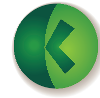


Open Kernel Labs™

Be open. Be safe.

SoC Developers Manual

Document Number: OK 40291:2008 (revision 1)

Software Version: 3.0

Date: September 12, 2008

Copyright © 2008 Open Kernel Labs, Inc.

This publication is distributed by Open Kernel Labs Pty Ltd, Australia.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT ANY WARRANTIES, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NON-INFIRNGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

This document may not be redistributed outside your organization without prior permission.

Authors:

Open Kernel Labs

Contact Details:

Open Kernel Labs Pty Ltd
Attention: Open Kernel Labs

Suite 3, 540 Botany Road
Alexandria, NSW 2015
Australia

email: **enquiries@ok-labs.com**
web: **http://www.ok-labs.com**/

Contents

Overview 5

1 Introduction to SoC SDK 7

2 Compiling the SoC Module 9

2.1 The SoC SDK Structure 9
2.1.1 The Directory Structure 9
2.1.2 The `include/soc` Directory 10
2.1.3 The `include/kernel` Directory 10
2.2 Constructing a Boot Image 11
2.2.1 Specifying the CPU 11
2.2.2 Specifying the Kernel 11

3 The SoC API 13

3.1 `soc/soc.h` 14
3.1.1 Versioning 14
3.1.2 System Start Up 14
3.1.3 Interrupt Configuration and Control 15
3.1.4 Timer 17
3.1.5 Cache 18
3.1.6 Debug Support 18
3.1.7 System Error 19
3.1.8 Platform Control 20
3.2 `soc/interface.h` 21
3.2.1 Versioning 21
3.2.2 Initialization 21
3.2.3 Thread Management 21
3.2.4 Memory Management 22
3.2.5 Memory Mapping 23
3.2.6 Interrupt Processing 24
3.2.7 Debugging 26

4 Debug Support 27

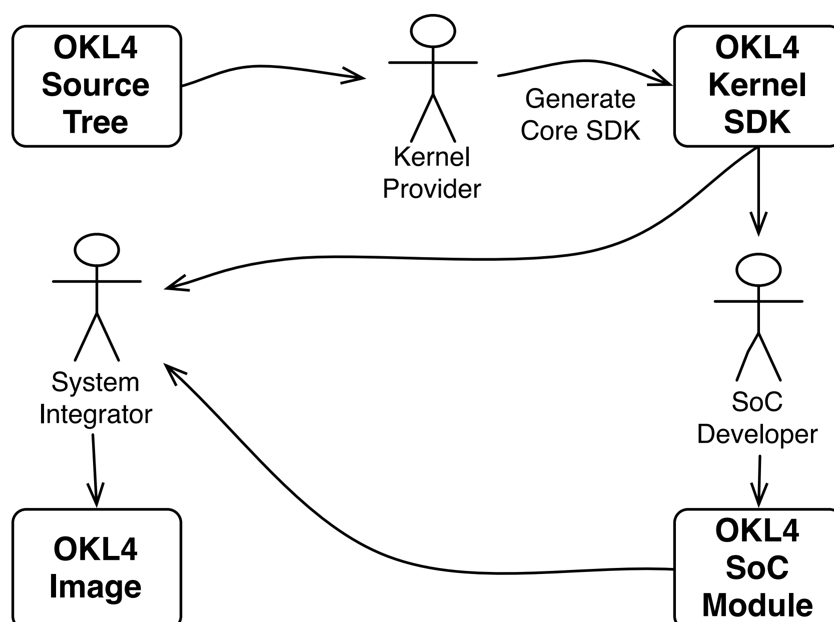
Overview

The *System on Chip Software Development Kit (SoC SDK)* serves two main purposes. Firstly, it enables a developer to easily port the OKL4 kernel to a new system-on-chip. Secondly, it allow a system integrator to easily build a system image consisting of an OKL4 kernel and a system on chip module provided by an SoC developer.

This manual provides an introduction to the SoC SDK followed by an overview of compiling the SoC module and an overview of SoC API and its main components. This is followed by a brief overview of the debugging support provided for the SoC SDK.

1 Introduction to SoC SDK

The *System on Chip Software Development Kit (SoC SDK)* serves two main purposes. Firstly, it enables a developer to easily port the OKL4 kernel to a new system-on-chip. Secondly, it allows a system integrator to easily build a system image consisting of an OKL4 kernel and a system on chip module provided by an SoC developer. The three main players in this scenario are the kernel developer (Open Kernel Labs), the SoC developer (a third party, separate from the kernel developer) and the system integrator (a third party, separate from the kernel and SoC developers). Thus the SoC SDK allows the *kernel* and *SoC module* to be distributed independently as shown in the diagram below.



A strong API allows kernel and SoC module developers to efficiently develop their modules independently of each other. A *Software Development Kit (SDK)* will be provided separately from the OKL4 source tree. The SDK contains the necessary components to build a new SoC module. The *Elfweaver* tool is then used to generate the final system image by combining the core kernel with the SoC module.

The kernel developer generates the the SDK from the raw kernel source tree. The resulting SDK contains binary object files of the kernel and any headers required by the SoC developer. The SDK can be distributed to both SoC developers and system integrators.

The SoC developer uses the headers in the SDK provided by the kernel developer to generate a SoC object file. The SoC developer has specific interfaces that must be provided in the SoC object file. Once completed the SoC developer distributes the SoC module to the system integrators.

The system integrator uses Elfweaver to integrate the SDK provided by the kernel developer with the SoC module provided by the SoC developer to build the final system image. The system integrator may also receive independent binaries from other sources for applications running on the final system.

For further information regarding Elfweaver, please refer to the *Elfweaver Reference Manual*.

2 Compiling the SoC Module

The SoC SDK is distributed as a tarball which is capable of building `soc.o`, the SoC image. This chapter provides a brief overview of the SoC SDK structure followed by a step by step guide for constructing a bootable image.

2.1 The SoC SDK Structure

The SoC SDK tarball contains the following files:

- Interface Header files.
- Pre-compiled versions of `kernel.o` files configured for the following conditions:
  - Production.
  - Benchmark.
  - Trace buffers.
  - Trace buffers and debugging structures.
  - Full debugging and KDB.
- Linker scripts and configuration header file for each version of `kernel.o`
- Supporting libraries for SoC code such as `libatomic_ops.a` and `libgcc.a`.
- An example Makefile to demonstrate building the SoC.

Each `kernel.o` is a partially linked kernel which contains all code required for the kernel with the exception of the SoC code and supporting libraries. The set of flags to be passed to the compiler and the *#defines* required to compile the kernel are also provided in the tarball.

2.1.1 The Directory Structure

An example directory structure for xscale is provided below.

```
sdk/
   kernel/
      xscale/
          micro-benchmark/
              include/
                  atomic_ops/
                  arch/
                  compat/
                  kernel/
                  soc/
```

Example directory structure continued.

```
sdk/
   kernel/
      xscale/
         micro-benchmark/
              include/
                   ...
              lib/
                  libatomic.a
              object/
                  xscale.o
                  defines
                  linker.lds
                  linker.sh
                  linker.bat
                  cxxflags
          micro-debug/
               ...
          micro-debug_no_console/
               ...
          micro-production/
               ...
          micro-tracebuffers/
               ...
          soc/
             pxa/
                    Makefile
                    machine.in.xml
                    src/
```

2.1.2 The `include/soc` Directory

The `include/soc` directory contains the following:

- The `soc_types.h`, `soc.h`, and `interface.h` headers including cpu-specific extension headers.
- Headers providing types common to the core kernel and SoC code, including:
  - Basic types such as `word_t`, `addr_t`.
  - Definition of a continuation function.
  - Standard macros such as `INLINE`.

2.1.3 The `include/kernel` Directory

The `include/kernel` directory contains the following:

- Headers providing common kernel utility routines including:
  - Bitmap management.
  - Address masking routines.
- Headers for system values including:

- Standard error values such as `EINVALID_PARAM`.
- System values as required on a per-CPU basis.

2.2 Constructing a Boot Image

It should be noted that Elfweaver will check that the `soc_api_version` symbol contained in the `soc.o` image matches the `kernel_api_version` symbol contained in the `kernel.o` image.

The `soc_api_version` and `kernel_api_version` symbols are further described in Sections 3.1.1 and 3.2.1, respectively.

2.2.1 Specifying the CPU

The generic kernel requires the type of the CPU on the target hardware. Elfweaver provides the *cpu* element, which is a sub-element of the *machine* element.

The `type` attribute of the `cpu` element is used to specify the CPU used by the target hardware. An error will be raised by Elfweaver if it fails to recognize the specified CPU.

Example: *Specifying the CPU type.*

```
<machine>
         ....
         <cpu type="xscale" />

</machine>
```

2.2.2 Specifying the Kernel

Elfweaver links the generic kernel to the SoC code as part of constructing the image. Elfweaver provides three ways of specifying the kernel components to support different development models. Each of these methods are described below.

2.2.2.1 Previously linked kernel

Where the kernel is already fully linked outside Elfweaver, the `file` attribute of the `kernel` element can be used to specify the file name of the fully linked kernel.

2.2.2.2 Link from an SDK directory

As Elfweaver is aware of the SDK directory structure, it can link from the kernel files within the tree. The `kernel` element has the attributes `sdk`, `configuration` and `platform` which are used specify the path to the top of the SoC directory, the kernel configuration to be constructed and the SoC module to be linked to the kernel, respectively. An example of the `kernel` element used in this format is provided below.

Example: *Linking from an SDK directory.*

```
<kernel sdk="/path/to/sdk_dir" configuration="micro-production"
        platform="imx31">
</kernel>
```

Elfweaver currently supports the following configurations:

- `micro-production`
- `micro-benchmark`
- `micro-tracebuffers`
- `micro-debug`
- `micro-debug-no-console`

2.2.2.3 Link from separate files

Where the required attributes are specified in directories different to that of the SDK, the individual parts required to link the kernel may be specified as attributes of the `kernel` element to provide greater control over the link process. The `kernel` element has the attributes `kernel`, `soc` and `linker` which are used to specify the path of the `kernel.o` file, the path of the `soc.o` file and the path of the linker wrapper script, respectively. When used in conjunction with the SDK linking attributes described in Section 2.2.2.2, these override the default paths calculated from the SDK. An example of the `kernel` element used in this format is provided below.

Example: *Linking from separate files.*

```
<kernel kernel="/path/to/kernel.o" soc="/path/to/soc.o"
        linker="/path/to/linker.sh">
</kernel>
```

3 The SoC API

The main functionality of the SoC API is provided by two header files, `soc/soc.h` and `soc/interface.h`. The `soc/soc.h` header file contains the functionality provided by the SoC implementation. The functionality provided by kernel to the SoC implementation is contained in the `soc/interface.h` header file. Note that CPU specific versions of these header files may also be provided depending on the CPU used in the supported hardware.

The `soc/soc.h` and `soc/interface.h` header files are further described in Sections 3.1 and 3.2, respectively. The SoC SDK provides varying levels of debug support. Debug support for the `soc/soc.h` and `soc/interface.h` headers are described in Sections 3.1.6 and 3.2.7, respectively. General debug support is described in Chapter 4.

3.1 soc/soc.h

The soc/soc.h header provides the functionality required by a SoC implementation. This functionality can be categorized as:

- Versioning
- System Start Up
- Interrupt configuration and control
- Timer
- Cache Operations
- Debug support
- System error
- Platform specific

Each of these categories is further described in Sections 3.1.1 to 3.1.8, below.

3.1.1 Versioning

The SoC code must provide the value of the symbol soc_api_version, which indicates the API version that the SoC code is built against. This should be set to the corresponding SOC_API_VERSION.

The value provided by soc_api_version symbol contained in the image soc.o will be checked against the value provided by kernel_api_version symbol contained in the image kernel.o by Elfweaver. If these values do not match, Elfweaver will exit with an error.

3.1.2 System Start Up

The SoC module provides the functions _start() and soc_init() for system start up. The initial power-up entry point for the kernel is provided by the function _start(). This function is called when the system is in the *power-up* state, that is the default state of the system when it is first turned on. This function must carry out the *"low-level"* initialization described below and end by calling the kernel_arch_init() function.

_start() must ensure that the platform is set to the following state:

- All caching is disabled.
- Virtual memory is disabled (MMU is turned off).
- Memory controllers are configured and running.
- The stack pointer is set.
- Other control registers required for further initialization are set.

It should be noted that _start() cannot use any interfaces provided by the core kernel with the exception of the kernel_arch_init() function. It should be noted that this function requires a CPU specific parameter.

The SoC module also provides the function soc_init(), responsible for carrying out all system-specific initialization once virtual memory is enabled. At a minimum this function should map IO areas into virtual memory, initialize timers and perform interrupt initialization specific to the SoC module.

The soc_init() function may use the following functions provided by the core kernel:

- kernel_add_mapping()
- kernel_mem_alloc()

- `kernel_mem_free()`
- `kernel_printf()`

Note that the `kernel_printf()` function can only be used if the a "debug" build is configured allowing tracing macros to be used. Compiling the SoC module is further described in Chapter 2.

Some CPUs contain a built-in interrupt controller whilst other CPUs have minimal core interrupt capabilities. Where the core CPU is capable of interrupt initialization and or configuration, it should occur after the `soc_init()` function has returned.

Certain parts of the hardware-level interrupt configuration occurring in SoC code and core kernel code is dependent on the CPU used and may require additional functionality to be implemented in the SoC code. Platform specific functionality is further described in Section 3.1.8.

3.1.3 Interrupt Configuration and Control

In this context, interrupt configuration refers to the process of setting up the threads to process interrupts once they are received. It does not refer to hardware configuration.

SoC module interrupt processing management involves the following steps:

- Receive the interrupt from the kernel.
- Decode the interrupt source and determine the handler thread.
- Invoke the kernel to perform interrupt delivery for each pending interrupt.
- Request the kernel to reschedule once all interrupts are processed.
- Clear the interrupt in the hardware after it is processed.

The SoC provides two functions to configure interrupts. The `soc_security_control_interrupt()` function described in Section 3.1.3.1 allows the core kernel to specify the the space associated with the interrupt and the `soc_config_interrupt()` function described in Section 3.1.3.2 allows the core kernel to map interrupts to specific handler threads. Note that the handler thread for a particular interrupt must reside in the space associated with the interrupt descriptor of that interrupt. The SoC also provides the function `soc_ack_interrupt()`, described in Section 3.1.3.3, which is used to acknowledge the processing of an interrupt.

3.1.3.1 `soc_security_control_interrupt()`

```
word_t soc_security_control_interrupt(struct irq_desc *desc,
                                      space_h         owner,
                                      word_t          control)
```

The `soc_security_control_interrupt()` function is used specify the address space considered to be the owner of a particular interrupt.

This function requires the following arguments:

desc	A pointer to the interrupt descriptor to be configured.
owner	Address space to be associated with the interrupt descriptor specified by the *desc* argument.

control The operation to be carried out, that is whether the space specified by the *owner* argument is to be associated with (indicated by a value of $(0 \ll 16)$) or disassociated (indicated by a value of $(1 \ll 16)$) from the interrupt descriptor specified by the *desc* argument.

Note that a particular interrupt descriptor can only be associated with a single address space at a given time. However, a particular interrupt descriptor that is currently associated with a particular address space may be disassociated from it and subsequently associated with a different one.

This function returns `0` on successful completion and has the following error codes:

EINVALID_PARAM The value specified by *desc* was invalid.

EINVALID_PARAM Attempting to disassociate when the interrupt descriptor is not associated with a particular address space.

EINVALID_PARAM Attempting to associate when the interrupt descriptor is currently associated with a particular address space.

3.1.3.2 soc_config_interrupt()

```
word_t soc_config_interrupt(struct irq_desc *desc,
                            int              notify_bit,
                            tcb_h            handler,
                            space_h          owner,
                            soc_irq_action_e action,
                            struct utcb_t   *utcb)
```

The `soc_config_interrupt()` function allows the core kernel to map interrupts to specific handler threads residing in the space owning the interrupt.

This function requires the following arguments:

desc A pointer to the interrupt descriptor.

notify_bit The *asynchronous notification* bit to be used when notifying the handler thread of the interrupt. This value indicates the bit number to be set in the *notification mask*.

handler The handler thread corresponding to the interrupt described by the interrupt descriptor specified by the *desc* argument.

owner The address space that owns the interrupt described by the interrupt descriptor specified by the *desc* argument. Note that the handler thread specified by the *handler* argument must reside in this address space. This argument is used for validation.

action Describes the operation to be carried out, register or de-register.

**utcb* User data associated with the handling thread. This parameter is provided to allow access to the address space for SoC reserved words. Note that the *utcb* value should not be stored for later use as the utcb information associated with the handler thread may change, for example as a result of a SpaceSwitch operation.

Note that only a single handler can be registered to a particular interrupt at any given time. Attempting to register an interrupt that is already successfully registered to a particular handler to a different handler will result in an error. Once de-registered, an interrupt may be subsequently registered to a different handler.

This function returns `0` on successful completion and has the following error codes:

ENINVALID_PARAM The owning address space does not match the owner specified by the *owner* argument.

EINVALID_PARAM There is no registered owner for the specified interrupt.

EINVALID_PARAM	The *desc* argument is invalid.
EINVALID_PARAM	The handler is not the registered handler for a de-register operation.

3.1.3.3 soc_ack_interrupt()

```
word_t soc_ack_interrupt(struct irq_desc *desc,
                         tcb_h            handler)
```

The `soc_ack_interrupt()` function allows the core kernel to indicate that a particular interrupt has been processed.

This function requires the following arguments:

desc	A pointer to the interrupt descriptor.
handler	The thread wishing to acknowledge the interrupt.

This function returns `0` on successful completion and has the following error code:

EINVALID_PARAM	The handler thread specified by the *handler* argument does not match the registered handler thread of the interrupt.

3.1.4 Timer

The SoC module may provide the kernel timer interrupt depending on the CPU. If this is the case, the CPU specific part of the API is extended with the timer control interface to allow the core kernel to enable and disable the time.

As the platform code may re-enable the timer in response to interrupts independently of the core kernel, this state must be stored by the platform code. The platform code should therefore provide a boolean value, `soc_timer_disabled`, which indicates whether timer ticks are currently enabled or disabled. The core kernel monitors this value and may disable the timer again as required.

3.1.4.1 soc_enable_timer()

```
void soc_enable_timer(void)
```

The `soc_enable_timer()` function is used to enable the timer.

3.1.4.2 soc_disable_timer()

```
void soc_disable_timer(void)
```

The `soc_disable_timer()` function is used to mask out a timer interrupt. It should be noted that this operation may be overridden by the presence of interrupt activity.

3.1.4.3 soc_timer_tick_length()

```
word_t soc_get_timer_tick_length(void)
```

The `soc_timer_tick_length()` function is used to obtain the platform specific timer tick length in microseconds. Note that this function is only available if debug is enabled.

3.1.5 Cache

The SoC module is required to implement cache functionality for any caches that are not CPU specific. If such cases do not exist, these functions do not have to be implemented.

3.1.5.1 soc_cache_range_op_by_pa()

```
void soc_cache_range_op_by_pa(addr_t pa,
                  word_t sizelog2, word_t attrib)
```

The `soc_cache_range_op_by_pa()` function is used to perform an outer cache range operation. This is normally called from the cache control system call.

This function requires the following arguments:

pa	Physical address contained by the cache line.
sizelog2	Log base 2 of the size of the cache range.
attrib	Instruction or data side of the cache.

3.1.5.2 soc_cache_full_op()

```
void soc_cache_full_op(word_t attrib)
```

The `soc_cache_full_op()` function is used to perform an outer cache full operation. This is normally called from the cache control system call. This function requires the argument, *attrib*, which is used to specify the Instruction or Data side of the cache.

3.1.5.3 soc_cache_drain_write_buffer()

```
void soc_cache_drain_write_buffer(void)
```

The `soc_cache_drain_write_buffer()` function is used to drain all write buffers on the outer cache or outer memory barrier. This is normally called from the cache control system call.

3.1.6 Debug Support

Debug support is conditionally compiled as described in Chapter 4.

The SoC code should implement the following functions under the following conditions:

soc_kdb_init()	Implemented if `CONFIG_KDB` is defined. This function should carry out all initialization relating to KDB. The extent of the work to be performed by this function is dependent on the KDB values specified at compile time.
soc_console_getc()	Implemented if `CONFIG_KDB` and `CONFIG_KDB_CONS` are defined. This function should return a character from the active console screen. This function requires a single argument *can_block* which is used specify whether the function may block whilst waiting for the character to arrive. *can_block* should be set to `true` if blocking is permitted. This function returns `-1` if no character is available.

soc_console_putc() — Implemented if `CONFIG_KDB` and `CONFIG_KDB_CONS` are defined. This function should output a character on the active console screen. This function requires a single argument, *c*, which is used to specify the character to output. Note that this function may block whilst waiting for the device to accept the character.

soc_reboot() — Implemented if `CONFIG_KDB` and `CONFIG_KDB_CONS` are defined. This function should reboot the platform. This function must not return under any circumstance.

soc_kdb_dump_whole_l2cache() — Implemented if `CONFIG_KDB` and `CONFIG_HAS_SOC_CACHE` are defined. This function should dump all cache lines in the Level 2 cache.

soc_kdb_dump_l2cache() — Implemented if `CONFIG_KDB` and `CONFIG_HAS_SOC_CACHE` are defined. This function should dump all ways that share the index specified by the *index* argument.

soc_kdb_read_event_monitor() — Implemented if `CONFIG_KDB` and `CONFIG_HAS_L2_EVTMON` are defined. This function should dump Level 2 event monitor counters.

soc_kdb_set_event_monitor() — Implemented if `CONFIG_KDB` and `CONFIG_HAS_L2_EVTMON` are defined. This function should set the Level 2 event monitor counter and start to count an event. This function requires the arguments, *counter*, which specifies the counter to use and *event*, which specifies the Level 2 event to be counted.

3.1.7 System Error

The SoC code provides the `soc_reboot()` and `soc_panic()` non-returning functions for dealing with unrecoverable errors.

3.1.7.1 `soc_reboot()`

```
void NORETURN soc_reboot(void)
```

The SoC code provides `soc_reboot()` function which may be used to reboot the platform. Note that this function does not return.

3.1.7.2 `soc_panic()`

```
void NORETURN soc_panic(void)
```

The SoC code provides the function, `soc_panic()`, to handle the case where an unrecoverable error has occurred. The most appropriate action to be taken in this instance is left to the discretion of the SoC developer. Note that this function does not return.

3.1.8 Platform Control

The SoC code provides the function `soc_do_platform_control`. The available operations and data are left to the discretion of the SoC developer.

```
word_t soc_do_platform_control(tcb_h           current,
                                plat_control_t  control,
                                word_t          param1,
                                word_t          param2,
                                word_t          param3,
                                continuation_t  continuation)
```

This function requires the following arguments:

current	The thread making the request.
control	The operation to be performed.
param1	SoC defined.
param2	SoC defined.
param3	SoC defined.
continuation	The continuation function to be used for return. If the SoC code decides that a reschedule or sleep is required, this continuation should be used in the call to `kernel_schedule()`.

This function should return a non-zero value on success and zero on failure. The error values should be set in the utcb of the thread specified by the *current* argument.

3.2 soc/interface.h

The soc/interface.h header file contains the API provided by the core kernel to the SoC module. This functionality can be categorized as:

- Versioning
- Initialization
- Thread Management
- Memory Management
- Memory Mapping
- Interrupt Processing
- Debugging

Each of these categories is further described in Sections 3.2.1 to 3.2.7, below.

3.2.1 Versioning

The kernel.o image provides the symbol kernel_api_version, which indicates the API version that it was build against. This value should be set to the corresponding SOC_API_VERSION.

The value provided by kernel_api_version symbol contained in the image kernel.o will be checked against the value provided by soc_api_version symbol contained in the image soc.o by Elfweaver. If these values do not match, Elfweaver will exit with an error.

3.2.2 Initialization

As described in Section 3.1.2, the SoC implementation must call the kernel_arch_init() function after the low-level system set-up is completed. The kernel then initializes the CPU and architecture specific features including the MMU and interrupt controllers as well as the core kernel features such as kernel heaps, address spaces, threads and debugging.

3.2.3 Thread Management

The SoC implementation is required to maintain thread references for purposes such as interrupt management. The core kernel uses thread references to provide revocable references for TCB objects. As such, a thread reference may be invalidated after it was stored by the SoC module, for example, in the event a thread is deleted after being stored by the SoC module.

The SoC module should not have any knowledge of the internals of threads. As such the SoC API provides the type tcb_h which is a handle to a thread and a SoC reference object, soc_ref_t, which is a wrapper of the tcb_h object. A thread handle can be stored within an SoC reference object for later use. In the event that a thread handle is no longer valid, for example if the corresponding thread has been deleted, the SoC will return NULL when an attempt is made to extract the thread handle from the SoC reference wrapper.

The SoC API provides the following thread management functions:

- kernel_ref_get_tcb()
- kernel_ref_init()
- kernel_ref_set_referenced()
- kernel_ref_remove_referenced()

A brief overview of each of these functions is provided below.

3.2.3.1 kernel_ref_get_tcb()

```
tcb_h kernel_ref_get_tcb(soc_ref_t ref)
```

The kernel_ref_get_tcb() function is used provide access to the thread handle stored in the SoC reference object. This function requires a single argument, ref, which is used to specify a SoC reference. This function will return a thread handle or NULL. If the thread handle stored in the SoC reference specified by ref refers to an existing thread, the thread handle is returned. This function returns NULL if the specified SoC reference does not contain a thread handle or if the thread handle is no longer valid as the corresponding thread no longer exists.

3.2.3.2 kernel_ref_init()

```
void kernel_ref_init(soc_ref_t *ref)
```

The kernel_ref_init() function is used to initialize the SoC reference object. This function requires a single argument, ref, used to specify the SoC reference object to be initialized. On successful completion, the specified SoC reference will not contain a thread handle.

3.2.3.3 kernel_ref_set_referenced()

```
void kernel_ref_set_referenced(tcb_h      obj,
                               soc_ref_t *ref)
```

The kernel_ref_set_referenced() function is used to store a reference to a particular thread. This function requires the arguments obj and ref which are used to specify the thread handle to be stored and the SoC reference in which the thread handle is stored, respectively. Once stored, the thread handle can be retrieved using the kernel_ref_get_tcb() function described above.

3.2.3.4 kernel_ref_remove_referenced()

```
void kernel_ref_remove_referenced(tcb_h      obj,
                                  soc_ref_t *ref)
```

The kernel_ref_remove_referenced() function may used to remove the thread handle stored in the SoC reference object. On completion, the SoC reference will no longer contain a reference to the thread handle.

3.2.4 Memory Management

The kernel provides the following functions for accessing kernel heap memory:

- kernel_mem_alloc()
- kernel_mem_free()

Each of these functions is briefly described below. Note that these functions will use the first kernel heap set up by Elfweaver. Elfweaver will place the largest heap first.

3.2.4.1 kernel_mem_alloc()

```
void * kernel_mem_alloc(word_t size, int zero)
```

The kernel_mem_alloc() function is used too allocate a block of memory from the kernel heap.

This function requires the following arguments:

size	The size of the block of memory to be allocated in bytes. Note that the minimum size is 1MB.
zero	Specifies whether the allocated memory is zeroed. *zero* should be set to true if the allocated memory is required to be zeroed.

3.2.4.2 kernel_mem_free()

```
void kernel_mem_free(void * address, word_t size)
```

address	A pointer to the start of the memory block to be freed.
size	Size of the memory block to be freed in bytes. Note that a memory block must be freed in its entirety, it cannot be partially freed.

3.2.5 Memory Mapping

As described in Section 3.1.2, the SoC code is required to map hardware-specific memory regions. The kernel_add_mapping() function described below is provided for this purpose. This function returns the virtual address of the mapping.

It should be noted that SoC code should not indicate the location of the mappings within the virtual address space. This is controlled by the core kernel.

3.2.5.1 kernel_add_mapping()

```
addr_t kernel_add_mapping(word_t size,
                          addr_t phys,
                          word_t cache_attr)
```

The kernel_add_mapping() function may be used to add a new mapping to the kernel.

This function requires the following arguments:

size	The size of the memory region in bytes.
phys	The physical starting address of the region.
cache_attr	The caching attributes of the mapping.

The cache attribute encoding specified by the cache_attr argument is dependent on the CPU for which the kernel is compiled. This encoding is further described in the *OKL4 CPU Reference Manual*. The *cache_attr* argument may be set to SOC_CACHE_DEVICE to assign normal caching attributes to the SoC module requested mapping. As only this default value is provided, all other attributes supported by the CPU must be explicitly provided by the SoC module.

It should be noted that the size specified by the *size* argument is rounded up to the nearest page size. This function returns the virtual address of the newly created mapping on successful completion and NULL on failure.

3.2.5.2 `kernel_add_mapping_va()`

```
bool kernel_add_mapping_va(addr_t virt,
                           word_t size,
                           addr_t phys,
                           word_t cache_attr)
```

This function is used to add a new mapping at a specified virtual address to the kernel. ***It should be noted that this function has been deprecated and will be removed in future releases***. As the SoC code is unaware of the layout of virtual memory, choosing an appropriate virtual address may involve some trial and error.

This function requires the following attributes:

virt	The virtual address at which to map the memory.
size	The size of the memory region to be mapped in bytes.
phys	The physical address of the memory region to be mapped.
cache_attr	The caching attributes of the mapping.

The cache attribute encoding specified by the `cache_attr` argument is dependent on the CPU for which the kernel is compiled. This encoding is further described in the *OKL4 CPU Reference Manual*. The *cache_attr* argument may be set to `SOC_CACHE_DEVICE` to assign normal caching attributes to the SoC module requested mapping. As only this default value is provided, all other attributes supported by the CPU must be explicitly provided by the SoC module.

It should be noted that the size specified by the *size* argument is rounded up to the nearest page size. This function returns a non-zero value on success and `0` on failure.

3.2.6 Interrupt Processing

All SoC implementations must provide a function `soc_handle_interrupt()` which processes interrupts at hardware level. The number of parameters and their meanings will vary with the architecture. Therefore, this function will be specified at the CPU level.

The SoC code receives interrupts via the `soc_handle_interrupt()` function and will normally call the `kernel_deliver_notify()` function described in Section 3.2.6.1, to deliver the interrupt to the registered handler thread.

The SoC API allows multiple pending interrupts to be delivered to different handlers from a single kernel trap. In such cases, the SoC code must set the *cont* argument of the `kernel_deliver_notify()` function to NULL. This function will then return without performing a reschedule operation. The SoC code may call the `kernel_deliver_notify()` function multiple times followed by performing a manual reschedule by calling the `kernel_schedule()` function described in Section 3.2.6.2.

The SoC API provides the following interrupt processing functions:

- `kernel_deliver_notify()`
- `kernel_schedule()`
- `kernel_scheduler_handle_interrupt()`

A brief overview of each of these function is provided below.

3.2.6.1 `kernel_deliver_notify()`

```
bool kernel_deliver_notify(tcb_h handler,
                            word_t notifybits,
                           continuation_t cont)
```

The `kernel_deliver_notify()` function is used to deliver an interrupt notification to a specified handler thread.

This function requires the following arguments:

handler The handler thread to which the interrupt notification is to be delivered.

notifybits The set of notification bits to be delivered to the handler.

cont The continuation function to be activated once the interrupt is processed. If this argument is set to NULL, this function will return and no reschedule operation will be performed. In the event *cont* is not set NULL, the kernel may either perform a reschedule operation or activate the continuation function directly.

This function returns `true` if the handler thread was woken up, the SoC code must then manually perform a reschedule using the `kernel_schedule()` function described below instead of activating the continuation function of the current thread. If the handler thread was not woken up, a reschedule is not required and this function should return `false`.

3.2.6.2 `kernel_schedule()`

```
void kernel_schedule(continuation_t cont)
```

The `kernel_schedule()` function is used to invoke the scheduler to determine the next thread to run. This function requires a single argument, *cont*, which is used to specify the continuation function to be stored for use when the thread is reactivated. This argument cannot be set to NULL. It should be noted that this function will not return.

3.2.6.3 `kernel_scheduler_handle_interrupt()`

```
void kernel_scheduler_handle_timer_interrupt(bool wakeup,
                                             word_t interval,
                                             continuation_t cont)
```

When the SoC module provides a timer interrupt for the kernel, the SoC code must decode this interrupt and call the `kernel_scheduler_handle_interrupt()` function to inform the scheduler to handle a timer interrupt.

This function requires the following arguments:

wakeup This argument should be set to `true` if there is a pending thread to be activated, for example if `kernel_deliver_notify()` was called prior to this function.

interval The time in microseconds since the last timer interrupt or since the last reactivation of the timer, whichever is smaller.

cont The continuation function to activate on function completion. Note that this argument may not be set to NULL.

It should be noted that this function will not return.

3.2.7 Debugging

The core kernel provides the `kernel_fatal_error()` function for cases in which an unrecoverable error occurs in the SoC code. Depending on the kernel this may result in the kernel jumping to KDB or a kernel panic.

```
void kernel_fatal_error(char *msg)
```

The `kernel_fatal_error()` function requires a single argument, *msg*, which is used to specify the error message to be printed if debugging support is enabled. This function will enter KDB if the kernel debugger is enabled and call `soc_panic()` described in Section 3.1.7 if it is not. It should be noted that this function will not return.

4 Debug Support

The SoC SDK provides several kernel images, each with a varying level of debug support compiled in. The SoC API uses a subset of the same conditional compilation options as that offered by the core kernel.

To enable basic debug support, the *CONFIG_KDB* macro must be defined.

Other macros that provide debug support include:

CONFIG_KDB_CONS Enables KDB console support

CONFIG_KDB_CLI Enables KDB interactive command line interface

CONFIG_KDB_NO_ASSERTS Forces assertions to be ignored by the compiler. Where this macro is not specified, assertions will result in a kernel panic.

CONFIG_KDB_TIMER Allows the timer function, `soc_timer_tick_length()` described in Section 3.1.4.3 to be used.

Copyright © 2008 Open Kernel Labs, Inc.