

```

/*
 * my_malloc.c
 *
 * This file provides an interface to malloc() and free() which
 *
 * (1) Keeps track of the number of calls to malloc() minus the number
 *     of calls to free(). This allows easy detection of programming
 *     errors which allocate more memory than they free (or vice versa).
 *
 * (2) Aborts the program if malloc() returns NULL. This saves having
 *     to check whether malloc() == NULL at each point where malloc()
 *     is called.
 *
 * All kernel routines use my_malloc() and my_free();
 * no UI routines do so.
 *
 * The UI should call verify_my_malloc_usage() upon exit, to verify
 * that the number of calls to my_malloc() was exactly balanced by
 * the number of calls to my_free().
 *
 * The remainder of this comment deals with a debugging feature, and
 * may be safely ignored until such time as you start freeing memory
 * you haven't allocated, or writing off the ends of arrays.
 *
 * If the constant DEBUG_MALLOC is #defined to be 1 (see below) then
 * my_malloc() and my_free() maintain a linked list
 * of the addresses of the blocks of memory which have been allocated.
 * If some other part of the kernel attempts to free memory which has
 * not been allocated (or free the same memory twice), free() generates
 * an error message and exits. Because this feature is for debugging
 * purposes only, no attempt is made to be efficient. (Obviously a
 * binary tree would be more efficient than a linked list, but you'd have
 * to account for the fact that the memory is most likely allocated
 * in linear order. Reversing the order of the bytes would be simpler
 * than implementing a balanced tree.)
 *
 * Note that the DEBUG_MALLOC feature itself uses memory, but does not
 * record its own usage in the linked list. This is OK. Its purpose
 * is to help debug SnapPea, not malloc().
 *
 * If DEBUG_MALLOC is turned on, my_malloc() tacks four extra bytes on
 * the end of every requested block of memory, and writes in an
 * arbitrary (but well defined) sequence of four characters. When the
 * memory is freed, my_free() checks those four characters to see whether
 * they've been overwritten. This is not a perfect guarantee against
 * writing past the ends of array, but it should detect at least some
 * errors.
 */

#include "kernel.h"
#include <stdlib.h> /* needed for malloc() */
#include <stdio.h> /* needed for sprintf() */

static int net_malloc_calls = 0;

/*
 * The debugging feature is normally off.
 */
#define DEBUG_MALLOC 0

#if DEBUG_MALLOC

#define MAX_BYTES 50000

typedef struct memnode
{
    void *address;
    size_t bytes;
    struct memnode *next;
} MemNode;

static MemNode mem_list = {NULL, 0, NULL};

```

```
static const char    secret_code[5] = "Adam";

static Boolean       message_given = FALSE;

#endif

void *my_malloc(
    size_t  bytes)
{
    void      *ptr;
#ifdef DEBUG_MALLOC
    MemNode  *new_mem_node;
    char      *error_bytes;
    int       i;
#endif

#ifdef DEBUG_MALLOC
    if (message_given == FALSE)
    {
        uAcknowledge("The my_malloc() memory allocator is in debugging mode.");
        message_given = TRUE;
    }
#endif

#ifdef DEBUG_MALLOC
    if (bytes < 0)
    {
        uAcknowledge("A negative number of bytes were requested in my_malloc().");
        exit(3);
    }
    if (bytes > MAX_BYTES)
        uAcknowledge("Too many bytes were requested in my_malloc().");
#endif

    /*
     * Most likely malloc() and free() would correctly handle
     * a request for zero bytes, but why take chances?
     */
    if (bytes == 0)
        bytes = 1;

#ifdef DEBUG_MALLOC
    ptr = malloc(bytes + 4);
#else
    ptr = malloc(bytes);
#endif

    if (ptr == NULL)
        uAbortMemoryFull();

    net_malloc_calls++;

#ifdef DEBUG_MALLOC

    error_bytes = (char *) ptr + bytes;
    for (i = 0; i < 4; i++)
        error_bytes[i] = secret_code[i];

    new_mem_node = (MemNode *) malloc((size_t) sizeof(MemNode));
    if (new_mem_node == NULL)
    {
        uAcknowledge("out of memory");
        exit(4);
    }
    new_mem_node->address    = ptr;
    new_mem_node->bytes      = bytes;
    new_mem_node->next       = mem_list.next;
    mem_list.next = new_mem_node;
#endif

    return ptr;
}
```

```

void my_free(
    void    *ptr)
{
#ifdef DEBUG_MALLOC
    Boolean  old_node_found;
    MemNode  *old_mem_node,
              *prev_mem_node;
    size_t   bytes;
    char     *error_bytes;
    int      i;
#endif

#ifdef DEBUG_MALLOC
    old_node_found = FALSE;
    for (    prev_mem_node = &mem_list, old_mem_node = mem_list.next;
          old_mem_node != NULL;
          old_mem_node = old_mem_node->next, prev_mem_node = prev_mem_node->next)
        if (old_mem_node->address == ptr)
        {
            old_node_found = TRUE;
            bytes = old_mem_node->bytes;
            prev_mem_node->next = old_mem_node->next;
            free(old_mem_node);
            break;
        }
    if (old_node_found == FALSE)
    {
        uAcknowledge("A bad address was passed to my_free().");
        exit(5);
    }

    error_bytes = (char *) ptr + bytes;
    for (i = 0; i < 4; i++)
        if (error_bytes[i] != secret_code[i])
        {
            uAcknowledge("my_free() received a corrupted array.");
            exit(6);
        }
#endif

    free(ptr);

    net_malloc_calls--;
}

int malloc_calls()
{
    return net_malloc_calls;
}

void verify_my_malloc_usage()
{
    char    the_message[256];

    if (net_malloc_calls != 0)
    {
        sprintf(the_message, "Memory allocation error:\rThere were %d %s calls to my_malloc
        ( ) than to my_free().",
            net_malloc_calls > 0 ? net_malloc_calls : - net_malloc_calls,
            net_malloc_calls > 0 ? "more" : "fewer");
        uAcknowledge(the_message);
    }
}

```