C API Reference

Welcome to the quick reference guide for the Brick Engine, v5.2. This document covers the C API.

Basic Engine Operation

Starting and stopping the Brick Engine

Just about everything you'll do with the Brick Engine requires that you initialize it first. After you're done, too, you may want to free up any resources held by the engine.

Function: init_brick()

```
void init_brick();
```

Prepares the engine internal data structures and activates the hardware.

Function: quit_brick()

```
void quit_brick();
```

Shuts down the engine, closes the graphics display, halts all sounds, and frees up any resources used by the engine.

Graphics

The graphics display may be started and stopped at any time (for example, to change from windowed to full-screen mode, or to change the display resolution), without interfering with the rest of the engine. These are the routines to activate and deactivate the graphics display.

Function: graphics_open()

```
int graphics_open(int mode, int w, int h, int fs, int zf);
```

Opens the graphics display. The **mode** value is one of the following: **GRAPHICS_SDL** (standard SDL display) or **GRAPHICS_ACCEL** (OpenGL-accelerated blit to screen). The graphics display size is given in **w** and **h**.

Note: There is a compile-time maximum width and height, which defaults to 640x480.

Full-screen mode is controlled by the boolean **fs** flag. In the accelerated graphics output mode, the zoom factor **zf** determines the amount by which the display is scaled. *Note that a zoom factor of either 0 or 1 has no effect on the output display.*

Returns 0 on success, or an error code on failure.

Function: graphics_close()

```
void graphics_close();
```

Closes the active graphics display.

Function: graphics_info()

```
int graphics_info(int *w, int *h);
```

Returns the active graphics mode, and stores the current display resolution in **w** and **h**.

Audio

Audio output may be activated or deactivated at any time. These routines let you do that.

Function: audio_open()

```
int audio_open(int mode, char *file);
```

Opens the audio output. **mode** is one of the following:

AUDIO_SPEAKER (speaker output).

Function: audio_close()

```
void audio_close();
```

Closes the active audio output.

Input

These are functions used to configure and read user input from keyboard, joystick, and mouse. The joystick is the primary type of input device recognized by the Brick Engine, and the engine supports up to eight joysticks at a time. You don't need any actual joysticks to play, because the engine will map all keyboard input onto one of eight "virtual joysticks": each keypress is turned into an axis, hat, or button motion on one of the eight joysticks, and this keyboard-input mapping can be altered at any time. If you've got actual hardware joysticks plugged in, the inputs provided by these joysticks (axes, hats, and buttons) can be read directly.

Function: io_fetch()

```
int io_fetch(int input, input *io);
```

Retrieves the status of the given input into **io**.

There are eight inputs from which movement and actions can be requested, numbered 0 through 7. Keys can be assigned to any action (axis, hat, or button) on any input. If joysticks are plugged in, they are assigned to the inputs starting at **0**.

Axes have a range from -127 to 127. Key presses will set the axis to the ends of this range, while an analog joystick input may return any value within this range. Up to eight axes are read, either from joystick or from the assigned keys.

Usually, the first two axes will represent horizontal and vertical motion, either from the keyboard or the joystick. Joysticks with more than one analog stick will use additional axes to represent the movement on the additional sticks. By default, the arrow keys are mapped onto axes 0 and 1 on input 0.

Hats represent the four-way directional inputs present on some joysticks. Hat values range from -1 to 1 and are returned as pairs of horizontal and vertical results. Note that a joypad- style controller with just one directional input probably returns its results as a pair of axes rather than a hat. By default, the keys **wasd** are mapped onto hat 0 of input 0.

Button presses return either 0 or 1. By default, the keys **left ctrl**, **left alt**, **z**, and **x** are assigned to button presses 0 through 3 on input 0.

There are eight axes, four hats, and twenty buttons available on each input. If a physical joystick has more axes, hats, or buttons than this, the excess will be ignored.

The keys **space**, **tab**, **enter** or **return** (as **select**), **pause**, and **escape** are always included in the results.

Function: io_mouse()

```
int io_mouse(int input, mouse *);
```

Reads the mouse motions on the specified mouse input.

Function: io_grab()

```
void io_grab(int flag);
```

Grabs the keyboard and mouse input, preventing interference from the window manager or operating system. *Warning: Be sure that your game is thoroughly debugged before using this routine!! If your game has an infinite loop and the input grab is enabled, there will be no way for the player to exit gracefully.*

Function: io_has_quit()

```
int io_has_quit();
```

Returns whether or not a quit signal has been received by the game, e.g. clicking the close button of the game window.

Function: io_wait()

```
int io_wait(int delay);
```

Waits until all input buffers are cleared, and then waits until any activity on any of the inputs is received. If the application-quit button is pressed, this immediately returns ERR. The **delay** value indicates how many times each second the inputs will be checked for activity, so that the processor use can be kept low.

Function: io_assign()

```
void io_assign(int input, int type, ... );
void io_assign(int input, int type = IO_AXIS, int axis_no,
    int dir, int keycode);
void io_assign(int input, int type = IO_HAT, int hat_no,
    int dir, int keycode);
void io_assign(int input, int type = IO_BUTTON,
    int button_no, int keycode);
```

Assigns a keycode to the action on the given input. The combination of axis, hat, button number, and direction specifies which joystick action will have a keyboard-mapping assigned. The type must be one of **IO_AXIS**, **IO_HAT**, **IO_BUTTON**. If the action type is **IO_AXIS**, then only the directions **IO_LEFT** and **IO_RIGHT** are permitted. If the action type is **IO_HAT**, then the directions **IO_LEFT**, **IO_UP**, **IO_DOWN**, and **IO_RIGHT** are permitted. Note that if the action type is **IO_BUTTON**, then the direction argument is omitted.

Function: io_read_key()

```
int io_read_key()
```

Halts until a single keypress can be read and a keycode returned.

This is not useful for general-purpose input, but is intended only to let the programmer determine keycodes interactively, for use with **io_assign()**.

The Graphics Subsystem

There are two parts of the Brick Engine graphics subsystem that deal with system-wide graphics configuration: render settings and font handling. Everything else (for example, sprites, strings, and tile-based maps) is addressed later on.

Rendering

These are the routines used to set system-wide rendering options.

Function: render_set_bg_fill()

```
void render_set_bg_fill(int fill);
```

Enable or disable the solid-color background fill.

Function: render_set_bg_color()

```
void render_set_bg_color(char r, char g, char b);
```

Sets the background fill color to the given RGB values.

Function: render_set_overdraw()

```
void render_set_overdraw(int w, int h);
```

Sets the amount of overdraw applied to the internal render canvas. This does not affect the displayed canvas size. Some sprite frame types (e.g. the pixel-repositioning frame) may depend on graphics data being drawn outside the screen borders for proper composition of the display, and the overdraw instructs the renderer to generate this extra data.

Function: render_display()

```
void render_display();
```

Renders and displays the current frame.

Function: render_to_disk()

```
int render_to_disk(char *file);
```

Renders the current frame to the so-named file.

Fonts

The Brick Engine has a simple and lightweight font renderer built in. Fonts are loaded into the engine as bitmaps, and characters are fixed-width. One font, named **default**, is built into the Brick Engine, and additional fonts can be loaded at any time.

Function: font_add()

```
void font_add(char *name, int w, int h, unsigned char *data,
    color *key);
```

Adds a new font named **name**, with the bitmap data stored in **data**. If **key** is not null, it will be used as a chroma key for the font display. The **data** buffer is three-bytes-per-pixel RGB data, consisting of an image of all characters in the font arranged in order. The dimensions of each character are given in **w** and **h**, and the font image must be 128 characters wide.

Function: font_info()

```
int font_info(char *name, int *w, int *h);
```

Retrieves the character dimensions of the named font. Returns 0 on success, or ERR if the font does not exist.

Function: font_from_disk()

```
void font_from_disk(char *name, char *file, color *key);
```

Adds a new font named **name** from the compressed image file named **file**. If **key** is not null, it will be used as a chroma key for the font display. The font image is assumed to be 128 characters wide. *Note: This routine is only available if the Brick Engine was built with SDL_image support.*

Function: font_from_buffer()

```
void font_from_buffer(char *name, int len,
    unsigned char *data, color *key);
```

Adds a new font named **name** from the **data** buffer which contain a compressed image file. If **key** is not null, it will be used as a chroma key for the font display. The font image is assumed to be 128 characters wide. *Note: This routine is only available if the Brick Engine was built with SDL_image support.*

The Audio Subsystem

Every game needs sound! The Brick Engine provides functionality that makes it possible to handle both song and sound playback with ease.

Sound playback

These routines let you load sounds from different sources (from a known

sound file format stored on disk or in a memory buffer, or as raw sound data), play them as needed, and stop them or adjust the sound volume/panning in mid-play.

Function: sound_load_from_disk()

```
sound *sound_load_from_disk(char *filename);
```

Loads a sound from a file on disk. The list of supported file formats can be found in the documentation for SDL_mixer.

Function: sound_load_from_buffer()

```
sound *sound_load_from_buffer(int length, unsigned char *data);
```

Loads a sound from the **data** buffer. The buffer length is given in **length**. The list of supported file formats can be found in the documentation for SDL_mixer.

Function: sound_load_raw()

```
sound *sound_load_raw(int length, unsigned char *data);
```

Creates a sound from the given data buffer. The data must match the audio format set at compile time. By default, the audio format is 8-bit unsigned data.

Function: sound_play()

```
int sound_play(sound *sound, int volume);
```

Plays the given sound. The volume may range from 0 to 128. This returns the ID of the audio channel that is playing the sound, so that the sound can be stopped or have its volume or panning values adjusted.

Function: sound_halt()

```
void sound_halt(int id);
```

Stops the sound playing on the given channel. If **id** is -1, halt all sounds.

Function: sound_adjust_vol()

```
void sound_adjust_vol(int id, int volume);
```

Sets the volume of the sound playing on the given channel ID. The volume may range from 0 to 128. If **id** is -1, sets the volume for all currently-playing sounds.

Function: sound_adjust_pan()

```
void sound_adjust_an(int id, int panning);
```

Sets the panning of the sound playing on the given channel ID. The panning value ranges from 0 (left speaker only) to 254 (right speaker only), and is balanced at 127. If **id** is -1, sets the panning for all currently-playing sounds.

Song playback

The song playback routines will let you start, stop, and otherwise control the background music for your game. *Note that because the Brick Engine relies on SDL_mixer for its music playback support, so the list of supported file formats depends on how SDL_Mixer has built for your system.*

Function: song_play_from_disk()

```
void song_play_from_disk(char *filename,
    int fade_in_delay);
```

Loads and plays the named song from disk, with a fade-in delay given in milliseconds.

Function: song_play_from_buffer()

```
void song_play_from_buffer(int length, unsigned char *buffer,
    int fade_in_delay);
```

Loads and plays the named song from a memory buffer of length **length**, with a fade-in delay given in milliseconds.

Function: song_pause()

```
void song_pause();
```

Pauses the currently-playing song.

Function: song_resume()

```
void song_resume();
```

Resumes the currently-paused song.

Function: song_stop()

```
void song_stop(int fade_out_delay);
```

Stops the currently-playing song with fade-out delay given in milliseconds.

Function: song_set_position()

```
void song_set_position(int pos);
```

Sets the position of the currently-playing song.

Function: song_adjust_vol()

```
void song_adjust_vol(int volume);
```

Sets the music playback volume. The volume can range from 0 to 128.

Items and Lists

These are the bread-and-butter routines in the Brick Engine, the API calls you'll use again and again in developing your games, so it's worth it to familiarize yourself with these.

Lists

Lists are everywhere in computing, and the Brick Engine is no different. The list implementation built into the engine is a pretty simple doubly-linked list, and you'll most often use it in two places: adding sprites and strings to the sprite- and string- display lists, and getting back lists of sprites from the introspection routines. (You may also find some more sophisticated uses for the Brick Engine lists, though, e.g. setting up some intricate collision-

detection schemes where you'll test certain groups of enemy sprites against certain player projectiles.) These routines are what you'll use to create and manipulate Brick Engine lists.

Function: list_create()

```
list *list_create();
```

Creates a new list.

Function: list_empty()

```
void list_empty(list *list);
```

Empties the given list. Note that this does not delete any of the items in the list.

Function: list_delete()

```
void list_delete(list *list);
```

Deletes the given list. Note that this does not delete any of the items in the list.

Function: list_add()

```
void list_add(list *list, void *item);
```

Adds the given item to the end of the list.

Function: list_prepend()

```
void list_prepend(list *list, void *item);
```

Adds the given item to the start of the list.

Function: list_shift()

```
void *list_shift(list *list);
```

Removes the first item from the head of the list and returns it.

Function: list_pop()

```
void *list_pop(list *list);
```

Removes the last item from the list and returns it.

Function: list_remove()

```
void list_remove(list *list, void *item, int direction);
```

Removes the given item from the list. The **direction** flag can be set to LIST_HEAD, LIST_TAIL, or LIST_ALL. If set to LIST_HEAD or LIST_TAIL, this routine removes the first matching entry it finds from the beginning or end of the list. If **LIST_ALL** is given, then all matching items are removed from the list.

Function: list_length()

```
int list_length(list *list);
```

Returns a count of the number of items in the given list.

Function: list_find()

```
int list_find(list *list, void *item);
```

Determines whether the given item is in the list.

Function: list_sort()

```
void list_sort(list *list, int(*)(void *, void *));
```

Sorts the list using the provided comparison function.

List Iterator Macros

There are now several macros provided in the Brick Engine header file to provide basic list iteration capability.

Function: iterator_start()

```
iterator_start(iterator i, list l);
```

Initializes an iterator struct to point to the head of the given list. You must call this before using any of the other iterator macros.

Function: iterator_next()

```
iterator_next(iterator i);
```

Advances the iterator one step forward. If there are no more list entries, then this does nothing.

Function: iterator_data()

```
iterator_data(iterator i);
```

Retrieves the item held in the current list position.

Function: iterator_ct()

```
iterator_ct(iterator i);
```

Gives the number of iterations through the list that have been made in this iteration.

Frames

A frame is a container for any sort of graphics data in the Brick Engine. Whether you are working with simple pixel data, color keyed data (i.e. one color treated as transparent), or one of the various visual effects (e.g. convolution kernel, desaturation, and so on), the data is always stored in a frame. Some of the sprite and tile routines, such as sprite_add_frame_data(), handle the process of creating and loading the frame for you, but others, such as sprite_add_subframe(), require that you create and prepare the frame before passing it to the routine.

Frames can be sliced into subframes (good for cutting up sprite sheets), and it's also possible to convert RGB frames into almost any other frame type, e.g. to create a desaturated version of a sprite, for use in some effect.

Last, if you've built the Brick Engine with the SDL_Image dependency, you can load and unpack a variety of image types directly into frames, either

from disk or from a memory buffer.

Function: frame_create()

```
frame *frame_create(int mode, int width, int height, void *data,
    void *auxiliary);
```

Creates a new graphics frame using the given frame data. **mode** is one of: FRAME_NONE (no display data), FRAME_RGB (RGB data with an optional chroma-key), FRAME_DISPL (pixel-displacement frame), FRAME_CONVO (convolution kernel), FRAME_BR (brightness-adjusting frame), FRAME_CT (contrast-adjusting frame), FRAME_LT (luminance-adjusting frame), FRAME_SAT (saturation-adjusting frame), FRAME_LUT (RGB lookup-table frame). The width and height are given in **width** and **height**.

A frame type of FRAME_NONE takes no display data and produces no output. The **data** and **auxiliary** arguments are ignored. This isn't very useful, except for situations where some unusual collision detection is needed.

If the frame type is FRAME_RGB, the data is a buffer of RGB pixels. An optional chroma key can be passed in as a **color** in **auxiliary**. The frame data will be drawn onto the render canvas.

If the type is FRAME_DISPL, the pixel data is an array of 16-bit (big-endian) X/Y coordinate pairs. Each coordinate pair represents an offset from the pixel in the frame to the location in the underlying image data from which to retrieve the given pixel. For example, pixel data consisting of only zeros has no effect, while pixel data containing all pairs of -1, -1 will create a frame that offsets the underlying image data up and to the left by one pixel.

If the type is FRAME_CONVO, then the **data** array is a pixel mask, in which each non-zero pixel applies the specified convolution kernel to the underlying image data. The convolution kernel definition is passed into **auxiliary** as a **convolution**.

If the type is FRAME_BR, the data is a buffer of RGB pixels. A pixel component value of 64 is neutral and doesn't alter the image brightness. Values less than 64 darken the image, and values greater than 64 brighten the image.

If the type is FRAME_CT, the data is a buffer of unsigned char values which adjust the contrast of the underlying image. A pixel component value of 64 is neutral and leaves the image contrast unaltered. Values less than 64 decrease the image contrast to a neutral grey, and values greater than

64 increase the contrast of the image.

If the type is FRAME_LT, the data is a buffer of RGB pixels. A pixel component value of 128 is neutral and doesn't alter image lightness. Values less than 128 darken the image toward black, and values greater than 128 lighten the image toward white.

If the type is FRAME_SAT, the data is a buffer of unsigned char values which adjust the saturation of the underlying image. A pixel value of 128 leaves the image unchanged. A value of 64 desaturates the image, and values between 64 and 128 give varying degrees of desaturation. Values less than 64 invert the hue of the image data. Values greater than 128 increase the saturation.

If the type is FRAME_LUT, the data is a buffer of unsigned chars which act as a pixel mask. Each non-zero pixel causes the underlying image data to be replaced according to that pixel's value in the lookup table. The lookup table is passed into **auxiliary** as a 256-element array of **color** values.

Function: frame_info()

```
int frame_info(frame *frame, int *mode, int *w, int *h);
```

Retrieves the frame type and dimensions. Returns 0 on success, or ERR if the frame does not exist.

Function: frame_copy()

```
frame *frame_copy(frame *frame);
```

Makes a copy of the given frame.

Function: frame_delete()

```
void frame_delete(frame *frame);
```

Deletes the given frame.

Function: frame_set_mask()

```
void frame_set_mask(frame *frame, unsigned char *data);
```

Sets the pixel mask for the given frame. This can be useful if you plan to use one frame both as a tile and as a sprite.

Function: frame_set_mask_from()

```
void frame_set_mask_from(frame *frame, frame *source);
```

Sets the pixel mask for the given frame one of two ways, depending on the source frame. If the source frame has a color key set, then the pixel mask is generated by non-transparent pixels. If the source frame has no transparency set, then each pixel is desaturated and any pixels darker than middle gray are treated as solid. This routine only accepts RGB-type frames.

Function: frame_slice()

```
frame *frame_slice(frame *frame, int x, int y, int width, int height);
```

Copies out a section out of the given RGB frame and returns it as a new frame.

Function: frame_convert()

```
frame *frame_convert(frame *frame, int mode, void *auxiliary);
```

Converts an RGB frame to almost any other frame type. This routine modifies the frame in-place, so you should make a copy if you plan to use the original again.

Function: frame_from_disk()

```
frame *frame_from_disk(char *file, color *key);
```

Loads and decompresses the given image file into an RGB frame. *Note: This routine is only available if the Brick Engine was built with SDL_image support.*

Function: frame_from_buffer()

```
frame *frame_from_buffer(int len, unsigned char *data, color *);
```

Loads and decompresses an RGB frame from an image file stored in the given data buffer. *Note: This routine is only available if the Brick Engine was built with SDL_image support.*

Layers

Layers are the basic building block of graphics programming with the Brick Engine. A layer consists of a sprite list, a tile- based map, and a string list. Any of these items can be omitted, and if all three are omitted, the layer is ignored. Each layer also has a camera position, which determines what part of the layer (i.e. the layer's sprites and map..more on strings in a minute) is visible. Strings are a little different, in that they're always rendered exactly where they're placed, and do not move when the layer camera is moved. The layers are rendered in order of creation, and they can be swapped with one another.

Each layer can also have a viewport set, which determines the region of the screen where the layer will actually be drawn. (This allows for easy split-screen gaming, e.g. create a layer, set its viewport to the left half of the screen, then copy it into a new layer and set that layer's viewport to the right side of the screen. Each layer's camera can then be focused on different players.)

Layers have two attributes which are worth mentioning: visibility and sorting. Layer visibility doesn't really need further explanation. The visibility attribute can be set at any time. Sorting, however, is more complicated. Some games may require that sprites are rendered in a certain order, e.g. a sprite that passes in front of another and then behind it, and sorting is a way to achieve that. When layer sorting is enabled, all sprites in that layer are sorted by their z-hint attribute before rendering, and are then rendered in order.

Function: layer_count()

```
int layer_count();
```

Returns the current number of layers.

Function: layer_add()

```
int layer_add();
```

Adds a new layer and returns its ID. When a layer is added, a sprite list, map, and string list are automatically created.

Function: layer_reorder()

```
void layer_reorder(int first, int second);
```

Swaps the first layer with the second.

Function: layer_remove()

```
void layer_remove(int id);
```

Removes the specified layer

Function: layer_copy()

```
int layer_copy(int id);
```

Makes a copy of the specified layer, assigning its properties (sprite list, map, etc) to the new layer.

Function: layer_get_sprite_list()

```
list *layer_get_sprite_list(int id);
```

Returns the sprite list for the given layer.

Function: layer_get_map()

```
map *layer_get_map(int id);
```

Returns the map for the given layer.

Function: layer_get_string_list()

```
list *layer_get_string_list(int id);
```

Returns the string list for the given layer.

Function: layer_get_visible()

```
int layer_get_visible(int id);
```

Returns a boolean value for the layer visibility setting.

Function: layer_get_sorting()

```
int layer_get_sorting(int id);
```

Returns a boolean value for sprite z-hint sorting on the given layer.

Function: layer_set_sprite_list()

```
void layer_set_sprite_list(int id, list *list);
```

Sets the sprite list for the given layer.

Function: layer_set_map()

```
void layer_set_map(int id, map *map);
```

Sets the map for the given layer.

Function: layer_set_string_list()

```
void layer_set_string_list(int id, list *list);
```

Sets the string list for the given layer.

Function: layer_set_visible()

```
void layer_set_visible(int id, int mode);
```

Sets the layer visibility setting to the boolean value specified in **mode**.

Function: layer_set_sorting()

```
void layer_set_sorting(int id, int mode);
```

Sets the sprite z-hint sorting value setting on the given layer.

Function: layer_get_camera()

```
int layer_get_camera(int id, int *x, int *y);
```

Retrieves the camera position on the specified layer. Returns 0 on success, or ERR if the layer ID is invalid.

Function: layer_set_camera()

```
void layer_set_camera(int id, int x, int y);
```

Sets the camera position on the specified layer.

Function: layer_adjust_camera()

```
void layer_adjust_camera(int id, int x, int y);
```

Adjusts the camera position on the specified layer.

Function: layer_get_view()

```
int layer_get_view(int id, box *);
```

Retreives the viewport on the specified layer. Returns 0 on success, or ERR if the layer ID is invalid.

Function: layer_set_view()

```
void layer_set_view(int id, box *);
```

Sets the viewport on the specified layer.

Maps

The tile-based map is an important part of much of 2D game programming, and the Brick Engine has some useful map functionality built in. Every display layer gets one map, and it's always optional whether or not to use the map for any given layer.

A map consists of two things: a set of tiles, and an array of map data. So, to get started with the tile-based maps, you'll first create one or more tiles using the tile-handling commands. You'll then add them to the map's tile index, a fixed-length array of tiles (there's a compile-time limit that determines the size of this array, by default limited to 4096 tiles), and set the tile size as well as the overall map dimensions. Last, you'll set the map data, either one element at a time or all at once. Each entry in the map data array is a number that corresponds to the tile index containing the tile you want to appear at that position on the map.

So, if you have a map that with a width of 3 and a height of 2, the map data array will consist of six numbers. The first three numbers indicate which tiles will show on the first row of the map, and the second three numbers

indicate which tiles show on the second row of the map.

If your map data has tile indices that haven't had any tiles set, then nothing will be rendered (e.g. a hole will appear, and whatever was underneath will be visible) at that point of the map.

Function: map_create()

```
map *map_create();
```

Creates a new map.

Function: map_empty()

```
void map_empty(map *map);
```

Empties a map and resets all map attributes, but does not delete the map itself.

Function: map_delete()

```
void map_delete(map *map);
```

Deletes a map.

Function: map_get_size()

```
int map_get_size(map *map, int *w, int *h);
```

Stores the size of the map into **w** and **h**. Returns 0 on success, or ERR if the map does not exist.

Function: map_get_tile_size()

```
int map_get_tile_size(map *map, int *tw, int *th);
```

Stores the map's tile size into **tw** and **th**. Returns 0 on success, or ERR if the map does not exist.

Function: map_get_tile()

```
int map_get_tile(map *map, int index, tile **tile);
```

Stores the tile pointer for the given map index into **tile**. Returns 0 on success, or ERR if the map does not exist.

Function: map_set_size()

```
void map_set_size(map *map, int w, int h);
```

Sets the overall dimensions for the given map.

Function: map_set_tile_size()

```
void map_set_tile_size(map *map, int tw, int th);
```

Sets the tile size for the given map.

Function: map_set_tile()

```
void map_set_tile(map *map, int index, tile *tile);
```

Loads the given tile into the map's tile index.

Function: map_set_data()

```
void map_set_data(map *map, short *data);
```

Sets the map data for the given map. **w** and **h** are the dimensions of the map. The map data is of 16-bit words, each word containing the index of the tile to display.

Function: map_set_single()

```
void map_set_single(map *map, int x, int y, short data);
```

Sets a single element in the map data. **x** and **y** are the tile coordinates to be set. The tile is a 16-bit word containing the index of the tile.

Function: map_animate_tiles()

```
void map_animate_tiles(map *map);
```

Animate all tiles on the given map.

Function: map_reset_tiles()

```
void map_reset_tiles(map *map);
```

Reset all tile animation on the given map.

Tiles

These routines allow you to create the tiles (and set the properties of same) that you can then load into your maps.

Function: tile_create()

```
tile *tile_create();
```

Creates a new tile.

Function: tile_delete()

```
void tile_delete(tile *tile);
```

Deletes a tile and frees all of its image data.

Function: tile_get_collides()

```
int tile_get_collides(tile *tile, int *mode);
```

Retrieves the collision mode for the specified tile. Returns 0 on success, or ERR if the tile does not exist.

Function: tile_set_collides()

```
void tile_set_collides(tile *tile, int mode);
```

Sets the collision mode for the specified tile. The **mode** must be one of: COLLISION_OFF (sprite does not collide), COLLISION_BOX (collision testing by bounding box), or COLLISION_PIXEL (collision testing by pixel-mask).

Function: tile_get_anim_type()

```
int tile_get_anim_type(tile *tile, int *type);
```

Retrieves the animation type for the specified tile. Returns 0 on success, or ERR if the tile does not exist.

Function: tile_set_anim_type()

```
void tile_set_anim_type(tile *tile, int type);
```

Sets the animation type for the specified tile. Valid animation types are: ANIMATE_OFF (do not animate), ANIMATE_FWD (forward looping animation), ANIMATE_REV (reverse looping animation), and ANIMATE_PP (ping-pong animation).

Function: tile_add_frame()

```
int tile_add_frame(tile *tile, frame *frame);
```

Adds the given frame to the tile. Please note that this does not make a copy of the frame, so you will likely want to make a copy of the frame before passing it to this routine (e.g. if you use that particular frame anywhere else).

Function: tile_add_frame_data()

```
int tile_add_frame_data(tile *tile, int mode, int width,
    int height, void *data, void *aux);
```

Loads the given graphics data into the tile. The documentation for **frame_create()** has a detailed description of the arguments, as this is essentially a wrapper for that routine.

Function: tile_set_pixel_mask()

```
void tile_set_pixel_mask(tile *tile, int index,
    unsigned char *data);
```

Sets a pixel-accurate collision mask for the specified frame of the tile. Any non-zero value in the **data** buffer counts as an active pixel.

Function: tile_set_pixel_mask_from()

```
void tile_set_pixel_mask_from(tile *tile, int index, frame *source);
```

Sets a pixel-accurate collision mask for the specified frame of the tile from a source frame. If the source frame has a color key set, then the opaque pixels represent the collidable portions of the pixel mask. If the source frame does not have a color key, then each pixel is desaturated and any pixel lighter than neutral gray (r, g, b = 128) counts as an active pixel.

Function: tile_animate()

```
void tile_animate(tile *tile);
```

Animates the specified tile.

Function: tile_reset()

```
void tile_reset(tile *tile);
```

Resets the tile animation for the specified tile.

Sprites

Sprites are movable graphical elements that comprise the basic building blocks of most games you'll make with the Brick Engine. They're similar to the idea of hardware sprites that you'll find in consoles and old computers, but with some very useful improvements.

Each sprite can have an unlimited number of graphics frames added to it for animation, and each frame can have subframes that are rendered in series to aid in composition of sprites and effects. The effects are the other unique thing about Brick Engine sprites. A given sprite frame (or subframe) isn't just a block of RGB data, but can be one of a variety of color-manipulation effects, such as brightness or saturation/desaturation effects, or pixel-manipulation effects, such as user-defined convolution kernels. These effects allow for a wide variety of engaging visuals, like shadows, real-time reflective mirrors, rippling water effects, heat-blurring of rocket jets, and so on.

Sprites can also have two kinds of collision-detection enabled, bounding box collision detection (very fast) and pixel-accurate collision detection (not as fast, but as accurate as you specify).

Brick Engine sprites offer two more seldom-used features that come in handy for specific situations, z-hinting and motion control programs. Z-hinting allows sprites to be drawn in a certain order, e.g. if a sprite passes alternately in front of and behind a level object. Z-hinting must enabled at

for a given layer, and the z-hint values are otherwise ignored. Motion control programs are little scripts attached to sprites that allow for some autonomous behavior, and have their own section in this document, which you ought to consult to learn more.

Function: sprite_create()

```
sprite *sprite_create();
```

Creates a new sprite.

Function: sprite_copy()

```
sprite *sprite_copy(sprite *sprite);
```

Makes a copy of the given sprite.

Function: sprite_delete()

```
void sprite_delete(sprite *sprite);
```

Deletes the given sprite and frees all of the frame data.

Function: sprite_set_frame()

```
void sprite_set_frame(sprite *sprite, int frame);
```

Selects a sprite frame for display.

Function: sprite_get_frame()

```
int sprite_get_frame(sprite *sprite, int *frame);
```

Stores the frame that is currently selected in **frame**. Returns 0 on success, or ERR if given an invalid sprite.

Function: sprite_set_z_hint()

```
void sprite_get_z_hint(sprite *sprite, int z_hint);
```

Sets the sprite's z-hint. When layer sorting is enabled, then sprites are sorted by z-hint before being drawn. When layer sorting is not enabled, the sprite's z-hint has no effect.

Function: sprite_get_z_hint()

```
int sprite_get_z_hint(sprite *sprite, int *z_hint);
```

Stores the sprite's z-hint into **z_hint**. Returns 0 on success, or ERR if given an invalid sprite.

Function: sprite_set_collides()

```
void sprite_set_collides(sprite *sprite, int mode);
```

Sets the sprite's collision mode. The **mode** must be one of: COLLISION_OFF (sprite does not collide), COLLISION_BOX (collision testing by bounding box), or COLLISION_PIXEL (collision testing by pixel-accurate mask).

Function: sprite_get_collides()

```
int sprite_get_collides(sprite *sprite, int *mode);
```

Stores the sprite's collision mode into **mode**. Returns 0 on success, or ERR if given an invalid sprite.

Function: sprite_set_bounding_box()

```
void sprite_set_bounding_box(sprite *sprite, int frame, box *box);
```

Sets the bounding box for the specified frame of the sprite.

Function: sprite_set_pixel_mask()

```
void sprite_set_pixel_mask(sprite *sprite, int frame, unsigned char *data);
```

Sets the pixel-accurate collision mask for the specified frame of the sprite. Any non-zero value in the **data** buffer counts as an active pixel.

Function: sprite_set_pixel_mask_from()

```
void sprite_set_pixel_mask_from(sprite *sprite, int frame, frame *source);
```

Sets the pixel-accurate collision mask for the specified frame of the sprite from a source frame. If the source frame has a color key set, then the

opaque pixels in the frame represent the collidable portions of the pixel mask. If the source frame does not have a color key, then the source frame is desaturated and any pixel lighter than neutral gray (r, g, b = 128) counts as an active, collidable pixel.

Function: sprite_set_position()

```
void sprite_set_position(sprite *sprite, int x, int y);
```

Sets the position of the sprite.

Function: sprite_get_position()

```
int sprite_get_position(sprite *sprite, int *x, int *y);
```

Stores the position of the sprite into **x** and **y**. Returns 0 on success, or ERR if given an invalid sprite.

Function: sprite_set_velocity()

```
void sprite_set_velocity(sprite *sprite, int x, int y);
```

Sets the velocity of the sprite. Note that this doesn't actually move the sprite or cause the sprite to be moved. This is only used in setting up your proposed sprite motions for collision detection, e.g. so that you can test how far your sprite may move before it hits a wall or another sprite.

Function: sprite_get_velocity()

```
int sprite_get_velocity(sprite *sprite, int *x, int *y);
```

Stores the velocity of the sprite into **x** and **y**. Returns 0 on success, or ERR if given an invalid sprite.

Function: sprite_add_frame()

```
int sprite_add_frame(sprite *sprite, frame *frame);
```

Adds the given frame to the sprite. Please note that this does not make a copy of the frame, so be aware that you will probably want to make a copy of your graphics frame before passing it to this routine.

Function: sprite_add_frame_data()

```
int sprite_add_frame_data(sprite *sprite, int mode, int w, int h,
    void *data, void *auxiliary);
```

Loads a graphics frame into the given sprite. The documentation for **frame_create()** has a detailed description of the arguments. This is essentially a wrapper for that routine.

Function: sprite_add_subframe()

```
int sprite_add_subframe(sprite *sprite, int index, frame *frame);
```

Adds the frame to the sprite as a subframe on the given frame index. Subframes are rendered in reverse order, i.e. the last- added subframe is rendered first. This allows for easy compositing of sprite frames together (e.g. a shadow, a lighting effect, etc) into a single sprite. Please note that this does not make a copy of the frame, however, so you'll probably want to make a copy of the frame before passing it into this routine.

Function: sprite_load_program()

```
int sprite_load_program(sprite *sprite, char *program);
```

This routine loads the given motion control program into the sprite. Please see the section on Motion Control Programs for a detailed description of how to write these programs.

Strings

Strings are what you'll use to display blocks of text onscreen, such as in character dialogue or as part of a heads-up display. These are the routines used to create and manipulate strings. After creating and configuring your text strings, you'll need to add them to a layer's string list in order for them to be displayed. *Note that strings are positioned absolutely on the display canvas and do not move when the layer camera moves.*

Function: string_create()

```
string *string_create();
```

Creates a new string.

Function: string_delete()

```
void string_delete(string *string);
```

Deletes the given string.

Function: string_set_font()

```
void string_set_font(string *string, char *font);
```

Sets the font for the given string. If an unknown font name is assigned to a string, the string will not be displayed.

Function: string_set_position()

```
void string_set_position(string *string, int x, int y);
```

Sets the position for the given string.

Function: string_set_text()

```
void string_set_text(string *string, char *text);
```

Sets the text contents of the given string.

The Motion Control Program System

The Language

Motion control programs are very short programs, written in a custom language, that give sprites some simple autonomy, i.e. without having to run callbacks in the main game loop. They're especially useful when the programmer is using a scripting language but still desires to animate great numbers of sprites., because they can eliminate the need to fire potentially-heavy script callbacks for sprite motion.

Motion control programs can also be used to generate simple particle systems, environmental effects, and the like. Note that motion control programs aren't intended to be a generic replacement for sprite movement. For more detailed information, please see the in-depth guide at the

[[motion control programs]] page.

The language consists of the following instructions. Instructions and their arguments appear one per line. Whitespace and empty lines are ignored.

```
set var, var/immediate  - store the right-side value in the left-side
                          named var
add var, var/immediate  - add the right-side value to the left-side
                          named var
stc var, var/immediate  - stochastic alter the left-side var with a
                          range of -(var/imm)..var/imm
trk var, id             - copy over the left-side named var contents
                          from another sprite
avg var, id             - average the left-side named var contents with
                          those of another sprite
beq var, var/immediate  - break (immediately exit the program) if equal
bne var, var/immediate  - break if not equal
blt var, var/immediate  - break if less than
bgt var, var/immediate  - break if greater than
bmp id                  - break if there is a collision with the given
                          map
bnm id                  - break if there is a not a collision with the
                          given map
bst var/immediate       - stochastic break, i.e. exit if random value
                          between zero and imm is zero
copy id                 - make a copy of the sprite and replace the
                          current sprite with the copy for the remain-
                          der of the program
ladd id                 - add the sprite to the given list
lrem id                 - remove the sprite from the given list
del                     - delete the sprite
xchgp ptr               - exchange the sprite's motion control program
                          with another sprite's program
sound id                - play the sound
eoc                     - end of code
```

The **var** is a named variable, one of the following: **xpos**, **ypos**, **xvel**, **yvel**, **frame**, **tick**. **xpos** and **ypos** refer to the sprite's position, and **xvel** and **yvel** refer to the sprite's velocity. Immediate values are integers, and **id** values specify a list, sprite, map, or sound. Argument order matches Intel-style assembly language syntax, i.e. instructions that set or change a variable have the destination given first (*set xpos, 4* can be read as *xpos = 4*)

Every sprite also has its own internal tick counter, and this increments every time the sprite's motion-control program is run.

Running motion control programs

These routines execute the motion control programs for a sprite or for a list of sprites.

Function: motion_exec_single()

```
int motion_exec_single(sprite *sprite);
```

Executes the motion control program for the given sprite. Returns 0 on success, or an error code on failure.

Function: motion_exec_list()

```
int motion_exec_list(list *list);
```

Executes the motion control program for every sprite in the given list. Returns 0 on success, or an error code if any motion control program fails to execute.

Introspection and Collision Detection

Inspection

It's often the case that you'll need to have a sprite query its surroundings or check to see if anything else is nearby. If you wanted to have a sprite avoid a patch of water, for example, you could use the introspection routines to read the nearby tiles and have the sprite govern itself accordingly. These introspection routines allow you to do that, along with a few other methods of examining the environment.

Function: inspect_adjacent_tiles()

```
void inspect_adjacent_tiles(map *map, sprite *spr, int dir,
    map_fragment *res);
```

Returns a buffer of tiles adjacent to the sprite on the specified map. The direction must be one of **INSPECT_NW**, **INSPECT_N**, **INSPECT_NE**, **INSPECT_E**, **INSPECT_SE**, **INSPECT_S**, **INSPECT_SW**, **INSPECT_W**. If the sprite is using bounding box collision, the buffer of tiles is determined by the edge of the bounding box. If pixel- accurate collision is enabled, the bounding edges of the pixel mask are used.

Function: inspect_obscured_tiles()

```
void inspect_obscured_tiles(map *map, sprite *spr, map_fragment *res);
```

Returns a buffer of tiles obscured by the sprite on the specified map. If the sprite is using bounding box collision, the tiles are determined by the edge of the bounding box. If pixel-accurate collision is enabled, the bounding edges of the pixel mask are used.

Function: inspect_line_of_sight()

```
int inspect_line_of_sight(map *map, sprite *spr, int xofs, int yofs,
    int dist, sprite *target);
```

Performs a line-of-sight test to determine visibility from the originating sprite to the target sprite within the given map. The **xofs** and **yofs** offsets determine the point, relative to the sprite's upper left corner, from which the visibility test is performed. These offsets can be negative, performing the visibility test from a point outside the originating sprite's frame. The **dist** value is the maximum range for the test.

The originating sprite does not need to have collision detection enabled, but the target sprite must have collision detection enabled. If the target sprite has bounding-box collision enabled, the four corners of the bounding box are checked for visibility from the originating sprite. If the target sprite has pixel-accurate collision enabled, the visibility test is performed on the four corners of the pixel mask's bounding edges.

Function: inspect_in_frame()

```
list *inspect_in_frame(list *list, box *range);
```

Returns a list of sprites that fall within the given rectangle.

Function: inspect_near_point()

```
sprite *inspect_near_point(list *list, int x, int y, int distance);
```

Returns a list of sprites near the given point.

Collisions

These routines detect collisions among sprites, and between sprites and maps.

Function: collision_with_map()

```
void collision_with_map(sprite *sprite, map *map, int slip,
    map_collision *res);
```

Checks the given sprite for collision against the given map. The optional slip argument indicates that, when a collision occurs, the sprite will be adjusted by the given number of one-pixel increments to continue motion. *Note: The slip factor allows sprites to travel near the corners of maps without stopping on all single-pixel collisions.*

The result is stored **res**. The **res.mode** value will be **COLLISION_ATSTART** to indicate that the sprite and map were colliding before motion began, **COLLISION_NEVER** to indicate that no collision has occurred, or **COLLISION_INMOTION** to indicate that collision occurred during motion. The pixel distance before the sprite hits the map is stored in **res.stop** and the distance that the sprite may travel after being adjusted by the given **slip** value is stored in **res.go**.

Function: collision_with_sprites()

```
int collision_with_sprites(sprite *sprite, list *sprite_list,
    int limit, sprite_collision res[]);
```

Tests the given sprite for collisions with all collidable members of the given sprite list. Only **limit** collisions will be returned, and **res** must be large enough to hold this number of collisions. The **res.mode** value will be **COLLISION_ATSTART** to indicate that the sprite and map were colliding before motion began, **COLLISION_NEVER** to indicate that no collision has occurred, or **COLLISION_INMOTION** to indicate that collision occurred during motion. The pixel distance before the sprite hits the target is stored in **res.dist** and the direction in which the collision occurred is stored in **res.hit**.

Utilities

Timing

A routine useful for setting up a basic delay loop and holding a steady framerate.

Function: delay()

```
int delay(int fps);
```

If the time between calls to **delay()** is less than necessary (i.e. the game would otherwise run too fast) to maintain the given **fps** rate, then the routine will delay until enough time has passed. if the game is running too slowly to maintain the requested **fps** rate, **delay()** will return the number of frames that must be skipped to maintain speed.

Scheduled events

The Brick Engine includes a simple event scheduler. Events are functions that take a single void * argument and return nothing. They run in their own thread, and can be schedule to run once, several times, or to be repeated indefinitely. They can also be paused, halted, or temporarily skipped.

Function: event_add()

```
int event_add(int delay, int count, event ev, void *data);
```

Schedules an event to run after **delay** milliseconds. The **count** determines how many times the event is run. If **count** is negative, the event will repeat indefinitely. **ev** is a pointer to the function to run. When **ev** is called, **data** is passed to it. The **data** argument can, of course, be null. The event ID is returned, so that messages can be passed to the event, e.g. to pause or cancel its execution.

Function: event_message()

```
void event_message(int id, int msg);
```

Sends a message to the given event. If the message is EVENT_GO, the event will run as normal. If the message is EVENT_STOP, the event will be cancelled. If the message is EVENT_PAUSE, event execution is paused until the event is resumed (with EVENT_GO) or halted (with EVENT_STOP). If the message is EVENT_SKIP1, the event will not execute the next time it's scheduled to, but will execute each subsequent time and its execution count will be decremented as though it ran (e.g. if it's scheduled to run 10 times, then after being sent EVENT_SKIP1 once, it will run 9 times total).