Vision SDK OAL Guide

ABSTRACT:
The document describes main traits of Operation System Abstraction Layer (OAL) of the Vision SDK.
KEYWORDS:
OAL, CMA, Linux
APPROVED:

Revision History

VERSION	DATE	AUTHOR	CHANGE DESCRIPTION
0.5	24-February-16	Rostislav Hulik	First draft
0.6	6-April-16	Rostislav Hulik	Examples added
0.6a	16-June-16	Rostislav Hulik	Memory Invalidate exception for some platforms documented
0.7	11-November-16	Rostislav Hulik	Updated to OAL VSDK 0.9.7
1.0	9-May-17	Rostislav Hulik	1.0 version released and reviewed
1.1	15-May-17	Rostislav Hulik	OAL Initialize calls removed
1.2	25-July-17	Rostislav Hulik	OAL lock warning added

Table of Contents

1 Introduction

Operation System Abstraction Layer is a library which aims to provide portable OS-specific functions to be used in Vision SDK applications. The functions such as memory allocation, mutex mechanisms, task support and others are provided by the OAL in order to abstract the OS approach and provide common interface to the application.

To provide support of an Operating System for the vSDK the underlying library shall be implemented for the particular OS.

1.1 Purpose

The purpose of this document is to present the Vision SDK OAL API and help users to understand the main concepts of this API library.

1.2 Audience Description

This document is targeted at S32V234 Vision SDK users and OAL library developers who intend to port for an OS.

1.3 Document Location

s32v234_sdk/docs/vsdk

2 OAL Library Concept

2.1 Single oal.h header

The main concept of the OAL library is to provide a single header named **oal.h** which is included to every application without taking into account the target platform (OS). The file consists of single interface to the OAL functions.

In order to compile the application for a specific OS, OAL has a standard Vision SDK build tree which links a correct platform .a library to the application.

Depending on the Operating System dedicated kernel modules may have to be loaded as well to enable specific OAL functions. For those specifics please refer to the related chapter in this document.

The installation and use is described in following chapters.

2.2 Interface overview

In this section, a brief overview of provided functions follows.

`OAL Contiguous Memory Allocator`

- **`void *OAL_MemoryAllocFlag( uint32_t size, OAL_MEMORY_FLAG flags);`**
  - Allocation with specified flags.
  - **Returned value is not a useable pointer – it serves as a handle for other OAL functions!**
  - **To be NOT used in interrupt handling routine – uninterruptible lock present!**

- **`void *OAL_MemoryReturnAddress(void *pHandle, OAL_MEMORY_ACCESS access);`**
  - Returns address of the specified mapping type based on allocated handle.
  - Only one mapping type can exist at a same time.
  - **To be NOT used in interrupt handling routine – uninterruptible lock present!**

- **`void OAL_MemoryUnmap(void * pHandle);`**
  - Unmaps the virtual mapping from specified handle.
  - After this call, OAL_MemoryReturnAddress can be called with different access flags.
  - **To be NOT used in interrupt handling routine – uninterruptible lock present!**

- `void OAL_MemoryFree(void * pHandle);`
  - Unmaps and frees an allocated memory chunk.
  - **To be NOT used in interrupt handling routine – uninterruptible lock present!**

- `void OAL_MemoryFlush(void *pAddr, uint32_t size);`
  - Flushes the specified address space.
- `void OAL_MemoryFlushAndInvalidate(void *pAddr, uint32_t size);`
  - Flushes and invalidates the specified address space.
- `void OAL_MemoryInvalidate(void *pAddr, uint32_t size);`
  - Invalidates the specified address space.
- `void OAL_MemoryFlushAndInvalidateAll (void);`
  - Invalidates the whole memory space.

OAL Semaphore

- `int32_t OAL_SemaphoreCreate(OAL_SEMAPHORE *pSem, const char *pName, OAL_SEMAPHORE_TYPE type, unsigned int count);`
  - Create an OS semaphore.
- `int32_t OAL_SemaphoreObtain(OAL_SEMAPHORE sem, unsigned int time);`
  - Wait on a semaphore until it is obtained or time is reached.
- `int32_t OAL_SemaphoreRelease(OAL_SEMAPHORE sem);`
  - Signal or release a semaphore assuming the passed Semaphore ID has already been created.
- `int32_t OAL_SemaphoreDelete(OAL_SEMAPHORE sem);`
  - Delete a semaphore.

OAL Task

- `int32_t OAL_TaskInit(OAL_TASK_ENTRY task_entry, void* argument, unsigned int stack_size, unsigned int priority, char* task_name, OAL_TCB *task);`
  - Creates an OS Task.

- `int32_t OAL_TaskDelete(OAL_TCB task);`
  - Deletes the OS task.

- **`int32_t OAL_TaskJoin(OAL_TCB task);`**
  - Waits for the task to terminate

OAL Interrupt

- **`int32_t OAL_IRQInit(void (*pServiceRoutineFunc)(void), unsigned int irq, unsigned int priority, IRQ_MODE mode);`**
  - Initializes the IRQ and links it with a specific service function.
- **`void OAL_IRQEnable(unsigned int irq);`**
  - Enables a specific IRQ.
- **`int32_t OAL_IRQDelete(unsigned int irq);`**
  - Deletes the IRQ service routine.
- **`void OAL_IRQDisable(unsigned int irq);`**
  - Disables a specific IRQ.

3 OAL Contiguous Memory Allocator

The OAL Contiguous Memory Allocator implements contiguous memory allocation functionality. This is allocation scheme required by all vision processing accelerators like ISP, APEX and other subsystem DMA's.

3.1 Detailed function descriptions

`void *OAL_MemoryAllocFlag(uint32_t size, OAL_MEMORY_FLAG flags);`

Allocate a block of memory that meets certain criteria. Memory will be allocated from the appropriate heap while satisfying all the properties of flags. Memory allocation of specified size will allocate a block of this size and return a handle which can be used in subsequent OAL functions.

The returned pointer is not a virtual mapping and cannot be used for direct memory access from ARM core.
To be NOT used in interrupt handling routine – uninterruptible lock present!

The flags can be defined as follows (values can be combined by logical OR):

- OAL_ALLOC_AUTO
  - Default flag
  - Memory will be allocated from autobalanced heaps (see the OAL settings in chapter 8) according to number of allocations. Typically, autobalanced memory heaps are DDR0 and DDR1.
- OAL_ALLOC_DDR0
  - Memory will be allocated on DDR0
- OAL_ALLOC_DDR1
  - Memory will be allocated on DDR1
- OAL_ALLOC_SRAM_SINGLE
  - Memory will be allocated in single banked part of SRAM
- OAL_ALLOC_SRAM_MULTI
  - Memory will be allocated in multi banked part of SRAM
- OAL_MEMORY_FLAG_ZERO
  - Memory is initialized with value zero
- OAL_MEMORY_FLAG_ALIGN(ALIGN2)
  - Memory is aligned as specified. The alignment must be a number which is power of 2.
  - ALIGN2_CACHELINE
    - Memory is aligned on a cache line
  - ALIGN2_PAGE
    - Memory is aligned on a page
  - ALIGN2_BYTE(N)

- Memory is aligned on an N-byte boundary

`void *OAL_MemoryReturnAddress(void *pHandle, OAL_MEMORY_ACCESS access);`

Takes the given memory handle and returns the pointer that allows proper access as identified in the access parameter

The value of pHandle passed has to be the value returned from OAL_MemoryAllocFlag function.

Only one mapping can exist at a time. When OAL_MemoryReturnAddress is called for the first time, the memory is mapped. When different mapping is requested afterwards, the function returns NULL. Please see OAL_MemoryUnmap for unmapping the buffer prior to changing the access flags.

To be NOT used in interrupt handling routine – uninterruptible lock present.

OAL_MEMORY_ACCESS has following values:

- ACCESS_PHY
  - Returns the physical address of memory. Accessing this via the CPU will fail.
- ACCESS_CH_WB
  - Returns the address that will be interpreted as cached with a write back policy.
- ACCESS_CH_WT
  - Returns the address that will be interpreted as cached with a write through policy.
- ACCESS_NCH_B
  - Returns the address that will be interpreted as non-cached but buffered.
- ACCESS_NCH_NB
  - Returns the address that will be interpreted as non-cached and not buffered.

If a specific platform does not support some of the caching policies, NULL is returned.

`void OAL_MemoryUnmap(void *pHandle);`

Unmaps all the mappings associated to the handle. This function must be called when different access flags are requested. The buffers mapped before call of this functions are no longer useable and will cause crash of the application is written. The parameter pHandle must be a pointer returned by OAL_MemoryAllocFlag() function.

To be NOT used in interrupt handling routine – uninterruptible lock present!

`void OAL_MemoryFree(void *pHandle);`

Release the allocated memory and its associated mappings, if any. The parameter pHandle must be a pointer returned by OAL_MemoryAllocFlag() function.

To be NOT used in interrupt handling routine – uninterruptible lock present!

`void OAL_MemoryFlushAndInvalidateAll (void);`

The function flushes and invalidates all the data memory regions in the system.

Flush and invalidate: If any region of this memory is currently held in the CPU cache, those contents will be written back to physical memory. On top of the flush operations, all the cache lines will be marked invalid. This will require a subsequent read operation to fetch data from physical memory.

`void OAL_MemoryFlush (void *pAddr, uint32_t size);`

The function flushes specific mapped region. The parameter pAddr must be a value returned by OAL_MemoryReturnAddress with flags ACCESS_CH_WB or ACCESS_CH_WT.

Flush: If any region of this memory is currently held in the CPU cache, those contents will be written back to physical memory.

`void OAL_MemoryInvalidate (void *pAddr, uint32_t size);`

The function invalidates specific mapped region. The parameter pAddr must be a value returned by OAL_MemoryReturnAddress with flags ACCESS_CH_WB or ACCESS_CH_WT.

Invalidate: Simply scratches out all the cache lines. Data kept in the cache is lost. Any subsequent read will fetch the data from physical memory.

Warning: Depending on mapping, the invalidate call may need a memory access to the whole buffer prior to call to the invalidate function due to delayed real allocation of the buffer. The memory invalidate function will hang if the buffer wasn't accessed before its call.

void OAL_MemoryFlushAndInvalidate (void *pAddr, uint32_t size);

The function flushes and invalidates specific mapped region. The parameter pAddr must be a value returned by OAL_MemoryReturnAddress with flags ACCESS_CH_WB or ACCESS_CH_WT.

Flush and invalidate: If any region of this memory is currently held in the CPU cache, those contents will be written back to physical memory. On top of the flush operations, all the cache lines will be marked invalid. This will require a subsequent read operation to fetch data from physical memory.

3.2 Example of use

```
// Allocate the memory - the returned value is the handle
void *handle_oal = OAL_MemoryAllocFlag(1024, OAL_ALLOC_AUTO);


// Get the physical address (e.g. for DMA purpose).
// Cannot be accessed by CPU!
void *ptr_physical = OAL_MemoryReturnAddress(handle_oal, ACCESS_PHY);


// Get the cached pointer for CPU write
void *ptr_cached = OAL_MemoryReturnAddress(handle_oal, ACCESS_CH_WB);


// Get the cached pointer for CPU write second time, result will be
// ptr_cached == ptr__cached2
void *ptr_cached2 = OAL_MemoryReturnAddress(handle_oal, ACCESS_CH_WB);


// Example of CPU access
memset(ptr_cached, 0, 1024);


// Flush and invalidate the written buffer
OAL_MemoryFlushAndInvalidate(ptr_cached, 1024);


// Remap the buffer for non-cached mapping
OAL_MemoryUnmap(handle_oal);
void *ptr_noncached = OAL_MemoryReturnAddress(handle_oal, ACCESS_NCH_NB);


// Try to map differently - will return NULL and outputs an error!
void *ptr_noncached = OAL_MemoryReturnAddress(handle_oal, ACCESS_CH_WB);


// Flush and invalidate the written buffer
OAL_MemoryFree(handle_oal);
```

4 OAL Semaphore

An OAL Semaphore interface provides a mechanism allowing management and usage of shared resources.

When OAL_SEmaphoreCreate is used to create a new semaphore, a semaphore type must be specified. OAL provides three types of semaphores:

- OAL_SEMAPHORE_BINARY
- OAL_SEMAPHORE_COUNTING
- OAL_SEMAPHORE_MUTEX

Semaphores types are differentiated based on two attributes; the legal values for the semaphore count and the resumption order of suspended tasks. Binary and mutex type semaphores are restricted to a count of 0 or 1. Counting semaphores are allowed to have count values greater than 1. The value of the count represents the number of times the semaphore can be obtained without suspending the calling task. A task trying to obtain a semaphore with a count of 0 will be suspended. Obtaining a semaphore will decrement the count if it is greater than 0 and releasing increments the count.

Binary and counting type semaphores will resume tasks in the order in which they tried to obtain the semaphore; a first in, first out order. Mutex semaphores will first resume the higher priority tasks followed by the lower priority tasks; within the same priority the order is first in, first out, (FIFO).

4.1 Detailed function description

```
int32_t OAL_SemaphoreCreate(OAL_SEMAPHORE *pSem,

    const char *pName, OAL_SEMAPHORE_TYPE type, unsigned int count);
```

Create an OS semaphore. If succeeds, the created Semaphore ID will be subsequently referred to in later Semaphore calls.

Semaphores types are differentiated based on two attributes; the legal values for the semaphore count and the resumption order of suspended tasks.

Binary and mutex type semaphores are restricted to a count of 0 or 1. Counting semaphores are allowed to have count values greater than 1. The value of the count represents the number of times the semaphore can be obtained without suspending the calling task. A task trying to obtain a semaphore with a count of 0 will be suspended. Obtaining a semaphore will decrement the count if it is greater than 0 and releasing increments the count.

Binary and counting type semaphores will resume tasks in the order in which they tried to obtain the semaphore; a first in, first out order. Mutex semaphores will resume first the higher priority tasks followed by the lower priority tasks; within the same priority the order is first in, first out,(FIFO).

- pSem [out] Semaphore ID will be written to (*pSem)
- pName [in] String identifying new semaphore(name will be truncated to 7 characters)
- type [in] Semaphore type:
    - OAL_SEMAPHORE_BINARY
    - OAL_SEMAPHORE_COUNTING
    - OAL_SEMAPHORE_MUTEX
- Count [in] Initial count for counting semaphores

`int32_t OAL_ SemaphoreObtain (OAL_SEMAPHORE sem, unsigned int time);`

Try to obtain a semaphore until it times out. If the Semaphore is not available, the calling task will be blocked until the Semaphore becomes available or times out.

NOTE : Timeout is expressed in microseconds but true timeout period will be rounded up to the granularity of the OS timer which is 10ms.

- sem [in] Semaphore ID
- time [in] Timeout in microseconds

`int32_t OAL_SemaphoreRelease(OAL_SEMAPHORE sem);`

Signal or release a semaphore assuming the passed Semaphore ID has already been created.

- sem [in] Semaphore ID

`int32_t OAL_SemaphoreDelete(OAL_SEMAPHORE sem);`

Delete a semaphore assuming the Semaphore ID argument has been already created.

- sem [in] Semaphore ID

4.2 Example of use

```
// Define the semaphore
OAL_SEMAPHORE my_semaphore

// Initialize the counting semaphore
OAL_SemaphoreCreate(&my_semaphore, “sema_1”, OAL_SEMAPHORE_COUNTING, 0);

// Wait until time is reached or semaphore is free
OAL_SemaphoreObtain(my_semaphore, 1000);

// SAFE SECTION …

// Release the semaphore
OAL_SemaphoreRelease(my_semaphore);

// Delete the semaphore
OAL_SemaphoreDelete(my_semaphore);
```

5 OAL Task

OAL Task group provides functionality for multi-threaded application. It implements a simple interface for task (thread) creation, deletion and joining.

5.1 Detailed function description

`int32_t OAL_TaskInit(OAL_TASK_ENTRY task_entry, void* argument, unsigned int stack_size, unsigned int priority, char* task_name, OAL_TCB *task);`

Create an OS Task. If succeeded, the created Task ID will be subsequently referred to in later OS Task calls.

• task_entry	Entry point function
• argument	Argument to be passed to the entry point
• stack_size	Stack size for the task in bytes, or OAL_TASK_DEFAULT_STACK
• priority	Priority for the task, or OAL_TASK_DEFAULT_PRIORITY
• task_name	Task name
• startedOption	Create and start the task right away or wait.
• task	pointer to OAL_TCB which will be updated with the TCB of the created task

`int32_t OAL_TaskDelete(OAL_TCB task);`

Delete the specified task assuming the specified task ID has already been created.

• Task	the created task ID

`int32_t OAL_TaskJoin(OAL_TCB task);`

Waits for the task to terminate.

• Task	the created task ID

5.2 Example of use

```
// Define the thread – with one parameter, threadid
void *my_thread(void *threaded)
{
}

// …

// Create some number of threads
OAL_TCB tasks[NUM_TASKS];

// start all threads
for (int i = 0; i < NUM_TASKS; i++)
{
  // i is the thread id passed as the parameter
  if (OAL_TaskInit(&my_thread, (void *)i, OAL_TASK_DEFAULT_STACK,
                   OAL_TASK_DEFAULT_PRIORITY, (char *)”mytask”,
                   &tasks[u]) == OAL_FAILURE)
  {
    printf(“Error in creating thread %d\n”, i);
    return -1;
  }
}

// Wait for all threads to finish
for (int i = 0; i < NUM_TASKS; i++)
{
  OAL_TaskJoin(tasks[i]);
}
```

6 OAL Interrupt

OAL Interrupt implements a simple interface for interrupt servicing.

6.1 Detailed function description

`int32_t OAL_IRQInit(void (*pServiceRoutineFunc)(void), unsigned int irq, unsigned int priority, IRQ_MODE mode);`

The function initializes the specific interrupt and links it with a provided service function. The priority and mode can be specified:

• pServiceRoutineFunc	[in] The service callback function.
• irq	[in] Interrupt number
• priority	[in] Priority of the interrupt to be set.
• mode	[in] Interrupt invocation mode (IRQ_LEVEL, IRQ_EDGE)

`void OAL_IRQEnable(unsigned int irq);`

Enables the interrupt – must be called after init function in order to enable interrupt callback.

• irq	[in] Interrupt number

`int32_t OAL_IRQDelete(unsigned int irq);`

Deletes the interrupt – the function unlinks the service function from the specified interrupt ID.

• irq	[in] Interrupt number

`void OAL_IRQDisable(unsigned int irq);`

Disables the interrupt – Can be used for temporal disablement of a specific IRQ. The IRQ can be re-enabled by IRQ_Enable function.

• irq	[in] Interrupt number

6.2 Example of use

```
// Define the interrupt handler routine
void IRQ_Handler()
{
}


// …


// Initialize and enable interrupt listener
OAL_IRQInit(IRQ_Handler, IRQ_NUMBER, 0xA0, OAL_IRQ_EDGE);
OAL_IRQEnable(IRQ_NUMBER);


// …


// Disable and delete the interrupt listener
OAL_IRQDisable (IRQ_NUMBER);
OAL_IRQDelete(IRQ_NUMBER);
```

7 OS Specific behavior

7.1 Linux OS

7.1.1 Contiguous Memory Allocator

7.1.1.1 Driver structure

The OAL for Linux consists of two parts – the kernel module (allocator part) and user space part. The kernel space part consists of the allocator which:

- manages all the allocations on the heap from specified memory region
- bookkeeps all the allocations and mappings from user-space part,
- provides the cache management functions and
- provides the debug interface.

7.1.1.2 Setting the Driver structure

Following parts must be set in the Linux Device tree file. The figure settings are for four memory heaps – DDR0, DDR1, SRAM Single banked region and SRAM Multi banked region:

```
/* Memory reservation is needed in order to remove this memory from Linux
heap */
reserved-memory {
  oalddr0_allocator_memory: oalddr0@0x8B000000 {
    reg = <0x0 0x8B000000 0x0 0x05000000>; /* 80 MB */
    no-map;
  };

  oalddr1_allocator_memory: oalddr1@0xCB000000 {
    reg = <0x0 0xCB000000 0x0 0x05000000>; /* 80 MB */
    no-map;
  };

  oalsramS_allocator_memory: oalsramS@0x3E800000 {
    reg = <0x0 0x3E800000 0x0 0x00300000>;  /* 3 MB */
    no-map;
  };

  oalsramM_allocator_memory: oalsramM@0x3EB00000 {
    reg = <0x0 0x3EB00000 0x0 0x00100000>;  /* 1 MB */
    no-map;
  };
};
```

Each heap needs to be then defined as separate device linked to the reserved memory region. **Autobalance** flag points to the region which will be in a pool used for automatic memory heap selection.

```
/* Each memory heap needs separate memory device */
oalmem0: oalmem0@80000000 {
  compatible = "fsl,s32v234-oal";               // OAL ALLOCATOR DEVICE
  reg = <0x0 0x8B000000 0x0 0x05000000>;        // MEMORY RANGE
  memory-region = <&oalddr0_allocator_memory>;  // RESERVED MEMORY
  id = <0>;                                     // ID (FOR ALLOCATORY USE
                                          // MUST MATCH WITH MEM REGION ID)
  align = <0x1000>;                             // DEFAULT ALIGNMENT
  autobalance;                                  // PRESENT IF AUTOBALANCE
                                                // IS NEEDED
};

oalmem1: oalmem1@C0000000 {
  compatible = "fsl,s32v234-oal";
  reg = <0x0 0xCB000000 0x0 0x05000000>;
  memory-region = <&oalddr1_allocator_memory>;
  id = <1>;
  align = <0x1000>;
  autobalance;
};

oalsramS: oalsramS@3E800000 {
  compatible = "fsl,s32v234-oal";
  reg = <0x0 0x3E800000 0x0 0x00300000>;
  memory-region = <&oalsramS_allocator_memory>;
  id = <2>;
  align = <0x8>;
  init;
};

oalsramM: oalsramM@3EB00000 {
  compatible = "fsl,s32v234-oal";
  reg = <0x0 0x3EB00000 0x0 0x00100000>;
  memory-region = <&oalsramM_allocator_memory>;
  id = <3>;
  align = <0x8>;
  init;
};
```

7.1.1.3Allocator restrictions

In current version, only ACCESS_PHY, ACCESS_CH_WB and ACCESS_NCH_NB are supported. The rest of accesses will return NULL.

7.1.1.4Using the OAL Allocator

Under Linux, the **oal_cma.ko** kernel module must be loaded prior to any call of following functions. Any other functionality (e.g. mutexes) are not affected.

7.1.2Interrupt

In current release, the interrupt servicing routines are not implemented for Linux OS.

7.2 Bare metal environment

7.2.1CMAL

7.2.1.1Setting the Driver structure

Contiguous memory allocator needs to be set in linker file (in order to restrict the memory heaps):

```
MEMORY
{
  /* Memory for OAL allocations */
  mem_oal_ddr0: ORIGIN = 0x8B000000, LENGTH = 0x5000000

  /* Memory for OAL allocations */
  mem_oal_ddr1: ORIGIN = 0xCB000000, LENGTH = 0x5000000

  /* Memory for OAL allocations */
  mem_oal_sram: ORIGIN = 0x3E800000, LENGTH = 0x400000
}
```

And then specified in the OAL_Initialize function in s32v234_sdk/libs/utils/oal/user/src/standalone/oal_memory_driver_if.cpp

```
////////////////////////////////////////////////////////////////////////
/// Initializes OAL
////////////////////////////////////////////////////////////////////////
int32_t OAL_Initialize()
{
  if (ddr_init == false)
  {       //     ID  BASE                LENGTH              ALIGN   AUTO
    OAL_InitRegion(0, 0x000000008B000000, 0x0000000005000000, 0x1000, 1);
    OAL_InitRegion(1, 0x00000000CB000000, 0x0000000005000000, 0x1000, 1);
    OAL_InitRegion(2, 0x000000003E800000, 0x0000000000300000, 0x8, 0);
    OAL_InitRegion(3, 0x000000003EB00000, 0x0000000000100000, 0x8, 0);
    ddr_init = true;
  }
  return OAL_SUCCESS;
}
```

7.2.1.2Restrictions

In current version, only ACCESS_PHY, ACCESS_CH_WB and ACCESS_NCH_NB are supported. The rest of accesses will return NULL.

7.2.2Semaphore

7.2.2.1Restrictions

In current version, the semaphore functionality is disabled for no OS environment (no multi-tasking available). All create, obtain, release and delete functions will return SUCCESS (to achieve compatibility).

7.2.3Task

7.2.3.1Restrictions

In current version, the task functionality is disabled for no OS environment. When task is created, it will be executed immediately sequentially (to achieve compatibility).