Memory Management

Part 1

Niall Murphy

Class ETP-246

Embedded Systems Conference West

San Francisco, March 2005

email: nmurphy@panelsoft.com

web: http://www.panelsoft.com

Part 2 of this class will be presented as ETP-266, and the paper for that portion of the class is available separately.

INTRODUCTION

Every program uses random access memory (RAM), but the ways in which that memory is divided out among the needy parts of the system varies widely. This paper surveys the options available, in the hope that the reader will be better equipped to choose an approach for a given project.

The mechanisms include statically allocating all memory, using one or more stacks, and using one or more heaps. In particular this paper will examine how the heap is implemented, and how that implementation an be modified to suit the needs of an embedded system. In part 2 of this paper we will examine adding debug code to the stack implementation in order to track down memory leaks.

STATIC ALLOCATION

If all memory is allocated statically, then it can be established at compile time exactly how each byte of RAM will be used during the running of the program. This has the advantage, for embedded system, that the whole issue of bugs due to leaks and failures due to fragmentation simply does not exist. Many compilers for 8-bit processors such as the 8051 or PIC are designed to perform static allocation. All data is either global, file or function static, or local to a function. The global and static data is allocated in a fixed location, since it must remain valid for the life of the program.

The local data is stored in a block set aside for each function. This means that if a function has a local variable *x*, then *x* is stored in the same place for every invocation of that function. When the function is not running, then that location is not used. This approach is generally used in C compilers when the hardware is not capable of providing suitable support for a stack.

This approach prohibits the use of recursion, function pointers, or any other mechanisms that require re-entrant code. For example an interrupt routine can not call a function that may also be called by the main flow of execution.

Some clever compilers may establish that two particular functions can not be simultaneously active, and so allow the memory blocks associated with those functions to overlap. Similarly the application could choose to reuse a globally declared buffer for a number of different purposes, so long as the programmer is happy that the buffer will not simultaneously be required for two different purposes. Such an approach is error prone, not least because the name of the buffer may only match one of its purposes.

To benefit from the inherit memory safety of a completely static environment, it is important that the programmer avoids introducing dangers by trying to implement dynamic memory (such as reusing global data for different purposes) on top of the static

environment. For large systems this is not feasible since a enormous amount of RAM would eventually be required to satisfy every possible requirement of the program.

STACK BASED MANAGEMENT

The next step up in complexity is to add a stack. Now a block of memory is required for every call of a function, and not just a single block for each function in existence. The blocks are now stored on a stack, which usually has some hardware support including special instructions in the processors instruction set.

The stack grows and shrinks as the program executes, and for many programs it is not possible to predict, at compile time, what the worst case stack size will be. In a multi-tasking system there will be one stack per task to manage, (plus possibly an extra one for interrupts). Some judgement must be exercised to make sure that each stack is big enough for all of its activities. It is an awful shame to suffer from an untimely stack overflow, when one of the other stacks has a reserve of space that it never uses. Unfortunately most embedded system do not support any kind of virtual memory management that would allow the tasks to draw from a common pool as the need arises.

One rule of thumb is to make each stack 50% bigger than the worst case seen during testing. In order to apply this rule it is necessary to know how big the stack, or stacks, became during testing. One simple technique is to paint the stack space with a simple pattern. As the stack grows and shrinks it will overwrite the area with its data. At a later time a simple loop can run through the stack's predefined area to detect the furthest extent of the stack. Figure 1 shows an example of the life of a simple stack.

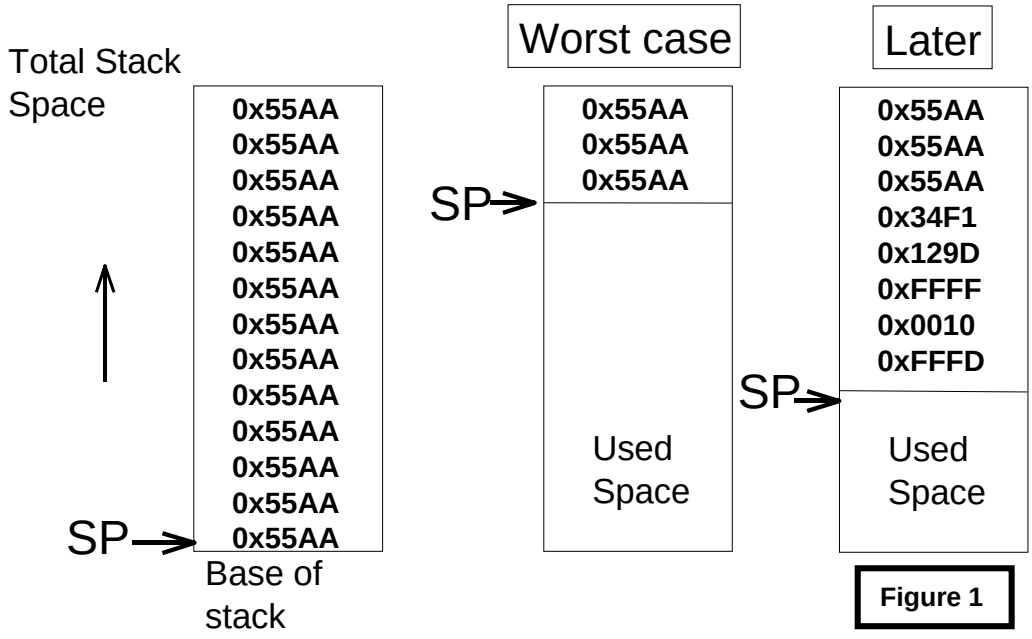

Figure 1

Many RTOS's support this mechanism. If yours does not, or if you have no RTOS, then it is not difficult to implement it yourself – though it is likely to be non-portable. The technique can be used during the test phase to decide on stack sizes, and it can also be used on a production system to give early warning of a stack that exceeds a watermark that the designers did not expect to be reached. In this case the watermark level on the stack is checked to see if the pattern has been overwritten. There is no need to do an expensive measurement of the exact extent of the stack. It would be difficult and expensive to check the watermark on every write to the stack, but it can easily be checked on a timed basis. I have found it convenient to check it at the same time that I am strobing the watchdog.

HEAP BASED MANAGEMENT

Many objects, structures or buffers require a lifetime that does not match the invocation of any one function. This is particularly true in event-driven programs, which is typical of many embedded systems. One event may cause an item to be created, and that item will remain in use until some other event leads to its demise.

In C programs heap management is carried out by the malloc() and free() functions. Malloc() allows the programmer to acquire a pointer to an available block of memory of a specified size. Free() allows the programmer to return a piece of memory to

the heap when the application has finished with it. In this way a piece of memory that is used to store a buffer of data from a serial port at one point in time may be used to store a structure controlling a graphics window at another time. The programmer has a simple interface to the heap so it is not necessary for the programmer to establish at design time which items are not going to be in use simultaneously.

While stack management was handled by your compiler, using heap management requires care by the programmer, or a number of particularly devious bugs can creep into your program.

At a certain point in the code you may be unsure if a particular block is no longer needed. If you free() this piece of memory, but continue to access it (probably via a second pointer to the same memory), then your program may function perfectly, until that particular piece of memory is reallocated to another part of the program. Then two different parts of the program will proceed to write over each other's data. If you decide to not free the memory, on the grounds that it may still be in use, then you may not get another opportunity to free it, since all pointers to the block may have gone out of scope, or been reassigned to point elsewhere. In this case the program logic will not be effected, but if the piece of code that leaks memory is visited on a regular basis then the leak will tend towards infinity, as the execution time of the program increases. So the amount of physical memory will decide how long the program can execute. On many desktop applications a small leak is acceptable, say a compiler which leaks 100 bytes for every 1000 lines compiled. Such a program can still happily compile a 100,000 line file on a modern PC, since on exit of the program all allocated memory will be recovered. However, on many embedded systems no upper limit on the life of the program is acceptable.

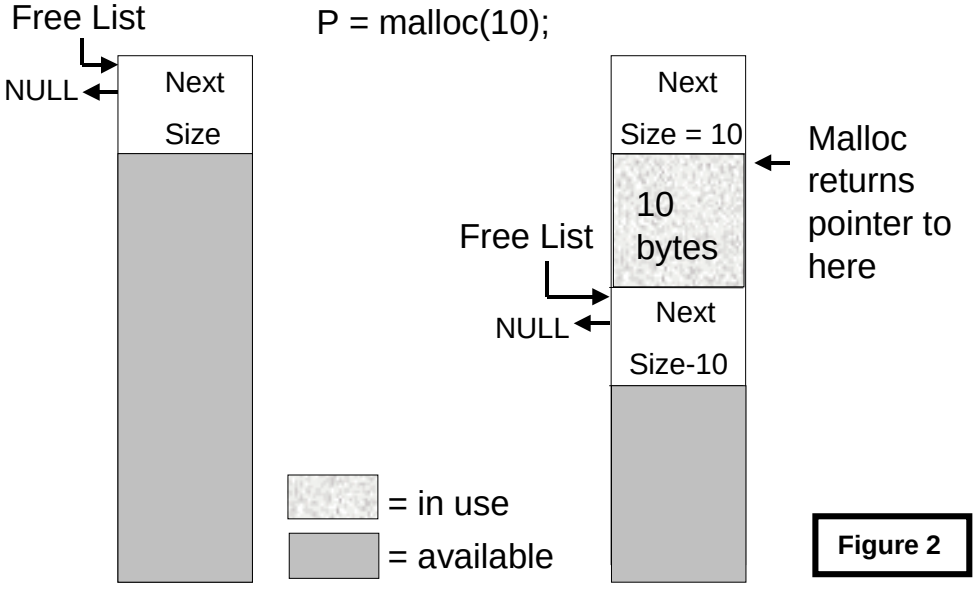

Figure 2

Any leak is a bug, which can be rectified by correcting the logic of the program. There is another problem called *fragmentation*, which can not be corrected at the application program level. This is a property inherent in most implementations of malloc(). It is caused by the blocks of memory available being broken down into smaller pieces as many allocations and frees are performed.

Does this mean that malloc() and free() can not be used in an embedded system? Well, they can, but there are so many restrictions that in many cases programmers choose against it, or they write their own restricted versions of malloc() and free(). We will examine how malloc works to better understand where its limitations lie. The following description is of a typical implementation, but the standard C specification does not demand that it be implemented this way.

The heap is a block of memory which will contain blocks of memory that have been allocated to the application, and blocks which are free. Each block also contains a header. Figure 2 shows a heap in its initial state and the result of a single allocation of 10 bytes. The Free List pointer always points to the first available block. When an allocation is requested this list is iterated, searching for a block to return. Ideally a block of exactly the right size is available, but if not, some larger block is broken into two. In this way, an initial heap of one large block can become a heap which contains a linked list of many small blocks which are free, interspersed with many blocks which have been allocated to the application.

Figure 3 shows the heap after a number of allocations. On the left hand side, the free list still only contains a single element. Now one of the blocks is freed and the right hand side shows a free list with a second element The available bock is of size 15 bytes. If an allocation of 10 bytes took place, then the block of 15 may be broken down into a block of 10 and a block containing the remainder. The remainder block may be so small that no request is ever made that it can satisfy. While free blocks such as this may be merged later with adjacent free blocks, there is the danger that some will be lost forever.

While the danger of fragmentation has been overestimated by many experiments that used random request patterns, it still adds a level of uncertainty that is unacceptable in many systems. In practice requests tend to come in a limited number of sizes. In a survey of a number of Unix applications it was found that 90% of allocations were covered by 6 sizes. 99.9% of allocations were covered by 141 sizes[1]. I believe that in embedded systems the range is far smaller, since file and string handling is much rarer in embedded applications. This means that the chances of finding a block to satisfy the exact size of any one request is far higher than would be estimated given a random distribution of requests.

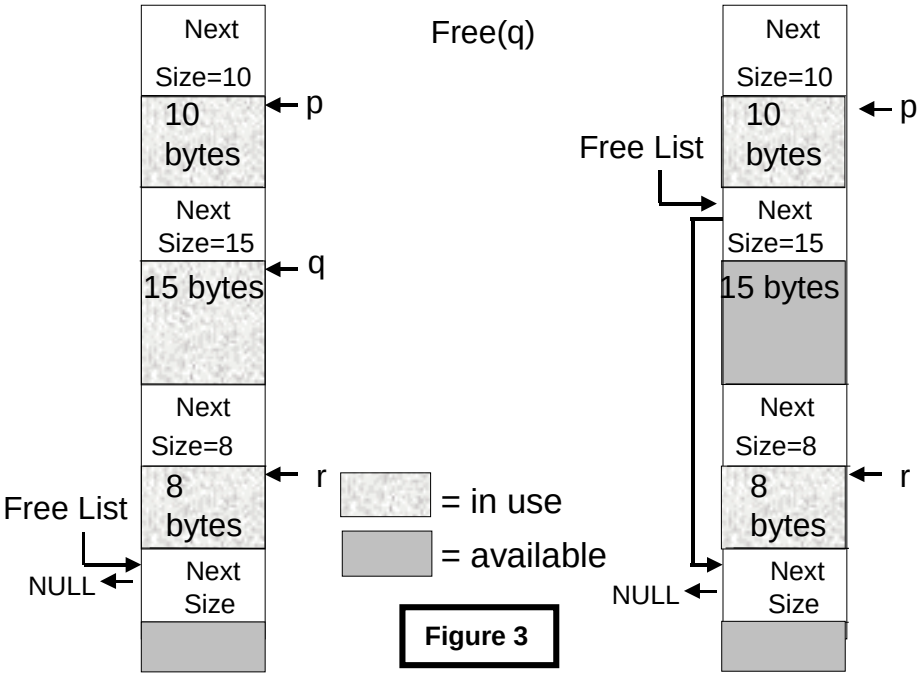

Figure 3

Fragmentation can also be reduced by using the appropriate policy when allocating and freeing blocks. Allocation policies include:

- Allocate (and possible split) first block found, larger than the request (First Fit)
- Allocate the best fit after an exhaustive search (Best Fit).

Free list policies include:

- Maintaining the list in order of address, to simplify merging of free blocks.
- Maintain the list in most recently used order, to match patterns of use where similar sizes are allocated and freed in bursts.

Unfortunately the policies that lead to least fragmentation (Best Fit and address order lists) take the most time to allocate and free blocks. So the choice of algorithm is going to involve trade-offs.

Careful design of the heap mechanism can lead to systems which suffer fragmentation loses of only 1% in unix applications. This is a small amount if it I constant, but it is difficult to establish that a program will not make a pattern of requests that increases that amount at some later point in its lifespan. The conclusion is that mission critical projects can not afford this mechanism, but systems that need to be very reliable, but not 100% reliable, can afford to use a heap, if appropriate testing and measurement is performed.

STATIC WITH ALLOCATION

Projects that either do not need the complexity of a full heap, or can not afford the risk of fragmentation, can use a technique which allows allocation, but not freeing. This means that so long as a program managed to complete its initialization code, the main loop of the program (or the loop of each of its tasks) will not allocate any further memory. This can be performed with the normal malloc() routine, but I find it useful to code a simple version which does not have the overhead of the block headers. It has the following advantages over using malloc():

- The overhead of the headers on each block is avoided.
- The routine can be disabled once initialization is complete.

It has the following advantages over declaring all memory globally.

- Different start up sequences can allocate memory to different purposes, without the programmer having to explicitly consider which items can be active simultaneously.
- The namespace does not get polluted as much. In many cases a pointer to an item may exist, but there is no need for a global or file static to exist for the item itself. Creating the global or file static allows access to the item from inappropriate parts of the code.
- It is easy to transition to using a free() function later.

The following code implements the simple allocator. The only thing that might need to be added for a production system is a locking mechanism to prevent simultaneous access from a number of tasks.

```
#define SALLOC_BUFFER_SIZE 90000
static unsigned char GS_sallocBuffer[SALLOC_BUFFER_SIZE];

static Boolean FS_enabled = TRUE;
int GS_sallocFree = 0;

void *salloc(int size)
{
    void *nextBlock;
    assert(FS_enabled);
    if(GS_sallocFree + size > SALLOC_BUFFER_SIZE)
        assert(FALSE);

    nextBlock = &GS_sallocBuffer[GS_sallocFree];
    GS_sallocFree += size;
    return nextBlock;
```

```
}

void sallocDisable(void)
{
    FS_enabled = FALSE;
}
```

While this approach is memory-safe in comparison to heap usage, it consumes far more RAM than a design that uses free(). However that amount of RAM can be determined by a single run of the system, and will not vary after the sallocDisable() function has been called at the end of the start-up sequence.

POOLS

Now we will return to schemes that allow the application to free memory. Pools, or partitions, of fixed size memory blocks are one technique that can be used to reduce 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.

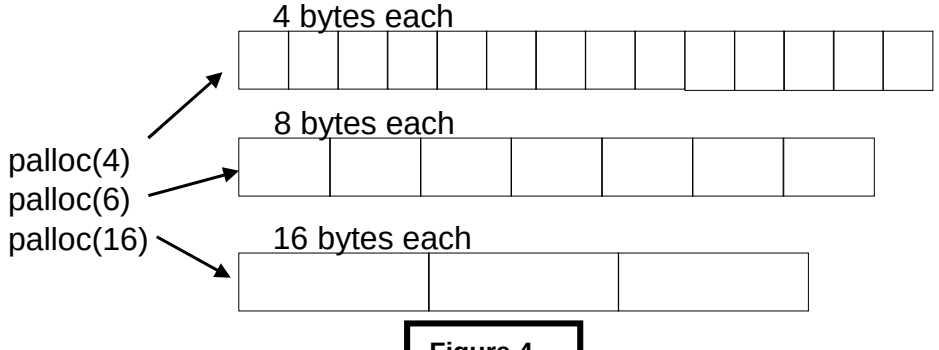

Figure 4

Each pool contains an array of blocks. Unused blocks can be linked together in a list. The pools themselves are declared as arrays, or an equivalent. This mechanism avoids the overhead of a header for each block, since the address of a block can be mapped to the pool and position in which it lives. Figure 4 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 in each pool. Defining pools at sizes which are powers of 2 (2, 4, 8, 16, 32, 64 …) is a good starting point to use if size measurements have not been taken for your application.

Many RTOS's provide a mechanism similar to this, but then make the mistake of forcing the application to refer explicitly to the pool from which the allocation is being made. A safer solution is to get the allocation routine to decide on the appropriate pool, so all of the information about the number and type of pools can reside in the allocation and free routines, so the application programmer need not be aware of it. The free routine can calculate the pool to which the block belongs based on the address of the pool, and then mark that block free by adding it to the linked list of free blocks within that pool.

By monitoring exactly 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 memory that can never be used by the program.

Some versions of malloc() actually use pools for small requests, and use a general purpose heap for large requests [2]. To make such an implementation general enough that it can be used without tuning the sizes of the pools, then it has to be possible to allocate space from the general purpose heap to create and extend the pools. While such a scheme is quite flexible, such a hybrid scheme can eventually have the same fragmentation problems of any general purpose heap.

MULTI-TASKING

While each task must have its own stack, it may or may not have its own heap, regardless of whether the heap is based on the static allocation scheme, pools, or a general purpose allocation scheme. Having more than one heap means that you have to tune the size of a number of heaps, which is a disadvantage. However one heap for many tasks must be reentrant, which means adding locks that will slow down each allocation and deallocation.

A single heap also allows one task to allocate a piece of memory which may be freed by another task. This is useful for passing inter-task messages. When memory is passed between tasks in this way, make sure that it is always well defined who owns the memory at each point. It is obviously important that two tasks do not both believe that they own a piece of memory at the same time leading to two calls to free the memory.

LIBRARIES

Libraries, whether written in-house or purchased from a third party, can cause many difficulties in garbage collection. The author of the library does not have full knowledge of how the library is going to be used. A library may allow the application code to create and object, or allocate memory in some way, but the library may not be

able to free that item because the library does not know when the application has finished with it.

Consider a library that concatenates two strings and returns the result in a newly allocated block. The library can not tidy up the string later, because the library does not know when the application has finished with it. One possibility is that the library has a routine which the application calls when it has finished with the item. Another approach is that the library always uses a static space, so that the string returned is valid until the next time that function is called. This latter idea is not suitable for reentrant code, which is so essential to many embedded systems.

So most libraries, especially object oriented libraries, will have to allocate storage at some time that the application will have to free. In such cases the rules must be very explicit and clear, and the author of the library must ensure that these rules are communicated to the application writer.

Some libraries will allow the application to specify which malloc() and free() functions should be used for its heap management. This allows the application to manage its own memory separately from the libraries. By using debug versions of malloc() and free(), the designer can distinguish between a leak in the application and one contained within the library.

The real difficulty with libraries allocating objects is that it can be difficult to force an application to abide by its memory management rules. In other cases the library may create objects of which the application is not explicitly aware, and therefore does not free. So libraries are a major motivation for automatic garbage collection. Automatic garbage collection is the next step in memory management, but is beyond the scope of this paper.

CONCLUSION

This paper has described memory management policies from simply statically allocating everything to sophisticated heap implementations. Hopefully the reader now has the resources to decide which level of management is appropriate to their next project.

REFERENCES

[1] Wilson, Paul and Johnstone, Mark, *The Memory Fragmentation Problem: Solved?*, International Symposium on Memory Management , Vancouver, Canada, October 1998. Available at http://www.cs.utexas.edu/users/oops/papers.html

[2] Lethaby, Nick and Black, Ken, *Memory Management Strategies for C++*, Embedded Systems Programming, July, 1993.

BIOGRAPHY

Niall Murphy has been writing software for user interfaces and medical systems for seven years. he is the author of *Front Panel: Designing Software for Embedded User Interfaces*, and writes the Murphy's Law column for Embedded Systems Programming magazine. Murphy's writing and consulting business is based in Galway, Ireland and he welcomes feedback at nmurphy@panelsoft.com, or via his web page at http://www.panelsoft.com.