Memory allocation in C

Leslie Aldridge, August 11, 2008

(This article first appeared in the August 1989 issue of Embedded Systems Programming magazine.)

The high cost of RAM ensures that most embedded systems will continue to experience a shortage of memory, The software you use to implement these systems will use queues, linked lists, task-control blocks, messages, I/O buffers, and other structures that require memory only for a short time and that may return it to serve other functions. This is known as dynamic memory allocation. If you're programming in C, this probably means using the memory allocation and release functions, `malloc()` and `free()`.

Dynamic memory allocation and the structures that implement it in C are so universal that they're usually treated as a black box. In the real world of embedded systems, however, that may not always be desirable or even possible. Not all vendors of C compilers for embedded systems provide the memory allocation and release functions. We recently encountered a compiler that purported to have them but didn't document their interfaces. Because we needed tight control over the memory allocation process, we decided to write our own routines.

Rather than start from scratch, the simplest solution seemed to be to copy the allocator from Kernighan and Ritchie's *The C Programming Language* (Englewood Cliffs, N.J.: Prentice-Hall, 1988). Unfortunately, the functions presented in that inestimable resource are meant to interface to an operating system that will supply large blocks of memory on request. The algorithm we'll use here doesn't differ drastically from K&R's version, but it's clearer and better suited to an embedded system environment. The code works for twobyte address pointers but can easily be modified to handle any size.

A dynamic memory allocator should meet certain minimum requirements:

• All internal structures and linkage must be hidden from the call-only the number of bytes required should have to be specified.

• The order of calls to `malloc()` and `free()` shouldn't matter (the order of calls to `free()` need not be the reverse of the order of calls to `malloc()`).

• `free()` must prevent fragmentation of freed memory; all small, freed blocks must be combined into larger, contiguous blocks where possible.

• Overhead in memory and execution time must be minimized.

• An error condition must be returned if no memory is available.

The design

`Malloc()` allocates memory in units of structures called header blocks. For systems with two-byte memory pointers, where available memory is less than 64 kbytes, each header block comprises four bytes. For larger memory models, the header must be expanded to allow a four-byte pointer and a long integer value for the size of the block. Header blocks are defined as follows:

```
typedef struct hdr {
  struct hdr *ptr;
  unsigned int size;
} HEADER;
```

A forward-linked list, the head of which is `frhd`, keeps track of available space. During system operation, contiguous free blocks combine to form larger blocks. We didn't implement a best-fit scheme for allocation; the first block that's large enough is allocated.

Each allocated block has a header that specifies the length of the block in `HEADER`-size units. When the block is freed, the `free()` function uses the header to link it back into the free list. The caller doesn't see or need to be aware of this header; the pointer to the allocated space points just beyond it.

Initialization

The external variables `_heap-start` and `_heapend` must be defined as the first and last bytes of RAM available to `malloc()`. The linker or binding utility used to build the object program normally provides a facility for defining external names and assigning absolute memory addresses to them. The addresses can be assigned directly in C; for example:

```
#define _heapstart (HEADER *)0x1000
#define _heapend   (HEADER *)0x2000
```

The application must call `i_alloc()` prior to calling `malloc()` or `free()`. Normally called from `main()`, `i_alloc()` initializes the free-space pointer to indicate the first available byte of heap space. (By convention, *heap* generally refers to the pool of memory available for dynamic allocation at run-time.) The free-space pointer initially indicates a header showing that the entire heap area is free and no other free blocks exist (see **Figure 1**). The size of the free space is calculated in `HEADER`-size units.

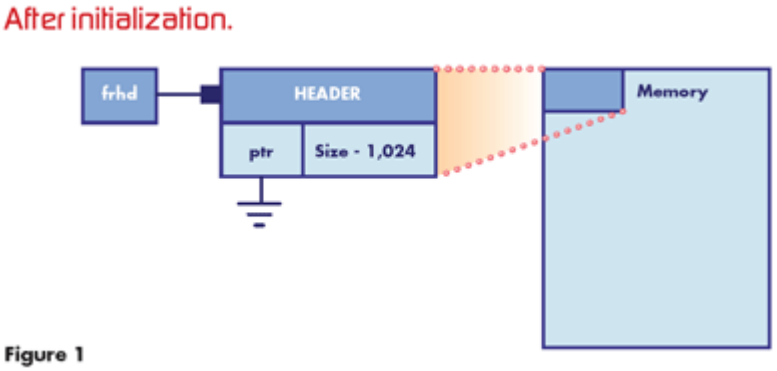

Figure 1

When pointers to a structure are added or subtracted in C, the result is divided by the size of the structure. For example, if `_heapstart` and `_heapend` differ by 4,096 bytes and the size of the structure `HEADER` is four bytes, the result of the pointer arithmetic is 1,024.

The available space can be thought of as an array of header blocks and the calculation as finding the number of entries in the array.

Allocating and freeing memory

`Malloc()` calculates the number of `HEADER`-size units required to satisfy the request. It rounds the result up and adds one unit for the header, which is part of the block allocated. For speed, the calculation could contain a shift instead of a divide.

This function searches the free list for a block large enough to meet our needs. If the block is exactly the right size, we remove it from the linked list and allocate the entire block to the caller. If the block is larger than required, `malloc()` splits it by creating a new header inside the block and decrementing the original block header size by the amount requested. The pointer is incremented by the remaining size, resulting in a pointer to a new header. The requested size is then put into this new header and, as a result, the block splits in two. The upper section (higher in memory) is allocated to the caller, while the lower section remains in the free list (see **Figure 2**). The caller receives a pointer to the block just beyond the header.

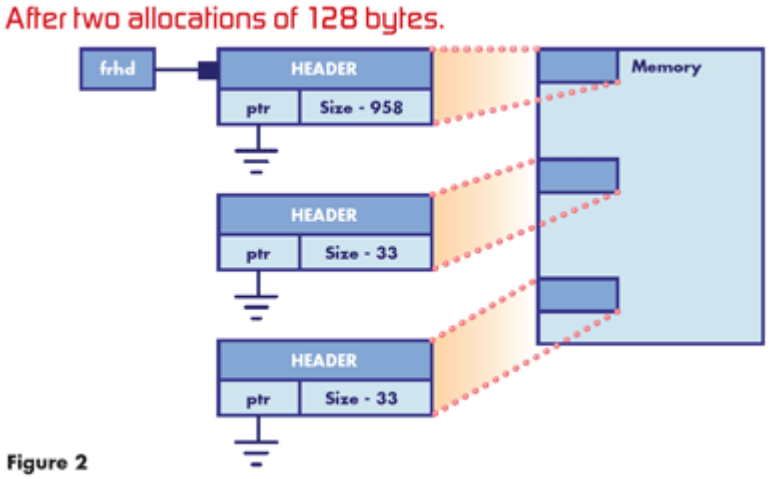

Figure 2

The application uses `free()` to return allocated blocks. Because the pointer returned by `malloc()` points one header unit beyond the actual header for the block, `free()` decrements it by one on entry to point to the original header for the block being returned. The block being freed could be located:

• Lower in memory than the first entry in the free list, in which case it becomes the new free-list head.

• Between entries in the free list.

• Higher in memory than the last entry and therefore linked to the end of the list.

If the returned block is lower in memory than the first entry, it's linked as the new first entry ahead of the previous free-list head. The `free()` function then calculates the address of the byte immediately following the returned block. If this address equals the address of the next free-space entry, the two entries combine to form one larger, contiguous free block.

If the returned block is found to lie between two entries of the free list, we check to see if it's contiguous to the block lower in memory. If so, a larger contiguous block is formed.

Similarly, we check to see if this new, larger block is contiguous to the entry just above it and, if so, make a larger contiguous block. If the entry isn't found, we make a new entry in the free queue for the returned block (see **Figure 3**) and we again attempt to form a block that's contiguous to the entry just above it in the free list.

After a free() on one of the allocated blocks. The returned block wasn't contiguous to the larger block, so the free list shows two free blocks. The pointer (ptr) in the free list header points to the returned block, which is now the last entry to the free list.

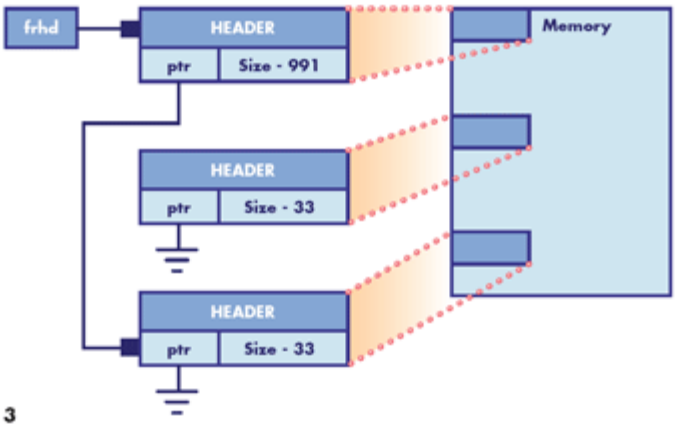

Figure 3

In this way, `malloc()` and `free()` assure that memory doesn't stay fragmented after it's freed. Eventually, if all the allocated blocks are freed, one free-list entry will point to one contiguous block containing all the heap space.

A practical example

Our allocator has proved quite fast and efficient in practice. For example, we used the allocator on a data communications switch based on a Motorola 6809 processor. The system switched data packets to and from an IBM PC and 10 terminals over 9,600-baud RS-232 serial lines. The terminals interfaced to other devices at lower baud rates.

We used the allocator to dynamically allocate and release memory for the data packets. System constraints required that no port be allowed to monopolize the system, so we modified the code presented here (see **Listing 1**) to limit the amount of memory allocated to any one port. For each port, we specified upper and lower threshold for available memory. When a port hit or passed the lower threshold, it was marked busy and received no more packets from the file server until it

freed enough memory to pass the upper threshold value. Because we wrote the allocator, modifying it for such low-level control was fairly easy.

Listing 1 :

```
/*
 * Produced by Programming ARTS
 * 8/18/88
 * Programmers: Les Aldridge, Travis I. Seay
 */

#define NULL (void *)0

typedef struct hdr {
        struct hdr      *ptr;
        unsigned int    size;
} HEADER;

/* Defined in the linker file. _heapstart is the first byte allocated to the heap; _heapend
   is the last. */

extern HEADER _heapstart, _heapend;

extern void    warm_boot(char *str);

static HEADER  *frhd;
static short   memleft;              /* memory left */

void free(char *ap)
{

/* Return memory to free list. Where possible, make contiguous blocks of free memory.
   (Assumes that 0 is not a valid address for allocation. Also, i_alloc() must be called
   prior to using either free() or malloc(); otherwise, the free list will be null.) */

  HEADER *nxt, *prev, *f;
  f = (HEADER *)ap - 1;              /* Point to header of block being returned. */
  memleft += f->size;

    /* frhd is never null unless i_alloc() wasn't called to initialize package. */

   if (frhd > f)
   {

      /* Free-space head is higher up in memory than returnee. */

      nxt = frhd;                    /* old head */
      frhd = f;                      /* new head */
      prev = f + f->size;            /* right after new head */

      if (prev==nxt)                 /* Old and new are contiguous. */
      f->size += nxt->size;
      f->ptr = nxt->ptr;             /* Form one block. */
      }
      else f->ptr = nxt;
      return;
   }
```

```
    /* Otherwise, current free-space head is lower in memory. Walk down free-space list look-
       ing for the block being returned. If the next pointer points past the block, make a new
       entry and link it. If next pointer plus its size points to the block, form one contigu-
       ous block. */

    nxt = frhd;
    for (nxt=frhd; nxt && nxt < f; prev=nxt,nxt=nxt->ptr)
    {
       if (nxt+nxt->size == f)
       {
          nxt->size += f->size;         /* They're contiguous. */
          f = nxt + nxt->size;          /* Form one block. */
          if (f==nxt->ptr)
          {

             /* The new, larger block is contiguous to the next free block, so form a larger
                block.There's no need to continue this checking since if the block following
                this free one were free, the two would already have been combined. */

             nxt->size += f->size;
             nxt->ptr = f->ptr;
          }
             return;
       }
    }

    /* The address of the block being returned is greater than one in the free queue (nxt) or
       the end of the queue was reached. If at end, just link to the end of the queue.
       Therefore, nxt is null or points to a block higher up in memory than the one being
       returned. */

    prev->ptr = f;                      /* link to queue */
    prev = f + f->size;                 /* right after space to free */
    if (prev == nxt)                    /* 'f' and 'nxt' are contiguous. */
    {
        f->size += nxt->size;
        f->ptr = nxt->ptr;              /* Form a larger, contiguous block. */
    }
    else f->ptr = nxt;
    return;
}

char * malloc(int nbytes)               /* bytes to allocate */
{
    HEADER *nxt, *prev;
    int     nunits;
    nunits = (nbytes+sizeof(HEADER)-1) / sizeof(HEADER) + 1;

    /* Change that divide to a shift (for speed) only if the compiler doesn't do it for you,
       you don't require portability, and you know that sizeof(HEADER) is a power of two.
       Allocate the space requested plus space for the header of the block. Search the free-
       space queue for a block that's large enough. If block is larger than needed, break
       into two pieces and allocate the portion higher up in memory. Otherwise, just allocate
       the entire block. */
```

```
    for (prev=NULL,nxt=frhd; nxt; nxt = nxt->ptr)
    {
        if (nxt->size >= nunits)        /* big enough */
        {
          if (nxt->size > nunits)
          {
             nxt->size -= nunits;       /* Allocate tell end. */
             nxt += nxt->size;
             nxt->size = nunits;        /* nxt now == pointer to be alloc'd. */
          }
          else
          {
             if (prev==NULL) frhd = nxt->ptr;
             else prev->ptr = nxt->ptr;
          }
          memleft -= nunits;

          /* Return a pointer past the header to the actual space requested. */

          return((char *)(nxt+1));
        }
    }

    /* This function that explains what catastrophe befell us before resetting the system. */

    warm_boot("Allocation Failed!");
    return(NULL);
}

void i_alloc(void)
{
    frhd = &_heapstart;                 /* Initialize the allocator. */
    frhd->ptr = NULL;
    frhd->size = ((char *)&_heapend -- (char *)&_heapstart) / sizeof(HEADER);
    memleft = frhd->size;               /* initial size in four-byte units */
}
```