I have a project with various communication stacks. Messages arrive through one port, and may be processed locally, or routed to another port. Additionally messages may be generated locally and sent to a port. These messages vary significantly in size, with most being less than 64 bytes, but occasionally there will be a packet approaching 1024 Bytes. However, the sizes may be any ware from 1 to 1024 bytes.

I am trying to understand if FreeRTOS supports a fragmentation-safe feature for passing these variable-length packets between threads and between ISR and thread. For example, is it possible to use malloc() in conjunction with Message Queues (with the producer passing a reference to the allocated memory) without fragmentation concern.

I see in the Heap_4.c source code that it, “combines (coalescences) adjacent memory blocks as they are freed, and in so doing limits memory fragmentation.” However, this still concerns me because my product may not be reset for months of run-time. It seems very difficult to test if the coalescence scheme is effective enough.

I was considering the scheme described here …

http://www.barrgroup.com/Embedded-Systems/How-To/Malloc-Free-Dynamic-Memory-Allocation

… which I’ll paste below:

`<Snip>`

Memory pools

We now return to schemes that allow the application to free memory. Pools, or partitions, of fixed-size memory blocks can be used to completely eliminate the potential for fragmentation. They are a compromise between static allocation and a general purpose heap, since this heap can be tuned at design time for the size of the requests that will be made. While the standard implementations of **malloc()** and **free()** have to be general purpose, many embedded systems consist of a single program, and your heap can be tuned so that it works brilliantly for this one program even though it might fail miserably for others.

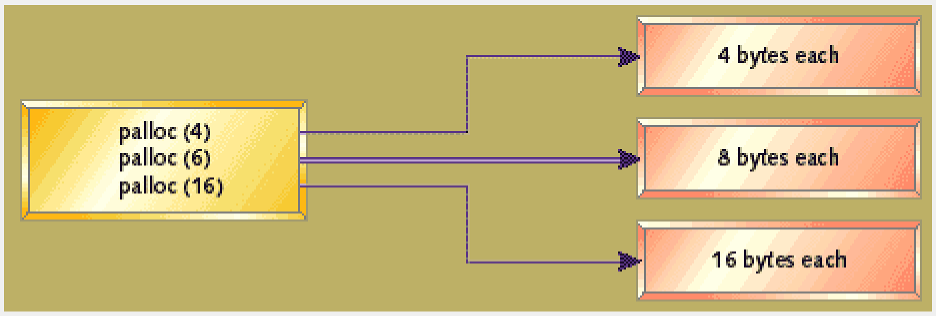

Figure 5. Using the appropriate pool

Each pool contains an array of blocks. Unused blocks can be linked together in a list. The pools themselves are declared as arrays. This mechanism avoids the overhead of a header for each block, since size information is fixed for each pool. Figure 5 shows the way in which requests are directed to the pool which is equal to the request, or the next larger block, if no exact match is available. This system must be tuned by deciding which size blocks to make available and how many blocks to provide in each pool. Defining pools at sizes which are powers of two (that is, 2, 4, 8, 16, 32, 64, etc.) is a good starting point to use if size measurements have not yet been taken for your application.

One of the major motivations for using pools to implement a heap is that a careful implementation can have fixed execution times for allocations and for freeing of blocks. More general heap implementations always involve iterating through lists which can vary in size.

By monitoring the size of each pool, and confirming that the number of blocks in use ceases to grow after extended use, the designer can be confident that leaks have been eliminated. While it is wise to size the pools larger than the worst case seen in test, designers should be aware that allowing too much "padding" leads to wasted memory.

Some implementations of **malloc()** use pools for small requests and a general purpose heap for large requests. [3]

`</Snip>`

I've read the "Memory Management" documentation of FreeRTOS …

http://www.freertos.org/a00111.html

… as well as skimmed the source code, but I am not confident that the heap implementation will meet my requirements.

I will paste "Memory Management" documentation below as well.

`<Snip>`

Memory Management

[More Advanced]

FreeRTOS offers several heap management schemes that range in complexity and features - the simplest of which can sometimes even meet the requirements of applications that, for safety reasons, do not permit dynamic memory allocation at all.

It is also possible to provide your own heap implementation, and even to use two heap implementations simultaneously. Using two heap implementations simultaneously permits task stacks and other RTOS objects to be placed in fast internal RAM, and application data to be placed in slower external RAM.

The RTOS kernel allocates RAM each time a task, queue, mutex, software timer, semaphore or event group is created. The standard C library malloc() and free() functions can sometimes be used for this purpose, but **...**

1. they are not always available on embedded systems,
2. they take up valuable code space,
3. they are not thread safe, and
4. they are not deterministic (the amount of time taken to execute the function will differ from call to call)

... so more often than not an alternative memory allocation implementation is required.

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, allowing an application specific implementation appropriate for the real time system being developed to be provided. When the RTOS kernel requires RAM, instead of calling malloc(), it instead calls pvPortMalloc(). When RAM is being freed, instead of calling free(), the RTOS kernel calls vPortFree().

Memory allocation implementations included in the RTOS source code download

The FreeRTOS download includes five sample memory allocation implementations, each of which are described in the following subsections. The subsections also include information on when each of the provided implementations might be the most appropriate to select.

Each provided implementation is contained in a separate source file (heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c respectively) which are located in the `Source/Portable/MemMang` directory of the main RTOS source code download. Other implementations can be added as needed. Exactly one of these source files should be included in a project at a time [the heap defined by these portable layer functions will be used by the RTOS kernel even if the application that is using the RTOS opts to use its own heap implementation].

Following below:

- heap_1 - the very simplest, does not permit memory to be freed
- heap_2 - permits memory to be freed, but does coalescence adjacent free blocks.
- heap_3 - simply wraps the standard malloc() and free() for thread safety
- heap_4 - coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option
- heap_5 - as per heap_4, with the ability to span the heap across multiple non-adjacent memory areas

---

heap_1.c

This is the simplest implementation of all. It does *not* permit memory to be freed once it has been allocated. Despite this, heap_1.c is appropriate for a large number of embedded applications. This is because the majority of deeply embedded applications create all the tasks, queues, semaphores, etc. required when the system boots, and then use all of these objects for the lifetime of program (until the application is switched off again, or is rebooted). Nothing ever gets deleted.

The implementation simply subdivides a single array into smaller blocks as RAM is requested. The total size of the array (the total size of the heap) is set by configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

The xPortGetFreeHeapSize() API function returns the total amount of heap space that remains unallocated, allowing the configTOTAL_HEAP_SIZE setting to be optimised.

The heap_1 implementation:

- Can be used if your application never deletes a task, queue, semaphore, mutex, etc. (which actually covers the majority of applications in which FreeRTOS gets used).
- Is always deterministic (always takes the same amount of time to execute) and cannot result in memory fragmentation.
- Is very simple and allocated memory from a statically allocated array, meaning it is often suitable for use in applications that do not permit true dynamic memory allocation.

---

heap_2.c

This scheme uses a best fit algorithm and, unlike scheme 1, allows previously allocated blocks to be freed. It does *not* however combine adjacent free blocks into a single large block (it does not include a coalescence algorithm - see heap_4.c for an implementation that does coalescence free blocks).

The total amount of available heap space is set by configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

The xPortGetFreeHeapSize() API function returns the total amount of heap space that remains unallocated (allowing the configTOTAL_HEAP_SIZE setting to be optimised), but does not provided information on how the unallocated memory is fragmented into smaller blocks.

This implementation:

- Can be used even when the application repeatedly deletes tasks, queues, semaphores, mutexes, etc., with the caveat below regarding memory fragmentation.
- Should *not* be used if the memory being allocated and freed is of a random size. For example:
    - If an application dynamically creates and deletes tasks, and the size of the stack allocated to the tasks being created is always the same, then heap2.c can be used in most cases. However, if the size of the stack allocated to the tasks being created was not always the same, then the available free memory might become fragmented into many small blocks, eventually resulting in allocation failures. heap_4.c would be a better choise in this case.
    - If an application dynamically creates and deletes queues, and the queue storage area is the same in each case (the queue storage area is the queue item size multiplied by the length of the queue), then heap_2.c can be used in most cases. However, if the queue storage area were not the same in each case, then the available free memory might become fragmented into many small blocks, eventually resulting in allocation failures. heap_4.c would be a better choise in this case.
    - The application called pvPortMalloc() and vPortFree() directly, rather than just indirectly through other FreeRTOS API functions.
- Could possible result in memory fragmentation problems if your application queues, tasks, semaphores, mutexes, etc. in an unpredictable order. This would be unlikely for nearly all applications but should be kept in mind.
- Is not deterministic - but is much more efficient that most standard C library malloc implementations.

heap_2.c is suitable for most small real time systems that have to dynamically create tasks.

---

heap_3.c

This implements a simple wrapper for the standard C library malloc() and free() functions that will, in most cases, be supplied with your chosen compiler. The wrapper simply makes the malloc() and free() functions thread safe.

This implementation:

- Requires the linker to setup a heap, and the compiler library to provide malloc() and free() implementations.
- Is not deterministic.
- Will probably considerably increase the RTOS kernel code size.

Note that the configTOTAL_HEAP_SIZE setting in FreeRTOSConfig.h has no effect when heap_3 is used.

---

heap_4.c

This scheme uses a first fit algorithm and, unlike scheme 2, it does combine adjacent free memory blocks into a single large block (it does include a coalescence algorithm).

The total amount of available heap space is set by configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

The xPortGetFreeHeapSize() API function returns the total amount of heap space that remains unallocated (allowing the configTOTAL_HEAP_SIZE setting to be optimised), but does not provided information on how the unallocated memory is fragmented into smaller blocks.

This implementation:

- Can be used even when the application repeatedly deletes tasks, queues, semaphores, mutexes, etc..
- Is much less likely than the heap_2 implementation to result in a heap space that is badly fragmented into multiple small blocks - even when the memory being allocated and freed is of random size.
- Is not deterministic - but is much more efficient that most standard C library malloc implementations.

heap_4.c is particularly useful for applications that want to use the portable layer memory allocation schemes directly in the application code (rather than just indirectly by calling API functions that themselves call pvPortMalloc() and vPortFree()).

If you need to place the heap at a specific memory address then set configAPPLICATION_ALLOCATED_HEAP to 1 in FreeRTOSConfig.h, and declare an array called ucHeap[]. For example, using IAR/MSP430 syntax, the following code will place the array from which heap memory is obtained at address 0x4000.

```
#pragma location=0x4000
```

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

heap_5.c

This scheme uses the same first fit and memory coalescence algorithms as heap_4, and allows the heap to span multiple non adjacent (non-contiguous) memory regions.

Heap_5 is initialised by calling vPortDefineHeapRegions(), and **cannot be used** until after vPortDefineHeapRegions() has executed. Creating an RTOS object (task, queue, semaphore, etc.) will implicitly call pvPortMalloc() so it is essential that, when using heap_5, vPortDefineHeapRegions() is called before the creation of any such object.

vPortDefineHeapRegions() takes a single parameter. The parameter is an array of HeapRegion_t structures. HeapRegion_t is defined in portable.h as

```
typedef struct HeapRegion
{
    /* Start address of a block of memory that will be part of the
heap.*/
    uint8_t *pucStartAddress;

    /* Size of the block of memory. */
    size_t xSizeInBytes;
} HeapRegion_t;
```

The HeapRegion_t type definition

The array is terminated using a NULL zero sized region definition, and the memory regions defined in the array **must** appear in address order, from low address to high address. The following source code snippets provide an example. The MSVC Win32 simulator demo also uses heap_5 so can be used as a reference.

```
/* Allocate two blocks of RAM for use by the heap.  The first is a block
of
0x10000 bytes starting from address 0x80000000, and the second a block of
```

```
0xa0000 bytes starting from address 0x90000000.  The block starting at
0x80000000 has the lower start address so appears in the array fist. */
const HeapRegion_t xHeapRegions[] =
{
    { ( uint8_t * ) 0x80000000UL, 0x10000 },
    { ( uint8_t * ) 0x90000000UL, 0xa0000 },
    { NULL, 0 } /* Terminates the array. */
};

/* Pass the array into vPortDefineHeapRegions(). */
vPortDefineHeapRegions( xHeapRegions );
```

Initialising heap_5 after defining the memory blocks to be used by the heap

`</Snip>`

Questions:

1) Am I correct in my assessment that that the FreeRTOS heap implementation is not fragmentation-safe?

… and if yes …

2) Is there a 'typical/canned solution' for this issue that can be implemented 'on top' of FreeRTOS/CMSIS? E.g. the Memory Pool scheme.

See also:

http://asf.atmel.com/docs/latest/uc3b/html/group__membag__group.html