SmartDSP Operating System

User Guide

Revised: November 23, 2015

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, TX 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

The SmartDSP Operating System (OS) is a Real Time Operating System (RTOS) that runs on the following StarCore DSP-based processors:

- B4860
- PSC9131
- PSC9132
- MSC815x/MSC825x
- MSC814x
- MSC812x
- MSC8101

The StarCore CodeWarrior (CW) Development Studio includes SmartDSP OS royalty-free source code. Further, SmartDSP OS has a high-level Application Programming Interface (API) that lets users develop integrated applications for StarCore processors.

This chapter provides information about SmartDSP OS—its features, architecture, startup processes and more.

1.1 Features

In SmartDSP OS, most functions are written in ANSI C. However, when needed, Assembly optimizes time-critical functions by maximizing StarCore's multiple execution units.

Other SmartDSP OS features include the following:

- small memory footprint ideal for high-speed StarCore processors
- priority-based event-driven scheduling (triggered by user applications or HW)

- dual-stack pointer for exception and task handling
- inter-task and inter-core communication using queues, semaphores, and events.

1.2 Architecture

SmartDSP OS design is Cooperative Asymmetric Multi-Processing (CAMP)-based:

- cores run their own OS instances; and,
- OS manages shared resources and supports inter-core communication.

SmartDSP OS also provides an SMP scheduler for MSC814x and MSC815x architectures.

The following sections explain different components of SmartDSP OS architecture:

- 1.2.1 Platforms
- 1.2.2 Kernel
- 1.2.3 Drivers
- 1.2.4 Utilities

1.2.1 Platforms

Table 1.1 lists SmartDSP OS-supported platforms.

Table 1.1 SmartDSP OS-supported Platforms

Platform	Specific Targets
B4860	• B4860
PSC9131/PSC9132	• PSC9x31, PSC9x32
MSC815x/MSC825x	• MSC8158/MSC8157 • MSC8156/MSC8154/MSC8152/ MSC8151 • MSC8256/MSC8524/MSC8252/ MSC8251
MSC814x	• MSC8144 • MSC8144E • MSC8144EC
MSC812x	• MSC8122 • MSC8126 • MSC8102

Table 1.1 SmartDSP OS-supported Platforms

Platform	Specific Targets
MSC810x	• MSC8101 • MSC8103

1.2.2 Kernel

The SmartDSP OS kernel is well-suited for multi-core processors and applications. The kernel has these features:

- supports HW and SW interrupts
- supports spinlocks to protect critical multicore data
- allows user-defined interrupt handlers
- Memory Management Unit (MMU) supports HW memory protection

SmartDSP OS kernel components:

- Caches
- Hardware Abstraction Layer (HAL)
- Inter-core messaging
- Interrupts (HW, SW, and virtual)
- MMU
- Memory Manager
- Message and event queues
- Scheduler
- Semaphores
- Spinlocks
- SW and HW timers
- Tasks

1.2.3 Drivers

The SmartDSP OS provides a unified cross-device API[1] for the support of generic HW drivers. Further, it supports drivers providing HW support for relevant Systems-on-a- Chip (SoC):

[1] See the *SmartDSP OS API Reference Manual* for more information.

- Direct Memory Access (DMA)
- On-chip Network DMA (OCeaN DMA)
- Ethernet
  - TSEC
  - Quick Engine (UEC)
- Serial Rapid Input/Output (sRIO)
  - RIONET
  - Doorbells
  - eMSG
- PCI Express (PEX)
- MAPLE
- Time Division Multiplexing (TDM)
- I^2C or Generic I^2C
- SPI or Generic SPI
- Universal Asynchronous Receiver/Transmitter (UART)
- CPRI
- AIC
- SEC
- HW_Timers

NOTE An SoC is an integrated circuit (IC) containing a core processor, special function processors, multiple memories and buses, standards-based peripherals, custom logic, and mixed-signal components. Together, these features communicate over a dedicated communication infrastructure.

1.2.4 Utilities

The SmartDSP OS provides utilities used to develop applications for StarCore processors:

Table 1.2 Utilities

Utility	Function
Configuration tool	Configures SmartDSP OS (os_config.h).
UDP/IP packet generator	Create UDP/IP random packets (with customizable parameters).

Table 1.2 Utilities

Utility	Function
SRTP client/server	
MJPEG player	Supports motion JPEG modules.
RTP player	Support RTP modules.
SmartDSP Host Exchange over Asynchronous Transfer (HEAT) server	Supports Ethernet remote file access.
CommExpert	Generates initialization code for supported DSPs.

1.3 Directory Structure

The SmartDSP OS is part of the StarCore CodeWarrior Development Studio and is found in the directory, `\StarCore_Support\SmartDSP`.

Table 1.3 lists SmartDSP subdirectories.

Table 1.3 SmartDSP Subdirectories

Subdirectory	Content
`demos`	Sample projects for supported platforms.
`drivers`	C source and header files for compiling a driver library.
`include`	C header files used by SmartDSP OS components.
`initialization`	C source and header files used for OS initialization.
`lib`	Compiled libraries used by SmartDSP OS components.
`source`	C source and header files for building compiled libraries.
`tools`	Utilities provided by SmartDSP OS,
`doc`	OS documentation files and device-specific documentation.

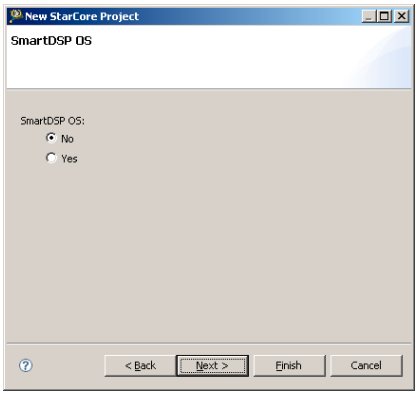
1.4 Initialization and Start Up

The SmartDSP OS uses a specified sequence of function calls during initialization.

Table 1.4

Initialization Stage	Description
Environment Initialization	1. CW startup file begins execution. 2. Calls SmartDSP OS hooks, global variables, and MMU. 3. Jumps to application's main() function. • Based on the SmartDSP OS option, select from the Build Settings page (see Figure 1.1). • SmartDSP OS linker files are included in a project. See 1.5.1 Initialization for more information.
SmartDSP OS Initialization	Main() function calls osInitialize() function.
Application Initialization	• Main() function initializes user application. • Sample applications, under the demo directory, use appInit().
Application Startup	• Main() function starts user application. • Creates background task in osStart() function. • Switches to pending task with the highest priority.

Figure 1.1 Build Settings Page



1.5 SmartDSP OS Linker

This section details 1.5.1 Initialization and 1.5.2 Guidelines.

1.5.1 Initialization

The **Build Settings** stage (Figure 1.1) of environment initialization offers Yes and No options.

- **No:** Activates StarCore sc3000-ld linker[1].
- **Yes:** Below noted linker files will be included in the application.

Table 1.5

File Type	Linker Files	Description
Application-dependant	`memory_map_link.l3k`	Splits memory into local/shared regions.
	`local_map_link.l3k`	Maps local sections to the memory.
	`os_msc815x_link.l3k`	Primary application-dependent file.
SmartDSP OS MSC815x-specific	`os_msc815x_link_mmu_defines.l3k`	SC3850 MMU attribute definitions
	`os_msc815x_link_defines.l3k`	MSC815x internal memory map
	`os_msc815x_link_app.l3k`	SmartDSP OS segment attribute definitions
	`os_msc8156_linker_ads_defines.l3k`	ADS (DDR) memory map

NOTE See the following guides for information on linker error messages and troubleshooting:
– *StarCore SC3000 Linker User Guide* for sc3000-ld linker (.l3k)
– *StarCore SC100 Linker User Guide* for sc100-ld linker (.lcf)

[1] For more information see *StarCore SC3000 Linker User Guide*.

1.5.2 Guidelines

This section lists guidelines for using StarCore Linker:

1. Consider the following if modifying the physical location of local and shared data.
   a. Consider performance issues; e.g., `local_non_cacheable` is currently in M2 memory.
   b. Check that program sections—used during initialization (before and during MMU programming)—maintain 1x1 virtual-to-physical mapping.
   c. Ensure that the segments `.att_mmu, .oskernel_local_data`, and `.oskernel_local_data_bss` are all local per core and confined by symbols `_VirtLocalData_b`, `_LocalData_size`, and `_LocalData_b` (`memory_map_link.l3k`). Symbols initialize the MMU and the stack thus enabling C code to run properly.
2. Consider the following if there is insufficient space in a specific memory location.
   a. Check if all allocated space is truly required.
   b. Check if there is a memory space or MMU alignment issue.

      Re-align MMU by modifying the linker command file such that larger sections/segments link first; this provides better aligned base addresses.
   c. Check if memory is truly lacking. If so, move non-critical portions to a distant memory.
3. SmartDSP OS uses on-chip memory by default.
   - DDR is not essential on a board.
   - If necessary, a portion of the OS can be moved to the DDR.
4. Changes to the linker command file:
   - If the debugger is still unable to reach the `main()` function, then there may be an issue of insufficient space at a specific memory location.
5. Memory allocated from a SmartDSP OS-managed heap:
   - May provide better linking flexibility.
6. Global variables:
   - If not set in a specific section then they will be placed in `.data` or `.bss`.
7. Asymmetric linking (supporting device):
   - Allows symbols to be linked to sections yet unmapped by all cores.
   - Allows cores to link symbols to different physical memories and/or implementations

1.6 Accompanying Documentation

The Documentation page describes the documentation included in this version of CodeWarrior Development Studio for StarCore DSP Architectures. You access the Documentation page by:

- a shortcut link on the Desktop that the CodeWarrior installer creates by default, or
- opening START_HERE.html in `CWInstallDir`\SC\Help.

1.7 More About SmartDSP OS

SmartDSP is the chosen OS for Freescale's highly optimized SW libraries—libraries supporting baseband, video, voice and other DSP applications.

Freescale and its SW partners—by offering a wide variety of SW libraries optimized on FSL's DSP platforms—enable efficient product development. The SW libraries support both up-to-date and legacy voice and video codecs.

Included in the SW libraries are optimized baseband kernel libraries. SmartDSP OS is based on optimized kernels; their unique FSL DSP capabilities enable users to develop highly effective SW applications.

All DSP libraries are implemented on SmartDSP OS—to make use of its rich feature set while maintaining expected levels of efficiency and performance.

For more information on Freescale DSP SW libraries, visit www.freescale.com\dsp.

2

Kernel Components

An OS kernel is an interface between user applications and HW components. The SmartDSP OS kernel has these features:

- Predictable and realtime oriented as it is comprised of a preemptive, priority-based, single core with an event-driven scheduler.
- Provides interrupt handling, memory management, and task scheduling.
- Allows both user and OS code to be executed in HW and SW interrupts, and tasks.
- Supports a dual-stack pointer—it uses both exception and normal stack StarCore pointer registers.
- Supports multi-core by executing an SDOS instantiation on each core and allowing inter-core communication and synchronization.
- Provides an API and functionality for most DSP core and DSP subsystem components.

This chapter details the following SmartDSP OS kernel components.

- 2.18 B4860 L1-Defense

2.1 Interrupts

An interrupt triggers a new event while an interrupt handler executes and services an interrupt request. An SoC can have multiple levels of interrupt concentrators; the kernel program them during application bring-up and runtime. For example,

- MSC8101, MSC812x — GIC, LIC, and PIC
- MSC814x, MSC815x, PSC9x3x, B4860 — GCR and EPIC

SmartDSP OS uses the ESP stack pointer for interrupt handling.

SC processor interrupts are either MI or NMI:

- MI
  - Application can ignore the interrupt request.
  - Priorities can vary between interrupt sources.
- NMI
  - Interrupt request is critical.
  - Interrupt handler must execute.

Consider the following SC interrupt characteristics:

- StarCore SR calls `osStart()` from `main()` to enable interrupts.
- User NMI ISR should not call OS functions guarded by disabling and enabling interrupts; e.g., `osHwiSwiftDisable()` and `osHwiSwiftEnable()`.
  - Prevents data or code corruption.
  - Based on the broken assumption of atomicity in these code portions—a false assumption in NMI.

NOTE Ensure that SoC-level NMIs, such as NMI_B and virtual NMIs, are NOT triggered during kernel initialization [prior to calling `osStart()`].

2.1.1 Interrupt Sources

SmartDSP OS interrupts are primarily generated from a single origin. MSC814x interrupt sources include the following:

- SC3400 core includes trap-generated interrupts (used for SW interrupts), illegal instructions, and debug exceptions.
- DSP subsystem includes MMU, cache, and timer interrupts.

- DSP SoC includes peripheral-generated interrupts, VIRQ, and external interrupts lines.

2.1.2 Interrupt Types

SmartDSP OS kernel supports HWI and SWI interrupts.

2.1.2.1 Hardware Interrupts

HW devices such as DMA, OCeaN DMA, and Ethernet controllers can generate HW interrupts. In SmartDSP OS kernel, the OS interrupt dispatcher handles HW interrupts in one of these ways:

- Calls an OS interrupt handler.
- Calls an OS interrupt handler before the interrupt calls a user-defined interrupt handler.
- Calls a user-defined interrupt handler.

HW interrupt characteristics include the following:

- Every interrupt handler should return void, and receive a single parameter.
- The received parameter is part of the interrupt registration; it is set at registration time.
- The parameter can act as a global variable pointer; the variable may change over the course of an application.
- Application-provided interrupt handles must be given a HW interrupt function; this is allocated upon registering the HW interrupt handle.
- HW interrupt prototype function:
  `typedef void (*os_hwi_function)(os_hwi_arg).`
- HW interrupts are HW platform-specific thus SmartDSP OS kernel contains header files specific to given HW platforms. For example,
  - `msc814x_hwi.h` header file defines HW interrupts for the MSC814x platform.
  - `msc815x_hwi.h` header file defines HW interrupts for the MSC815x platform.

SmartDSP OS kernel features include the following:

- Supports edge and level HW interrupts.
- MMU exception interrupts are hardwired for edge mode.
- An application can define edge and level HW interrupts as follows:
  - #define LEVEL_MODE
  - #define EDGE_MODE
- Enables high priority interrupt handlers to finish before low priority interrupt handlers begin executing.

  - Respectively, parameters `OS_HWI_PRIORITY0` and `OS_HWI_LAST_PRIORITY` specify the highest and lowest priority for HW interrupts.
  - Number of priorities depends on PIC or EPIC.
- Interrupt controller is located in the DSP subsystem of these processors:
  - MSC8101, MSC812x — PIC
  - MSC814x, MSC815x, PSC9x3x, B4860 — EPIC
- Maximum number of HW interrupt priorities:
  - SC140-based SoC = 7
  - SC3400/SC3850-based SoC = 31
  - See *SC3850/SC3400/SC3900 DSP Core Reference Manual*.

Table 2.1 lists SmartDSP OS HW interrupt module functions.

Table 2.1 Functions Available for Hardware Interrupts

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osHwiInitialize()`	Initializes all interrupt handlers to default ones and enables all NMI in EPIC (where supported).
Application bring up	`osHwiCreate()`	Installs an interrupt handler in the OS dispatcher. It sets the interrupt priority in the interrupt assignment register and enables the interrupt.
Application tear down	`osHwiDelete()`	Detaches an interrupt handler from the OS dispatcher.

Listing 2.1 shows how an application installs a hardware interrupt in the OS dispatcher.

1. Application calls the kernel interrupt handler, `osHwiCreate()`.
2. Kernel interrupt handler sets interrupt priority in the interrupt assignment register and enables the interrupt.

See *SmartDSP OS API Reference Manual* for more information.

Listing 2.1 Installing a HWI in the OS Dispatcher

```
os_status osHwiCreate(os_hwi_handle hwi_num,
                      os_hwi_priority priority,
                      os_hwi_mode mode,
                      os_hwi_function handler,
                      os_hwi_arg argument);
```

2.1.2.2 Software Interrupts

A SWI is a program-generated interrupt. SmartDSP OS kernel supports nesting SWIs with HWIs and other priority SWIs. Any HWI or high priority SWI preempt a low priority SWI.

- Highest priority SWIs are specified by parameter `OS_SWI_PRIORITY0`.
- Lowest priority SWIs are specified by `OS_SWI_PRIORITY15`.

An application, using the definition `#define OS_TOTAL_NUM_OF_SWI <num>` in the `os_config.h` file, can specify the number of SWIs per core.

The SWI prototype function is `typedef void (*os_hwi_function)(os_swi_count)`.

NOTE An array of SWI handlers is allocated from the `OS_MEM_LOCAL` heap. There is a memory penalty if too high a value is set.

Table 2.2 lists SmartDSP OS SWI module functions.

Table 2.2 Available SWI Functions

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osSwiInitialize()`	Initializes the software interrupts module.
Application bring up	`osSwiFind()`	Finds the first available software interrupt number.
	`osSwiCreate()`	Attaches an interrupt handler to a software interrupt.

Table 2.2 Available SWI Functions

Flow State	Name	Description
Application runtime	`osSwiActivate()`	Activates the given software interrupt.
	`osSwiCountSet()`	Sets the count of the given software interrupt.
	`osSwiCountInc()`	Increments the count of the given software interrupt and activates it.
	`osSwiCountDec()`	Decrements the count of the given software interrupt. This function also activates the software interrupt if the count is zero after the operation.
Application tear down	`osSwiDelete()`	Detaches the given software interrupt from its handler.

Listing 2.2 shows how to create a SWI using functions `osSwiFind()` and `osSwiCreate()`.

- SWI, `g_swi_holder1`, is generated with the highest priority.
- An interrupt handler, `foo`, services the interrupt request.

Refer to *SmartDSP OS API Reference Manual* for more information.

Listing 2.2 Creating a SWI

```
osSwiFind(&g_swi_holder1);
status = osSwiCreate(foo,                    //software interrupt function
                     g_swi_holder1,          //software interrupt number
                     OS_SWI_PRIORITY0,       //software interrupt priority
                     50);                    //user Id
if (status != OS_SUCCESS) OS_ASSERT;
osSwiCountSet(g_swi_holder1, 100);

status = osSwiActivate(g_swi_holder1);
OS_ASSERT_COND(status != OS_SUCCESS);

void foo(os_swi_count count)
{
    uint16_t self;
    if (count != 100) OS_ASSERT;
    status = osSwiSelf(&self);
```

```
    if (status != OS_SUCCESS) OS_ASSERT;
    status = osSwiDelete(self);
    if (status != OS_SUCCESS) OS_ASSERT;
}
```

2.2 Scheduler

A scheduler determines when a specific task or a process must utilize the CPU. In the SmartDSP OS kernel, the scheduler is preemptive and event driven; that is, an event, such as an interrupt, triggers the scheduler to switch to the next higher priority task or process. The SmartDSP OS kernel does not enforce time slots or any other mechanism of load balancing.

In SmartDSP OS, each core has its own scheduler. The scheduler handles the events in this priority order:

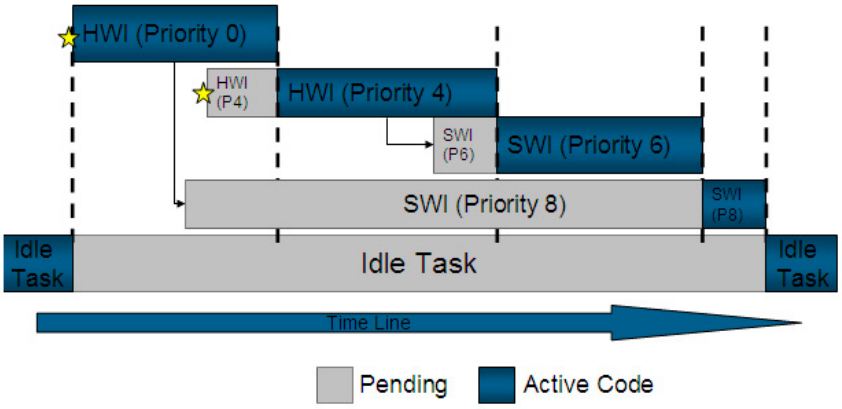
1. NMI
2. HWI
3. SWI
4. Tasks

In other words, the scheduler considers an NMI as the top priority event, and a task as the lowest priority event.

Software interrupts can be activated by hardware interrupts, other software interrupts, and by tasks. Software interrupts can be preempted by hardware interrupts or higher-priority software interrupts. Hardware and software interrupts can be nested. The idle or background task has the lowest priority, and it should never end. It can include user code and is preempted by software interrupts as well as hardware interrupts and higher priority tasks.

Figure 2.1 shows an example of how the scheduler operates with a single background task in the SmartDSP OS kernel. A higher-priority SWI runs to completion before a lower-priority SWI executes. SWIs are preempted by any HWI or a higher-priority SWI. Hardware interrupts always have priority over software interrupts. Hardware and software interrupts can be nested, and the priorities of each are considered.

Figure 2.1 SmartDSP OS Scheduler Operation



In Figure 2.1, the SmartDSP OS Scheduler operates as follows:

1. A HWI with priority 0 (the highest) activates a SWI with priority 8.
2. The SWI with priority 8 waits for HWI with priority 0 to finish.
3. Another HWI with priority 4 is activated as soon as HWI with priority 0 finishes – the HWI with priority 4 is scheduled before the pending SWI with priority 8 (all HWI are scheduled before SWI).
4. The HWI with priority 4 begins executing.
5. The HWI with priority 4 activates another SWI, with priority 6.
6. SWI with priority 6 is scheduled before the pending SWI with priority 8 (SWI with higher priority preempt SWI with lower priority).
7. The SWI with priority 6 begins executing.
8. The SWI with priority 8 begins executing after SWI with priority 6 finishes.

2.3 Tasks

A task represents a specific part of a larger program that is currently loaded in memory. Each task in the SmartDSP OS kernel has its own:

- priority.
- stack and therefore a set of core registers.
- PID and DID (where supported).
- name.
- private data.

A task in the SmartDSP OS kernel is:

- different from SWIs and HWIs because a task has its own context and stack.
- attached to the core on which it was created.

The minimum and default number of tasks in the SmartDSP OS kernel is one. An application can override this setting in the `os_config.h` file using this definition:

```
#define OS_TOTAL_NUM_OF_TASKS  <num>
```

Tasks cannot be shared between cores or stolen. A task can sleep and/or yield the core. An array of task handlers will be allocated off of the `OS_MEM_LOCAL` heap; therefore you should be aware of the memory penalty of defining too many tasks. The `OS_TASK_PRIORITY_HIGHEST` and `OS_TASK_PRIORITY_LOWEST` parameters respectively specify the highest and the lowest priority for tasks. In addition, an application can use the following parameters for specifying a priority other than the highest and the lowest:

- `OS_TASK_PRIORITY_01`,
- `OS_TASK_PRIORITY_02`,
- …
- `OS_TASK_PRIORITY_30`

The maximum number of priorities for tasks is 32. An application can create multiple tasks with the same priority that will be scheduled in round-robin.

The SmartDSP OS kernel implements cooperative and priority-based scheduling among tasks. The priority of a task can be changed during run time by calling the `osTaskPrioritySet()` function. This function can be called from any context, HWI, SWI, or by another task.

An application can prohibit the scheduler from switching to a higher priority task by locking it. Locking will prevail even if the dispatcher is invoked due to some other event such as a HWI. Use these functions to lock and unlock the scheduler, respectively:

- `osTaskSchedulerLock()`
- `osTaskSchedulerUnlock()`

The SmartDSP OS kernel assigns each task with a unique program ID (PID) or data ID (DID) wherever supported in MMU. However, tasks can share a PID and/or a DID if an application specifically adds them to a SmartDSP OS MMU context using these functions:

- `osTaskMmuDataContextSet()`
- `osTaskMmuProgContextSet()`

An MMU context defines a PID or DID (depending on the type of context) as well as the set of enabled MMU segments. By default, all tasks have the set of segments enabled in the system context. A task is not required to belong to the same program and data contexts. For example, task A and B may belong to the same data context while task A and C belong to the same program context. This does not require task B and C to share their data context.

Refer to *SmartDSP OS API Reference Manual* for more information.

Table 2.3 lists the functions available in the tasks module of the SmartDSP OS.

Table 2.3 Functions Available for Tasks

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osTasksInitialize()`	Allocates a scheduler and then allocates tasks and events objects and inserts them in the scheduler.
Application bring up	`osTaskFind()`	Retrieves the index of an unused task.
	`osTaskCreate()`	Creates a task. The task is created in the suspended state. Activate the task using `osTaskActivate()`. For the background task, pass a pointer to the task created using `osTaskCreate()` as an argument to the `osStart()` function.

Table 2.3 Functions Available for Tasks (*continued*)

Flow State	Name	Description
Application runtime	`osTaskActivate()`	Removes the created task from the suspend state. If the task is not suspended, the function has no effect. If the task is not delayed or pending, it is moved to the ready state, and scheduler is called. If this is the highest priority ready task and it is the first in its priority queue, then it will resume or start.
	`osTaskSuspend()`	Moves the task to the suspend state.
	`osTaskDelay()`	Delays the running task by blocking the task for the specified number of ticks.

Table 2.3 Functions Available for Tasks (*continued*)

Flow State	Name	Description
	`osTaskYield()`	Schedules a different task in same priority or higher if available. The SmartDSP OS scheduler allows several tasks to have the same priority level. In this case, the tasks should implement cooperative scheduling to allow other tasks to execute first. The `osTaskYield()` function allows a task to preempt itself in favor of another specific task or next ready task of the same priority.
	`osTaskPriorityReadyCount()`	Counts the number of tasks that are ready in a specified priority.
	`osTaskMmuDataContextSet()`	Sets a data context for a task. It effects the data MMU segments enabled and the DID.
	`osTaskMmuProgContextSet()`	Sets a program context for a task. It effects the program MMU segments enabled and the PID.
	`osTaskSchedulerLock()`	Locks the scheduler so that the running task cannot be replaced.
	`osTaskSchedulerUnlock()`	Unlocks the scheduler so that the running task can be replaced.
Application tear down	`osTaskDelete()`	Deletes a task.

Listing 2.3 shows how to find an existing task and create a new task using the `osTaskFind()` and `osTaskCreate()` functions.

Listing 2.3 Finding and creating a task

```
uint8_t task_stack[TASK_STACK_SIZE];
```

```
status = osTaskFind(&task_handle);
OS_ASSERT_COND(status == OS_SUCCESS);

os_task_init_param.task_function = F;
os_task_init_param.task_name = N;
os_task_init_param.stack_size = SZ;
os_task_init_param.task_arg = A;
os_task_init_param.task = task_handle;
os_task_init_param.task_priority = OS_TASK_PRIORITY_27;
os_task_init_param.private_data = D2;
os_task_init_param.top_of_stack = S;

/* Stack should be in a memory accessible to the task given its MMU
data context */

status = osTaskCreate(&os_task_init_param);
OS_ASSERT_COND(status == OS_SUCCESS);
```

Listing 2.4 shows how to activate and suspend a task using the `osTaskActivate()` and `osTaskSuspend()` functions.

Listing 2.4 Activating and suspending a task

```
status = osTaskActivate(task_handle);
OS_ASSERT_COND(status == OS_SUCCESS);

status = osTaskSuspend(task_handle);
OS_ASSERT_COND(status == OS_SUCCESS);
```

Use the source code shown in Listing 2.5 to verify the task suspension. The source code verifies the task status and deletes the task (using the `osTaskDelete()` function) if the task is suspended. If the task is not suspended, the source code delays the task using the `osTaskDelay()` function.

Listing 2.5 Verifying task suspension

```
if (osTaskStatusGet(task_handle) & OS_TASK_SUSPEND)
{
    status = osTaskDelete(task_handle);     //Deleting the task
    OS_ASSERT_COND(status == OS_SUCCESS);
}
status = osTaskDelay(DELAY_TIME);           //Delaying the task
OS_ASSERT_COND(status == OS_SUCCESS);
```

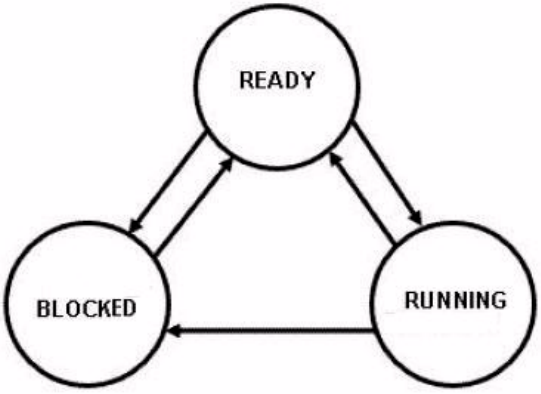
2.3.1 Task States

A task exists in one of these states:

- Ready
- Running
- Blocked

Figure 2.2 shows how a task transitions from one state to another.

Figure 2.2 Task States



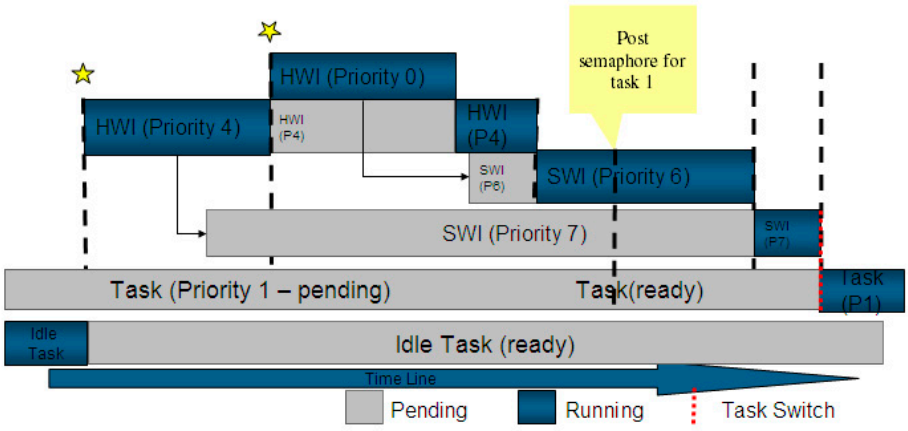
A task changes its state from one to another when:

- A task that is ready to run is in a READY state.
- A ready to run task becomes the highest priority task and starts running, its state changes from READY to RUNNING.
- A higher priority task becomes READY and preempts the current task; the currently running task state changes from RUNNING to READY.
- Another higher priority task suspends a READY task; the suspended task state changes from READY to BLOCKED.
- The task suspends or delays itself; the task state changes from RUNNING to BLOCKED.
- A blocked task gets activated by another task, its delay expires, or a waited event occurs; the task state changes from BLOCKED to READY.
- A task, by default, is created in a BLOCKED state.

Figure 2.3 shows how a task state changes from one to another. In this figure, the arrows represent the SWI activation by HWI and stars represent events, which may or may not immediately trigger their HWI.

An idle task is running. A HWI with priority 0 activates a SWI with priority 6. The SWI (Priority 6) posts the task on the semaphore, which makes the task move from pending to READY state. The task is not yet activated because SWI (Priority 6) and SWI (Priority 7) are still running in the system. The READY task becomes active only when the higher priority event or HWI or SWI finishes execution. Once all the higher priority events finish executing, the scheduler is called and SWI returns back to the interrupt dispatcher. This moves the task from READY to RUNNING state because it is the highest priority task in the system now.

Figure 2.3 Task States Transition



2.3.2 Background Task

The SmartDSP OS kernel requires a task in the background so the system always has at least one task running. The last step of OS initialization activates the background task with the lowest priority. You do not need to explicitly find or create the background task; it is created by the operating system. The background task function pointer is passed as an argument to the `osStart()` function. The background task uses system MMU settings (wherever supported). MMU has system data settings and system program settings. The background task should not return.

An application can override the default stack size for the background task in the `os_config.h` file using this definition:

```
#define OS_BACKGROUND_STACK_SIZE  <size>
```

NOTE An application cannot block, suspend, or delete the background task at any stage during execution.

2.4 Spinlocks

A spinlock is a binary lock which the currently running program waits to become available before continuing the code execution flow. In the SmartDSP OS kernel, spinlocks protect the critical data or code among multiple cores. In addition, spinlocks can be used to synchronize the tasks executing on multiple cores. The spinlock uses atomic test-and-set operations (`bmtset.w`) for synchronization.

In the SmartDSP OS kernel, a spinlock either:

- protects the critical sections from access by other cores and Interrupt Service Routines (ISR) running on the same core, or
- protects the critical section from access by other cores, but not from access by ISRs running on the same core.

NOTE `bmtset.w` is StarCore specific. Refer to *SC3850 DSP Core Reference Manual* for more information about `bmtset.w`.

The data is guarded locally on the core among tasks by enabling/disabling interrupts or by using `osHwiSwiftDisable()`/`osHwiSwiftEnable()` functions. Spinlocks should be used only when data is shared between cores.

A spinlock in the SmartDSP OS kernel is 32 bits in memory, non-cacheable, and in memory which supports atomic operations, such as, M2 in MSC814x architecture and M3 in MSC815x architecture.

NOTE Do not use the same spinlock on the same core in different priorities (HWI, SWI, and/or tasks) without closing interrupts. A deadlock may occur.

The following functions are used for acquiring spinlocks:

- `osSpinLockGet()` — Acquires the spinlock. This is a blocking operation. Use this function with caution because of the possibility of deadlock.
- `osSpinLockIrqGet()` — Disables interrupts and then acquires the spinlock. This is a blocking operation.
- `osSpinLockTryGet()` — Tries to acquire the given spinlock. Returns whether or not the spinlock is acquired. This is a non-blocking operation.

The following functions are used for releasing spinlocks:

- `osSpinLockRelease()` — Releases the spinlock.
- `osSpinLockIrqRelease()` — Releases the spinlock and then enables interrupts.

NOTE The `osSpinLockGet()` and `osSpinLockRelease()` calls do not disable interrupts, which can lead to deadlocks if an ISR tries to access the critical section. Also, performance degradation can occur if one core gets the spinlock and jumps to an ISR from within the critical section, leaving the other cores waiting for the spinlock while the ISR is being served.

To use a fake spinlock, define its address as `OS_GUARD_DISABLE`. `OS_GUARD_DISABLE` is a SmartDSP OS defined macro that when assigned to the spinlock, indicates that no action will take place on the spinlock and the core will not execute any `syncio` (blocking) instructions. The following source code shows how to use the `OS_GUARD_DISABLE` macro:

```
uint32_t dummy_spinlock_addr = OS_GUARD_DISABLE; /* Won't take
spinlock. Won't execute syncio */
osSpinLockGet((volatile uint32_t *)dummy_spinlock_addr);
```

2.5 Memory Manager

The memory manager component of the SmartDSP OS kernel module handles memory allocation and buffer management. The memory manager allocates memory blocks dynamically and frees them when they are no longer in use. The SmartDSP OS memory manager enables efficient allocation, retrieval, and release of memory blocks of known size and alignment without memory fragmentation.

Memory management in SmartDSP OS is split into two stages, initialization and runtime. All initialization functions can be called at runtime and all runtime functions can be called at initialization.

The SmartDSP OS default linker file provides a symmetrical view into the memory of each core, which:

- allows easier transition from single to multicore.
- allows easier reassignment of tasks from one core to another. For example, if a core starts and runs tasks A, B, and C and finds that it does not have the necessary bandwidth to create task D, the other core can create the task D and run it. However, it is not possible for one core to run a task created by another core.
- allows easier debugging of memory-related bugs.
- comes at the cost of aligning all cores to the core with the heaviest requirements.

The SmartDSP OS partitions each shared physical memory into two main fragments:

- Local, which is further fragmented based on the number of cores (Figure 2.4)
- Shared (Figure 2.5 and Figure 2.6)

Figure 2.4 Local Memory

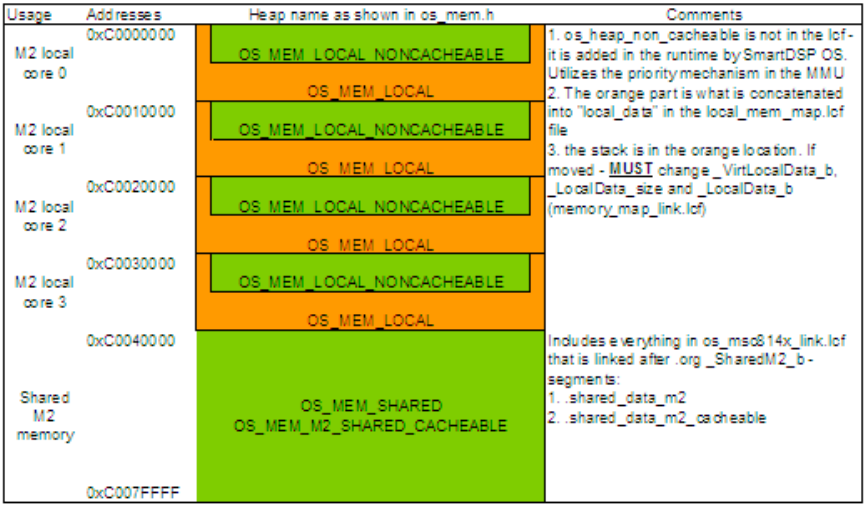


Figure 2.4, Figure 2.5, and Figure 2.6 shows the MSC814x default memory map.

Figure 2.5 Shared M3 Memory

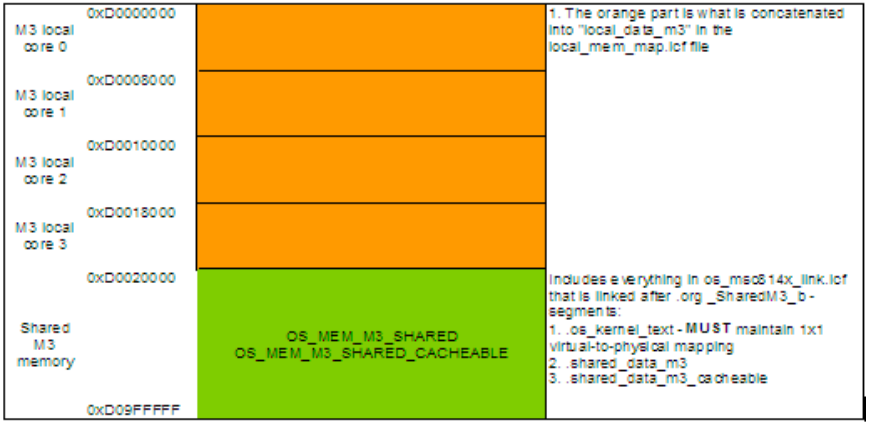
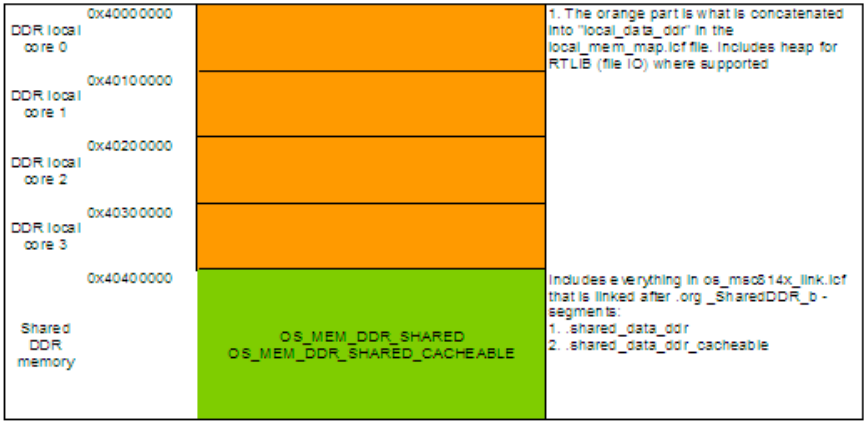


Figure 2.6 Shared DDR Memory



2.5.1 Memory Allocation

The memory manager—using the `osMalloc()` and osFree() functions—supports memory allocation from varying memory sections:

- osMalloc allocates memory
- osFree deallocates memory

Check memory consumption by following these steps:

1. Modify the value of the `OS_PRINT_MALLOC_STATISTICS` flag in `include\common\os_mem.h`.
2. Recompile.
3. Each call to `osMalloc()` will now appear on the output console.

```
Core <core_num>. File <file_name>. Line <line_num>. osMalloc (<size>,
<heap_name>)\n
```

4. If required, redirect output to a given file using `os_malloc_file_handle = fopen(w,`<user_file_name>);

Characteristics `of osMalloc()heaps` are noted below:

- The flag can be called at any time in order to allocate—from a predefined heap—a slab on continuous (virtual) memory.
- `osMalloc()` heaps are either shared or private:
  - Shared heaps are an array—`g_mem_heap_shared[]`—in the `mscxxxx_shared_mem.c` file.

- Private heaps are an array—`g_mem_heap_local[]`—in the `mscxxxx_init.c` file.
- An `os_mem_heap_t` heap has several distinctive characteristics: base address; size; and type (`os_mem_type`)

Listing 2.6 is an example of a heap structure.

Listing 2.6 Structure of a Heap

```
typedef struct
{
volatile uint32_t* mem_start;
volatile uint32_t* mem_size;
volatile uint32_t* mem_type;

volatile void* busy_list_head;
volatile void* free_list_head;
} os_mem_heap_t;
```

A SmartDSP OS application can add heaps into the arrays that can be allocated at compile time. Each heap has a distinctive enumeration. The SmartDSP OS defined heaps are enumerated in `os_mem_type`. The SmartDSP heap enumeration is split into the following three fields, each having a bit mask:

- The valid field verifies the validity of a heap
- The flags field specifies the characteristics (for example, shared, or cacheable) of the heap
- The memory type field specifies the type of memory (for example, M1, M2, M3, DDR) in which the heap resides

All heaps provided by SmartDSP will have `OS_SMARTDSP_HEAP` as one of the flags defined in their enumeration. This flag signifies that the heap is defined by the SmartDSP OS. The following source code snippet shows an example of a heap enumeration:

```
OS_MEM_DDR0_LOCAL = (OS_VALID_NUM | OS_SMARTDSP_HEAP |
OS_MEM_CACHEABLE_TYPE | OS_MEM_DDR0_TYPE),
```

All other flags are assumed to be valid in any user-defined heap identifier.

The main header file for memory management is `os_mem.h`. The memory types, flags, and validity checks are defined in `os_mem.h` as shown in Listing 2.7:

Listing 2.7 Memory type, flag, valid defined in os_mem.h

```
// MASK to extract memory type
#define OS_VALID_NUM_MASK                              0xFFF00000

/**< SmartDSP validity check number is encoded here */
#define OS_MEM_FLAGS_MASK                               0x000FF000
```

```
/**< Memory flags should be encoded in these bits */
#define OS_MEM_TYPE_MASK                                0x0000001F

/**< Memory type should be encoded in these bits */
#define OS_VALID_NUM                                    0xEC900000

/**< Used by SmartDSP code to ensure that this is a valid heap */

// Memory flags

#define OS_SMARTDSP_HEAP       0x00080000   /**< Bit signifying that
the heap is defined by SmartDSP */
#define OS_MEM_CACHEABLE_TYPE  0x00040000   /**< Bit signifying that
the heap is cacheable */
#define OS_MEM_SHARED_TYPE     0x00020000   /**< Bit signifying that
the heap is shared */

// Memory types
#define OS_MEM_M1_TYPE          0x00000001   /**< Heap in M1 memory */
#define OS_MEM_M2_TYPE          0x00000002   /**< Heap in M2 memory */
#define OS_MEM_M3_TYPE          0x00000003   /**< Heap in M3 memory */
#define OS_MEM_DDR0_TYPE       0x00000004   /**< Heap in DDR0 memory */
#define OS_MEM_DDR1_TYPE       0x00000005   /**< Heap in DDR1 memory */
#define OS_MEM_QE_PRAM_TYPE    0x00000006   /**< Heap in QE PRAM memory
*/
#define OS_MEM_MAPLE_PRAM_TYPE 0x00000007   /**< Heap in MAPLE PRAM
memory */
```

NOTE To prevent a user-defined heap identifier from clashing with a SmartDSP identifier; do not define the `OS_SMARTDSP_HEAP` flag as part of the heap identifier.

2.5.2 Buffer Management

The SmartDSP OS memory manager manages a pool of buffers, which can either be allocated statically during compilation or dynamically using `osMalloc()`.The size of the memory manager is dependant on the number of buffers in the managed buffer pool. An application should use the `MEM_PART_SIZE` (<num buffers>) macro to define the memory size to allocate for the memory manager. The number of buffers to be managed by the memory manager is passed as an argument to the macro. An application can also use the `MEM_PART_DATA_SIZE` (<num buffers>, <buffer size>, <alignment>) macro for setting aside the buffer pool.

For example, to manage 100 buffers that are dynamically allocated each having a size of 73 bytes, with an alignment of 8 bytes, your application may call `osMalloc()` function as:

```
osMalloc(MEM_PART_DATA_SIZE(100, 73, ALIGNED_8_BYTES), OS_MEM_LOCAL)
```

to allocate the buffer pool. Also, the macro `MEM_PART_SIZE` will take 100 as the argument while creating memory manager. Memory managers can manage shared as well as private buffer pools. Listing 2.8 displays the source code that an application uses for buffer management. The source code uses the function, `osMemPartCreate()`, which initializes a memory structure for aligned and fixed-size blocks.

Listing 2.8 Initializing memory structure for aligned and fixed-size blocks

```
os_mem_part_t * osMemPartCreate(uint32_t         block_size,
                                uint32_t         num_of_blocks,
                                uint8_t          *data_address,
                                uint32_t         alignment,
                                uint16_t         buffer_offset,
                                os_mem_part_t    *mem_part
 #if (OS_MULTICORE == 1)
                                 ,bool shared
 #endif
                          );
```

The `osMemPartCreate()` function contains the following parameters:

- `block_size` — Specifies the size of the allocatable block.
- `num_of_blocks` — Specifies the maximum number of allocatable blocks.
- `data_address` — Specifies the address of the contiguous block of buffers to manage. The user application may use the `MEM_PART_DATA_SIZE()` macro to set aside this memory.
- `alignment` — Specifies how the blocks are aligned (in bytes).
- `buffer_offset` — Specifies an offset in the buffer to be reserved for Low Level Driver (LLD) or application use.
- `mem_part` — Specifies the address of the memory manager space to initialize. The user application may use the `MEM_PART_SIZE()` macro to set aside this memory.
- `shared` — Specifies a flag that indicates whether this memory partition is shared among cores or not. It receives a value of `TRUE` or `FALSE`.

Listing 2.9 shows an example of how to allocate memory statically and then call the `osMemPartCreate()` function to manage the memory.

Listing 2.9 Allocating memory statically

```
os_mem_part_t *jobs_pool;
uint8_t       jobs_space[MEM_PART_DATA_SIZE(NUM_JOBS_PER_CORE,
sizeof(app_type_job), ALIGNED_4_BYTES)];
uint8_t       jobs_mem_manager[MEM_PART_SIZE(NUM_JOBS_PER_CORE)];
jobs_pool=
osMemPartCreate(ALIGN_SIZE(sizeof(app_type_job),ALIGNED_4_BYTES),
```

```
                                    NUM_JOBS_PER_CORE,
                                    jobs_space,
                                    ALIGNED_4_BYTES,
                                    OFFSET_0_BYTES,
                                   (os_mem_part_t *)&jobs_mem_manager,
                                    FALSE);
```

In the above source code example, `jobs_pool` is the memory manager that manages the buffer pool.

`jobs_space` is the actual buffer structure that is to be managed. It stores the number of bytes that are to be managed by the memory manager. It can be initialized using the `MEM_PART_DATA_SIZE` macro as an argument, which contains the following three arguments:

- Number of buffers — `(NUM_JOBS_PER_CORE)`
- Size of buffers — `(sizeof(app_type_job))`
- Alignment of buffers — `ALIGNED_4_BYTES`

`jobs_mem_manager` manages job space itself using `jobs_pool`. It is a private partition and not shared. It can be initialized using the `MEM_PART_SIZE(NUM_JOBS_PER_CORE)` macro, which contains the `NUM_JOBS_PER_CORE` argument.

Based on this memory allocation, the `osMemPartCreate()` function is called where:

- `ALIGN_SIZE (sizeof(app_type_job), ALIGNED_4_BYTES)` — Indicates that memory manager manages the memory blocks where the size of each block is the size of the structure aligned. The aligned structure is defined in `os_mem.h`. The alignment of the structure is four bytes.
- `NUM_JOBS_PER_CORE` — Indicates the number of blocks that the memory manager will manage.
- `jobs_space` — Indicates the address of the blocks.
- `jobs_mem_manager` — Indicates the memory manager containing the number of jobs per core to be managed.
- `FALSE` — Indicates that the memory partition is not shared.

The `osMemPartCreate()` function returns a pointer to the initialized memory managing structure `(os_mem_part_t)`.

Use these functions to retrieve buffers:

- `void * osMemBlockGet(os_mem_part_t *mem_part)` — Retrieves a block form the memory manager.
- `void * osMemBlockUnsafeGet(os_mem_part_t *mem_part)` — Retrieves a block form the memory manager (unsafe version). Use this function when you are sure that the memory manager cannot be accessed from any other source.
- `void * osMemBlockSyncGet(os_mem_part_t *mem_part)` — Retrieves a block from the memory manager (multicore safe version).

Use these functions to free buffers:

- `void osMemBlockFree(os_mem_part_t *mem_part, void *addr)` — Releases a block back to the memory manager.
- `void osMemBlockUnsafeFree(os_mem_part_t *mem_part, void *addr)` — Releases a block back to the memory manager (unsafe version). Use this function when you are sure that the memory manager cannot be accessed from any other source.
- `void osMemBlockSyncFree(os_mem_part_t *mem_part, void *addr)` — Releases a block back to the memory manager (multicore safe version).

2.6 Memory Management Unit (MMU)

An MMU is a hardware component that is used to extend memory addressing capabilities and control access to Random Access Memory (RAM) and entire SoC. The MMU translates the logical addresses into physical memory addresses. An MMU is used to map and protect memory segments on each core.

The SmartDSP OS for MSC814x and later platforms provides MMU support, which:

- supplies hardware memory protection for instruction and data accesses.
- supports the bridging of two privilege levels (user and supervisor).
- implements high-speed address translations from virtual to physical address.
- provides cache and bus control for advanced memory management.
- supports software coherency with cache sweep operations that invalidate, flush, and synchronize.

For address translation, there is a software model in which the source code is written in virtual addresses that are translated to physical addresses before issuing them to memory.

The SmartDSP OS distinguishes between program and data attributes of the MMU. The SmartDSP OS extensively uses the MMU for building the same virtual memory map that includes both local and shared memory for each core. Shared memory is accessible by all the cores and local regions are accessible by only one core. The MMU makes it easier to port single core applications by creating an identical single core alike virtual map. In this way, every core owns its local memory with explicitly defined regions for shared memory.

2.6.1 MMU Features

As part of the DSP Subsystem, an MMU:

- provides memory region support as follows:
  - Provides read allowed/not allowed for both supervisor and user levels in the program memory region
  - Provides read/write allowed/not allowed for both supervisor and user levels in the data memory region

- defines address translation for each data/program memory region so that a virtual memory region can be allocated to a valid physical memory space.
- stores the program task ID and data task ID for multi-task mechanism.
- handles access error detection with support for non-mapped memory access and misaligned memory access.
- presents two abstractions, Segment and Context, in SmartDSP.
- defines cache policy on each segment.

2.6.2 MMU Segment

An MMU segment defines a memory region and its attributes, protection, and address translation. Each MMU segment is either a program or data but not both. Segment programming is predefined in the Linker Command File (LCF), during runtime, or a combination of both. Refer to the relevant DSP subsystem Reference Manual for MMU alignment restrictions.

An MMU segment includes:

- memory translation
- permissions
- cache properties
- memory properties

NOTE The SmartDSP OS collectively refers to data and program segments as system segments.

2.6.2.1 MMU Segment Abstraction

MMU segment abstraction includes cache attributes and provides a single translation from virtual to physical addresses (see Figure 2.7).

Figure 2.7 MMU Segment

Both system and user segments are available in the SmartDSP OS. System segments are included in the SmartDSP OS application contexts. User segments are only included through a specific context. Multiple contexts can include the same user segment.

2.6.2.2 MMU Segment Creation

In the initialization phase, the SmartDSP OS creates MMU segments according to the executable, using an environment-specific format. In CodeWarrior IDE, for example, an application uses the Linker Command File (LCF) for segment definition and attribute and ID associations. In this case, the SmartDSP OS enables all system segments and adds them to the system context. The SmartDSP OS adds user segments to the user contexts according to the context definition.

An application creates a MMU segment explicitly using the API functions, `osMmuProgSegmentCreate()` and `osMmuDataSegmentCreate()`. These functions create MMU segments according to the given parameters. After creating the segment, the segment can be added to any context using the `osMmuDataContextSegmentsAdd()` or `osMmuProgContextSegmentsAdd()` functions. In such cases, the segment is active when the context is active. It is also possible to enable the segment using the `osMmuDataContextActivate()` function.

NOTE If a MMU segment is added to the current context, the SmartDSP OS activates it upon the next activation of the context.

An example of the activation of an MMU segment is shown in Listing 2.10.

Listing 2.10 Activating a MMU segment

```
status = osMmuDataSegmentFind(&data_segment);
if (status != OS_SUCCESS) OS_ASSERT;

status = osMmuDataSegmentCreate(data_segment,(void*)VIRT_BASE1,
(void*)PHYS_BASE1,
SEGM_SIZE,
MMU_DATA_DEF_SYSTEM,
NULL);
if (status != OS_SUCCESS) OS_ASSERT;

status = osMmuDataSegmentEnable(data_segment, TRUE);
if (status != OS_SUCCESS) OS_ASSERT;
```

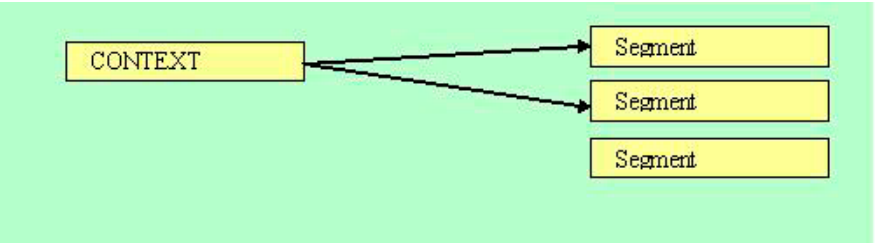
In the above example, the application activates a segment rather than adding it to a given context. The segment is active only until the next context switch.

2.6.3 MMU Context

A SmartDSP OS MMU context is a group of segments that can be associated with a task. An MMU context defines a subset of software context.

An MMU context defines a set of active segments as well as an ID. This ID is the PID/DID of all tasks belonging to the context. An MMU context is shown in Figure 2.8.

Figure 2.8 MMU Context



There are two types of contexts. A *program* context is a set containing a unique program ID and a subset of defined program segment descriptors. A *data* context is a set with a unique data ID and a subset of defined data segment descriptors.

Each task belongs to a single data context and program context. By default, tasks belong to system contexts but for segment mapping, they maintain their own unique PID/DID. A task belonging to a context, which has been associated to the task during runtime, has data ID at that context and the set of segments enabled in that context. Tasks do not need to belong to the same context for data and program. Each task is created using the MMU segment of the system context. The system PID/DID is 1.

2.6.3.1 MMU Context Abstraction

The SmartDSP OS preserves data and program contexts on each HWI and SWI. System context is enabled on each task and HWI/SWI interrupt. The system contexts begin by including segments for which `MMU_PROG_DEF_SYSTEM` or `MMU_DATA_DEF_SYSTEM` was set in the segment attributes in LCF.

User context normally includes all system context segments because they must be enabled continuously. However, in some cases, system context segments are removed from the user context. System context usually only includes system segments but it can also include user segments.

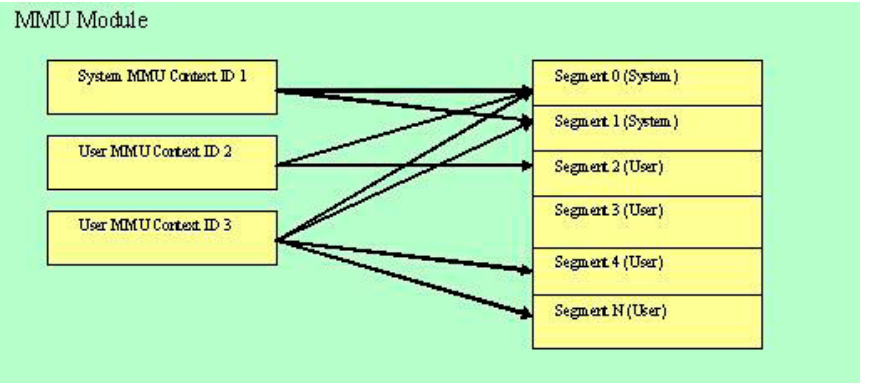
NOTE Changes to the system contexts are reflected in all contexts.

2.6.3.2 MMU Context Creation

The SmartDSP OS creates all MMU contexts using information from LCF. An application creates a MMU context explicitly by using `osMmuDataContextFind()` and `osMmuProgContextFind()`. These functions send a pointer to the first available context and set a supplied ID to the context. An application modifies the MMU context by adding new segments using `osMmuProgContextSegmentsAdd()` or `osMmuDataContextSegmentsAdd()`.

The `osMmuProgSystemContextSegmentAdd()` function adds a given segment to the system program context and this reflects in all program contexts. You can add a segment to the system context only by using the `osMmuDataContextFind()` function with the parameter, `OS_MMU_SYSTEM_CONTEXT`. You can associate an MMU context with a SWI by calling the `osSwiMmuDataContextSet()` function. This allows SmartDSP OS to activate the system context when a SWI starts and then replace the system context with the context you specify. An application enables the context by calling the `osMmuDataContextActivate()` function, which has the same effect as a task switch. An MMU context creation is shown in Figure 2.9.

Figure 2.9 MMU Context Creation



2.6.4 MMU Configuration

The main header file of the SmartDSP OS MMU is `include\common\os_mmu.h`. The number of data and program contexts have default values which can be overridden in `os_config.h` in the following definitions:

- `#define MMU_DATA_CONTEXT_NUM <num>`
- `#define MMU_PROG_CONTEXT_NUM <num>`

An array of data and program contexts will be allocated from the `OS_MEM_LOCAL` heap. The user application will therefore face the penalty in terms of memory footprints for defining an improper number

of contexts or for defining too many contexts that override the default value. Tasks and contexts share the reserved number of IDs that are defined in the MMU (platform specific), which is `1+num_tasks+num_contexts<=max_ID`.

MMU segments (both data and program) that are not defined in the LCF can be added during initialization or runtime using the following functions:

- `osMmuDataSegmentFind()`
- `osMmuDataSegmentCreate()`
- `osMmuDataSegmentEnable()`
- `osMmuProgSegmentFind()`
- `osMmuProgSegmentCreate()`
- `osMmuProgSegmentEnable()`

Once an MMU segment is created, it can be added to a MMU context using the following functions:

- `osMmuDataContextSegmentsAdd()`
- `osMmuProgContextSegmentsAdd()`
- `osMmuDataSystemContextSegmentAdd()`
- `osMmuProgSystemContextSegmentAdd()`

MMU Segments can also be removed from MMU contexts using the following functions:

- `osMmuDataContextSegmentRemove()`
- `osMmuProgContextSegmentRemove()`
- `osMmuDataSystemContextSegmentRemove()`
- `osMmuProgSystemContextSegmentRemove()`

To perform address translation, an application can call the following functions:

- `osMmuProgVirtToPhys()`
- `osMmuDataVirtToPhys()`

To perform address validation, an application can call the following functions:

- `osMmuProgVirtProbe()`
- `osMmuDataVirtProbe()`

2.6.5 MMU Exceptions

MMU exceptions are non-maskable and precise. The SmartDSP OS has default handlers for program and data MMU exceptions, which can perform the following tasks:

- Detect the context from which the error occurred
- Detect the root cause of the error and clear the interrupt in the MMU

- Call a user-defined debug hook if enabled
- Stop the core

An application can override these exceptions by calling `osHwiDelete()` followed by `osHwiCreate()` for `OS_INT_DMMUAE` (data MMU) and/or `OS_INT_IMMUAE` (instruction MMU). Use the function, `osMmuDataErrorDetect()` or `osMmuProgErrorDetect()` to clear the interrupt and return one of the root causes of the exception, as shown:

- `status = osMmuDataErrorDetect (struct os_mmu_error *)`
- `status = osMmuProgErrorDetect (struct os_mmu_error *)`

An example of returning the cause of an MMU error is shown in Listing 2.11.

Listing 2.11 Returning the cause of MMU error

```
struct os_mmu_error
{
  uint32_t error_address;         //violation address
  uint32_t error_pc;             //program counter that caused exception
  uint32_t rw_access;            //read is 0, write is 1
  uint32_t privilege_level;  //user is 0, supervisor is 1
  uint32_t access_width;     //Access Violation Width
};
```

There are two non-contradicting procedures to debug a MMU error.

First procedure:

1. Look at the root cause of the error using status of the `os_mmu_error` structure.
2. View memory as program at the `err.error_pc` to look at the program counter where error happened.
3. See what you are doing.
4. Compare with what you meant to have been doing.
5. See where the error originated.

Second procedure:

1. Step out of the handler.
2. Step through the interrupt dispatcher (`_osHwiPreciseDispatcher`).
3. Step back into the code.
4. Look at the stack and see where the error originated.

2.7 Caches

A cache is a high-speed memory area where the information is stored temporarily for fast access. The information retrieved from main memory is copied to the cache. The CPU fetches the information from the cache instead of main memory. This reduces the access time and enhances memory performance.

The SmartDSP OS supports different types and hierarchies of cache memories. Level 1 or L1 cache is an internal cache that is generally very small and private. Level 2 or L2 cache is the secondary cache, also referred to as external cache. It is larger and slower, compared to L1 cache.

The SmartDSP OS kernel has extensive cache functionality for the MSC814x and MSC815x platforms. The SoC caches supported by StarCore processors are shown in Table 2.4.

Table 2.4 StarCore SoC Caches

Processor	Cache
MSC8101	No cache
MSC812x	private L1-I$
MSC814x	private L1-I$/D$, shared L2-I$
MSC815x	private L1-I$/D$, private unified L2

NOTE For MSC812x, the SmartDSP OS only enables or disables the cache.

2.7.1 Cache Sweep Commands

Each MMU segment has specific cache characteristics defined during segment creation or in the LCF. An application utilizes cache sweep commands, which are a mechanism to support platform cache coherency. This mechanism enables data sweeping (such as invalidate, flush, and synchronize) of a specific range of addresses that are programmed in the cache registers.

The SmartDSP OS supports two types of sweep commands:

- Synchronous (or blocking) — The sweep command is accomplished after return. The synchronous function, `osCacheDataSweep()` performs the requested action on data cache and polls until the operation is finished.
- Asynchronous (or non-blocking) — The sweep operation can be in process after return. The asynchronous function, `osCacheDataSweepAsync()` performs the requested action on data cache and returns.

Both the synchronous and asynchronous functions initially check whether previous commands have been completed or not. An application can check the status of the sweep operation by using the `osCacheDataInProgressSweep()` function.

2.7.2 Configuration of Caches in SmartDSP OS

The main header file of the SmartDSP OS caches is `include\common\os_cache.h`.

The SmartDSP OS distinguishes between the various caches in the system and provides an API for each supported type. At initialization, caches are configured based on the following definitions in the `os_config.h` file:

- `#define DCACHE_ENABLE  ON`
- `#define ICACHE_ENABLE  OFF`
- `#define L2CACHE_ENABLE  ON  //MSC815x only`
- `#define OS_L2_CACHE_SIZE  ((uint32_t)&_L2_cache_size)`
  `// in MSC815x only for specifying how much shared M2 and L2 be used as core M2 memory or as L2 cache`

There are two types of hardware-supported cache operations: Global and By Address. The function, `osCacheL2ProgSweepGlobal()` sweeps the L2 program globally on the entire cache. The function, `osCacheProgSweep()`, performs a sweep operation by address on program cache.

The L1 caches in MSC814x and MSC815x platforms work with virtual addresses. L2 caches do not work with virtual addresses. Non-global L1 cache operations require a task ID. On shared segments, an application should use `MMU_SHARED_PID`.

Non-global cache operations must specify addresses with the granularity of cache lines. A cache line contains multiple Valid Bit Resolution (VBR) chunks of memory.

Although cache operations for hardware can operate on VBR-by-VBR basis, the minimum granularity for L1 cache operations is the cache line. This means that cache operations are performed on all VBR in the cache line. Therefore, the SmartDSP OS enforces the granularity of cache lines while calling the cache sweep operation.

The SmartDSP OS provides the `ARCH_CACHE_LINE_SIZE` macro for allowing easy alignment to the L1 cache line size. The following macros are available for aligning cache operations to line size:

- `#define CACHE_OP_LOW_ADDR (VIRT_BASE, GRANULARITY)`
- `#define CACHE_OP_HIGH_ADDR (VIRT_BASE, SIZE, GRANULARITY)`

The alignment of cache operations to cache line size is shown in Listing 2.12.

Listing 2.12 Alignment of cache operations to the cache line size

```
long read_from_maple(void* addr, void* data, uint32_t size, uint32_t
dev_id)
{
```

```
    os_status  status;
    uint32_t   cache_addr_lo,cache_addr_hi;

    cache_addr_lo=CACHE_OP_LOW_ADDR(data, ARCH_CACHE_LINE_SIZE);
    cache_addr_hi=CACHE_OP_HIGH_ADDR(data, size,ARCH_CACHE_LINE_SIZE);

    osCacheDataSweep((void*)cache_addr_lo,
                     (void*)cache_addr_hi,
                      MMU_SHARED_PID,
                      CACHE_FLUSH);

    return (long)size;
}
```

2.8 Queues

A queue is a linear data structure which contains a collection of entities. The entities are added at the rear end and removed from the front end. A SmartDSP OS queue implements a First in First Out (FIFO) algorithm. Queues in SmartDSP OS are used as a basic component in various abstraction layers and utilities. The queues are private per core and they queue data that is four bytes in size (`uint32_t`). A queue can be either specific to a core, or can be shared by all cores. All queues are protected from access by the same core.

There is a special version of queues called multiple queues, which queue different sizes of data. The size is determined while creating the queue. Currently, the queue component supports the queuing of 32 bit values only. In the configuration file, an application can define the maximum number of both regular queues and shared queues allowed by the system.

Queues send one or more messages to a task. A task can use a queue to post messages to software or hardware interrupts. A task or interrupt handler can place a message in the form of a pointer in the queue as well as receive messages from the queue.

The SmartDSP OS provides two kinds of queue, shared and private. Shared queues use spinlocks and are allocated using `OS_MEM_SHARED`. Private queues are allocated using `OS_MEM_LOCAL` and override the spinlock mechanism using `OS_GUARD_DISABLE`.

Queue runtime API may be either safe or unsafe, which are equally legal for both shared and private queues. The functions in the safe queue runtime API are protected with `osSpinLockIrqXxx()`. No protection is provided in the unsafe queue runtime API.

An application defines the number of queues in the `os_config.h` file as:

```
#define OS_TOTAL_NUM_OF_QUEUES    48
#define OS_TOTAL_NUM_OF_SHARED_QUEUES  0
```

Perform the following steps to create a queue. You should perform these steps in your application's initialization source code because these steps involve memory allocations.

1. Get a handle to a queue by calling `osQueueFind()`, specifying whether you need to find a shared queue or a regular queue.
2. Call `osQueueCreate()` after you find a valid queue handle. This function initializes the queue with the required size.
3. Use `osQueueEnqueue()` to start inserting items into the queue, and `osQueueDequeue()` to remove items.

The functions available in the queue module of the SmartDSP OS are listed in Table 2.5.

Table 2.5 Functions Available for Queues

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osQueueInitialize()`	Initializes the OS queues.
Application bring up	`osQueueFind()`	Finds the first available queue number.
	`osQueueCreate()` or `osQueueCreateMultiple()`	Prepares a queue for operation.

Table 2.5 Functions Available for Queues (*continued*)

Flow State	Name	Description
Application runtime	`osQueueEnqueue()` or `osQueueEnqueueMultiple()`	Inserts a value into a queue.
	`osQueueUnsafeEnqueue()` or `osQueueUnsafeEnqueueMultiple()`	Inserts a value to a queue (unsafe version). Use this function when a queue object cannot be accessed from any other source.
	`osQueueHeadEnqueue()`	Inserts a value into the front of a queue.
	`osQueueUnsafeHeadEnqueue()`	Inserts a value into the front of a queue (unsafe version). Use this function when a queue object cannot be accessed from any other source.
	`osQueueDequeue()` or `osQueueDequeueMultiple()`	Removes the first value from a queue.
	`osQueueUnsafeDequeue()` or `osQueueUnsafeDequeueMultiple()`	Removes the first value from a queue. Use this function when a queue object cannot be accessed from any other source.
	`osQueueTailDequeue()`	Removes the last value from a queue.
	`osQueueUnsafeTailDequeue()`	Removes the last value from a queue. Use this function when a queue object cannot be accessed from any other source.

Table 2.5 Functions Available for Queues (*continued*)

Flow State	Name	Description
	`osQueueStatus()`	Queries the status of a queue, such as `OS_QUEUE_EMPTY` or `OS_QUEUE_FULL`.
	`osQueueUnsafeStatus()`	Queries the status of a queue, such as `OS_QUEUE_EMPTY` or `OS_QUEUE_FULL`. Use this function when a queue object cannot be accessed from any other source.
	`osQueueReset()` or `osQueueResetUnsafe()`	Empties the given queue
	`osQueuePeek()`	Queries the first value in the queue without removing it.
Application tear down	`osQueueDelete()`	Deletes the given queue from the system.

All queues, including unshared queues, are protected from access by the same core. In a multicore system, an application can request the queue to be a shared queue. The shared queue object resides in the shared memory and is protected from access by other cores.

2.9 Multicore Programmable Interrupt Controller (MPIC)

This section describes SmartDSP OS support for the Multicore Programmable Interrupt Controller (MPIC) device. MPIC support has two functions—to initialize and configure the MPIC block and to manage device IRQ resources so that each peripheral may assign itself an IRQ.

The driver allows the masking of each interrupt source. The interrupt controller, which provides multiprocessor interrupt management, is responsible for the following stages:

1. Receives hardware-generated interrupts from varying sources, both internal and external.
2. Prioritizes the interrupts in the context of having been generated from within the MPIC; e.g., messaging, timer, and inter-processor interrupts.
3. Delivers the interrupts to the appropriate destination for servicing.

2.9.1 Features

Primary MPIC driver features are noted below:

- Interrupt sources:
  - 64 shared message signal interrupt (MSI) sources (equally divided between four groups).
    - Maximum of 32 events for each shared interrupt register.
  - Sixteen 32-bit messaging interrupt channels (equally divided between two groups):
    - used for cross-program communication
    - triggered on a write register
    - cleared on a read register
  - 12 external (off-chip) interrupt sources are supported.
  - 256 internal interrupt sources are supported.
  - Four inter-processor interrupt channels.
  - Eight global, high-resolution timers (equally divided between two groups) that can be clocked with the platform clock or RTC input.
- All interrupts have 16 programmable priority levels.
- Able to recover from false interrupts.
- Programming model is compliant with OpenPIC architecture.

2.9.2 Programming Models

2.9.2.1 API Units

The MPIC supported API is separated into three units:

- Initialization—initializes registers and data structures required by the hardware, firmware, and software modules.
- Runtime control—controls runtime routines; e.g., on-the-fly enabling of controllers and features.
- Exceptions—reports errors and special events.

2.9.2.2 General Programming Model

MPIC programming involves the following steps:

- MPIC construction—creates an MPIC driver; it has a default configuration.
- MPIC initialization—transfers the selected configuration (e.g., default) to the hardware.
- IRQ assignment—user can configure and assign selected IRQ resources to a peripheral driver.

- MPIC control operations—MPIC driver supports several control operations; e.g., enabling/disabling selected interrupt sources, issuing inter-processor interrupts, and read/write of MPIC message registers.

Following SmartDSP OS initialization of the interrupt controller, the user can occupy and assign interrupt sources by calling API routines; e.g., osMpicSetIntr. Once an interrupt source is assigned to a peripheral, the user should enable the interrupt source so MPIC can generate an assigned interrupt.

2.9.3 Example Calling Sequence

The below table outlines an MPIC calling sequence.

Table 2.6 Example Calling Sequence

Call Function	Description
osMpicConfig	Function is called by the OS and, • Instantiates the MPIC driver handle. • Saves configuration parameters. • Assigns default values to some driver parameters.
osMpicInit	Function is called by the OS and, • Writes the MPIC configuration to the MPIC hardware. • Initializes all data structures. • Registers the appropriate values.
osMpicSetIntr	Assigns a chosen interrupt source to a specific interrupt handler.
osMpicEnableIntr	Enables an assigned IRQ.
osMpicDisableIntr	Disables an assigned IRQ.
osMpicFreeIntr	Frees an assigned IRQ resource.

2.9.4 Specific Functionalities

2.9.4.1 Assign and Enable Interrupt Sources

Interrupt sources can be assigned to a specific interrupt handler (e.g., peripheral drivers) and, thereafter, released. The following API routines should be used:

- osMpicSetIntr
  - Assigns a chosen interrupt source to a specific interrupt handler.
  - Routine expects to receive the chosen interrupt handler routine and argument.
- osMpicFreeIntr—releases a previously assigned interrupt source.

Following an interrupt assignment, a user can enable and disable an interrupt using the osMpicEnableIntr and osMpicDisableIntr routines.

2.9.4.2 Inter-Processor Interrupts

Inter-processor interrupts are registered for handling in exactly the same manner as other interrupt sources.

The MPIC controller supports four different inter-processor interrupt sources (events).

- Any core can be registered to any event.
- Events can be received concurrently by multiple cores (multicasting).
- Each core can freely issue inter-processor interrupts using the osMpicInterruptCores routine.
- The calling core should indicate both the inter-processor event ID as well as a mask representing which processors to interrupt.

2.9.4.3 MPIC Message Interrupt Use

The MPIC controller supports sixteen different message interrupts (divided equally between two groups).

- Message interrupts can be used as a doorbell mechanism amongst cores.
- Cores are assigned to message interrupts—as with any other interrupt source.

A user can deliver a message interrupt as follows:

- Write a 32-bit value to a message register using the osMpicInterruptCores routine.
- As a result, the chosen core is interrupted.
- The core can then read message data using the osMpicReadMessage routine.

2.9.4.4 MPIC Shared Message Signal Interrupt Use

The MPIC controller supports 64 different shared MSI (divided equally over four groups).

- Each shared MSI can serve up to 32 events.
- MPIC supports a shared MSI that features a coalescence configuration. This feature will enable invocation of an MSI only if all coalescing events have taken place.

2.9.4.5 Resource Management

The MPIC driver uses one main data structure to implement MPIC functionality. Memory allocation for this structure is done during MPIC initialization routines using osMpicConfig and osMpicInit.

2.9.5 Demo Use Cases

\SmartDSP\demos\starcore\b4860\mpic_demo

2.10 Inter Process Communication (IPC)

The IPC mechanism is useful for communications between SoC processes. IPC enables efficient and flexible message transfers between the L1 - L2 processes; it can also be used for L1 - L1 process communications.

IPC is required for a number of interfaces; however, this chapter focuses specifically on IPC communications between the L1-L2 and L1-L1 domains. Communication mechanisms are provided for the following:

- L1/L2 interface as required for communication between MAC and PHY layers.
- L1/L1 communication between cores for single- and multi-mode scenarios.
- L1-L2 framework synchronization for a fully synchronized system startup procedure.
- Status/command exchange between StarCore (SC) and Power Architecture (PA).
- Clock synchronization.
- Trace/debug—details TBD.

2.10.1 Functional Specifications

The IPC module supports the following:

- Dispatch and receipt of messages between cores.
- Cores (SC or PPC) can function as message producers (generate) and/or consumers (receive).

- Interrupts are optional but VIRQ, DSP mesh, and MPIC (MSI) are supported.
- Messages can contain data or a pointer to the data. In the latter case, the message should also contain the length of the data.
- Application should define the depth of each BD ring.

2.10.2 Initialization

PA initialization of the common data structure should be done before SC initialization.

NOTE The assumption is that SmartDSP OS IPC initialization is handled following completion of the Linux boot.

2.10.3 Runtime

- Producer and/or consumer cores can track the number of pending messages.
- Application defines the maximum message length.
- BD rings and messages are located in a shared memory region.
- BD ring physical memory is independent of both message physical location and transported payloads.
- Channel's consumer side can have a callback function—callback function is enabled per channel, as per application requirements.
- Supports debug hooks.

2.10.4 Functional Details: Design Concepts

2.10.4.1 Memory

PA allocates memory regions for heterogeneous PA-SC IPC channel BD rings and messages.

The IPC module holds a local cacheable memory structure that is used, as much as possible, to avoid unnecessary accesses to non-cacheable locations; e.g., producer and consumer indices.

DSP allocates all used memory for SC-SC (DSP-only) channels; this improves performance as it makes non-cacheable memory unnecessary.

2.10.4.2 Channels

The IPC module allows PA or SC to open channels.Two types of channels are supported, pointer and message channels.Messages are passed on unidirectional channels.

- Each channel has a producer and a consumer core.
  - Only the producer core can send messages.
  - Consumer core selects the indication type for use when receiving messages. Indication type configuration is done at channel opening.
- Heterogeneous channels are characterized as follows:
  - PA allocates memory for channels and BD rings.
  - Supplies a pointer to the channel structure.
  - Supplies a pointer to the BD ring of each channel.

2.10.4.3 Pointer Channel

The BD ring pointers of pointer channels are flexible. Regarding functioning, the consumer application allocates buffers while the IPC module sends messages as follows:

1. Producer application passes a pointer to the data; it can also send the length.
2. IPC module places the pointer on the correct BD ring.
3. Message is sent.

2.10.4.4 Message Channel

The BD ring pointers of the message channels are preset and should not be changed. Only the data in the buffers can be changed.

If SC is the (channel) consumer then—

- IPC module allocates buffers per a given heap.
- The heap is supplied by the consumer application.
- The application can choose—for a specific IPC channel—upon which memory heap the buffers will be allocated.

NOTE If PA is the consumer on a message channel then it handles allocations.

If SC is the (channel) producer then—

- Producer application needs the buffer to write from the IPC module.
- The buffer then writes the message data.

- Thereafter the application can send the message.

The application can get several pointers before sending. The number of pointers is ≤ the number of available BDs. Pointers are sent in the same order they were received.

IPC can reserve buffers for applications:

- An application is working with a channel and wants a buffer reserved.
- IPC module replaces the buffer in the BD ring with a new buffer—this step allows the application to use the reserved buffer.
- Work continues with the BD ring.
- When ready, the application releases the buffer.

2.10.4.5 Heterogeneous and DSP-only Channels

Channel structures are roughly divided into two types, heterogeneous and DSP. Each type is initialized separately and is independent of the other. However, it is possible to use only heterogeneous or DSP channels, or both.

- Heterogeneous channels used for PA-SC single- or multi-mode communications.
- DSP channels used for SC-SC communications.

2.10.4.6 Interrupts

Each channel can use interrupts to receive message indicators—VIRQ, DSP mesh, and MPIC (MSI).

- SC may be a consumer of VIRQ, DSP mesh, or MPIC (MSI) interrupts.
- IPC module should ensure the interrupt is not in use by using osMessageQueue.
- Use of MSI interrupts enables coalescing.
- When several channels are ALL confirmed as having coalesced then an interrupt is generated.

2.10.4.7 Debug Hooks

Debug hooks are used on send and receive message functions. The type of debug hooks added to existing debug hooks are as follows:

- `OS_DEBUG_IPC_BASIC_SEND`
  - Argument: pointer to os_het_ipc_channel_t, void pointer to data
  - Debug hook is called in osIpcMessageSendPtr.
- `OS_DEBUG_IPC_BASIC_RECEIVE`
  - Argument: pointer to os_het_ipc_channel_t, void pointer to data

– Debug hook is called in osIpcMessageReceiveCb.

2.10.5 Initialization API

Initialization is compromised of heterogeneous (PA-SC) and DSP-only (SC-SC) IPC initializations.

The PA allocates the IPC structure before running the DSP.

- IPC structure is part of an heterogeneous control structure.
- IPC parameters include the number of channels, channel ID, etc.
- IPC parameters are defined by the PA on the heterogeneous control structure.
- The DSP reads the parameters during initialization and initializes as per the internal structures.
- During application initialization, the application opens channels as producer/consumer as per the desired configuration.

DSP IPC is initialized solely by the DSP.

- IPC parameters are passed by the application with other init parameters (usually inside b4860_config.c).
- Application is given control over the memory location within which each BD ring channel is located.

2.10.6 Runtime API to Application

Heterogeneous IPC-shared data structures are located in a shared, non-cacheable memory. As this memory takes time to read/write the SmartDSP OS, the IPC module strives to minimize transactions to/from these memory locations. This is achieved through the following efforts:

- Data structure information is copied to local, cacheable memory regions; these memory regions are very active.
- Local data structures contain more information than shared structures.
- API, which is given to the application, uses the local data structures.
- The majority of data handling is done on local cacheable data structures.
- Only information, relevant to PA, is copied to the shared memory.

2.10.7 IPC Functions

Flow State	Name	Description
Kernel Bring-up	`ipcInit()`	• User application does not specifically call these functions. • osInitialize() calls the function based on os_config.h parameters and b4860_config.c / psc9x3x_config.c configurations. • The IPC *initialize* function checks the validity of the heterogeneous channel structures as created by PA. • Local cacheable data structures are also created and data is copied to them.

Application Bring-up	`osIpcMultimodeChannelIdFind()`
	`osIpcMultimodeChannelIdFind()`
	`osIpcDspChannelIdFind()`
		Checks if a specified channel is available and returns a handle. Three functions are available:
		1. Find single mode heterogeneous channels (in the one IPC region).
		2. Find multi-mode heterogeneous channels (in several regions).
		3. Find DSP channels (in the one IPC region).

	`osIpcChannelProducerOpen()`	Initializes a channel wherein SDOS is a producer.
	`osIpcChannelConsumerOpen()`	Initializes a channel wherein SDOS is a consumer.

Flow State	Name	Description
Application Runtime	`osIpcMessageSendPtr()`	Function is used to send messages via both message and pointer channels. Message channel: • Pointer must be received by `osIpcMessagePtrGet` before calling this function. • Pointer is passed back for sending; the sending order must be the same as the receipt order. Pointer channel: • Function uses the supplied pointer without limitations.
	`osIpcMessagePtrGet()`	Gets a pointer in order to send a message on a message channel.
	`osIpcMessageReceiveCb()`	• Callback function is used if a message exists on the channel. • The function increments the consumer index after returning from the callback function.
	`osIpcChannelPeek()`	• Peeks into the channel to check if the channel contains a message (to be received). • As a result, returns a pointer. • Function doesn't increment the consumer index and doesn't request a callback.

Flow State	Name	Description
Application Runtime, continued	`osIpcMessageChannelBufferReplace()`	• Relevant when a message channel replaces one buffer with another. • Replacement buffer is used when the current BD is next activated. • Useful when the application receives a buffer and decides to hold it for future use.
	`osIpcMessageChannelBufferRelease()`	Free a buffer held by the application by using osIpcMessageChannelPointerReplace.
	`osIpcIndicationSend()`	Send an indication to the consumer core.
Application Teardown	`osIpcChannelConsumerClose/ osIpcChannelProducerClose`	Not currently supported.

2.10.8 Initialization API Flow

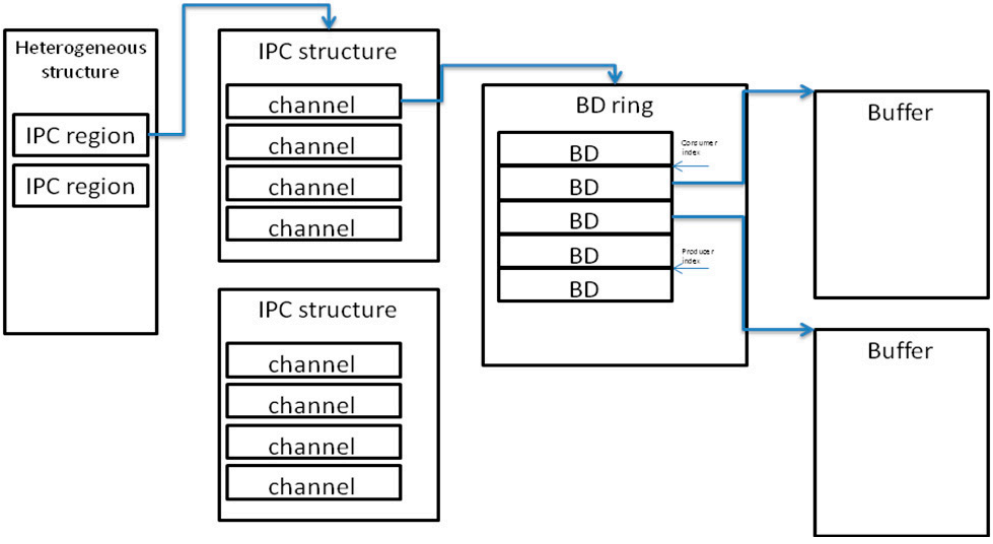
```
main()
osInitialize()
ipcInit()
appInit()
osIpcMultimodeChannelIdFind()
osIpcChannelProducerOpen()
osIpcChannelConsumerOpen()
```

2.10.9 Runtime API Flow

```
Main()
appBackground()
osIpcMessageSendPtr()
```

```
osIpcMessagePtrGet() //for message channels only - must be
        called before osIpcMessageSendPtr
osIpcMessageReceiveCb() //can also be called by interrupt
osIpcChannelPeek()
osIpcMessageChannelBufferReplace()
osIpcMessageChannelBufferRelease()
osIpcIndicationSend()
```

Figure 2.10 Structure and Connections



2.11 Intercore Messaging

In a multicore environment, intercore messaging allows applications running on different cores to communicate with each other. The intercore messaging component of the SmartDSP OS module allows communication between applications through messages by associating interrupts with handlers written in C or assembly language.

Intercore messages in the SmartDSP OS are point-to-point. An application can post an intercore message at any time. Reception of intercore messaging is possible only after creating the message or listening to the message. Intercore messaging in the SmartDSP OS may be either synchronous or asynchronous.

Synchronous intercore messaging uses the following process:

1. The sending core takes a mailbox spinlock.
2. The sending core writes data to a mailbox.
3. The sending core interrupts a recipient.
4. The recipient is interrupted. Mailbox number is returned as an interrupt handler argument (`os_hwi_arg`).
5. The recipient reads mailbox data.
6. The recipient frees the spinlock.

Asynchronous intercore messaging uses the following process:

1. Data is predefined during initialization and set as an interrupt handler argument (`os_hwi_arg`).
2. The sending core interrupts the recipient.
3. The recipient is interrupted.

The VIRQ were used for intercore messaging in MSC812x and MSC814x. In MSC815x, the interrupt mesh between cores is used for intercore messaging. The kernel initializes the inter-core messaging module of the SmartDSP OS using the `osMessageInitialize()` function.

2.11.1 Configuration of Intercore Messaging

The main header file of the SmartDSP OS intercore messages is: `include\common\os_message.h`.

Intercore messaging is only enabled where `OS_MULTICORE` is predefined as `1` (that is, not enabled in MSC8101). The maximum number of supported messages in the architecture is defined in `smartdsp_os_device.h` as:

```
#define MSC814X_MAX_NUM_OF_MESSAGES  4
#define MSC815X_MAX_NUM_OF_MESSAGES  2
```

The number of intercore messages is defined by an application in `os_config.h`, as:

```
#define OS_TOTAL_NUM_OF_INTERCORE_MESSAGES  1
```

NOTE The number of messages should be interpreted as the number of point-to-point messages; that is, multiplied by the number of cores.

An application defines the maximum number of intercore messages allowed by the system in the configuration file. Intercore messaging mailboxes are allocated from the `OS_SHARED_MEM` heap during initialization. Because intercore messaging uses spinlocks, a requirement is placed on the physical memory in which this heap is linked. Refer to *SmartDSP OS API Reference Manual* for more information.

Functions available for intercore messaging are listed in Table 2.7.

Table 2.7 Functions Available for Intercore Messaging

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osMessageInitialize()`	Initializes the multicore synchronization module.
Application bring up	`osMessageFind()`	Finds the first available inter-core message number.
	`osMessageCreate()` or `osMessageCreateAsync()`	Installs an intercore message handler for the calling core to start the communication. When a specific message arrives, this function enables the calling core to be notified. The calling core can then handle the message.

Table 2.7 Functions Available for Intercore Messaging (*continued*)

Flow State	Name	Description
Application runtime	`osMessagePost()` or `osMessagePostIrq()` or `osMessagePostAsync()`	Posts the given message number or ID with the given data to the destination core. This function performs the following tasks: • Posts a virtual interrupt to the destination core and associates a message with this interrupt • Gets a message ID and pass it to handler as parameter While the message is posted, no other core can post the same message number to the same core. As a result, the data is safe until the destination core fetches the message.
	`osMessageGet()`	Retrieves the data from a message (with the message ID) that was posted to the calling core. When your application receives a message, you must call `osMessageGet()` to dismiss the interrupt, even if you are not interested in the contents of the message.
Application tear down	`osMessageDelete()`	Deletes an existing intercore message handler. The deleted message number cannot be used after it is deleted.

2.12 Intercore Message Queues

The intercore message queue module of the SmartDSP OS is used for inter-process communication within the same process or intercore communication which is not necessarily point-to-point. The intercore message queue uses a queue for transferring the messages or content between processes.

Intercore message queues in the SmartDSP OS are point-to-multipoint. Enqueuing and de-queuing from a message queue is possible only after creating it. In all architectures, message queues use VIRQ. As a result, the number of message queues is bound only by the number of other VIRQ users, such as intercore messaging (on MSC812x/MSC814x) and RIONET (on MSC814x/MSC815x), and the number of VIRQ in the SoC.

The kernel initializes the intercore message queue module of the SmartDSP OS by using the `osMessageQueueInitialize()` function.

2.12.1 Configuration of Intercore Message Queues in SmartDSP OS

The main header file of the SmartDSP OS intercore message queue is `include\common\os_message_queue.h`. Intercore message queues are allocated from the `OS_SHARED_MEM` heap during initialization. Intercore messaging is only enabled where `OS_MULTICORE` is predefined to 1; therefore it cannot be enabled in the MSC8101 platform.

The number of intercore message queues is defined by an application in `os_config.h` as:

```
#define OS_TOTAL_NUM_OF_MESSAGE_QUEUES  1
```

Refer to *SmartDSP OS API Reference Manual* for more information.

The functions available in the intercore message queue module of the SmartDSP OS are listed in Table 2.8.

Table 2.8 Functions Available for Intercore Message Queues

Flow State	Name	Description
Application bring up	`osMessageQueueCreate()`	Installs an intercore message handler for the calling core. Then, the calling core can handle the message.
Application runtime	`osMessageQueuePost()`	Posts the given message number with the given data to the destination core.

Table 2.8 Functions Available for Intercore Message Queues

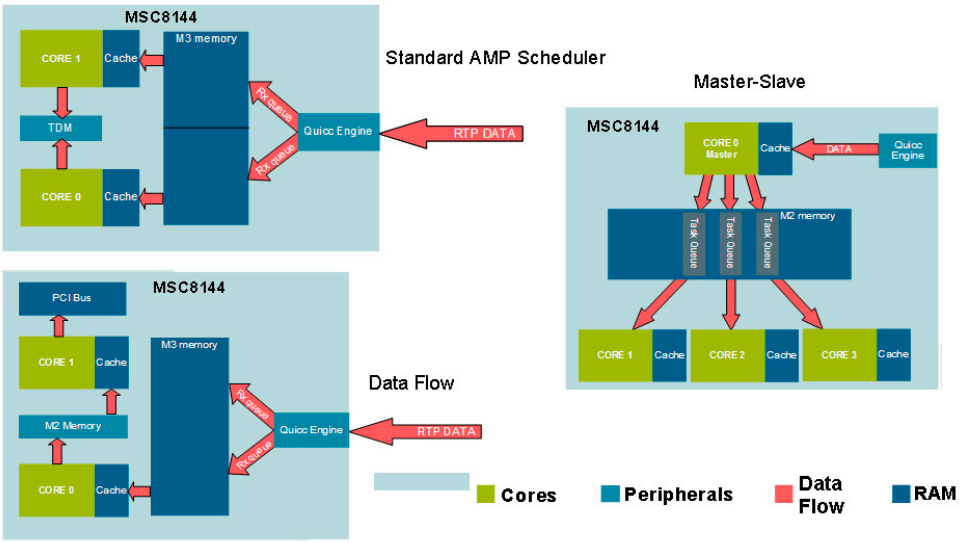
Flow State	Name	Description
	`osMessageQueueGet()` or `osMessageQueueDispatcher()`	Retrieves the data from a message that was posted to the calling core. Note: User may call the function repetitively until the queue is empty: `while (OS_ERR_Q_EMPTY != osMessageQueueGet());`
Application tear down	Not supported	

2.12.2 Intercore Options

The SmartDSP OS provides various intercore options to establish communication between cores in a multicore platform. An application can use any of the following options while using intercore messaging APIs (see Figure 2.11):

- Standard AMP scheduler where each core handles it's own tasks
- Data flow between core using intercore messages
- Master-Slave or producer-consumer using message queues

Figure 2.11 Intercore Options



2.13 Events

An event is a kernel object that enables communication between tasks and the system. Tasks, except background task, can remain pending on events until an event is available or when a timeout occurs. An HWI, SWI, or another task can post the event to restart a blocked task. An event can also be referred as a function in the system that an HWI, SWI, or task can accept to check if it is available. Events enable signals to the blocking task to notify that the task should wake up and give some data to the event, if necessary. An event may include data; therefore, it is possible to post and receive a message from an event.

Events are a ‘base class’ in the SmartDSP OS. There is no public header file for events and an application is not expected to call its functionality directly.

Events cannot be shared among different cores. Events and their derivatives are private in each core. The following lists some of the scenarios of events:

- Events have a list of pending tasks for the specific event.
- If an event is available, the pending task is not blocked and there is no delay in its execution.
- If a function in the system posts an event and a task is pending on that event then the posting function will get the pending task ID.
- If no task is pending on a posted event, it will return `TASK_NULL_ID`.

- If an event is not available, the task gets blocked and a context switch occurs.
- If a task is blocking the number of seconds it specified as timeout, it becomes ready and the pending function returns `OS_ERR_EVENT_QUEUE_TIMEOUT` or `OS_ERR_EVENT_QUEUE_TIMEOUT`.

There are two types of events in the SmartDSP OS: semaphores and queues.

2.13.1 Event Semaphores

A semaphore event is used to synchronize a task with an HWI, SWI, or another task. Semaphore events can be posted multiple times or accepted an equal number of times. To use a semaphore event for synchronization purposes, set its value to 0. For example, a task initiates an (I/O) operation and then waits for a semaphore. When the I/O operation finishes, an SWI, HWI, or another task sets the semaphore and the task resumes.

The initialization of event semaphores is handled by the events base class.

An application defines the number of event semaphores in the `os_config.h` file, using this definition:

```
#define OS_TOTAL_NUM_OF_EVENT_SEMAPHORES  5
```

A software or hardware interrupt handler that waits for a semaphore, must call the non-blocking command `osEventSemaphoreAccept()`, or it will fail. Background tasks cannot pend on a semaphore because such tasks must never block. The running task can pend on a semaphore with a possible timeout time measured in ticks. The running context (task or interrupt) can always post to a semaphore.

To create and delete a semaphore, use the functions shown in Listing 2.13:

Listing 2.13 Functions for creating and deleting semaphore

```
status = osEventSemaphoreFind(&s);
OS_ASSERT_COND(status == OS_SUCCESS);

status = osEventSemaphoreCreate(s, 0);
OS_ASSERT_COND(status == OS_SUCCESS);

status = osEventSemaphoreDelete(s);
OS_ASSERT_COND(status == OS_SUCCESS);
```

The following code snippet (Listing 2.14) shows an example of a task pointing to a semaphore, another task pending on it, and interrupt handler accepting it:

Listing 2.14 Tasks pointing to and pending on a semaphore

```
status = osEventSemaphorePend(S, TIMEOUT);           //Task1
OS_ASSERT_COND(status == OS_SUCCESS);

status = osEventSemaphorePost(S, &resumed_task);    //Task 2
```

```
OS_ASSERT_COND(status == OS_SUCCESS);

status = osEventSemaphoreAccept(s);                    //Interrupt Handler
```

The functions available in the event semaphores module of the SmartDSP OS are listed in Table 2.9.

Table 2.9 Functions Available for Event Semaphores

Flow State	Name	Description
Kernel bring up		Already handled by the Events 'base class'.
Application bring up	`osEventSemaphoreFind()`	Receives a handle to create a semaphore event.
	`osEventSemaphoreCreate()`	Creates a semaphore event. Initializes a semaphore counter and an empty waiting list and places them in an event structure.
Application runtime	`osEventSemaphorePend()`	Pends on a counting semaphore if semaphore counter is zero; otherwise, decrements and continues.
	`osEventSemaphoreAccept()`	Continues semaphore without pending. Checks if semaphore counter is not zero, whereupon it decrements the counter.
	`osEventSemaphorePost()`	Posts a counting semaphore. If tasks are pending on the semaphore, it schedules them; otherwise, increments the counter and continues.
	`osEventSemaphoreReset()`	Removes all tasks pending on the event and schedule them if they are ready.
Application tear down	`osEventSemaphoreDelete()`	Resets semaphore and deletes the semaphore. Releases the event structure used for the counting semaphore and resumes all tasks pending on it.

2.13.2 Event Queues

Event queues are used to store events, which are due to be processed. Event queues require both events and queues to be in the system. An application defines the number of event queues in the `os_config.h` file, as:

```
#define OS_TOTAL_NUM_OF_EVENT_QUEUES  3
```

NOTE The function for deleting an event queue is not supported, yet.

A handler that receives a message must call the non-blocking command, `osEventQueueAccept()`, or it will fail. Background tasks cannot pend on a queue, because such tasks must never block.

To find and create an event queue, use the functions in Listing 2.15:

Listing 2.15 Finding and creating an event queue

```
status = osEventQueueFind(&Q);
OS_ASSERT_COND(status == OS_SUCCESS);
status = osEventQueueCreate(Q, SIZE);
OS_ASSERT_COND(status == OS_SUCCESS);
```

The following code snippet (Listing 2.16) illustrates an example of a task pointing to a queue, another task pending on it, and an interrupt handler accepting it:

Listing 2.16 Tasks pointing to and pending on a queue

```
status = osEventQueuePend(Q, &message, TIMEOUT);          //Task 1

status = osEventQueuePost(Q, message, &resumed_task);    // Task 2
OS_ASSERT_COND(status == OS_SUCCESS);

status = osEventQueueAccept(Q, &message);           //Interrupt Handler
```

The functions available in the event queues module of the SmartDSP OS are listed in Table 2.10.

Table 2.10 Functions Available for Event Queues

Flow State	Name	Description
Kernel bring up		Already handled by the Events ‘base class’.

Table 2.10 Functions Available for Event Queues (*continued*)

Flow State	Name	Description
Application bring up	`osEventQueueFind()`	Receives a handle to create a queue event.
	`osEventQueueCreate()`	Creates a queue event. Initializes a queue and an empty waiting list. Then, it places a handle to this queue and to the empty waiting list in an event structure.
Application runtime	`osEventQueuePend()`	Pends on an event queue if queue is empty; otherwise, dequeues and continues.
	`osEventQueueAccept()`	Continues without pending. It checks if there is a message available in the queue. If there is a message, it returns the message to the caller.
	`osEventQueuePost()`	Posts a message to a queue. If tasks are pending on the queue, it posts messages on the tasks and schedules them; otherwise, enqueues.
	`osEventQueueReset()`	Removes all tasks pending on the queue and schedules them if they are ready.
Application tear down	Not supported	

2.14 OS Tick Timers

In the devices incorporating DSP subsystems with the SC3400 core and above, such as MSC814x and later, the tick functionality runs on timer zero on each individual DSP subsystem. The timer expiration triggers a hardware interrupt which activates a software interrupt.

The OS provides `#define` constants for standard tick resolutions in `smartdsp_os_device.h`. Refer to *SmartDSP OS API Reference Manual* for details.

2.14.1 Configuration of OS Tick Timers

The tick functionality is defined in `os_config.h` as:

```
#define OS_TICK  ON
#define OS_TICK_PRIORITY  OS_SWI_PRIORITY0
#define OS_TICK_PARAMETER MSC814X_TICK_DEFAULT
```

The `#define` constant for tick period is calculated by the number of interrupts per second. Therefore, for 10[ms] period that would be 10/10^3 or:

```
#define MSC815X_TICK_010MS          INTERRUPTS_PER_SEC_100
#define INTERRUPTS_PER_SEC_100      100
```

For example, if the intended tick period is 7[ms], the number of ticks per second and the `OS_TICK_PARAMETER` will be 7/(10^3) or 142.85. This number will be rounded down to 142 by the compiler since the kernel expects a uint32_t for the parameter.

The DSP subsystem timer is configured by `osTickIntProgram()`:

```
divider_value = g_core_clock*1000000/g_tick_parameter;
```

Since the `g_tick_parameter` will be rounded off, the calculation comes to, 1000*1000000/142 which is equal to 0x6B74CD or 7042253. Having a 1Ghz (ergo the 1000) count to this number will give 7.042253[ms].

NOTE `g_tick_parameter` is the equivalent of `OS_TICK_PARAMETER` and will be rounded down by the compiler (defined as a `uint32_t`).

NOTE This is the current implementation and prone to change. For tick frequency guarantee users must use the `MSC81XX_TICK_XXXMS` macros defined in `smartdsp_os_device.h`.

2.15 Software Timers

The software timers component of the SmartDSP OS kernel supports runtime association of software timer handlers, written in C or a C callable language. Software timers use the hardware tick timer as its point of reference. All software timer handlers run from the OS SWI assigned to the tick timer.

The software interrupt handler checks the software timers, which are scheduled to expire on this tick, and calls their handler. Because software timers run as software interrupts, they are not as precise as hardware timers. Software timer handlers run in an interrupt (that is, system) context with the interrupt stack.

You can enable and disable individual software timers, and set and change timer periods in the OS configuration file. At creation, a timer is specified as a one-shot timer or a periodic timer.

A one-shot timer runs the associated timer function when the timer expires and does not run again. A periodic timer reloads its interval value each time the handler runs, and runs periodically. You can define the maximum number of software timers allowed by the system in the configuration file.

Software timers have the granularity of the tick. A higher tick frequency allows better precision of software timers. A lower tick frequency causes less interrupt overhead on the core.

Software timers require the tick functionality in the system that is defined in `os_config.h`, see 2.14.1 Configuration of OS Tick Timers.

2.15.1 Configuration of Software Timers

The availability of software timers in the system is also configured in `os_config.h` as:

```
#define OS_TOTAL_NUM_OF_SW_TIMERS  5
```

The functions available for the software timers module of the SmartDSP OS are listed in Table 2.11.

Table 2.11 Functions Available for Software Timers

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osTimerInitialize()`	Initializes the software timers module.
Application bring up	`osTimerFind()`	Finds the first available software timer number.
	`osTimerCreate()`	Creates a software timer and sets its parameters, such as timer number, timer mode, timer interval, and handler.
Application runtime	`osTimerStart()` or `osTimerStartDelayed()`	Starts the given software timer.

Table 2.11 Functions Available for Software Timers (*continued*)

Flow State	Name	Description
	`osTimerStop()`	Stops the given software timer.
	`osTimerSetInterval()`	Sets the interval of the given software timer.
	`osTimerSelf()`	Retrieves the number of the currently active software timer.
Application tear down	`osTimerDelete()`	Deletes the given software timer from the system.

The source code in Listing 2.17 dynamically creates a timer using `osTimerFind()`. The new timer expires only once (`OS_TIMER_ONE_SHOT`). The timer expires after five system ticks. When the timer is created, it does not start automatically. The next part of the listing starts the timer using `osTimerStart()`.

Listing 2.17 Dynamically creating a one-shot timer

```
os_timer_handle timer1;
status = osTimerCreate(timer1,                          // timer object
                       OS_TIMER_ONE_SHOT,              // timer mode
                       5,                             // ticks timeout
                       timerTest1);                  // timer handler
if (status != OS_SUCCESS) OS_ASSERT;

// Start the timer
status = osTimerStart(timer1);
if (status != OS_SUCCESS) OS_ASSERT;

void timerTest1()
{
// Do something...
}
```

Listing 2.18 uses the previous timer, changes the tick timeout from 5 to 10, and restarts the timer for a single expiration.

Listing 2.18 Changing the timeout period of the timer

```
// Set the timer's timeout
status = osTimerSetInterval(timer1, 10);
if (status != OS_SUCCESS) OS_ASSERT;
```

```
// Start the timer
status = osTimerStart(timer1);

if (status != OS_SUCCESS) OS_ASSERT;
```

Listing 2.19 dynamically creates a timer using `osTimerFind()`. The new timer expires repetitively because it is a periodic timer. The timer expires every time 20 system ticks elapse. When the timer is created, it does not start automatically. The next part of the listing starts the timer. To stop a timer, use the `osTimerStop()` function.

Listing 2.19 Dynamically creating a periodic timer

```
os_timer_handle timer2;
status = osTimerCreate(timer2,                // timer object
                       OS_TIMER_PERIODIC,    // timer mode
                       20,                   // ticks timeout
                       timerTest2);         // timer handler

if (status != OS_SUCCESS) OS_ASSERT;

// Start the timer
status = osTimerStart(timer2);

if (status != OS_SUCCESS) OS_ASSERT;
void timerTest2(void)
{
// Do something else...
}

// Stop the timer
status = osTimerStop(timer2);
if (status != OS_SUCCESS) OS_ASSERT;
```

Listing 2.20 dynamically creates a timer using `osTimerFind()`. The new timer expires once because it is a one-shot timer. The timer expires after 20 system ticks elapse. When the timer is created it does not start automatically. The timer handler deletes the timer.

Listing 2.20 Dynamically creating a one-shot timer that deletes itself after it expires

```
os_timer_handle timer3

osTimerFind(&timer3);
status = osTimerCreate(timer3,                 // timer object
                       OS_TIMER_ONE_SHOT,     // timer mode
                       20,                    // ticks timeout
                       timerTest3);          // timer handler
```

```
if (status != OS_SUCCESS) OS_ASSERT;

// Start the timer
status = osTimerStart(timer3);
if (status != OS_SUCCESS) OS_ASSERT;

void timerTest3()
{
os_timer_handle self;

// Delete the timer
status = osTimerSelf(&self);
if (status != OS_SUCCESS) OS_ASSERT;

status = osTimerDelete(self);
if (status != OS_SUCCESS) OS_ASSERT;

}
```

2.16 Hardware Timers

The hardware timers are a group of timers running at system (not core) frequency. The hardware timers are shared among all cores and their sharing is handled by the SmartDSP OS drivers. Hardware timers are part of the SoC. The SoC hardware timer module contains four quadrates of four timers each. In devices with four cores, each core owns a quadrate. In devices with six cores, each core owns half a quadrate. This means that there are four hardware timers not enabled by the OS

The hardware timers in the 8144/8156 processors are 16-bit timers which can be clocked from multiple optional input clocks. In 8144/8156 processors, there are four quadrates of clocks that make a total of sixteen 16-bit clocks.

The SmartDSP OS supplies hardware timer implementation for the different architectures, wrapped with a common, minimal API. Hardware timers exist on most underlying hardware and work in a similar way.

The SmartDSP OS API for hardware timers is similar to the software timers API, except that you must specify the clock source for the timer when they are created. Hardware timers have a finer granularity than the OS clock because they do not need to be tied to the OS clock. Hardware timers have the granularity of the driving clock, based on SoC support.

2.16.1 Configuration of Hardware Timers

The availability of hardware timers in the system is configured in `os_config.h`, as:

```
#define OS_HW_TIMERS  ON
```

The functions available for the hardware timers module of the SmartDSP OS are listed in Table 2.12.

Table 2.12 Functions Available for Hardware Timers

Flow State	Name	Description
Kernel bring up (the user application does not call these functions specifically. Instead, `osInitialize()` calls these functions based on the parameters from `os_config.h` and configurations from `msc81xx_config.c`)	`osHWTimerInitialize()`	Initializes the hardware timers module.
Application bring up	`osHWTimerFind()`	Finds the first available hardware timer number.
	`osHWTimerCreate()`	Creates a hardware timer and sets its parameters, such as timer number, timer mode, clock source, timer interval, handler, and priority.
Application runtime	`osHWTimerStart()`	Starts the given hardware timer.
	`osHWTimerStop()`	Stops the given hardware timer.
	`osHWTimerSetInterval()`	Creates a hardware timer and sets its parameters.
	`osHWTimerClearEvent()`	Clears the event bit of the given hardware timer.

Hardware timers can be set as `OS_TIMER_ONE_SHOT`, `OS_TIMER_PERIODIC` or `OS_TIMER_FREE_RUN`. When the timer expires, an application defined handler (set in `osHwTimerCreate`) is called.

NOTE OS does not call `osHwTimerClearEvent` function. The application defined handler must call this function in order to clear the interrupt.

2.17 Debug and Trace Unit (DTU)

This section describes SmartDSP OS support for the Debug and Trace Unit (DTU) driver. The main support goal is to initialize and configure the Profiling Unit (PU) block. The PU is characterized as follows:

- includes a set of configurable counters;
- counters may be configured to count between a large number of subsystem and core input events;
- input events can be used to create an applicaton benchmark.

2.17.1 Features

Main DTU driver features include support for the following:

- DTU monitor mode
- six profiling counters
- 98 counting events

2.17.2 Programming Model

DTU programming steps are as follows:

1. DTU initialization which transfers a selected configuration to the hardware.
2. DTU control operations are supported—read counter results and start and stop profiling.

2.17.3 Example Calling Sequence

Table 2.13 Example Calling Sequence

Function	Description
`osDtuInitProfiler`	Writes a selected configuration to the DTU hardware.
`osDtuStartProfiling`	Starts benchmark counting.
`osDtuStopProfiling`	Stops benchmark counting.
`osDtuReadCount`	Reads a requested counter value.
`osDtuDisableProfiler`	Disables DTU control. Note! It must be called before reconfiguration.

2.17.4 Specific Functionalities

2.17.4.1 Assigning and Enabling Counting Events

Users can select between 98 counting events spread over two triad counters.

1. Select a Counting Events Value (CEV) in order to assign a counting event to the two triad counters; each CEV value defines a group of three events relating to a given triad.
2. Use the API routine osDtuInitProfiler to assign a CEV value to the two triad counters.
3. Users can—following CEV assignment—start or stop counting using the osDtuStartProfiling and osDtuStopProfiling routines, respectively.

2.17.5 Reading a Counted Value

Use the osDtuReadCount routine to read a counted value. The calling core should indicate the specific counted event (from amongst the 98 counted events).

2.17.6 Resetting the DTU

The DTU driver supports reconfiguration of counted events.

1. Use the osDtuDisableProfiler routine to reset the triad counters.
2. Use the osDtuInitProfiler routine to assign different counting events (following triad counter reset).

2.17.7 Demo Use Cases

\SmartDSP\demos\starcore\b4860\advanced_kernel_demo

2.18 B4860 L1-Defense

B4860 L1-Defense enables recovery—without having to restart a device—from software errors that have caused a core to “get stuck” (stray) or enter an irrecoverable state (while still responding to NMI).

SmartDSP OS, which runs on DSP cores, will recover following a warm PPC reset of DSP (single/multiple/all) cores. This recovery process is called L1-Defense.

2.18.1 Functionality

L1-Defense is an integral part of SmartDSP OS and will be rooted in the Kernel.

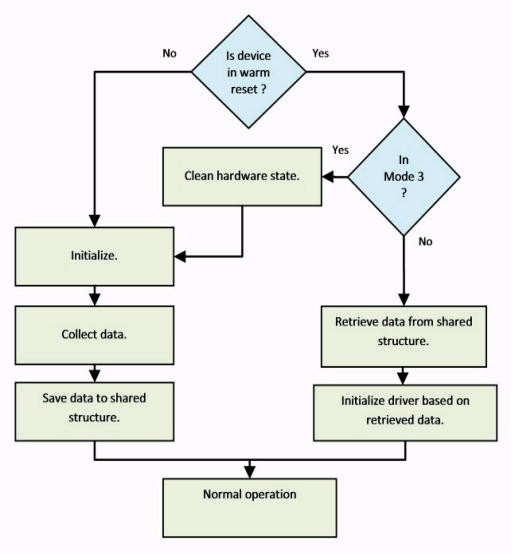
- Required functionality will be added to drivers to enable L1-Defense.
- User *init* parameters, as with every other kernel/driver module, will activate the mechanism.
- All data relevant to L1-Defense will be kept in shared memory in a dedicated L1-Defense data structure.

L1-Defense modules enable support for three L1-Defense operation modes:

Table 2.14 L1-Defense Operation Modes

L1-Defense Mode	Function	Description
1	• Core-level warm reset flow. • For single or multiple cores.	• Local data of specific core(s) is zeroed. • Shared data remains.
2	• L1-level warm reset flow. • For all DSP cores. • No CPRI reset. • Possible MAPLE reset.	• Local data of all DSP cores is zeroed. • Shared data remains.
3	• L1-level warm reset flow. • For all DSP cores. • Resets CPRI and MAPLE.	• All local and shared DSP data is zeroed. • Heterogeneous memory remains.

Workflow for device reset modes (1-3, none):



2.18.2 Application operation after warm reset

This chapter describes the impact of warm reset on a running application.

2.18.2.1 MPIC

For applying L1 Defense support in MPIC message and MSI interrupts application keeps MPIC message and MSI interrupt handlers in a shared location, where they can be picked up and be used after warm reset.

MPIC events are cleared by SmartDSP OS, application needs to resend messages for the reset core to respond.

Already-used MSI interrupts are not returned by `osMpicFindMsiIntr()`.

2.18.2.2 Core timers

L2 cache partitioning and locking remains untouched during warm reset in modes 1 and 2. Application should not relock L2 caches after warm reset (or to clear locking) in that case.

In mode 3 all locking and partitioning are cleared as a part of SmartDSP OS initialization flow.

2.18.2.3 Spinlocks

During warm reset flow, SmartDSP OS releases all spinlocks acquired by the reset core, as a part of the warm reset initialization flow. It is possible that all other cores are kept busy waiting until this point in time.

Application initializes spinlocks using `osSpinLockInitialize()`, to enable the OS to keep tracking of all spinlocks used on the system.

2.18.2.4 OS Barriers

Core is reseted while other cores are waiting for it to reach barrier. In this case, SmartDSP OS releases this barrier.

Application should skip usages of `osWaitForAllCores()` in L1 defense mode 1 application initialization, to avoid core hanging.

2.18.2.5 Queues

Application should not initialize shared queues after warm reset, since other cores may still be working with them.

NOTE In case application is using a shared queue only by the reset core (not recommended) it may have items enqueued which should be dequeued by application init in order to enable working with the queue.

2.18.2.6 Memory allocation (memory manager)

For L1 Defense mode 1 and 2, memory allocated by application should make sure not to reallocate shared memory (unless freeing it before hand).

2.18.2.7 Memory blocks (memory manager)

The memory blocks are allocated by application. Therefore, shared memory blocks are nor reallocated. Application manages allocation and release of all shared buffers and store the core ID for each buffer. This data is used after warm reset to ability reuse or release the memory buffers in case of these buffers are allocated by reset core.

2.18.2.8 CPRI

A dedicated CPRI mechanism for stop-restart exists in the SoC. This mechanism enables to stop the CPRI DMA while keeping the CPRI framer active (hence not impacting CPRI chain) and restart and synchronize IQ data. This mechanism is used for mode1 and mode2 L1 defense scenarios.

When recovering from mode 3 warm restart scenario, CPRI initialization runs the same as any cold reset (CPRI reset and auto negotiation).

When recovering from mode1 and 2 warm reset scenario, SmartDSP OS stops CPRI using this stop-restart mechanism, and then upon user command, `CPRI_DEVICE_DMA_RESTART` it restarts CPRI (both TX and RX). The application should not disable or enable IQ data since it is already enabled from before the reset.

To enable this mechanism by CPRI SmartDSP OS driver, CPRI initialization parameters are enhanced to include necessary stop-restart enablement and information. For more information, see `cpri_iq_init_params_t` definition.

2.18.2.9 MAPLE

During L1 defense mode 3 warm reset, MAPLE is reset and initialized.

During L1 defense mode 2 warm reset, MAPLE is optionally reset and initialized, according to the description in heterogeneous structure.

During L1 defense mode 1 warm reset, MAPLE is never reset.

If MAPLE is not reset, the application reopens the MAPLE devices using the same configuration and claim all MAPLE related resources (COP devices, channels …).

NOTE Jobs that were not reaped in the BD ring before the resets, are discarded during warm reset and are not completed.

2.18.2.10 Local Access Windows (LAW)

In case of an application directly defines Local Access Windows (LAW), it reuses them after warm reset without initialization (LAW is not impacted during the warm reset initialization flow - in all modes).

2.18.2.11 SRIO

SRIO configuration is not impacted by L1 defense warm reset in all scenarios.

All SRIO configurations including ATMU configurations are kept after warm reset, application reuses the already existing configuration after warm reset (ATMU window that was opened prior to warm-reset remains open after warm-reset).

2.18.2.12 Message queues

Application do not re-create the message queues. It is possible for application to continue using the message queues after warm reset without re-initialization.

2.18.2.13 OCN DMA

For L1 defense mode 3, application keeps the same reset flow for OCN DMA.

OCN DMA channels owned by reset core (private and shared) are disabled and closed during

warm reset initialization. In case OCN DMA channel is transferring data, SmartDSP OS driver stops it during warm reset initialization, and unbinds any chain on the channel.

After warm reset application reallocates memory for chains (in case of chaining transfer mode), reopen the closed channels of reset core (cores) and rebind the chains with these channels.

2.18.2.14 Debug print

Debug print driver is using trace buffer to send debug messages. The trace buffer is a global asset in the system used by all cores. In case one core is reset, SmartDSP OS enables the rest of the cores to send debug print messages.

Reset core continues sending messages regularly after warm reset initialization process is completed.

Application must use shared memory for trace buffer defined in driver initialization parameters `b486x_debug_print_init_params_t->vtb_mem_location` to avoid trace buffer from being zeroed.

Watermark interrupts handling is modified to be handled by the first available core (instead of only the master core), to enable debug print to function in case the master core is reset (L1 defense mode1).

2.18.2.15 32bit timers (device level timers)

32 bit timers are a global asset in the system. Each 32-bit device timer can be used by any of the DSP cores within the device as well as by an external host. Core can be the owner of a timer, acquired by `hwTimer32Open()`.

In case a reset core is the owner of 32-bit timers, SmartDSP OS will free and clear the timers during warm reset initialization flow. Application initialization reopens such timers and start using them as in usual reset.

Timers which are not owned by a reset core are not impacted and continues to function as usual.

2.18.2.16 General

Application keeps the normal initialization flow for below models for all L1 defense modes.

IPC, Core timers, Watchdog, MMU, CME, L1 cache, L3 cache (CPC), Tasks, EPIC, Queues, Software interrupts, Hardware abstraction layers (HAL), Debug hooks, Memory frames, Virtual interrupts, and Hardware semaphore.

2.18.3 Configuration of L1-Defense

Hardware timer availability is configured in `os_config.h` as,

```
#define B4860_L1_DEFENSE                    ON
#define OS_L1_DEFENSE                       B4860_L1_DEFENSE
```

B4860 L1-Defense functions are listed below.

Table 2.15 B4860 L1-Defense Functions

Flow State	Name	Description
OS Initialization after warm reset	`osL1dInitialize()`	Initialization method.
	`osL1dHardwareClean()`	Cleans HW registers and reverts them to their original state after reset.
	`osL1dResetFlowSet()`	Checks if reset has occurred and the presence of a functioning reset flow.
Application Bring-up	`getResetCoresId()`	• Gets a mask of warm reset cores. • Mask core IDs: – core0 - `0x0100000000000000` – core1 - `0x0001000000000000` – core2 - `0x0000010000000000` – core3 - `0x0000000100000000` – core4 - `0x0000000001000000` – core5 - `0x0100000000010000`
	`osL1dGetResetMode()`	Initiates a warm reset mode (1,2, or 3).
Application Runtime	`osL1dReportStatus()`	Reports the warm reset status to L2.
	`osL1dResetRequest()`	Prepares a core for warm reset.

2.18.4 Source Code

SmartDSP OS B4860 L1-Defense header file:

- `include\arch\starcore\b4860_family\ b486x_l1_defense.h.`

2.18.5 Demo Use Cases

- demos\starcore\b4860\L1_Defense_integration_demo

3

Hardware Abstraction Layers (HAL)

Hardware Abstraction Layers (HAL) allow an application to stream Input/Output (I/O) data to a hardware device using the device's Low Level Driver (LLD). In SmartDSP OS, a HAL is also referred to as an Abstraction Module.

This chapter contains the following topics:

3.1 HAL in SmartDSP OS

This section explains the 3.1.1 Conceptual Model and 3.1.2 Conceptual Workflow of HAL in the SmartDSP OS.

3.1.1 Conceptual Model

HAL in the SmartDSP OS consists of two layers: Serializer and LLD.

Serializer:

- Interacts with the user application to stream I/O data to a hardware device.
- SmartDSP OS implements different serializer modules for various types of I/O data and hardware devices.
- Each serializer module provides a high-level API to the user application; the API is used to interact with a corresponding LLD/hardware device.
- Table 3.1 lists supported serializer modules and their functionality in the SmartDSP OS.

LLD:

- Interacts with the hardware device to perform I/O operations.

- Uses LLD API to communicate with a serializer.

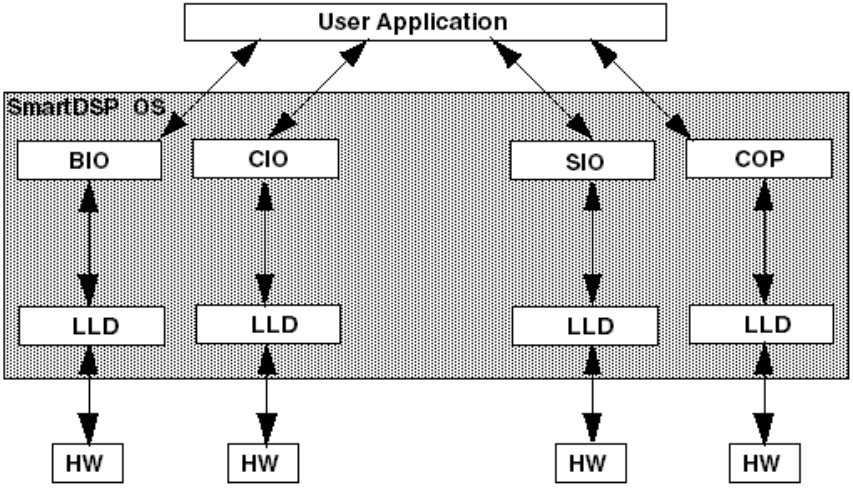
.

Table 3.1 Serializer Modules in SmartDSP OS

Serializer Modules	Functionality
Buffered I/O (BIO)	Supports hardware devices that transmit and receive data using data packets; e.g., Ethernet.
Coprocessor (COP)	Supports co-processors; e.g., MAPLE.
Synchronized I/O (SIO)	Supports hardware devices whose transmit and receive processes are hardware-timed; e.g., TDM.
Character I/O (CIO)	Supports character and stream oriented devices; e.g., UART.

Figure 3.1 shows the conceptual model of HAL in the SmartDSP OS. The user application calls the hardware-independent high-level API functions, and the LLD implements the device specific functionality.

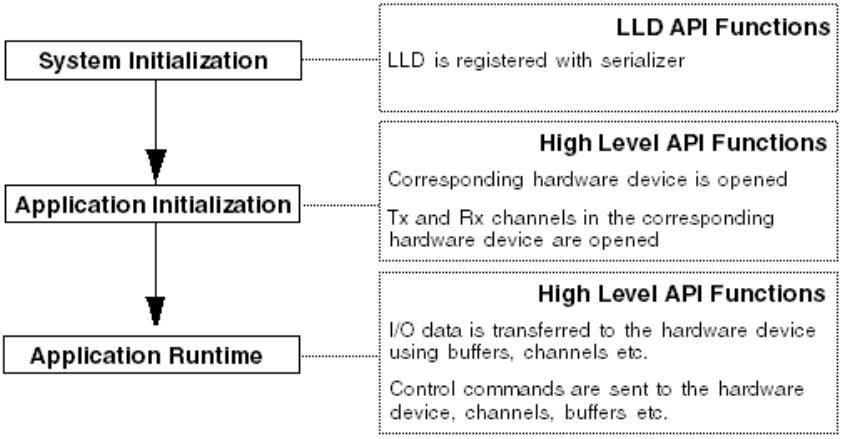
Figure 3.1 Conceptual Model of HAL in SmartDSP OS



3.1.2 Conceptual Workflow

Figure 3.2 shows a conceptual workflow sending/receiving data to/from a hardware device. All I/O operations use channels. A channel is an abstract data path between the LLD and the hardware device. Duplex communication requires two channels (transmit and receive).

Figure 3.2 Conceptual Workflow of HAL in SmartDSP OS



3.2 Buffered I/O (BIO) Module

The BIO module provides high-level API and LLD for streaming I/O data to hardware devices that transmit/receive data using data packets; e.g., Ethernet. The BIO module also serializes transmitted/ received data packets.

The BIO module uses a frame data structure to transfer data between the user application, the BIO serializer, and the BIO LLD. The user application and LLD should use the same data structure.SmartDSP OS API provides a SmartDSP OS frame data structure. Frames are also managed with user-defined data structures.

This section contains the following subjects:

3.2.1 BIO Layers

The BIO module consists of the upper BIO Serializer and lower BIO LLD layers.

BIO Serializer:

- Interacts with the user application using a high-level API.
- Performs I/O data packet serialization.
- Defines callback functions used to pass control back to the user application.

BIO LLD:

- Interacts with the BIO hardware device to perform I/O operations.
- Must support the BIO LLD API in the SmartDSP OS.
- See the *SmartDSP OS API Reference Manual* for more information on the BIO LLD API.

3.2.2 BIO Initialization Workflow

This section examines the BIO module workflow for 3.2.2.1 Smart DSP OS Kernel System and 3.2.2.2 User Applications initializations.

3.2.2.1 Smart DSP OS Kernel System

System initialization includes the following stages:

1. SmartDSP OS kernel initializes the LLD of all BIO devices. Each LLD is initialized only once, even in a multi-core environment.
2. LLD initialization function initializes the corresponding hardware device.
3. LLD initialization function registers the LLD with the BIO serializer.
   - Registration information contains function pointers.
   - During application runtime, the BIO serializer uses function pointers to translate high-level API functions to corresponding LLD functions.
4. An `os_config.h` file definition sets the number of BIO devices supported by the system:

   ```
   #define OS_TOTAL_NUM_OF_BIO_DEVICES \
   MSC815X_UEC0 + MSC815X_UEC1 + MSC815X_RIONET0 + \
   MSC815X_RIONET1)
   ```

5. Sets the maximum number of channels supported by a BIO device.

3.2.2.2 User Applications

User application initialization includes the following stages:

1. Calls high-level API function, `osBioDeviceOpen`, to open BIO device.
2. Calls high-level API function, `osBioChannelOpen`, to open BIO device Tx/Rx channel.
3. Allocate memory required for BIO channels—create a static array of the `bio_channel_t` structure and specify the array size as the number of required channels.

NOTE `bio_channel_t` defines the structure of a BIO channel. See *SmartDSP OS API Reference Manual* for more information.

4. Sets the number of queues.The BIO serializer uses queues to hold Tx and Rx frames.
   - For each BIO Tx channel:
     - BIO serializer uses a Tx confirmation queue to hold frames being transmitted to the LLD.
     - If the BIO serializer fails to transmit a frame to the LLD then it notifies the user application; it does not put a corresponding frame into the confirmation queue.
   - For each BIO Rx channel:
     - BIO serializer uses one queue to hold received frames and another to hold empty data buffers.
     - LLD must request empty data buffers from the BIO serializer before trying to receive new data from the BIO device. LLD gets empty buffers before identifying an Rx channel.
     - LLD can choose a common buffer pool for all Rx channels.SmartDSP OS memory manager maintains a buffer pool consisting of unordered blocks of memory.
     - LLD chooses one of two options after data is received into empty buffers. If there are errors in the received data (e.g., CRC) then the frame is discarded. The serializer does not build a new frame but can report the error using the application's Rx callback function.
       – [Recommended] BIO serializer builds a new frame structure—taking buffers from the buffer queue—according to buffer length and puts the new frame into the received frames queue.
       – Self-built frames are used to receive extended frame structures containing additional (non data-buffer) information. LLD passes the pointer of a built frame to the BIO serializer. The serializer inserts the frame into the received frames queue.

Use the method outlined in Table 3.2 to determine the number of queues required by the BIO serializer.

Table 3.2 Determining the Number of Required Queues

Operating Mode	Number of Required Queues
BIO serializer builds a frame without using a common buffer pool.	BIO_dev_queues = num_of_tx_channels + (2*num_of_rx_channels)

Table 3.2 Determining the Number of Required Queues (*continued*)

Operating Mode	Number of Required Queues
BIO serializer builds a frame using a common buffer pool.	BIO_dev_queues = num_of_tx_channels + 1 + num_of_rx_channels_that_use_the_common_pool + (2*num_of_rx_channels_that_use_another_pool)
LLD builds a frame using any type of buffer pool.	BIO_dev_queues = num_of_tx_channels + num_of_rx_channels

3.2.3 BIO Runtime Workflow

This section examines the BIO module workflow during application 3.2.3.1 Transmit and 3.2.3.2 Receive runtimes.

3.2.3.1 Transmit

BIO Tx channel must be in `write` mode before a user application can transmit any data to a BIO device. The Tx process consists of the following steps:

1. User application employs a high-level API function, `osBioChannelTx`. Pointers are specified for both the Tx channel and the to-be-transmitted frame.
2. BIO serializer calls the LLD transmit function and specifies which frame to transmit. In addition, the serializer queues the frame in the Tx confirmation queue.
3. BIO LLD transmits the frame to the BIO device.
4. BIO LLD calls the LLD API function, `bioChannelTxCb`. Serializer is notified of the number of frames for transmission.
5. BIO serializer de-queues specified frames from the Tx confirmation queue.
6. BIO serializer calls the user application Tx callback function for each transmitted frame.
   - If the user application does not specify a Tx callback function then the serializer releases all Tx frames back to the frame pool.

3.2.3.2 Receive

BIO Rx channel must be in `read` mode before a user application can receive any data from a BIO device. The Rx process consists of the following steps:

1. BIO LLD requests an empty buffer from the BIO serializer; it is for a specific Rx channel using the LLD API function, `bioChannelRxBufferGet`. BIO LLD can also request an empty buffer from a common buffer pool.
2. BIO serializer allocates a new buffer to the LLD.
3. BIO LLD replaces an old buffer with a new one if the buffers use Buffer Descriptors (BD). Empty buffer receives the data.
4. BIO LLD notifies the BIO serializer using the LLD API function, `bioChannelRxCb`.
   - BIO LLD specifies a pointer for the Rx channel and Rx frame length.
   - BIO LLD can self-build the frame and then use LLD API function `bioChannelRxFrameCb` to notify the serializer.
5. BIO serializer puts the received frame into the Rx frame queue.
6. BIO serializer calls the user application Rx callback function for each frame put into the Rx frame queue.
7. User application de-queues the frames from the Rx frame queue.

3.3 Coprocessor (COP) Module

NOTE The COP module was introduced in MSC814x; it is not available for earlier devices.

A COP module performs job serialization and provides high-level API and LLD for dispatching jobs to a COP hardware device for execution. The SmartDSP OS currently supports SEC and MAPLE COP devices.

This section contains the following subjects:

3.3.1 COP Layers

The COP module consists of the upper COP serializer and the lower COP LLD layers.

COP Serializer:

- Interacts with the user application using a high-level API.
- Performs job serialization before dispatching to the COP device.
- Ensures job results are returned from the COP LLD to the user application as per dispatch order.

COP LLD:

- Interacts with the COP device to dispatch jobs—generally jobs dispatched to a COP device begin and end within the device.
- COP LLD for each COP device must support the COP LLD API in the SmartDSP OS.
- See the *SmartDSP OS API Reference Manual* for more information on the COP LLD API.

3.3.2 COP Initialization Workflow

This section examines the COP module workflow for 3.2.2.1 Smart DSP OS Kernel System and 3.3.2.2 User Application initializations.

3.3.2.1 Smart DSP OS Kernel System

System initialization includes the following stages:

1. COP device is registered with the COP serializer using the LLD API function, `copRegister`.
   - Registration information contains function pointers. The COP serializer uses function pointers—during application runtime—to translate high-level API functions to corresponding LLD functions.
2. An `os_config.h` file definition sets the number of COP devices supported by the system:

```
#define OS_TOTAL_NUM_OF_COP_DEVICES \
MSC815X_SEC + MAPLE + MAPLE_TVPE + MAPLE_FFTPE + \
MAPLE_DFTPE + MAPLE_CRCPE)
```

3. Sets the maximum number of channels supported by a COP device.

3.3.2.2 User Application

User application initialization includes the following stages:

1. Calls high-level API function, `osCopDeviceOpen`, to open COP device.
2. Calls high-level API function, `osCopChannelOpen`, to open COP channel.
3. Allocate memory for COP channels—create a static array of the `cop_channel_t` structure and specify the array size as the number of required channels.

NOTE `cop_channel_t` structure defines the structure of a COP channel. See *SmartDSP OS API Reference Manual* for more information.

4. Use the `os_config.h` file definition, `OS_TOTAL_NUM_OF_QUEUES`, to set the total number of queues per channel.

3.3.3 COP Runtime Workflow

This section examines the COP module 3.3.3.1 Transmit workflow during application runtime. The COP module does not have an Rx path.

3.3.3.1 Transmit

COP Tx channel must be in `write` mode before a user application can transmit jobs to a COP device. The Tx process consists of the following steps:

1. User application employs a high-level API function, `osCopChannelDispatch`. Pointers are specified for both a valid COP channel and an array of job handles.
2. COP LLD is assigned the Tx request.
3. COP LLD calls the COP LLD API function, `copChannelDispatchCb`. Serializer is notified that dispatched jobs have completed execution.
4. COP serializer calls the user application callback function—if the latter was defined.

3.4 Synchronized I/O (SIO) Module

The SIO module provides high-level API and the LLD for streaming I/O data to hardware devices whose Tx and Rx processes are hardware-timed; e.g. TDM. The SIO module also serializes transmitted/ received data buffers.

SIO usage is recommended for hardware devices that use a cyclic buffer--one that can be divided into sub-buffers used to transmit and receive data during specific time slots. This approach imposes hard real-time constraints on a user application. Examples:

- Under-run error: user application does not free a buffer until it is time for it to be reused.
- Unused time slot: user application does not provide any Tx data during a specified time slot.

This section contains the following subjects:

3.4.1 SIO Layers

The SIO module consists of the upper SIO Serializer and lower SIO LLD layers. SIO Serializer:

- Interacts with the user application using a high-level API.
- Performs data buffer serialization.

SIO LLD

- Interacts with the SIO hardware device to perform I/O operations.
- SIO LLD, for each SIO hardware device, must support the SIO LLD API in the SmartDSP OS.
- See the *SmartDSP OS API Reference Manual* for more information on the SIO LLD API.

3.4.2 SIO Initialization Workflow

This section examines the SIO module workflow for 3.4.2.1 Smart DSP OS Kernel System and 3.4.2.2 User Application initializations.

3.4.2.1 Smart DSP OS Kernel System

System initialization includes the following stages:

1. SmartDSP OS kernel initializes the LLD of all SIO devices. Each LLD is initialized only once, even in a multi-core environment.
2. LLD initialization function initializes the corresponding hardware device.
3. LLD initialization function registers the LLD with the SIO serializer.
   - Registration information contains function pointers.
   - During application runtime, the SIO serializer uses function pointers to translate high-level API functions to corresponding LLD functions.
4. An `os_config.h` file definition sets the number of SIO devices supported by the system:

   ```
   #define OS_TOTAL_NUM_OF_SIO_DEVICES \
   MSC815X_TDM0 + MSC815X_TDM1 + MSC815X_TDM2 + \
   MSC815X_TDM4)
   ```

5. Sets the maximum number of channels supported by a SIO device.

3.4.2.2 User Application

User application initialization includes the following stages:

1. Calls high-level API function, `osSioDeviceOpen`, to open SIO device.
2. Calls high-level API function, `osSioChannelOpen`, to open SIO device Tx/Rx channel.
   - When the SIO serializer opens a channel, the serializer submits a buffer array to LLD memory and a reference to a buffer array index.
   - LLD maintains a current buffer index and returns this reference to the user application.
   - LLD serializer maintains an index of buffers currently held by the user application.
   - Comparing the LLD index to the user application updates the serializer to the following: under-run occurrences and if a new buffer is ready to be passed to the user application.
3. Allocate memory required for SIO channels—create a static array of the `sio_channel_t` structure and specify array size as the number of required channels.

NOTE `sio_channel_t` structure defines the structure of a SIO channel. See *SmartDSP OS API Reference Manual* for more information.

3.4.3 SIO Runtime Workflow

This section examines the SIO module workflow during application 3.4.3.1 Transmit and Receive runtimes.

3.4.3.1 Transmit and Receive

SIO channel must be in `write/read` mode before a user application can Tx/Rx data to an SIO device. The Tx/Rx process consists of the following steps:

1. User application employs a high-level API function, `osSioBufferGet`.
2. User application receives a buffer and buffer length from the Tx or Rx channels.

NOTE User application can use the buffer as long as it is not requested by the LLD. If requested—while in use by the user application—then the serializer calls the application's under-run callback and the buffer is seized.

3. User application actions reflects Tx or Rx channel functioning.
   - Tx channel: data fills the buffer and serializer increments its index.
   - Rx channel: reads the buffer and LLD increments its index.

4. User application uses high-level API function, `osSioBufferPut`, to return the buffer to the LLD.

3.5 Character I/O (CIO) Module

The CIO module provides high-level API and LLD for streaming I/O data to hardware devices that transmit/receive data as a stream of characters/bytes; e.g., UART. The CIO module also serializes transmitted/received character streams.

This section consists of the following topics:

3.5.1 CIO Layers

The CIO module consists of the upper CIO Serializer and lower CIO LLD layers.

CIO Serializer:

- Interacts with the user application using a high-level API.
- Performs serialization the I/O character stream.

CIO LLD:

- Interacts with the CIO hardware device to perform I/O operations.
- Must support the CIO LLD API in the SmartDSP OS.
- See the *SmartDSP OS API Reference Manual* for more information on the CIO LLD API.

3.5.2 CIO Initialization Workflow

This section examines the CIO module workflow for 3.5.2.1 Smart DSP OS Kernel System and 3.5.2.2 User Application initializations.

3.5.2.1 Smart DSP OS Kernel System

System initialization include the following stages:

1. SmartDSP OS kernel initializes the LLD of all CIO devices. Each LLD is initialized only once, even in a multi-core environment.
2. LLD initialization function initializes the corresponding hardware device.
3. LLD initialization function registers the LLD with the CIO serializer.
   - Registration information contains function pointers.
   - During application runtime, the CIO serializer uses function pointers to translate high-level API functions to corresponding LLD functions.
4. An `os_config.h` file definition sets the number of CIO devices supported by the system:

```
#define OS_TOTAL_NUM_OF_CIO_DEVICES \
    (MSC815X_DOORBELL + MSC815X_UART + MSC815X_I2C + \
    MSC815X_SPI)
```

5. Sets the maximum number of channels supported by a CIO device.

3.5.2.2 User Application

User application initialization includes the following stages:

1. Calls high-level API function, `osCioDeviceOpen`, to open CIO device.
2. Calls high-level API function, `osCioChannelOpen`, to open CIO device Tx/Rx channel.
   - When the CIO serializer opens a channel, the user application sends a character array to the serializer.
   - Serializer assigns the character array to both Tx and Rx CIO channels.
3. Allocate memory required for CIO channels—create a static array of the `cio_channel_t` structure and specify the array size as the number of required channels.

NOTE `cio_channel_t` structure defines the structure of a CIO channel. See *SmartDSP OS API Reference Manual* for more information.

3.5.3 CIO Runtime Workflow

This section examines the CIO module workflow during application 3.5.3.1 Transmit and 3.5.3.2 Receive runtimes. The CIO serializer uses two dynamic pointers to Tx and Rx a character stream:

- A pointer points to the current Tx/Rx data.

- A second pointer points to the next free location to which new data will be transmitted/received.

3.5.3.1 Transmit

CIO Tx channel must be in `write` mode before a user application can transmit a character stream to a CIO device. The Tx process consists of the following steps:

1. User application employs a high-level API function, `osCioChannelBufferGet`, to get a pointer to a free buffer.
2. User application fills the buffer with data and uses a high-level API function, `osCioChannelTxBufferPut`, to notify the serializer that data is ready for transmission.
3. CIO serializer calls the CIO LLD Tx function.
4. CIO LLD transmits data to the CIO device.
5. CIO LLD calls the LLD API function, `cioChannelTxCb`, and notifies the serializer of the number of transmitted bytes.
6. CIO serializer calls the user application callback function—if the latter was defined—for each transmitted data buffer.

NOTE Serialization process can lead to corrupt data unless all `osCioChannelBufferGet` calls have a corresponding `osCioChannelTxBufferPut` call.

3.5.3.2 Receive

CIO Rx channel must be in `read` mode before a user application can receive a character stream from a CIO device. The Rx process consists of the following steps:

1. CIO LLD uses a high-level API function, `osCioChannelBufferGet`, to get a pointer to a free buffer.
2. CIO device sends an interrupt to the CIO LLD. CIO LLD fills the buffer with received data.
3. CIO LLD calls the LLD API function, `cioChannelRxCb`, to notify the serializer of the received data.
4. CIO serializer calls the application's Rx callback function—if the latter was defined—for each received data buffer.
5. User application uses a high-level API function, `osCioChannelRxBufferGet`, to get a pointer to the received data.
6. User application uses a high-level API function, `osCioChannelRxBufferFree`, to notify the serializer that data processing is complete.

NOTE Serialization process can lead to corrupt data unless all `osCioChannelRxBufferGet` calls have a corresponding `osCioChannelRxBufferFree` call.

4

Drivers

A driver program allows SW, which utilizes the OS, to interact with a peripheral device. The SmartDSP OS has the following capabilities:

- Supports a unified, cross-device API for generic HW drivers;
- Provides a device-specific API for device-specific HW; and,
- Supports MSC814x, MSC815x, and MSC825x processor driver families.

The five SmartDSP OS driver types are noted in the below table.

Table 4.1 SmartDSP Drivers

Driver	Description	Example
DMA	Provides a unified API.	DMA controllers
BIO	Provides a unified API for all frame type drivers.	ETH and RapidIO® messages
COP	Provides a unified API for all co-processors.	SEC and MAPLE-B
SIO	Provides a unified API for devices with HW-timed Rx/Tx processes.	TDM
CIO	Provides a unified API for non frame-based devices (that lack a logical division of the data into frames or packets).	UART and I²C

This chapter has sections detailing the following SmartDSP OS drivers:

4.1 Direct Memory Access (System DMA)

A System DMA is a HW controller; it off loads, from the core, the task of moving data from one location to another.

SmartDSP OS provides a unified, cross-platform API for the System DMA driver and, where necessary, a platform-specific API.

4.1.1 Features

SmartDSP OS System DMA LLD features are noted below.

- Full exposure of all HW features.
- Functions inlined for performance gain.
- Generic and SoC-specific APIs.
- Ability to pre-program BD rings (chains).
- Ability to modify existing BDs (transfer/buffer).
- Interrupt/polling at job completion.
- Assignment of channels-to-cores at compilation time.

4.1.2 Architecture

This section covers the SW architecture of the System DMA driver.

4.1.2.1 Driver Components

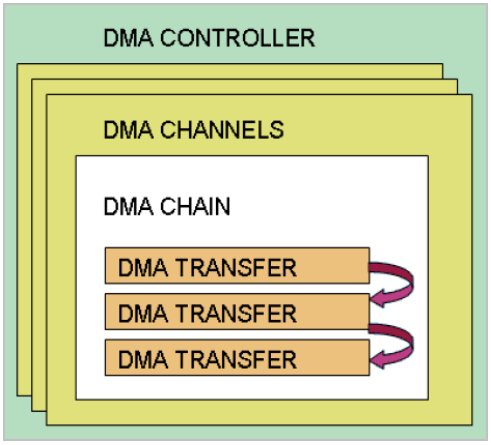
This section details System DMA driver components found in the SmartDSP OS. See Table 4.2 and Figure 4.1 for further details.

Table 4.2 System DMA Components

Components	Description
Controller	• Represents the actual HW block. • HW block acts as follows: – generates access to the system bus; – manages data transfers between memory and HW devices; and – manages multiple channels. • Each channel can be programmed to move data from one place to another, independent of other channels.
Channel	• Represents the actual HW channel for data transfers. • Executes chains of DMA transfers. • Handles interrupts generated by DMA transfers. • Pre-allocated to a given core at compilation time.
Chain	• Represents a BD ring[1]. • BD ring assembles DMA transfers in a BD chain. • Can be inactive. • Executable by any channel.
Transfer/Buffer	Represents a BD; each BD describes DMA transaction attributes.
	Transfer • Symmetric 1D (classic) DMA transaction. • Includes all related R/W transaction attributes.
	Buffer • Representation of a specific HW DMA BD implementation. • Describes R/W transaction attributes of 1D or more. • Can include other HW-specific features. • See API.

1. A set of buffer descriptors that are sequentially executed by the HW; e.g., BD0 then BD1, BD2, and so on until the final BD and then again to BD0.

Figure 4.1 System DMA Module Structure



4.1.2.2 Design Decisions

Design decisions, along with their reasoning, are noted below.

1. The System DMA driver does not support cycle consuming cache and virtual addressing operations in the LLD.
   - Driver was written to minimize runtime overhead.
   - Driver API enables runtime modifications of pre-programmed BDs.
2. Many System DMA driver functions are statically inlined.
   - Minimizes the number of function calls as well as driver overhead.
3. Assigning channels-to-cores occurs at compilation time.
   - Minimizes the driver's program memory footprint.

4.1.3 Data Flow

This section outlines the runtime data flow.

1. Application pre-configures one (or more) BD chains.
2. Application attaches one of the chains to a channel and activates the channel.
3. Channel chronologically iterates the BDs—in the order they were attached to the chain—and executes them.

4. Every application-modifiable BD attribute is defined; they cannot be changed during execution.
5. Channel completes execution.
6. Application reactivates a given chain, swaps it for another, or becomes idle.

In System DMA implementations, as configured by the application, the channel can be frozen during chain execution. Later, the application can unfreeze the channel.

4.1.4 Programming Model

This section covers driver bring-up—first by the OS, then by the application-chosen API.

4.1.4.1 General

The System DMA consists of two API families.

- Generic DMA concept—API is prefixed with osDma.
- Device-specific family—API is prefixed by the device name; e g., msc815xDma.

See the SmartDSP OS API Reference Manual for a complete list of DMA LLD API.

Table 4.3 describes both architecture-independent and dependent APIs.

Table 4.3 Architecture-Independent/Dependant APIs

Architecture API	Functions	Description
Architecture-independent API (see Table 4.4)	Functions include: • osDmaChannelOpen() • osDmaChainCreate() • osDmaChannelBind()	Uses functions regardless of the HW architecture; e.g., Works.
Architecture-dependant API[1]	Functions include: • msc814xDmaChannel Defrost() • msc815xDmaChannel Disable()	• Uses functions when the HW cannot be used for all architectures; e.g., Common. • Header file: include\arch\starcore\ msc81xx\msc81xx_d ma.h

1.MSC81xx can be either MSC814x or MSC815x.

4.1.4.2 Calling Sequence Example

Table 4.4 describes functions included in the architecture-independent API of the SmartDSP OS DMA driver. The functions include kernel bring-up, application bring-up, application runtime, and application teardown.

Table 4.4 Architecture-Independent API

State	Function	Description
Kernel Bring-up	`osInitialize()->` `osArchDevicesInitialize() ->` `osDmaInitialize()`	• User application does not directly call these functions. • `osInitialize()` calls `osDmaInitialize()` – as based on os_config.h defines and msc81xx_config.c configuration structures. • Initializes System DMA driver.

Table 4.4 Architecture-Independent API

State	Function	Description
Application Bring-up	`osDmaControllerOpen()`	• Must be first function called. • Returns a DMA controller handle. • Handle is a parameter in calling other functions.
	`osDmaChannelOpen()`	• Opens a pre-allocated channel. • Initializes channel structure as per configuration parameters.
	`osDmaChainCreate()`	• Creates a DMA chain. • Initializes chain structure as per configuration parameters.
	`osDmaChainTransferAdd()`	Adds a transfer to a DMA chain.
	`osDmaChainTransferAddEx()`	• Adds a transfer to a DMA chain. • Returns a handle to the transfer. • Handle can be used to modify transfer attributes.
Application Runtime	`osDmaChannelBind()`	Binds a chain to a channel.
	`osDmaChannelStart()`	Starts execution of a bound chain(s) on a DMA channel.
	`osDmaChannelIsActive()`	Polls the DMA channel to check its state (active/idle).
Application Teardown	`osDmaChainDelete()`	• Releases DMA chain resources (except memory). • Removes a chain from the DMA controller.
	`osDmaChannelClose()`	Releases DMA channel resources (except memory).

To populate a chain, the application can use the architecture-independent transfer concept, architecture-dependent buffer concept, or a combination of both.

The buffer allows users to program each side (R/W) of the chain as an individual entity; this allows for a mix of HW-enabled features such as BD dimension, freeze dimension, etc. The application is responsible for balancing the number of DMA channel R/W bytes.

Table 4.5 Architecture-Dependent API

State	Function	Description
Bring-up	`msc815xDmaChainBufferAdd()`	• Appends a buffer to a DMA chain.
	`msc815xDmaChainBufferAddEx()`	• Appends a buffer and returns a handle to a DMA chain.
Runtime	`msc815xDmaChannelFreeze()`	• Freezes a DMA channel.
	`msc815xDmaChannelDefrost()`	• Unfreezes a DMA channel.
	`msc815xDmaChannelDisable()`	• Disables a DMA channel.

NOTE Common MSC814x and MSC815x architecture-specific features:
- share a similar API
- prefixed by the device name.

4.1.4.3 Functionality

4.1.4.3.1 Initialization

Follow these steps to initialize the DMA:

1. Enable DMA support by setting `#define MSC81XX_DMA ON in` the application `os_config.h` file.
2. Allocate memory; this enables the driver to handle a DMA channel.
   - User application determines memory size:

   ```
   /* Allocate memory for channel use. */
   uint8_t dma_channel[DMA_SIZE_OF_MEMORY_FOR_CHANNEL_USE];
   ```
3. Allocate memory for a DMA chain handle. Allocation reflects both the DMA chain type and the maximum number of associated DMA transfers.

- User application determines DMA chain, memory size:
  `DMA_SIZE_OF_MEMORY_FOR_CHAIN_USE(<NUM_BUFFERS>)`
- Above code example allocates memory for a default chain type:

```
/* Allocate memory for chain with two transfers. */
uint8_t dma_chain_memory[DMA_SIZE_OF_MEMORY_FOR_CHAIN_USE(2)];
```

4. Get DMA controller handle using `osDmaControllerOpen()`.
   - Function returns the handle to the driver-allocated DMA controller.
   - User application must retrieve this handle to open DMA channels and create DMA chains.
5. Open controller DMA channel using `osDmaChannelOpen()`.
6. Create a DMA chain using `osDmaChainCreate()`.
   - Chain is created using allocated memory, a parameter structure, and a DMA controller handle.
7. Add DMA transfers to the DMA chain using `osDmaChainTransferAdd()` or `osDmaChainTransferAddEx()`.
   - `osDmaChainTransferAddEx()` returns a handle to added transfers; this allows for attribute modification.

4.1.4.3.2 Runtime

Follow these steps to run DMA in the application.

1. Reset or empty the DMA chain using `osDmaChainReset(dma_chain)`.
2. Program the DMA channel with the DMA chain using

```
status = osDmaChannelBind(dma_channel, dma_chain);
```

NOTE DMA channel and chain must be unbound when executing function,
`status = osDmaChannelBind(dma_channel, dma_chain);`

If, however, an application reprograms the DMA channel with the same DMA chain then these conditions must be met:
– DMA chain parameter must be `NULL`.
– DMA channel must be bound using the following:
`status = osDmaChannelBind(dma_channel, NULL);`

3. Start the DMA channel, after it is bound to a chain, using `osDmaChannelStart()`.
4. [Optional] Poll a DMA channel using `osDmaChannelIsActive()` to check if the channel is active.
   - After completion of the DMA run and chain, the driver disables the interrupt.
   - Driver can be enabled at `osDmaChannelOpen()`.

- If the interrupt was enabled, the driver calls the interrupt handler with interrupt parameters specified in `OsDmaChannelOpen()`.

4.1.4.3.3 Free

Follow these steps to enable DMA Free.

1. Ensure DMA channel is inactive.
2. Unbind DMA channel and chain using the function, `osDmaChannelUnbind()`.
   - Indicates DMA chain can be deleted or bound to a different DMA channel.
   - Indicates DMA channel can be closed or bound to a different DMA chain.
3. Release DMA channel using `osDmaChannelClose()`.
   - Releases all DMA channel resources (except memory) that belong to a user application.
4. Release DMA chain using `osDmaChainDelete()`.
   - Releases all DMA channel resources (except memory) that belong to a user application.

4.1.4.3.4 Performance Rules

Performance Rules guidelines—for user applications—improve MSC814x and MSC815x System DMA performance.

- Place BD close to Port A (one of two ports).
  - Port A is always used for BD fetch.
  - Rule is primarily applicable to MSC8144.
- Minimize WAR penalties on the SoC CLASS fabric.
  - Application should configure the System DMA to read from a port close to the data source and write to the other port.
  - Rule is primarily applicable to MSC8144.
- Consider target-switching when programming channels to access various targets.
- Set BTSZ to the 64-byte maximum as part of the BD attribute.
- Set TSZ to ‘high’ as per application requirements.

4.1.4.4 Source Code

SmartDSP OS DMA driver header files:

- `include\common\os_dma.h`.
- `include\arch\starcore\msc815x\msc815x_dma.h`

- `initialization\arch\msc815x\include\msc815x_dma_init.h`
- `include\arch\starcore\msc814x\msc814x_dma.h`
- `initialization\arch\msc814x\include\msc814x_dma_init.h`

4.1.5 Resource Management

Minimize a driver's data footprint by assigning channels-to-cores at compilation time.

- Driver configures the master core (`osGetMasterCore()`) to handle error interrupts during `osDmaInitialize`() execution.
- Driver handles HW-related errors.
- User callback functions[1] handle functional errors.

4.1.6 Demo Use Cases

- demos\starcore\msc814x\dma_multicore
- demos\starcore\msc815x\dma_demo.

4.2 OCeaN DMA

OCeaN DMA, also known as OCN DMA and On-chip Network DMA, is used to generate RapidIO and PCIe (MSC815x only) buses.

4.2.1 Features

SmartDSP OS OCeaN DMA features are noted below.

- OCeaN DMA is used to generate DMA accesses in the OCeaN fabric.
  - OCeaN fabric is a non-blocking, high-speed interconnect used for embedded system devices.
- Able to generate transactions towards HSSI peripherals (sRIO and PCIe).
- Accesses OCeaN ports as per address ranges configured in its ATMU windows.
- Generates accesses in a 36-bit address space.
  - Relies on the target OCean ATMU mechanism to resolve the actual address.
- Address space limitations:
  - System (CLASS): 32 bits. System interface shortens four MSB; it does not resolve addresses.

1. Optional parameters that the application passes to the System DMA driver during `osDmaInitialize()`.

  - RapidIO: 34 bits. sRIO ATMU converts a 36-bit OCeaN address to a 34-bit sRIO address.
  - PCIe (MSC815x only). PCIe ATMU can convert a 36-bit OCeaN address to either a 32- or 64-bit PCIe address.
- MSC814x supports a 34-bit ATMU bypass address space.
  - Makes write access part of the transfer description.
  - MSC815x has no HW support for ATMU address bypassing.

4.2.1.1 Relevant SoC

MSC815x and B4860; the latter only supports DDR transactions.

4.2.2 Architecture

This section covers the SW architecture of the OCeaN DMA driver. The OCeaN fabric requires no programming and provides a seamless interface for the HSSI. See 4.6 OCean DMA Architecture4.6 OCean DMA ArchitectureTable 4.6.

Table 4.6 OCean DMA Architecture

Processor	Description
MSC815x	• Two OCeaN DMA instantiation. • Supports sRIO and PCIe.
B4860	• Two OCeaN DMA instantiations. • Supports only DDR transactions.

NOTE System DMA cannot be used to create transactions over I/O ports.

4.2.2.1 Design Decisions

Design decisions, along with their reasoning, are noted below.

OCeaN and System DMA channels share two design decisions.

1. OCeaN DMA driver does not support cycle consuming cache and virtual addressing operations in the LLD.
   - Driver was written to minimize runtime overhead.

- Driver API enables runtime modifications of pre-programmed BDs.

2. Assigning channels-to-cores occurs at compilation time.
   - Minimizes the driver's program memory footprint.

4.2.2.2 Calling Sequence Example

Table 4.7 describes functions included in the SmartDSP OS OCeaN driver. The functions include kernel bring-up, application bring-up, application runtime, and application teardown.

Table 4.7 OCeaN DMA API

State	Function	Description
`os_config.h` file	`#define MSC815X_OCN_DMA0 ON` `#define MSC815X_OCN_DMA1 ON`	• [MSC815x] OCeaN DMA support
	`#define OCN_DMA0 ON` `#define OCN_DMA1 ON`	• [B4860] OCeaN DMA support

Kernel Bring-up	`osInitialize()->`
	`osArchDevicesInitialize()->`
	`ocnDmaInitialize()`
		• User application does not directly call these functions.
		• `osInitialize()` calls the noted functions based on the following:
		– `os_config.h` defines
		– `msc81xx_config.c` (MSC815x) configuration structure.
		– `b4860_config.c` (B4860) configuration structure.
		• Initializes the OCeaN DMA driver.

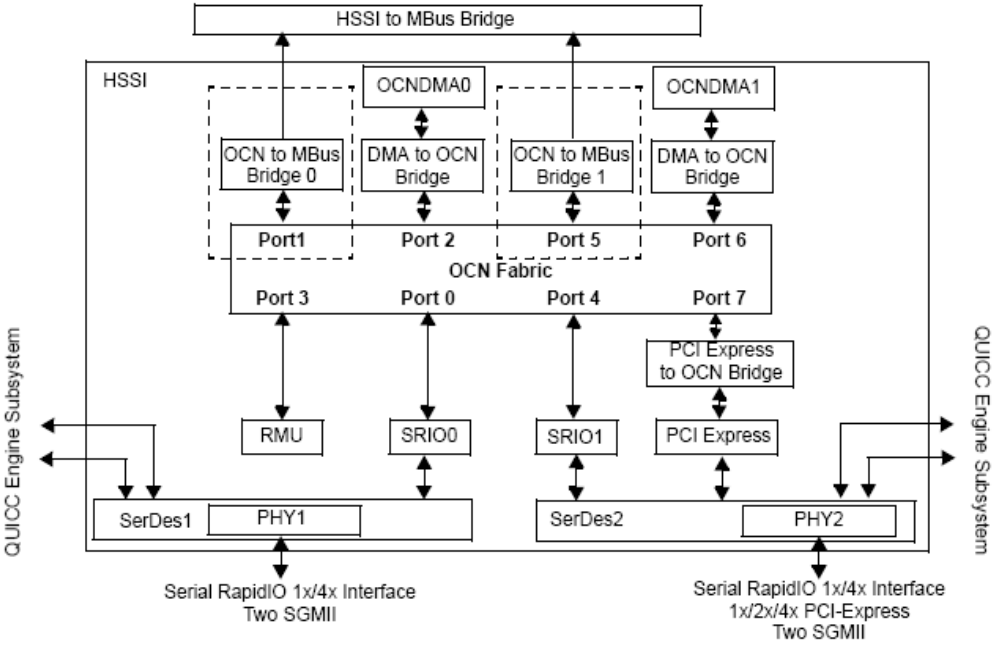
Table 4.7 OCeaN DMA API (*continued*)

State	Function	Description
Application Bring-up	`ocnDmaControllerOpen()`	• Must be first function called. • Returns an OCeaN DMA controller handle; handle is a parameter in calling other functions.
	`ocnDmaChannelOpen()`	• Opens a pre-allocated channel. • Initializes channel structure as per configuration parameters.
	`ocnDmaChainCreate()`	• Creates an OCeaN DMA chain. • Initializes chain structure as per configuration parameters.
	`ocnDmaChainTransferAdd()`	• Adds a transfer to an OCeaN DMA chain.
Application Runtime	`ocnDmaChannelBind()`	• Binds a chain to a channel.
	`ocnDmaChannelStart()`	• Starts execution of a bound chain(s) on an OCeaN DMA channel.
	`ocnDmaChannelIsActive()`	• Polls the OCeaN DMA channel to check its state (active/idle).
	`ocnDmaTransferWait()`	• Waits for the release of a specific OCeaN DMA channel.
Application Teardown	`ocnDmaChainDelete()`	• Releases OCeaN DMA chain resources (except memory). • Removes a chain from the OCeaN DMA controller.
	`ocnDmaChannelClose()`	• Releases OCeaN DMA channel resources (except memory).

4.2.2.3 Functionality

Figure 4.2 shows an MSC8156 HSSI block diagram and illustrates the role of OCeaN DMA.

Figure 4.2 MSC815x HSSI Block Diagram



4.2.2.4 Source Code

Below is a list of MSC815x and B4860 SmartDSP OS OCeaN driver header files that relate to initialization and runtime:

- initialization\arch\msc815x\include\msc815x_ocn_dma_init.h
- include\arch\starcore\msc815x\msc815x_ocn_dma.h
- initialization\arch\peripherals\ocn_dma\include\ocn_dma_init.h
- drivers\ocn_dma\include\ocn_dma_.h
- include\arch\peripherals\ocn_dma\ocn_dma.h

4.2.3 Demo Use Cases

- demos\starcore\msc815x\rio_dma
- demos\starcore\b4860 \ocn_dma

4.3 Queue Manager (QMAN)

4.3.1 Introduction

This chapter describes the architecture and design of the Queue Manager driver. It also describes the high-level application programing interface (API).

4.3.2 Functionality

The Queue Manager (QMan) SoC block manages the movement of data ("frames") along uni-directional flows ("frame queues") between different software and hardware end-points ("portals"). This allows software instances to communicate with other software instances and/or datapath hardware blocks using a hardware-managed queueing mechanism. QMan provides a variety of features to manage this data movement, including tail-drop or weighted-red congestion/flow-control, congestion group depletion notification, order restoration, and order preservation.

It is beyond the scope of this document to fully explain all the QMan-related notions that are essential for using datapath functionality effectively. This chapter covers basic elements essential for the software interface. For more information about what QMan does and how it behaves, consult the"B4860RMDPAAAP_RevD".

4.3.2.1 Frame Descriptors

Frames are represented by "frame descriptors" (or "FD"s) which are 16-byte structures consisting of fields which describe:

- contiguous or scatter-gather data,
- a 32-bit per-frame-descriptor token value (called "cmd/status" because of its common usage in processing data to/from hardware blocks),
- trace-debugging bits,
- a partition ID, used for virtualizing memory access to frame data by datapath hardware blocks (CAAM, PME, FMan),
- A QMan buffer pool ID, used to identify frames whose buffers are sourced from (or are to be recycled to) a QMan buffer pool.

A third ("nested") mode of the scatter-gather representation allows a frame-descriptor to reference more than one frame-this is referred to as a compound frame, and is a mechanism for creating an indissociable binding of more than one data descriptor, eg. this is used when sending an input descriptor to PME or CAAM and providing an output descriptor to go with it.

Frame descriptors that are under QMan's control reside in QMan-private resources, comprised of dedicated on-board cache as well as system memory assigned to QMan on initialization. When frames are enqueued to (and dequeued from) frame queues by QMan on behalf of software portals or hardware blocks, the frame descriptor fields are copied in to (and out of) these QMan-private resources.

As with QMan not caring whether the 48-bit tokens it manages are real buffer addresses or not, the same is mostly true for QMan with respect to the frame descriptors it manages. QMan ignores the memory addresses present in the frame descriptor, unless it is dequeued via a portal configured for data stashing and is dequeued from a frame queue that is configured for frame data (or annotation) stashing.However QMan always pays attention to the length field of frame descriptors. In general, the only field that can be safely used as a "pass-through" value without any QMan consequences is the 32-bit cmd/status field.

4.3.2.2 Frame Queue Descriptors

Frame queues are uni-directional queues of frames, where frames are enqueued to the tail of the frame queue and dequeued from the head. A frame queue is represented in QMan by a "frame queue descriptor" (or "FQD"), and these reside in a private system memory resource configured for QMan on initialization.

A frame queue is referred to by a "frame queue identifier" (or "FQID"), which is the index of that FQD within QMan's memory resource. As such, FQIDs form a global name-space, even in an otherwise virtualized environment, so two entities of software cannot simultaneously use the same FQID for different purposes.

4.3.2.3 Work Queues

Work queues or "WQ"s are uni-directional queues of "scheduled" frame queues. QMan supports a fixed collection of work queues, to which QMan appends frame queues when they are due to be serviced i.e. multiple FDs can be linked to a single FQ, and multiple FQs can be linked to a single WQ.

4.3.2.4 Channels

A channel is a fixed, hardware-defined association of 8 work queues, also thought of as "priority work queues". The QMan provides sophisticated prioritization support for dequeueing from entire channels rather than specific work queues. Specifically, the 8 work queues within a channel are divided into 3 tiers according to QMan's "class scheduler" logic:

1. work queues 0 and 1 form the high-priority tier and are treated with a strict priority semantic,
2. work queues 2, 3, and 4 form the medium-priority tier and are treated with a weighted interleaved

3. round-robin semantic, and work queues 5, 6, and 7 form the low-priority tier and are also treated with a weighted interleaved round-robin semantic.

Apart from the top-tier, the weighting within and between the other two tiers is programmable.

4.3.2.5 Portals

The QMan has hardware and software portals. The hardware portals (also called "direct connect portals", or "DCP"s) allow QMan to be used by other hardware blocks, and there are software portals allow QMan to be used by logically separated units of software. A software portal consists of two sub-regions of QMan's CoreNet region, in precisely the same way as with QMan.

4.3.2.6 Dedicated Portal Channels

Each software portal has its own dedicated channel of 8 work queues that it can dequeue from. A frame queue is "scheduled to a portal" i.e. the frame queue is scheduled to a work queue within that portal's dedicated channel. Hardware portals have one or more dedicated channels.

4.3.2.7 Portal Sub-Interfaces

Each portal exposes cache-inhibited and cache-enabled registers that can be read and/or written by software to achieve various ends. With some necessary exceptions, the software interface hides most of these details. The portals have four decoupled sub-interfaces:

- EQCR (EnQueueCommand Ring) is an 8-cacheline ring containing commands from software to QMan. These commands perform enqueues of frame descriptors to frame queues.
- DQRR (DeQueue Response Ring) is a 16-cacheline ring containing dequeue processing results from QMan to software. These entries usually contain a frame descriptor (except when the dequeue action produced no valid frame descriptor) as well as status information about the dequeue action, the frame queue being dequeued from, and other context for software's use. This ring is unique in that QMan can be configured to stash new ring entries to processor cache, rather than relying on software to (pre)fetch ring entries into cache explicitly.
- MR (Message Ring) is an 8-cacheline ring containing messages from QMan to software, most notably for enqueue rejection messages and asynchronous retirement processing events. Unlike DQRR, this ring does not support stashing.
- Management commands, consists of a Command Register (CR) and two Response Register locations (RR0 and RR1), used for issuing a variety of other commands toQMan. EQCR and DQRR (and to a lesser extent, MR) are intended to provide the communications with QMan that represent the fast-path of data processing logic, and the management command interface is where "everything else happens".

4.3.2.8 Frame Queue Dequeuing

Enqueuing a frame to a frame queue is an unambiguous mechanism; an enqueue command in the EQCR specifies a frame descriptor and a frame queue ID, and the intention is clear. Dequeuing is more subtle, and falls into two general classes depending on what one is dequeuing from-these are "scheduled" or "unscheduled" dequeues.

4.3.2.8.1 Unscheduled Dequeues

One can dequeue from a specific frame queue, but that frame queue must necessarily be "idle"-or in QMan terminology, "unscheduled". It is an illegal action to attempt to dequeue directly from a frame queue that is in a "scheduled" state. Specifically, unscheduled dequeues require the frame queue to be in the "Parked" or "Retired" state.

4.3.2.8.2 Scheduled Dequeues

If a frame queue is "scheduled" then the management of the frame queue is (until further notice) under QMan's control and may change state according to events within QMan or via actions on other software or hardware portals. A "scheduled dequeue" does not target a specific FQ, but targets a specific WQ or a collection of channels. QMan processes scheduled dequeue commands within a portal by selecting from the non-empty WQs. It dequeues a FQ from selected WQ and then dequeue a FD from that FQ. QMan portals implement two dequeue command modes: "push" and "pull."

4.3.2.8.3 Push Mode

The "push" mode provides a familiar "DMA-style" interface to software, i.e. where hardware performs work and fills in a kind of "Rx ring" autonomously. In the case of the QMan portal's DQRR sub-interface, the push mode is driven by two dequeue command registers, one for scheduled dequeues (SDQCR-Static DeQueue Command Register), and other for unscheduled dequeues (VDQCR-Volatile DeQueue Command Register). The reason for the static/volatile terminology (rather than scheduled/unscheduled), as well as the presence of two command registers instead of one, relates to how QMan schedules execution of the dequeue commands.

Unlike "pull" mode, QMan is not prodded by a write to the command register each time a dequeue command should occur, it must autonomously execute commands when appropriate. So it is clear that scheduled dequeues can only be performed when the targeted work queue or channels have truly scheduled frame queues available to dequeue from. Note that this is not an issue with "pull" mode, as a scheduled dequeue command can be issued when there are no available frame queues and QMan will simply publish a DQRR entry containing no frame descriptor to mark completion of the command-for "push" mode, this semantic cannot work. When in "push" mode, the QMan portal has a (possibly NULL) scheduled dequeue command for dequeuing from a selection of available channels. QMan executes this command only when there is matching scheduled dequeue work available on one of the channels, that is,

the scheduled dequeue command (for channels) is static. If software writes SDQCR with a command to dequeue from a specific WQ, the command is executed only once (like the pull command), at which point it reverts to the static dequeue command for channels.

For unscheduled dequeues, a single Parked or Retired frame queue is identified for dequeuing, and as QMan does not manipulate the state of such frame queues in reaction to enqueue or dequeue activity (that is, there is no "scheduling"), there is no mechanism for QMan to "know" when this frame queue becomes non-empty sometime in the future. So like "pull" mode, unscheduled dequeues must be done when explicitly demanded by software, and as such they must also (a) expire after a configurable number of frame descriptors are dequeued from frame queue or once it is empty, and (b) even if the frame queue is already empty, a DQRR entry with no frame descriptor should be used to notify software that the unscheduled dequeue command has expired. I.e. the unscheduled command "goes live" when written and becomes inactive once completed-it is volatile. Unlike "pull" mode however, the volatile command can perform more than a single dequeue action, and it can even block or flow-control while active, however it always runs to completion and then stops.

As "push" mode supports two dequeue commands, i.e. it can service both at once. The VDQCR command register contains a precedence option that QMan uses to determine whether SDQCR or VDQCR work be favored in the situation where both are active.

4.3.2.8.4 Stashing to Processor Cache

When dequeueing frame queues and publish entries in DQRR, QMan provides stashing features that involve prepositioning data in processor cache. The main benefit of hardware-instigated stashing is that the data will already be in cache when the processor needs it, avoiding the need to explicitly prefetch it in advance or stalling the processor to fetch it on-demand. As we will see, there is another benefit in the specific case of DQRR stashing. Each portal supports two types of stashing, for which distinct PAMU entries are configured.

4.3.2.8.4.1 DLIODN

The DLIODN setting configures PAMU authorization and/or translation of transactions to stash DQRR ring entries as they are produced by QMan. The stashing of DQRR entries is not just a performance tweak, it changes the way driver software operates the portal. Rather than needing to invalidate and prefetch the DQRR cache lines to see (or poll for) new DQRR entries, software can simply reread the cached version until it "magically changes". The stashing transaction is then the only implied traffic across the CoreNet bus (reducing bandwidth) and it is initiated by hardware at the first instant at which a software-initiated prefetch could have seen anything new (minimum possible latency).

Note that if the driver does not enable DQRR stashing, then it is a requirement to manipulate the processor cache directly, so its run time mode of operation must match device configuration. Note also that if DQRR stashing is used, software cannot trust the DQRI interrupt source nor read PI index registers to determine that a new DQRR entry is available, as they may race against the stash transaction. On the

other hand, software may use the interrupt source to avoid polling for DQRR production unnecessarily, but it does not guarantee that the first read would show the new DQRR entry.

4.3.2.8.4.2 FLIODN

QMan can also stash per-frame-descriptor information, specifically:

- Frame data, pointed to by the frame descriptor,
- Frame annotations, which is anything prior to the data due to a non-zero offset, and
- Frame queue context (for the frame queue from which the frame descriptor was dequeued).

In all cases, the FLIODN setting is used by PAMU to authorize/translate these stashing transactions.

4.3.2.9 Frame Queue States

Frame queues are managed by QMan via state-transitions, and some of these states are of interest to software. From software's perspective, a simplification of the frame queue states is to group them as follows:

- Out of service: the frame queue is not in use and must be initialized. Neither enqueues nor dequeues are permitted.
- Parked: the frame queue is initialized and in an idle state. Enqueues are permitted, as are unscheduled dequeues, neither of which change the frame queue's state. Scheduled dequeues will not result in dequeues from parked frame queues, as a parked frame queue is never linked to a work queue.
- Scheduled: the frame queue has been scheduled, implying that hardware will modify its state as/when relevant events occur. Enqueues are permitted, but unscheduled dequeus are not. This is not a real state, but actually a set of states that a frame queue moves between-as hardware performs these moves internally, it's useful to treat them as one, because changes between them are asynchronous to software. The real states are;
  - Tentatively Scheduled: the frame queue isn't linked to a work queue (yet), the frame queue must therefore be empty and no retirement or force-eligible command has been issued against the frame queue.
  - Truly Scheduled: the frame queue is linked to a work queue, either because it has become non-empty or a force-eligible command has occurred.
  - Active: the frame queue has been selected by a portal for scheduled dequeue and so is removed from the work queue.
  - Held Active: the frame queue is still held by the portal after scheduled dequeuing has been performed, it may yet be dequeued from again, depending on scheduling configuration, priorities, etc.

  - Held Suspended: the frame queue is still held by the portal after scheduled dequeuing has been performed but another frame queue has been selected "active" and so no further dequeuing will occur on this frame queue.

- Retired: the frame queue is being "closed". A frame queue can be put into the retired state as a means of (a) getting it back under software's control (neither under QMan's control nor the control of another hardware block), e.g. for closing down "Tx" frame queues, and (b) blocking further enqueues to the frame queue so that it can be drained to empty in a deterministic manner. Enqueues are therefore not permitted in this state. Unscheduled dequeues are permitted, and are the only way to dequeue frames from a frame queue in this state.

4.3.2.10 Hold Active

The QMan portal sub-interfaces are generally decoupled or asynchronous in their operation-for example, the processing of software-produced enqueue commands in EQCR is asynchronous to the processing of dequeue commands into DQRR, and both of these are asynchronous to the production of messages into MR and the processing of management commands.

There is however a specific coupling mechanism between EQCR and DQRR to address a certain class of requirements for datapath processing. Consider first that it is possible for multiple portals to dequeue independently from the same data source, e.g. for the purposes of load-balancing, or perhaps idle-time processing of low-priority work. This could occur because multiple portals issue unscheduled dequeue commands from the same Parked (or Retired) frame queue, or because they issue scheduled dequeue commands that target the same pool channels (or the same specific workqueue within a pool channel). So we describe here the "hold active" mechanisms that help maintain some synchronicity of hardware dequeue processing (and optionally software post-processing) on multiple portals/CPUs.

The unscheduled dequeue case is not covered by the mechanisms described here-QMan will correctly handle multiple unscheduled dequeues from the same frame queue, but the "hold active" mechanisms have no effect in this case. For scheduled dequeues however, there are two levels of "hold active" functionality that can be used for software to synchronize multiple portals dequeuing from the same source.

4.3.2.10.1 Dequeue Atomicity

As described in the previous section ("Frame queue states"), the Active, Held Active, and Held Suspended states are for frame queues that have been selected by a portal for scheduled dequeuing. These states imply that the frame queue has been detached from the workqueue that it was previously "scheduled" to, but not yet moved to the Parked state nor rescheduled to the Tentatively Scheduled or Truly Scheduled state after the completion of dequeuing.

Normally, a frame queue is rescheduled by QMan as soon as it is done dequeuing, potentially even before the resulting DQRR entries are visible to software. However, if the frame queue has been configured for "Held active" behavior, then this will not happen-the frame queue will remain in the Held Active or Held Suspended state once QMan has finished dequeuing from it. QMan will only reschedule or park the frame

queue once software consumes all DQRR entries that correspond to that frame queue-the default behavior is to reschedule, but this "held" state of the frame queue allows software an opportunity to request that the final action for the frame queue be to park it instead.

A consequence of this mechanism is that if a DQRR entry is seen that corresponds to a frame queue configured for "held active" behaviour, software implicitly knows that there can be no other (unconsumed) DQRR entry on any other portal for that same frame queue. (Proof: if there was, the frame queue would be currently "held" in that portal and not in this one.) For an SMP system where each core has its own portal, this would obviate the need to (spin)lock software context related to a frame queue when handling incoming frames-the "lock" is implicitly obtained when the DQRR entry is seen, and it is implicitly released when the DQRR entries are consumed. This is what is meant by "dequeue atomicity".

4.3.2.10.2 Parking Scheduled FQs

If a FQ is currently "held active" in the portal, software can request that it be move to the Parked state once its final DQRR entry is consumed, rather than rescheduled which is the normal behavior. This is not necessarily limited to FQs that are configured for "hold active" behavior, but can also be applied to regular FQs by issuing a Force Eligible command on them.

4.3.2.10.3 Order Preservation & Discrete Consumption Acknowledgement

In addition to the dequeue atomicity feature, it is possible to obtain a stronger property from QMan to aid with datapath situations that "spread" incoming data overmultiple portals. Specifically, if incoming frames are to be forwarded via subsequent enqueues, then dequeue atomicity does not prevent the forwarded frames from getting out of order. I.e. multiple CPUs (using multiple portals) may be using dequeue atomicity in order to write enqueue commands to their EQCR rings before consuming the DQRR entries, and thus ensuring that EQCR entries are published in the same order as the incoming frames. But as there are multiple portals, this does not ensure that QMan will necessarily process those EQCR entries in the same order. Indeed if the portals' EQCR rings have significantly varied fill-levels, then there is a reasonable chance that two enqueue commands published in quick succession via different portals could get processed in the opposite order by QMan.

Instead, software can elect to only consume DQRR entries when no forwarding is to be performed on the corresponding frames (eg. when dropping a packet), and for the others, it can encode the EQCR enqueue commands to perform an implicit "Discrete Consumption Acknowledgement" (or "DCA")-the result of which is that QMan will consume the corresponding DQRR entry on software's behalf once it has finished processing the enqueue command. This provides a cross-portal, order preservation semantic from end-to-end (from dequeue to enqueue) using hardware assists. Note, QMan has other functionality called

Order Restoration that is completely unrelated to the above-Order Restoration is a mechanism to restore frames into their intended order once they been allowed to get out of order, using sequence numbers and

"reassembly windows" within QMan. The above "hold active" mechanisms are to prevent frames from getting out of order in the first place.

4.3.2.11 Force Eligible

QMan portals support a management command called "Force Eligible" which allows software to regain control of a scheduled frame queue, usually with the intention to park it. When a frame queue is scheduled, QMan is responsible for its state and software cannot meaningfully query it, as any snapshots are implicitly out of date by the time software sees them. Software only knows the frame queue state once QMan-generated events indicate that the frame queue is "quiesced" somehow. Moreover, if the frame queue is not configured for "hold active" behavior, then even the presence of DQRR entries does not help in this regard, as the portal may well have rescheduled the frame queue before software sees the first DQRR entry.

When QMan processes a Force Eligible command, it does two things-it tags the frame queue descriptor with a flag that is visible in subsequent DQRR entries, and, if the frame queue is Tentatively Scheduled (because it is empty), it will move the frame queue to the Truly Scheduled state (linked to a work queue).

The result is that the frame queue "will receive dequeue processing soon", whether that was already happening or not. Fundamentally, when QMan is dequeuing from the FQ a short while later, it will treat the frame queue as "hold active", even if it isn't configured for hold active treatment. As such, software can request that the FQ be parked rather than rescheduled once the DQRR entry is consumed.

4.3.2.11.1 Enqueue Rejections

Enqueues may be rejected, immediately or after any delay due to order restoration, and the enqueue mechanisms themselves do not provide any meaningful way to convey the rejection event to the software portal. For this reason, Enqueue Rejection Notifications (ERNs) are messages received on a message ring that carry frames that did not successfully enqueue together with the reason for their rejection.

4.3.2.11.2 Order Restoration

Frame Queue Descriptors can serve one or both of two complimentary purposes. Small subsets of fields in the FQDs are used to implement an "Order Restoration Point", which allows an FQD to act as a reassembly window for out-of-sequence enqueues. FQDs also contain a sequence number field that generates increasing sequence numbers for all frames dequeued from the FQ. This dequeue activity sequence number is also called an "Order Definition Point".

The idea is that frames dequeued from a given FQ (ODP) may get out-of-sequence during processing before they're enqueued onto an egress FQ, so the enqueue function allows one to not only specify the destination FQD, but also an ORP that the enqueue command should first pass through-which might hold up the intended enqueue until other, missing, sequence elements are enqueued. I.e. an ORP-enabled enqueue command requires two FQID parameters, which need not necessarily be the same-indeed in

many networking examples, the Rx FQ serves as both the ODP and the ORP when enqueuing to the Tx FQ. To see why this choice of ORP FQ makes sense, consider that many Rx flows may need to be order-restored independently, even if all of them are ultimately enqueued to the same destination Tx FQ.

It's also possible to enqueue using software-generated sequence numbers, i.e. without any FQ dequeue activity acting as an ODP. An ODP is any source of sequence numbers starting at zero and wrapping to zero at 0x3fff (214-1). ORP-enabled enqueue functions provide various features, such as filling in missing sequence numbers (e.g. when dropping frames), advancing the "Next Expected Sequence Number" despite missing frames (that may or may not show up later), etc.

There are also numerous options that can be set in ORP-enabled FQDs, and these are achieved via the same functions that allow you to manipulate FQDs for any other purpose.

4.3.3 Functionality

The driver provides:

- Simple initialization and configuration API for the following QMan blocks:
  - common registers
  - QMan Portal registers
  - QMan Frame Queue Ranges (FQR)
- IRQ handling.
- Congestion Groups (CG) support

4.3.4 Driver Architecture

4.3.4.1 Driver Components

The QMan driver contains three basic modules, as shown in Figure 15-1.

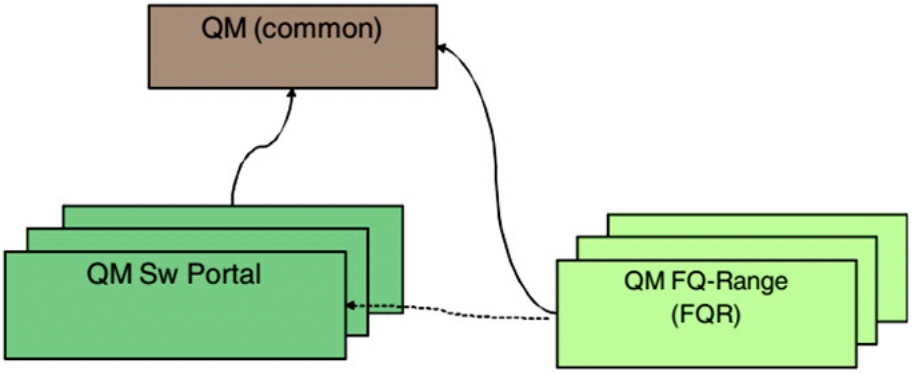


Figure 4.3 QMan driver Modules (from a partition point of view)

The modules are as follows:

- Queue Manager (common)- It is responsible for the common hardware registers initialization, and runtime (for example, management of the QMan pool ids). This module must always be initialized when working with any QMan module. The module will mainly be used internally by the other QMan modules except for its initialization by the user. This module has an instance for each partition. However, only the driver that is on the master-partition has access to the hardware registers.
- Queue Manager Portal (QM-Portal)- This module is responsible for all QMan-portal related register space. This module should be initialized once per QMan-portal. Note that the QMan-portal may be either affine to core or affine to user application (that is, application may run on single-core or several). QMan-portal should be affine to partition (that is, it cannot be shared among partitions.
- Buffer manager Pool (QM-Pool)-This module is responsible of initializing and controlling a certain QM-pool. User may create several instances per QMan-pool; In this case, the first module will be initialized in the standard way while all other instances of this QMan-pool will be initialized in "Shadow Mode". Note that software stockpiles are maintained by the driver per QMan-pool instance. So, user may not put locks when accessing this pool unless the QMan-portal that is being used to access this pool is being shared by several cores or applications.

4.3.4.2 Congestion Groups Support

The QM Congestion Groups (CG's) driver is designed to support the HW CG functionalities and to facilitate their activation and manipulation. The CG is a module within the QM driver, but on top of defining a CG module, in order to use a CG, a number of other driver modules also need to participate.

Congestion Groups Module Functionality

- CG Creation/Deletion-The driver defines a set of parameters to be configured by the user. According to these parameters the driver creates a congestion group and returns its handle to the caller.
- CG Modification-The driver supports WRED curve modification at runtime.
- CG Change Notification-The driver automatically enables change notification exceptions if the user's definitions imply it. It also supports the option to notify the hardware (in a direct-portal case) of such a change. In addition, the driver supplies API for disabling/enabling this interrupt (Congestion-Group-Change-Notification).

QM Module Driver Congestion Groups Support

When initializing the QM, the user must define the base and the number of CG's they'll use for the initialized partition.

QM Portal Module Driver Congestion Groups Support

When initializing the QM, the user must define a callback for returning rejected frames. In fact, this The callback routine (of type qm_received_frame_callback_t) passes, among other parameters, a structure of rejection information which is currently defined:

QM FQR Module Driver Congestion Groups Support

When creating a FQR, it may have the CG attribute assigned to it and be linked to a previously created CG.

The FQR parameter 'congestion_avoidance_enable' indicates the use of CG, and requires a set of CG parameters.

4.3.4.2.1 Flow

In order to use the CG mechanism, the following sequence should be followed:

- On QM initialization, define the number of CG's will be used and there base index (from the available 0-255 CG id's).
- When a QM portal is created, if rejected frames are to be forwarded to SW, define a rejected frame callback routine.
- Create one or more congestion groups by defining their behavior.
- When creating Frame Queue Ranges (FQR's), user may link a certain FQR to a pre-created CG.

4.3.5 Programming Model

4.3.5.1 General

Initialization of the QMan driver is carried out by the application according to the following sequence:

- The QMan common functionality initialization uses the following steps:

  - Calling the configuration routine with basic parameters
  - Calling the advance initialization routines to change driver defaults
  - Calling the initialization routine
- For each QMan-portal required by the user, initialize the QM-Portal module according to the following steps:
  - Calling the configuration routine with basic parameters
  - Calling the advance initialization routines to change driver defaults
  - Calling the initialization routine
- For each QMan-CG required by the user, initialize the QM-CG module with the following steps:
  - Calling the creation routine with all required parameters
- For each QMan-FQR required by the user, initialize the QM-FQR module with the following steps:
  - Calling the creation routine with all required parameters

4.3.5.2 Example Calling Sequence

Table 4.8 Example Calling Sequence

Function to Call	Description
qmConfig	Configures QM using user's basic parameters and default values
qmConfigXxx	Optional-configure selected QM parameters to be other then default
qmInit	Initializes the QM by applying configured parameters
qmPortalConfig	Configures QM Portal using user's basic parameters and default values
qmPortalConfigXxx	Optional-configure selected QM Portal parameters to be other then default
qmPortalInit	Initializes the QM Portal by applying configured parameters
qmFqrCreate	Configures QM FQR using user's parameters
qmFqrFree	Frees QM FQR resources and SW structures
qmPortalFree	Frees QM Portal resources and SW structures
qmFree	Frees QM resources and SW structures

4.3.5.3 Source code

SmartDSP OS Qman B4860 driver header files:

- initialization\arch\peripherals\dpaa\qman\include\qman_init.h
- include\arch\peripherals\dpaa\qman\qman.h
- include\arch\peripherals\dpaa\dpaa.h

4.3.6 Demo Use Cases

- demos\starcore\b4860\dpaa_demo
- demos\starcore\b4860\dpaa_integration

4.4 Buffer Manager (BMAN)

4.4.1 Introduction

This chapter describes the architecture and design of the Buffer Manager (BM) driver. It describes the functionality and the high level Application Programming Interface (API). The BMan device exposes two interfaces to software control. One interface is the Configuration and Control Status Register map (CCSR), which provides global configuration of the device, registers related to global device errors, performance, statistics, debugging, etc. The other interface is the CoreNet interface that provides a memory map with multiple "portals" located in separable sub-regions for independent/parallel run-time use of the devices.

4.4.2 Functionality

The QorIQ Buffer Manager (BMan) SoC block manages pools of buffers for use by software and hardware in the "Data path" architecture. BMan maintains state for 64 "Buffer Pools" which are typically used by hardware blocks for constructing output data for returning to software, where software cannot (or does not wish to) pre-allocate an output descriptor. In particular:

- This provides an efficient use of buffer resources because the output will only occupy as many buffers as required (whereas pre-allocation must provide for the worst-case scenario each time if it wishes to avoid truncation and information-loss).
- Software does not need to provide resources for every queued operation nor handle the complications of recycling unused output buffers, etc.
- The footprint for buffer resources for a variety of different flows (and even different guest operating systems) can be "pooled".

With respect to "buffers", BMan acts as an allocator of any 48-bit tokens the user wishes-BMan does not interpret these tokens at all, it is only the software and hardware blocks that use BMan that may assume these to be memory addresses. In many cases, the BMan acquired and released interfaces are likely to be more efficient than software-managed allocators due to the proximity of BMan's corenet-based interfaces to each CPU and its on-board caching and pre-fetching of pool data. Possible examples include a BMan-oriented page-allocator for operating system memory management, a "frame queue" allocator to manage unused QMan frame queue descriptors (FQD), etc. In particular, the frame queue example provides a simple mechanism for sharing a range of frame-queue IDs across different partitions/operating systems in a virtualized environment without needing inter-partition communications in software.

The driver provides:

- Initialization and configuration API for the following BMan blocks:
  - Common registers
  - BMan Portal registers
  - BMan Pool
- IRQ handling
- Software stockpile per BMan-Pool instance

4.4.3 BMan's Interfaces

The BMan block includes CCSR register space. In addition an interrupt line associated with the block for global configuration and management is used. Specifically, BMan includes:

- Private system memory range (invisible to software) needed by BMan
- CCSR control registers
- Cache-enabled and cache-inhibited control memory region
- Software and hardware depletion interrupt thresholds for each pool
- Device error handling uses the global interrupt line and the CCSR register space contains error-capture and error-status registers

The BMan block also exposes a Corenet memory space for low-latency interaction by the multiple SoC cores, and this Corenet region is divided into a geometry of "portals" to allow independent access to BMan functionality in a partitioned (and/or virtualized) environment. Each portal consists of one 16 KB cache-enabled and one 4 KB cache-inhibited sub-range of the Corenet region, as well as a per-portal interrupt line. There are a variety of possible reasons for using distinct portals;

- Partitioning between distinct guest operating systems
- Dedicating a portal for each CPU to reduce locking and improve cache-affinity
- Making distinct portal configurations available
- Giving certain applications their own portal rather than enforcing a mux/demux layer to share a portal between applications [etc.]

Each portal presents the following BMan functionality:

- A "release command ring" (RCR), a pipelined mechanism for software to hardware commands that release buffers to BMan-managed buffer pools
- A "management command" interface (MC), a low-latency command/response interface for acquiring buffers from buffer pools, and querying the status of all buffer pools
- An interrupt line and associated status, disable, enable, and inhibit registers

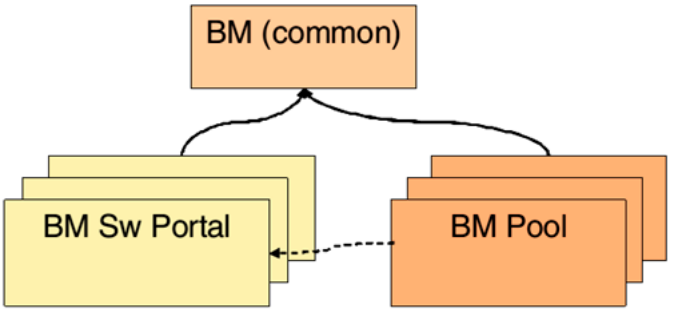
These portal interfaces will be described in more detail in their respective sections.

4.4.4 Driver Architecture

4.4.4.1 Driver Components

The BMan driver contains three basic modules, as shown in Figure 4.4.

Figure 4.4 BMan Driver Modules (from a partition point of view)



The modules are as follows:

- Buffer Manager (common) - The BMan module is a singleton module within its partition. It is responsible for the common hardware registers initialization, and runtime control routines (for example, management of the BMan pool ids). This module must always be initialized when working with any BMan module. This module has an instance for each partition. However, only the driver that is on the master-partition has access to the hardware registers.
- Buffer Manager Portal (BM-Portal) - this module is responsible for all BMan-portal related register space. This module should be initialized once per BMan-portal. Note that the BMan-portal may be either affined to core or affined to user application (that is, application may run on single-core or several). BMan-portal should be affined to partition (that is, it cannot be shared among partitions.
- Buffer Manager Pool (BM-Pool) - this module is responsible of initializing and controlling a certain BM-pool. User may create several instances per BMan-pool; In this case, the first module will be

initialized in the standard way while all other instances of this BMan-pool will be initialized with "shadow_mode" activated. Note that software stockpiles are maintained by the driver per BMan-pool instance. So, user may not put locks when accessing this pool unless the BMan-portal that is being used to access this pool is being shared by several cores or applications.

4.4.5 Programming Model

4.4.5.1 General

Initialization of the BMan driver is carried out by the application according to the following sequence:

- The BMan common functionality initialization uses the following steps:
  - Calling the configuration routine with basic parameters
  - Calling the initialization routine
- For each BMan-portal required by the user, initialize the BM-Portal module according to the following steps:
  - Calling the configuration routine with basic parameters
  - Calling the advance initialization routines to change driver defaults
  - Calling the initialization routine
- For each BMan-pool required by the user, initialize the BM-Pool module with the following steps:
  - Calling the configuration routine with basic parameters
  - Calling the advance initialization routines to change driver defaults
  - Calling the initialization routine

Table 4.9 Functions in chronological order

Function to Call	Description
bmConfig	Configures BM using user's basic parameters and default values
bmConfigXxx	Optional - configures selected BM parameters to be other then default
bmInit	Initializes the BM by applying configured parameters.
bmPortalConfig	Configures BM Portal using user's basic parameters and default values

Table 4.9 Functions in chronological order

Function to Call	Description
bmPortalConfigXxx	Optional - configures selected BM Portal parameters to be other then default
bmPortalInit	Initializes the BM Portal by applying configured parameters
bmPollConfig	Configures BM Pool using user's basic parameters and default values
bmPoolInit	Initializes the BM Pool by applying configured parameters

4.4.6 Source code

SmartDSP OS Bman B4860 driver header files:

- initialization\arch\peripherals\dpaa\bman\include\bman_init.h
- include\arch\peripherals\dpaa\bman\bman.h
- include\arch\peripherals\dpaa\dpaa.h

4.4.7 Demo Use Cases

- demos\starcore\b4860\dpaa_demo
- demos\starcore\b4860\dpaa_integration

4.5 Serial RapidIO (sRIO)

4.5.1 Introduction

Serial RapidIO (R) (sRIO) is a point-to-point, packet-based, switched-system, interconnect technology. sRIO provides high performance and bandwidth as well as faster bus speed interconnect technology.

sRIO is an intra-system interface that provides chip-to-chip and board-to-board communication at GBps performance levels.It was developed to control data and signal plane applications of large-scale embedded systems.

RIO is characterized by the following:

- One or two hosts but up to 65,534 agents.
- Agents can scan and exchange data in the system.
- RIO agents are connected via RIO switches.

The RIO standard defines multiple transaction types:

Table 4.10 RIO Transaction Types

Transaction	Description	Module Applications
Direct I/O	Writes to destination memory.	Modules can work simultaneously. • Bus: specifies the address space going to the SRIO interface. • Packet: exchanges messages between SRIO agents. • Doorbell: messages contain 16-bit information, source ID, and destination ID.
Doorbell	Short (2 bytes) messages from source to destination.	
Messaging	Transfers a message (max. 4 KB) from source to destination.	
Maintenance Accesses	• Standard or implementation-specific registers in a destination's RIO address space. • Port-Write: passes error information.	

4.5.2 Features

A sRIO driver supports the following RIO transaction types:

Table 4.11 RIO Transaction Types

Transaction	Description
Type5	RIO NWRITE
Type6	RIO SWRITE
Type8	RIO Maintenance
OCeaN DMA	Direct access
ATMU Bypass	MSC814x supports small system maintenance.
ATMU	MSC815x supports small & large system maintenance.

4.5.2.1 Relevant SoC

SmartDSP OS sRIO drivers support MSC814x, MSC815x, and B4860 devices; see Table 4.12.

Table 4.12 SoC RIO Support

Support	Description
Direct I/O	• Supported by OCeaN DMA (in the HW). • Programmable via the following: – Standard SmartDSP OS DMA API channels, chains, and transfers – Direct access to OCeaN DMA
Maintenance	• Supported by `demos\starcore\msc815x\uart` – MSC814x supports small systems maintenance using ATMU bypass. Maintenance access is generated using `srioDmaMaintenanceAccess()`. – MSC815x and B4860 support maintenance transactions in small and large systems using ATMU.

4.5.3 Architecture

This section covers the SW architecture of the system sRIO driver.

4.5.3.1 Components

This section details sRIO driver components.

Table 4.13 sRIO Driver Components

Component	Description
Controller	• Represents the HW block. • 34-bit address support. • I/O ATMU windows.
SRIO Window	ATMU window for I/O type 5/6 transactions.
Maintenance ATMU	ATMU window for maintenance transactions.
Port	Represents the HW port.

4.5.4 Data Flow

This section outlines transaction and maintenance data flows.

NWRITE and SWRITE transaction flows:

1. Application pre-configures I/O windows.
2. Application writes data to the outbound ATMU window address space.
3. MSC814x OCN DMA transfers:
   - For sRIO transactions executed with OCN DMA API over B4860 and MSC8156_FAMILY.
   - [Optional} For MSC8157

Maintenance data flow:

1. Open maintenance ATMU.
2. Write data via SRIO maintenance access procedure.
3. Close maintenance ATMU.

4.5.5 Programming Model

This section covers driver bring-up—first by the OS, then by the application-chosen API.

4.5.5.1 General

The sRIO driver API has the following functionality:

- write transactions
- maintenance accesses
- sRIO generic

See the SmartDSP OS API Reference Manual for a complete list of sRIO LLD API.

Table 4.14 describes the MSC815x driver API in light of driver functionality.

NOTE For the MSC814x API, replace the function prefix ocnDMA with srioDMA.

Table 4.14 Functionality Perspective of MSC815x API

Functionality API	Functions	Description
Write Transactions	• srioOutboundWindowOpen • srioOutboundWindowFind • srioOutboundWindowFree • srioOutboundWindowEnable • srioOutboundWindowDisable • srioInboundWindowOpen • srioInboundWindowFind • srioInboundWindowFree • srioInboundWindowEnable • srioInboundWindowDisable	I/O window management.
Maintenance API	• srioMaintenanceAtmuOpen • srioMaintenanceAtmuFree • srioMaintenanceAccess • srioMaintenanceTargetSet	Maintenance access management.
sRIO Generic API	• srioRecover • srioClearPortErrors • srioAlternateIdSet • srioAlternateIdDisable • srioAcceptAllConfigure • srioDeviceAdd	sRIO management and controls.

4.5.5.2 Calling Sequence Example

Table 4.15 lists functions included in the write transactions API of the SmartDSP OS MSC8156_FAMILY sRIO driver. Functions include kernel bring-up, application bring-up, application runtime, and application teardown.

Table 4.15 N/SWRITE Transactions API

State	Function	Description
Kernel Bring-up	`osInitialize() -> archDeviceInitialize() -> srioInitialize()`	• User application indirectly calls these functions. • osInitialize() calls oInitialize()— as based on os_config.h defines and msc81xx_config.c configuration structures. • Initializes sRIO driver. • Opens ATMU I/O windows. • Enables I/O windows. • Initializes sRIO system on the ADS.
Application Bring-up	`ocnDmaControllerOpen()`	• Must be first function called. • Returns an OCN DMA controller handle. • Handle is a parameter in calling other functions.
	`ocnDmaChannelOpen`	• Opens a pre-allocated channel. • Initializes channel structure as per configuration parameters.
	`ocnDmaChainCreate`	• Creates an OCN DMA chain. • Initializes chain structure as per configuration parameters.
	`ocnDmaChainTransferAdd`	Adds a transfer to the OCN DMA chain.

Table 4.15 N/SWRITE Transactions API

State	Function	Description
Application Runtime	`ocnDmaTransfer`	Performs a transfer & transaction starts.
	`ocnDmatransferAdd`	Adds a transfer to a chain.
	`ocnDmaChannelBind`	Binds a chain to a channel.
	`ocnDmachannelStart`	Starts transaction.
Application Teardown	`ocnDmaChannelUnbind`	Unbinds chain from a channel.
	`ocnDmaChannelClose`	Frees OCN DMA channel.

Table 4.16 lists functions included in the maintenance transactions API of the SmartDSP OS MSC8156_FAMILY sRIO driver. The functions include kernel bring-up, application bring-up, application runtime, and application teardown.

Table 4.16 MAINTENANCE Transactions API

State	Function	Description
Kernel Bring-up	`osInitialize() -> archDeviceInitialize() -> srioInitialize()`	• User application indirectly calls these functions. • osInitialize() calls oInitialize()— as based on os_config.h defines and msc81xx_config.c configuration structures. • Initializes sRIO driver. • Initializes sRIO system on the ADS
Application Bring-up	`ocnDmaControllerOpen()`	• Must be first function called. • Returns an OCN DMA controller handle. • Handle is a parameter in calling other functions.
	`ocnDmaChannelOpen`	• Opens a pre-allocated channel. • Initializes channel structure as per configuration parameters.
	`srioMaintenanceAtmuOpen`	Opens ATMU window for RIO maintenance.
Application Runtime	`srioMaintenanceAccess`	Executes maintenance access (W/R).
Application Teardown	`srioMaintenanceAtmuClose`	Frees a maintenance ATMU window.
	`ocnDmaChannelClose`	Frees an OCN DMA channel.

4.5.5.3 Functionality

The below description provides information for operating data transactions over sRIO.

4.5.5.3.1 Initialization

Follow these steps to initialize data transaction functionality over the sRIO driver:

1. Enable sRIO support in the application os_config.h file as follows:
   a. As per the architecture, set #define MSC81XX_SRIO ON or #define B4860_SRIO ON.
   b. For B4860, define the maximum number of connected devices using #define SRIO_MAX_NUM_CONNECTED_DEVICES <number>.
2. Enable OCN DMA support in the application os_config.h file as follows:
   a. For MSC81xx set #define MSC81XX_OCN_DMAx ON.
   b. For B48660, use one of the following:

      #define OCN_DMA0 ON

      #define OCN_DMA1 ON
3. Allocate memory—enables driver to handle an OCN DMA channel.
4. Allocate memory for a OCN DMA chain handle.
5. User application determines DMA chain parameters.
6. Get an OCN DMA controller handle using osDmaControllerOpen().
   - Function returns the handle to the driver-allocated OCN DMA controller.
   - User application retrieve this handle to open OCN DMA channels and create OCN DMA chains.
7. Create an OCN DMA chain using ocnDmaChainCreate().
8. Add DMA transfers to the DMA chain using ocnDmaChainTransferAdd() or ocnDmaChainTransferAddEx().
   - ocnDmaChainTransferAddEx() returns a handle to added transfers; this allows attribute modification.

4.5.5.3.2 Runtime

Follow these steps to run sRIO in the application:

1. Bind the OCN DMA channel with a chain using ocnDmaChannelBind().
2. Start the OCN DMA transaction using ocnDmaChannelStart().
3. Poll a channel to check if it is active using ocnDmaChannelIsActive().

4.5.5.3.3 Free

Follow these steps to free OCN DMA:

1. Ensure OCN DMA channel is inactive.
2. Unbind DMA channel and chain using the function, osDmaChannelUnbind().
   - Indicates DMA chain can be deleted or bound to a different DMA channel.
   - Indicates DMA channel can be closed or bound to a different DMA chain.
3. Release DMA channel using osDmaChannelClose().
   - Releases all DMA channel resources (except memory) belonging to a user application.
4. Release DMA chain using osDmaChainDelete().
   - Releases all OCN DMA channel resources (except memory) belonging to a user application.

4.5.5.4 Source Code

SmartDSP OS sRIO 81xxx driver header files:

- include\arch\starcore\msc815x\msc815x_srio.h
- initialization\arch\msc815x\include\msc815x_srio_init.h
- include\arch\starcore\msc814x\msc814x_srio.h
- initialization\arch\msc814x\include\msc814x_srio_init.h

SmartDSP OS sRIO B4860 driver header files:

- initialization\arch\peripherals\srio\include\srio_init.h
- include\arch\peripherals\srio\srio.h

4.5.6 Resource Management

Minimize driver data footprints by assigning OCN DMA channels-to-cores at compilation time.

- Driver configures the master core (osGetMasterCore()) to handle errors during srioInitialize() execution.
- Driver handles HW-related errors.
- User callback functions handle functional errors.

4.5.7 Demo Use Cases

- demos\starcore\msc815x\rio_dma
- demos\starcore\msc814x\rio_dma
- demos\starcore\b4860\srio

4.5.8 Resource Management

Minimize driver data footprints by assigning OCN DMA channels-to-cores at compilation time.

- Driver configures the master core (osGetMasterCore()) to handle errors during srioInitialize() execution.
- Driver handles HW-related errors.
- User callback functions handle functional errors.

4.5.9 Demo Use Cases

- demos\starcore\msc815x\rio_dma
- demos\starcore\msc814x\rio_dma

4.6 Multi Accelerator Platform Engine–Baseband (MAPLE–B/B2/B3)

4.6.1 Introduction

MAPLE-B (MSC8156) is a baseband, algorithm accelerator for Turbo, Viterbi, FFT/iFFT, DFT/iDFT, and CRC algorithms. MAPLE-B is a PSIF—a programmable controller with CRC accelerator and DMA capabilities.

MAPLE-B2/B2F/B2P (MSC8157 / PSC913x) is a multi-standard, baseband, algorithm accelerator for Channel Decoding/Encoding, Fourier Transforms, UMTS chip-rate processing, OFDMA and SC-FDMA equalization, and CRC algorithms. MAPLE-B2 is a second generation PSIF (PSIF2)—a programmable controller with DMA capabilities.

MAPLE-B3 (B4860) is a multi-standard, baseband, algorithm accelerator; it targets Layer 1 processing acceleration of common wireless standards such as UMTS, 3G-LTE, TDD-LTE, LTE-Advanced, TD-SCDMA, and WiMAX. MAPLE-B3 is comprised of three instances—MAPLE-B3W, which primarily targets UMTS chip-rate processing, and MAPLE-B3LW0 and MAPLE-B3LW1. The latter are identical instances that target LTE/WiMAX processing and UMTS symbol rate processing.

4.6.2 Features

Features of MAPLE–B/B2/B3 drivers are noted below.

- Implemented as part of SmartDSP OS drivers.
- Complies with SmartDSP OS driver rules and API.
- COP-related features:

  - Each processing element functions as a COP device.
  - Belongs to the COP LLD layer that manages jobs and queues.
  - MAPLE also acts as a COP device
- Allows multiple hosts to share a single MAPLE–B/B2/B3 device OR a single host to use multiple MAPLE–B/B2/B3 devices.
  - Driver must identify the master host controlling the MAPLE–B/B2/B3.
  - Each MAPLE device must have a single master—more than one will cause an error.

MAPLE–B/B2/B3 goals:

- Minimum overhead.
- Seamless integration of SoC bring-up code.
- Support B4860, PSC913x, and MSC815x families.
- User chooses the configuration interface.
- User dispatches multiple BDs in a single function call.

4.6.2.1 Relevant SoC

B4860, PSC913x, and MSC815x families.

4.6.3 Architecture

This section covers the SW architecture of MAPLE–B/B2/B3 drivers.

4.6.3.1 Components

This section details MAPLE–B/B2/B3 driver components found in the SmartDSP OS. See Table 4.17 for further details.

Table 4.17 Maple–B/B2/B3 Components

Components	Description
COP Abstraction Layer	COP module layer; see 3.3Chapter 3, COP Module.
Maple LLD	Functions are activated via COP abstraction layer functions.

Table 4.17 Maple–B/B2/B3 Components

Components	Description
Device	• Processing element abstraction. • Relevant registers and parameters of a given processing element are handled as follows: – unified under the device concept – initialized when opening the device
Channel	• BD ring abstraction. • A channel, at any given time, can only contain a limited number of jobs: – each reaped job is removed from the channel – ensures space for newly dispatched jobs
Job	• BD abstraction. • Similar structure to the HW BD.

4.6.3.2 Design Decisions

Design decisions, along with their reasoning, are noted below.

1. Restricted channel usage—two different cores cannot open the same channel.
   - Driver was written to minimize runtime overhead.
   - Minimum mutual-exclusion locks.
2. Device-sharing—first core to open the device initializes all the registers and parameters; other cores will use the existing LLD device handle.
   - Support multicore usage of MAPLE–B/B2/B3.
   - Support multicore usage of all processing elements.
3. During the MAPLE open device routine, each core subscribes to all MAPLE error interrupts.
   - Provide error handling and failure recovery mechanism.
4. Each channel can handle a limited (user-defined) number of jobs.
   - Allows driver to maintain the channel (for the user).
   - Allows driver to maintain the BD ring DRAM (for the user).

4.6.4 Data Flow

This section outlines the dispatch flow.

4.6.4.1 Dispatch Flow

Jobs are dispatched to the PE following correct channel initialization. Listing 4.1 outlines a structure for a NULL-terminated, linked, COP, job dispatch list. The application, with a single function call, can dispatch multiple jobs to each PE.

Listing 4.1 COP Job Dispatching

```
typedef void* cop_job_id;
 typedef struct cop_job_handle_s
 {
cop_job_id  job_id;
/**< Used by the application to identify finished jobs. */
void *device_specific;
/**< LLD specific parameters */
struct cop_job_handle_s* next;
/**< next job */
 } cop_job_handle;
```

The job dispatch process (see Figure 4.5) is outlined in the following steps:

1. User application handles the following:
   a. Prepares a job array (may be a single element).
   b. Calls `osCopChannelDispatch(jobs)` to send a channel to enqueue jobs.
   c. Passes the array and element(s) to `osCopChannelDispatch()`.

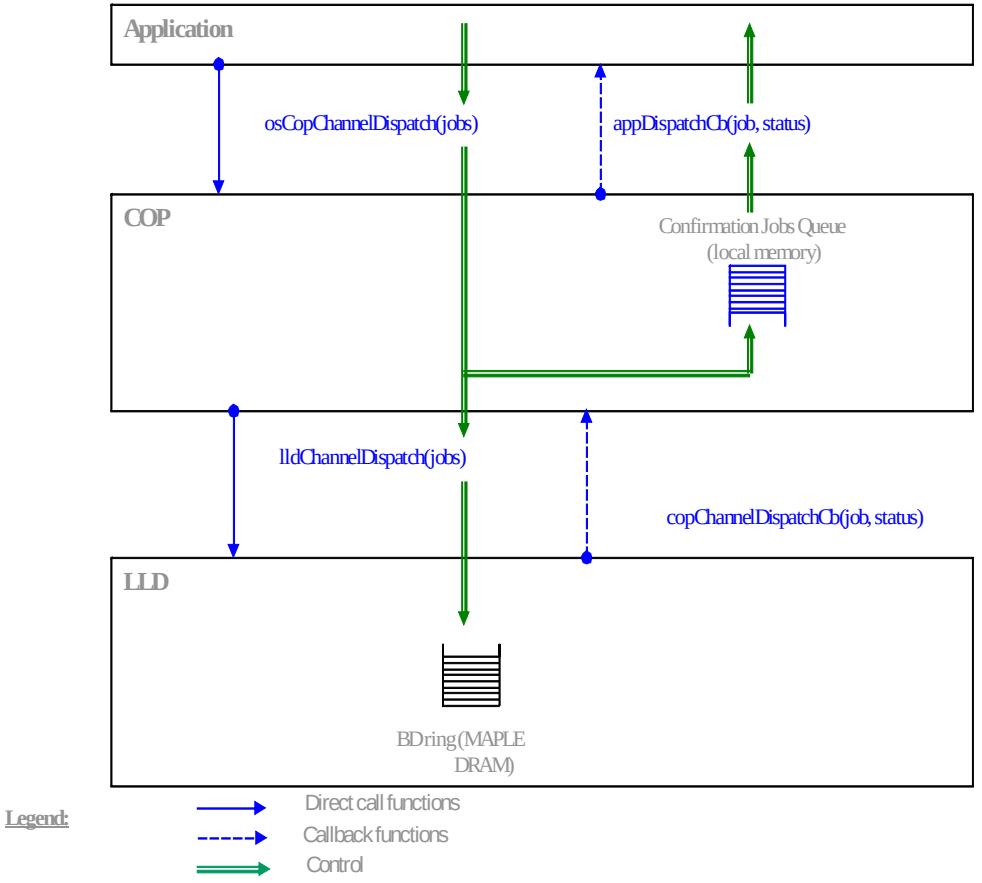
NOTE Use jobs from the memory pool; do not allocate stack jobs. A job pointer is returned after reaping.

2. COP layer handles the following:
   a. Queues all job pointers, taken from the application layer, into the confirmation job queue of the local memory.
   b. Uses `lldChanelDispatch(jobs)` to dispatch/add jobs to the MAPLE–B/B2/B3 memory BD ring.
   c. Passes the array to the LLD at job completion.
3. LLD handles the following:
   a. Prepares BD(s).
   b. Initiates a callback function. `copChannelDispatchCb(job,status)` is used to send BD(s) from MAPLE-B/B2/B3 to local memory.
   c. Copies the Job ID into the BD Task ID field. LLD job description tries to match the BD to allow for a simple sanity check and easy BD preparation.

NOTE During release, sanity checks are not asserted in order to save runtime code.

4. COP calls a user-defined callback function; see Figure 4.5.

Figure 4.5 COP Job Description Procedure



NOTE Data cache coherency is unsupported as neither COP nor LLD manage data.

4.6.4.2 Consume (Reap) Flow

Job reaping is performed under the following circumstances:

- Triggered interrupt is enabled.
- Interrupt is disabled following job dispatch.
- Channel control function is called during runtime.

Job reaping process covers the following:

- LLD iterates over a BD ring.
- LLD calls the COP callback function with the BD status—for each BD.
- COP dequeues a job pointer and calls a user callback function. Before dispatching the job, the user application should have data ready in the MAPLE–B/B2/B3 accessible memory:

4.6.5 Programming Model

This section covers driver bring-up—first by the OS, then by the application-chosen API.

4.6.5.1 General

The MAPLE driver is part of the COP abstraction layer; all driver functions are activated via COP-layer functions.

Bring-up initialization is based on initialization structures defined in `xxxx_config.c`.

Table 4.18 Driver Functions

Function	Description
`osCopDeviceOpen()`	Opens a MAPLE or XxPE[1] device.
`osCopChannelOpen()`	Opens a channel for an XxPE device.
`osCopSharedChannelOpen()`	A COP shared channel can be dispatched from one core and reaped by another (or the same one).
`osCopChannelCtrl()`	Manipulates an XxPE channel.
`osCopDeviceCtrl()`	Manipulates an XxPE device.
`osCopChannelDispatch()`	Dispatches a job list to a channel.
`osInitialize()`	SmartDSP OS initialization using structures defined in `xxxx_config.c`

1.XxPE refers to any supported PE.

4.6.5.2 Calling Sequence Example

Table 4.19 MAPLE–B/B2/B3 Driver API

Status	Name	Description

Kernel Bring-up	`osInitialize()->`
	`osArchDevicesInitialize()->`
	`mapleInitialize()->`
	`copRegister()`
		• User application indirectly calls these functions.
		• osInitialize() calls mapleInitialize() based on os_config.h defines and xxxx_config.c configuration structures.
		• Initializes MAPLE driver.
		• copRegister() takes and registers the driver in the COP LLD layer.

	`osInitialize()->`
	`osArchDevicesInitialize()->`
	`mapleXxpeInitialize()->`
	`copRegister()`
		• mapleXxpeInitialize() initializes MAPLE XXPE.
		• mapleXxpeInitialize() calls copRegister().

Status	Name	Description

Application Bring-up	`osCopDeviceOpen()->`
	`mapleOpen()->`
	`maple_init()`
		• Using given parameters, osCopDeviceOpen() opens a COP device for operation; it returns a device handle or NULL.
		• mapleOpen() opens a MAPLE–B/B2/B3 channel and returns a channel handle.
		• maple_init() initializes the MAPLE–B/B2/B3 controller.

	`osCopDeviceOpen()->` `mapleXxpeOpen()`	• Using given parameters, osCopDeviceOpen() opens a COP device for operation; it returns a device handle or NULL. • mapleXxpeOpen()opens all supported PEs.
	`osCopDeviceCtrl(`MMU_ENABLE`)->` `mapleDeviceCtrl()`	• Only relevant for MAPLE-B3 and higher versions. • Control API for Maple MMU initialization: MAPLE_CMD_MMU_ENABLE MAPLE_CMD_MMU_DISABLE MAPLE_CMD_MMU_PROTECT_DISABLE MAPLE_CMD_MMU_PROTECT_ENABLE MAPLE_CMD_MMU_SEGMENT_FIND MAPLE_CMD_MMU_SEGMENT_UPDATE MAPLE_CMD_MMU_SEGMENT_ENABLE MAPLE_CMD_MMU_SEGMENT_DISABLE

Status	Name	Description
Application Bring-up, continued	`osCopChannelOpen()` or `osCopSharedChannelOpen()->` `mapleXxpeChannelOpen()`	• Using given parameters, osCopDeviceOpen() opens a COP device for operation; it returns a device handle or NULL. • mapleXxpeChannelOpen() opens a MAPLE PE channel.
Application Runtime	`osCopChannelDispatch()->` `mapleXxpeChannelDispatch()`	• osCopChannelDispatch() sends a job to a channel previously opened for transmission. • mapleXxpeChannelDispatch() sends a message.
	`osCopDeviceCtrl()->` `mapleDeviceCtrl()`	• osCopDeviceCtrl() performs control commands on a device. • mapleDeviceCtrl() performs a device-level controlling function.
	`osCopChannelCtrl()->` `mapleXxpeChannelCtrl()`	• osCopChannelCtrl() performs control commands on a channel. • mapleXxpeChannelCtrl() performs a channel control-level function.
	`mapleXxpeIsr()->` `mapleXxpeChannelReap()->` `copChannelDispatchCb()`	• Invokes an interrupt service routine when the PE triggers an interrupt. • While polling a channel for finished jobs, the reap process can be indirectly activated using osCopChannelCtrl().

4.6.5.3 Functionality

The following section covers MAPLE-B/B2/B3 initialization processes and runtime.

4.6.5.3.1 Configuration File

SmartDSP OS configuration file, `os_config.h`, enables MAPLE-B/B2/B3 and PE by setting the below noted MAPLE and MAPLE_XXPE flags to `ON`.

For example MAPLE-B/B2:

- `#define MAPLE  ON`
- `#define MAPLE_TVPE  ON`
- `#define MAPLE_FFTPE ON`
- `#define MAPLE_DFTPE ON`
- `#define MAPLE_CRCPE ON`

The SmartDSP OS configuration file noted below is MAPLE-B3-specific as MAPLE-B3 has three independent MAPLE devices.

- **#define** MAPLE_0 ON /**< Using Maple Device */
- **#define** MAPLE_1 OFF /**< Using Maple Device */
- **#define** MAPLE_2 ON /**< Using Maple Device */
- **#define** MAPLE_0_FTPE_0 OFF /**< Using Maple 0 eFTPE 0 Device */
- **#define** MAPLE_0_FTPE_1 OFF /**< Using Maple 0 eFTPE 1 Device */
- **#define** MAPLE_0_FTPE_2 OFF /**< Using Maple 0 eFTPE 2 Device */
- **#define** MAPLE_1_FTPE_0 OFF /**< Using Maple 1 eFTPE 0 Device */

4.6.5.3.2 Initialization Processes

Table 4.20 outlines the three initialization processes.

Table 4.20 MAPLE–B/B2/B3 Initialization Processes

Process	Description
Driver Initialization	• Initializes MAPLE-B/B2/B3. • Enables a PE-specific driver at compilation time.

Table 4.20 MAPLE–B/B2/B3 Initialization Processes

Process	Description
PE Initialization	• Opens the device (MAPLE and all XxPE) via the COP interface. • Returns a device handle.
Channel Initialization	• Opens a MAPLE-B/B2/B3 channel via the COP interface using the device handle returned by PE initialization. – Returns a channel handle. • One-to-one relationship exists between a channel and a BD ring. – Every MAPLE-B/B2/B3 PE supports up to eight high- and low-priority channels. • COP channels are not statically assigned to cores. – Cores can open channels by specifying a required channel number. – Every core opens at least one channel to dispatch jobs. – A channel, opened by a given core, cannot be opened by another core.

Table 4.21 details driver, PE, and channel initializations. Bring-up initialization is based on initialization structures defined in the architecture specific `xxxx_config.c` file.

Table 4.21 MAPLE Initialization Processes

Driver Initialization	PE Initialization	Channel Initialization
1. Initialize using functions osInitialize() or runtime osCopDeviceOpen(). – Configures MAPLE–B/B2/B3, DSP, and (optionally) MMU. – Uploads a µcode image based on a selected standard: 3GLTE, 3GPP, 3GPP2, WiMAX, or 3GPP2HS. – Divided into specific initializations: SmartDSP OS or MAPLE–B/B2/B3. 2. Access MAPLE–B/B2/B3 with MAPLE_IN_PLATFORM value at smartdsp_os_device.h. – Set to OFF: To access MAPLE, user provides callback functions with a 32-bit access resolution. – Set to ON: User provides callback functions; if none are provided then driver reverts to internal functions. – If the driver is unfamiliar with the interface used to access MAPLE–B/B2/B3, then user application must provide a callback function to control R/W access. – If the driver does not perform data access, then user application does not need to provide a callback function.	1. Initialize MAPLE–B/B2/B3. 2. Call osCopDeviceOpen() to open a COP device. 3. Initialize PE registers. 4. User application provides MAPLE–B/B2/B3 instantiation (to which the PE belongs). 5. User application provides PE configuration: – MAPLE–B/B2/B3 master configures PE. – First core to call function osCopDeviceOpen() configures the PE. – MAPLE–B/B2/B3 driver provides device-to-core synchronization mechanisms.	1. User application calls osCopChannelOpen(). 2. User application is provided with specific information: – channel number – channel priority – interrupt line (if applicable) – number of BDs in the ring 3. Initialize COP and LLD channels. – COP channel initialization structure contains a pointer to the LLD channel initialization structure. 4. LLD uses MAPLE–B/B2/B3 DRAM to synchronize the following: memory, interrupts, and BD ring assignments of multiple devices sharing MAPLE–B/B2/B3. – Multiple channels—on the same core—can share the same MAPLE–B/B2/B3 interrupt line.

4.6.5.3.3 MAPLE-B/B2/B3 Runtime

The MAPLE–B/B2/B3 driver does not handle data processed by the MAPLE-B/B2/B3. User applications must ensure that all required data is in place before dispatching a job(s) using the MAPLE–B/B2/B3 driver.

The COP job list consists of a `NULL`-terminated, linked, structure list. Each structure points to the LLD job description.

The following functions detail MAPLE–B/B2/B3 driver runtime:

1. Dispatch a job to a channel and send BD(s) to the PE channel using `osCopChannelDispatch()`.
2. Returns `OS_SUCCESS` or an error code from `os_error.h`. The function takes the following arguments:
   - Channel handle as returned from `osCopChannelOpen()`.
   - Pointer to a job array.
   - Number of jobs for dispatch.
3. `osCopDeviceCtrl()` controls COP devices using the following commands:
   - `MAPLE_CMD_SEMAPHORE_GET`
     – Retrieves semaphore used for sharing the MAPLE-B/B2/B3 driver.
   - `MAPLE_CMD_SEMAPHORE_RELEASE`
     – Releases semaphore used for sharing the MAPLE-B/B2/B3 driver.
   - `MAPLE_CMD_MALLOC`
     – Allocates memory from the MAPLE-B/B2/B3 DRAM.
   - `MAPLE_CMD_MMU_ENABLE`
     – Enable mapleb3 mmu address translation.
     – By default enables MMU protection.
     – Use MAPLE_CMD_MMU_PROTECT_DISABLE to disable.
   - `MAPLE_CMD_MMU_DISABLE`
     – Disable mapleb3 mmu address translation.
   - `MAPLE_CMD_MMU_PROTECT_DISABLE`
     – Disable mapleb3 mmu memory protection disable.
   - `MAPLE_CMD_MMU_PROTECT_ENABLE`
     – Disable mapleb3 mmu memory protection enable.
   - `MAPLE_CMD_MMU_SEGMENT_FIND`
     – Find available mapleb3 mmu segment.
     – Pass pointer to os_mmu_segment_handle as parameters.

- MAPLE_CMD_MMU_SEGMENT_UPDATE
  - Update mapleb3 mmu segment.
  - Pass pointer to maple_mmu_seg_update_t as parameter
  - Can only be used on descriptors found by MAPLE_CMD_MMU_SEGMENT_FIND.
- MAPLE_CMD_MMU_SEGMENT_ENABLE
  - Enable mapleb3 mmu segment.
  - Pass pointer to os_mmu_segment_handle as parameter.
  - Possible to enable segment usage of MAPLE_MMU_ATTR_ENABLE with MAPLE_CMD_MMU_SEGMENT_UPDATE.
- MAPLE_CMD_MMU_SEGMENT_DISABLE
  - Disable mapleb3 mmu segment.
  - Pass pointer to os_mmu_segment_handle as parameter.
- MAPLE_TVPE_CMD_SET_VITERBI_POLY
  - Sets viterbi polynomial sets (MTVPVSxCyP).
  - Set BD VIT_SET field.
  - Select between sets using the maple_tvpe_vit_set_t enumeration.
- MAPLE_TVPE_CMD_LOAD_INTERLEAVER
  - Used for 3GPP2HS EVDO.
  - Sets (MP_MBUS + 0x464) = &maple_tvpe_interleaver[0].
  - Execute **after** the maple_init function but **before** BD activation.

NOTE EVDO is not supported on all MAPLE platforms. See the *Core Reference Manual* for further information.

4. osCopChannelCtrl() controls COP channels using the following commands.
   - MAPLE_PE_CH_CMD_RX_POLL
     - Polls xxPE channel and BD ring for completed jobs.
   - MAPLE_PE_CH_CMD_VIRT_TRANS_ENABLE
     - Update channel translation mode to enable virtual to physical translation.
   - MAPLE_PE_CH_CMD_VIRT_TRANS_DISABLE
     - Update channel translation mode to disable virtual to physical translation.

4.6.5.4 Source Code

The MAPLE SmartDSP OS driver can be used *as is* with inclusion of the Driver Initialization and Runtime header files.

The other files—µcode initialization API, µcode header file, and MAPLE Driver memory map—are included in the driver initialization and runtime files.

Table 4.22 Source Codes

Source	Type	Detail
Library	Driver	• "SmartDSP\lib\msc815x*_drivers.elb" • "SmartDSP\lib\psc9131*_drivers.elb"
	Kernel	• "SmartDSP\lib\msc815x*.elb" • "SmartDSP\lib\psc9131*.elb"
	MAPLE µcode	"SmartDSP\lib\maple*.elb"
Files	Driver Initialization Header File	"SmartDSP\initialization\arch\peripherals\maple\include*.h"
	Driver Runtime Header File	• "SmartDSP\drivers\maple\rev1\include*.h" • "SmartDSP\drivers\maple\rev3\include*.h"
	µcode Initialization API	"SmartDSP\initialization\arch\peripherals\maple\maple_api\msc81XX\include\maple_api.h"
	µcode Header File	"SmartDSP\initialization\arch\peripherals\maple\maple_api\msc81XX\revX\include*"
	MAPLE Driver Memory Map	• "SmartDSP\include\arch\peripherals\maple\rev1*" • "SmartDSP\drivers\maple\rev3\include*.h"

4.6.6 Demo Use Cases

- "SmartDSP\demos\starcore\msc815x\maple_*"

- “SmartDSP\demos\starcore\msc815x\mapleb2_*”
- “SmartDSP\demos\starcore\psc913x\maple*”
- “SmartDSP\demos\starcore\b4860\maple*”

4.7 Common Protocol Radio Interface (CPRI)

4.7.1 Introduction

Common Protocol Radio Interface (CPRI) supports a connection between Radio Equipment (RE) and a Radio Equipment Controller (REC). This capability allows an antenna and RF amplifier to be separated—normally they need to be located within range of the RF controller handling data collection and net connection. RE and REC topologies are flexible with one REC dealing with several, variously connected, REs.

4.7.2 Features

The CPRI driver supports a number of main features as well as data traffic formats. The latter include IQ, VSS, and C/M interfaces (Ethernet and HDLC). Their initialization and usage is optional as per application needs.

The CPRI driver supports these main features:

- CPRI link rates:
  - 1228.8 Mbps
  - 2457.6 Mbps
  - 3072.0 Mbps
  - 4915.2 Mbps
  - 6144.0 Mbps
  - 9830.4 Mbps
- LTE sampling rates:
  - 1.92 MHz (channel BW 1.25 MHz)
  - 3.84 MHz (channel BW 2.5 MHz)
  - 7.68 MHz (channel BW 5 MHz)
  - 15.36 MHz (channel BW 10 MHz)
  - 23.04 MHz (channel BW 15 MHz)
  - 30.72 MHz (channel BW 20 MHz)
- WiMAX sampling rates:

  - 4 MHz
  - 5.6 MHz
  - 8 MHz
  - 10 MHz
  - 11.2 MHz
- All carriers—of the same link—run with the same sampling rate. An auxiliary interface is necessary to support different radio interfaces, running at different sampling rates, on the same link.
- Mapping methods:
  - LTE: IQ sample-based mapping method 1 and the backward compatible mapping method 3.
  - WiMAX: IQ sample-based mapping method 1 and backward compatible mapping method 3.
  - WCDMA: backward compatible mapping method 3.
- Auxiliary Mode used in advanced mapping modes/methods to handle load balancing between CPRI pairs.
- Multicast Mode sees IQ data go to system memory and MAPLE-CRPE.
- Auto-negotiation support includes the following:
  - Link Rate setup has an initial rate based on RCW.
  - Protocol setup has current support for CPRI v4.0, v4.1, and v4.2.
  - C/M Plane setup negotiates Ethernet and HDLC speeds.
- Ports:
  - Master port
  - Slave port has auto negotiation timings that differ from the Master port.
  - Endpoint Slave port does not have auxiliary and passes sync from Rx to Tx ports.
  - Each port supports up to 24 antenna carriers.
- Packed and flexible positions for each AxC container.
- Ethernet work around mode to address errata A-007968.

The CPRI driver supports the following data traffic formats:

- IQ antenna samplings of up to 24 channels per lane.
- Vendor Specific Sub-channel (VSS)
- Fast Control and Management MAC (Ethernet)
- Slow Control and Management MAC (HDLC)

4.7.3 Relevant SoC

The relevant SoC are MSC8157, PSC9132, and B4860.

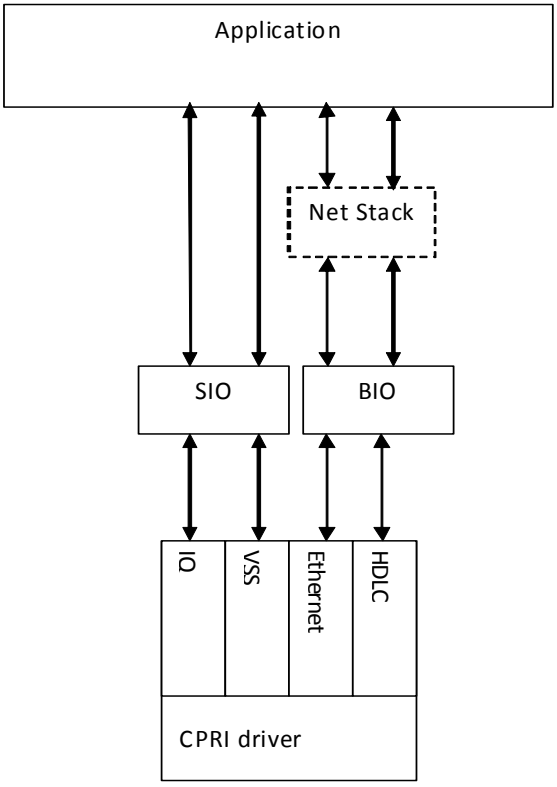
4.7.4 Architecture

This section covers the SW architecture of the CPRI driver. Each CPRI unit data type has separate runtime functionality that is activated by the driver's main initialization function.

4.7.5 Components

This section illustrates CPRI driver components.

Figure 4.6 CPRI Driver Components



4.7.6 Design Decisions

Design decisions are noted below.

- Each CPRI unit is a treated as independently as possible with minimal use of CPRI-global information and structures.
- Data types:
  - User can enable selected data types.
  - User allocates buffers for each data type.
  - Each system data type is defined as a different device.
    - IQ and VSS are handled as SIO devices.
    - Ethernet and HDLC are handled as BIO devices.
  - IQ data type: a core, defined in *init* structure, initializes the data type. A list of supplied cores handle runtime data traffic.
  - VSS, Ethernet, and HDLC data types: a core, defined in *init* structure, initializes the interface and handles runtime.
- Interrupts scheme is as flexible as the hardware permits.
  - CPRI general interrupt can be used for single or multiple events and cores.
  - Interrupt tables are supplied in the initialization structure.
- Initialization structure supplies CPRI with most of the initialization information.
- All CPRI devices, together, perform auto-negotiation phases using a single function.
- CPRI driver allocates Ethernet/HDLC BD rings.
- Ethernet Errata A-007968 is implemented using workaround #1 (see *Errata* details).
  - In Ethernet workaround mode, the VSS buffer contains both Ethernet and VSS data. Both Ethernet and VSS handling procedures are invoked following the VSS transmit threshold event. For more details, see 4.7.8 CPRI Ethernet Errata A-007968 Workaround.

4.7.7 Reconfiguration

The CPRI can be reconfigured during operation (without resetting the SoC or impacting other IPs) in one of the following levels:

- **Level 0**

  Reconfiguration of all settings of all CPRI units in the group.

  Level 0 completely resets and performs an auto negotiation of the CPRI units in the group. Level 0 is invoked with CPRI_DEVICE_RECONFIGURATION_LEVEL0 SIO device control command.

A group consists of all CPRI units operating on the same PLL. The group is defined by the SERDES1 RCW.

A core belongs to a group if it accesses a CPRI unit of that group.

In order to perform a reconfiguration level 0 on the group, all cores which belong to the group must invoke the reconfiguration level 0 command. All cores which have already invoked level 0 are held in a barrier point waiting for the remaining cores of the group, which makes it a blocking command.

Two groups cannot go through reconfiguration level0 in parallel. Thus if one group is already going through level 0, invocation for the other group will return OS_ERR_BUSY.

A successful completion of reconfiguration level 0 returns OS_SUCCESS.

Reconfiguration level 0 command input is of type cpri_reconfiguration_level0_param_t. It consists of 3 fields; Minimal link rate, maximal link rate and a pointer to an array of elements of type cpri_init_params_t. This input is very similar to the input of cpriInitialize().

The CPRI units in the array must all belong to the appropriate group.

There are limitations on the CPRI settings for units to be level 0 reconfigurable:

– Only a single core accesses the unit (ie no multiple cores accessing different AxC's of the same unit).

There are a few limitations over the settings which can be changed in reconfiguration:

– HDLC and Ethernet data types must be active or non active just as in previous setting.

– For all the C&M data types the initializing core must remain same.

– For HDLC and Ethernet data types the RX and TX BD rings must remain same.

- **Level 1**

  When there is no need to change the link rate, each of the 8 lanes can be reconfigured independently. The reconfiguration starts with loss of synchronization, reconfiguration and resynchronization. IQ data is lost and Eth, HDLC, VSS should not be sent or should be ignored. The changes should be done in parallel on both sides of the link.

  Level 1 is invoked with CPRI_DEVICE_RECONFIGURATION_LEVEL1 SIO device control command.

  Reconfiguration level 1 command input is of type cpri_init_params_t.

  Level 1 shares the limitations as level 0.

  In addition, Level1 doesn't modify interrupts scheme. The interrupts table field in cpri_init_params_t is ignored.

- **Level 2**

  When there is no need to change the link rate, each of the eight lanes can be reconfigured independently. The reconfiguration is done without loss of synchronization, so the C&M remains

active and so is the daisy chain unless the Auxiliary mask is changed. Only the IQ data of the reconfigured DMA is lost during the reconfiguration so it is not sent and should be ignored. The changes should be done in parallel on both sides on the link.

Level 2 is invoked with CPRI_DEVICE_RECONFIGURATION_LEVEL2 SIO device control command.

Reconfiguration level 2 command input is of type cpri_iq_init_params_t.

Reconfiguration level 3 which is mentioned in the CPRI chapter in RM is also part of reconfiguration level 2, because the Auxiliary parameters are part of the input structure cpri_iq_init_params_t. It allows the application to modify IQ parameters and/or Auxiliary parameters.

The following limitations apply over settings modifications:

– Only a single core accesses the unit (ie no multiple cores accessing different AxC's of the same unit).

– Initializing core must not be changed.

– Shared RX and TX synchronization must remain same.

- **Level 3**

  Included in Level 2. For more information, see level 2 description above.

4.7.8 CPRI Ethernet Errata A-007968 Workaround

The driver incorporates the support for B4860/B4420 CPRI errata (A-001968) workaround #1 implementation.

The solution is applicable when you need to send the Ethernet packets with 7 PREAMBLE bytes as required in the Ethernet standard.

The CPRI Ethernet hardware in B4 generates 3 PREAMBLE bytes. The CPRI Ethernet workaround support is a software implementation replacing the CPRI Ethernet hardware transmit data path.

A flag is added to the CPRI Ethernet initialization parameters structure to allow application to switch the CPRI Ethernet TX data path for enabling the solution.

NOTE The solution executes 4B5B and CRC encodings, which results in the performance impact compared to the regular hardware (non-workaround) mode.

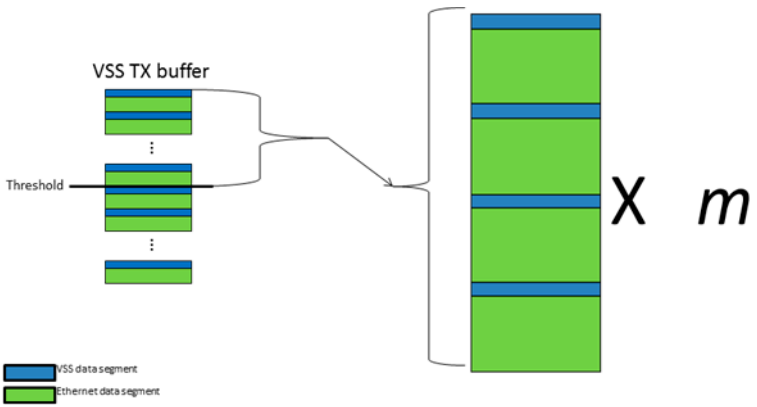
4.7.8.1 Driver Mechanism

For a detailed description, see SRS. In the workaround mode of operation, Ethernet data type is handled following the VSS TX threshold event. The VSS data type, when enabled in this mode is also handled using the VSS TX threshold event.

The solution works only for the TX Ethernet data path and the VSS when Ethernet is in workaround mode. The RX path remains unchanged.

The driver internally allocates a VSS TX buffer containing the Ethernet data segments and VSS data segments, alternately.

Figure 4.7 VSS TX buffer



For information about the value *m*, see 4.7.8.1.3 value m.

4.7.8.1.1 Ethernet

The application provides a packet to transmit. If there is a free buffer descriptor (BD) in the BD ring, the driver calculates the cyclic redundancy check (CRC). If CRC insertion is enabled, then the packet is assembled encoded with 4B5B and a new BD is set.

Using a VSS TX threshold event, the driver processes the BDs available in the ready state. Each ready BD is copied to the Ethernet data segments.

If all of the Ethernet data segments are exhausted, the driver continues the copying in next VSS TX threshold event.

4.7.8.1.2 VSS

Due to the layout of the VSS TX data, the application can no longer fill the VSS data to transmit in a continuous manner. The application fills the data in the VSS segments.

For more information, see 4.7.8.2.2 VSS.

4.7.8.1.3 value m

The VSS transmits the threshold event. The buffer size is not explicitly set by the application, instead it sets the buffer value as *m*.

The VSS TX threshold size becomes equal to the data size in value *m* number of high frequency (HF). The VSS TX buffer size is twice the threshold.

4.7.8.2 Application

Using Ethernet workaround mode requires further application adjustments, that are demonstrated in the introduction demo.

4.7.8.2.1 Ethernet

For workaround mode, consider the following modifications:

- The block size provided as first parameter in the function `osMemPartCreate()` must be enlarged by 25% + 20 bytes. This is to ensure the space for required by the driver for internal operation.
- Since the WA solution based on employing the VSS transmit activity for Ethernet data, the driver allocates internally a VSS transmit buffer. The application must provide the memory type (like local, shared, and so on) and the buffer attributes (cacheable, write back, guarded and so on).
- You have to provide a *value_m* value.
- In the interrupts table, `CPRI_VSS_EVENT` transmit event must be enabled even when the VSS data types are not used and only the Ethernet data types are enabled. `CPRI_ETHERNET_EVENT` should not be enabled as a transmit event.

4.7.8.2.2 VSS

The CPRI VSS TX mechanism is used by the driver for Ethernet packet transmission in the workaround mode (see errata description).

The application can use the VSS transmission in parallel with the Ethernet, even when the Ethernet TX is configured to work in the workaround mode. Only the VSS transmit data path is impacted by the workaround implementation.

These are the limitations and modifications for VSS:

- The VSS TX threshold is not set by the application, but relates to *value_m*.
- In VSS TX callback, the VSS TX buffer to fill is not continuous but divided into equal sized segments. These segments uses the `osSioBufferGet()` and `osSioBufferPut()` SIO API functions.
- To allow pre-filling of the VSS transmit data before VSS is enabled, the application uses the `CPRI_VSS_ETHERNET_WA_BUFFER_FILL` SIO command control function.
- "The SIO `osSioChannelOpen()` function receives as an input structure `sio_ch_open_params_t`. The structure fields `buffer_size` and `buffer_data_size` uses the value provided by the driver. These values are fetched using the `CPRI_VSS_ETHERNET_WA_INFO_GET` SIO control command function.

4.7.8.2.2.1 VSS TX Callback Flow

The application uses the transmit VSS data generated by the following pseudo-code:

```
a. vss_segment = osSioBufferGet(&vss_tx_ch, &length);
b. if (vss_segment == NULL) return;
c. Fill buffer "vss_segment" of size "length" with data.
d. osSioBufferPut(&vss_tx_ch);
e. Goto (a)
```

4.7.8.3 Additional API

The additional APIs are as follows:

- Two new VSS SIO control commands:

```
#define CPRI_VSS_ETHERNET_WA_INFO_GET      0x00000001            /**< Returns
the buffer size and actual VSS data size, through structure
#cpri_ethernet_wa_vss_buffers_size.  */
#define CPRI_VSS_ETHERNET_WA_BUFFER_FILL   0x00000002            /**< Copies
the given buffer content to the VSS segments of the VSS buffer; As a
parameter, pointer to the source buffer is expected and the source buffer must
be of size at least as returned in #vss_data_size by command
#CPRI_VSS_ETHERNET_WA_INFO_GET.  */
```

- The VSS initialization structure:

```
typedef struct
{
.
.
.
    uint32_t                    eth_wa_mode;                /**< Active if
Ethernet WA mode (A_007968) is desired.*/
} cpri_vss_init_params_t;
```

- Ethernet initialization structure:

```
typedef struct
{
.
.
.
                                ,eth_wa_mode:1,                     /**<
Active if Ethernet WA mode (A_007968) is desired.*/
#if defined(CPRI_BLOCK_VER_4_2)
                                eth_wa_vss_buffer_steering_bits:3   /**< In WA
mode, steering bits of the VSS TX buffer, which contains both VSS and Ethernet
data.*/
#endif //CPRI_BLOCK_VER_4_2
.
.
.
    uint16_t                    eth_wa_value_m;                     /**< Only
if "eth_wa_mode" is enabled; value m as mentioned in Ethernet WA SRS; the TX
VSS buffer size will equal to the total of VSS&Ethernet data size of value_m
number of HF's */
    vss_bus_transaction_size_t  eth_wa_vss_transaction_size;        /**< In WA
mode, the VSS TX transaction size; if VSS is enabled, it must be identical to
the collateral parameter in VSS initialization parameters. */
    os_mem_type                 eth_wa_vss_buffer_heap;             /**< In WA
mode, the heap where the VSS TX buffer, which contains both VSS and Ethernet
data, will be allocated; If application enables VSS and the VSS is owned by a
different core to the Ethernet core, the heap must be SHARED. */
#if defined(CPRI_BLOCK_VER_4_2)
    cpri_buffer_attributes_t    eth_wa_vss_buffer_attributes;       /**< In WA
mode, buffer attributes of the VSS TX buffer, which contains both VSS and
Ethernet data. */
#endif //CPRI_BLOCK_VER_4_2
} cpri_ethernet_init_params_t;
```

- The structure of the output parameter of the control command:

  `CPRI_VSS_ETHERNET_WA_INFO_GET`

4.7.8.4 Considerations

- The higher *value_m* results in higher latency of the packets transmission, and the lower interrupts frequency.
- The packet are assembled through the software, resulting in the negatively impacted performance compared to the regular (non-workaround) mode.
- In the workaround mode, if Ethernet data type is used and not the VSS data type, the VSS TX DMA in hardware still remains active. Thus the hardware bus load needs to be considered.

4.7.8.5 Limitations

The initializing core for both the Ethernet and VSS data types must be the same.

4.7.9 Programming Model

4.7.9.1 General

CPRI IQ/VSS driver is implemented under the SIO abstraction layer.

- User calls SIO methods to indirectly activate CPRI driver methods.
- SIO methods receive IQ/VSS parameters; e.g., flags and structures.
- Parameters are passed to the LLD.

CPRI Ethernet/HDLC driver is implemented under the BIO abstraction layer.

- User calls BIO methods to indirectly activate CPRI driver methods.
- BIO methods receive Ethernet/HDLC parameters; e.g., flags and structures.
- Parameters are passed to the LLD.

4.7.9.2 Initialization Functions Hierarchy

```
osInitialize()
                                                        cpriInitialize()
            cpriIqInitialize()
            cpriVssInitialize()
            cpriEthernetInitialize()
            cpriHdlcInitialize()

appInit()  //application
```

```
osSioDeviceOpen()
            cpriIqOpen()
            cpriVssOpen()

osSioChannelOpen()
            cpriIqChannelOpen()
            cpriVssChannelOpen()

osBioDeviceOpen()
            cpriEthernetOpen()
            cpriHdlcOpen()

osBioChannelOpen()
            cpriEthernetChannelOpen()
            cpriHdlcChannelOpen()
```

4.7.9.3 Runtime Functions Hierarchy

```
appBackground()  //application

osSioDeviceCtrl()
            cpriIqCtrl
            cpriVssCtrl

osBioDeviceCtrl()
            cpriEthernetCtrl
            cpriHdlcCtrl
                                                    osBioChannelTx()
            cpriEthernetChannelTx
            cpriHdlcChannelTx
                                                    osBioChannelRx()
            cpriEthernetChannelRx
            cpriHdlcChannelRx

cpriCalculateDelays()
```

4.7.10 CPRI API

The CPRI driver is both a SIO (IQ/VSS) and BIO (Ethernet/HDLC) driver.

The driver's architecture-independent header files are located as follows:

- include\arch\peripherals\cpri\cpri.h
- initialization\arch\peripherals\cpri\include\cpri_init.h

- include\arch\peripherals\cpri\cpri_memmap.h – CPRI memory map

The architecture-specific header file is found as follows:

- initialization\arch\peripherals\cpri\include\arch\msc8157_family_cpri.h
- initialization\arch\peripherals\cpri\include\arch\psc9x32_family_cpri.h
- initialization\arch\peripherals\cpri\include\arch\b4860_family_cpri.h

4.7.10.1 Calling Sequence Example

Table 4.23 CPRI Driver API

Status	Name	Description
Kernel Bring-up	osInitialize() -> osArchDevicesInitialize()->cpriInitialize()	• General initialization of all CPRI units. • Allocation of global CPRI structure. • Initialization of interrupt scheme. • Calls to specific interface initialization functions. • Use of msc815x_config.c parameters for the following: – global parameters; e.g., link rates, number of CPRI units used. – unit specific parameters; e.g., slave/master modes, sync source, loopback, and Rx/Tx sample width. • Auto-negation execution (all phases).
	osInitialize() ->osArchDevicesInitialize()->cpriInitialize()->cpriIqInitialize()	• Initialization of IQ data type for a single CPRI unit. • Registration to SIO abstraction layer. • Use of IQ initialization parameters (msc815x_config.c) for data type initialization; including mapping configuration.
	osInitialize() ->osArchDevicesInitialize()->cpriInitialize()->cpriVssInitialize()	• Initialization of VSS data type for a single CPRI unit. • Registration to SIO abstraction layer. • Use of VSS initialization parameters (msc815x_config.c) for data type initialization.

Table 4.23 CPRI Driver API

Status	Name	Description
Kernel Bring-up, continued	osInitialize() -> osArchDevicesInitialize()-> cpriInitialize()-> cpriIEthernetInitialize()	• Initialization of Ethernet data type for a single CPRI unit. • Registration to BIO abstraction layer. • Use of Ethernet initialization parameters (msc815x_config.c) for data type initialization.
	osInitialize() -> osArchDevicesInitialize()-> cpriInitialize()-> cpriIHdlcInitialize()	• Initialization of HDLC data type for a single CPRI unit. • Registration to BIO abstraction layer. • Usage of HDLC initialization parameters (msc815x_config.c) for data type initialization.

Table 4.23 CPRI Driver API

Status	Name	Description
Application Bring-up	osSioDeviceOpen()	Call to cpriIqOpen or cpriVssOpen with user parameters.
	osSioDeviceOpen()-> cpriIqOpen()	• Open CPRI IQ device with user parameters. • Return handle to user.
	osSioDeviceOpen()-> cpriVssOpen()	• Open CPRI VSS device with user parameters. • Return handle to user.
	osSioChannelOpen()	• Open CPRI IQ or CPRI VSS channels. • Initialize SIO data structure.
	osSioChannelOpen()-> cpriIqChannelOpen	• Open CPRI IQ channel. • Configures channel buffer.
	osSioChannelOpen()-> cpriVssChannelOpen	• Open CPRI VSS channel. • Configures channel buffer.
	osBioDeviceOpen	Call to cpriEthernetOpen or cpriHdlcOpen with user parameters.
	cpriEthernetOpen()	• Open CPRI Ethernet device with user parameters. • Return handle to user.
	cpriHdlcOpen()	• Open CPRI HDLC device with user parameters. • Return handle to user.
Application Bring-up, continued	osBioChannelOpen()	• Open CPRI Ethernet or CPRI HDLC channels. • Initialize BIO data structure.
	osSioChannelOpen()-> cpriEthernetChannelOpen	• Open CPRI Ethernet channel. • Allocate and initialize channel BD ring.
	osSioChannelOpen()-> cpriHdlcChannelOpen	• Open CPRI HDLC channel. • Allocate and Initialize channel BD ring.

Table 4.23 CPRI Driver API

Status	Name	Description
Application Runtime	osSioDeviceCtrl()	Calls cpriIqCtrl or cpriVssCtrl.
	osSioDeviceCtrl()-> cpriIqCtrl	Executes one of the following commands: • IQ Tx enable/disable • IQ Rx enable/disable • Tx control table write/read • Rx control table read • BFN Rx counter read • HFN Rx counter read • Calculate delays • Set statistics • Activate reset request (master) • Reset requests enable • Check is reset / acknowledge was detected
	osSioDeviceCtrl ()-> cpriVssCtrl	Execute one of the following commands: • VSS Tx enable/disable • VSS Rx enable/disable
	osBioDeviceCtrl()	Calls cpriEthernetCtrl or cpriHdlcCtrl
	osBioDeviceCtrl()-> cpriEthernetCtrl	Execute one of the following commands: • Ethernet Tx enable/disable • Ethenret Rx enable/disable
	osBioDeviceCtrl()-> cpriEthernetChannelTx	Transmit an Ethernet packet.
Application Runtime, continued	osBioDeviceCtrl ()-> cpriHdlcCtrl	Execute one of the following commands: • HDLC Tx enable/disable • HDLC Rx enable/disable
	osBioDeviceCtrl()-> cpriHdlcChannelTx	Transmit an HDLC packet.

4.7.10.2 Functionality

The following section deals with operating data transactions over CPRI.

4.7.10.2.1 Initialization

Follow the below noted steps to initialize data transaction functionality over the eMSG driver:

Table 4.24 CPRI Initialization

Step	Status	Task	Comment
1	Mandatory	Enable CPRI unit support.	• Define MSC815X_CPRIn as ON in os_config.h. • n = number of CPRI units used.
2	Mandatory	Enable CPRI IQ support.	Define MSC815X_CPRIn_IQ as ON in os_config.h.
3	Optional	Enable CPRI VSS support.	Define MSC815X_VSSn_VSS as ON in os_config.h.
4	Optional	Enable CPRI Ethernet support.	Define MSC815X_CPRIn_ETHERNET as ON in os_config.h.
5	Optional	Enable CPRI HDLC support.	Define MSC815X_CPRIn_HDLC as ON in os_config.h.
6	Mandatory	Prepare cpri_global_init_params_t structure for each CPRI unit used.	Prepared in msc8x57_config.c.
7	Mandatory	Prepare cpri_iq_init_params_t structure for each CPRI unit used.	Prepared in msc8x57_config.c.
8	Optional	Prepare cpri_vss_init_params_t structure for each CPRI unit used.	Prepared in msc8x57_config.c.
9	Optional	Prepare cpri_ethernet_init_params_t structure for each CPRI unit used.	Prepared in msc8x57_config.c.
10	Optional	Prepare cpri_hdlc_init_params_t structure for each CPRI unit used.	Prepared in msc8x57_config.c.
11	Mandatory	Get cpri_iq handle.	Use osSioDeviceOpen().

Table 4.24 CPRI Initialization

Step	Status	Task	Comment
12	Optional	Get cpri_vss handle.	Use osSioDeviceOpen().
13	Optional	Get cpri_ethernet handle.	Use osBioDeviceOpen().
14	Optional	Get cpri_hdlc handle.	Use osBioDeviceOpen().
15	Mandatory	Open cpri_iq channels.	Use osSioChannelOpen.
16	Optional	Open cpri_vss channels.	Use osSioChannelOpen.
17	Optional	Open cpri_ethernet channels.	Use osBioChannelOpen.
18	Optional	Open cpri_hdlc channels.	Use osBioChannelOpen.
19	Optional	Wait for select CPRI frame to begin Tx/Rx.	Use osSioDeviceCtrl to read BFN/HFN counters.
20	Mandatory	Enable IQ Tx/Rx.	Use osSioDeviceCtrl.
21	Optional	Enable VSS Tx/Rx.	Use osSioDeviceCtrl.
22	Optional	Enable Ethernet Tx/Rx.	Use osBioDeviceCtrl.
23	Optional	Enable HDLC Tx/Rx.	Use osBioDeviceCtrl.

4.7.10.2.2 Runtime: IQ/VSS and Ethernet/HDLC Data

IQ and VSS are synchronic interfaces; they achieve periodic Tx/Rx thresholds and call corresponding interrupts according to buffer size and the user-defined division of buffer thresholds.

The application must handle runtime information in a manner sufficient to support the threshold rate; it can call osSioDeviceCtrl() to execute runtime commands. The following considerations ensure the application can handle data before it is overrun by the driver:

- Number of CPRI units used.
- Number of channels used in each block.
- Channel bandwidth.
- Size of Tx/Rx buffers used.

Tx and Rx characteristics include the following:

- Calls user-provided Tx callback when the Tx data threshold is reached.
- Callback handles writing new data to the Tx buffer for further Tx.
- During a Tx callback event the data can be written directly to the channel Tx buffer.
- Calls user-provided Rx callback when Rx data threshold is reached.
- Callback handles reading data received from the Rx buffer.
- During an Rx callback event the data can be read directly from the Rx buffer.

Ethernet and HDLC are asynchronic interfaces; frames can be transmitted only if data exists to be sent and received.

Transmit data as follows:

1. Call osFrameGet to get a frame.
2. Call osFrameBufferNew to get a buffer.
3. Fill data in the buffer.
4. Set the buffer to the frame using osFrameSingleBufferSet.
5. Transmit the frame using osBioChannelTx.
6. After sending the data, release the frame using osFrameRelease.

Receive data as follows:

1. Get a buffer pointer (from the frame) using osFrameSingleBufferGet.
2. Release the frame using osFrameRelease.

4.7.11 Demo Use Cases

- demos\starcore\msc815x\cpri_advanced_mapping_demo
- demos\starcore\msc815x\cpri_auxiliary_demo
- demos\starcore\msc815x\cpri_ethernet
- demos\starcore\msc815x\cpri_full_negotiation_demo
- demos\starcore\msc815x\cpri_hdlc
- demos\starcore\msc815x\cpri_multi_uni_cast
- demos\starcore\msc815x\cpri_multicore
- demos\starcore\msc815x\cpri_multicore_amcx
- demos\starcore\msc815x\ cpri_multicore_on_same_cpri
- demos\starcore\msc815x\ cpri_vss
- demos\starcore\b4860\cpri_ethernet_workaround_vss_and_ethernet\

4.8 HW_Timer32

4.8.1 Introduction

The HW_TIMER32 driver supplies API for initializing and controlling SOC 32-bit timers. The HW_TIMER32 API is designed to replace previous HW_TIMER API.

4.8.2 Features

HW_Timer32 features are noted below:

- SmartDSP OS driver compatibility.
- Quad initialization.
- HW independent driver:
  - Generic API.
  - Support for SOC-specific parameters.
- Support for the noted counting modes:
  - Rising edge.
  - Rising and fall edge.
  - Count rising primary source.
  - Count quadrature.
  - Count rising primary when secondary source supplies direction.
  - Cascaded.

4.8.2.1 Relevant SoC

MSC8157 and B4860 families.

4.8.3 Architecture

This section details HW_Timer32 driver components found in the SmartDSP OS; see Table 4.25.

Table 4.25 HW_Timer32 Components

Components	Description
Timer32Module	• Consists of four HW_TIMER32 components. • Share clock inputs and triggers. • Configuration is SOC-specific and is initialized in hwTimer32ModuleInitialize. • Enabled in os_config.h. • Initialization parameters are supplied by the application in xxx_config.c.
Timer32	• Operation is based on module configuration. • Application manages distribution when calling hwTimer32Open(). • Starts counting after hwTimer32Start.
Multiplexed Interrupts (B4860 only)	• Transparent to the application. • Initialized with hwTimer32Open(). • Consistent Interrupt Handling from an application perspective.

4.8.3.1 Design Decisions

Design decisions, along with their reasoning, are noted below.

1. HW timer peripheral is included in the driver library.
   - Shares the same concept as other peripherals supported by SmartDSP OS.
2. QUAD input and triggers.
   - HW_TIMER32_MODULEx is defined in os_config.h.
   - There is a separation between timer and module.
   - The module is configured during initialization with XXX_timer32_moduleX_params; the latter is supplied by the user in xxx_config.c.
3. SOCs and SOC-specific files.
   - Module initialization parameters are implemented in SOC-specific files.

4. HW_TIMER component in the OS library features a HW_TIMER API[1].
   - HW_TIMER is backward compatible as long as HW_TIMER32_MODULE is undefined in os_config.h[2].

4.8.4 Programming Model

This section covers driver bring-up as handled by HW_Timer32.

[1] Not in B4860 family

[2] Not in B4860 family

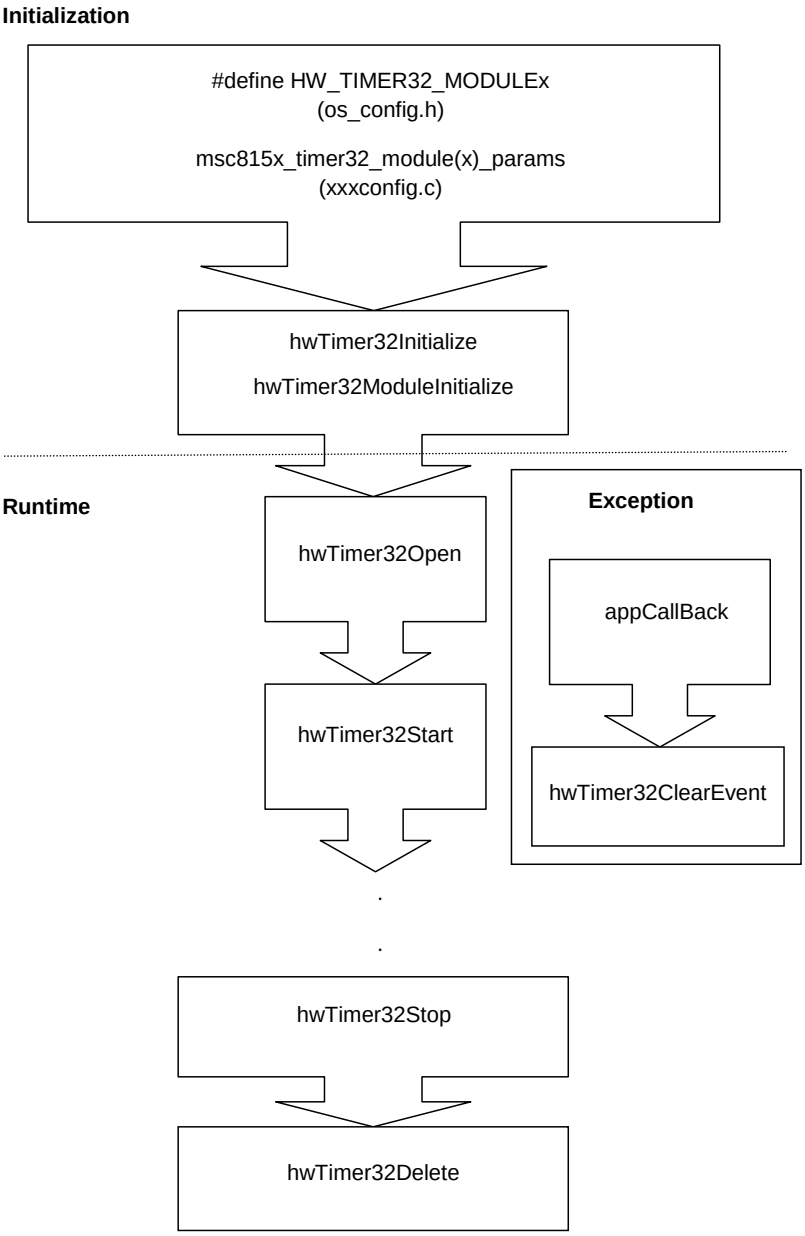
4.8.4.1 Calling Sequence Example

Table 4.26 HW_Timer32 Driver API

Status	Name	Description
Initialization	hwTimer32Initialize()	Initialization of global internal structures.
	hwTimer32ModuleInitialize()[1]	• Initializes the quad timer module. • Selects the source clock and trigger input.
	hwTimer32GroupInitialize()[2]	• Initializes all timers in a group. • Sets a trigger for all group trigger inputs.
Runtime	hwTimer32Open()	Opens a timer.
	hwTimer32Start()	Starts counting.
	hwTimer32Stop()	Stops a given timer.
	hwTimer32Delete()	Stops and deletes a given timer from the system.
	hwTimer32ValueGet()	Gets a snapshot of the timer counter value.
	hwTimer32HoldGet()	Gets the hold value of a given timer.
	hwTimer32CounterSet()	Sets a value for a given timer counter.
	hwTimer32CompareSet()	Sets compare values to a given timer.
	hwTimer32SetAndForget()	Locks timer until hwTimer32LockClear is called.
	hwTimer32LockClear()	Releases timer lock.
	hwTimer32GlobalConfig()	Configures global system timer.
	hwTimer32GlobalGet()	Returns global timer to the module.
	osHwTimerClearEvent()	Clears the event bit of the given timer.

1.Not in B4860 family.

2.Only in B4860 family.


Initialization
#define HW_TIMER32_MODULEx
(os_config.h)
msc815x_timer32_module(x)_params
(xxxconfig.c)
hwTimer32Initialize
hwTimer32ModuleInitialize
Runtime
hwTimer32Open
hwTimer32Start
Exception
appCallBack
hwTimer32ClearEvent
hwTimer32Stop
hwTimer32Delete

NOTE HW_TIMER32_MODULEx driver and the HW_TIMER OS component cannot coexist in the same application. Defining both in os_config.h will lead to compilation errors.[1]

4.8.4.2 Source Code

The below table lists source code and Library output.

Table 4.27 HW_Timer32 Source Code

Source Code	Directory
Common Code	• drivers\timers\rev1\hw_timers32.c • drivers\timers\rev1\hw_timers32_init.c
SOC-specific	initialization\arch\peripherals\timers
Library	• os_msc8157_drivers • os_b4860_drivers_family

4.8.5 Demo Use Cases

HW_Timer32 use cases are noted in the below table.

Table 4.28 HW_Timer32 Use Cases

Use Case	Directory
Simple multicore	demos\starcore\msc815x\hw_timers32
Counting CPRI events with HW_TIMER32	demos\starcore\msc815x\hw_timers32_cpri_count_demo

4.9 Antenna Interface Controller (AIC)

4.9.1 Introduction

The Antenna Interface Controller (AIC) driver supports radio antenna communication as part of a heterogeneous system—the AIC block is handled by both StarCore (SC) and an external master. SC

[1] Irrelevant to B4860 family.

maintains runtime data processing while the external master (PSC9131 and PSC9132 e500 power core) handles control and network configurations.

AIC implements ‘Synchronous Input Output’ (SIO) abstraction layer API and complies with SmartDSP OS driver and API rules.

4.9.2 Features

The AIC driver and external master have the below noted features.

The AIC driver has the following features:

- Supports ADI RFIC.
- Multiple AIC ADI lanes that can be activated.
  - Each lane transmits and/or receives IQ data via a maximum of two antennas.
- Three possible Rx data paths:
  - Unicast to SoC
  - Unicast to MAPLE
  - Multi-unicast to SoC and MAPLE
- Symbol Tx or Rx interrupt can target core and/or MAPLE.
- Sniffing allows IQ data to be captured in order to identify and analyze an environment.

The external master controls the following features:

- Supports (and was tested in) LTE-FDD and LTE-TDD modes.
- Supports LTE-FDD 5, 10, 15, and 20MHZ bandwidth modes.

4.9.2.1 Relevant SoC

AIC driver supports the PSC9131 and PSC9132 AIC blocks.

4.9.3 Architecture

This section covers the SW architecture of the AIC driver.

4.9.3.1 Components

The AIC driver has multiple SIO devices that form hierarchical relations, from highest to lowest:

1. AIC
2. ADI

3. ADI lanes; e.g., PSC9131 with three lanes.

It is only possible to open an SIO AIC device if a higher hierarchy is already open; e.g., ADI if AIC is open. The driver enforces the hierarchy by setting AIC and ADI handlers as parameters in the lower-hierarchy device (ADI lanes).

See Figure 4.8 for the AIC driver hierarchical structure.

Figure 4.8 AIC Driver Components and Environment

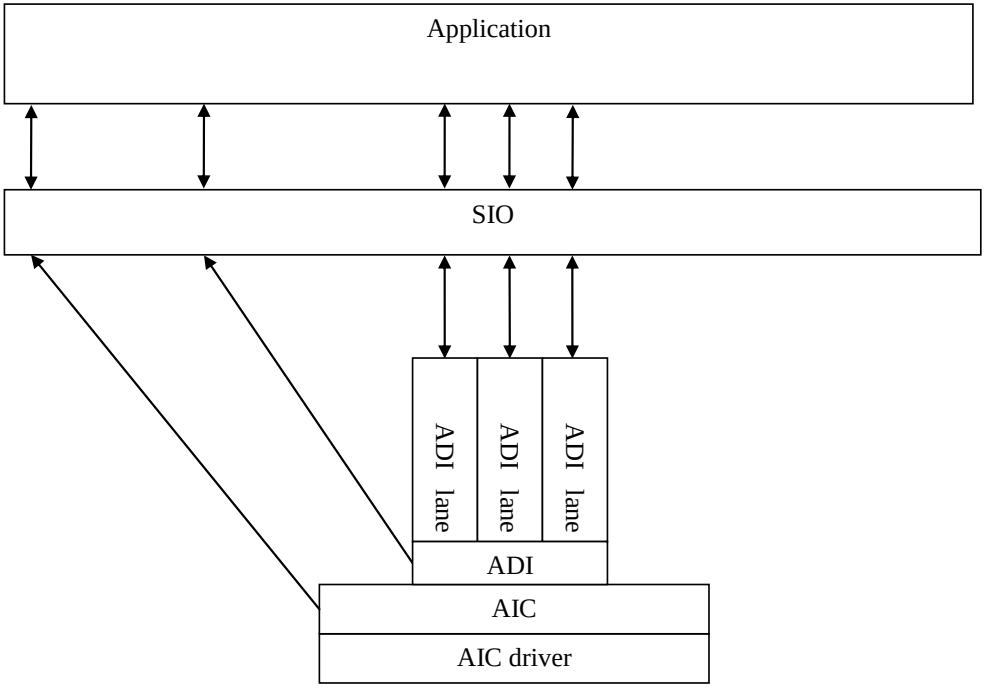


Table 4.29 AIC Driver Objects

Components	Description
aic_init_params_t	• Global initialization AIC block driver parameters for `aicInitialize()`. • Content is provided per AIC block. • Content is known during compilation.
aic_open_params_t	• SIO AIC device open parameters. • Empty and used as a place-holder for future enhancements.
aic_adi_open_params_t	• SIO AIC ADI device open parameters. • Provides a list of driver-allocated ADI lanes. • Holds the handler of its related SIO AIC device.

Table 4.29 AIC Driver Objects

Components	Description
aic_adi_lane_open_params_t	• SIO AIC ADI lane device open parameters. • Provides AIC ADI lane configurations—buffer size, number of antennas, sniffing mode, etc. • Holds the handler of its related SIO AIC ADI device.
aic_channel_open_params_t	• SIO AIC ADI lane channel open parameters. • Routes Rx & Tx interrupts to MAPLE and/or SC.
aic_error_cb_params_t	• Error callback input parameter. • Holds lane number and error events.
aic_int_cb_params_t	• Rx\Tx callback input parameter. • Holds lane and channel (antenna) number.
aic_sniff_open_params_t	• SNIFF open parameters. • Supplied as part of AIC ADI lane opening or reconfiguration.

4.9.3.2 Design Decisions

AIC block lanes are not completely independent as global configurations can be applied to all or only part of a device; e.g., AIC ADI and the currently unsupported AIC MAXIM sub-block.

Architectural factors are addressed by driver hierarchical design:

- Application feeds AIC-global configurations when opening an SIO AIC device.
- Application feeds AIC-ADI-global configurations when opening an SIO AIC ADI device.
- Driver forces the user to open and configure a device; however, this can only be done after the respective higher-hierarchy device has been opened. The user must provide the handler of the higher-hierarchy device.

4.9.3.3 Initialization Flow

The below table outlines a general initialization sequence.

Table 4.30 AIC Initialization Sequence

Function	Description
`aicInitialize()`	Called by OS.
`osSioDeviceOpen()`	Opens SIO AIC device.
`osSioDeviceOpen()`	Opens SIO AIC ADI device.
`osSioDeviceOpen()`	Opens SIO AIC lane device for each lane used by the application.
`osSioChannelOpen()`	Opens lane channels used by the application.

4.9.4 Data Flow

Data receive/transmit begins after user activates Rx/Tx AIC DMA through call to `osSioDeviceCtrl()`.

4.9.4.1 Transmit Flow

User-provided Tx callback is used when a symbol is transmitted; it is also responsible for accessing buffer data. Buffer management is handled by the application. The application provides both a buffer address and size in `aic_adi_lane_open_params_t`.

The channel—configured via `aic_channel_open_params_t`— must interrupt SC in order to invoke a callback. Each channel corresponds to a lane antenna.

4.9.4.2 Receive Flow

The Rx flow is the same as the Tx flow except for the addition of a MAPLE data path. If data is transferred to MAPLE then MAPLE is responsible for buffer management. The ACK mechanism synchronizes between MAPLE and AIC.

4.9.4.3 Sniffing

Similar to Rx except that no channels are opened. Sniffing parameters, `aic_adi_lane_open_params_t`, are set in a sniffing-dedicated structure. Upon completion of the sniffing capture, the core is triggered by a Sniffer capture-complete interrupt.

4.9.5 Programming Model

4.9.5.1 General

All AIC API, except `aicInitialize()`, are called via SIO API. The below table outlines a typical calling sequence that uses two antennas on a single AIC lane.

Table 4.31 Driver Functions

Function	Description
`aicInitialize()`	Initiates AIC.
`osSioDeviceOpen(AIC_DEV_NAME0)`	Opens AIC SIO device.
`osSioDeviceOpen(AIC_ADI_DEV_NAME0)`	Opens AIC ADI SIO device.
`osSioDeviceOpen(AIC_ADI_DEV_NAME3)`	Opens AIC ADI lane 3 SIO device.
`osSioChannelOpen(WRITE, 0)`	Opens channel 0 (antenna 1) for Tx.
`osSioChannelOpen(READ, 0)`	Opens channel 0 (antenna 1) for Rx.
`osSioChannelOpen(WRITE, 1)`	Opens channel 1 (antenna 2) for Tx.
`osSioDeviceCtrl(SIO_DEVICE_RX_TX_ENABLE)`	Activates lane 3 Rx and Tx pipes.
—	AIC finishes reading Tx symbol from system buffer.
—	Calls application's Tx callback.
—	AIC finishes writing Rx symbol to system buffer.
—	Calls application's Rx callback.

4.9.5.2 Calling Sequence Example

Table 4.32 AIC Driver API

Status	Name	Description
OS Initialization	osInitialize() –> osArchDevicesInitialize() –> aicInitialize()	• osInitialize(), not the application, calls this function as per *os_config.h* parameters and *psc9x3x_config.c* configurations. • Allocates and initializes AIC internal global structures.
Application Initialization	osSioDeviceOpen() –> aicOpen()	• Empty function; a place-holder for future extensions. • Must be called before opening SIO ADI device.
	osSioDeviceOpen() –> aicAdiOpen()	• Initializes AIC ADI. • Allocates driver internal structures for a given number of lanes.
	osSioDeviceOpen() –> aicAdiLaneOpen()	• Initializes AIC ADI lane. • Configures registers corresponding to a given lane. • Registers the core to AIC interrupts. • Can assign the lane for sniffing.
	osSioChannelOpen() –> aicAdiLaneChannelOpen()	• Initializes AIC channel. • Enables desired interrupts. • Sets a buffer.

Table 4.32 AIC Driver API

Status	Name	Description
Control	osSioDeviceCtrl() –> aicAdiLaneCtrl()	• Performs device control commands: – Enable/disable AIC ADI lane Rx/Tx. – Fast “unsafe” version for enabling/disabling Rx/Tx of AIC ADI lane; action is immediately reversed. – Lane reconfiguration. – Polls completion of Sniffer capture.
Runtime	Application given Rx / Tx callback.	• If core interrupt is enabled and a callback is provided then: – Callback is used upon completion of a transfer– of a symbol or user-defined number of bytes– between AIC and a system. – Or, at completion of a Sniffer capture. • Callback handles the data. – Identifies the channel (antenna) that caused the interrupt or the Sniffer ID. – Application provides user callbacks to the driver via the SIO abstraction layer.
Runtime	Application given UL_OFF/ DL_OFF callback.	• In LTE-TDD users can enable interrupts and supply callbacks for these events.
Exception	Application given error callback.	• User-enabled error interrupt will call an error callback in the event of under/over flow and JESD time-out errors. • Callback parameters: – Holds the channel within which the error originated. – Holds associated error events.

4.9.5.3 Functionality

The following section covers AIC functionality and precautions.

The AIC driver application—in data management role—provides buffer and maximum symbol (or threshold) sizes.

The AIC, following a symbol (or threshold) size transfer, transfers to\from a buffer address (previous_address + size of last transferred data).

The buffer is cyclic; i.e., after completing\exhausting a buffer the AIC repeats the process in its entirety.

Table 4.33 AIC Functionality

Stage	Description
Configure SmartDSP OS to support required SIO AIC devices.	• Edit *os_config.h*. • Set AIC and AIC_ADI to ON. • [Optional] Set ADI1, ADI2, ADI3, and ADI4 to ON.
Rx data path is defined in aic_adi_lane_open_params_t. mode_select_flag.	If MAPLE is an Rx target then provide the following: • aic_adi_lane_open_params_t.rx_maple_base_address • aic_adi_lane_open_params_t.rx_maple_buffer_size
Defines the number of lane antennas. Indicates if AIC can expect ACK before transferring the next symbol/thresholdl.	aic_adi_lane_open_params_t.mode_select_flag
Application will set if the symbol/threshold completion interrupt triggers as follows: • core or MAPLE • core and MAPLE • neither	Achieved at the time of opening a channel via aic_channel_open_params_t.interrupt_mode.

Table 4.33 AIC Functionality

Stage	Description
Each lane works in either normal or sniffing mode.	For sniffing, the following structure is needed: aic_adi_lane_open_params.aic_sniff_open_params
Switch lanes from normal to sniffing mode or visa-versa.	• Disable the lane via the SIO control command, aic_adi_lane_open_params. • Reconfigure the lane. • Call the SIO reconfiguration control command. • Note! A lane (or any SIO device) can only be opened once.

4.9.5.4 Precautions

The following is a list of AIC-related precautions.

- AIC lane and channel parameters must obey aic_adi_lane_open_params.rx_buffer_size == sio_ch_param.buffer_size * sio_ch_param.num_of_buffers. The same applies to Tx pipe.
  - AIC lane parameter: aic_adi_lane_open_params.rx_buffer_size
  - AIC channel parameters: sio_ch_param.buffer_size and sio_ch_param.num_of_buffers.
- All AIC configurations, including channel opening, must be executed when the lane is inactive.
- When configuring a lane to sniffing mode,
  - aic_adi_lane_open_params.aic_sniff_open_params must point to a valid configured structure; and,
  - aic_adi_lane_open_params.mode_select_flag must be set to zero.
- When configuring a lane to normal mode,
  - aic_adi_lane_open_params.aic_sniff_open_params must be set to NULL.

4.9.5.5 Source Code

Header files:

- OS initialization API: initialization\arch\peripherals\aic\include\aic_init.h
- AIC memory map: include\arch\peripherals\aic\aic_memmap.h
- Remaining API: include\arch\peripherals\aic\aic.h

Architecture-specific header files:

- initialization\arch\peripherals\aic\include\arch\psc9131_aic.h
- initialization\arch\peripherals\aic\include\arch\psc9132_aic.h

4.9.5.6 Resource Management

The AIC driver application—in data management role—provides buffer and maximum symbol sizes; in Stream Mode it provides threshold.

The AIC, following a symbol transfer, transfers to\from a buffer address (previous_address + maximum_symbol_size).

The buffer is cyclic; i.e., after completing\exhausting a buffer the AIC repeats the process in its entirety.

4.9.5.7 Demo Use Cases

demos\starcore\psc9x3x\aic_loopback\

4.9.6 Appendix: References

An AIC chapter is found in the PSC9131 and PSC9132 Reference Manuals.

4.10 SmartDSP OS Recovery Support

SmartDSP OS Recovery Support enables the SmartDSP OS application to be restarted without having to reset the entire PSC913x device.

4.10.1 Features

Recovery Support features are noted below.

- DSP image reload—for the same image reload—does not require resetting the PSC913x device.
- Checksum routine identifies recovery program corruptions.
- Following the recovery procedure,
  a. PPC, using non-maskable virtual interrupt, triggers recovery activation.
  b. Upon completion of the recovery procedure, DSP notifies PPC by setting the GCR DSP_READY bit.
  c. PPC reloads the DSP image.
  d. PPC notifies DSP by setting the GCR PPC_READY bit.
  e. PPC_READY bit triggers the DSP core to jump to the entry point.

f. SmartDSP OS restarts.

g. Prior to the PPC_READY indication, PPC must ensure correct initialization of the shared heterogeneous area.

- Support limitations following recovery:
  - TDM, CLASS, and DMA are not part of the recovery process.
  - WDT, once enabled, should be constantly served. During recovery, the WDT expire count is updated to maximum. However, users must ensure that WDT is constantly served during and after DSP image reload.

4.10.2 Architecture

PSC9131 software architecture is supported by SmartDSP OS Recovery Support.

4.10.2.1 Driver Components

This section details SmartDSP OS Recovery Support driver components as found in the SmartDSP OS. See Table 4.34 for further details.

Table 4.34 SmartDSP OS Recovery Components

Components	Description
Initialization	• Activate prior to recovery. • Finds available virtual interrupts, and creates and initializes MMU debug hooks.
Recovery	Activate inside the virtual interrupt callback function to start the recovery flow.
Handshake with PPC	• Activate when the Recovery Support component—found inside the virutal interrupt callback function—is successful. • Handles handshake between DSP and PPC in a manner similar to that occuring during the boot flow.
Recovery checksum	Performs checksum on the Recovery Support component.

4.10.2.2 Design Decisions

Design decisions, along with their reasoning, are noted below.

1. Select parts of Recovery use existing SmartDSP OS driver API:
   - By not writing directly to the registers then maintenance, following a SmartDSP OS component update, is minimized.
2. Select parts of Recovery DO NOT use existing SmartDSP OS driver API:
   - By writing directly to the registers, driver overhead and the number of function calls are minimized.
   - This is preferred for flat Recovery checksum calculations; the latter don't take function calls into consideration.
3. Use separate functions for Recovery and for handshake with PPC.
   - If Recovery fails then DSP must notify PPC to trigger device reset.
   - Allows user flexibility in implementing their own PPC and DSP handshake.

4.10.3 Data Flow

This section outlines the runtime data flow.

1. PPC triggers virtual NMI on DSP side.
2. DSP side activates Recovery from within ISR.
3. If Recovery succeeds:
   - DSP activates Handshake with PPC and sets DSP_READY.
   - DSP waits for PPC_READY and then jumps to BOOT_JMP_ADDR.
4. If Recovery fails then DSP notifies PPC and waits for device reset.

4.10.4 Programming Model

This section covers driver bring-up—first by the OS, then by the application-chosen API. Table 4.35 describes both architecture-independent and dependent APIs.

Table 4.35 SmartDSP OS Recovery Support APIs

Component	Functions	Description
Initialization	psc9x3xOsRecoveryInit()	• Recovery functionality. • Use inside appInit.
Recovery	psc9x3xOsRecovery()	• No device reset before DSP image reboot. • Activated in function, recovery_cb which is passed to psc9x3xOsRecoveryInit().

Table 4.35 SmartDSP OS Recovery Support APIs

Component	Functions	Description
Handshake with PPC	psc9x3xOsRecoveryReady()	• Activate following psc9x3xOsRecovery() in recovery_cb. • Handles—as during real boot flow—the handshake between PA and DSP. • Set DSP_READY poll on PPC_READY then jump to BOOT_JMP_ADDR. • PPC must ensure that shared heterogeneous space is cleared and correctly set before PPC_READY bit is set. • Disable SC MMU before the jump to BOOT_JMP_ADDR.
Recovery Checksum	psc9x3xOsRecoveryCheck()	Performs checksum from psc9x3xOsRecovery to psc9x3xOsRecovery_end; it must be defined inside the application linker file.

4.10.4.1 Calling Sequence Example

Table 4.36 describes functions included in the architecture-dependent API of SmartDSP OS Recovery Support.

Table 4.36 Architecture-Dependent API

State	Function	Description
Kernel Bring-up	–	N/A
Application Bring-up	`psc9x3xOsRecoveryInit()`	Initializes recovery.
	`psc9x3xOsRecoveryCheck()`	• Calculates recovery checksum. • Saves for comparison.

Table 4.36 Architecture-Dependent API

State	Function	Description
Application Runtime	`psc9x3xOsRecoveryCheck()`	• Calculates recovery checksum. • Compares with previous calculation to ensure recovery has not been corrupted.
	`psc9x3xOsRecovery()`	Activates recovery to prepare the system for DSP image reload.
	`psc9x3xOsRecoveryReady()`	• Notifies PPC of recovery completion. • Waits for DSP image reload.

4.10.4.2 Functionality

4.10.4.2.1 Initialization

Follow these steps to initialize SmartDSP OS Recovery Support:

1. Activate psc9x3xOsRecoveryInit() and get virtual interrupt handler.
2. Notify PPC regarding the number of virtual interrupts.

4.10.4.2.2 Runtime

Follow these steps to run SmartDSP OS Recovery Support.

1. Activate psc9x3xOsRecovery() inside the virtual interrupt callback function.
2. If psc9x3xOsRecovery() returns OS_FAIL then notify PPC of the recovery failure.
3. If psc9x3xOsRecovery() returns OS_SUCCESS then activate psc9x3xOsRecoveryReady().
4. Success or failure:
   - If successful then there is no psc9x3xOsRecoveryReady() return.
   - If the function is returned then notify PPC about the handshake failure.

4.10.4.3 Source Code

SmartDSP OS Recovery Support header files: include\arch\starcore\psc9x3x\psc9x3x_recovery.h

4.10.5 Demo Use Cases

SmartDSP OS Recovery Support demos:

- `demos\starcore\psc9x3x\recovery_demo`
- `demos\starcore\psc9x3x\aic_loopback (recovery target)`

4.11 Enhanced Serial Peripheral Interface (eSPI)

This section describes the Enhanced Serial Peripheral Interface (eSPI) for SmartDSP operating system.

4.11.1 Introduction

The Enhanced Serial Peripheral Interface (eSPI) driver supports all eSPI module features. It is used to exchange data with peripheral devices such as EEPROM, FLASH and RF cards.

4.11.2 Features

All eSPI module features are supported. In addition, the driver features the following:

- Interrupt and polling mode
- Receive and transmit data which is bigger than the physical FIFO size
- Non-blocking calls except a condition in which it is impossible (see details in 4.11.3.2 Design Decisions)

4.11.2.1 Relevant SoC

The SoC supporting eSPI modules are:

- 9131 - 4 eSPI modules
- 9132 - 2 eSPI modules

4.11.3 Architecture

This section covers the SW architecture of the AIC driver. The driver (aka LLD) is accessed through the CIO abstraction layer.

4.11.3.1 Components

Table 4.37 Components

Component	Description
`spi_init_params_t`	Global initialization eSPI driver parameters for `spiInitialize()`. • Per eSPI module. • Content is known in compile time.
`spi_cs_params_t`	Holds Chip Select parameters. • Defines a single Chip select. • `spi_init_params_t` holds an array of `spi_cs_params_t` – one for each Chip Select. • Also a parameter to CIO device control command `SPI_CMD_SET_CS_PARAMS`.
`spi_open_params_t`	CIO device open LLD parameters • Empty and used as a place-holder for future enhancements.
`spi_channel_params_t`	CIO channel open LLD parameters • Empty and used as a place-holder for future enhancements.

4.11.3.2 Design Decisions

Considering the following HW constraint:

- RX FIFO and TX FIFO buffers are limited (e.g. 32 bytes in 913x)

And in addition to features listed in 4.11.2 Features, the driver is designed to:

- Best performance
- Provide user with flexibility

In order to allow non-blocking operation (i.e. functions which do not wait for hardware before returning to application), the eSPI transmission is divided to 2 calls: **issuing** and **confirming**. Between these 2 calls the execution is prompted back to the application.

The eSPI transmission must be confirmed before attempting to issue another one.

A limitation on this non-blocking approach may be encountered if a bigger-than-FIFO-size eSPI transmission is **issued** by the application while working in polling mode, it is impossible to return

immediately from the **issuing** call. Since there are no interrupts, the driver cannot ensure data shifting between the buffers and the FIFO's to avoid overrun and under-run.

Thus in that case the **issuing** returns as soon as the remainder of data is shifted between buffers and FIFO's.

The driver is flexible. The application may or may not provide callbacks. The application may receive data and free the CIO buffer either inside the RX callback or after returning from the **confirmation** call. All flows are detailed in next topic.

4.11.4 Data Flow

Derived from eSPI HW specification, there are no separate activations of TX and RX channels. There is single eSPI transmission session activation.

Prior to issuing an eSPI session, the application shall specify transaction length (number of characters) and RX_SKIP value, which refers to number of characters to skip in the RX data path from the beginning of the transaction.

4.11.4.1 Transmit Flow

The application fills-in the assigned TX buffer and issues an eSPI transmission session. The HW eSPI transmission session begins.

If the data to transmit is bigger than the TX FIFO size, the data will be shifted periodically each time the TX FIFO is half empty, until completion of the session.

If interrupt mode is enabled, the data shifting takes place in ISR. Otherwise, i.e. in polling mode, the issue call is blocking, which means it returns after remainder of data is shifted into the TX FIFO.

When the session is over, application-given TX callback, if provided, is invoked.

If interrupt mode is enabled the callback is called asynchronously. Otherwise, in polling mode, the callback is invoked by the driver following application calling the confirmation function.

In either mode, and regardless of the callback provided or not, the confirmation function returns TRUE. Before end of session the confirmation function returns FALSE.

4.11.4.2 Receive Flow

If RX_SKIP parameter is set to a value smaller than transaction length, it means the RX data path will be active at least part of the next eSPI session.

When an eSPI session is issued, the HW eSPI transmission session begins.

If the data to receive is bigger than the RX FIFO size, the data will be shifted periodically from RX FIFO to a buffer, each time the RF FIFO is half full.

If interrupt mode is enabled, the data shifting takes place in ISR. Otherwise, i.e. in polling mode, the issue call is blocking, which means it returns after remainder of data is shifted from the RX FIFO.

When the session is over, application-given RX callback, if provided, is invoked.

If interrupt mode is enabled the callback is called asynchronously. Otherwise, i.e. in polling mode, the callback is invoked by the driver following application calling the confirmation function.

In either mode, and regardless of the callback provided or not, the confirmation function returns TRUE. Before end of session the confirmation function returns FALSE.

4.11.5 Programming Model

This section describes the programming model for eSPI driver.

4.11.5.1 General

All eSPI API functions, except `spiInitialize()`, are called via CIO API. Table 4.38 lists all the functions by phase.

Table 4.38 eSPI API functions

Phase	Name	Description
OS Initialization	`osInitialize()`→ `osArchDevicesInitialize()`→ `spiInitialize()`	OS initialization, not the application, invokes this function as per *`os_config.h`* definitions and *`psc9x3x_config.c`* configurations. Initializes driver internal structures and registers the core to corresponding eSPI interrupt line (if interrupt mode is enabled). Partial eSPI HW configurations.
Application Initialization	`osCioDeviceOpen()`→`spiOpen()`	Almost empty. Mainly a place-holder for possible future modifications.
	`osCioChannelOpen()`→`spiChannelOpen()`	Internal driver channel initialization.

Table 4.38 eSPI API functions

Phase	Name	Description
Control	`osCioDeviceCtrl()`→`spiCtrl()`	Performs device control commands: • Enable/Disable eSPI device. • Set Chip Select settings. • General SPI command setting. • Enable/Disable loopback mode. • Polling on eSPI status (busy in middle of command execution, or available for issuing a new command). This is the **confirmation** call. • Set `RX_SKIP` parameter.
Runtime	`osCioChannelTxBufferPut()`→ `spiChannelTx()`	**Issues** an eSPI transmission session. In polling mode, with bigger-than-FIFO size of data, the call returns after the remainder of data is shifted to/from FIFO's. Otherwise this is a non-blocking call.
	RX/TX callbacks, if provided by application	In interrupt mode, these callbacks are invoked when the eSPI session is complete. This is a HW triggered invocation. In polling mode, these callbacks are invoked when the following 2 conditions apply: The eSPI session is complete. Application called the confirmation command, i.e. spiCtrl() with command `SPI_CMD_POLL`, for the 1st time since eSPI session completion. Note that, any subsequent call to the confirmation command will not result in callbacks call. The Confirmation command though returns `OS_SUCCESS` since the eSPI sessions is complete.

4.11.5.2 Example Calling Sequence

This section lists the examples calling sequence for eSPI driver.

4.11.5.2.1 Application initialization

To initialize the application:

1. Open device and get a handler.

   ```
   spi_device = osCioDeviceOpen(SPI0_DEVICE_NAME, spi_open_params_t);
   ```

2. Open read channel using the device handler.

   ```
   osCioChannelOpen(spi_device, &spi_rx_channel, CIO_READ, );
   ```

   `spi_rx_channel` is a returned channel handler.

3. Open write channel using the device handler.

   ```
   osCioChannelOpen(spi_device, &spi_tx_channel, CIO_WRITE, );
   ```

   `spi_tx_channel` is a returned channel handler.

4. Enable eSPI device.

   ```
   osCioDeviceCtrl(spi_device, SPI_CMD_ENABLE, NULL);
   ```

4.11.5.2.2 Application runtime

To run the application:

1. Confirm previous eSPI session is complete before you issue a new one.

   ```
   while (osCioDeviceCtrl(spi_device, SPI_CMD_POLL, NULL) !=
   OS_SUCCESS) {}
   ```

2. Set RX_SKIP parameter.

   ```
   uint16_t skip = 4;
   osCioDeviceCtrl(spi_device, SPI_CMD_SET_SKIP_PARAM, (void *)&skip);
   ```

3. Get TX buffer of size 24 bytes. The 2nd parameter defines the size of the TX buffer as well as the length of the next eSPI transaction (in this case – 24 bytes).

   ```
   uint8_t  *tx_space;
   tx_space = osCioChannelBufferGet(&spi_tx_channel, 4 + 20);
   ```

   Application shall fill the buffer with data to be sent.

4. Issue eSPI session.

   ```
   osCioChannelTxBufferPut(&spi_tx_channel);
   ```

5. Inside RX callback or after confirming as in 1st bullet, we shall fetch the received data from CIO.

   ```
   uint16_t data_size = 0;
   uint8_t *rx_space = osCioChannelRxBufferGet(rx_channel, &data_size);
   ```

   `rx_space` points to the received data buffer.
6. After consuming the received data, clear the RX buffer.

   ```
   osCioChannelRxBufferFree(rx_channel, data_size);
   ```

4.11.5.3 Precautions

Following are the precautions for eSPI programming modules:

- The application is allowed to issue another eSPI command (ie another data transmission) only after the previous command is confirmed, otherwise behavior is undefined.
- Part of settings update (such as Chip Select parameters) is allowed only when the device is not busy (ie in middle of transaction).

4.11.5.4 Source Code

- External header files:
  - `drivers\spi\include\spi_init.h`
  - `drivers\spi\include\spi.h`
- Internal header files:
  - `drivers\spi\include\spi_init_.h`
  - `drivers\spi\include\spi_.h`
  - `drivers\spi\include\spi_shared_.h`
- Architecture specific internal header files:
  - `drivers\spi\include\device_specific\spi_9131_.h`
  - `drivers\spi\include\device_specific\spi_9132_.h`
- C source files:
  - `drivers\spi\spi_init.c`
  - `drivers\spi\spi.c`
- Architecture specific C source files:
  - `drivers\spi\device_specific\spi_9131.c`
  - `drivers\spi\device_specific\spi_9132.c`

4.11.6 Resource Management

The buffers of RX and TX are managed by the CIO abstraction layer.

The application fetches the received buffer using the CIO API. The application is also responsible, using the CIO API, to free the buffer. See example in 4.11.5.2.2 Application runtime.

The application gets the TX buffer from CIO. The driver is freeing the buffer after exhausting it.

4.11.7 Demo Use Cases

SmartDSP OS eSPI use case demo:

`demos\starcore\psc9x3x\spi_demo\`

4.11.8 Appendix: References

For more information about SmartDSP OS eSPI driver, refer to the *eSPI* chapter in 9131 and 9132 chip Reference Manuals.

A

Using C++ with SmartDSP OS

To use C++ in SmartDSP OS, make sure that all the C projects are modified to compile and link under C++. When compiled under C++, your application is able to use all external header files in C++ without interfering with the SmartDSP OS files.

NOTE Use SmartDSP OS linker files to enable C++ support when your application uses SmartDSP OS stationery.

To modify SmartDSP OS linker files do the following steps:

1. To define `ENABLE_EXCEPTION`, add `ENABLE_EXCEPTION=0x1` in the LCF file or add `Xlnk -DENABLE_EXCEPTION=0x1` in the linker command line.

   Enables exception handling to work properly.

2. To support C++ symbols, add the following lines to `local_map_link.l3k`:

```
_cpp_staticinit_start= originof(".staticinit");
_cpp_staticinit_end= endof(".staticinit");
__exception_table_start__ = (ENABLE_EXCEPTION)
?originof(".exception_index"):0;
__exception_table_end__ = (ENABLE_EXCEPTION)
?endof(".exception_index"):0;
```

3. Add `.exception` segment to `local_map_link.l3k`, see Listing A.1.

Listing A.1 Adding Support for Exception Handling

```
SECTIONS {
descriptor_local_data {
.oskernel_local_data
.data
ramsp_0
.oskernel_rom
.rom
.exception
```

```
.exception_index
……..
```

4. Add `.unlikely` segment to `os_msc815x_link.l3k`, see Listing A.2.

Listing A.2 Adding .unlikely Segment

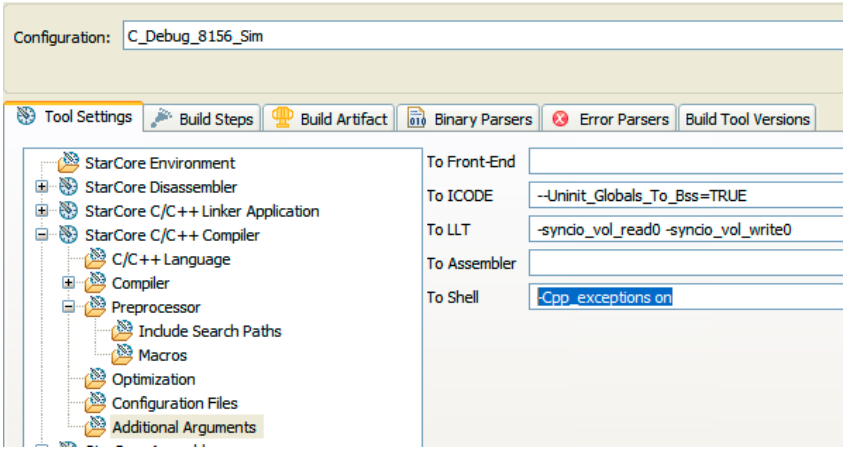
```
descriptor_os_kernel_text {
.osvecb
.oskernel_text_run_time
.oskernel_text_run_time_critical
.oskernel_text_initialization
.unlikely
.text
.private_load_text
.default
.intvec
.......
```

5. Enable the compiler rtlib at `os_msc815x_link.l3k`:

   `#define USING_RTLIB 1`

6. To enable exception handling, type `-Cpp_exceptions on` in the To Shell field, see Figure A.1.

Figure A.1 Properties Window—Tool Settings Page



7. For more information on enabling exception handling, refer to *How to Compile C Source Files* section of the *StarCore C-C++ Compiler User Guide*.

Index