



# *White Paper*

## **Using the RDM Server Resource Manager**

December 2001

---

## Preface

This White Paper contains a discussion of the Resource Manager portion of the RDM Server product. It explains the specific functions of Resource Manager, the various categories of the API (such as thread functions and queuing functions), and provides a detailed example of the use of many of the RM functions.

The White Paper will be of interest to Birdstep engineers who plan to work on the project, and to programmers who are potentially interested in purchasing RDM Server.

---

# Table of Contents

- Overview .....4
- General Purpose RM Usage.....7
- Thread Functions.....8
- Synchronization Functions .....9
- Queuing Functions..... 10
- Dynamic Memory Functions..... 11
- File Handling Functions ..... 11
- Example Program ..... 12

---

## Overview

The development of a multithread application requires the use of operating system functions to control threads and provide for inter-thread synchronization and communication. Although most modern operating systems support similar functionality, the manner for accessing this functionality is platform-dependent. Thus it is difficult to build sophisticated, platform-independent multithread application programs. If you do not require platform independence then you are certainly free to use your operating system functions directly. However, if your application needs to be scalable to many different operating system platforms, you should use the RDM Server Resource Manager (RM) interface. The multithread RDM Server database engine itself utilizes all of the functions in the RM interface. This provides you with the assurance that these functions are reliable and that they are available on all of the platforms RDM Server supports.

The RDM Server RM provides a rich platform-independent function interface to sophisticated operating system tasks such as those listed below:

- thread management
- concurrency control
- message queuing
- dynamic memory allocation
- file management

All the RM functions are designed to operate in a single process, multiple thread application environment. Note that use of these functions for multiple process applications will not work. They can be used in RDM Server extension modules or in application-specific database servers, which embed the RDM Server database engine.

A complete list of the RM interface functions is provided in the following table.

Table 1 - Resource Manager Function List

Function	Class	Description
rm_startup	general	Startup resource manager.
rm_cleanup	general	Shutdown and cleanup resource manager.
rm_sleep	general	Suspend thread for specified milliseconds.
rm_log	general	Write message to RDM Server message log.
rm_getenv	general	Get environment definition string.
rm_interrupt	general	Enable user interrupt control handling.
rm_threadBegin	thread	Begin execution of a new thread
rm_threadEnd	thread	End thread execution.
rm_syncCreate	synchronization	Create synchronization object
rm_syncDelete	synchronization	Delete synchronization object.
rm_syncEnterExcl	synchronization	Enter a n exclusive access (mutex semaphore) ) section
rm_syncExitExcl	synchronization	Exit and exclusive access (mutex semaphore) ) section
rm_syncStart	synchronization	Start event (set event semaphore to non-signaled).
rm_syncResume	synchronization	Resume eve nt (signal event semaphore).
rm_syncWait	synchronization	Wait for an event to be resumed (signaled).
rm_syncEnableQ	synchronization	Enable quiescent semaphore (let readers in).
rm_syncDisableQ	synchronization	Disable quiescent semaphore (block readers).
rm_syncSendQ	synchronization	Send reader exit notification on quiescent semaphore.
rm_syncReceiveQ	synchronization	Receive reader access permission on quiescent semaphore.
rm_syncWaitQ	synchronization	Wait for all readers to exit on quiescent semaphore .
rm_queueCreate	queuing	Create a message queue.
rm_queueDelete	queuing	Delete message queue.
rm_queuePurge	queuing	Purge messages from queue.
rm_queueWrite	queuing	Write message to queue.
rm_queueRead	queuing	Read message from queue.
rm_createTag	memory	Create a dynamic memory allocation tag.

rm_resetTag	memory	Reset tag's out-of-memory jump buffer.
rm_freeTagMemory	memory	Free all memory allocated on tag.
rm_getMemory	memory	Allocate a block of memory.
rm_cGetMemory	memory	Allocate and clear a block of memory.
rm_freeMemory	memory	Free allocated memory block.
rm_extendMemory	memory	Extend (or reallocate) a memory block.
rm_growMemory	memory	Grow memory block, if necessary.
rm_allocPool	memory	Allocate memory pool
rm_getFromPool	memory	Get buffer from pool
rm_putInPool	memory	Put buffer into pool
rm_zFreePool	memory	Free memory pool
rm_fileOpen	file	Open/Create a file.
rm_fileClose	file	Close a file.
rm_filePrintf	file	Write formatted output to file.
rm_fileSeek	file	Seek to absolute position in file.
rm_fileRead	file	Read from file.
rm_fileWrite	file	Write to file.
rm_fileSeekRead	file	Read from a specified location in file.
rm_fileSeekWrite	file	Write to a specified location in file.
rm_fileLength	file	Get the current length of a file.
rm_fileSync	file	Flush file to disk.
rm_fileRemove	file	Remove (delete) a file.
rm_fileRename	file	Rename a file.
rm_fileTempName	file	Create a unique temporary file name.

## General Purpose RM Usage

Some built-in data type and constant definitions are used by the RDM Server Resource Manager to provide platform independence. The data types include function declaration attributes and handle definitions. The constants are used for return codes and function control parameters. The handle and function control parameter definitions are specified within the function descriptions that use them. The function declaration attributes are provided below. These attributes are specified in the function declaration between the data type and the function name. Note that these declarations are used for platforms that have special architectural features that either require or provide performance benefits for their use. Hence, we have encapsulated these platform-specific declarations into the platform-independent definitions. On many platforms, these are actually empty.

Table 2 - RM Function Attributes

Attribute	Description
REXTERNAL	Specifies a publicly accessible function in a shared library (DLL).
RINTERNAL	Specifies a private function only called from within a single shared library.
REXTVARARG	Specifies a variable argument list function.
RTHREAD	Specifies a thread function that will be invoked by <b>rm_threadBegin</b> .

The status codes that can be returned by the RM functions are listed in the following table. They are declared as type short. Note that some of the RM functions return -1 to indicate a failure and 0 (RM\_OKAY) for success.

Table 3 - RM Return Codes

Return Code	Description
RM_OKAY	Function was successful.
RM_TIMEOUT	Operation could not complete before specified timeout occurred.
RM_QUEUE_EMPTY	There are no messages on queue.
RM_NOCATPATH	The catalog path was invalid. Must specify fully-qualified name of directory that contains the RDM Server catalog. Returned from <b>rm_startup</b> .
RM_INVCATALOG	The RDM Server catalog is invalid. Either the file name was not fully-qualified or the file was not found in the specified directory. Returned from <b>rm_startup</b> .

Some general purpose functions are provided in the RM API. The **rm\_startup** and **rm\_cleanup** functions are provided as alternatives to **s\_startup** and **s\_terminate**. If you plan to use *only* the RM API functions and not any of the RDM Server database functions (i.e., **s\_**, **d\_**, or SQL functions), then calling **rm\_startup** in place of **s\_startup** will be faster and utilize less memory. This is because function **s\_startup** launches the RDM Server database management threads. The **rm\_startup** function will only initialize the RDM Server Resource Manager subsystem.

You can call function **rm\_getenv** to retrieve standard operating system environment strings. Function **rm\_interrupt** provides access to the user control break-interrupt handling. Function **rm\_sleep** can be called to allow a thread to suspend for a specified number of milliseconds.

---

## Thread Functions

A thread is a function that can be executed independently from other threads of the same process. A thread shares the same address space, global variables, and resources of its parent process. Each process begins with a single thread. Once a thread function is invoked by the process' main thread, it is executed in parallel with the main thread. Theoretically, any number of threads can be launched. However, some operating system platforms may limit the number of threads. Moreover, having too many threads can actually have a negative impact on system performance due to the overhead incurred by the operating system in managing and scheduling the threads. The cost of this overhead, however, will vary between systems based on the architecture of the computer and the particular operating system. Threads provide greater performance gains on symmetric multiprocessor (SMP) systems that consist of two or more CPUs.

In the RDM Server RM API, you call function **rm\_threadBegin** to initiate thread execution. The thread consists of a standard C function of type void that has been declared with the RTHREAD attribute. You can pass a single pointer parameter to the thread function when **rm\_threadBegin** is called. The thread will continue executing until either it simply exits or it calls function **rm\_threadEnd**.

Each thread has its own local storage allocated from the process stack. The amount of stack space to be assigned to a thread is specified by a **rm\_threadBegin** parameter. A system default value can be specified using the constant `THREAD_STACK`.

Threads can execute at one of three priority levels. RDM Server session threads (i.e., threads that will perform an **s\_login**) should, in general, only be assigned priority `RM_THREAD_HIGH` to threads that need to perform at an equal or higher priority than the RDM Server system threads. Excessive use of `RM_THREAD_HIGH` could negatively impact RDM Server system throughput.

---

## Synchronization Functions

Multiple threads cannot safely manipulate global variables. At least one of two threads that attempt to simultaneously update the same global variable will lose its update. At worst, some systems may even crash. To ensure that updates to global variables by multiple threads can safely occur, some kind of *synchronization* must be performed. Mutual exclusion synchronization causes multiple thread updates to be *serialized* meaning that the updates occur one-at-a-time in the order in which they are requested. The RDM Server RM functions **rm\_syncEnterExcl** and **rm\_syncExitExcl** can be used to provide mutual exclusion through a *mutex semaphore*. Any number of mutex semaphores can be created to protect different global data. Note that read-only access to a global variable is safe and does not need to be serialized.

Two threads that update the same shared data can sometimes be synchronized through use of *event semaphores*. With an event semaphore, one thread accesses and updates the shared data and then signals the other thread (which has been waiting) when it is complete. Functions **rm\_syncStart**, **rm\_syncResume** and **rm\_syncWait** are used to provide event semaphore synchronization.

A semaphore called a *quiescent semaphore* is used to provide a special purpose synchronization between multiple readers and a single writer to shared data. It is used in situations where the readers must be prevented from accessing the shared data while it is being updated but the writer

cannot perform the update until all readers have finished. For example, quiescent semaphores are used in RDM Server to control the checkpoint process. (Functions that modify the database cache cannot begin while a checkpoint is occurring, and a checkpoint cannot begin until all currently executing cache modification functions have completed.) Quiescent semaphores are available through use of functions **rm\_syncEnableQ**, **rm\_syncDisableQ**, **rm\_syncSendQ**, **rm\_syncReceiveQ**, and **rm\_syncWaitQ**.

You can create a semaphore of any of these three types by calling function **rm\_syncCreate**. Semaphores are freed by calling **rm\_syncDelete**.

---

## Queuing Functions

It is often necessary to send data to a thread. The message queuing functions of the RM API provide this capability. Any number of threads can send messages to any number of other threads. Each queue has a separate name and is accessed through a queue handle associated with the queue. Queue handles are assigned by function **rm\_queueCreate**.

You can place a message on a queue by calling function **rm\_queueWrite**. The message consists of a pointer, size of block pointed to by the pointer, and a (long) message identifier. A simple priority mechanism allows you to specify that a message is to be placed at the head of the queue. Normally queues are processed in a first-in, first-out method. A thread can read the next available message from a queue by calling function **rm\_queueRead**. You can specify how long the thread will wait for a message to be sent when the queue is empty. If the specified wait time expires, the function will return status **RM\_TIMEOUT**. If you specify a timeout value of **RM\_INDEFINITE\_WAIT**, the thread will wait until the next message has been placed on the queue.

---

## Dynamic Memory Functions

The RDM Server Resource Manager includes a robust set of tagged memory allocation functions that allow you to allocate memory associated with *tags*. To create a memory tag, call function **rm\_createTag**. In your call to **rm\_createTag** you can provide a *setjmp* jump buffer. Control is passed to this buffer in the event of insufficient memory available for a memory allocation call for that tag. Typically, you call the standard C function, **setjmp** from your top level function (you want control to pass *up* the function call stack). Use of the jump buffer means that you do not have to inspect each call to **rm\_getMemory**, **rm\_cGetMemory**, **rm\_extendMemory**, or **rm\_growMemory** for NULL. All memory associated with a tag can be freed in one efficient function call, **rm\_freeTagMemory**.

You can also specify the address of a function that will be called by the RM memory allocation functions when there is insufficient memory for a particular allocation. This function, written by you, would be responsible for freeing enough memory to satisfy the request. You can use this call back function to implement a "garbage collection" routine.

In addition, a memory pool interface exists that allows you to easily reuse buffers to minimize the overhead associated with many dynamic memory allocations and frees.

---

## File Handling Functions

Some operating systems impose a limit on the maximum number of files that a process can have open at a time. The RM file handling interface keeps track of this limit and will dynamically open and close operating system files so that this limit is not reached.

The RM file interface also includes the ability to perform asynchronous I/O, allowing multiple threads to simultaneously read or write the same file. In addition, asynchronous writing to a file allows a write call to return before the write has completed.

---

## Example Program

Uses of many of the RDM Server Resource Manager functions are illustrated in the following "fastdump" example program. The example program is a multithreaded hexadecimal dump utility that only uses the Resource Manager API functions, i.e., no RDM Server database calls are performed. The program launches up to 9 threads. The main process thread uses a queue to send requests to dump a block from the input file into the formatted output file. Each dump thread reads the next dump request message from the queue, asynchronously reads the specified block from the input file, formats the output block, and asynchronously writes the formatted block to the output file. When the dump message for the final block has been sent, the main process thread sends one terminate message for each dump thread. When the dump thread reads the message, it terminates and signals an event semaphore to inform the main thread that it has completed. When all threads have signaled their completion, the program terminates.

The fastdump program can be executed using the following command line:

```
fastdump [-b blocksize] [-t nothreads] inputfile  
outputfile
```

-b *blocksize* Specifies the size of the block from the input file that will be dumped by each thread. If size of the block is not a multiple of 512, it will be truncated to a multiple of 512. Default value is 4096.

-t *nothreads* Specifies the number of dump threads to be used. Maximum is 9 and default is 3.

The initial code of fastdump's main program that processes the command-line arguments is shown below.

```

/* =====
Multithreaded hex dump program
*/
void main( int argc, char *argv[] )
{
    int argi;
    char *inFileName, *outFileName;
    short *threadSems, tno, stat;
    unsigned long filelen, pos;
    jmp_buf noMoreMem;
    DUMPMSG *pMsg;
    /* process command line options */
    for (argi = 1; argi < argc && argv[argi][0] == '-'; ++argi) {
        char opt = argv[argi][1];
        switch ( opt ) {
            case 'b': /* set size of input file block buffer */
                if ( ++argi == argc ) usage();
                buffSize = (size_t)atoi(argv[argi]);
                if (buffSize % 512) {
                    /* make sure its a multiple of 512 */
                    buffSize = (buffSize/512)*512;
                }
                break;
            case 't': /* set # of dump threads (max. of 9) */
                if ( ++argi == argc ) usage();
                maxThreads = (short)atoi(argv[argi]);
                if (maxThreads >= 10)
                    maxThreads = 9;
                break;
            default:
                usage();
        }
    }
    if ( argi < argc - 1 ) {
        /* fetch file names */
        inFileName = argv[argi++];
        outFileName = argv[argi];
    }
    else
        usage();
}

```

Function *usage* simply prints the program usage information and exits the program. The *buffSize* and *maxThreads* variables are global integer variables (of type *size\_t* and *short*, respectively). The next portion of the main program starts the Resource Manager.

```

/* startup RDM Server resource manager */
stat = rm_startup(NULL, MessageConsole, LOG_ERROR|LOG_WARN|LOG_INFO);
if ( stat != RM_OKAY ) {
    printf("Unable to start up resource manager. Status = %d\n", stat);
    exit(1);
}

```

The first argument of function **rm\_startup** is the path name to the RDM Server catalog directory. In this case, a NULL is passed, which causes the catalog path to be extracted from the CATPATH environment variable. The catalog directory contains the **velocis.ini** file from which the Resource Manager extracts its configuration information. If the CATPATH is not present, the function returns error code RM\_NOCATPATH.

The second argument is the address of a user-defined RDM Server message log function. When provided, RDM Server will call this function to process all log messages generated by RDM Server or any calls to function **rm\_log** that are issued by the application.

The third argument consists of a bit map of log message types. Here only errors, warnings, and information type messages are to be logged. Other message types include, for example, the RDM Server startup messages (LOG\_STARTUP) and user login messages (LOG\_LOGIN).

Function **MessageConsole** is shown below.

```

/* =====
   Log console messages
*/
void REXTERNAL MessageConsole(
    RDSLOG_CTRL fcn,    /* call type: open, close, message */
    short       type,   /* message type */
    char        *msg)   /* message to be logged */
{
    if ( fcn == RDSLOG_MESSAGE ) {
        switch ( type ) {
            case LOG_ERROR: printf("***ERROR: "); break;
            case LOG_WARN:  printf("***WARINING: "); break;
        }
        printf("%s\n", msg);
    }
}

```

There is nothing very sophisticated about this function. It simply prints error and warning messages to the server console display. Of course, you can write more sophisticated message handlers. For example, on Windows NT/2000 you could write the message to the event log.

The next section of the main program that opens the input and output files follows.

```

/* open files */
hInFile = rm_fileOpen(inFileName, O_RDONLY|O_BINARY, RM_SHARE, 0, 0);
hOutFile = rm_fileOpen(outFileName, O_CREAT|O_RDWR|O_BINARY,
    RM_SHARE|RM_DIRECTIO, 0, 0);
if ( ! hInFile || !hOutFile ) {
    printf("Unable to open files\n");
    rm_cleanup();
    exit(1);
}

```

The file handles, **hInFile** and **hOutFile** are global variables of type **RM\_PFILE**. These will be shared by the dump threads. Both files are opened as shared, binary files (not text). The input file is read only. The output file is opened as read/write (although write only would work also). The output file is created if it does not already exist (O\_CREAT). In addition, the output file includes the **RM\_DIRECTIO** flag to indicate the writes to be written directly to the disk at the time the write is performed.

Function **rm\_fileOpen** returns a NULL if the files cannot be opened. The reason for the failure will be indicated through a message that has been logged by the **rm\_fileOpen** function prior to returning.

The next section of the main program sets up the program's dynamic memory handling.

```

/* set up memory management */
if ( setjmp(noMoreMem) ) {
    rm_log(LOG_ERROR, "FATAL EXIT: out of memory");
    goto shutdown;
}
/* use C malloc function to allocate a reserve threshold amount */
threshold = malloc(2*buffSize);

/* allocate memory tag, RM_USESEM => multithread allocations are serialized */
mTag = rm_createTag("fastdump", 0, noMoreMem, FreeReserve, 0, RM_USESEM);

/* allocate pool for queue messages */
qPool = rm_allocPool(mTag, "qPool");

```

Notice that the local variable declarations shown earlier for the main program include **noMoreMem** being declared as type `jmp_buf` (declaration comes from the `#include <jmpbuf.h>` that comes in through `#include rds.h`). The call to the standard C function **setjmp** stores the return location information in **noMoreMem**. This is passed to function **rm\_createTag**. Thus if any memory allocation call associated with **mTag** fails due to insufficient memory, a **longjmp** to the specified `jmp_buf` is performed. This results in a non-zero value being returned by **setjmp** invoking the call to **rm\_log** to report the out of memory condition followed by the program shutdown.

The *threshold* is a global void pointer used to provide a reserved chunk of memory (in this case, equal to two blocks) that will be freed by function **FreeReserve** when sufficient memory is unavailable for an allocation request on tag **mTag**. Note that once this buffer is freed, any subsequent calls to **FreeReserve** will be unable to make more memory available, and control will be passed ultimately to the **setjmp**.

The call to **rm\_createTag** includes the `RM_USESEM` flag. It is required because each one of the multiple threads will make memory allocation calls with **mTag** and this forces those allocation calls to be serialized. If `RM_USESEM` is not specified, the memory allocation calls from multiple threads can (and probably will) result in damage to the memory allocation data structures. The result might be a general protection fault or even an infinite loop. In general, you need to carefully identify which threads will perform the allocations from each memory tag. If you are certain that all allocations for a given tag are performed from the same thread, then this flag can be specified as `RM_NOSEM`. The result will be faster allocations. If

there is any chance that allocations for a particular tag can come from more than one thread, then the flag must be specified as `RM_USESEM` (indicating that a mutex semaphore is to be used to protect access to the memory allocation data structures for the tag).

Most of the memory allocations that occur involve allocating and freeing the dump messages sent from the main thread to the dump threads. To avoid the overhead associated with many allocations and frees of identically sized blocks, `fastdump` uses a memory pool. New queue messages will be allocated from the pool and when the dump threads finish with them, the messages will be returned to the pool. The call to `rm_allocPool` allocates the pool handle, `qPool`. The first argument is the memory tag where the memory allocations will be performed by `qPool`. The second argument is the name of a semaphore, created by `rm_allocPool` and used to serialize the pool calls. The need for this is exactly the same as that described previously for all multithread memory allocations. It is needed here because the pool functions will be called from multiple threads.

The code for function `FreeReserve` is shown below. Note that the arguments include the memory tag and the requested amount of memory.

```

/* =====
   Insufficient memory reserve function
*/
void REXTERNAL FreeReserve(
    RM_MEMTAG    tag,
    size_t       size)
{
    if ( threshold && size <= 2*buffSize ) {
        /* free reserve if it has sufficient space */
        free(threshold);
        threshold = NULL;

        /* print notification message */
        rm_log(LOG_WARN, "running low on memory");
    }
}

```

After setting up the dynamic memory handling, the main program creates the queue, creates a global mutex semaphore, allocates an array for each thread's completion event semaphore, and then launches each dump thread. This is all illustrated in the following code segment.

```

/* Create message queue for sending dump instructions to threads */
dumpQueue = rm_queueCreate("dumpQueue");
/* A global variable, blockCount, will be updated by the threads.
   This mutex semaphore will be used to synchronize access to it.
*/
countSem = rm_syncCreate("countSem", RM_MUTEX_SEM, RM_EXCLUSIVE);
/* allocate array of event semaphores, 1 per thread */
threadSems = rm_getMemory(maxThreads*sizeof(short), mTag);
rm_log(LOG_INFO, "Launching %d dump threads", maxThreads);
for ( tno = 0; tno < maxThreads; ++tno ) {
    threadSems[tno] = rm_syncCreate("doneSem", RM_EVENT_SEM, RM_EXCLUSIVE);
    rm_syncStart(threadSems[tno]);
    rm_threadBegin(DumpThread, THREAD_STACK, &threadSems[tno], RM_THREAD_HIGH);
}

```

A single queue, *dumpQueue*, is used to send the dump messages from the main thread to the dump threads. The *countSem* mutex semaphore is used to serialize update access to the global long variable named *blockCount*, which is incremented by each dump thread upon completing each dump request. This illustrates how mutex semaphores are used to provide safe access to global data.

The **threadSems** array contains an event semaphore per thread. These are created within the **for** loop that also launches the threads. The call to **rm\_syncStart** places the event semaphore in an unsignaled state, which causes any subsequent calls to **rm\_syncWait** for that semaphore to suspend until some other thread calls **rm\_syncResume**. The call to **rm\_threadBegin** initiates a separate execution of the specified thread function. The amount of stack space to allocate to the function is specified by the second argument. The **THREAD\_STACK** constant is the RDM Server default stack size. The stack must be of sufficient size to accommodate the function arguments and local variables for the thread function, as well as for all the functions that will be called from within it. Thus, if you are not certain what the size should be, it is best to use the RDM Server **THREAD\_STACK** value (its size is based on the maximum depth of the RDM Server calls). The third argument to **rm\_threadBegin** is a pointer variable that is passed in as the thread function argument. It can be anything you like. In this case, we are passing in the address of the thread's completion event semaphore. The final argument to **rm\_threadBegin** is the thread priority. In this example, it is set to **RM\_THREAD\_HIGH**, since the program only uses the Resource Manager functionality. (The other options are **RM\_THREAD\_LOW** and **RM\_THREAD\_NORMAL**.) For RDM Server applications that will perform logins and make database calls this setting should be either normal or low priority.

After launching the threads the main program starts sending dump request messages to the threads through *dumpQueue*. The following code shows how the dump is partitioned along with the creation and setting of the dump messages, and the call to **rm\_queueWrite** to place them on the queue.

```

/* get length of file */
filelen = rm_fileLength(hInFile);
/* send dump messages to threads */
for (pos = 0; filelen > 0; pos += buffSize) {
    pMsg = rm_getFromPool(sizeof(DUMPMSG), qPool);
    pMsg->startpos = pos;
    pMsg->blocklen = filelen >= buffSize ? buffSize : filelen;
    rm_queueWrite(dumpQueue, pMsg, sizeof(DUMPMSG), 0, 0);
    filelen -= buffSize;
}

```

The call to **rm\_fileLength** returns the size of the input file in bytes and stores it in the local unsigned long variable *filelen*. This is then used to control the **for** loop, which partitions the input file into dump messages each of length *buffSize* (except perhaps the last message). The DUMPMSG type declaration is as follows.

```

typedef struct dumpmsg {
    unsigned long startpos; /* byte offset from start of file */
    size_t blocklen; /* length of block (= buffSize, 0 = end) */
} DUMPMSG;

```

Each message buffer is allocated from the pool. The message is then placed on the *dumpQueue* by the call to **rm\_queueWrite**. The first three arguments are self-explanatory. The fourth argument is a long integer value that can be used as a message identifier. In this program it is used as a Boolean flag that will be true (or 1) when the last message is sent. The final argument is a Boolean that indicates the message is a priority message and, thus should be placed at the head of the queue. In this example, all the messages have normal priority and are placed on the queue in the usual first-in, first-out manner.

At this point, we leave the main program and look at the dump thread function shown below.

```

/* =====
Dump thread - dumps 1 block at a time
*/
static void RTHREAD DumpThread(void *pDoneSem)
{
    short        doneSem;
    char         *inBuff;
    char         *outBuff;
    unsigned int len;
    DUMPMMSG     *pMsg = NULL;
    long         end_of_dump = 0;
    /* end-of-dump event semaphore is thread argument */
    doneSem = *(short *)pDoneSem;
    /* allocate input buffer for 1 dump block */
    inBuff = rm_getMemory(buffSize, mTag);
    /* allocate output buffer for each formatted dump line */
    outBuff = rm_getMemory(77*(buffSize/16)+1, mTag);
    while ( ! end_of_dump ) {
        rm_queueRead( dumpQueue, &pMsg, NULL, &end_of_dump, RM_INDEFINITE_WAIT );
        if ( pMsg ) {
            /* read block to be dumped */
            len = rm_fileSeekRead(hInFile, pMsg->startpos, inBuff, pMsg->blocklen);
            if ( len != pMsg->blocklen )
                rm_log(LOG_ERROR, "error reading input file");
            else
                DumpBlock(inBuff, pMsg->startpos, pMsg->blocklen, outBuff);
            /* update count of dumped blocks */
            rm_syncEnterExcl(countSem);
            ++blockCount;
            rm_syncExitExcl(countSem);
            /* free queue message */
            rm_putInPool(pMsg, qPool);
        }
    }
    /* signal foreground that we're done */
    rm_syncResume(doneSem);
    /* terminate thread */
    rm_threadEnd();
}

```

As was mentioned earlier, the argument to the thread function is a pointer to the completion event semaphore. This is retrieved into *doneSem*. Each thread will have its own input buffer, *inBuff*. The call to **rm\_getMemory** allocates *buffSize* bytes to this buffer. The formatted output for each input block will be written into a single output buffer, *outBuff*, which will be written to the output file using a single write call. Each formatted output line consists of the dump of 16 bytes from the input block. These lines are exactly 77 characters long.

The main body of the thread executes inside the **while** loop. The loop terminates when the *end\_of\_dump* flag, returned by the **rm\_queueRead** call, becomes true. The arguments to the **rm\_queueRead** function are quite simple. The last argument specifies how long (in milliseconds) the thread is to wait for a message to be written to the queue. In this case, the

thread needs to wait indefinitely. If the timeout argument is 0, then the function will immediately return either `RM_OKAY` and deliver the message, or it will return `RM_QUEUE_EMPTY` indicating that there is no message on the queue.

A pointer to the message is returned in *pMsg*. A NULL value for *pMsg* is sent with the last message (i.e., the one where *end\_of\_dump* is 1). The input block specified by the message is read into *inBuff* by the call to **rm\_fileSeekRead**. This function returns the actual number of bytes that were read. If this value (*len*) is not equal to the requested number of bytes, then an error occurred (or, possibly, you tried to read past the end-of-file, which, in this case, cannot happen). If the read was successful, function *DumpBlock* is called to format the input block into the output buffer.

The *blockCount* is a global long variable that keeps a count of the number of blocks that have been dumped by all threads. Since it is being updated by more than one thread, the update needs to be serialized. The call to function **rm\_syncEnterExcl** will suspend until this thread is granted control of the semaphore. Once control is granted the function returns and the thread can now safely update *blockCount*. All other threads that call **rm\_syncEnterExcl** on *countSem* will be queued. When the controlling thread releases the semaphore through the call to **rm\_syncExitExcl**, control will pass to the first thread on the semaphore's queue.

The thread next calls **rm\_putInPool** to return the message buffer to the pool.

When the message to terminate is received from the foreground (main) thread, the loop is broken and the thread calls **rm\_syncResume** to place its completion event semaphore into a signaled state. In general, an event semaphore remains in its signaled state until it is restarted by a call to **rm\_syncStart**. Note that this means that as long as it is signaled, all calls to **rm\_syncWait** on that semaphore will return immediately.

Function **rm\_threadEnd** is called to terminate the thread. Note that the thread will automatically end when the function returns. However, some future RDM Server platforms may require a call to a specific function, hence, it is best to include the call.

Now, returning to the main program, the code segment below shows the main program sending a termination message for each thread. It does not matter which thread processes which message because a given thread will not process any additional messages once it has read the completion message.

```

/* send terminate thread messages */
for (tno = 0; tno < maxThreads; ++tno)
    rm_queueWrite(dumpQueue, NULL, 0, 1, 0);

/* wait for each thread to signal they're done */
for (tno = 0; tno < maxThreads; ++tno)
    rm_syncWait(threadSems[tno], RM_INDEFINITE_WAIT);

rm_log(LOG_INFO, "Dump complete: %ld %d-byte blocks written to file %s.",
    blockCount, buffSize, outFileName);

```

After sending the termination messages, the program loops through each of the thread completion semaphore array entries and calls **rm\_syncWait** to wait for all threads to complete. Finally, function **rm\_log** is called to log the completion of the dump and to report the total number of blocks that were written. Function **rm\_log** functions just like **printf** except that its first argument is the message type. The second is a **printf** format specifier followed by zero or more variables referenced in the format specifier. This message will be processed by the **MessageConsole** function that was specified in the call to **rm\_startup**.

The last part of the program closes the files and frees all of the Resource Manager resources that were used.

```

/* close files */
rm_fileClose(hInFile);
rm_fileClose(hOutFile);
/* delete the queue */
rm_queueDelete(dumpQueue);
/* free the message pool */
rm_zFreePool(&qPool);
/* free semaphores */
rm_syncDelete(countSem);
for (tno = 0; tno < maxThreads; ++tno)
    rm_syncDelete(threadSems[tno]);
/* free allocated memory */
rm_freeTagMemory(mTag, 1);
if ( threshold )
    free(threshold);

rm_cleanup();

```

The second argument in the call to **rm\_freeTagMemory** is a flag that indicates if the tag itself is to be freed. Often, the tag will be reused and will not need to be freed. Reuse of a tag occurs when all of the memory is allocated and freed during a single execution of a function. In these situations, since the tag already exists, you need some way of specifying a new out-of-memory jump buffer. This can be done through a call to function **rm\_resetTag** .

And that completes our discussion of a multithreaded hexadecimal dump utility program. This example, while recognizably trivial, does illustrate use of most of the Resource Manager functions necessary for your more sophisticated application database server programs or extension modules.