PHILIPS

PHILIPS INTERACTIVE MEDIA

Technical Note #87

Memory Allocation in CD-RTOS

Charles Golvin

October 4, 1993

This technical note describes how memory is allocated in CD-RTOS. It also provides guidelines for how to best control memory allocations and, therefore, prevent fragmentation.

Table of Contents

Memory Allocation in CD-RTOS

Overview

Memory remains a precious resource in CD-i applications. It is critical for applications to control memory allocation as tightly as possible, minimizing fragmentation so that as much memory can be used as possible. This technical note attempts to give a complete overview of memory allocation in CD-RTOS. This ranges from the list of functions that allocate memory to the algorithm used by the operating system to satisfy memory requests, to recommended strategies to avoid memory fragmentation. These guidelines are all targeted to the base-case system; some remarks are made in preparation for the introduction of Digital Video and the additional memory that is provided in the Philips implementation thereof.

Resources

The base-case CD-i system provides one megabyte of random access memory (RAM). CD-i players may provide additional memory beyond this requirement, but no less. This one megabyte is divided equally into two 512 Kilobyte banks, each devoted to a video controller. This memory is commonly referred to as "colored" memory, and the two banks are commonly referred to as "plane A" and "plane B" or "path 0" and "path 1." The system also provides a minimum of 8 Kilobytes of non-volatile RAM, which is not treated in this document.

The "CD-I Full Functional Specification" (Green Book) provides that some portion of the memory in the player will be consumed by operating system data structures. This consumption is limited to a total of 64 Kilobytes and may not exceed 32 Kilobytes in either of the memory planes. Thus, prior to a CD-i application being started by the player shell, there must exist at least 480 Kilobytes of **contiguous** memory in each plane. The dynamic allocation of memory by the base-case CD-i file managers is well documented in Section VIII.7.5 of the Green Book. While some of the information in this section is reiterated in this note, the author heartily recommends that you read this portion of the Green Book in detail.

Memory Allocation Functions

There are two system-level functions for obtaining memory from CD-RTOS: **F$SRqMem** (request memory) and **F$SRqCMem** (request colored memory)—these functions are bound by the C functions **_srqmem()** and **srqcmem()**, respectively. Both of these functions take as a parameter the size of the requested memory. In addition, **F$SRqCMem** takes a parameter instructing it from which plane the memory is to be supplied. If the requested plane does not contain a contiguous block of the desired size, the request will fail, indicating that memory is not available. It is possible to supply the value 0 for the desired color, indicating that it does not matter from which plane the memory is to come. That is, **F$SRqCMem** with a color of 0 is completely identical in behavior to **F$SRqMem**.

The operating system has a minimum process memory allocation size stored in one of the system global variables (**D_MinBlk**). Not only is this the smallest block of memory the system makes available to a process, it is also the basic unit of memory allocation. All process memory requests are rounded up to the nearest multiple of this number. Similar values exist for system (versus process) memory requests. In the CD-i system, this size is sixteen bytes for both process and system.

The standard C memory allocation function **malloc()** and its derivatives are, eventually, bindings to **F$SRqMem**. These functions are not very efficient for managing memory in a constrained system like CD-i. They work by allocating a large block (4K in the current Microware C library implementation) at the first request, doling out pieces of this block to subsequent requests until a request cannot be satisfied, in which case one or more additional large blocks must be allocated. The advantage of this approach is that one is somewhat insulated from having to manage every byte of memory. One disadvantage is that these functions must consume a portion of this memory with their own internal tracking information (eight bytes per allocated :hunk), replicating a function that the operating system is already providing. A more obvious disadvantage is that the functions are wasting memory that your application (or the operating system) could be using, namely the portion of each 4 Kb block that is not currently used..

In addition to **F$SRqMem** and **F$SRqCMem**, the following OS-9 system calls can or will cause memory to be allocated:

F$Load	Load module
F$DatMod	Create data module
F$IRQ	Add or remove device from IRQ table
I$Open	Open a device or file
F$Send	Send a signal
F$Fork and **F$Chain**	Execute new process
F$Mem	Resize primary data area
F$SrtMem	Return allocated memory
F$Alarm	Create timer
:V$Create	Create event

This should not be taken as an exhaustive list.

Memory Allocation Algorithm

In addition to the system free memory pool, a fragment list is maintained for each process, one list per plane of colored memory (per process). The system attempts to first fulfill requests from the process's fragment list for the plane of memory being requested (if present), then from the system free pool. When satisfying requests from the process fragment list, the list is searched until a fit is found. When satisfying requests from the system free pool, memory is searched from high memory to low memory, and the first block the system encounters of sufficient size is used to satisfy the memory request.

Different types of memory are assigned different priority values. In the base case, plane A and plane B memory have identical priorities. In the case of specific colored memory requests, memory priority is not considered, because the request will either be satisfied from the requested plane or it will fail. However, when memory of type "don't care" is requested, the system searches the various memory types in descending order of priority. If separate banks of memory possess the same priority, then the plane that is less used at that moment is searched first, from high to low addresses, and the first block of sufficient size is used. If this still does not succeed, then the next least used plane is searched, high to low, and so forth. In this case, the measurement of which plane is less used is simply the total amount of free memory available in that plane of memory.

To summarize, "don't care" requests are first satisfied from the process fragment lists, then in order of memory priority, then based on least used plane from high memory to low memory. This search terminates when the list of memory "planes" has been exhausted, or when the only planes not yet searched have priority zero. Priority zero memory may be allocated only via a specific **F$SRqCMem** request. (The MPEG decoder RAM is an example of priority zero memory. This is because this memory must all be available for the decoder when it is active, so the operating system must be prevented from inadvertently allocating from it.) However, there is one exception to this high to low memory address rule. In its header, each OS-9 program module specifies the amount of local data and stack area it requires. When a new process is started (via **F$Fork** or **F$Chain**), the operating system allocates the required amount of memory on behalf of the incipient process via the **F$Mem** system call. These "don't care" memory requests are always satisfied from low memory. (I believe the reason for this comes from less static implementations than CD-i, where processes are born more frequently and die more frequently. This algorithm attempts to concentrate the free memory pool between the executing applications and their local data spaces. This is speculation on my part.)

There is a simple method for determining the size of the memory that will be allocated on behalf of your process for its primary data area. The program **ident** prints out the module header information for any OS-9 module. A program module contains fields that describe the data and stack requirements for the module. In viewing the output of **ident**, look at the fields called "Data size" and "Stack size"—the sum of these two numbers, rounded up to the nearest multiple of 16, is the size of the module's primary data area. (For you bitheads, you can equivalently dump the module's contents and examine the long word values at offsets $38 and $3C respectively.)

Operating System Requests

The number of CD-RTOS calls that allocate memory in a plane-specific manner is small. Essentially, they are the routines that rely on the memory they allocate residing in a specific plane—the functions to create draw maps and to create Display Control entities (FCTs and LCTs) and the function to execute a DCP. Nearly all other system memory requests are of the "don't care" flavor.

Because most of the system memory allocations are of this "don't care" flavor, it is critical that an application be very cognizant of the effect on memory of the system calls that it makes. A slight difference in the initial memory configuration of one player versus another may change the balance in the memory usage and force allocations that you have unknowingly relied upon to fall in a specific plane falling into the other, causing a subsequent plane-specific memory request to fail. Sadly (or luckily, depending on your view), it is not possible as an application developer to test all the possible memory configurations that the player manufacturers may present to your application. Therefore, it is critical that you control as much of the memory allocation by the system as possible, and that you leave nothing to the vagaries of the player's initial memory configuration.

Returning Memory

Irrespective of the system call used to allocate memory for a process, memory can be returned to the system using the **F$SrtMem** system call, bound by the C function **srtmem()**. This function takes as parameters both the pointer to the memory to return and the size of the block being returned. In this way, the correlation between memory requests and memory returns is not one to one—a large block may be returned to the system in pieces.

There are some caveats about returning memory. The first is that, as mentioned above, **F$SrtMem** can actually cause memory to be allocated. This occurs only when the return of this memory creates a process-specific fragment and there does not yet exist a process fragment list descriptor for the plane in which the piece of memory resides. As discussed above, when fulfilling memory requests, the system first checks the list of free fragments of the requesting process. The system data structures that head the list of these fragments in each plane are allocated dynamically when the process's first fragment is created. A fragment is, in this case, a piece of memory that is abutted on either side by a piece of memory allocated to the same process. There can exist a fragment list descriptor for each plane of available memory, each of which consumes $30 (#48) bytes. In order to prevent this from happening at inopportune moments, you should force the system to create these descriptors when your application starts (an xample is provided below).

Now some system bookkeeping information. A process is only allowed to possess up to 32 distinct allocated memory blocks, and this array is maintained in the process's descriptor. If memory is allocated to a process and the allocated memory is adjacent to one or two blocks (one on each side) already allocated to the process, the system simply coalesces these blocks into a single block for the purposes of the process's allocated block list. In this way, the process is able to maximize the use of its 32 blocks. (A memory request that cannot be satisfied because this limit of 32 has been exceeded will return error number 207: memory full.)

The second caveat concerns the primary data area for a process. As mentioned before, this memory allocation is made from low memory, while all other requests come from high memory. Also, while the system actually allocates this memory when a process is starting, the memory allocation is attributed to the application and, hence, counts as one of its 32 allocated blocks. Therefore, it is possible that the operating system could coalesce the process memory allocation with this primary data area. Here's an example of how this could cause a problem.

F$SRqCMem takes -1 as a valid size parameter. In this case, the operating system returns the largest contiguous chunk available in the requested plane, updating a global variable (**_srqcsiz**) with the actual size of the chunk that was returned. Some applications use this feature to obtain the largest chunk possible at start up and then return a portion of the chunk to the operating system for its overhead. (In using such a strategy, this document should be a guideline for determining how much memory must be left to the system. Issues such as number of open files, double buffering of FCTs, etc., must then be considered.) The following diagram illustrates the situation that could occur in this case.

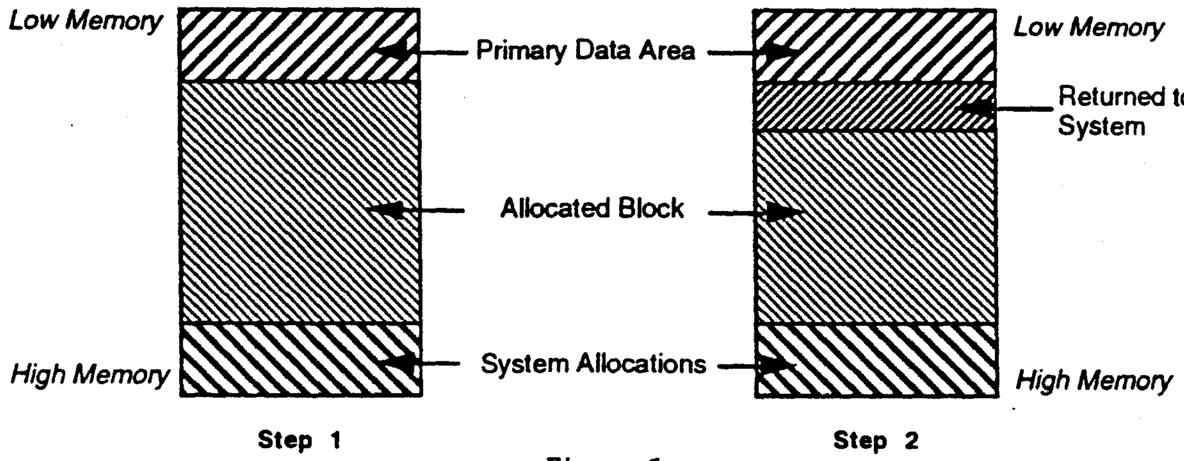

Figure 1.

The problem is that when the block is returned to the system it will not go into the system free pool, but rather into the process fragment list because it is adjacent on both sides to blocks already allocated to this process. Thus, the application's intent has been subverted—the memory intended for return to the system is available to the process only for subsequent requests. The strategy is sound, but should instead be achieved in the following way. Allocate the largest possible block to determine its size (or, equivalently, use the system call **F$GBlkMp** and figure it out); then return it to the system and request this size minus what is intended for the system's use.

*Note: There is a known bug in **srqcmem()**, the C binding to the function **F$SRqCMem**. This bug pertains to the case where size of the requested block is -1, and the result is that the global variable **_srqcsiz** does not hold the size of the chunk, but rather the pointer to the allocated block. This problem can be averted by providing your own binding to **F$SRqCMem** as shown on the following page.*

```
        /* put the following in all files where srqcmem is used */
#define srqcmem _srqcmem

#asm
_srqcmem:
   movem.l    d1/a2,-(sp)
   os9        F$SRqCMem
   bcs.b      oops
   move.l     d0,_srqcsiz(a6)
   move.l     a2,d0
ok
   movem.l    (sp)+,d1/a2
   rts
oops
   moveq.l    #-1,d0
   move.l     d1,errno(a6)
   bra.b      ok
#endasm
```

System Data Structures

There are several system data structures related to memory. The first is the "colored memory definition list" which is contained in the **init** module in the player. This list essentially defines what memory exists in the system, along with attributes of each piece of memory such as size, address, priority, etc. This list is a static structure that is used at start up of the player.[1]

The dynamic memory data structure is the "memlist," a doubly linked list that carries some of the information contained in the colored memory definition list, as well as updated information including the size of available memory per plane, the list of free blocks per plane, etc. This is the data structure that is used by the system call **F$GBlkMp** (get free memory block map). Applications are free to examine this data structure, which can be found at the system global **D_FreeMem**. From C language, the start of the list can be found via the call **_getsys()**. This returns a pointer to the head of the list. Each element in this list has the structure shown on the following page:[2]

[1] See the OS-9 Technical Manual, page 2-12.

[2] Dayan, Paul. The OS-9 Guru, 1992. Galactic Industrial Limited, United Kingdom, pp. 53-54.

Offset	Type	Description
$000	long	Start address of memory area
$004	long	End address plus 1 of memory area
$008	long	Address of next (lower priority) memory color node in list
$00C	long	Address of previous (higher priority) memory color node in list
$010	long	Address of first free memory area within this area
$014	long	Address of last free memory area within this area
$018	long	Reserved
$01C	long	Start address of memory area as seen by alternate bus master (local start address plus translation offset given in memory list in init module)
$020	long	Sum of sizes of free memory areas in this memory area
$024	word	Attributes of this memory area (as given in the memory list in the init module)
$026	word	Color number
$028	word	Priority

In the case of the Philips Digital Video cartridge, it is important to note that both the MPEG 0.5 Mb and SYSRAM 1.0 Mb blocks are dynamically inserted into this list at start up and are not reflected in the init module. Therefore, applications considering Digital Video implications for memory management should use this data structure rather than relying on the contents of the init module.

CD-i Start-Up Trace

Armed with all this information, we can now describe the sequence of memory allocations made by the system as the player shell starts a CD-i application. The player shell process is already active and, by the time the user has selected "Play," the shell will have read the disc label to determine the name of the application and, therefore, the system will have allocated the directory cache ($800 bytes) and the path table buffer (variable size). Both of these requests are of type "don't care." Next, the player shell loads the application module into high memory of plane B and executes **F$Chain** to start the application. At this time, the primary data area for the application is allocated; this is also a "don't care" request. However, in most cases, the loading of the application is enough to tilt the memory balance so that plane A is less used; so, in general, the primary data area will reside in low memory of plane A. (You can find out for yourself where it ended up— simply print the address that is in register a6 and determine in which plane that address resides.) The system will attempt to grow or shrink the primary data area of the player shell to satisfy the needs of the application (that is what **F$Mem** is for); if the operating system is not successful, then a fragment will be created when the primary data area for the player shell is released. Because the application is chained (rather than forked) by the player shell, the system re-uses the player shell's process descriptor area for the disc application.

Now the application's requests begin. Let us postulate the following pseudo code to describe the initialization done by the application:

```
create plane A and B FCTs of 150 instructions each
create plane A and B LCTs of 278 physical lines
create plane A and B LCTs of 2 physical lines
allocate 380 Kb buffer in plane A
allocate 350 Kb buffer in plane B
create two soundmaps of 3 sectors each
open real time file
load first picture
execute DCP (via DC_Exec)
```

The following diagram summarizes the state of memory at the end of all this work, assuming that the application is 78 Kb in size, its local data area is 24 Kb, the path table is 24 bytes, and the initial state of memory has 32 bytes more memory available in plane A than plane B prior to the player shell reading the disc. Take special note in this case of where the "don't care" requests have fallen.

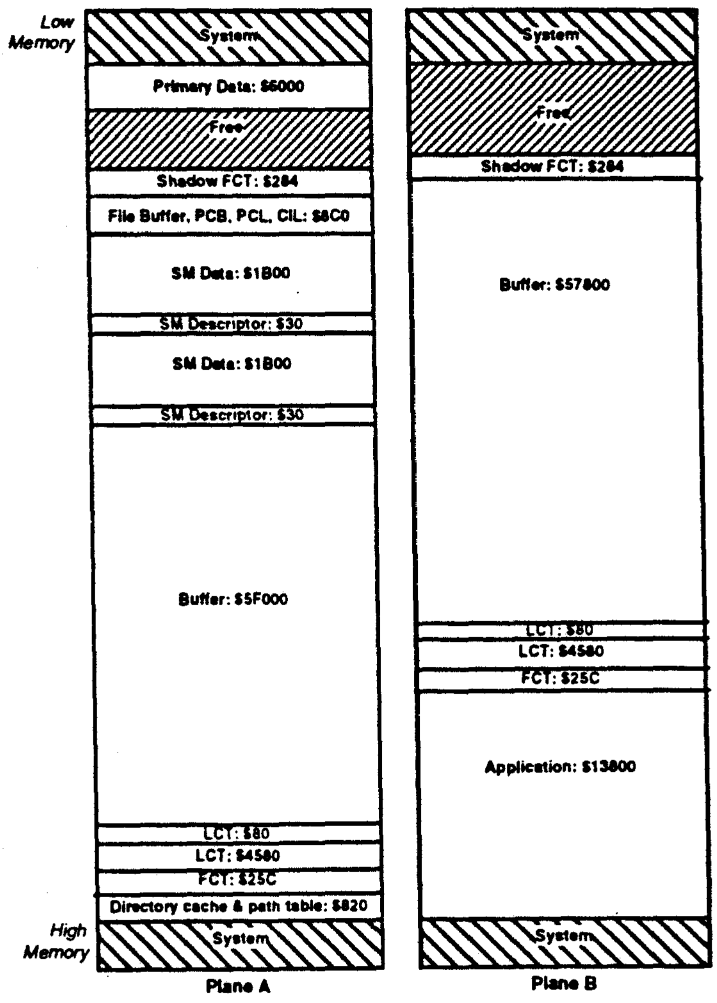

Figure 2.

The following diagram shows the situation that obtains from the same application being loaded on a player with one megabyte of system RAM (for example using the Philips Digital Video cartridge).

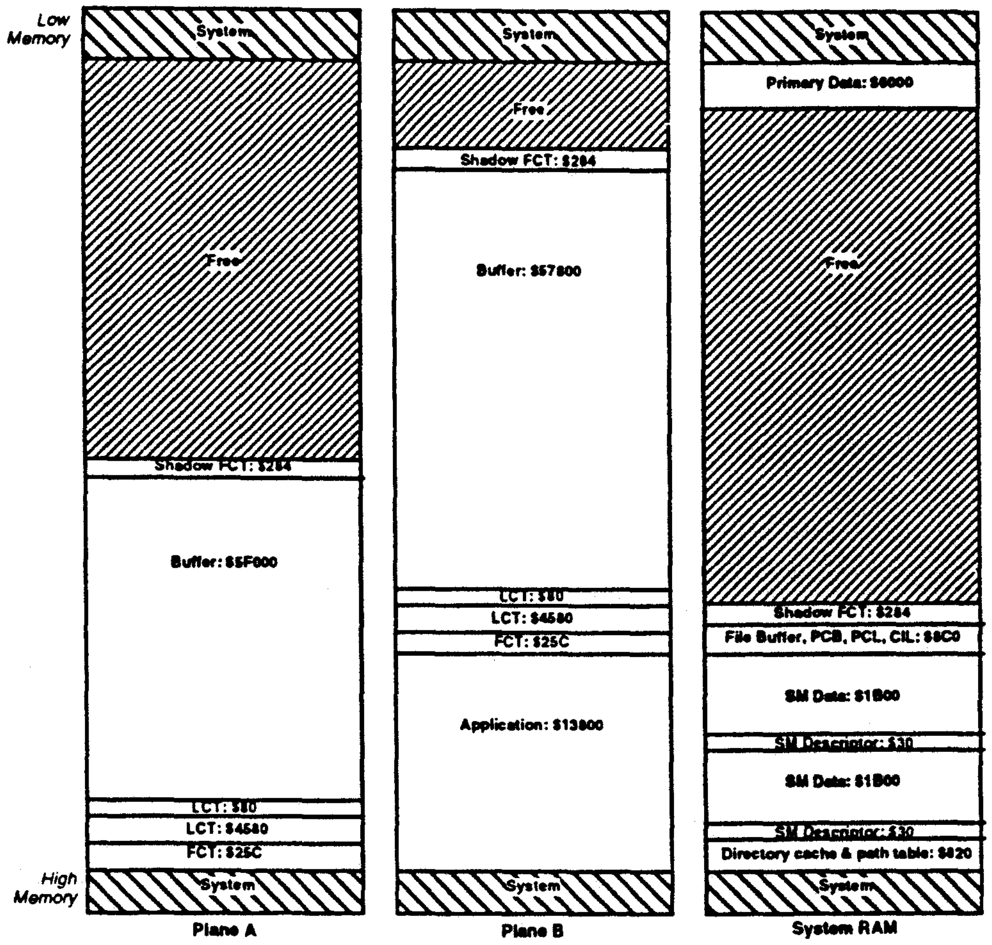

Figure 3.

Gotchas

There are a few subtle points that have been partially covered in this document. In most of these cases, by understanding the condition, it is possible to prevent the system from allocating memory at an inopportune moment. Following are the cases that I can recall.

Fragment descriptors. As described above, when returning memory to the system, it is possible that the system will allocate a fragment descriptor. The best method to prevent further memory fragmentation (caused by allocation of fragment descriptors) is to force the system to allocate these structures in an area that will not cause a problem. For example, as part of its initialization, an application could allocate three blocks of $10 bytes from a plane of memory and then return the second block to the system. This forces the system to create a fragment descriptor for that process; this should be done for each plane of memory. Once the fragment descriptor has been allocated, it is not returned to the system until the process terminates, so it's fine to go right back and reclaim this $10 byte block. A similar strategy, more sensible for applications that simply allocate a large block in each plane and partition the block internally, is to allocate your large buffer, point $10 bytes into it, and pass this pointer to **F$SrtMem** with a size of $10. This will create the fragment descriptor and the small block can then be reallocated, restoring the original large block that was allocated. This reallocation should be done immediately in order to reclaim this particular block, but this block will never be allocated by the system—only the process that created it.

Queued signals. Just as hardware interrupts may be masked, software interrupts (that is, signals) can be masked, both intentionally by the application and indirectly by the operating system. When the operating system attempts to deliver a signal to a process and finds that signals are masked, it is forced to queue the signal until the mask is removed. An internal data structure of $10 bytes is used for each queued signal and, once created, these data structures are not returned to the system free pool until the process terminates (just as with the fragment descriptors described above). Therefore, it is possible that the untimely delivery of a signal could cause memory fragmentation due to the creation of this data structure.

Each process descriptor contains an entry to hold one queued signal descriptor, so provided no more than one signal is ever queued, no memory is allocated by the system for this purpose. To further reduce this probability, it is simple for an application to force one or more of these allocations. To do this, a process need only mask signals, send signals to itself, and then unmask signals. After the process receives these signals, the signal queue descriptors remain for the system's use. Here is an example piece of code to perform this task:

```
int idx, pid = getpid();      /* Get our process id */

sigmask(1);  /* Mask signals */
for  (idx=0;idx  <  number_of_queues_needed;idx++)
   kill(pid,QUEUE_SIGNAL);    /* Send some dummy signals */
sigmask(-1); /* Unmask signals */
```

DC_Exec. As described in the Green Book, Section VIII.7.5.2.2.4, and referenced in the above diagrams, the execution of a new DCP via the function **DC_Exec** causes the operating system to allocate new "shadow" FCTs. Applications with dynamic memory allocation and frequent calls to this function can easily encounter memory fragmentation due to this allocation.

It is possible to control the system's allocation of the shadow FCTs by using the system's memory allocation algorithm. Since these allocations are plane specific, they are fulfilled from the first block available starting at high memory. The idea is to ensure the availability of a free block of the correct size at the time **DC_Exec** is executed. To do this, as one of its first memory allocations (therefore coming from the highest addresses), the application should allocate a block of size equal to the total amount of memory needed to contain the two shadow FCTs in that plane (two since, in order to use **DC_Exec** for this purpose, the FCTs must be double buffered). Prior to executing **DC_Exec**, the application then releases the piece of this memory that is not being used by an FCT at that moment, executes **DC_Exec**, and then reclaims the "other" piece that was just released by the system as the previous FCT was discarded. Of course, this represents some overhead for the application, which must also keep track of which block belongs to which FCT. Also, because memory is being returned, it is critical that the application ensure that the release of memory does not cause the allocation of a fragment descriptor. PIMA has developed library calls that provide these "wrappers" around **DC_Exec**. Contact the Developer Services organization if you are interested in receiving them.

Growth of Tables. CD-RTOS uses several tables to keep track of identifiers and, in general, applies a common strategy to handling the growth of these tables. When a table is full and a request is made that would cause the table to grow, new memory is allocated of sufficient size to hold the new table. These tables can only grow in size—they will never contract.

In the cases of draw map descriptors, region descriptors, and sound map descriptors, the growth is always made by doubling the size of the current table. However, the module directory (the table of modules currently known to the system) always grows by 16 entries ($100 bytes). This growth occurs as the new module is being loaded into the directory—during the execution of **F$Load**—and is a "don't care" request. The number of modules in the module directory at the time an application is loaded depends entirely on the player manufacturer, how many modules are in ROM, plus any data modules that might be created during system initialization.

Many application developers concatenate other needed modules to the application program in order take advantage of the initial loading of the application by the player shell. In this case, there is no way to control the potential growth of the module directory. However, for modules that may be loaded during the remainder of the title, there is a defense. Since the application will know the maximum number of modules that can be loaded or created during the title, during initialization it can create (using **F$DatMod**) that maximum number of modules. The module directory is then guaranteed to not grow further, and the data modules can then be unlinked to return their memory to the system. The minimum size of each of these modules is $40 bytes, assuming the "cdi_" naming conventions for application-specific modules. If this strategy is employed, this potential expansion should be performed **after** the application's allocations are made so that, if the module directory grows, the new table is forced to one of the extremities of memory.

Alarm Threads. CD-RTOS allocates memory for each alarm that a process requests; each alarm thread block is $80 bytes. Unlike signal queue descriptors, the alarm thread blocks are returned to the system when the alarm expires or is deleted. However, the creation of an alarm at the wrong time can cause a memory fragment when the alarm expires. The memory allocated for a cyclic alarm will remain allocated until the process deletes the alarm or the process exits. The best strategy for preventing this possible fragmentation is to delay the creation of any alarms until after the application has allocated its large contiguous blocks. More dynamic memory allocation schemes must consider carefully the state of their allocations when creating alarms.

Open Paths. As mentioned above in the memory trace section, CDFM allocates memory for the directory cache and path table when the disc is first read. This cache will be reused whenever CDFM reads a new directory sector. However, in the case that a new disc is inserted, the system will release the buffer and allocate a new one when a path to the new disc is opened. Of course this is only an issue for titles that allow the use of more than one disc (for example, **Caricature** or any multi-volume title). The reason that the system cannot reuse its buffer is because, while the directory cache is always the same size, the path table can vary in size from disc to disc. Therefore there is no guarantee that all the information about the new disc will fit within the allocated space. However, because of the manner in which this buffer is allocated (don't care), provided the new path table is no greater in size than the path table on the last disc accessed, the system will likely receive the same block when allocating the new cache. Applications ishing to ensure this can modify their disc image to artificially inflate the size of the path table to a size beyond what most discs can be expected to use (e.g. $800 bytes). Contact PIMA Developer Services if you would like to receive a tool to perform this task.

Summary

This document has described many of the internal allocation strategies used by CD-RTOS, as well as some of the more esoteric allocations made by the system. In most of these cases, a method has been proposed to reduce the impact of unexpected system memory requests. Following is a checklist for the items that should be considered when designing a memory strategy for a title.

- The application is loaded into high memory of plane B.
- The primary data area (data plus stack) is allocated from low memory of the least used, highest priority piece of memory—in base-case systems this is generally plane A, and in DV-extended systems this is system RAM.
- Allow $8C0 bytes per path open to CDFM.
- Allow approximately $830 bytes per process.
- If memory is ever returned, force the system to create fragment descriptors (one per plane of memory) at application start up.
- Allow memory for the creation of the shadow FCTs; if FCTs are double-buffered, consider the strategies described for eliminating the potential fragmentation caused by DC_Exec.
- Force the creation of additional signal queue elements.
- Allow for the growth of the module directory and, if appropriate, other id tables.
- Account for the creation of alarm threads.
- Never use **malloc()**.