

(Part of the)
FYSOS Specification
Revision 3.xx.xx

20.2 Memory Allocation

The Kernel can allocate memory using the following attributes passed to the Heap Allocator. Some of these attributes can be combined while others must be specifically individualized.

Table 20.2.1: Memory Allocate Attribute

Value	Description
MALLOC_FLAGS_VIRTUAL ¹	Any memory region, virtually contiguous
MALLOC_FLAGS_PHYSICAL ¹	Must be a Physical Address memory region returned
MALLOC_FLAGS_CLEAR	Clear memory region before returning to caller
MALLOC_FLAGS_LOW1MEG	Must be a region all below the 1 Meg mark
MALLOC_FLAGS_LOW16MEG	Must be a region all below the 16 Meg mark
MALLOC_FLAGS_LOW4GIG	Must be a region all below the 4 Gig mark
MALLOC_FLAGS_ALIGNED	Must be aligned on a given boundary
MALLOC_HARDWARE32	MALLOC_FLAGS_LOW4GIG MALLOC_FLAGS_PHYSICAL MALLOC_FLAGS_ALIGNED
MALLOC_HARDWARE64	MALLOC_FLAGS_PHYSICAL MALLOC_FLAGS_ALIGNED

¹One or the other (not both) must be used for every call.

20.2.1 Virtual

This attribute, with the exception of the CLEAR attribute, is assumed to be the only attribute given and can be any memory region, contiguous or not, physically allocated or linear. This will be the most common type of memory allocated. Most allocation will use this type, so this must be the most efficiently coded allocation. If this flag is not set, the PHYSICAL flag must be set.

20.2.2 Physical

This attribute may be combined with most other attributes and instructs the Allocator that all memory for this request must be physically allocated. i.e.: the memory must be physically present and use a physical address, not a linear address for returned memory region. All memory in this region must also be physically contiguous. If this flag is not set, the VIRTUAL flag must be set.

20.2.3 Clear

This attribute may be combined with any other attribute and simply instructs the Allocator to clear the memory region(s) to zero before returning to the caller. The act of clearing it does not have to be done at this point, just so long as the memory is clear. i.e.: A kernel task can be created to clear memory in the background, when idle, marking regions as clear, allowing the Allocator to simply return.

20.2.4 Low1Meg

This attribute may be combined with most other attributes and instructs the Allocator that all memory for this request must be below the 0x00000000_00100000 (1 Meg) memory mark.

20.2.5 Low16Meg

This attribute may be combined with most other attributes and instructs the Allocator that all memory for this request must be below the 0x00000000_01000000 (16 Meg) memory mark.

20.2.6 Low4Gig

This attribute may be combined with most other attributes and instructs the Allocator that all memory for this request must be below the 0x00000001_00000000 (4 Gig) memory mark.

20.2.7 Aligned

This attribute may be combined with most other attributes and instructs the Allocator that the starting address for this request must be aligned on the given alignment. If an alignment value is given, it must be a power of two. The actual alignment may be adjusted to a larger power of two, simply to ease the alignment code.

20.2.8 Hardware32

This attribute is a combination of previous attributes indicating to the Allocator that all memory must be within the lower 4 Gig Address Space, all must be physically present, and must be continuous, and usually aligned on a given boundary. This attribute is usually used by 32-bit only hardware.

20.2.9 Hardware64

This attribute is a combination of previous attributes indicating to the Allocator that all memory must be physically present, must be continuous, and usually aligned on a given boundary, though can be anywhere in the 64-bit addresses space. This attribute is usually used by 64-bit aware hardware.

20.3 The Virtual Heap

To make it easier on the kernel, a virtual memory heap is created. The heap is a number of blocks of memory available to the kernel. The heap is used as a simple way to allocate and free virtual memory without calling the underlining physical allocator for each and every memory request, making it much faster to allocate memory.

On Kernel start up, it calls the Virtual Memory Allocator to allocate a single linear block of memory, 64 Meg is suggested but not required, for this heap. The memory in the heap may or may not be physically contiguous. This heap is then further managed by a Heap Allocator. All requests for memory go through this Heap Allocator.

If a memory request can be satisfied from this initial block of memory, the Heap Allocator records the allocation within its catalog and returns a pointer to a region large enough to satisfy the request. No call to the Virtual Memory Allocator or Physical Memory Allocator is needed, drastically speeding up the memory request.

If the memory request cannot be satisfied, due to lack of remaining room in the current catalog, a special alignment request, a physical address request, or other situation, the Heap Allocator may then call upon the Virtual Memory Allocator to satisfy the request. The Virtual Memory Allocator may then call upon the Physical Memory Allocator to satisfy the request.

This Virtual Heap Allocator maintains a catalog of virtual memory addresses from `0x00000000_00000000` to the size of the heap set by the Kernel. The Kernel's heap may set this limit to any amount. Any Heap created for a user task may set an initial limit, resizing as needed.

20.3.1 Heap Catalog

The Heap Catalog consists of Buckets and Pebbles. Initially there is a single Bucket of any size holding one to an arbitrary number of Pebbles within it. Each Pebble is marked used or free. If a Pebble is marked free, it may be used to satisfy a memory request. This Pebble can be split into two Pebbles so that the remaining part of the Pebble can then be used to satisfy another request.

If at any time there is not a Pebble within a Bucket to satisfy a memory request, a new Bucket may be allocated, placing free Pebbles within, to satisfy the request.

20.3.2 Buckets

A Bucket is allocated by the Virtual Memory Allocator, is of an arbitrary size, and is considered a contiguous number of virtually allocated bytes from start to end. The starting memory location and any contiguous byte to the end may or may not be physically contiguous, but must be virtually contiguous.

A Bucket is allocated and freed from the Virtual Memory Allocator using two functions, the first may be a function called `mmap()` and the second may be called `mmap_free()`. No other system memory function is required

as far as the Heap Allocator is concerned. All Virtual Memory allocated for a Bucket is cataloged by the Virtual Memory Allocator.

A Bucket maintains only the Pebbles within it and has no awareness of any other Pebbles within another Bucket.

A Bucket may be resized at any time as long as the Pebbles within are maintained. If a Bucket has excess free Pebbles, or a single free Pebble with excess memory allocated for it, a Bucket may shrink to accommodate. If a Bucket doesn't have enough free space within, it may expand to create an addition number of Pebbles. However, any memory added to the Bucket must be contiguously added at the end of the currently allocated memory. i.e.: A Bucket may only increase in size if the added memory is Virtually consecutively added at the end and the new sized Bucket is still completely continuous.

A Bucket has the following format and must be stored in the first bytes of the allocated space returned from the Virtual Memory Allocator.

Table 20.3.2.1: Heap Bucket Format

Name	Size	Description
Magic	4	Signature identifying a valid Bucket
Local Flags	4	Flags for this Bucket
Size	8	Count of pages allocated for this Bucket
Largest	8	Size of largest free Pebble in this Bucket
Previous	8	Pointer to a previous Bucket
Next	8	Pointer to the next Bucket
First Pebble	8	Pointer to the first Pebble in this Bucket

20.3.2.1 Magic

This field is a signature of 0x4255434B, ('BUCK' in little-endian format) indicating that this is actually a Bucket.

20.3.2.2 Local Flags

This field is used to indicate a few attributes about the Pebbles within this Bucket.

Table 20.3.2.2.1: Local Flags

Value	Description
Bit 0	If set, use best method fit. If clear, use first find fit.
other	Not used and is in error if found

Any flag given is for this Bucket only.

20.3.2.2.1 Method Used

Bit 0 is used to indicate which allocation method to use for this Bucket.

If set, the allocator is to use the Best Fit method when allocating a free Pebble within this Bucket. This is to use the smallest free Pebble that will satisfy the request.

If clear, the allocator must use the first free Pebble it finds in this Bucket. This is to speed up the process. See Section 20.3.2.6 for another speed-up process.

20.3.2.3 Size

This field is used to store the number of pages allocated for this Bucket. This value is used for the free function of the Virtual Memory Allocator, so it knows how many virtually contiguous pages to free.

20.3.2.4 Largest

This field holds the size of the largest free block within this Bucket. The Heap Allocator code can compare this field with the request to see if there is a Pebble large enough to satisfy the request. This is so the Heap Allocator does not have to parse any of the Pebbles to see if one is large enough, and can move on to another Bucket if there is none. This field must be maintained after a Pebble in this Bucket is modified.

20.3.2.5 Previous/Next

These two fields create a linked list of Buckets. If a field is non-zero, there is a Bucket in that direction. At least one Bucket must always exist and its previous field must be NULL. If it is the only Bucket, its next field must also be NULL.

20.3.2.6 First Pebble

This field points to the first Pebble in this Bucket. A Bucket must always have at least one Pebble, even if it is a free-for-use Pebble. After a call to free a Pebble, after any merging of Pebbles, if this happens to create a single Pebble within a Bucket and this Pebble is free for use, it is up to the Heap Allocator if it would like to free this Bucket from the Heap's list. Again, making sure to not "free" the first Bucket.

The First Pebble, the Bucket's *First* field, must not be assumed to be just after the Bucket. The First Pebble field may point to any Pebble within the Bucket. An advantage of this is to update the Bucket's first Pebble field to point to the newly freed Pebble, so that when the next allocation is to take place, the first Pebble found is a free Pebble. However, this creates more work when finding a used Pebble. Also, see Sections 20.3.3.8 and 20.3.6 for more reasons why you must not assume the first Pebble starts at offset 48 in the memory region.

20.3.3 Pebbles

Pebbles are stored within a bucket. A Bucket must have at least one Pebble. A Pebble is used to satisfy a memory allocation request.

A Pebble has the following format.

Table 20.3.3.1: Heap Pebble Format

Name	Size	Description
Magic	4	Signature identifying a valid Bucket
Local Flags	4	Flags for this Pebble
Sent Flags	4	Flags from the Sender
Alignment	4	Alignment value
Size	8	Number of available bytes in this Pebble
Name	32	name/comment/signature for this Pebble
Parent	8	Pointer to the parent Bucket of this Pebble
Previous	8	Pointer to a previous Pebble in this Bucket
Next	8	Pointer to the next Pebble in this Bucket

20.3.3.1 Magic

This field is a signature of 0x524F434B, ('ROCK' in little-endian format) indicating that this is actually a Pebble.

20.3.3.2 Local Flags

This field is used to indicate a few attributes about this Pebble.

Table 20.3.3.2.1: Local Flags

Value	Description
Bit 0	If set, is used. If clear, is free for use.
other	Not used and is in error if found

Any flag given is for this Pebble only. No field, flags or other, are used in any other Pebble.

20.3.3.3 Sent Flags

This field is used to indicate the initial flags. For example, if you allocate memory with an alignment value, what happens when you use `realloc()` to request a larger size? This is so your `realloc()` function can fail if it finds that you originally allocated with an alignment attribute and you are asking for a larger size. See Section 20.3.7 for more information on this field.

20.3.3.4 Alignment

This field is used to pass on the Alignment value if this Pebble is to be aligned. This way, a function, such as a call to `realloc()`, can maintain this alignment. Allowed alignment values passed to the Heap Allocator are any power-of-two, from 1 to 1Meg, inclusive. Any value that is not a power-of-two, will be increased to the next closest power-of-two. However, the Heap Allocator's minimal alignment will be 64. Any value below will be returned on an alignment of at least 64 bytes. A value from 64 to 1 Meg fits within a 32-bit unsigned integer, hence the size of the field. However, when using calculations on 64-bit values, remember to use correct casts to a 64-bit value to eliminate errors.

20.3.3.5 Size

This field is the number of bytes within this Pebble assigned for use. This size must be at least 64 and must be a count of 64 bytes in size. i.e.: It must be at least 64, 128, 192, 256, 320, etc. This field may not actually represent the exact number of bytes requested, but must be at a 64-byte count at or above the amount requested. This is to align the next Pebble on a 64-byte count from the end of the previous pebble. This alignment is not to be relied upon, nor assumed. See Section 20.3.6 for reasons why. This Size value does not include the size of the Pebble Structure described in Table 20.3.3.1.

20.3.3.6 Name

This field is an optional field and may contain anything the Kernel wishes to place here. This could simply be a character string of the function or filename that called the allocator. It can also be a signature associated with a driver who has requested it. It is purely optional and used for debugging purposes. Your Heap Allocator must ignore this field other than maintaining this field throughout its life. Your Heap Allocator must work as expected if this field is removed from Table 2.3.3.1. i.e.: No field within that table may rely upon its offset, nor can you assume the size of the table.

20.3.3.7 Parent

This field points to the Parent Bucket this Pebble resides in. All Pebbles, free or not, must maintain this pointer. See Section 20.3.5 on an important safety catch.

20.3.3.8 Previous/Next

These two fields create a linked list of Pebbles. If a field is non-zero, there is a Pebble in that direction. The first listed Pebble must have a Previous field of NULL and the last listed Pebble must have a Next field of NULL.

All Pebbles must remain within the allocated memory of the Bucket. i.e.: When adding a Pebble, it must be within an address that is contained within the memory allocated for the Bucket. No Pebble may be allocated outside of its intended Bucket and no Pebble may point to another Bucket's Pebble.

20.3.4 Splitting Pebbles

When Splitting a Pebble to allocate a portion of the memory allocated to it, you must add at least one Pebble in either direction, updating the two surrounding Pebbles to point to this newly added Pebble. You may only split a Pebble when there is enough room to satisfy the current allocation request *and* a new Pebble will have at least the 64-byte minimum free space allotted for it. If splitting a Pebble does not leave enough room for a new Pebble and this 64-byte space, the Pebble is not to be split and the allocated size must remain as is. This 64-byte space is determined with a `#define` and can be changed. A value that is a power of two is recommended.

When using the Alignment functionality, you may split a Pebble into three contiguous Pebbles. The first will be a free Pebble simply to pad to the next Pebble, which will be the Aligned Pebble. Then this Aligned Pebble may possibly split again to free any excess space after it. This may create three Pebbles when splitting one Pebble for an aligned request. If doing so, the first and the possible third Pebble must remain available for use by another request.

Please note that when splitting in this manner, if the first Pebble will have an available size less than 64, it is recommended to align on the next available alignment, the one after the current calculated alignment, to make the first Pebble have an available space of at or more than 64 bytes. Also note that the Aligned Pebble must start 80 bytes (the size of a Pebble Structure) *before* the alignment value so that the available memory starts on the requested alignment.

Also, please note that if the current Pebble, before you split it into three, already is aligned on the requested alignment, there is no need to create three. Simply return the current Pebble, possibly first splitting it into two to free the excess space.

20.3.5 Merging Pebbles

When two consecutive Pebbles are free, your Heap Allocator should merge these Pebbles to create a single Pebble. To do this, the first Pebble needs to consume the second Pebble. All linked-list pointers must be updated for any surrounding Pebbles. The Size field in the first Pebble is now increased to the size of both Pebbles including the deleted Pebble's block (currently 80 bytes).

Only merge pebbles of the same allocation type. i.e.: Watch when merge two pebbles if one is Virtually allocated and another is Physically allocated. See Section 20.3.7 for more on merging mixed types. This can be determined by the *sflags* field of the Pebble. Be sure to clear the Allocation attribute, as well as any now unused attribute, of any Pebble once you mark it as free.

Be sure to watch that if you delete a Pebble by merging, the Bucket's First Pebble pointer is still valid. i.e.: by deleting this Pebble, first make sure the Bucket wasn't pointing to this Pebble, and adjust if needed.

20.3.6 Aligned Requests

When the caller is requesting a memory address that is aligned on a supplied boundary requirement, it is up to your Heap Allocator to satisfy this request. The manner in which you do is not specified here. However, here are a few examples. See Section 20.3.4 on splitting a Pebble for more on accommodating Aligned requests.

- 1) Your Heap Allocator may adjust a Pebble's location so that the byte just after the Pebble Structure satisfies the alignment request. To do this, you forgo the 64-byte alignment request specified in section 20.3.3.5 and place a Pebble so that it is just before the aligned location. To do this, you must have an original free Pebble large enough to split and place a new Pebble within the allocated space so that this new Pebble's memory region is now aligned.
- 2) You may insert a new Bucket into the Heap, so that the newly created first Pebble is now aligned on the requested alignment. However, this makes it so that your code cannot assume the first Pebble is just after the memory used for the Bucket. This means that there may be unused space between the memory used to store the Bucket data and the first Pebble. However, this is an easy way to allocate aligned and/or physically aligned memory. See the next section for more on this use.

20.3.7 Physical Requests

Physical contiguous memory requests must also be satisfied by the Heap Allocator. Physically allocated Pebbles are allocated, stored, and freed exactly like Virtually allocated Pebbles, only that the *sflags* field is adjusted to indicate which is which. When a new Bucket is created, the Heap Allocator passes a flag to the Virtual Memory Allocator to return memory that is Physically and Contiguous allocated. The Heap Allocator requests the Virtual Memory Allocator to make sure and allocate Physical Memory addresses that are contiguous. There is no difference in the Heap allocation scheme between Virtually stored Pebbles and Physically stored Pebbles, other than the *sflags* member of the Pebble indicating the difference. However, you

should not merge Pebbles that differ in Virtual or Physical allocation unless you mark the newly sized Pebble as Virtual. Splitting either type is allowed as long as the newly split Pebble maintains the *sflags* indication.

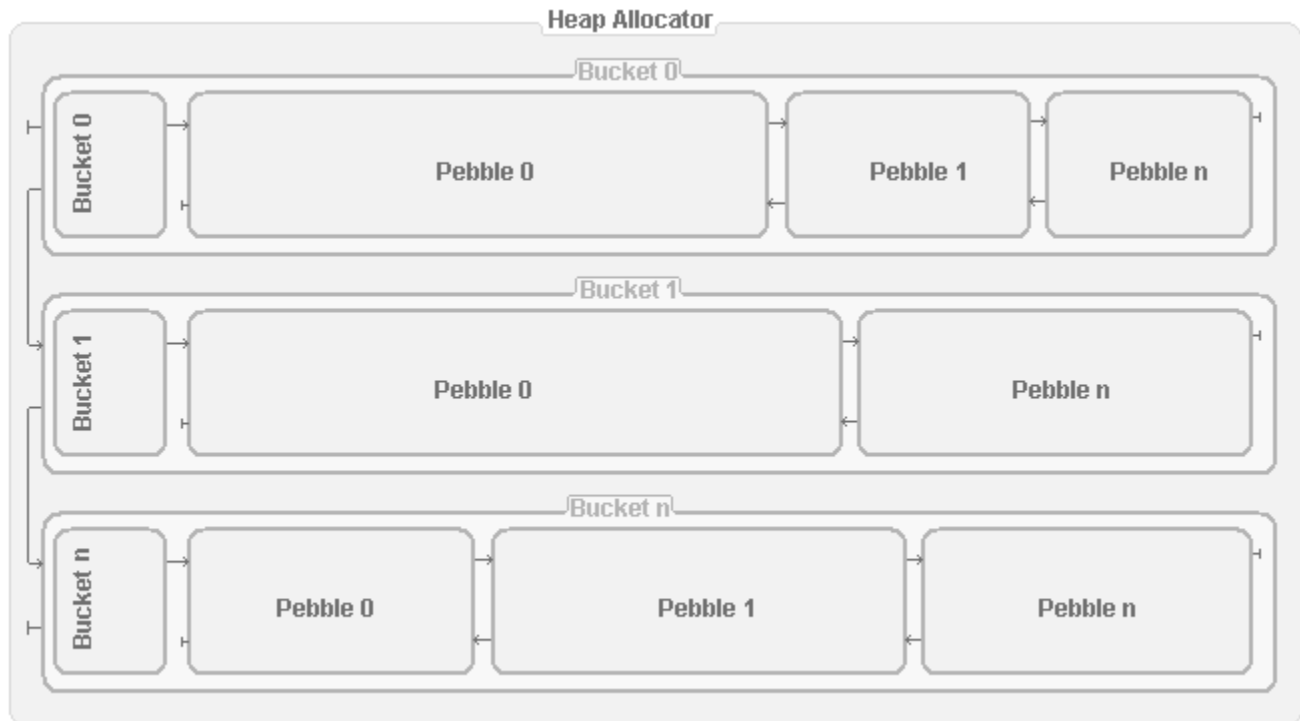
20.3.8 Multithreading/Multitasking

If your Heap Allocator will be used on a multi-process platform, a locking mechanism is strongly suggested. A simple Spinlock will suffice. Lock any code that modifies a Bucket or a Pebble within that Bucket, remembering to unlock when finished.

20.3.9 Example Layout

The following is an example layout of the memory allocated by a Heap Allocator. This example has three Buckets with each Bucket having an arbitrary number of Pebbles.

Figure 20.3.9.1: Example Layout



In this example, there are three Buckets, each having more than one Pebble. The example shows that each Bucket is the same size. This usually is the case, but doesn't have to be. Each bucket can be a different size due to different allocation strategies.

In a Bucket, there are usually no connecting Free Pebbles. i.e.: If two connecting Pebbles are marked Free, the Heap Allocator should merge them.