **Freescale® Cortex-M4, Kinetis-K:** location: <install_folder>\sciopta\<version>\bsp\arm\kinetis-k\src\ **NXP® Cortex-M3, LPC17xx:** location: <install_folder>\sciopta\<version>\bsp\arm\lpc17xx\src\ **NuMicro® Cortex-M0, NUC1xx:** location: <install_folder>\sciopta\<version>\bsp\arm\nuc1xx\src\ **Texas Instrument® Cortex-M3, Stellaris:** location: <install_folder>\sciopta\<version>\bsp\arm\stellaris\src\ **STMicroelectronics® Cortex-M3/4:** STM32: location: <install_folder>\sciopta\<version>\bsp\arm\stm32\src\ STM32F2xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32f2\src\ STM32F4xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32f4\src\ STM32L1xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32l1\src\ **Texas Instrument® Cortex-R4, TMS570:** location: <install_folder>\sciopta\<version>\bsp\arm\tms570\src\ **Texas Instrument® Cortex-R4, RM4:** location: <install_folder>\sciopta\<version>\bsp\arm\rm48\src\
config.h	Configuration definitions and settings included by the driver. This file is application specific. Examples can be found here: <install_folder>\sciopta\<version>\bsp\arm\<architecture>\<board>\include\

3.8.3. System Tick Driver Interrupt Process

The system tick driver consists of a static interrupt process called **SCI_sysTick**. As a static process it must be defined in the **SCONF** kernel configuration utility (see below chapter SCONF System Tick Interrupt Process Configuration). Therefore the process will be automatically created at system start.

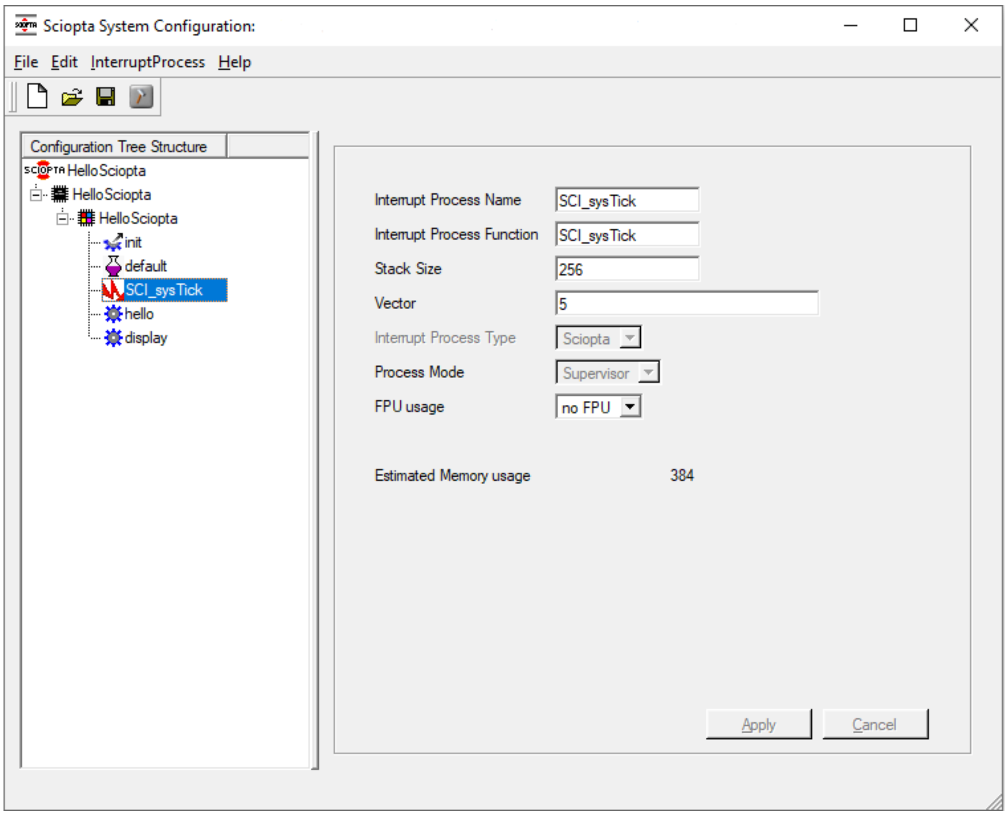
At process creation (source parameter **src** equal **SC_PROC_WAKEUP_CREATE**) the driver will initialize the the SysTick for the tick timing functions and enable the interrupt. The length of the tick interval will be calculated by including the **CNF_TICKLENGTH** value from the file config.h. The tick length will be stored in the kernel by calling

the **sc_tickLength** function. At every hardware interrupt (source parameter **src** equal **SC_PROC_WAKEUP_HARDWARE**) the SCIOPTA system call sc_tick will be called and the interrupt reloaded. Therefore **sc_tick** will be called every **CNF_TICKLENGTH**.
Please consult the **systick.c** source file for more information.

There are no specific system tick driver messages and there is no specific system tick driver function interface. The system tick driver generates no user errors.

3.8.4. System Tick Driver Configuration

3.8.4.1. SCONF System Tick Interrupt Process Configuration



Interrupt Process Name: SCI_sysTick
Interrupt Process Function: SCI_sysTick
Stack Size: 256
Vector: 5
Interrupt Process Type: Sciopta
Process Mode: Supervisor
FPU usage: no FPU

Estimated Memory usage: 384

Figure 4. System Tick Driver Interrupt Process Configuration Parameters

Use these configuration parameters for the included system tick driver (**systick.c**). If you have modified the driver the parameters might need to be adapted (e.g. larger stack).

3.8.4.2. Tick Length

The tick **length CNF_TICKLENGTH** must be defined in microseconds. Recommended values range from 1000 us to 10'000 us. In the delivered SCIOPTA BSP **CNF_TICKLENGTH** is defined in a **config.h** file which will be included by the driver file **systick.c**.

3.8.5. Using the System Tick Driver

Once configured and (automatically) started by the kernel there is no user intervention required. The driver is interrupt driven and runs continuously.

3.9. Simple UART Driver

3.9.1. Introduction

The SCIOPTA standard delivery contains a **kprintf** function which allows application programs (processes) to output status information. The **kprintf** function is sending characters over a user supplied __putchar function. The **kprintf** function is included in the **kprintf.c** file (see below) or in the **utilities** libraries.
This simple UART driver contains low-level functions to initialize the standard UART and to send and to receive characters over it.

The **__putchar** function must call the simple UART output function (**uart_putchar**). The **uart_putchar** function is not interrupt driven.

The drivers might include definition and source files of CPU manufacturer delivered firmware. If this is the case, these files must be included in the build process. Please consult the driver source and include files.

3.9.2. Files

simple_uart.c	Simple UART driver. **Freescale® Cortex-M4, Kinetis-K:** location: <install_folder>\sciopta\<version>\bsp\arm\kinetis-k\src\ **NXP® Cortex-M3, LPC17xx:** location: <install_folder>\sciopta\<version>\bsp\arm\lpc17xx\src\ **NuMicro® Cortex-M0, NUC1xx:** location: <install_folder>\sciopta\<version>\bsp\arm\nuc1xx\src\ **STMicroelectronics® Cortex-M3/4:** STM32: location: <install_folder>\sciopta\<version>\bsp\arm\stm32\src\ STM32F2xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32f2\src\ STM32F4xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32f4\src\ STM32L1xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32l1\src\ **Texas Instrument® Cortex-R4, TMS570:** location: <install_folder>\sciopta\<version>\bsp\arm\tms570\src\ **Texas Instrument® Cortex-R4, RM4:** location: <install_folder>\sciopta\<version>\bsp\arm\rm48\src\

simple_uart.h	Simple UART driver definitions. **Freescale® Cortex-M4, Kinetis-K:** location: <install_folder>\sciopta\<version>\bsp\arm\kinetis-k\include\ **NXP® Cortex-M3, LPC17xx:** location: <install_folder>\sciopta\<version>\bsp\arm\lpc17xx\include\ **NuMicro® Cortex-M0, NUC1xx:** location: <install_folder>\sciopta\<version>\bsp\arm\nuc1xx\include\ **STMicroelectronics® Cortex-M3/4:** STM32: location: <install_folder>\sciopta\<version>\bsp\arm\stm32\include\ STM32F2xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32f2\include\ STM32F4xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32f4\include\ STM32L1xx: location: <install_folder>\sciopta\<version>\bsp\arm\stm32l1\include\ **Texas Instrument® Cortex-R4, TMS570:** location: <install_folder>\sciopta\<version>\bsp\arm\tms570\src\ **Texas Instrument® Cortex-R4, RM4:** location: <install_folder>\sciopta\<version>\bsp\arm\rm48\src\
kprintf.c	Minimal printf. A function __putchar needs to be provided. Location: <install_folder>\sciopta\<version>\util\logd\
system.c	Startup functions for the system module. It includes the common function **__putchar()**. This file is application specific. Examples can be found here: Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

3.9.3. Simple UART Driver Functions

There is no specific process for the simple UART driver. The driver consists of the functions **uart_init**, **uart_putchar** and **uart_getchar**.

The **uart_init** initializes the peripheral to be used for the simple uart functions. Besides the initialization of the USART units the baudrate can be defined. The **uart_init** function is usually called from the SCIOPTA **start hook**.

The **uart_putchar** function puts a character to the UART to be sent over a serial line.

The **uart_getchar** function gets a character from the UART received from a serial line.

3.9.4. Simple UART Driver Configuration

The driver will be configured by calling the uart_init function for the selected unit. See function interface below.

3.9.5. Simple UART Driver Function Interface

3.9.5.1. uart_init

3.9.5.1.1. Description

This function is used to initialize the UART interfaces to be used for the simple uart functions.

3.9.5.1.2. Syntax

```
void uart_init (
  unsigned int unit,
  unsigned int baudrate
);
```

3.9.5.1.3. Parameters

unit	UART unit
baudrate	Baudrate to be used in the UART unit

3.9.5.1.4. Return Value

None.

3.9.5.2. uart_putchar

3.9.5.2.1. Description

This function is used to put a character to the UART to be sent over a serial line.

3.9.5.2.2. Syntax

```
void uart_putchar (
  int unit,
  int ch
);
```

3.9.5.2.3. Parameters

unit	UART unit
ch	Character to be sent

3.9.5.2.4. Return Value

None.

3.9.5.3. uart_getchar

3.9.5.3.1. Description

This function is used to get a character from the UART controller received from a serial line.

3.9.5.3.2. Syntax

```
int uart_getchar(
  int unit
);
```

3.9.5.3.3. Parameters

unit	UART unit

3.9.5.3.4. Return Value

Received character.

3.9.6. Simple UART Driver User Errors

The SCIOPTA Simple UART Driver does not generate specific user errors.

3.9.7. Using the Simple UART Driver

3.9.7.1. Simple UART Driver Configuration

To configure the simple UART driver use the **uart_init** function described in chapter uart_init.

A preferred location to include the call to the **uart_init** function and therefore to define the baudrate is the SCIOPTA start hook. In the standard SCIOPTA deliveries the start hook is included in the file **system.c**.

3.10. LED Feedback Functions

3.10.1. Description

If a standard board is equipped with status LEDs the **__putchar()** function might using these LEDs to change status after a full line has been sent out.
There is no specific process for the LED feedback functions.

The driver consists of the functions **initFeedback** and **feedback**.

The **initFeedback** function initializes the MCU pins which are connected to the board LEDs. It is usually called from the SCIOPTA **start hook**.

The **feedback** function will display at every call another set of LEDs on the board.

The **initFeedback** and **feedback** functions are usually included in the file **led.c** located in the BSP.

3.10.2. Files

led.c	Board LED functions. Location: <install_folder>\sciopta\<version>\bsp\arm\<arch>\<board>\src\
led.h	Board LED functions header file. Location: <install_folder>\sciopta\<version>\bsp\arm\<arch>\<board>\include\

3.11. __putchar

The function **__putchar** is included in **system.c** which is used by kprintf.c and uses the simple UART device driver (**uart_putchar**). It can be adapted for your project.

Example:

```
extern void feedback();
void __putchar(int c)
{
  if ( c == '\n' ){
    uart_putchar(CNF_DBGU, '\r');
    feedback();
  }
  uart_putchar(CNF_DBGU, c);
}
```

3.11.1. Files

system.c	System start functions for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

3.12. Board Configuration

3.12.1. Board Configuration File

This file is usually named **config.h** and is a board specific file containing CPU, board and application specific definitions.

3.12.2. Files

config.h	Configuration definitions and settings included by SCIOPTA drivers. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu_family>\<board>\include\

3.13. System Start Functions

3.13.1. Reset Hook

The reset hook is called by the kernel after the reset. Please consult Sciopta Architecture Manual chapter Start Sequence for a detailed description of the SCIOPTA start sequence and chapter Reset Hook for more information about the reset hook.

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

The reset hook must be adapted for your project. usually the files **boardsetup.c** or **resethook.c** contain the reset hook.

3.13.2. C Startup

The ARM C startup is called by the kernel directly after the reset hook. It is initializing the C system and replaces the GCC library crt0. Please consult Sciopta Architecture Manual chapter Start Sequence for a detailed description of the SCIOPTA start sequence and chapter C Startup for more information about C startup. Do not modify this file.

3.13.2.1. Files

cortexm0_cstartup.S	Cortex-M0 Exception handler for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
cortexm3_cstartup.S	Cortex-M3/4 C startup code for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
cortexm0_cstartup.s	Cortex-M0 C startup code for ARM DS-5. Location: <install_folder>\sciopta\<version>\bsp\arm\src\arm\
cstartup.S	C startup code for all ARM excluding Cortex-M for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\

The Cortex-M C startup code must be assembled and linked with the application.
For IAR Embedded Workbench there is no separate C startup function needed.

3.13.3. Start Hook

The start hook is called by the kernel after the reset hook and C initialization but before the kernel has been started. Please consult Sciopta Architecture Manual chapter Start Sequence for a detailed description of the SCIOPTA start sequence and chapter Start Hook on page 111 for more information about the start hook.

In SCIOPTA a start hook must always be present and must have the name **start_hook**.

Example start hooks are included in the SCIOPTA example deliveries in the file **system.c**. It can be adapted for your project.

3.13.3.1. Files

system.c	System start functions for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

The start hook must be compiled and linked to the application.

3.13.3.2. Example

```
void start_hook(void)
{
  /* Insert functions here that should run before Sciopta starts */
  uart_init(CNF_DBGU,115200);
  initFeedback();
  kprintf(0,CLS"STM3210E-EVAL up and running\n");
#if SC_ERR_HOOK == 1
  sc_miscErrorHookRegister(error_hook);
#endif
}
```

Please note:

- The call to **uart_init** initializes the serial interface to be used by simple_uart (kprintf). See chapter uart_init.
- The call to **initFeedback** initializes the LED driver of a board having LEDs. See chapter LED Feedback Functions.
- The global error hook must be registerred in the start_hook. The macro **SC_ERROR_HOOK** is defined if the error hook checkbox has be selected in the hooks configuration TAB of the kernel configuration (see chapter Hooks Configuration TAB).

3.13.4. System Module Start Function

The **system module start function** is called by the init process of the systems module.

Please consult Sciopta Architecture Manual chapter Start Sequence for a detailed description of the SCIOPTA start sequence and chapter Start Function of the System Module for more information about the system module start function.

In SCIOPTA the **system module start function** must always be present and must have the same name as the system module. An example **system module start function** (which does nothing) is included in the SCIOPTA example delivery in the file **system.c**. It can be adapted for your project.

```
void HelloSciopta(void)
{
  /* Place further code executed at init priority */
}
```

Please note:

- The name of the system module is **HelloSciopta**.

3.13.4.1. Files

system.c	System start functions for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

3.13.5. User Modules Start Function

All other modules (except the system module) have also own individual module start functions.

Please consult Sciopta Architecture Manual chapter Start Sequence for a detailed description of the SCIOPTA start sequence and chapter Start Function of User Modules for more information about the user module start function.

These functions must have the same name as the module. The user modules start functions will be called by the init Process of the module.

After returning from the module start functions the init Processes of the module will change its priority to 32 and go into sleep.

These start functions can use all SCIOPTA system calls.

3.14. Error Handling

3.14.1. Error Hook

It is good design practice in SCIOPTA to include an error hook in the application.

Please consult Sciopta Architecture Manual chapter Error Hook for a detailed description about error hooks.

In order to use an error hook the error hook checkbox must be selected in the hooks configuration TAB of the kernel configuration (see chapter Hooks Configuration TAB).

The error hook must be registered by the sc_miscErrorHookRegister system call. This can be done in the SCIOPTA start hook (see chapter Start Hook) for the global error hook. Local error hooks should be registered in the module init function (user module start function) or in a user process.

An example error hook is included in the delivery. It can be adapted for your project.

3.14.1.1. Files

error.c	Basic error hook. Location: <install_folder>\sciopta\<version>\exp\krn\arm\common\

3.15. Writing the User Application

These files contain the user processes, messages and functions for your application.
For the simple kernel example (hello) the following files are used.

3.15.1. Files

hello.c	Process “hello” for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\common\hello\.
display.c	Process “display” for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\common\hello\
hello.msg	Message declarations for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\common\hello\

3.15.2. SCIOPTA Design Rules

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

- Correct designed SCIOPTA systems should use only a few priority levels. When designing a system avoid to control it with priorities. A system should be controlled by message passing and message flow. Priorities should be used to guarantee fast response time to external events.
- If you identify work which is concurrent do not try to place the code in one process. Simultaneous work should be placed in different processes.
- Avoid to send a lot of messages from a process without waiting for reply messages. The receiving process might not be activated until the sender process becomes not ready.
- Methods and functions which will be accessed from more than one process must be re-entrant while executing. There are system calls to handle per-process local data (sc_procVar*).
- To simplify the calculation of stack requirements, try to avoid using of large auto arrays in processes written in C. Rather allocate a buffer from a message pool.
- I/O-ports must be encapsulated in a SCIOPTA process. Otherwise they must be treated the same way as global variables.
- As it is true for all well designed systems, it is strongly recommended to not using global variables. If it cannot be avoided you must disable interrupts or lock the scheduler while accessing them.
- Always include an Error-hook in your system. Setting a breakpoint there allows you to track down system errors easily.
- Do not modify message data (buffers) after you have sent it.

3.16. SCIOPTA Daemons

3.16.1. Process Daemon

The process daemon (**sc_procd**) is identifying processes by name and supervises created and killed processes. Whenever you are using the sc_procIdGet system call you need to start the process daemon.
The process daemon is part of the kernel. But to use it you need to define and declare it in the SCONF configuration utility.
The process daemon can only be created and placed in the system module.

Please consult Sciopta Architecture Manual chapter Process Daemon for more information about the process daemon.

3.16.2. Kernel Daemon

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

Whenever you are using the **sc_moduleCreate**, **sc_moduleKill**, **sc_procPrioCreate**, **sc_procIntCreate**, **sc_procTimCreate** and **sc_procKill** system calls 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 can only be created and placed in the system module.

Please consult Sciopta Architecture Manual chapter Process Daemon for more information about the kernel daemon.

3.17. Include Files

3.17.1. Kernel Include Files Search Directories

Please make sure that the environment variable **SCIOPTA_HOME** is defined as explained in chapter SCIOPTA_HOME Environment Variable.
Define the following entries the include files search directories field of your IDE:

```
%(SCIOPTA_HOME)\include
%(SCIOPTA_HOME)\include\sciopta\arm
```

3.17.2. CPU-Family and BSP Include Files Search Directories

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

```
%(SCIOPTA_HOME)\bsp\arm\include
%(SCIOPTA_HOME)\bsp\arm\<cpu-family>\include
%(SCIOPTA_HOME)\bsp\arm\<cpu-family>\<board>\include
```

3.17.3. CPU-Family Firmware Include File Search Directories

If files of CPU manufacturer delivered firmware is used, specific file search directories might be defined.
Please consult the BSP driver source and include files and the CPU manufacturers documentation.

3.17.4. Main Include file sciopta.h and sciopta_sc.h

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

sciopta.h	Main SCIOPTA include file. File location: <installation_folder>\sciopta\<version>\include\
sciopta_sc.h	Main SCIOPTA include file using the trap interface. File location: <installation_folder>\sciopta\<version>\include\

3.17.5. Configuration Definitions sconf.h

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

The file **sconf.h** needs to be included if for instance you want to know the maximum number of modules allowed in a system. This information is stored in **SC_MAX_MODULES** in the file **sconf.h**.
Please remember that **sconf.h** is automatically generated by the **SCONF** configuration tool and placed at a convenient location in your project folder.

3.17.6. Main Data Types types.h

These types are introduced to allow portability between various SCIOPTA implementations.
The main data types are defined in the file types.h. These types are not target processor dependent.

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

3.17.7. Architecture Dependent Data Types types.h

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

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

3.17.8. 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\

3.17.9. Board Configuration

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

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

3.18. GDD Libraries

SCIOPTA device drivers are often using some global device driver functions (GDD Functions) such as **sdd_devOpen, sdd_devWrite** etc. The sources of these functions are included in the standard SCIOPTA delivery (**<install_folder>\sciopta\<version>\gdd\sdd...**).

Also the code for the SCIOPTA device manager (**manager.c**) is included in gdd. The device manager maintains devices if the SCIOPTA device driver concept is used.

Please consult the **SCIOPTA GDD and Utilities User's and Reference Manual** for a detailed description of the GDD interface.

Additional to the source code delivery of the gdd function interface there are also pre-built libraries included. If you are using SCIOPTA GDD (sdd) functions or using the SCIOPTA device manager you need to include the libraries for the correct CPU family into your build system.

3.18.1. GDD Library Files Locations

The GDD libraries are located here:
<install_folder>\sciopta\<version>\lib\arm\<compiler>_<arch>\

3.18.2. GDD Libraries for GCC and ARM Architecture

3.18.2.1. GDD GCC Library File Names

FPU	Endian	Trap Interface	No Optimization	Optimized for size	Optimized for speed
No	little	No	libgdd_0.a	libgdd_1.a	libgdd_2.a
		Yes	libgdd_0t.a	libgdd_1t.a	libgdd_2t.a
	big	No	libgdd_0b.a	libgdd_1b.a	libgdd_2b.a
		Yes	libgdd_0tb.a	libgdd_1tb.a	libgdd_2tb.a

Please Note: There are no big endian library files included in the Cortex-M delivery.

3.18.2.2. Building the GDD GCC Libraries

The standard SCIOPTA delivery contains makefiles to build the gdd libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

Procedures to generate the libraries:

1. Navigate to the gdd route source directory: <installation_folder>\sciopta\<version>\gdd\
2. Execute the makefiles:
   - **Makefile.cm3** gdd for GCC and ARM Cortex-M3
   - **Makefile.cm4f** gdd for GCC and ARM Cortex-M4 with floating point support
   - **Makefile.rf** gdd for GCC and ARM Cortex-R with floating point support
   - **Makefile.rfb** gdd for GCC and ARM Cortex-R with floating point support, big endian
3. The libraries will be installed in the directory:
   <installation_folder>\sciopta\<version>\lib\arm\<arch>\

3.18.3. GDD Libraries for IAR and ARM Architecture

3.18.3.1. GDD IAR Library File Names

FPU	Endian	Trap Interface	No Optimization	Optimized for size	Optimized for speed
No	little	No	gdd_0.a	gdd_1.a	gdd_2.a
		Yes	gdd_0t.a	gdd_1t.a	gdd_2t.a
	big	No	gdd_0b.a	gdd_1b.a	gdd_2b.a
		Yes	gdd_0tb.a	gdd_1tb.a	gdd_2tb.a

Please Note: There are no big endian library files included in the Cortex-M delivery.

3.18.3.2. Building the GDD IAR Libraries

The standard SCIOPTA delivery contains makefiles to build the gdd libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.

Procedures to generate the libraries:

1. Navigate to the gdd route source directory: <installation_folder>\sciopta\<version>\gdd\
2. Execute the makefiles:
    - **Makefile.cm0iar** gdd for IAR and ARM Cortex-M0
    - **Makefile.cm3iar** gdd for IAR and ARM Cortex-M3
    - **Makefile.cm4iar** gdd for IAR and ARM Cortex-M4
    - **Makefile.cm4fiar** gdd for IAR and ARM Cortex-M4 with floating point support
    - **Makefile.rfiar** gdd for IAR and ARM Cortex-R with floating point support
    - **Makefile.rfbiar** gdd for IAR and ARM Cortex-R with floating point support, big endian
3. The libraries will be installed in the directory:
   <installation_folder>\sciopta\<version>\lib\arm\<arch>\

3.19. Utilities Libraries

The standard SCIOPTA delivery contains some often used utilities such as **log daemon, kprintf, shell** and others. The sources of the functions are located here: **<install_folder>\sciopta\<version>\util...**).
Please consult the **SCIOPTA GDD and Utilities User's and Reference Manual** for a detailed description of the SCIOPTA utilities.

Additional to the source code delivery of the utilities functions there are also pre-built libraries included. If you are using SCIOPTA utilities you need to include the libraries for the correct CPU family into your build system.

3.19.1. Utilities Library Files Locations

The utilities libraries are located here:
<install_folder>\sciopta\<version>\lib\arm\<compiler>_<arch>\

<compiler> = **gnu** for GCC, **iar** for IAR, **ccs** for TI Code Composer Studio, **rv41** for ARM DS-5
<arch> = **cm0** ¦ **cm3** ¦ **cm4** ¦ **cm4f**
<arch> = **v7rf** for Cortex-R

3.19.2. Utilities Libraries for GCC and ARM Architecture

3.19.2.1. Utilities GCC Libraries File Names

FPU	Endian	Trap Interface	No Optimization	Optimized for size	Optimized for speed
No	little	No	libutil_0.a	libutil_1.a	libutil_2.a
		Yes	libutil_0t.a	libutil_1t.a	libutil_2t.a
	big	No	libutil_0b.a	libutil_1b.a	libutil_2b.a
		Yes	libutil_0tb.a	libutil_1tb.a	libutil_2tb.a

Please Note: There are no big endian library files included in the Cortex-M delivery.

3.19.2.2. Building the Utilities GCC Libraries

The standard SCIOPTA delivery contains makefiles to build the util libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.
Procedures to generate the libraries:

1. Navigate to the gdd route source directory: <installation_folder>\sciopta\<version>\util\
2. Execute the makefiles:
   - **Makefile.cm3** utilities for GCC and ARM Cortex-M3
   - **Makefile.cm4f** utilities for GCC and ARM Cortex-M4 with floating point support
   - **Makefile.rf** utilities for GCC and ARM Cortex-R with floating point support
   - **Makefile.rfb** utilities for GCC and ARM Cortex-R with floating point support, big endian
3. The libraries will be installed in the directory:
   <installation_folder>\sciopta\<version>\lib\arm\<arch>\

3.19.3. Utilities Libraries for IAR and ARM Architecture

3.19.3.1. Utilities IAR Library File Names

FPU	Endian	Trap Interface	No Optimization	Optimized for size	Optimized for speed
No	little	No	util_0.a	util_1.a	util_2.a
		Yes	util_0t.a	util_1t.a	util_2t.a
	big	No	util_0b.a	util_1b.a	util_2b.a
		Yes	util_0tb.a	util_1tb.a	util_2tb.a

Please Note: There are no big endian library files included in the Cortex-M delivery.

3.19.3.2. Building the Utilities IAR Libraries

The standard SCIOPTA delivery contains makefiles to build the util libraries yourself. If you want to change compiler switches or other system settings you need to modify the makefiles.
Procedures to generate the libraries:

1. Navigate to the gdd route source directory: <installation_folder>\sciopta\<version>\util\
2. Execute the makefiles:
   - **Makefile.cm0iar** utilities for IAR and ARM Cortex-M0
   - **Makefile.cm3iar** utilities for IAR and ARM Cortex-M3
   - **Makefile.cm4iar** utilities for IAR and ARM Cortex-M4
   - **Makefile.cm4fiar** utilities for IAR and ARM Cortex-M4 with floating point support
   - **Makefile.rfiar** utilities for IAR and ARM Cortex-R with floating point support
   - **Makefile.rfbiar** utilites for IAR and ARM Cortex-R with floating point support, big endian
3. The libraries will be installed in the directory:
   <installation_folder>\sciopta\<version>\lib\arm\<arch>\

3.20. Building ARM Systems with GCC and Makefile

For all delivered SCIOPTA examples there are Makefiles included to build the systems with GCC.
You can use any suitable editor to write the application files.

3.20.1. Environment Variables

The following environment variables need to be defined:

- **SCIOPTA_HOME** needs to point to the SCIOPTA delivery. Please consult SCIOPTA Installation Manual chapter SCIOPTA_HOME Environment Variable for more information.
- Be sure that the GCC compiler bin directory is included in the PATH environment variable.
- Include the SCIOPTA bin directory in the PATH environment variable as described in SCIOPTA Installation Manual chapter Setting SCIOPTA PATH Environment Variable.

3.20.2. GCC Project Files

Project file examples can be found in the getting started projects included in the standard SCIOPTA delivery.

Makefile	Makefile file for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\

3.20.3. GCC Linker Script

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.

GCC Linker script examples can be found in the standard SCIOPTA board support package deliveries.

3.20.3.1. Files

<board_name>.ld	GCC linker script example for a specific CPU and board. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

Usually the main linker scripts includes another generic linker script:

module.ld	Generic module linker script for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\include\

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

3.20.3.2. Memory Regions

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

The following regions are typically defined by the user:

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

3.20.3.3. Module Sizes

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

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

The module name for the system module is system.

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

```
MEMORY
{
  rom (rx) : ORIGIN = 0x08000000, LENGTH = 1M
  SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 192K
  system_mod : ORIGIN = 0x20000000, LENGTH = 20K
  dev_mod : ORIGIN = 0x20000000+20K, LENGTH = 32K
  ips_mod : ORIGIN = 0x20000000+20K+32K, LENGTH = 20K
  user_mod : ORIGIN = 0x20000000+20K+32K+20k, LENGTH = 20K
  sfs_mod : ORIGIN = 0x20000000+20K+32K+20k+20k, LENGTH = 0
}
/*
** Make above sizes available as symbol.
*/
system_size = LENGTH(system_mod);
dev_size = LENGTH(dev_mod);
ips_size = LENGTH(ips_mod);
user_size = LENGTH(user_mod);
sfs_size = LENGTH(sfs_mod);
```

Note: sfs_mod (file system) is not used in this example and therefore the length is set to zero.

Module size calculation:

size_mod = **p** * **128** + **stack** + **pools** + **mcb** + **initsize**

where:

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

3.20.3.4. Module Section Definitions

The module section definitions are contained in the file **module.ld** which is included by the linker script.

.<module>_text	RAM copy for rodata and text if wanted.
<module>_data	r/w data
.<module>_bss	BSS
.<module>_free	free RAM to be used by Sciopta for runtime allocated memory (pools, pcbs, stacks)

For each module the following sections must be defined in module.ld:

<module>_start	Start address of module-RAM
<module>_initsize	Size of initialized RAM (that is the size *_text + *_data + *_bss)
<module>_size	Complete size of module. This is defined in the main linker script and should be equal to the memory section size (**<module>_mod**).
<module>_mod	A structure that holds the above three.

3.20.3.5. Module Memory Map for GCC

Example of a system memory map containing the system module and one user module.

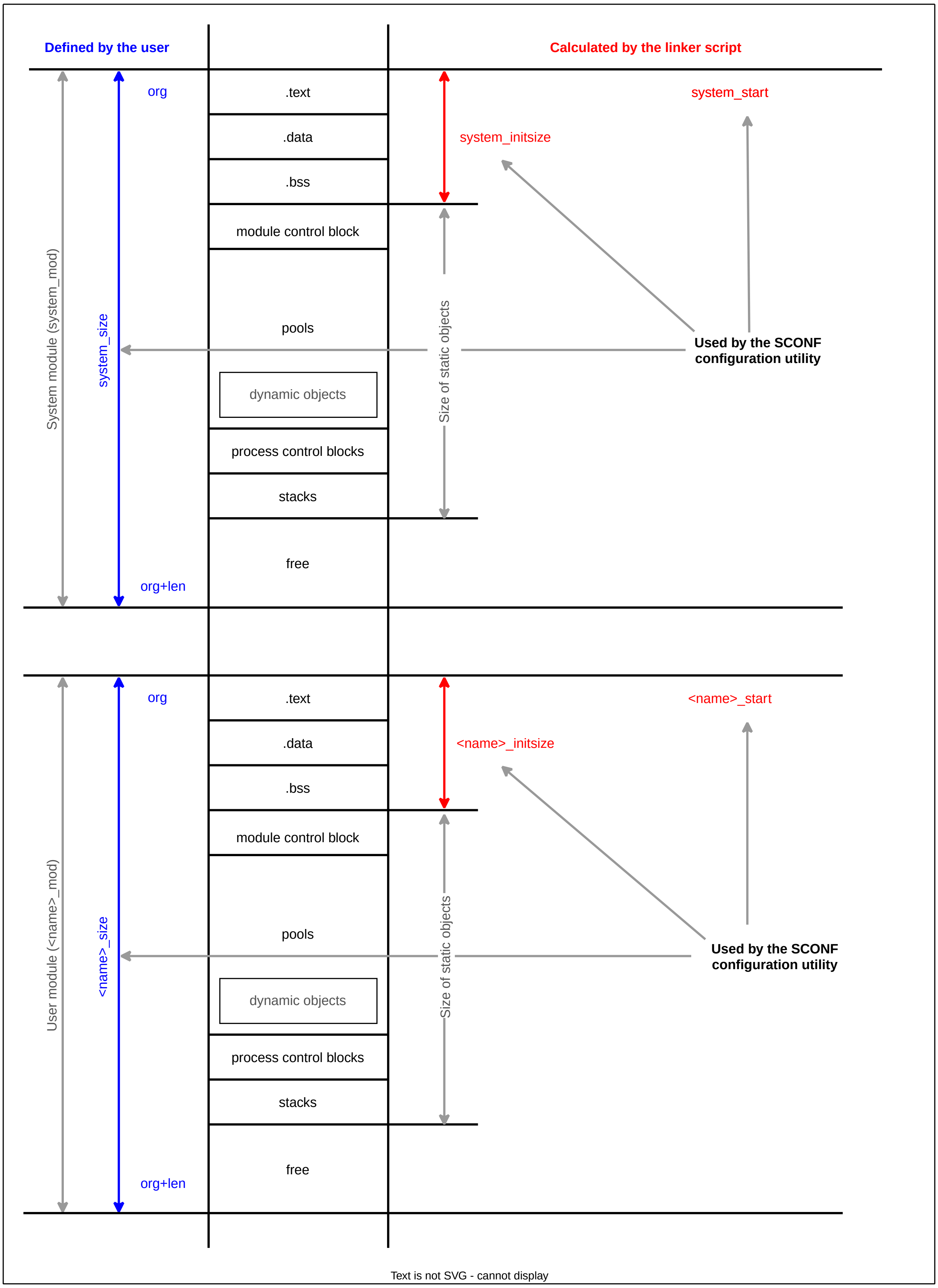


Figure 5. SCIOPTA Memory Map for GNU GCC

3.21. Building ARM Systems with IAR Embedded Workbench

IAR Embedded Workbench (EW) 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.

3.21.1. Environment Variables

The following environment variables need to be defined:

- **SCIOPTA_HOME** needs to point to the SCIOPTA delivery. Please consult SCIOPTA Installation Manual chapter SCIOPTA_HOME Environment Variable for more information.
- Include the SCIOPTA bin directory in the PATH environment variable as described in SCIOPTA Installation Manual chapter Setting SCIOPTA PATH Environment Variable. This will give access to the sconf.exe utility.

3.21.2. IAR EW for ARM Project Files

Project file examples can be found in the getting started projects included in the standard SCIOPTA delivery.

***.eww**	IAR EW ARM **workspace** file for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\
***.ewp**	IAR EW ARM **project** file for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\
***.ewd**	IAR EW ARM **debugger** file for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

3.21.3. IAR EW for ARM C-SPY Board Setup File

C-SPY is the name of the IAR Embedded Workbench debugger. Some examples can be found in the standard SCIOPTA board support package deliveries (not for all boards).

<board>.mac	IAR EW C-SPY board setup example. Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

3.21.4. IAR EW for ARM Linker Script

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.

Example of a IAR EW ARM linker script for a simple application:

```
/*-Memory Regions-*/
define memory mem with size = 4G;
/*-Specials-*/
define symbol _intvec_start__ = 0x00000000;
/*-Memory Regions-*/
define symbol _region_ROM_start__ = 0x00000000;
define symbol _region_ROM_end__ = 0x0003FFFF;
define symbol _region_RAM_start__ = 0x20000000;
define symbol _region_RAM_end__ = 0x2000fFFF;
/* define addresses and size for SCIOPTA modules */
define exported symbol system_mod_start = _region_RAM_start__;
define exported symbol system_mod_size = 16K;
define exported symbol dev_mod_start = system_mod_start + system_mod_size;
define exported symbol dev_mod_size = 16K;
define exported symbol ips_mod_start = dev_mod_start + dev_mod_size;
define exported symbol ips_mod_size = 20K;
define exported symbol user_mod_start = ips_mod_start + ips_mod_size;
define exported symbol user_mod_size = 12K;
define symbol free_ram = user_mod_start + user_mod_size;
/*-Sizes-*/
define symbol _size_cstack__ = 0x0;
define symbol _size_heap__ = 0x4;
/* needed by resethook */
export symbol _region_RAM_end__;
/* to make linker happy, we define some symbols */
define symbol IRQ_STACK$$Limit = _region_RAM_end__;
export symbol IRQ_STACK$$Limit;
define symbol FIQ_STACK$$Limit = _region_RAM_end__;
export symbol FIQ_STACK$$Limit;
define symbol CSTACK$$Limit = _region_RAM_end__;
export symbol CSTACK$$Limit;
/* SCIOPTA modules */
define region SYSTEM_region = mem:[from system_mod_start size system_mod_size];
define region DEV_region = mem:[from dev_mod_start size dev_mod_size];
define region IPS_region = mem:[from ips_mod_start size ips_mod_size];
define region USER_region = mem:[from user_mod_start size user_mod_size];
/* define some block to keep certain segments together */
define block DEBUG with alignment = 4, maximum size = 512K { section .debug* };
define block KERNEL with alignment = 4, maximum size = 512K { section .text_krn* };
define block INTVEC with alignment = 4, maximum size = 0x100 { section .intvec };
define block NO_CACHE with alignment = 4, maximum size = 0x1000 { section .no_cache_sram };
/* define memory */
define region ROM_region = mem:[from _region_ROM_start__ to _region_ROM_end__];
define region RAM_region = mem:[from _region_RAM_start__ to _region_RAM_end__];

/* place data in system module */
```

```
define block system_mod { rw, section .textrw, zi };
define block user_mod { rw object hello.o };
define block dev_mod {};
define block ips_mod {};
initialize by copy { readwrite };
do not initialize { section .noinit, section .no_cache_sram };
/* place SCIOPTA modules */
place in SYSTEM_region { block system_mod } ;
place in DEV_region { block dev_mod };
place in IPS_region { block ips_mod };
place in USER_region { block user_mod };
/* be sure INTVECs are first ! */
place in ROM_region { block INTVEC};
place in ROM_region { readonly,block DEBUG ,block KERNEL };
place in RAM_region { block NO_CACHE };
```

IAR EW linker script examples can be found in the standard SCIOPTA board support package deliveries.

3.21.4.1. Files

<board>.icf	IAR EW linker script for the kernel getting started example (Hello). Location: <install_folder>\sciopta\<version>\bsp\arm\<cpu>\<board>\include\

3.21.4.2. Module Memory Map for IAR EW

Example of a system memory map containing the system module and one user module.

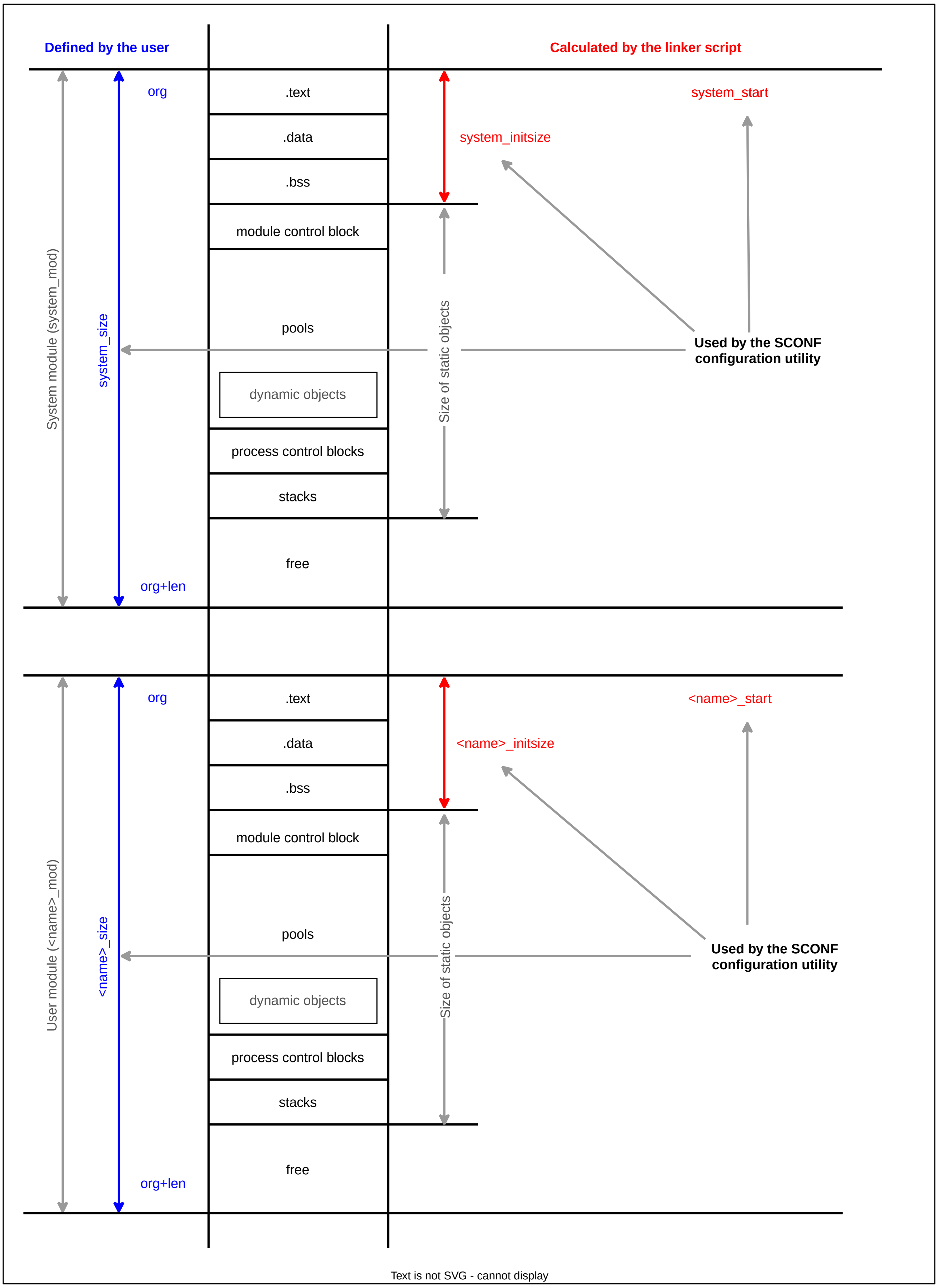


Figure 6. SCIOPTA Memory Map for GNU GCC

3.21.4.3. SCIOPTA Memory Map for GNU GCC

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.

Example of a module mapping file for a simple application:

```
/*
** This file describes the module mapping
*/
#include <sciopta.h>
DECL_MODULE(system);
DECL_MODULE(dev);
DECL_MODULE(ips);
DECL_MODULE(user);
```

Module mapping file examples can be found in the getting started projects included in the standard SCIOPTA delivery.

map.c	Module mapping for the kernel getting started example for IAR (Hello).
	Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello\<board>\

The **DECL_MODULE** macro is defined in the file:

os.h	Location:<install_folder>\sciopta\<version>\include\sciopta\arm\arch\

3.22. Trap Interface

In a typical monolithic SCIOPTA systems the kernel functions are directly called.
In more complex dynamic systems using load modules or if you want to access kernel system calls from user (unprivileged) modules in MPU protected modular systems the kernel functions cannot be accessed any more by direct calls.

In such systems SCIOPTA offers a trap interface. To include the trap interface you need to assemble the CPU dependent file **syscall.S**. A system call will then generate a software interrupt. See chapter Interrupt Vector Table for software interrupt exception vector.

Additionally you have to select “trap interface” in the kernel configuration (SCONF). See SCIOPTA Kernel Configuration SCONF Manual chapter General System Configuration TAB.

3.22.1. Files

syscall.S	ARM trampoline functions for system-calls for GCC. Location: <install_folder>\sciopta\<version>\include\machine\arm
syscall_iar.S	ARM trampoline functions for system-calls for IAR. Location: <install_folder>\sciopta\<version>\include\machine\arm\

The file **syscall.S** for GCC includes another file with the same name containing CPU independent trap interface functions.

syscall.S	CPU independent trampoline functions for system-calls for GCC. Location: <install_folder>\sciopta\<version>\include\machine\

3.23. Cortex-M and Cortex-R Memory Protection Unit

3.23.1. Introduction

To support a user (unprivileged) and supervisor (privileged) software model, a memory protection scheme is required to control the access rights. ARMv7-M (Cortex-M) and ARMv7-R (Cortex-R) support the Protected Memory System Architecture (PMSAv7). The system address space of a PMSAv7 compliant system is protected by a Memory Protection Unit (MPU).

Please consult chapter B3.5 “Protected Memory System Architecture” of the ARMv7 Architecture Reference Manual (document ARM DDI 0403C) for a detailed description of the ARMv7-M (Cortex-M) MPU.

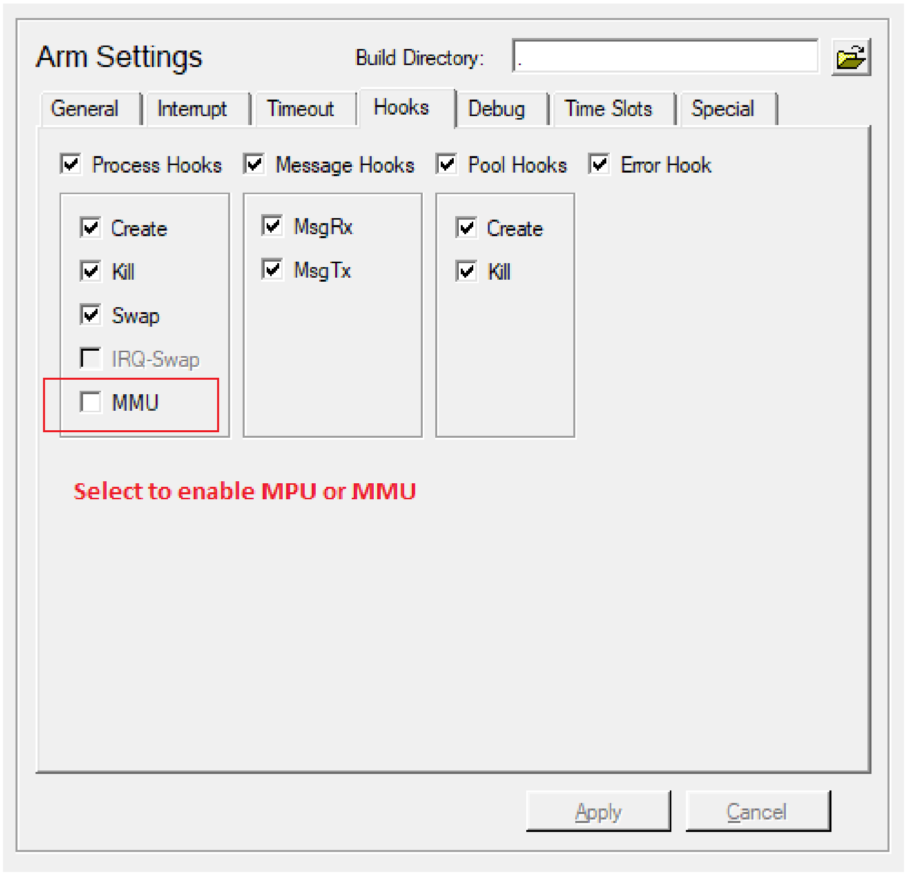
Please consult chapter 7 “Memory Protection Unit” of the ARM Cortex-R and Cortex-R4F Technical Reference Manual (document ARM DDI 0363G (ID041111)) for a detailed description of the ARMv7-R (Cortex-R) MPU registers.

The protected memory is divided up into a set of regions, with the number of regions supported is CPU dependent. While PMSAv7 supports region sizes as low as 32 bytes, finite register resources for the 4GB address space make the scheme inherently a coarse-grained protection scheme. The protection scheme is 100% predictive with all control information maintained in registers closely-coupled to the core.

In SCIOPTA memory protection is managed by the kernel and defined per SCIOPTA module. Please consult SCIOPTA Architecture Manual chapter Modules for more information about modules.

3.23.2. Enable the Memory Protection Unit

Most of the MPU code is included in the kernel. To enable this code you must select the MPU checkbox in the Hooks TAB of the system configuration of the SCONF kernel configuration. Please consult SCIOPTA Kernel Configuration SCONF Manual chapter Hooks Configuration TAB .



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

☑ Create
☑ Kill
☑ Swap
☐ IRQ-Swap
☐ MMU

☑ MsgRx
☑ MsgTx

☑ Create
☑ Kill

Figure 7. Hooks Configuration Window

3.23.3. Configuring the MPU

The user must supply an array named **sc_mpuTable[]** which contains the SCIOPTA modules. Each array element points to a module MPU array defining the regions and characteristics of a MPU segment.

These arrays are project specific and it is recommended to place it in a file called **mpu_table.c**.

Module mapping file examples can be found in the getting started projects included in the standard SCIOPTA delivery.

For Cortex-4R

3.23.3.1. Files

mpu_table.c	MPU configuration examples. Location: <install_folder>\sciopta\<version>\exp\krn\arm\hello_mmu\<board>\
core_cr4.S	Cortex-R4 core functions (including MPU functions) for GCC. Location: <install_folder>\sciopta\<version>\bsp\arm\src\gnu\
core_cr4.s	Cortex-R4 core functions (including MPU functions) for IAR. Location: <install_folder>\sciopta\<version>\bsp\arm\src\iar\
core_cr4.asm	Cortex-R4 core functions (including MPU functions) for TI CodeComposerStudio. Location: <install_folder>\sciopta\<version>\bsp\arm\src\ccs\

3.23.3.2. MPU Table Array

The MPU table array contains a list of all modules. Each array member point to another array containing MPU region definitions and settings for the respective module.

Example of a an MPU table array containing the system module and two user modules.

```
const uint32_t * const sc_mpuTable[] = {
  system_module,
  user_module1,
  user_module2
};
```

Note

The order of the modules in the MPU table array must be equal to the order of the modules defined in the kernel configuration (see chapter Creating Modules).

The kernel is not able to verify this!

3.23.3.3. Module MPU Array for Cortex-M

For each module listed in the MPU table array there must be an array defining the MPU regions.

Module MPU array format:

```
static const uint32_t <module_name>[] = {
  /* Region 0 */ MPU_RBAR0, MPU_RASR0,
  /* Region 1 */ MPU_RBAR1, MPU_RASR1,
  /* Region 2 */ MPU_RBAR2, MPU_RASR2,
  /* Region 3 */ MPU_RBAR3, MPU_RASR3,
  /* Region 4 */ MPU_RBAR4, MPU_RASR4,
  /* Region 5 */ MPU_RBAR5, MPU_RASR5,
  /* Region 6 */ MPU_RBAR6, MPU_RASR6,
  /* Region 7 */ MPU_RBAR7, MPU_RASR7
}
```

MPU_RBARn is the MPU Region Base Address Register and **MPU_RASRn** is the MPU Region Attribute and Size Register of the Cortex-M MPU registers.

The module base addresses are usually defined in the linker script.

Each array must contain 8*2 entries (for all 8 regions).

Please consult chapter B3.5 “Protected Memory System Architecture” of the ARMv7 Architecture Reference Manual for a detailed description of the ARMv7-M (Cortex-M) MPU registers.

Example of module MPU array for a system module and a user module:

```
static const uint32_t system_module[] = {
  /* Flash 128K */
  0x00000000|MPU_BASE_VALID,(MPU_ATTR_AP_RO_RO|
            MPU_ATTR_SIZE_128K|
            MPU_ATTR_ENABLE),
  /* Peripherals */
  1|MPU_BASE_VALID,0,
  /* SRAM */
  system_mod_start+(2|MPU_BASE_VALID),(MPU_ATTR_AP_RW_RW|
                     MPU_ATTR_SIZE_16K|
                     MPU_ATTR_ENABLE),
  /* SRAM bitband */
  0x22000000|3|MPU_BASE_VALID,(MPU_ATTR_AP_RW_RW|
                              MPU_ATTR_SIZE_2M|
                              MPU_ATTR_ENABLE),
  4|MPU_BASE_VALID,0,
  5|MPU_BASE_VALID,0,
  6|MPU_BASE_VALID,0,
  7|MPU_BASE_VALID,0
};


static const uint32_t user_module[] = {
  /* Flash 128K */
  0x00000000|MPU_BASE_VALID,(MPU_ATTR_AP_RO_RO|
                              MPU_ATTR_SIZE_128K|
                              MPU_ATTR_ENABLE),
  /* user module */
  user_mod_start+(1|MPU_BASE_VALID),(MPU_ATTR_AP_RW_RW|
                   MPU_ATTR_SIZE_16K|
                   MPU_ATTR_ENABLE),
  /* system module read only */
  system_mod_start+(2|MPU_BASE_VALID),(MPU_ATTR_AP_RW_RO|
                     MPU_ATTR_SIZE_16K|
                     MPU_ATTR_ENABLE),
  /* SRAM bitband */
  0x22000000|3|MPU_BASE_VALID,(MPU_ATTR_AP_RW_RW|
                              MPU_ATTR_SIZE_2M|
                              MPU_ATTR_ENABLE),
  4|MPU_BASE_VALID,0,
  5|MPU_BASE_VALID,0,
  6|MPU_BASE_VALID,0,
  7|MPU_BASE_VALID,0
};
```

3.23.3.4. Module MPU Array for Cortex-R

For each module listed in the MPU table array there must be an array defining the MPU regions.
Module MPU array format:

```
static const uint32_t <module_name>[] = {
  <region number>, <base address>, <access rights>, <size subregion and enable bit>,
  <region number>, <base address>, <access rights>, <size subregion and enable bit>,
  <region number>, <base address>, <access rights>, <size subregion and enable bit>,
  <region number>, <base address>, <access rights>, <size subregion and enable bit>
}
```

The module base addresses are usually defined in the linker script.

Each modul must contain the same number of regions. If a slot is not used, then an invalidate region must be added containing only region number and the other parameters all zeroes. The end of the array must be marked by a region number -1 with the other parameter all zeroes.

Please consult chapter 7 “Memory Protection Unit” of the ARM Cortex-R and Cortex-R4F Technical Reference Manual (document ARM DDI 0363G (ID041111)) for a detailed description of the ARMv7-R (Cortex-R) MPU registers.

Example of module MPU array for a system module and a user module:

```
static const int32_t system_module[] = {
  /* Peripherals (needed for boot-up and logd) */
  /* region */ 1,
  /* base */ 0xff000000,
  /* rights */ MPU_ATTR_AP_S_RW_U_RW|MPU_ATTR_STRONGLY_ORDERED|MPU_ATTR_XN,
  /* size */ MPU_SIZE_256M|MPU_REGION_ENABLE,

  /* SRAM for system module (overrides region 0 setting !) */
  /* region */ 2,
  /* base */ (int32_t)system_mod_start,
  /* rights */ MPU_ATTR_AP_S_RW_U_RW|MPU_ATTR_NORMAL_CACHE_WB|MPU_ATTR_XN,
  /* size */ MPU_SIZE_16K|MPU_REGION_ENABLE,

  /* invalidate regions used by other modules ! */
  /* region */ 3,
  /* base */ 0,
  /* rights */ 0,
  /* size */ 0,

  /* region */ 4,
  /* base */ 0,
  /* rights */ 0,
  /* size */ 0,

  /* EOF */
  -1,0,0,0
};

static const int32_t user_module[] = {
  /* SRAM: user_mod */
  /* region */ 1,
  /* base */ (int32_t)user_mod_start,
  /* rights */ MPU_ATTR_AP_S_RW_U_RW|MPU_ATTR_NORMAL_CACHE_WB|MPU_ATTR_XN,
  /* size */ MPU_SIZE_16K|MPU_REGION_ENABLE,

  /* invalidate regions used by other modules ! */
  /* region */ 2,
  /* base */ 0,
  /* rights */ 0,
  /* size */ 0,
  /* region */ 3,
  /* base */ 0,
  /* rights */ 0,
  /* size */ 0,

  /* region */ 4,
  /* base */ 0,
  /* rights */ 0,
  /* size */ 0,

  /* EOF */
  -1,0,0,0
};
```

3.23.3.5. Additional MPU Functions for Cortex-R

3.23.3.5.1. set_MPU_Region

Description

This function is used to initialize a Cortex-R MPU region. It is located in the files **core_cr4.*** and will usually be called from a project specific file **mpu_table.c** containing the MPU configuration.

Syntax

```
void set_MPU_Region(
  int region,
  int32_t base,
  int32_t size,
  int32_t attr
)
```

Parameters

region	Region number. One of the four region numbers of the Cortex-R. Value: 0..11
base	Base address of the region
size	Size of the region
attr	Region attributes

Return Value
None.

3.23.3.5.2. enable_MPU

Description
This function is used to enable the MPU of the Cortex-R CPU. It is located in the files **core_cr4.*** and will usually be called from a project specific file **mpu_table.c** containing the MPU configuration.

Syntax

```
void enable_MPU(void)
```

Parameters
None

Return Value
None.

3.23.3.5.3. disable_MPU

Description

This function is used to disable the MPU of the Cortex-R CPU. It is located in the files **core_cr4.*** and will usually be called from a project specific file **mpu_table.c** containing the MPU configuration.

Syntax

```
void disable_MPU(void)
```

Parameters
None

Return Value
None.

3.23.3.5.4. mpu_setup

Description
This is a application specific and user written function to setup the Cortex-R MPU. It is located in the file **mpu_table**.

This function should setup the mpu for startup. The settings will then be overwritten by the kernel on the first context switch with the module specific MPU settings.

Note: This is the place to setup global regions.

Syntax

```
void mpu_setup(void)
```

Parameters
None

Return Value
None.

Example
Typical example of a Cortex-R setup:

```
void mpu_setup()
{
  int i;
  const int32_t *p = system_module;
  /* invalidate all regions */
  for(i = 0; i < 12; ++i ){
    set_MPU_Region(i,0,0,0);
  }
  /* setup region for system module */
  while( (int)*p >= 0 ){
    set_MPU_Region(p[0],p[1],p[3],p[2]);
    p += 4;
  }
  /* setup "global" regions */
  /* Flash 2M, read only */
  set_MPU_Region(/* region */ 11,
                 /* base */ 0x00000000,
                 /* size */ MPU_SIZE_2M|MPU_REGION_ENABLE,
                 /* rights */ MPU_ATTR_AP_S_RO_U_RO|MPU_ATTR_NORMAL_CACHE_WB);
  /*
  ** Allow RO access to the system module for user processes.
  ** Supervisor mode needs write access, else the abort handler fails :(
  */
  set_MPU_Region( /* region */ 0,
                  /* base */ (int32_t)system_mod_start,
                  /* size */ MPU_SIZE_16K|MPU_REGION_ENABLE,
                  /* rights */ MPU_ATTR_AP_S_RW_U_RO|MPU_ATTR_NORMAL_CACHE_WB);
  enable_MPU();
}
```

3.23.4. Kernel MPU Management

The Cortex-M and Cortex-R MPU is under full control of the SCIOPTA kernel. The user just needs to define the MPU table array and the module MPU array(s) as described above.

At each process switch (swap) the kernel reprograms the MPU registers according to the values of the module MPU array, if the new process is within another SCIOPTA module.

Any access rights violation will generate an exception and will end up in the SCIOPTA error hook if such an error hook is defined (which is strongly recommended).

MPU protection works only for processes running in user mode.

4. Manual Versions

4.1. Manual Version 1.0

- Whole manual restructured and rewritten.

4.2. Manual Version 1.1

- Chapter folding
  - Initial chapters are folded.
  - Some clarifcations.
  - Layout fixes.