BARR group
The Embedded Systems Experts

How to Allocate Dynamic Memory Safely

By Niall Murphy

Whether you're using only static memory, a simple stack, or dynamic allocation on a heap, you have to proceed cautiously. Embedded programmers can't afford to ignore the risks inherent in memory utilization.

Every program uses random access memory (RAM), but the ways in which that memory is divided among the needy parts of the system varies widely. This article surveys the options available in hopes 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 a heap. We will examine how the heap implementation can impact fragmentation and real-time performance.

Static Memory Allocation

If all memory is allocated statically, then exactly how each byte of RAM will be used during the running of the program can be established at compile time. The advantage of this in embedded systems is that the whole issue of memory-related bugs—due to leaks, failures, and dangling pointers—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 static 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, that location is usually not used. This approach is used in C compilers when the hardware is not capable of providing suitable support for a stack. Figure 1 shows the memory organization with no heap and no stack, just globals and one static block per function.

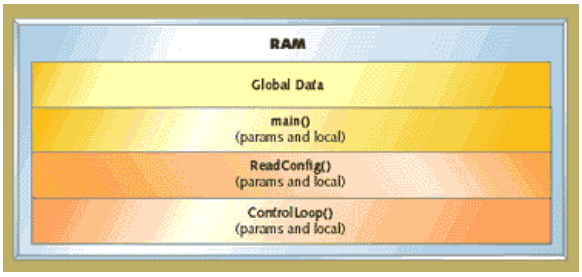

Figure 1. Memory organization with no heap and no stack

This approach prohibits the use of recursion or any other mechanism that requires reentrant code. For example, an interrupt routine can't call a function that may also be called by the main flow of execution. In return for this loss of flexibility, the programmer is guaranteed no run-time memory allocation issues. It might be useful if all compilers gave the programmer the option of not using the stack. By statically

defining all of the space, the programmer sacrifices some flexibility and efficiency, in exchange for extra robustness.

Some clever compilers may establish that two particular functions can't be simultaneously active and, so, allow the memory blocks associated with those two functions to overlap. This approach puts an extra restriction on the code that function pointers can't be used.

To benefit from the inherent memory safety of a completely static environment, it's important that the programmer avoid 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, completely static allocation is not feasible since an enormous amount of RAM would eventually be required to satisfy every possible execution path of the program.

Stack-Based Memory 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 stored on a stack, and are usually called stack frames.

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

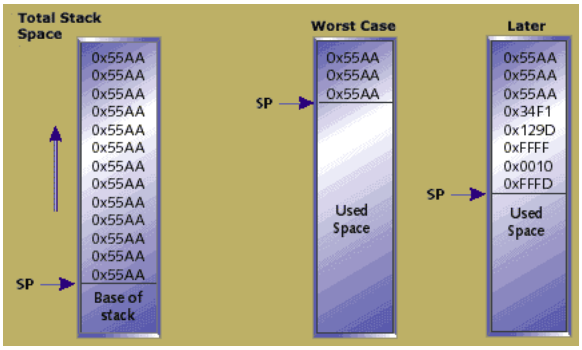

Figure 2. The life of a simple stack

One rule of thumb is to make each stack 50% bigger than the worst case seen during testing. In order to apply this rule, the programmer must 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 2 shows an example of the life of a simple stack. The simple pattern written to the stack should be non-zero, since it is quite common to have data on the stack which has been assigned to zero. It would be difficult to distinguish this data from unused stack space.

Many RTOSes offer a stack size tracing feature. If yours does not, or if you are not using an RTOS, it's not difficult to implement it yourself, though it is likely to be non-portable. The technique can be used during the testing phase to refine the 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. An expensive measurement of the exact extent of the stack is unnecessary. Checking the watermark on every write to the stack would be difficult and expensive, but it can be checked easily on a timed basis. This may not catch a stack overrun due to infinite

recursion, which would overflow the stack very quickly, but it would catch a case where the stack grew a small amount bigger than the designers expected.

The previously described technique fails in one scenario. Consider a large local array which extends beyond the top of the stack. If the program does not write any data to the array, the pattern will not get overwritten. The highest legal piece of stack space will contain the pattern, and so it will look as if the stack did not overflow. Data pushed onto the stack will overwrite some other area of memory, but checking the stack will indicate no problem. If you guess that this is what has happened then the easiest way to check is to make the stack size much bigger, and check the size again. Now that the array is within the bounds of the bigger stack, the true worst case stack size will be found.

Heap-Based Memory 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. The malloc() function allows the programmer to acquire a pointer to an available block of memory of a specified size. The free() function allows the programmer to return a piece of memory to the heap when the application has finished with it.

While stack management is handled by your compiler, heap management requires care by the programmer. A number of particularly devious bugs can creep into your program by way of the heap.

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), 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 affected. But if the piece of code that leaks memory is visited on a regular basis, the leak will tend towards infinity, as the execution time of the program increases.

Ultimately, 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 1,000 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. Any memory leak is a bug and should be rectified by correcting the logic of the application program.

In addition to leaks, there is another problem called fragmentation, which can't be corrected at the application level. This problem is 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() cannot be used in embedded systems? No, 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(). In order to better understand where the

limitations lie, we will now examine how malloc() works. The following description is of a typical implementation, but the standard C specification does not demand that it be implemented this way.

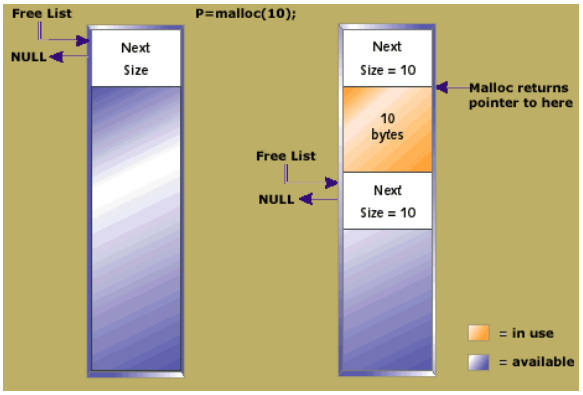

Figure 3. A heap in its initial state and after a single allocation of 10 bytes

The heap is a large block of memory that is made up of smaller blocks of memory to to the application and blocks that are free. Each block, allocated or freed, contains a header. Figure 3 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. If not, some larger block is broken into two. In this way, an initial heap of one large block can become a heap containing a linked list of many small blocks that are free, interspersed with many blocks that have been allocated to the application.

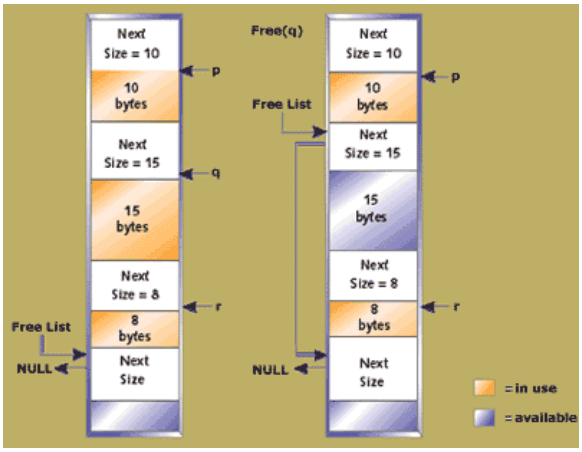

Figure 4. The heap after a number of allocations

Figure 4 shows the heap after a number of allocations. On the left-hand side, the free list still only contains a single element. Next, one of the blocks is freed and the right-hand side shows a free list with a second element. The available block is of size 15 bytes. If an allocation of 10 bytes took place, 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 a danger that some will be lost forever.

The danger of fragmentation has been overestimated by academic experiments that focused on randomly sized allocations. In practice, allocations 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 six sizes, and 99.9% of allocations were covered by 141 sizes. [1] This means that the probability of finding a block that exactly matches the size of any given request is far higher than would be estimated given a random distribution of allocation sizes.

I believe that, in embedded systems, the variety of sizes allocated in any one application is even smaller. File and string handling are rarer in embedded applications, and those are areas

where allocation sizes tend to vary the most. Allocation of space for data structures will be more restricted, since their size does not vary at run time. While the request pattern may reduce fragmentation, we still want our malloc() and free() code to keep it to a minimum.

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

- *First Fit*: allocate (and possibly split) the first block found that is large enough to fulfill the request
- *Best Fit*: allocate the best fit after an exhaustive search

Possible free list management policies include:

- *Address Order*: Sort the free list in order of address, to simplify merging of adjacent free blocks
- *Recently-Used Order*: 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 that suffer fragmentation losses of only 1% in Unix applications.[2] This is a small amount if it is constant, but it's difficult to establish that a program will not make a pattern of requests that increases that amount at some later point in its execution. The conclusion is that heap use does involve an element of risk, which the programmer may choose to accept in return for a more flexible, RAM-efficient system.

Static Memory Preallocation

Projects that either do not need the complexity of a full heap or can't afford the risk of fragmentation, can use a technique that allows allocation, but not freeing. This means that after a program has completed its initialization code, the main loop of the program (or the loop of each of its tasks) will not allocate any further memory. This technique can be implemented with the normal malloc() routine, but I've always found it useful to write a custom version. My custom version has the following advantages over using the normal malloc() routine:

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

This technique also 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.

```
#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;
}
```

Listing 1. Simple allocator code

Listing 1 contains the simple allocator code. One thing that might need to be added for a multitasking system is a locking mechanism to prevent simultaneous access from a number of tasks. This allocator also does not handle memory alignment issues that you may need to address, depending on the alignment restrictions of your platform. While this approach is memory-safe in comparison to heap usage, it consumes far more RAM than a design that also 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.

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.

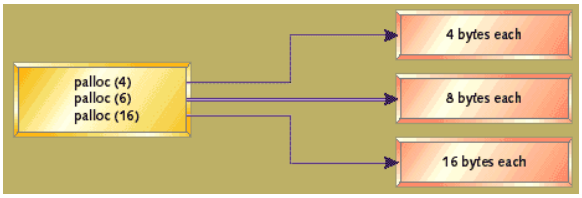

Figure 5. Using the appropriate pool

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]

RTOS Memory Partitions

Many RTOSes provide a memory pool mechanism, usually called partitions. Partitions are useful for implementing the pool-based heap described above. If you use a number of partitions to implement your heap, then avoid using the RTOS calls directly. You may have calls such as those in Listing 2.

```
block1Ptr = partitionGetBlock(partitionOfBlocksSized1000);
block2Ptr = partitionGetBlock(partitionOfBlocksSized200);

partitionFreeBlock(block1Ptr, partitionOfBlocksSized1000);
partitionFreeBlock(block2Ptr, partitionOfBlocksSized200);
```

Listing 2. The **wrong** way to allocate from partitions

This format is typical of many RTOSes. Clearly, the onus is on the application programmer to be certain that a block is returned to the partition from which it was allocated. When implementing your pools, hide the calls to the partition code. Pass the size required to the allocation function and allow it to decide on the best partition to use. Only pass the pointer to the block to the free() function. It should be able to derive the partition from the address of the block. So the code snippet above will change to something much more maintainable like:

```
block1Ptr = myAlloc(1000);
block2Ptr = myAlloc(200);

myFree(block1Ptr);
myFree(block2Ptr);
```

Listing 3. A better way to allocate from partitions

Multitasking Memory Management

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.

It may be necessary to allow 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 simultaneously. If this happened, it could lead to two calls to free the same memory block.

Memory Management Libraries

Libraries, whether written in-house or purchased from a third party, can cause many difficulties in memory management. 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 an 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't tidy up the string later because the library doesn't know when the application has finished with it. One possibility is that the library has a destroyString() routine

that the application calls when it has finished with the item. This has the disadvantage that the onus is on the application to remember to call this function. 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, at which time that space will be overwritten with the next result. This latter idea is not suitable for reentrant code, which is so essential to many embedded systems.

Many libraries, especially object-oriented libraries, will 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.

Final Decisions

This overview of memory management should have given the reader some ideas about what approach is right for their project. The final design decisions will be based on a combination of how much RAM is available, the complexity of the application, and whether third-party software is involved.

Endnotes

1. Johnstone, Mark S., and Paul R. Wilson. "The Memory Fragmentation Problem: Solved?". International Symposium on Memory Management, Vancouver, British Columbia, Canada, October 1998. Available at CiteSeerX @ Penn State [back]

2. Ibid. [back]

3. Lethaby, Nick and Ken Black. "Memory Management Strategies for C++". Embedded Systems Programming, July 1993, p. 28. [back]

Related webinars and topics can be found at:
https://citeseer.ist.psu.edu/myciteseer/login