GBDK libraries documentation

Michael Hope

GBDK libraries documentation
by Michael Hope

Libraries for GBDKJuly 3rd 1998

This document describes the functions included in the libraries for Pascal Felber's GBDK that are Gameboy specific. Currently included are sections on the malloc lib and the multiple font support routines. Unfortunately I'm still learning how to use SGML. Any help would be appreciated :)

Table of Contents

Chapter 1. sys/malloc.h - malloc, free and related functions

Functions

malloc

```
void *malloc( size_t numbytes )
```

Allocate numbytes of memory from the free memory pool and return a pointer to the base of the newly allocated region.

Description

Note that the memory is not cleared upon allocation.

Parameters

numbytes: The number of bytes to allocate

Returns

On success, returns a pointer to the newly allocated region. On failure returns NULL.

calloc

```
void *calloc( size_t nmem, size_t size )
```

Attempt to allocate space for nmem objects of size size and return a pointer to the allocated region.

Description

calloc is very similar to malloc but it also clears (fills with zero) the memory region before returning.

Parameters

size: size of one object

nmem: Number of objects to allocate space for

Returns

On success, returns a pointer to the newly allocated and cleared region. On failure, returns NULL.

realloc

```
void *realloc( void *current, size_t size )
```

Attempt to re-size a currently allocated region.

Description

realloc is used to resize a currently allocated block without loosing the data contained within. If the current block is larger than the requested block the the trailing data is lost. Note that the end of the block is not cleared if the current block is smaller than the requested one. If current is NULL, then this is equivalent to malloc. If size is zero, then this is equivalent to free.

Parameters

current: Pointer to the currently allocated block.

size: Size of the new block

Returns

On success returns a pointer to the newly allocated block. Note that the new block may be at a different location to the old. On failure or if size is zero, then NULL is returned.

free

```
int free( void *region )
```

Free a previously allocated region.

Description

Attempts to free a region previously allocated by malloc, realloc or calloc. Note that only valid regions can be freed. Note also that this prototype differs from the standard C free as it returns an error code.

Parameters

region: Pointer to a previously allocated region.

Returns

On success, returns zero (0). If the region is already free, returns -1. If the region was never allocated, returns -2.

Data types

NULL

NULL is returned by malloc and others upon failure. Currently defined to be equal to zero.

size_t: UWORD

size_t is used to specify the size of an object in bytes. As the GB has an 8 bit processor with a 16 bit address space, size_t is currently defined as a UWORD. Note that due to the banked nature of most GB programs, it might be changed to UDWORD at a later date.

Implementation

This malloc library is implemented using a simple signally linked list of hunks where a hunk is a a header and section of memory that can be either free or used. There is nothing particularly clever about the allocation algorithms. I'm an engineer as opposed to a computer scientist, so if the code gets the job done then it's close enough. Note that I do not know how well this system will hold up against heavy fragmentation caused by allocating many small blocks and keeping some of them. However, I also cant think of a program that you'd want to run on a GB that would do this.

Functions

malloc_init

```
int malloc_init( void )
```

If the malloc system is currently uninitialised, initialise it.

Description

Checks to see if the header malloc_first is valid by checking its magic number. If it is not, initialise it by marking it as free, setting it to occupy all of the free ram in the area C000h to D000h, setting the region header pointer to NULL and finally setting the magic number.

Parameters

None

Returns

On success, returns zero (0). If the malloc system is already setup, return -1.

malloc_gc

```
void malloc_gc( void )
```

Perform garbage collection on the malloc hunk list by joining consecutive free blocks.

Description

malloc_gc is called by malloc when an attempt to allocate a region fails. malloc_gc attempts to improve the situation by scanning through the malloc hunk list and joining adjacent free blocks into one larger block. In each scan, if two adjacent free blocks are found then they are combined and the scan continued from the first block. At the end of a scan, if any combinations were made then the list is rescanned. I'm not really sure why the rescan is there as in theory all combinations should be made on the first pass.

Parameters

None

Returns

Nothing

Data types

mmalloc_hunk

```
struct smalloc_hunk {
UBYTE magic;
pmmalloc_hunk next;
UWORD size;
int status;
```

```
};
```

mmalloc_hunk is an internal structure used by the malloc library to keep track of allocated and free regions of memory.

Members

magic: A magic number that identifies this as a valid mmalloc_hunk next: Pointer to the next hunk, NULL if this is the last. size: Size in bytes of the hunk. status: Current status of the block that this hunk refers to. One of MALLOC_UNUSED (0), MALLOC_FREE (1) and MALLOC_USED (2). I have no idea why I defined both a MALLOC_UNUSED and a MALLOC_FREE :)

MALLOC_MAGIC: UBYTE

The magic number associated with a valid malloc hunk header. Currently set at a really boring 123. Suggestions for something better will be greatly appreciated.

Notes

The header for a hunk occurs just before the region. Any errant programs that write past their region could overwrite this header and break the linked list. But you get that. Most routines check the magic number while walking the list and abort if a broken header is found.

The amount of memory free after static variables are allocated is determined by using a new linker area called _HEAP, defined in crt0.s. _HEAP occurs after _BSS, so it should occur at the start of free memory. The only data initially in _HEAP is a reference to malloc_heap_start which is used by malloc_init. Note that it is possibly on a real GB for the magic number to occur in a bad place. This should be fixed in crt0.s at the initialisation time. malloc also shares the ram with the stack. Currently the problem of the stack growing down into allocated memory is lessened by allocating from low memory first and by providing a 512 byte buffer (set in malloc_init).

On cartridges with internal memory an extra 8k from A000h to BFFFh is available. Unfortunately free assumes that hunks are consecutive which causes problems. Two solutions are shifting _BSS to start at A000h or defining an extra flag in the header that is set if the next hunk is consecutive to the current one. The second option would also allow paged ram to be used, although it would have to be managed carefully.

Chapter 2. sys/font.h - Support for multiple fonts

Functions

init_font

```
void init_font(void)
```

Initialise the font system by clearing all font handles and releasing all tiles.

Description

Initialises the font system. This routine should be called at the start of the program before any calls to load_font.

Parameters

None.

Returns

Nothing.

load_font

```
FONT_HANDLE load_font( void *font_structure )
```

Attempt to load the font font, returning a FONT_HANDLE on success or NULL on failure.

Description

load_font should be called once for each font that is required. Note that currently there is no unload_font support.

Parameters

font: pointer to the base of a valid font structure

Returns

On success, returns a FONT_HANDLE. On failure, returns NULL (0)

set_font

```
void set_font( FONT_HANDLE font_handle )
```

Set the current font to the previously loaded font font_handle

Description

set_font changes current output font to the one specified by font_handle

Parameters

font_handle: handle from a previous load_font call.

Returns

Nothing.

Bugs

No check is made to see if font_handle is a valid font handle.

mprint_string

```
void mprint_string( char *string )
```

Print string string using the current font.

Description

This is a temporary routine used to test the font library.

Parameters

string: null terminated string to print.

Returns

Nothing.

Data types

FONT_HANDLE: UWORD

Description

A FONT_HANDLE is a 16 bit value returned from load_font and used by set_font. Physically it is a pointer to an entry in the font_table.

font_structure

Description

A font_structure is a container for the data related to a font, including the encoding data and tile (bitmap) image data. Due to the variable length nature of the encoding table and the tile data no default C structure exists.

Format

A font structure is made up of four fields - the font type, the number of tiles used, the encoding table and the tile data.

type: UBYTE

The Font type is a single bit that describes the encoding table and the format of the tile data. The encoding table length is specified by the lower two bits 00: 256 byte encoding table 01: 128 byte encoding table 10 and 11 are reserved. The third bit (0x04) is used to determine if the tile data is compressed. Many tiles do not use shades of grey, and so can be represented in 8 bytes instead of 16. If bit 2 is set, then the tiles are assumed to be 8 bytes long and are expanded to 16 bytes at the load stage.

num_tiles: UBYTE

num_tiles gives the number of tiles present in the tile data and hence the number of tiles required by the font.

encoding: array of UBYTE

The encoding table is an array that maps an ASCII character to the appropriate tile in the tile data. For example, suppose that the letter ’A’ was the tenth tile. Then the 65th (the ASCII code for ’A’) entry in the encoding table would be 10. Any ASCII characters that don’t have a corresponding tile should be mapped to a default tile. Space is recommended tile.

tile_data: array of UBYTE

The final field is the actual tile data. Note that tile numbering starts from zero.