Dynamic Memory Allocation
Implementation Explained
Author: Zachary Tyynismaa

Background:

The purpose of this library is to dynamically allocate memory on the ATMEGA48P. Loosely defined dynamic memory management means the managing of memory resources at runtime such that (within certain limitations) the allocation and use of memory automatically provides a pointer to a memory location that is continuous for as many bytes as requested and providing that all memory allocation is handled by the routine, will not be allocated by any subsequent calls until it is freed.

Implementation:

In the provided implementation it is necessary to divide the memory space so that the dynamically allocated memory(from here on referred to as the “Heap”) is free from the interference of other operations which may use memory namely the system Stack. As the Stack can grow during runtime it is necessary that there be a sufficiently large gap between our heap and stack so that the Stack does not overflow into our heap.

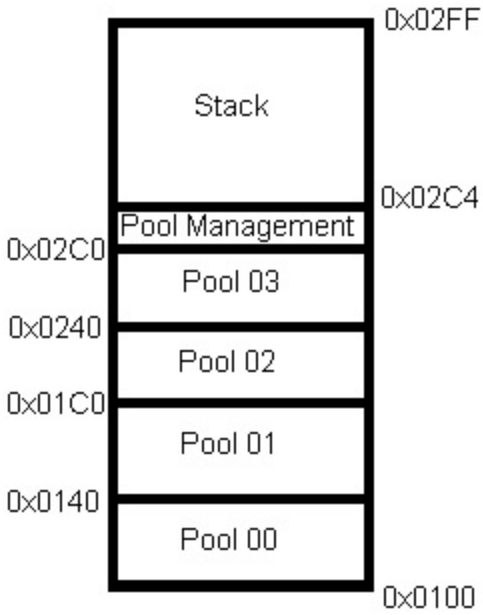
For this implementation, working with our limited amount of memory, the division was determined as follows. As our memory is limited to 512 Bytes it is obviously necessary that the heap does not surpass this size. Secondly while it must have an adequate buffer between itself and the stack it should not be unnecessarily small. Based on this I arbitrarily decided that ~1/8th of the system memory (~64 Bytes) would provide enough buffer for the Stack during most imaginable routines.

Proceeding from here as it was decided I would implement a pool and block Heap where the Heap is divided into several pools and each pool into subsequent blocks. In this implementation it was necessary to reserve an additional byte of memory in addition to the Heap size for each pool being implemented. Recognizing this I wished to limit this number to 4 Bytes which would come from the lower portion of what would otherwise be Stack memory. This now puts the Stack buffer firmly at 60 Bytes. This also leaves us with 7/8ths of the memory available for use by the Heap. In dividing this up it worked out well so that 3 pools had 1/4th of the memory will the 4th had 1/8th.

Doing the math, the size for the pools works out to the following:

POOL00 = 64 Bytes
POOL01 = 128 Bytes
POOL02 = 128 Bytes
POOL03 = 128 Bytes

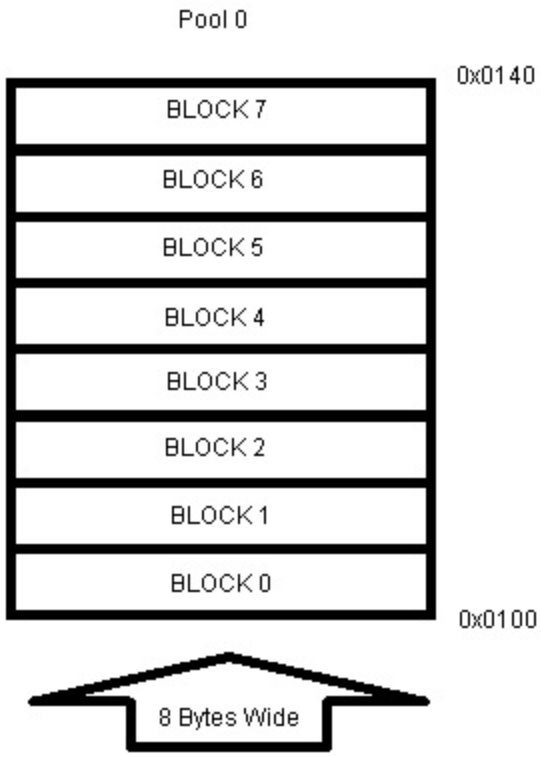
It follows from this that the allocation of memory in the system, depicted graphically, would resemble the following:



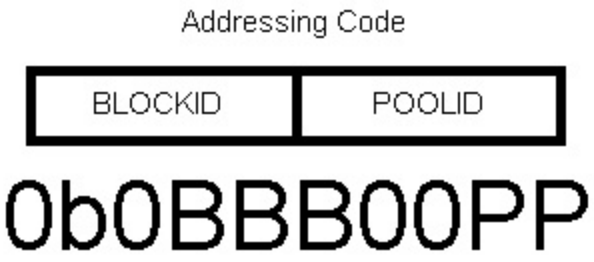
Note: Not to Scale

Management:

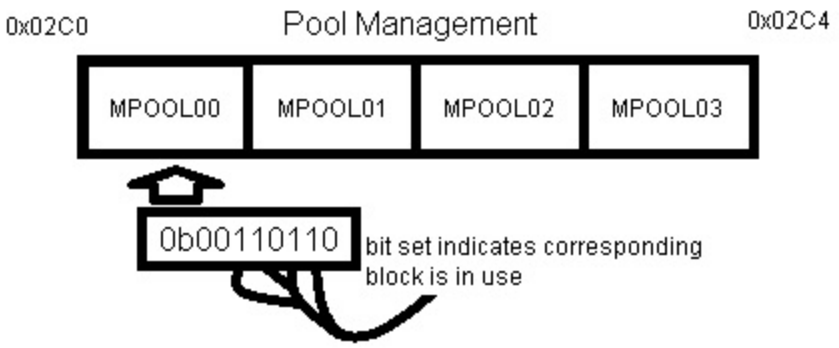
Being able to visualize this you should now understand where the Heap is and how it is divided in memory. The next question is how do we manage this memory? In order to understand how this management is performed it is first necessary to understand the pool blocks. With one bytes corresponding to each Pool we can start to manage the pools if we divide the pools into 8 blocks(see below).



With each block corresponding to both a location and amount of memory it is now possible to refer to an address space by a handle or addressing code.



With these our routine can load and set bits in out management bytes as well as return addresses. To update our management bytes a simple encoding mechanism is used.



As you can see we use our pool ID to specify which Pool management byte to load and we update to reflect any changes in pool availability. In addition we can use the addressing code to determine and return the block address.

Conclusion:

> In conclusion you should now understand at least vaguely how this library handles memory management. The end user just needs to call malloc to get memory and free when done and this routine will handle all the memory for you.