GBDK libraries documentation

Michael Hope

Pascal Felber

GBDK libraries documentation

by Michael Hope and Pascal Felber

GBDK-2.0b13 August 31st 1998

This document is a first try at documenting the Gameboy Developers Kit (GBDK) by Pascal Felber which consists of a C compiler, assembler, linker and libraries. Included are methods of writing efficent eight bit code in C, ways of accessing the GBs hardware, interfacing assembler and C and a list of the functions provided by the libraries.

Table of Contents

Preface

Why

Chapter 1. Using GBDK and maccer

Compiling programs

This section assumes that you have installed GBDK properly and build the libraries as per the instructions on Pascal's homepage.

The program 'lcc' is a front end for the actual compiler, assembler and linker. It works out what you want to do based on command line options and the extensions of the files you give it, computes the order in which the various programs must be called and then executes them in order. Some examples are:

- `lcc -o image.gb source.c` - compile the C source 'source.c', assemble and link it producing the Gameboy image 'image.gb'
- `lcc -o image.gb source.s` - assemble the file 'source.s' and link it producing the Gameboy image 'image.gb'
- `lcc -c -o object1.o source1.c` - compile the C program 'source1.c' and assemble it producing the object file 'object1.o' for later linking.
- `lcc -c -o object2.o source2.s` - assemble the file 'source2.s' producing the object file 'object2.o' for later linking
- `lcc -o image.gb object1.o object2.o` - link the two object files 'object1.o' and 'object2.o' and produce the Gameboy image 'image.gb'
- `lcc -o image.gb source1.c source2.s` - do all sorts of clever stuff by compiling then assembling source1.c, assembling source2.s and then linking them together to produce image.gb.

Arguments to the assembler etc can be passed via lcc using `-Wp..., -Wf..., -Wa...` and `-Wl...` to pass options to the pre-processor, compiler, assembler and linker respectivly. Some common options are:

- `-Wa-l` to generate an assembler listing file.
- `-Wl-m` to generate a linker map file which can then be turned into a no$gmb .sym file for debugging using maptosym.
- `-Wl-gvar=addr` to bind var to address 'addr' at link time.

For example, to compile the example in the memory section and to generate a listing and map file you would use:

```
lcc -Wa-l -Wl-m -Wl-g_snd_stat=0xff26 -o image.gb hardware.c
```

Note the leading underscore that C adds to symbol names.

Unfortunatly maccer, Michael Hope’s macro preprocessor for the assembler has to be run seperatly as lcc dosnt know about it. To turn the assembler file with macros ’source.ms’ into the assembler file ’source.s’, use

```
maccer -o source.s source.ms
```

If the `-o source.s` option isnt specified then maccer writes to stdout. If `source.ms` isnt specified then maccer reads from stdin.

Using Makefiles

One of the most useful development tools is ’make’, a program that automatically keeps your object files and images up to date with your source code. It works by having a set of rules of how to get from one file type to another, which normally involves running a program. For example by typing ’make image.gb’ make will look for a C or assembler file called image.c or image.s, compile it and link it automatically creating the image image.gb. It really comes into its own when you have a project made up of multiple files as it only recompiles those that have changed since the last make.

A copy of GNU make for DOS is available as part of DJGPP. Most Unix systems come with make installed.

A general Makefile for Gameboy projects follows:

```
AS = lcc -c
CC = lcc -Wa-l -Wl-m

BIN = astro.gb
OBJS =  render.o sin_table.o stars.o debug.o fp_long.o sqrt.o floor.o \
        ftou.o sin_cos.o const.o inv_trig.o

all: $(BIN)

%.s: %.ms
     maccer -o $@ $<

$(BIN): $(OBJS)
     $(CC) -o $(BIN) $(OBJS)

clean:
     rm -f $(BIN) $(OBJS) *~
```

The first two lines convince make to use lcc as the assembler and linker instead of the default system ones. The BIN line specifies the name of the image file you want at the end. The OBJS line specifies what object files must be made for BIN to be built. The %.s: %.ms line tells make how to generate .s files for lcc from .ms (macro) files.

For example, suppose that I change the assembler file ’render.ms’. When make is run, it will find that astro.gb depends on render.o which in turn depends on render.ms. It will then use maccer to change the astro.ms file to astro.s, then it will use lcc to assemble astro.s to astro.o which is finally linked with all the other .o files to create astro.gb

Chapter 2. The Gameboy as a Target

8 bits + 6 registers + C = tricky

The Gameboy is not an ideal target for C code due to a combination of being eight bit, having a small register set (pity those who have the 6502 as a target...), no indexed addressing mode and no hardware multiplication or division. The processor being eight bit is the biggest limitation as most modern C assumes that an int is at least 16 bits - see later.

This section is on the methods used to get around or avoid the limitations of the processor which boils down to ways of staying within eight bits.

Size of variables

The size of the various types as at 2.0b13 are:

- `char` eight bit signed
- `unsigned char` eight bit unsigned
- `int` eight bit signed
- `unsigned int` eight bit unsigned
- `long` sixteen bit signed
- `unsigned long` sixteen bit unsigned
- `long long` 32 bit signed
- `unsigned long long` 32 bit unsigned
- `float` 8 bit exponent, 24 bit mantissa
- `pointer` two byte

Please see the section on the float library for more information on the format of floating point numbers. As at 2.0b13 neither float or long long support is complete.

lcc uses 'int' as the default type for most operations including array indexing and constants. To make the code more efficient the ints were made eight bit which unfortunately limits local arrays to less than 128 elements. Note that statically allocated arrays such as image or tile data dont have this limitation.

To make it easier for porting and if/when the size of an int gets changed, it is recommended that the types `BYTE, UBYTE, WORD` and `UWORD` defined in `type.h` are used.

Avoiding Promotion

lcc assumes that any parameters in an expression are signed by default which can cause unnecessary promotion. Promotion is where the compiler believes that the result of an operation will overflow and so it promotes the variable up a size, performs the operation and then demotes it back to the proper size. For example in:

```
UBYTE i, j = 0;
i = j+0x80;
```

The compiler assumes that the argument is signed, and as 0x80 is greater than the biggest eight bit signed number 0x7f it is promoted to sixteen bit. The operation is performed then the result truncated.

This can be solved by explicitly telling the compiler that the argument is unsigned by adding a trailing U. A better version of the above code is:

```
UBYTE i, j = 0;
i = j+0x80U;
```

Changing the order of operations in a function can also stop promotion. ??why??

Using global variables

The local variables of a function are stored on the stack, requiring the compiler to calculate a variables absolute address every time its used. By declaring a variable as global it becomes statically allocated at an address in ram which in most cases makes the code more efficient.

Don't forget that global variables are still bad - but in this case its more efficient to use them.

Globals are especially good for large structures. The easiest way to access data on the stack is with the `lda hl,x(sp)` where x is a signed byte. If the size of the local variables is greater than 127 bytes (the upper limit of a signed byte) then significantly slower code is used.

Other

Preferably use the equality operators `==` and `!=` over the inequality operators `>, <, <=` and `=>`. If the operands are signed or long then the code for inequality significantly more complex then for unsigned bytes which can be done in two operations.

Global variables that are initialised when they declared (for example `int i = 0;`) are put into the _DATA segment by lcc. This means that the variable cant be changed as for the GB the _DATA segment is in ROM. To avoid this, its best to initialise any global variables after declaring them i.e. use

```
int i;

int main(void)
{
    i = 0;
};
```

instead of the above code.

Chapter 3. Accessing hardware

Memory

There are two main ways of accessing the memory of the GB directly. The first which I recommend is using casting, where the second is by declaring an address as external and defining it at link time. Both are best illustrated by example. Suppose that you want to turn on sound by writing 8fh to SND_STAT (FF26h). The code using casts is

```
#define SND_STAT    (UBYTE *)0xFF26U

void sound_on(void)
{
    *SND_STAT = 0x8FU;
}
```

while the code using late linking is

```
extern UBYTE snd_stat;

void sound_on(void)
{
    snd_stat = 0x8FU;
}
```

where snd_stat is defined at link time by adding `-gsnd_stat=0xff26` to the linker arguments.

The most comonly used hardware registers are pre-defined in hardware.h using the late linking method. This code

```
#include <hardware.h>

void sound_on(void)
{
    NR52_REG = 0x8fU; /* ’NR52_REG’ maps to ’_reg_0x26’ or SND_STAT */
}
```

achieves the same as the other two examples.

Interrupts

Interrupts allow execution to jump to a different part of your code as soon as an external event occurs - for example the LCD entering the vertical blank period, serial data arriving or the timer reaching its end count. For an example see `irq.c`

Interrupts in GBDK are handled using the functions `disable_interrupts(), enable_interrupts(), set_interrupts(UBYTE ier)` and the interrupt service routine (ISR) linkers `add_VBL, add_TIM, add_LCD, add_SIO` and `add_JOY` which add interrupt handlers for the vertical blank, timer, LCD, serial and joypad interrupts respectively. The system supports up to eight ISRs per interrupt, executing the first one installed first.

As an example, this code installs an interrupt handler that increases `count` every time the timer runs out.

```
...
UWORD count;

void timer_isr(void)
{
    count++;
}

int setup_isr(void)
{
    disable_interrupts();
    add_TIM(timer_isr);
    enable_interrupts();

    set_interrupts(TIM_IFLAG);
}
...
```

Note that this assumes that the timer is setup elsewhere and the use of the global variable. All registers are pushed before the ISR is called.

As an interrupt can occur at any time, an ISR cannot take any arguments or return anything. Its only way of communicating with the greater program is through the global variable above. Note how interrupts have to be disabled before adding the timer ISR and that the set_interrupts call will disable any other interrupts. To use multiple interrupts, or the relevant IFLAGs together.

ISRs must be kept as small and short as possible and as at 2.0b13 cannot use any `long long`s or floating point variables as the code for them is not re-entrant. It is possible to write a ISR long enough so that the GB spends all of its time servicing interrupts and has no time spare for the main code.

Multiple banks

The GB supports ROMS of up to 1.5MB and up to 32kB of RAM by using a bank switching method. Bank 0 of the ROM is always located in the region 0000h - 3FFFh and cannot be swapped but any other bank can be swapped into the high ROM region between 4000h to 7FFFh. Unfortunately GBDK doesn't support programs bigger than 32kB at the moment which is partly due to lcc assuming a flat address space. However you can manually access the other banks using the `switch_rom_bank(UBYTE)` and `switch_ram_bank` functions. See `banks.c` for an example.

The ROM and RAM bank that the code should exist in is specified at compile time using the `-Wf-box` and `-Wf-bax` compiler options where x is the bank number between 1 and 31. Note that this means that you cant switch banks within one code file but multiple files can exist in the same bank. If neither the -bo or -ba options are given then the default _CODE and _BSS segments are used. Dont forget that local variables are allocated on the stack inside _BSS.

For example, suppose the code:

```
int silly_fun( int a )
{
    printf("%i times %i is ", a, a, a*a+1 );
    return 0;
}
```

was compiled with the `-Wf-bo1` compiler option then the code would be stored in segment `_CODE_1`. To call this function from your main routine you would use:

```
int main(void)
{
    switch_rom_bank( 1 );  /* Select bank 1 with segment _CODE_1 */
silly_fun(5);      /* Prints "5 times 5 is 26" */

return 0;
}
```

Note that you obviousally cannot do a `switch_rom_bank` call from inside any segment but _CODE as otherwise you'd switch yourself out. Note also that all global routines like `printf` must fit within the limited 16k of bank 0.

When linking all the object files together the number of banks used should be specified with the `-Wl-yox` and `-Wl-yax` flags and the MCB type with the `-Wl-ytx` flag. The current supported values for x in `-Wl-ytx` are:

```
0 : ROM ONLY
1 : ROM+MBC1
2 : ROM+MBC1+RAM
3 : ROM+MBC1+RAM+BATTERY
5 : ROM+MBC2
6 : ROM+MBC2+BATTERY
```

Functions in RAM

It is possible to execute functions in RAM or HIRAM by first copying them there using `memcpy()` or `himemcpy()` and then calling them in similar ways to accessing memory directly. See `ram_fn.c` for an example.

Chapter 4. Assembly language

When C isn't fast enough

For many applications C is fast enough but in intensive functions you are sometimes better to write them in assembler. This section deals with interfacing your core C program with fast assembly sub routines.

Calling convention

lcc in common with almost all C compilers prepends a '_' to any function names. For example the function `printf(...)` begins at the label `_printf::`. Note that all functions are declared global.

The parameters to a function are pushed in right to left order with no aligning - so a byte takes up a byte on the stack instead of the more natural word. So for example the function `int store_byte( UWORD addr, UBYTE byte)` would push 'byte' onto the stack first then addr using a total of three bytes. As the return address is also pushed, the stack would contain:

- At SP+0 - the return address
- At SP+2 - addr
- At SP+4 - byte

Note that the arguments that are pushed first are highest in the stack due to how the GB's stack grows downwards.

The function returns in DE. I'm not sure how a FP number is returned.

Variables and registers

C normally expects registers to be preserved across a function call. However in this case as DE is used as the return value and HL is used for anything, only BC needs to be preserved.

Getting at C variables is slightly tricky due to how local variables are allocated on the stack. However you shouldn't be using the local variables of a calling function in any case. Global variables can be accessed by name by adding an underscore.

Segments

The use of segments for code, data and variables is more noticeable in assembler. lcc defines a number of default segments - `_CODE, _DATA` and `_BSS` for storing code, static data and variables in respectively. Two extra segments `_HEADER` and `_HEAP` exist for the GB header and malloc heap respectively. The order these segments are linked together is determined by `crt0.s` and is currently _CODE then _DATA in ROM and _BSS then _HEAP in RAM. _HEAP is placed after _BSS so that all spare memory is available for the malloc routines. To place code in other than the first two banks, use the segments _CODE_x where x is the 16kB bank number.

As the _BSS segment occurs outside the ROM area you can only use `.ds` to reserve space in it.

While you don't have to use the _CODE and _DATA distinctions in assembler it is recommended for consistancys sake.

Chapter 5. Libraries

Text IO - console.h and output.s

Blah

Floating point support - fp_long.s

Number format

The number encoding used in GBDK was chosen make fp operations easier to implement. Unfortunatly it doesnt conform to any standards but it does help speed the routines. The high byte contains the sign and exponent, while the lower three bytes contain the mantissa.

The most significant bit of the exponent is the sign bit which is set if the number is negative. The lower seven bits are the exponent biased around 0x40. The mantissa is normalised and includes the normally implicit one in the most significant bit. Note that the mantissa for a negative number is not in twos complement. A zero is represented by either by both the mantissa and exponent being zero.

Some examples:

- `41 800000` - sign = 0 (positive), exponent equals 2^(0x41-0x40) = 2^1 = 2, mantissa = 0.800000 = 0.5 so the number is +1 * 2 * 0.5 or ’1’

- `C4 A00000` - sign = 1 (negative), exponent equals 2^(0x44-0x40) = 2^(4) = 16, mantissa = 0.A00000 = 0.625 so the number is -1 * 16 * 0.625 or ’-10’

Notes

The core floating point library functions fadd, fdiv, fsub and fmul were originally based on the algorithims in HI-TECH Software ’s free CP/M C compiler. However the code has been almost completly re-written and optimised and now only bears a passing resemblance to HI-TECH’s code.

The left hand argument to a function is stored in HLDE and the right hand on the top of the stack. Most functions use static scratch registers in RAM which unfortunatly means that they arnt re-enterant. As almost all of the functions in this section take floats as parameters and return floats, the format is different to the following sections.

Functions

.fadd32

Description

Perform a floating point addition on HLDE and the floating point number on the stack, returining the result in HLDE with the stack number removed.

Method

The two operands are recovered and the stack unjunked. The sign is removed from both operands and the magnitudes compared. If the difference in magnitude is greater than 24 bits then the greater number is returned. If not, the smaller number is shifted right and its magnitude increased until both have the same magnitude. If an operand was originally negative, its magnitude is negated. The magnitudes are added and the new mantissa computed.

.fsub32

Description

Subtract the floating point number on the stack from the value in HLDE, returning the result in HLDE

Method

The sign bit on the stack operand is toggled which negates the right operand. Execution then falls through to .fadd32

.fmul32

Description

Performs a floating point multiplication on the number in HLDE and the one on the stack. Returns the result in HLDE with the stack operand removed.

Method

The product is stored in HLBC, the right hand operand in DE.ft1.ft0 and the left hand operand in .fw. First the product is zeroed then a multiplication is done on the mantissas of the two operands. The multiply is a standard (right shift - add if carry) but uses a few tricks to keep the result within 32 bits instead of the maximum 48. HLDE is then shifted down until H is zero. The number of shifts is added to the exponents of the two operands to give the resultant exponent. Finally the sign bit is computed by a XOR of the signs of the operands.

.fdiv32

Description

Divide the floating point number in HLDE by the value on the stack. Return witht the result in HLDE and the stack unjunked.

Method

The divisor is stored in HLBC, the dividend in DE.fw1.fw0 and the quotent in .q. The mantissa of the dividend is divided by the mantissa of the divisor. This uses the standard compare - subtract if less than - multiply by two method. The quotient is then copied into HLDE and rotated right until H is zero. The exponent is calculated by adding the number of shifts to the exponent of the left operand and subtracting the exponent of the right operand. The sign bit is calculated using an XOR of the sign of the operands.

Full screen graphics - drawing.h

Blah

Graphics/Tile functions

GBDK graphic primitives allow the host to manage tiles, background, window, and sprites. Some of these primitives depend on the contents of the LCDC register, which specify the addresses in the VRAM at which graphics data are located. Therefore, the LCDC register must be set prior to using these primitives.

By default, the background and window TDT is located at addresses 0x8800-0x97FF, the sprite TDT at addresses 0x8000-0x8FFF, the BTM at addresses 0x9800-0x9BFF, and the WTM at addresses 0x9C00-0x9FFF. It is possible to modify these addresses (except the sprite TDT) using the LCDC register.

Functions

set_xxx_data / get_xxx_data

```
void
set_bkg_data(UBYTE first_tile,
             UBYTE nb_tiles,
             unsigned char *data);

void
set_win_data(UBYTE first_tile,
             UBYTE nb_tiles,
             unsigned char *data);

void
set_sprite_data(UBYTE first_tile,
                UBYTE nb_tiles,
                unsigned char *data);

void
set_data(unsigned char *vram_addr,
         unsigned char *data,
         UWORD len);

void
get_bkg_data(UBYTE first_tile,
             UBYTE nb_tiles,
             unsigned char *data);
```

```
void
get_win_data(UBYTE first_tile,
             UBYTE nb_tiles,
             unsigned char *data);

void
get_sprite_data(UBYTE first_tile,
                UBYTE nb_tiles,
                unsigned char *data);

void
get_data(unsigned char *data,
         unsigned char *vram_addr,
         UWORD len);
```

Description

The `set_bkg_data()`, `set_win_data()`, and `set_sprite_data()` functions copy tile data to the background, window, or sprite TDT in VRAM. The `get_bkg_data()`, `get_win_data()`, and `get_sprite_data()` functions copy data from VRAM to an address specified by the user. The address of data in VRAM depends of the contents of the LCDC register.

The `set_data()` function copies data to any address in VRAM. The `get_data()` function copies data from any address in VRAM.

The operations that read data can be used to save the current state of the screen (for instance for displaying a dialog box, and restoring the screen afterwards).

Paramenters

`first_tile`: number of the first tile in VRAM (between 0 and 255).

`nb_tiles`: number of tiles to copy (0 for copying 256 tiles).

`data`: pointer to the user tile data.

`vram_addr`: address in VRAM.

`len`: number of bytes to copy.

Returns

Nothing.

set_xxx_tiles / get_xxx_tiles

```
void
set_bkg_tiles(UBYTE x,
              UBYTE y,
              UBYTE w,
              UBYTE h,
              unsigned char *tiles);

void
set_win_tiles(UBYTE x,
              UBYTE y,
              UBYTE w,
              UBYTE h,
              unsigned char *tiles);

void
set_tiles(UBYTE x,
          UBYTE y,
          UBYTE w,
          UBYTE h,
          unsigned char *vram_addr,
          unsigned char *tiles);

set_sprite_tile(UBYTE nb,
                UBYTE tile);

void
get_bkg_tiles(UBYTE x,
              UBYTE y,
              UBYTE w,
              UBYTE h,
              unsigned char *tiles);

void
get_win_tiles(UBYTE x,
              UBYTE y,
              UBYTE w,
              UBYTE h,
              unsigned char *tiles);

void
get_tiles(UBYTE x,
          UBYTE y,
          UBYTE w,
          UBYTE h,
          unsigned char *tiles,
          unsigned char *vram_addr);

UBYTE get_sprite_tile(UBYTE nb);
```

Description

The `set_bkg_tiles()` and `set_win_tiles()` functions copy a rectangular area of background or window tile numbers to the BTM in VRAM. The `get_bkg_tiles()` and `get_win_tiles()` functions copy data from VRAM to an address specified by the user. The address of tile numbers in VRAM depends of the contents of the LCDC register.

The `set_tiles()` function copies a rectangular area of tile numbers to any address in VRAM (this address will be generally 0x8000 or 0x8800). The `get_tiles()` function copies data from VRAM to an address specified by the user.

The `set_sprite_tiles()` function set the tile number of a specific sprite. The `get_sprite_tiles()` function returns the tile number of a sprite. These functions do not directly access the VRAM.

The operations that read data can be used to save the current state of the screen (for instance for displaying a dialog box, and restoring the screen afterwards).

Parameters

`x`: horizontal coordinate of the top-left corner of the rectangular area in VRAM.

`y`: vertical coordinate of the top-left corner of the rectangular area in VRAM.

`w`: width of the the rectangular area in VRAM.

`h`: height of the the rectangular area in VRAM.

`tiles`: pointer to the user tile numbers. The size of the tile numbers should be `w*h`.

`vram_addr`: address in VRAM.

`nb`: index of the sprite to access (between 0 and 39).

`tile`: tile number of the sprite.

Returns

The `get_sprite_tiles()` function returns the tile number of the sprite.

move_xxx

```
void
move_bkg(UBYTE x,
         UBYTE y);

void
move_win(UBYTE x,
         UBYTE y);

void
move_sprite(UBYTE nb,
            UBYTE x,
            UBYTE y);
```

Description

The `move_bkg()` and `move_win()` functions modify the position of the background or window display. These functions expect the absolute value of the new position. Note that these functions should preferably not be used during screen redraw. To avoid unwanted visual distortions, it is better to call these functions upon VBL.

The `move_sprite()` function modifies the position of a specific sprite. This function expects the absolute value of the new position. It does not directly access the VRAM.

Parameters

`x`: horizontal coordinate of the new display position. `y`: vertical coordinate of the new display position. `nb`: index of the sprite to access (between 0 and 39).

Returns

Nothing.

scroll_xxx

```
void
scroll_bkg(BYTE x,
           BYTE y);

void
scroll_win(BYTE x,
           BYTE y);

void
scroll_sprite(UBYTE nb,
              BYTE x,
              BYTE y);
```

Description

The `scroll_bkg()` and `scroll_win()` functions modify the position of the background or window display. These functions expect a relative displacement relative to the current position. Note that these functions should preferably not be used during screen redraw. To avoid unwanted visual distortions, it is better to call these functions upon VBL. The `scroll_sprite()` function modifies the position of a specific sprite. This function expects a relative displacement relative to the current position. It does not directly access the VRAM.

Parameters

`x`: horizontal displacement relative to the current display position.

`y`: vertical displacement relative to the current display position.

`nb`: index of the sprite to access (between 0 and 39).

Returns

Nothing.

set_sprite_prop / get_sprite_prop

```
/* Sprite properties bits */

#define S_PALETTE    0x10U
#define S_FLIPX      0x20U
#define S_FLIPY      0x40U
#define S_PRIORITY   0x80U

void
set_sprite_prop(UBYTE nb,
                UBYTE prop);
```

```
UBYTE
get_sprite_prop(UBYTE nb);
```

Description

The `set_sprite_prop()` function modifies the attributed of a specific sprite. The `get_sprite_prop()` function returns the attributed of a sprite. These functions do not directly access the VRAM.

Paramters

`nb`: index of the sprite to access (between 0 and 39).

`prop`: new value of the sprite attributes.

Returns

The `get_sprite_prop()` returns the attributes of the sprite.

Joypad - gb.h

Functions

joypad / waitpad / waitpadup

```
/* Joypad bits */

#define J_START      0x80U
#define J_SELECT     0x40U
#define J_B          0x20U
#define J_A          0x10U
```

```
#define J_DOWN       0x08U
#define J_UP         0x04U
#define J_LEFT       0x02U
#define J_RIGHT      0x01U

UBYTE
joypad(void);

UBYTE
waitpad(UBYTE mask);

void
waitpadup(void);
```

Description

The `joypad()` function reads the current status of the joypad. It does not block the current thread of control. The joypad status returned by the function uses one bit for each button, according to the definitions given above.

The `waitpad()` function waits for any button from a given mask to be pushed. The `waitpadup()` function waits for all buttons to be released.

Parameters

`mask`: set of buttons to wait for.

Returns

The `joypad()` and \`waitpadup()` functions return the joypad status.

Hardware registers - hardware.h

Blah

Types - types.h

Blah

malloc, free and related functions - stdlib.h

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.

Support for multiple fonts - font.h

Functions

font_init

```
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 font_load.

Parameters

None.

Returns

Nothing.

font_load

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

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

Description

font_load should be called once for each font that is required. Note that currently there is no way to unload a font except by calling font_init().

Parameters

font: pointer to the base of a valid font structure

Returns

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

font_set

```
void font_set( 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 font_load call.

Returns

Nothing.

Bugs

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

Data types

FONT_HANDLE: UWORD

Description

A FONT_HANDLE is a 16 bit value returned from font_load and used by font_set. 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: No encoding table - the font is 256 tiles long. 11 is 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.

Backwards compatibilty

The original text routines were replaced by the font routines in GBDK 2.0.18. This section describes the differences and compatibility between the old and new text routines.

The old font routines stored a bitmap of the entire IBM-ASCII character set in the Gaemboy tiles, such that the ASCII value of the letter mapped to the appropriate tile - for example, the bitmap of an ’A’ (ASCII 65) was stored in tile 65. The new system allows multiple fonts to fit into the same 256 tiles by removing unused characters - for example, if one font only needs the letters ’A-Z’ and numbers ’0-9’ (36 tiles), then the remaining 220 tiles can be used for other fonts or game tiles.

If no fonts are loaded before the first call to putchar() then the original IBM-ASCII font is automatically loaded. This should make the font routines backwards compatible with older code and standard C. Note that the text input routines currently only work with the IBM-ASCII font.

One side effect is that the IBM-ASCII font is always linked in occuping 2.3k of ROM, even if it is not used. If the extra ROM is required, add the line:

```
UBYTE font_ibm_fixed[];
or
_font_ibm_fixed::
```

into the project. Note that in 2.0.18, the graphics mode text routines only use the IBM-ASCII font.

Fonts can also be used to store game tiles. To make the tiles allocated more predictable, load the game tile font first, as the first tile in the font will be allocated to tile 01h (tile 00h is always clear). Game tile fonts should use ’no encoding’ for their encoding table type to save memory. Two features allow the standard C routines to also be used for game tiles - namely M_NO_INTERP and M_NO_SCROLL. The first stops the routines from interpreting carridge returns (\n), the second stops the screen from scrolling. Use

```
mode(get_mode() | M_NO_INTERP | M_NO_SCROLL);
```

to turn both on. See font_draw in Michael Hopes ’metro’ package for an example.