Application Note

AN2345/D
Rev. 1, 9/2003

Real-Time Memory Manager for StarCore DSPs

by Joe Liccese

CONTENTS

1 Memory Manager Limitations 1
2 VSMM Basics 6
3 VSMM Distribution ... 114
4 Building Your Application with VSMM 15
5 VSMM Benchmarking.. 21
6 VSSM Functions 23
7 Building VSMM Examples 1–3 26
8 VSMM Configuration Source Listing 28
9 References 32

Freescale Semiconductor, Inc.

A common limitation of many general-purpose memory managers is that they cannot run within real-time operating system (RTOS) environments typically used with digital signal processors (DSPs), because of reentrancy issues, lack of mutual exclusion, or their non-determinism. Also, general-purpose memory managers do not provide sufficient speed, flexibility, or efficiency and often cause the memory space to become fragmented. Memory fragmentation is a serious problem because the time required to de-fragment memory is not necessarily predictable and is consequently non-deterministic. This application note describes the Very Small Memory Manager (VSMM), which addresses these memory manager limitations and meets the critical need of Motorola StarCore® DSP customers for a small, simple, efficient, flexible, fast, and deterministic memory manager within their real-time applications and systems. Although the VSMM operates within an RTOS environment, an RTOS is not required.

1 Memory Manager Limitations

Whether you are using only static memory or dynamic allocation on a heap, you must proceed cautiously. Programmers cannot afford to ignore the risks inherent in memory usage. This section describes these risks in detail and discusses how the VSMM overcomes them.

1.1 Fragmentation

Fragmentation is a common and serious problem inherent in many memory managers. Often, fragmentation cannot be corrected at the application level. Fragmentation results when tasks randomly allocate and free arbitrarily sized amounts of contiguous memory (see **Figure 1**). If such a process continues long enough, the free memory becomes so fragmented that there is insufficient contiguous memory available to fill an allocation request. Then the system must reorganize the heap via garbage collection before it can fill any new allocation requests. The routine to reform the free pool requires an indeterminate amount of time, depending on the severity of the fragmentation.

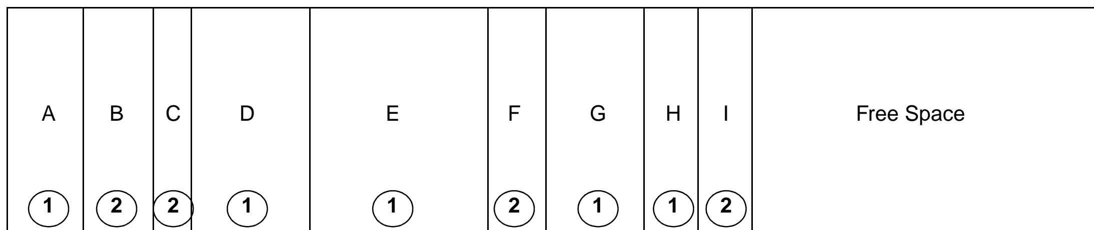
The danger of fragmentation has been overestimated in academic experiments that focus on randomly sized allocations. In practice, allocations tend to come in a limited number of sizes. A survey of several UNIX applications revealed that 90 percent of allocations are covered by six sizes, and 99.9 percent of allocations are covered by 141 sizes[3]. Therefore, the probability of finding a memory block that exactly matches the size of any given request is far higher than estimated, given a random distribution of allocation sizes. Despite this data, any fragmentation within a real-time system eventually requires garbage collection for an indeterminate amount of time. This poses a problem for real-time systems because a real-time system must be deterministic to meet real-time constraints.

To eliminate the potential for fragmentation completely, you can use heaps, which are partitions of fixed-sized memory blocks that can be tuned at design time for the size of the requests to be made. Each heap contains an array of blocks, and unused blocks can be linked together in a list. The heaps themselves are declared as arrays. This mechanism avoids the overhead of a large header for each block, since size information is fixed for each heap. The VSMM employs heaps to eliminate fragmentation.

1.2 Deterministic Systems

Real-time systems are prolific within embedded environments. A real-time system must be deterministic, that is, it must provide a required level of service in a bounded response time. Since most general memory managers require some form of garbage collection, a process that is inherently non-deterministic, real-time embedded systems are often unable to use these memory managers.

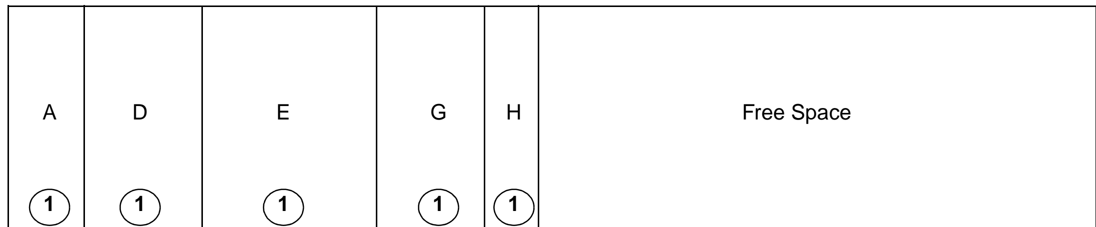
When memory becomes low in the traditional *mark and sweep* garbage-collection algorithm, the system stops all user processes, locates the set of unreachable objects or memory blocks, and frees them. To accomplish this task, it must examine every object reference, such as local variables and static structures. Each referenced object is checked to determine whether it is already marked. If an object is not marked, it is marked and all its references are processed; otherwise the system moves on to the next reference. When this process is complete, any unmarked objects are deemed unreachable and can therefore be recycled. This marking technique is superior to reference-counting schemes because it correctly detects groups of dead objects that are referenced only by other dead objects. **Figure 1** and **Figure 2** show before and after pictures of this process, respectively. Of course, the heap can become low on free space at any time, and the time required to mark and sweep is proportional to the number of objects and references. The result is a pattern in which the application periodically becomes unresponsive or stalls for a brief time. Such stalls are unacceptable in real-time systems, even *soft* real-time systems.



1 Marked space

2 Unreachable space

Figure 1. Fragmented Memory Before Garbage Collection



1 Marked space

2 Unreachable space

Figure 2. Memory After Garbage Collection

One way to prevent fragmentation is to allocate pools of the same fixed-sized memory blocks so that a careful implementation can have fixed execution times for allocating and freeing the blocks. More general heap implementations always involve iterating through lists that can vary in size. The VSMM eliminates fragmentation by using heaps that consist only of fixed-sized memory blocks. Moreover, all

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

VSMM routines execute in a known and deterministic way; that is, each VSMM routine has a known and bounded response time. Details on this response time can be found within the interface control document (ICD) [6] of each VSMM module.

In the VSMM heap schema, freeing a memory block merely points the current block to the memory block to which the global free list pointer is pointing. Then the global free list pointer is updated to point to the starting address of the freed memory block, a very efficient and deterministic process (see **Figure 3**). Notice that the free space is not contiguous. Rather, each free block points to the next free block within the heap, and there are no *unreachable* blocks.

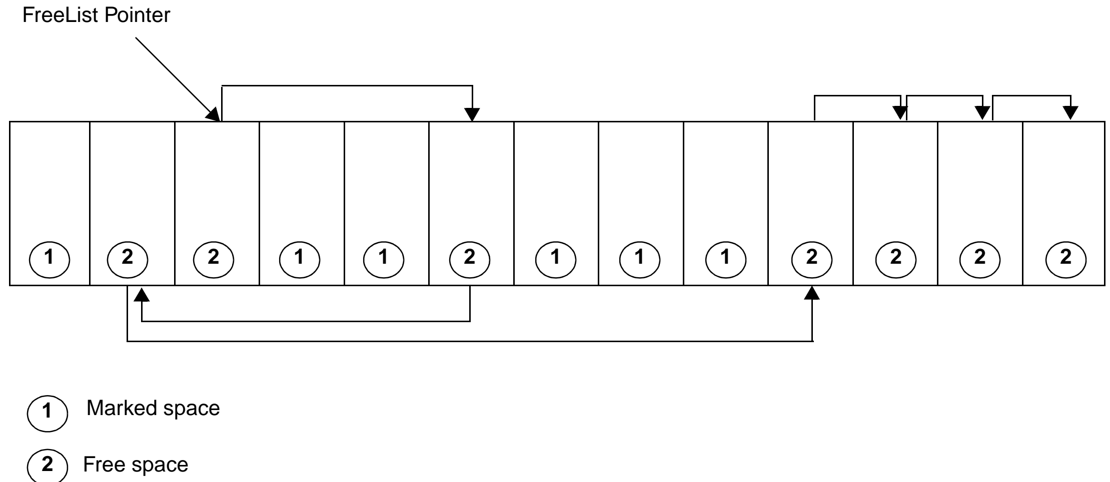


Figure 3. Heap with Fixed-Sized Memory Blocks

1.3 Reentrancy

Most embedded systems use interrupts, and many support multitasking or multi-threaded operations. In such applications, the program control flow can change contexts at any time. When an interrupt occurs, the current operation is put on hold, and another operation starts. If these operations share variables, you must ensure that one routine does not corrupt the data of another. By carefully controlling how data is shared, you create reentrant functions that allow multiple concurrent invocations but do not interfere with each other. *Reentrancy* is a term to describe a module in which multiple processes can share the same copy in memory. A reentrant module ensures that no instructions modify the contents of variable values outside its context. A reentrant module must satisfy the following reentrancy rules:

1. An atomic operation is one that cannot be interrupted. Use all shared variables in an atomic way, unless each is allocated to a statically declared specific instance of the function.

   An instance is a path through the code. There is no reason a single function cannot be called from many places. In a multitasking environment, it is possible that several copies of the function may execute concurrently. The use of automatic variables—that is, local variables that are statically declared within a module—ensures reentrancy since each call to this module has its own copy of these variables created on the stack. Another option is to allocate memory dynamically so that each iteration uses a unique data area.

2. Do not call non-reentrant functions.

   Calling a non-reentrant routine effectively makes the routine that performed the call non-reentrant.

3. Do not use the hardware in a non-atomic way.

   Hardware looks a lot like a variable. If more than a single I/O or hardware operation is required to handle a device, reentrancy problems can result. To prevent reentrancy issues on shared hardware,

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

mutual exclusivity must be ensured. The following operation is atomic because nothing but a reset can stop or interrupt the instruction:

```
move.l d1, r0
```

Never assume that a compiler generates atomic code. For example, if we assume that the variable gusiFooBar in the following instruction is global, this instruction may appear to be an atomic instruction.

```
gusiFooBar += 1
```

However, the compiler may generate the following code, which is clearly not an atomic instruction and therefore not reentrant:

```
move.w _gusiFooBar, d1
inc d1
move.w d1, _gusiFooBar
```

The best approach to ensuring reentrancy is to avoid the use of shared variables, but it is not always possible to eliminate them in real-time systems. Usually, when shared variables are necessary, interrupts are disabled until the shared variable is updated and then re-enabled. However, disabling interrupts increases system latency, reducing the ability of the system to respond rapidly to external events. Another approach to ensuring reentrancy is the use of a mutex, also referred to as a binary semaphore. Mutexes are simple on-off state indicators whose processing is inherently atomic. Semaphores consume more overhead than interrupt disabling, but they do not affect interrupt latencies.

Both of these reentrancy approaches can cause priority inversion. Priority inversion occurs when a higher-priority task must wait on a lower-priority task to complete its use of a shared resource. In the mutex approach, the task that owns a mutex retains control of its associated resource until the owner determines that exclusive access is no longer needed. If another task with a higher priority attempts to acquire the mutex, the kernel blocks the higher-priority task. In the interrupt disabling approach, a higher-priority process is unable to interrupt a lower-priority process while interrupts are disabled, forcing it to wait until the lower-priority process re-enables interrupts.

With the exception of two shared variables, all VSMM modules employ statically declared variables so that each call has its own copy of these variables. To ensure reentrancy when either of the shared global variables is updated, the VSMM can enable/disable interrupts or use a binary semaphore. The exact method implemented depends on the *critical method* selected by the programmer. Refer to **Section 3.6**, *VSMM Critical Methods,* on page 14 for details on the available VSMM critical methods. Since all VSMM variables are either allocated to a specific instance or the enter/exit critical methods are used when the shared variables are updated, all VSMM modules are reentrant.

1.4 Mutual Exclusion

While each process must have its own stack, it may or may not have its own heap, regardless of the heap allocation scheme. If your allocation scheme has more than one heap, you must tune the size of a number of heaps. A heap to be shared by many processes must be reentrant, which requires adding locks that may slow down each allocation and deallocation. It may be necessary to allow one process to allocate a block of memory that can be freed by another process, which is useful for passing inter-process messages. When memory is passed between processes, it is important to ensure that the memory owner at each point is well-defined. Two processes must not act as if they own a block of memory simultaneously. Otherwise, there may be two calls to free the same memory block. Also, two processes must never attempt to manage

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

the same heap simultaneously. For example, if one process attempts to allocate a block of memory and fails to update the global free list pointer before it is preempted by a process with a higher priority, the global free pointer can easily become corrupt.

All memory managers have sections of code deemed critical, such as the update of the global free list pointer or the memory control block (MCB). Critical sections are the sections of a program that are vulnerable to undesirable interruption or corruption. To provide exclusivity to these sections, a locking mechanism is needed. Mutual exclusion is the implementation of a locking mechanism to ensure that only one process has access to a resource or critical section of code at any given time. The VSMM guarantees mutual exclusion through the use of "critical methods," which are associated with routines used prior to entry into and exit from critical sections. Before entry into a critical section, these routines can perform the following tasks:

- Save the current interrupt state and disable interrupts
- Change the interrupt priority level so that only processes with a specific priority are allowed to generate and service interrupts
- Use a binary semaphore to ensure mutual exclusivity.

Upon exit from a from a critical section, these critical methods can re-enable interrupts or restore interrupts to their previous saved state or unlock a binary semaphore.

As a result of ensuring mutual exclusivity of critical sections, priority inversion is possible. However, since all VSMM critical sections are extremely short and execute rapidly, this issue is minor but one that deserves attention.

1.5 Memory Leaks

Many objects, structures, or buffers exist for a period of time that does not match the invocation of any one function. This is particularly true in event-driven programs, such as many embedded real-time systems. One event may cause an item to be created, and that item remains in use until some other event leads to its elimination. At a certain point in the code, you may be uncertain whether a particular block is still needed. If you free this block of memory but continue to access it, say, via a second pointer to the same memory, the program may function well until that particular block of memory is reallocated to another part of the program. Then, two different parts of the program proceed to write over the shared space. If you decide not to free the memory because it may still be in use, then there may not be another opportunity to free it if all pointers to the block are out of scope or reassigned to point elsewhere. In this case, the program logic is not affected, but if the piece of code that leaks memory is regularly visited, the leak tends towards infinity as the execution time of the program increases.

Ultimately, the amount of physical memory determines how long a program can execute. On many desktop applications, a small leak may be acceptable. For example, a compiler that leaks 100 bytes for every 1,000 lines compiled can still successfully compile a 100,000 line file on a modern PC, since all allocated memory is recovered on exit of the program. However, in many embedded systems in which minimal memory is available, no upper limit on the life of the program is acceptable. Any memory leak is an error to be rectified by correcting the logic of the application program. By monitoring the size of each heap and confirming that the number of blocks in use ceases to grow after extended use, the programmer can be confident that leaks are eliminated. While it is wise to size the heaps larger than the worst case seen in testing, too much *padding* leads to wasted memory.

The VSMM is verified to contain zero memory leaks. However, this does not guarantee that your system or application will not leak. The VSMM provides a query routine that reports the current statistics of a heap. This routine can be very useful in determining whether your system contains any memory leaks. For example, if you perform a query on each heap before the system exits, if any heap statistics indicate memory blocks still in use, there are memory leaks within the system.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

2 VSMM Basics

This section presents an overview of the VSMM requirements, and architecture. The VSMM allows an application to obtain fixed-size memory blocks from a partition (heap) consisting of a contiguous area of memory. All memory blocks within a heap must be of the same size, and each heap may have an integral number of blocks. Allocation of a block of memory from a heap occurs in real time and is deterministic.

The VSMM allows multiple heaps, and the number of heaps is limited only by the amount of physical memory available, allowing various sizes of memory blocks to be accessed by an application. Heaps can be created and eliminated at run-time, and a memory block from one heap may be partitioned into smaller block sizes. The VSMM can create heaps dynamically at run-time, providing heaps of varying block sizes. The VSMM also allows heaps to be created from other heaps, partitioning the new heap into smaller block sizes than those of the heap from which it was created.

2.1 VSMM Target System Properties

This section presents the features of the VSMM that define the capabilities of the VSMM architecture and implementations based on this architecture, as follows:

- *Compactness*. Because of the extremely limited memory available to most DSPs, VSMM code space is limited to no more than 2 KB. The maximum VSMM code size is approximately 1.5 KB with a minimum code size of approximately 700 bytes.
- *Speed*. In the highly complex tasks that today's DSPs execute; speed is essential. Typically, the more efficiently an application executes on a DSP, more users are supported by the system, making the cost less prohibitive. Therefore, all VSMM routines are geared toward optimal efficiency. There is no maximum cycle requirement for a VSMM routine, but each routine is fully optimized to execute within the least amount of cycles, and all routines must have a bounded response time.
- *Efficiency*. To attain the best use of available memory, VSMM overhead must be minimized. An overhead of 32 bytes per heap and 16 bytes per memory block allocation is acceptable. The VSMM requires only 24 bytes per heap and 8 bytes per memory block.
- *Flexible operation*. The VSMM routines not required by an application or system can be omitted from a build. Additionally, the design is not restricted to a specific DSP or operating system (OS) but instead is configurable based on the targeted DSP and operating system, including the RTOS.
- *RTOS-compatible*. The VSMM can operate within an RTOS environment, so its routines are reentrant and deterministic, and they observe mutual exclusion during critical pointer updates. All critical sections of code, that is, code requiring mutual exclusion, are kept short and efficient. The VSMM can also operate efficiently and reliably within non-RTOS environments.
- *Byte Alignment*.The VSMM adheres to the Motorola StarCore DSP byte-alignment rules. Refer to [1] for information on byte alignment requirements for Motorola StarCore devices.
- *Fragmentation*. The VSMM design results in no fragmentation of available memory. Garbage collection is not implemented.

2.2 Memory Block Header

Each memory block within a heap contains an 8-byte header, referred to as the memory block header (MBH). The MBH identifies the heap from which the block is allocated. The handle or address of each heap is stored within the MBH when a block is allocated from the heap. The heap handle is the address of the associated heap memory control block. See **Figure 4** for an illustration of an allocated memory block and its associated MBH. When a memory block is freed via the VSMMMemFree routine, the memory block is returned to the heap indicated by the data within its associated MBH.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

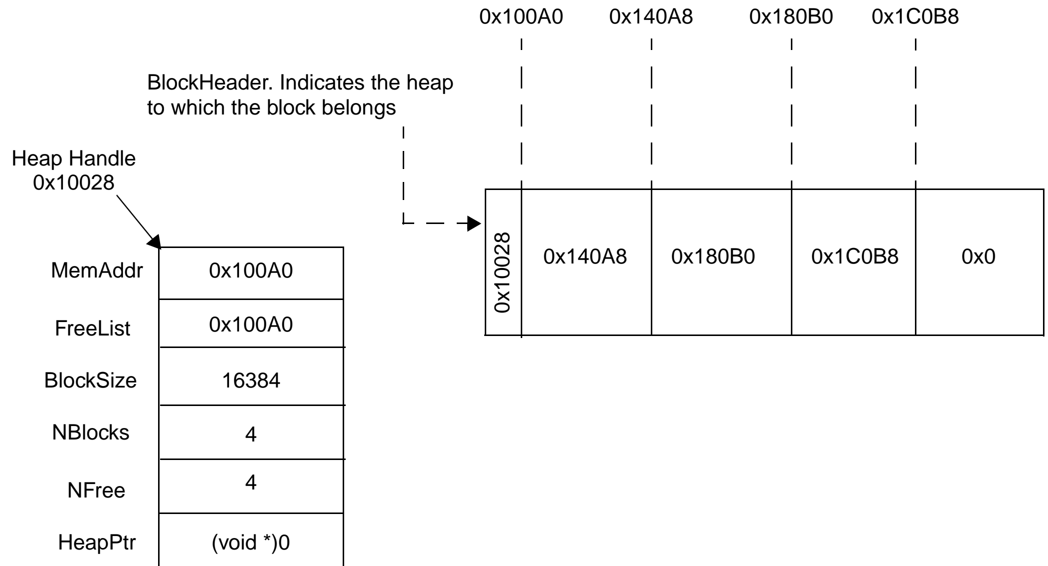


Figure 4. Heap MCB Following its Creation

2.3 Memory Control Blocks

Each heap requires its own 24-byte MCB that contains all required heap information. The MCB is defined by the following code:

```
typedef struct
{
   void   *pvVSMMMemAddr;
   void   *pvVSMMMemFreeList;
   INT32U  uliVSMMMemBlkSize;
   INT32U  uliVSMMMemNBlks;
   INT32U  uliVSMMMemNFree;
   INT32U  uliVSMMMemHeapAddr;
} t_VSMM_MEM;
```

Table 1. MCB Contents Defined

MCB Variable	Description
VSMMMemAddr	A pointer to the base address of the heap
VSMMMemFreeList	A pointer to the next available free memory block within the heap.
VSMMMemBlkSize	The size of each block contained within this heap.
VSMMMemNBlks	The number of blocks contained within this heap.
VSMMMemNFree	The number of memory blocks within this heap that are still available.
VSMMMemHeapAddr	Pointer to a parent heap. If the heap associated with this MCB was created directly from free memory, the value is NULL; otherwise, if the heap was created from a memory block associated with another heap, the value is this heap’s handle or address.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

When a heap is created, an MCB is removed from the free list and updated to reflect the characteristics of the new heap. Each of the heap's memory blocks is initialized so that it points to the next available memory block within the heap. The last memory block within a heap contains a NULL. The header of the first memory block is also initialized to the address of its heap (see **Figure 4**). Notice that the MCB *heapAddr* element is set to NULL, indicating that this heap was created in free memory space and not from a memory block of another heap.

When a heap is eliminated, the MCB is reinitialized and its free list pointer is set to the global free list pointer. The global free list pointer is updated to point to this newly freed space.

When a memory block is allocated from a heap, the MCB global free list pointer is updated to point to the next available memory block after the allocated space. When a memory block is freed, it is added back to the head of the free list (see **Figure 5**). The MCB free list is then updated to point to this newly freed memory block.

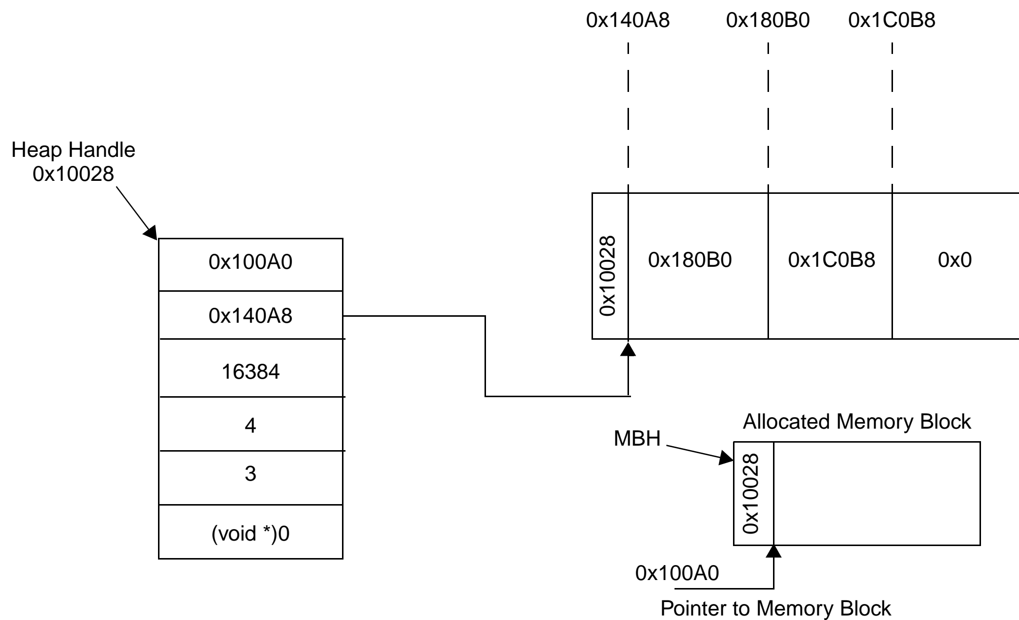


Figure 5. Memory Block Allocation

Figure 6 shows a memory block that is partitioned into another heap. The MCB heap handle is that of the heap from which the block was allocated, and the MBH of the first memory block within this new heap is initialized to this new heap handle. **Figure 7** depicts the allocation of several memory blocks from the two created heaps. Notice how each memory block header contains the heap handle of the heap from which it was allocated. When each block is freed, it is returned to its respective heap, identified by this handle. When the second heap is destroyed, its memory block is returned to the heap identified by its MCB heap handle.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

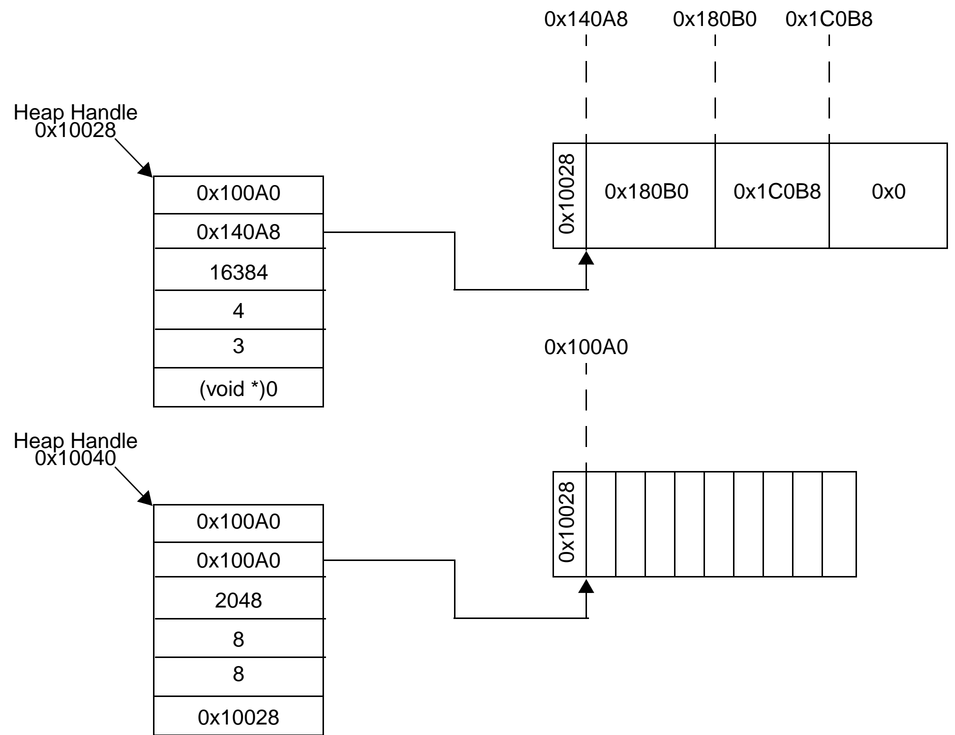


Figure 6. A Heap Created From a Memory Block of Another Heap

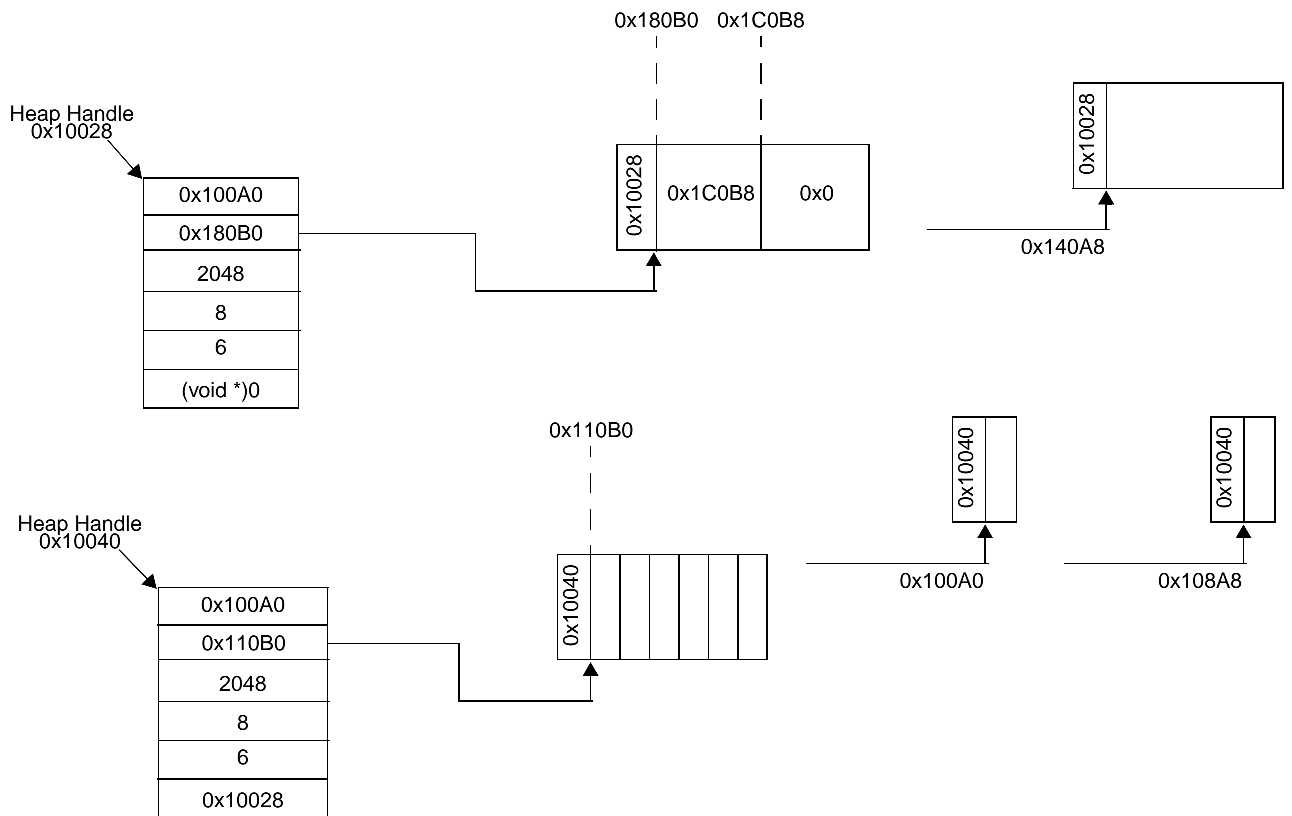


Figure 7. Multiple Memory Block Allocations from Two Unique Heaps

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

2.4 VSMM Global Objects

The VSMM uses two global objects to assist in managing each heap:

```
t_VSMM_MEM   *pstVSMMMemFreeList;

t_VSMM_MEM astVSMMMemTbl[VSMM_MAX_MEM_PART];
```

Table 2. Global Objects for Managing Heaps

Variable	Description
VSMMMemFreeList	A pointer to the next available MCB,
VSMMMemTbl	Array of free MCBs.

The *VSMMMemInit* routine initializes each MCB within the *VSMMMemTbl* and sets the global *VSMMMemFreeList* pointer to the first memory control block within this array. Each MCB *VSMMMemFreeList* pointer is updated to the next free MCB within the *VSMMMemTbl* or to NULL if it is the last MCB in the table. When a heap is destroyed, its associated MCB is inserted at the head of this free list and the global *VSMMMemFreeList* pointer is updated. **Figure 8** illustrates the global FreeList pointer and MCB array following VSMM initialization with a maximum of three MCBs specified. Mutual exclusion is ensured when these global variables are updated.

In addition to these two global objects, VSMM uses a global flag to enable or disable VSMM routine parameter verification. This flag should be statically initialized and not updated at run-time because it has no associated mutex (binary semaphore):

```
unsigned char gucVSMM_ARG_CHK_EN
```

If the global *gucVSMM_ARG_CHK_EN* contains a value greater than zero, each VSMM routine requiring passed parameters validates these parameters. The final global used by the VSMM is declared only if the critical method 2 option is specified within the header file: `VSMM_cfg.h`.[1]

```
unsigned long int guliDSPSR
```

The *guliDSPSR* global variable is used only by critical method 2 to store the DSP's core status register, which contains the current state of interrupts. When a critical area is exited, the value in the guliDSPSR is evaluated, and the interrupts are restored to their previous state.

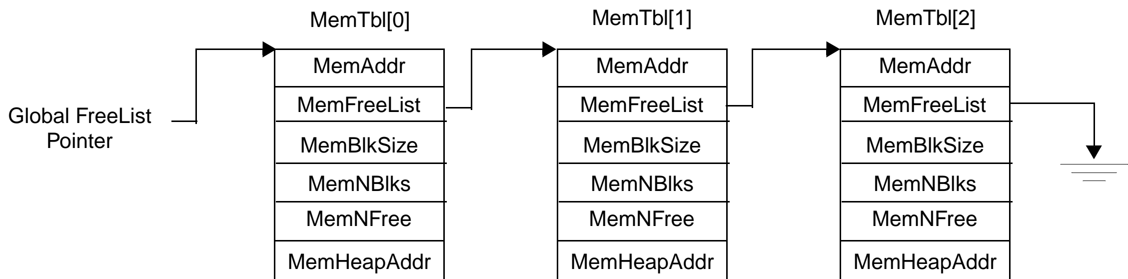


Figure 8. Global FreeList Pointer and MemTbl

[1] For details on the possible critical methods, refer to **Section 3.6**, *VSMM Critical Methods,* on page 14 or the *VSMM User's Guide* [4].

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

2.5 Data Control Block

The data control block (DCB) is a 28-byte VSMM object used by the VSMM query routine to store specific information about a heap. The VSMM DCB is defined as follows:

```
typedef struct {
  void    *pvVSMMAddr;
  void    *pvVSMMFreeList;
  INT32U  uliVSMMBlkSize;
  INT32U  uliVSMMNBlks;
  INT32U  uliVSMMNFree;
  INT32U  uliVSMMNUsed;
  INT32U  uliVSMMHeapAddr;
} t_VSMM_MEM_DATA;
```

Table 3. VSMM Data Control Block Definition

Variable	Description
VSMMAddr	A pointer to the base address of a heap.
VSMMFreeList	A pointer to the next available free memory block within a heap.
VSMMBlkSize	The size of each block contained within a heap.
VSMMNBlks	The number of blocks contained within a heap.
VSMMNFree	The number of memory blocks that are still available within a heap.
VSMMNUsed	The number of memory blocks in use.
VSMMHeapAddr	A pointer to a parent heap. If the heap associated with this MCB was created directly from free memory, the value is NULL. Otherwise, if the heap was created from a memory block associated with another heap, the value is this heap's handle or address.

3 VSMM Distribution

The VSMM distribution code is an official release that is available to all Motorola customers via their marketing representative and the Motorola website listed on the back of this document. It contains all the necessary components to build an application using the VSMM routines within an RTOS or non-RTOS environment running the StarCore MSC81xx family of DSPs. **Table 4** lists the items supplied with VSMM distribution. The remainder of this section discusses the components provided in this distribution.

Table 4. VSMM Released Items

Item Description	Item Identification
VSMM Reference Manual	VSMM_ReferenceManual.doc
VSMM User's Guide	VSMM_UsersGuide.doc
VSMM Interface Control Documents	<routine>_ICD.doc where <routine> is one of the following: • Init • Create • AllocCreate • Destroy • Alloc • Free • Query
VSMM Executive Summary	VSMMPresentation.ppt

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 4. VSMM Released Items (Continued)

Item Description	Item Identification
VSMM Object Library	VSMMLibrary1_3.elb
VSMM Configuration Source File	VSMM_cfg.c
VSMM Configuration Header File	VSMM_cfg.h
VSMM DSP Setup File	VSMM_dsp.h
VSMM General Header File	VSMM.h
VSMM Master Include File	VSMM_Includes.h
VSMM Enter Critical Method 2 Source	VSMMEnterCritical.asm
VSMM Exit Critical Method 2 Source	VSMMExitCritical.asm
VSMM Non-RTOS Example Archive	VSMMExamplesCR1_3.zip
VSMM OSEck RTOS Example Archive	VSMMExampleRTOS2.zip

Freescale Semiconductor, Inc.

3.1 VSMM Documentation

The *VSMM Reference Manual*[5] describes the common symptoms of general memory managers and presents a high-level overview of the VSMM architecture and a reference guide to the VSMM features. A *VSMM User's Guide*[4] provides additional information for tailoring and incorporating VSMM in your applications. An Interface Control Document (ICD) is also provided for each VSMM routine. These ICDs provide the following information on each routine:

- *Function prototype*. Function name and parameter list.
- *Functional description*. Brief summary of the primary function of each module.
- *Interface*. Lists all programmer-defined data types used by this module.
- *Argument description*. Lists arguments and briefly describes the data types defined within the interface section.
- *Design and implementation notes*. Summarizes algorithms, handling methods, and implementation issues.
- *Performance*. Describes code size, data size, and cycles required.
- *Precision*. Describes any precision limitations

3.2 VSMM Object Library

The primary component of the VSMM distribution is the VSMM object library, `VSMMLibrary1_3.elb`. This library contains the entire suite of VSMM routines and was built using Metrowerks® Codewarrior® v2.02 targeted for the Motorola StarCore DSPs. The library was built using Level 3 optimization, which implements several general, target-independent optimizations. The output from the target-independent optimizations is linear assembly code. The Codewarrior librarian grouped these object files into a linkable library.

3.3 VSMM Configuration 'C' Source File

The VSMM 'C' source file, `VSMM_cfg.c`, is provided so that programmers can tailor VSMM to their applications. Configurable parameters supported by the use of this file include:

- *Enabling or disabling parameter verification within each VSMM routine.* Setting the *gucVSMM_ARG_CHK_EN* global constant to 1 enables parameter verification, which ensures that all

parameters passed to a VSMM routine are within allowable range. Setting this global constant to 0 disables this feature, thus reducing the cycle overhead and code footprint of each VSMM routine.

- *Critical entry and exit routines*. Although the release provides several critical methods, you can add or modify these on a per application basis. For details on the critical methods provided with this release, see **Section 3.6**, *VSMM Critical Methods,* on page 14.

In addition to these configurable parameters, the `VSMM_cfg.c` source file also contains three global VSMM declarations. Do not edit these declarations because VSMM execution may become unpredictable. These global declarations must reside here to allow the additional configuration capabilities described within the next section. These global declarations are:

- **gpstVSMMMemFreeList*. A global pointer to a free list of memory partitions.
- *gastVSMMMemTbl[VSMM_MAX_MEM_PART]*. This is a global array of memory control blocks used by VSMM. See the next section for details on the VSMM_MAX_MEM_PART constant.
- *guliDSPSR*. Storage for DSP status register contents. This global is declared only if Critical Method 2 is selected and used to store the contents of the DSP status register prior to entering a critical section. The value stored within this global variable is tested when a critical section is exited to determine the state to which the interrupts are to be set. If the interrupts are already disabled before a critical section is entered, the interrupts remain disabled during the exit. Otherwise, interrupts are re-enabled.

3.4 VSMM Configuration Header File

The `VSMM_cfg.h` file is a standard 'C' header file to provide additional VSMM configuration capabilities. Configurable items within this file include the following three constants:

- *OSE_RTOS*. Defining this constant to a value of 1 indicates to that VSMM that it is to be used in an OSEck RTOS environment. This flag causes the alternate critical methods associated with methods 1 or 2 to be built in. These alternate methods use OSEck RTOS macros to enter and exit critical sections.
- *VSMM_CRITICAL_METHOD*. Specifies the desired critical method to be utilized for entering and exiting critical sections.VSMM provides several default methods and allows user-defined methods.
- *VSMM_MAX_MEM_PART*. Specifies the maximum number of partitions allowed within your application. The VSMM requires at least two partitions, also called heaps, and allows an indefinite number of heaps, based only on the amount of physical memory available. Each heap requires 24 bytes of overhead. See **Section 8.1**, *Configuration Header File, VSMM_cfg.h,* on page 28 for a listing of `VSMM_cfg.h`.

3.5 VSMM Additional 'C' Header Files

Three additional 'C' header files provided with the VSMM release are as follows:

- `VSMM_dsp.h`. Defines DSP-specific constants such as data types and byte alignment macro.
- `VSMM.h`. Defines VSMM error codes, function prototyping, and VSMM data structure types.
- `VSMM_Includes.h`. A master include file for VSMM that contains all of the "includes" required by the VSMM modules. The order of the includes within this file is important and should never be modified.

Use caution when editing any of these header files because they are critical to VSMM operation.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

3.6 VSMM Critical Methods

Many VSMM routines have critical sections, in which the program is vulnerable to undesirable interruption or corruption and therefore requires interrupts to be disabled during execution of these sections. The VSMM release provides several default critical methods associated with disabling and re-enabling interrupts upon entry and exit of critical sections or the use of a binary semaphore, also referred to as a spin-lock. Most of the VSMM critical methods are defined in the `VSMM_cfg.c` source file.[2] The exception is the enter and exit critical methods associated with critical method 2. The `VSMMEnterCritical.asm` and `VSMMExitCritical.asm` assembly files provide the entry and exit routines for critical method 2. These two files must be included in your application project file if you have set the #define constant VSMM_CRITICAL_METHOD to a value of 2 within the `VSMM_cfg.h` header file and the OSE_RTOS #define constant is defined as 0. A developer may also create his or her own critical methods. The critical methods are as follows:

- *Critical Method 1*. Two available options are determined by the OSE_RTOS constant. Upon exit of a critical section, both options always re-enable interrupts even if they are disabled before entry into the critical section. If the OSE_RTOS constant is set to 1 within the `VSMM_cfg.h` header file, the OSEck RTOS LOCK macro disables interrupts, and the OSEck RTOS RESTORE macro re-enables interrupts. If OSE_RTOS is set to 0, the StarCore DSP assembly **di** instruction disables interrupts and the **ei** instruction re-enables interrupts.
- *Critical Method 2*. Two available options are determined by the OSE_RTOS constant. Both options save the current interrupt state before interrupts are disabled upon entry into critical sections and restore the interrupts to their saved state when critical sections are exited. If the OSE_RTOS constant is set to 1, the OSEck RTOS LOCK_SAVE macro stores the interrupt state and disables interrupts, and the OSEck RTOS LOCK_RESTORE macro restores the saved interrupt state. If OSE_RTOS is set to 0, the critical method assembly routine in the `VSMMEnterCritical.asm` file: saves the interrupt state into a global variable named: guliDSPSR and disables the interrupts. The assembly routine in the `VSMMExitCritical.asm` file restores the interrupt state by testing the saved status register interrupt bit within the guliDSPSR global variable, and, if the bit is cleared, re-enables the interrupts. Otherwise, the interrupts remain disabled. Note that critical method 2 consumes the most cycles of all the default methods provided.
- *Critical Method 3*. This method disables interrupts, adjusts the interrupt priority mask (IPM), and re-enables interrupts when a critical section is entered. When a critical section is exited, interrupts are disabled, the IPM is restored, and interrupts are re-enabled. This method is useful when you want to allow higher-priority processes, such as an OS kernel, the ability to continue to generate and service interrupts. It is extremely important that these higher-priority processes do not access any VSMM routines since doing so could potentially interrupt one of these routines while it is within a critical section. If you are using VSMM within an unmasked priority process, it is highly recommended that you either use an alternate critical method or place an access function around the VSMM function call to ensure mutual exclusivity.
- *Critical Method 4*. This method can be used only with the OSE Systems OSEck RTOS because it uses the OSE spin-lock mechanism. This mechanism is a binary semaphore that can be shared across cores on a DSP such as the MSC8102. To use this critical method, the VSMM_CRITICAL_METHOD #define constant must be set to 4, and the OSE_RTOS #define constant must be set to 1. You must also initialize a spin-lock for VSMM critical sections. Using OSEck, add the following code snippet to your start handler 2 routine. If you are not using a start handler, simply add this code snippet to a module so that it executes only once, for the life of the program and at program start-up.

[2] See **Section 8.2**, *Configuration 'C' Source File, VSMM_cfg.c,* on page 29 for a listing of this file.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

```
#if      ((VSMM_CRITICAL_METHOD == 4) && (OSE_RTOS > 0))
       // allocate a spinlock for critical method & release same
       bsp_spinlock_release((gpusiCMSpinLock = bsp_spinlock_alloc()));
#endif
```

Ensure that the OSEck `bspspinlock.h` header file is included in your module containing this code snippet.

4 Building Your Application with VSMM

This section describes the steps necessary to include the VSMM functionality within your application, and it concludes with examples. For additional details on the VSMM, refer to the *VSMM Reference Manual* [5].

4.1 Initial Configuration

The first step to perform when using the VSMM is to gain an understanding of your application and its memory requirements so that your application uses the available memory most efficiently. **Section 3.3**, *VSMM Configuration 'C' Source File,* on page 12 describes the configurable VSMM parameters and the files requiring modifications based on your criteria. Edit these files carefully and save them when done. The minimum recommended modifications are to set the maximum number of partitions or heaps that your system requires, keeping in mind that the VSMM requires at least two heaps and each heap requires a minimum of two memory blocks.

Perform the following steps:

1. Set the maximum number of heaps by editing the VSMM_MAX_MEM_PART #define constant in the `VSMM_cfg.h` header file.

   Recall that the memory control block associated with each heap requires 24-bytes of memory.

2. In `VSMM_cfg.h`, specify the critical method for entering and exiting critical sections. Edit the VSMM_CRTICAL_METHOD #define constant appropriately.

   Of course, you can add your own critical methods and/or enable the OSE_RTOS #define constant to use the OSEck RTOS critical method macros or spin-lock. To use the OSEck RTOS critical method macros/spin-locks, edit the OSE_RTOS #define constant to a value of 1 or 0 if you do not wish to use these macros. Refer to **Section 7**, *Building VSMM Examples 1–3,* on page 26 for a source listing of `VSMM_cfg.h`.

3. Edit the `VSMM_cfg.c` configuration source file if you want to change the current VSMM argument verification mode.

   To enable VSMM argument verification, edit the gucVSMM_ARG_CHK_EN global variable to a value of 1; to disable argument verification clear this variable to 0. Do not modify this global variable at run time since a mutex is not assigned to it. You may also add your own critical methods within this module or modify those that are here. Refer to **Section 7**, *Building VSMM Examples 1–3,* on page 26 for a source listing of `VSMM_cfg.c`.

4.2 Project Set-up

Your application must include the VSMM master include header file, `VSMM_Includes.h`. This master header file contains all VSMM header files required to build your application with the VSMM library. The order of the #includes within this file is important, so do not edit this file. Like all header files, this file contains a mechanism to prevent multiple inclusions.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

Your application project must also include the VSMM_cfg.c source file. You are required to add the two assembly files VSMMEnterCritical.asm and VSMMExitCritical.asm if you have elected to use the default non-OSEck RTOS critical method 2 routines.

You must add a call to the VSMM routine *VSMMMemInit()* to your application at a point before any VSMM routines are called. *VSMMMemInit()* must execute once within your system before your application can use any of the VSMM features. After this initialization routine executes, VSMM is available for use by your application.

4.3 Creating Heaps

Once VSMM is enabled, you can create any necessary memory partitions (heaps), from which all allocations are performed. Before creating the heaps, you must decide whether to allocate specific statically declared areas of memory from which the heaps are created (see **Figure 9**) or to use the system heap space, as illustrated in **Figure 10**. If you chose to create heaps from several statically declared memory areas, you must accurately define the size of these areas. If you opt to use the system heap area, the system stack and system heap often share the same memory area, so the stack address runs up while the heap runs down. It is essential to ensure that these two pointers never overlap.

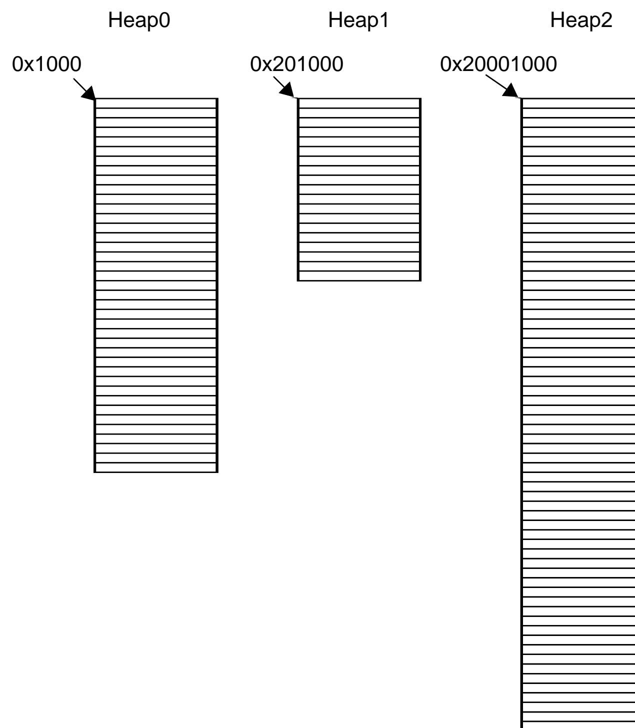


Figure 9. Statically Declared Heap Areas

Freescale Semiconductor, Inc.

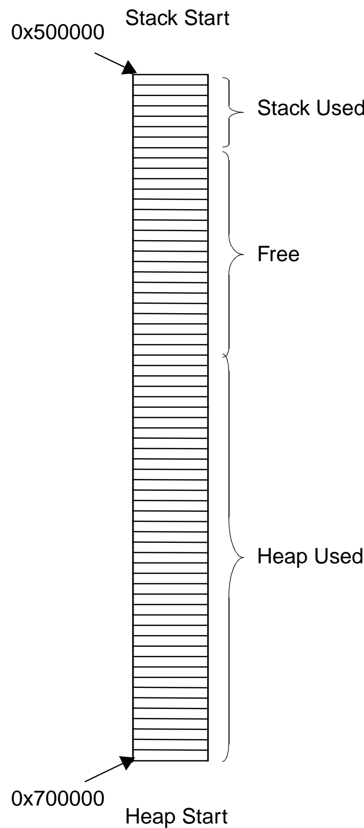


Figure 10. System Stack and Heap Space

After identifying the memory block sizes an application requires, the worst-case allocation of these blocks, and the location of the heaps, you are finally ready to allocate the heaps. It is not necessary to allocate the heaps statically all at once. VSMM allows dynamic creation of heaps so that the programmer can eliminate a heap with a memory block of one size and create another heap with a different memory block size. In addition, VSMM allows programmers to partition a memory block of a heap, therefore creating a second heap with memory blocks of a smaller size than the heap from which the block was allocated. A heap cannot be eliminated until all of its associated memory blocks are freed.

To determine the memory usage the VSMM requires for your heaps, you must consider the overhead associated with the VSMM. The overhead for each heap is 24 bytes for the MCB. Therefore, if you defined the VSMM_MAX_MEM_PART #define constant to a value of five times the total memory for the MCBs, $5 \times$ 24-bytes equals 120 bytes.

By default, the memory area in which the MCB allocations occur extends from either your designated system's data space, if statically declared, or from the system stack space if locally declared. It does not extend from the designated system heap space because the heap space is consumed only through dynamic memory allocations. The overhead associated with each memory block is at least 8 bytes (VSMM_MEMBLK_HDR_SIZE). However, if your block size is not 8-byte aligned, additional overhead is added to ensure this alignment. BALIGN is a VSMM macro that returns an equivalent 8-byte aligned block size based on the block size specified:

```
actual block size (ABS) = BALIGN(desired block size (in bytes))
```

Therefore to determine the total heap size, use the following algorithm:

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

```
total heap size (THS) = number of blocks * (ABS + VSMM_MEMBLK_HDR_SIZE)
```

For example, suppose we are creating heaps for an application that requires at most:

- four 16 KB blocks or twenty 774 byte blocks
- or three 16 KB blocks and fifteen 774 byte blocks
- or two 16 KB blocks and ten 774 byte blocks

We create an initial heap of four 16 KB blocks from a specified memory area and then create a second heap from a memory block of the first heap. We want to fit 20 × 774 blocks within this second heap, but we recall that each memory block has an additional overhead of 8 bytes (VSMM_MEMBLK_HDR_SIZE) plus any byte-alignment performed. We verify that we can fit 20 × 774 blocks within a single 16 KB block, as follows:

```
ABS = BALIGN(774) = 776-bytes
THS = 20 * (776+ VSMM_MEMBLK_HDR_SIZE) = 15,680 or approximately 15.31 KB
```

Having verified that heap 2 can indeed be created from the heap 1 memory block, we create our heap memory area from which heap 1 is to be created. Notice that in this example we are not using the system heap area but rather a specified area of memory within the data space.

```
// Define Heap 1's Memory Block Size
#define HEAP1_BLOCK_SIZE (BALIGN(16384))  // ABS = 16384

// Define Heap 2's Memory Block Size
#define HEAP2_BLOCK_SIZE (BALIGN(774))   // ABS = 776

//  Heap1[# blocks] [Actual Block Size + 8-byte Overhead]
unsigned char ucHeap1[4] [(HEAP1_BLOCK_SIZE + VSMM_MEMBLK_HDR_SIZE)];
```

With the memory area defined we can now create our heaps, as follows:

```
//  VSMMMemCreate(HeapMCB, # blocks, actual block size, errorCode)
pstHeap1Ptr = VSMMMemCreate(ucHeap1, 4, HEAP1_BLOCK_SIZE, &ucErrCode);

//  VSMMMemAllocCreate(HeapMCB, # blocks, actual block size, errorCode)
pstHeap2Ptr = VSMMMemAllocCreate(pstHeap1Ptr,  20, HEAP2_BLOCK_SIZE, &ucErrCode);
```

For an actual application, we would typically not create heap 2 until we need it, and we would destroy it when it is no longer needed. However, for purposes of illustration, we create it here at the same time we create heap 1. Notice that 16384 is already 8-byte aligned, so the ABS for this block size is 16384.

4.4 Allocating Memory

After the heaps are created, we begin allocating memory to them. We call the VSMMMemAlloc routine, passing it a pointer to the heap MCB from which the block is to be allocated and a storage area for the returned error code. It is good practice to validate that the allocation succeeded.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
// Allocate a Memory Block from Heap2
apucBlockPtrArray2[0] = (unsigned char *) VSMMMemAlloc(
                                        pstHeap2Ptr,
                                        &ucErrCode);
// Verify allocation was a success
if(ucErrCode != VSMM_NO_ERR)
{
  printf("ERROR %d -- Unable to Allocate Block from Heap2\n",
            ucErrCode);
  asm(" debug");
}
```

Upon successful completion of this call, apucBlockPtrArray[0] points to the start of a memory block of size 784 bytes, of which 776 bytes is available for use by the application with the remaining 8 bytes used by VSMM to manage this block. **Figure 5** on page 8 illustrates a memory block allocation.

4.5 Freeing Memory

Freeing a memory block is handled much like the standard C free call. We call the *VSMMMemFree* routine, passing it the pointer to the memory block to be freed.

```
if((ucErrCode = VSMMMemFree(apucBlockPtrArray2[0])) != VSMM_NO_ERR)
{
  printf("ERROR %d -- Unable to Free Memory Block \n",ucErrCode);
  asm(" debug");
}
```

As with memory allocation, it is good practice to verify that the operation succeeded.

4.6 Eliminating Heaps

Heaps can be eliminated at run-time as long as all their memory blocks are freed. To eliminate a heap, we call the *VSMMMemDestroy* routine, passing it a pointer to the MCB of the heap to be removed. Remember to verify that the operation succeeded.

```
if((ucErrCode = VSMMMemDestroy(pstHeap2Ptr)) != VSMM_NO_ERR)
{
  printf("ERROR %d -- Unable to Destroy Heap 2\n",ucErrCode);
  asm(" debug");
}
```

4.7 Querying Heaps

To obtain information on a heap, we call the *VSMMMemQuery* routine and pass it a pointer to the MCB of a specified heap and a pointer to a data control block. *VSMMMemQuery* returns a success status so there is no need to perform a separate verification.

```
ucErrCode = VSMMMemQuery(pstHeap2Ptr, &stPartQ);
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

4.8 Examples

This section introduces four examples that illustrate the configuration and use of VSMM. Three of these examples run within a non-RTOS environment on the Motorola MSC8101 Application Development System (MSC8101ADS), and the fourth uses the OSEck RTOS. See **Section 7** for instructions on how to build, download, and execute these examples.

The following software and tools are required in order to build and run the three non-RTOS examples:

- PC running Windows 9x, NT, or Win2000 or Solaris Platform
- Metrowerks Codewarrior v2.02 targeted for Motorola StarCore DSPs
- VSMM `Examples1_3.zip`, an archive containing the necessary files to build each of the three non-RTOS examples
- Motorola MSC8101ADS board and appropriate command converter and inter-connecting cable

The following software and tools are required in order to build and run the VSMM RTOS example:

- PC running Windows 9x, NT, or Win2000 or Solaris Platform
- Metrowerks Codewarrior v2.02 Targeted for Motorola StarCore DSPs
- VSMM `ExampleRTOS2.zip`, an archive containing the necessary files to build the RTOS example
- OSEck RTOS v3.1.0.6 or later and OSE Heap Manager v2.0.0.2 or later if you desire to compare the OSE heap with that of the VSMM.
- Motorola MSC8101ADS board and appropriate command converter and inter-connecting cable.

Simply download and execute the supplied RTOS example without the need for the OSEck RTOS and heap manager libraries. The VSMM examples follow:

- Example 1 demonstrates the creation of heaps from specified memory areas as well as from an allocated memory block of another heap. It also demonstrates how to allocate memory blocks, write data to these allocated blocks, free memory blocks, destroy heaps, and query heaps.
- Example 2 demonstrates many of the same features as Example 1 but uses the system heap space instead of creating heaps from a specified statically declared memory area.
- Example 3 illustrates how to incorporate the two critical method 2 assembly files into an application (refer to **Section 3.6**, *VSMM Critical Methods,* on page 14). The only difference between this example and Example 1 is that the VSMM_CRITICAL_METHOD #define constant in `VSMM_cfg.h` is set to 2 and the `VSMMEnterCritical.asm` and `VSMMExitCritical.asm` critical method assembly files are added to the example 3 project sources.
- Example RTOS shows how the VSMM operates within an RTOS environment and uses the spin-lock critical methods. The RTOS is OSEck and the OSE heap manager targeted for the Motorola MSC8101 DSP. Because of OSE Systems licensing requirements, the OSEck and heap libraries are not supplied as part of this example.To modify and rebuild this example, you must have the OSEck and heap manager libraries installed on your system. Also, the Codewarrior project access paths and include file paths that must be updated to reflect your specific OSE installation paths. In addition to the OSE libraries, you must also use the OSE make utility to generate the `appcon.c` and `appcon_asm.asm` files from the supplied `appcon.con` configuration file.

If you have access to the OSE libraries required by this example, you can switch the ExampleRTOS2 between the OSE heap manager and the VSMM, as follows:

- Define the preprocessor VSMM macro within the Codewarrior project to use VSMM.
- Remove the preprocessor VSMM macro definition to use the OSE heap manager.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

The ExampleRTOS2 binary, `ExampleRTOS2.eld`, was built with the VSMM preprocessor macro defined, so only VSMM is used for all memory allocations within the supplied executable. Download ExampleRTOS2 without requiring a build, as follows:

1. Select **Preferences** from the Codewarrior IDE Edit menu.
2. From within the resulting window select **Build Setting** from under **General**.
3. Set the **Build before running** statement to **Never**.
4. Select **OK** to accept your changes and close this window.
5. Download the example by clicking on the Run Debug icon. Refer to **Figure 11** on page 26 to learn the location of this icon.

5 VSMM Benchmarking

This section describes the VSMM performance and memory requirements. Since the VSMM can be tailored for a specific application, we used Metrowerks Codewarrior v2.02 Level 3 optimization to build a simple application that exercises all VSMM functionality. **Table 5** details the functionality exercised and the associated cycle counts. The cycle counts were recorded via the MSC8101 EOnCE with the application running on a Motorola MSC8101ADS board. The cycles reported for the VSMM in **Table 5** are the results when the following are true:

- Argument verification is used
- gucVSMM_ARG_CHK_EN = 1
- Critical method 1, VSMM_CRITICAL_METHOD = 1
- The 20 cycles associated with the EOnCE overhead are included. These cycles are slightly higher if critical method 2 or 3 is used, with critical method 2 resulting in the highest number of cycles. See **Table 9** for the total task cycle numbers with critical method 2 selected.

Table 5. Memory Manager Task Cycles

Task	VSMM Function	VSMM Cycles
Memory initialization	VSMMMemInit	30
Static heap creation	VSMMMemCreate	147
Dynamic heap creation	VSMMMemAllocCreate	220
Dynamic heap destruction	VSMMMemDestroy	159
Memory allocation	VSMMMemAlloc	71
Memory free	VSMMMemFree	68
Memory query	VSMMMemQuery	68

Table 6 shows the VSMM code space requirements to achieve the tested functionalities. The recorded numbers are from the resulting MAP file with Level 3optimization applied. The VSMM code space size shown in **Table 6** is with argument verification enabled and critical method 1 selected. The worst case code space requirement occurs when critical method 2 is used, which increases the code space size by 32 bytes to a total of 1,460 bytes.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 6. Code Space Requirements

Manager	Code Space Requirements (Bytes)
VSMM	1,428
OSE-HM	2,352

Another important aspect of the VSMM is the overall memory efficiency. For this portion of our benchmarking, we statically created two heaps of sufficient size to allocate thirty-two 230-byte buffers and four 16 KB buffers. The number of memory blocks and their associated sizes were chosen because they represent actual numbers used within the Motorola 3G Symbol Rate Real-Time Demo System. **Table 7** details the VSMM heap requirements to support these buffer allocations.

Table 7. Heap Size Requirements

Heap	VSMM Heap Size (Bytes)
230-byte partitions	7,704
16 KB partitions	65,592
Total Memory	73,296

The overhead associated with the memory sizes noted in **Table 7** is listed in **Table 8**. VSMM adds 8 bytes to each allocated memory block (buffer), so if you allocate a memory block of 1,024 bytes within the VSMM, the heap requires a memory block of at least 1,032 bytes.

Table 8. Associated Overhead

Type	VSMM (Bytes)	OSE-HM (Bytes)
Per heap	24	2,048
Per memory block	8	16

The VSMM always performs an 8-byte alignment of the memory blocks, so when you create a heap with block sizes of 230 bytes, the actual block sizes created are 232 bytes. Therefore, if you allocate a buffer of 230 bytes, the VSMM returns a buffer of 240 bytes (including 232 bytes plus the 8-byte overhead). However, only 232 bytes of this returned buffer are available to your application. The VSMM uses the remaining 8 bytes to define the memory block heap.

Table 9 presents the total cycles for each VSMM task when critical method 2 is used. As noted earlier, critical method 2 is the most cycle intensive of the five predefined critical methods. **Table 10** presents the overall statistics for each VSMM module. The C module cycle numbers were collected with argument verification enabled, critical method 1 defined, and Level 3 optimization applied. The VSMM requires a minimum of two heaps, and each heap requires a minimum of two memory blocks. Therefore, the minimum memory overhead is 24 bytes per heap plus 8 bytes per memory block $(2 \times 24) + (4 \times 8) = 80$ bytes.

Table 9. VSMM Task Cycles With Critical Method 2 Enabled

Task	VSMM Cycles
Memory initialization	30
Static heap creation	167

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

Table 9. VSMM Task Cycles With Critical Method 2 Enabled (Continued)

Task	VSMM Cycles
Dynamic heap creation	250
Dynamic heap destruction	189
Memory allocation	81
Memory free	78
Memory query	78

Table 10. VSMM Module Statistics

Module	Cycles	Code Size (Bytes)	Data Size (Bytes)
VSMMMemInit.c	30	128	0
VSMMMemCreate.c	147	308	0
VSMMMemAllocCreate.c	220	416	0
VSMMMemDestroy.c	159	240	0
VSMMMemAlloc.c	71	160	0
VSMMMemFree.c	68	128	0
VSMMMemQuery.c	68	144	0
VSMMEnterCritical.asm	31	18	0
VSMMExitCritical.asm	31	14	0

6 VSSM Functions

This section describes the VSMM functions.

6.1 VSMMMemInit()

Syntax

```
void VSMMMemInit(void)
```

Description Initializes the VSMM supporting data structures and global variables.

Parameters None

Return Value None

Error Checks None

Example Usage

```
VSMMMemInit();
```

6.2 VSMMMemCreate()

Syntax

```
t_VSMM_MEM  *VSMMMemCreate (void *pvAddr,
                            INT32U uliNBlks,
                            INT32U uliBlkSize,
                            INT8U *pucErr)
```

Description Creates a fixed-size memory partition (heap).

Parameters **pvAddr*. A pointer to the starting address of the heap.

uliNBlks. The number of memory blocks to create within this heap.

For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

uliBlkSize. The size, in bytes, of each memory block within this heap.

**pucErr*. A pointer to an error message that is set by this function.

Return Value Address of a memory control block if the heap is created; otherwise (t_VSSM_MEM *)0

Error Checks VSMM_NO_ERR. If the heap is successfully created.

VSMM_MEM_INVALID_PART. No free partitions (heaps) available.

VSMM_MEM_INVALID_BLKS. The user has specified an invalid number of blocks (must be >= 2).

VSMM_MEM_INVALID_SIZE. The user has specified an invalid block size (must be greater than the size of a pointer).

Example Usage

```
pstHeap1Ptr = VSMMMemCreate(aucHeap1, 4, 16384, &ucErrCode);
```

6.3 VSMMMemAllocCreate()

Syntax

```
t_VSMM_MEM  *VSMMMemAllocCreate (
                               t_VSMM_MEM *pstPMem,
                               INT32U uliNBlks,
                               INT32U uliBlkSize,
                               INT8U *pucErr)
```

Description Allocates a memory block from the specified partition and, if successful, creates a fixed-sized memory partition from the allocated memory block.

Parameters **pstPMem*. A pointer to the memory control block from which the memory block allocation occurs.

uliNBlks. The number of memory blocks to create within this heap.

uliBlkSize. The size, in bytes, of each memory block within this heap.

**pucErr*. A pointer to an error message that is set by this function.

Return Value Address of a memory control block if the heap is created; otherwise (t_VSMM_MEM *)0

Error Checks VSMM_NO_ERR if the heap is successfully created.

VSMM_MEM_INVALID_PART. No free partitions (heaps) available.

VSMM_MEM_INVALID_BLKS. The user specified an invalid number of blocks (must be >= 2).

VSMM_MEM_INVALID_SIZE. The user specified an invalid block size (must be greater than the size of a pointer).

Example Usage

```
pstHeap2Ptr = VSMMMemAllocCreate(pstHeap1Ptr, 4, 16384, &ucErrCode);
```

6.4 VSMMMemAlloc()

Syntax

```
void  *VSMMMemAlloc (t_VSMM_MEM *pstPMem,
                               INT8U *ucErr)
```

Description Allocates a memory block from the specified heap.

Parameters **pstPMem*. A pointer to the MCB of the specified heap.

**pucErr.* A pointer to an error message that is set by this function.

Return Value Address of a memory block if the allocation is successful; otherwise (void *)0

Error Checks VSMM_NO_ERR. If the memory block allocation is successful.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

VSMM_MEM_NO_FREE_BLKS. If there are no more free memory blocks, within this heap, to allocate to the caller.

Example Usage

```
apucBlockPtrArray2[0] = \
(unsigned char *) VSMMMemAlloc(pstHeap2Ptr, &ucErrCode);
```

6.5 VSMMMemFree()

Syntax

```
INT8U  VSMMMemFree (void *pvPBlk)
```

Description Determines heap of the specified memory block and returns the memory block to the identified heap.

Parameters **pvPBlk*. A pointer to the memory block being freed.

Return Value VSMM_NO_ERR. If the memory block is successfully freed.

VSMM_MEM_FULL. If you are returning a memory block to an already FULL heap.

Error Checks VSMM_NO_ERR. If the memory block was successfully freed.

VSMM_MEM_FULL. If you are returning a memory block to an already FULL heap.

Example Usage

```
ucErrCode = VSMMMemFree(apucBlockPtrArray2[0]);
```

6.6 VSMMMemDestroy()

Syntax

```
INT8U  VSMMMemDestroy (t_VSMM_MEM *pstPMem)
```

Description Removes a heap from the system and if the eliminated heap was created from a memory block belonging to another heap, this memory block is freed as well.

Parameters **pstPMem*. A pointer to the memory control block of the heap being destroyed.

Return Value VSMM_NO_ERR. If the memory block is successfully freed.

VSMM_MEM_INVALID_PART. Partition does not exist.

Error Checks VSMM_NO_ERR. If the memory block is successfully freed.

VSMM_MEM_INVALID_PART. Partition does not exist.

Example Usage

```
ucErrCode = VSMMMemDestroy(pstHeap2Ptr);
```

6.7 VSMMMemQuery()

Syntax

```
INT8U  VSMMMemQuery (t_VSMM_MEM *pstPMem,
t_VSMM_MEM_DATA *pstPData)
```

Description Reports the state of the specified heap by dumping the contents of its associated memory control block. Use this function to determine how many free memory blocks are still available, how many memory blocks are in use, and whether this heap was created from a memory block of another heap.

Parameters **pstPMem*. A pointer to the memory control block of the heap being queried.

**pstPData*. A pointer to the structure to contain the information about the specified heap.

Return Value VSMM_NO_ERR. Always returned.

Error Checks None

Example Usage

```
ucErrCode = VSMMMemQuery(pstHeap2Ptr, &stPartQ);
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

7 Building VSMM Examples 1–3

This section describes the Metrorwerks Codewarrior v2.02 project associated with building, downloading, and debugging VSMM examples 1 through 3. To build each example, open the project file entitled `VSMMExamplesCR.mcp`. If Metrowerks CodeWarrior v2.02 is properly installed on your system, the project file illustrated in **Figure 11** appears.

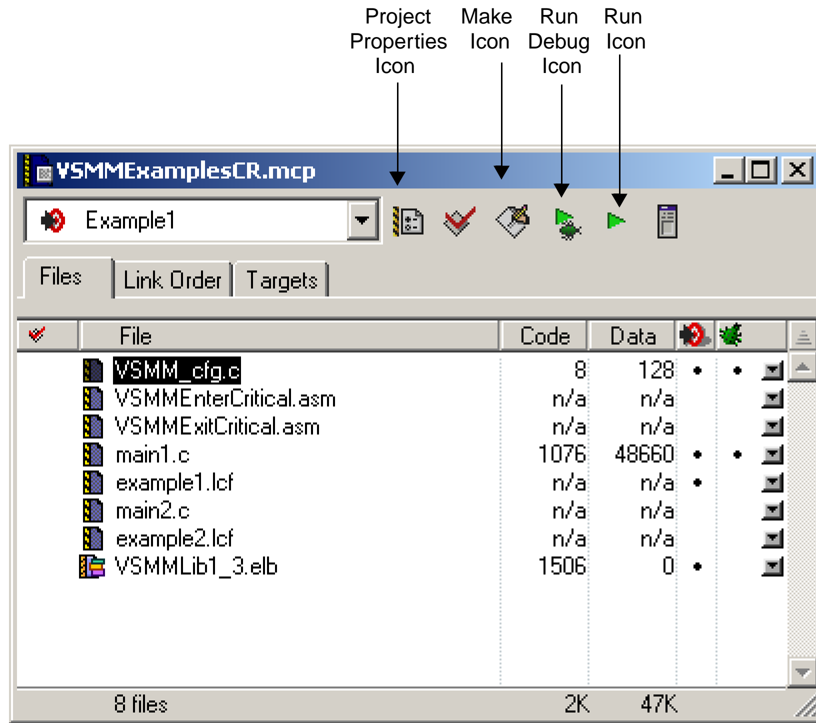


Figure 11. VSMM Example Project File Pane

Freescale Semiconductor, Inc.

Figure 12 shows the three targets within this project, Example1, Example2, and Critical Method 2.

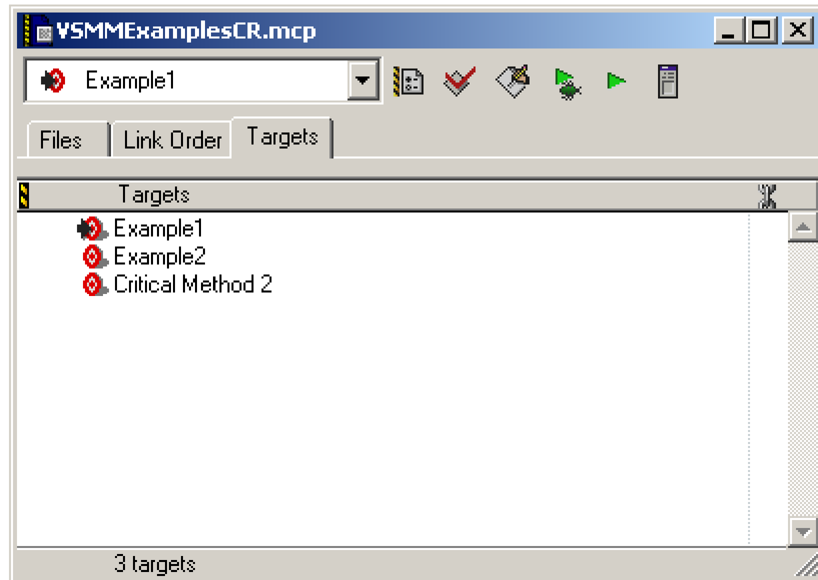


Figure 12. VSMM Example Project Target Pane

Example1 is built as follows:

1. Select this target from the project target pane.
2. Edit the `VSMM_cfg.h` file within your project directory
3. Ensure that the VSMM_CRITICAL_METHOD #define constant is set to either 1 or 3.
4. Ensure that the OSE_RTOS #define constant is set to 0.
5. Save the changes and click the Make icon.
6. After the build completes, click on the Run Debug icon to download and execute the example.

Example2 is built as follows:

1. Select this target from the project target pane.
2. Edit the `VSMM_cfg.h` file within your project directory.
3. Ensure that the VSMM_CRITICAL_METHOD #define constant is set to either 1 or 3.
4. Ensure that the OSE_RTOS #define constant is set to 0.
5. Save your changes and then click on the Make icon.
6. After the build completes, click on the Run Debug icon to download and execute this example.

Critical Method 2 is built as follows:

1. Select this target from the project target pane.
2. Edit the `VSMM_cfg.h` file within your project directory.
3. Ensure that the VSMM_CRITICAL_METHOD #define constant is set to 2.
4. Ensure that the OSE_RTOS #define constant is set to 0.
5. Save your changes and then click the Make icon.
6. After the build completes, click on the Run Debug icon to download and execute this example.

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

8 VSMM Configuration Source Listing

This section contains the source code for the VSMM configuration source files, which include the header file, `VSMM_cfg.h` and the 'C' source file `VSMM_cfg.c`. These two files allow customization of VSMM on an application-by-application basis. For details on customization, refer to the *VSMM Reference Manaual*[5].

8.1 Configuration Header File, VSMM_cfg.h

```
/*
***************************************************************************
* File : VSMM_CFG.H
* Description: Memory Manager Configuration
***************************************************************************
*/

#ifndef VSMM_CFG_H
#define VSMM_CFG_H

/*
***************************************************************************
*                                    CONFIGURATION
***************************************************************************
*/
// RTOS Defines
/* 1 = Used with OSEck RTOS; else    0                           */
#define OSE_RTOS                       0

// Maximum Number of MCBs to Create
/* Max. number of memory partitions; MUST be >= 2                 */
#define VSMM_MAX_MEM_PART           5

// Critical Method of Choice
/* Specifies the Critical Method that will be utilized by VSMM  */
#define  VSMM_CRITICAL_METHOD        1

/*
***************************************************************************
*                                   GLOBAL VARIABLES
***************************************************************************
*/

/* Enable (1) or Disable (0) argument checking */
extern unsigned char gucVSMM_ARG_CHK_EN;

/* Pointer to free list of memory partitions   */
extern  t_VSMM_MEM   *gpstVSMMMemFreeList;

/* Storage for memory partition manager        */
extern  t_VSMM_MEM    gastVSMMMemTbl[];

#if       VSMM_CRITICAL_METHOD == 2
/* Storage for DSP status register          */
extern unsigned long int guliDSPSR;
#endif
```

Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
/*
**************************************************************************
*                                FUNCTION PROTOTYPES
**************************************************************************
*/
#if VSMM_CRITICAL_METHOD == 2 && !OSE_RTOS
   void VSMM_ENTER_CRITICAL(void);
   void VSMM_EXIT_CRITICAL(void);
#endif

#endif   // end ifndef vsmm_cfg_h
// end vsmm_cfg.h
```

8.2 Configuration 'C' Source File, VSMM_cfg.c

```
/*
**************************************************************************
* File : VSMM_CFG.C
* Description: Memory Manager Configuration
**************************************************************************
*/

#include "VSMM_Includes.h"

/*
**************************************************************************
*                              CONFIGURATION
**************************************************************************
*/

#if      ((VSMM_CRITICAL_METHOD == 4) && (OSE_RTOS > 0))
   #include "bspspinlock.h"
   extern unsigned short int *gpusiCMSpinLock;
#endif

// System Defines
unsigned char gucVSMM_ARG_CHK_EN = 1; /* Enable (1) or Disable (0)
                                                   argument checking         */

/*
**************************************************************************
*                              GLOBAL VARIABLES
**************************************************************************
*/

/* Pointer to free list of memory partitions       */
t_VSMM_MEM    *gpstVSMMMemFreeList;

/* Storage for memory partition manager            */
t_VSMM_MEM    gastVSMMMemTbl[VSMM_MAX_MEM_PART];

#if       VSMM_CRITICAL_METHOD == 2
   /* Storage for DSP status register             */
   unsigned long int guliDSPSR;
#endif
```


Freescale Semiconductor, Inc.

For More Information On This Product,
Go to: www.freescale.com

```
/*
************************************************************************
*                              Motorola MSC8101 (Big Model)
*
* Method 1:  Disable/Enable interrupts using simple instructions.  After * critical
section, interrupts will be enabled even if they were disabled * before entering the
critical section.
*
* Method 2:  Disable/Enable interrupts by preserving the state of
* interrupts.  Generally speaking you would store the state of the
* interrupt disable flag in the local variable 'guliDSPSR' and then
* disable interrupts. You restore the interrupt disable state by testing
* the interrupt enable bit, in 'guliDSPSR', and if clear re-enable
* interrupts; else leave interrupts disabled.
*
* Method 3:  Disables interrupts, adjusts the Interrupt Priority Mask
* (IPM), and re-enables interrupts when entering a critical section. When
* exiting a critical section interrupts are disabled, the IPM restored,
* and interrupts re-enabled. This method is useful when you want
* higher-priority processes to continue to generate and service interrupts,
* i.e. an OS Kernel. Note that it is extremely important that these higher-
* prioritiy processes do not access any VSMM routines.
*
* Method 4: Utilizes an OSEck spin-lock (binary semaphore) to ensure
* mutual exclusivity. VSMM_CRITICAL_METHOD must be set to 4 and OSE_RTOS
* must be set to 1 to implement this method.
************************************************************************
*/
#if      VSMM_CRITICAL_METHOD == 1

#if OSE_RTOS > 0

void VSMM_ENTER_CRITICAL(void)
{

  LOCK            /* Disable interrupts */
}

void VSMM_EXIT_CRITICAL(void)
{
  UNLOCK       /* Enable  interrupts */
}

#else
void VSMM_ENTER_CRITICAL(void)
{
  asm  (" di");    /* Disable interrupts  */
}

void VSMM_EXIT_CRITICAL(void)
{
  asm  (" ei");   /* Enable  interrupts   */
}

#endif   // end ose_rtos > 0

#endif   // end vsmm_critical_method == 1
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

```
#if      VSMM_CRITICAL_METHOD == 2

#if     OSE_RTOS > 0
void VSMM_ENTER_CRITICAL(void)
{
LOCK_SAVE       /* Disable interrupts  */
}

void VSMM_EXIT_CRITICAL(void)
{
LOCK_RESTORE   /* Restore interrupts back to their prev state  */
}

#endif   // end ose_rtos > 0

#endif   // end vsmm_critical_method == 2

#if      VSMM_CRITICAL_METHOD == 3

// Disable all interrupts except those with a priority > 5.
void VSMM_ENTER_CRITICAL(void)
{
asm(" di");
asm(" bmclr #7<<5,SR.H");
asm(" bmset #5<<5,SR.H");
asm(" nop");
asm(" nop");
asm(" ei");
}

// Enable ALL Interrupts
void VSMM_EXIT_CRITICAL(void)
{
asm(" di");
asm(" bmclr #7<<5,SR.H");
asm(" nop");
asm(" nop");
asm(" ei");
}

#endif   // end vsmm_critical_method == 3

// if critical method == 4 & OSE_RTOS == 1 then use spin-locks
#if ((VSMM_CRITICAL_METHOD == 4) && (OSE_RTOS > 0))
void VSMM_ENTER_CRITICAL(void)
{
bsp_spinlock_aquire(gpusiCMSpinLock);
}

void VSMM_EXIT_CRITICAL(void)
{
bsp_spinlock_release(gpusiCMSpinLock);
}
#endif   // end vsmm_critical_method == 4

// end vsmm_cfg.c
```


For More Information On This Product,
Go to: www.freescale.com

Freescale Semiconductor, Inc.

9 References

[1] *Motorola SC100 Application Binary Interface Reference Manual*, (MNSC100ABI/D).

[2] *OSE for DSP Kernel Reference*, (420e/OSE133-1R 1.1).

[3] Mark S. Johnstone and Paul R. Wilson. “The Memory Fragmentation Problem: Solved?.” *International Symposium on Memory Management*, Vancouver, B.C., Canada, 1998.

[4] *Very Small Memory Manager (VSMM) User’s Guide*, Motorola, Rev. 1.4, 7/2002.

[5] *Very Small Memory Manager (VSMM) Reference Manual*, Motorola, Rev. 1.4, 7/2002.

[6] *Very Small Memory Manager (VSMM) Interface Control Documents (ICD)*, Motorola.

Freescale Semiconductor, Inc.

HOW TO REACH US:

USA / EUROPE / Locations Not Listed:

Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:

Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:

Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors/

Information in this document is provided solely to enable system and software implementers to use Motorola products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.

MOTOROLA and the Stylized M Logo are registered in the U.S. Patent and Trademark Office. OnCE and digital dna are trademarks of Motorola, Inc. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

© Motorola, Inc. 2003

AN2345/D, Rev. 1