KATSU
DEVELOPER MANUAL

Version Beta 1.0

Table of Contents

1 Introduction

Katsu is a light cross-platform C library for 2D video game programming. It includes an API for 2D rendering, a simple input system, and sound output. Tile renderers of old school computers and video game consoles are the basis for Katsu's 2D rendering capabilities. The kind of images that Katsu can produce outperforms most of these systems; this is to allow programmers some freedom when designing games. With this in mind, some features present in old systems go against modern GPU hardware and graphics APIs, which is why they are not present (like some per horizontal line effects).

1.1 Initial steps

Firstly include **katsu/kt.h** to all source files that use the library. To initialize Katsu you call `kt_Init()`, this will initialize all of the subsystems that it uses (Video, Joypad, Graphics and Audio), if an error occurs then the return value will be non-zero. After Katsu is initialized an *operating system window* (OSW) will be created and shown on screen. To end all systems, the user can call `kt_Exit()` where a status code can be passed (you can freely define the meaning of each code). After Katsu is initialized an *operating system window* (OSW) will be created and shown on screen.

1.1.1 Main application loop

A normal Katsu application should do the following things in order: read input, update the state of the application, draw to the screen. It is recommended that the application loops forever until the end user or the application decides to exit, this is called the *main application loop*. To read the input use the `kt_Poll()` function (this will also handle events received by the OSW). To draw to the screen use the `kt_Draw()` function. The following code snippet is a simple example of this:

Code 1.1

```c
#include <katsu/kt.h>
int main()
{
    /* Initialize Katsu */
    if (kt_Init()) {
            return 0;
    }
    /* Initialize application state */
    /* Main application loop */
    while(1) {
            kt_Poll(); /* Handle and read inputs */
            /* Update application state */
            kt_Draw(); /* Draw to the screen */
    }
}
```

1.1.2 Custom defined exit function

Since an infinite loop is used, a close event of the OSW will exit the application (same as calling `kt_Exit()`). It is common to ask if it is fine for the application to exit or do other background work before exiting (saving the application state to an external file, for example). You can use `kt_ExitFuncSet()` to pass a user defined function that Katsu will call before exiting the application. This function will receive the status code passed to `kt_Exit()` and returns a code that can cancel `kt_Exit()` from running. This behavior is shown in the example below:

Code 1.2

```
#include <katsu/kt.h>
u32 askToClose(u32 status)
{
    /* Ask user for confirmation on progam exit via a loop */
    if (/* Wants application to exit*/) {
            /* Save application state */
            return KT_EXIT;
    } else { /* Wants to remain */
            return KT_EXIT_CANCEL;
    }
}

int main()
{
    /* Initialize Katsu and application state */
    kt_ExitFuncSet(askUserClose);
    /* Main program loop */
}
```

1.2 Predifined primitive types

Katsu has a number of predefined primitive types that are included in the **katsu/kt.h** header (more specifically, the **katsu/types.h** header). The type names use the following nomenclature:

[V][F][B]

- **F** : This specifies the *format* of the primitive, the available options are: **u** for unsigned integers, **s** for signed integers, **f** for floating point.
- **B** : This specifies the *number of bits* that the type occupies, available options are: **8**/**16**/**32**/**64** for integers and **32**/**64** for floating point.
- **V** (optional) : if a **v** is at the start of a type then the type will be volatile (used for informing the compiler not to optimize out memory accesses for asynchronous events or stop it from caching data). This applies to all previous types.

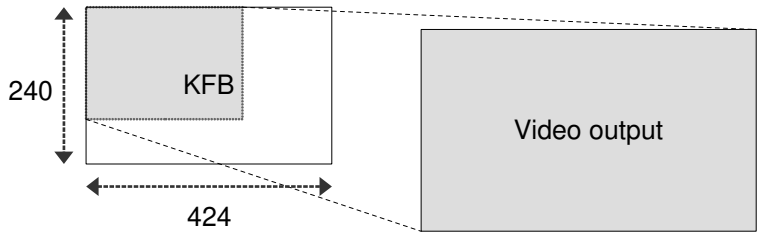
For example, the type **u32** refers to an unsigned 32 bit value, while **f64** refers to a floating-point 64 bit value (or a double precision float). Most common commercial hardware support these primitives so there should be nothing to worry about, these types use the **stdint.h** standard header so some will refer to the same type in hardware, refer to the header documentation to see it's limitations.

2 Video

As mentioned, Katsu presents an OSW when initialized. The Video subsystem can control some aspects of the OSW and how the drawn image is presented inside its frame. One can programatically change the OSWs *frame* with preset sizes by using `kt_VideoFrameSet()`, for example, passing `KT_VIDEO_FRAME_2X` shows the frame at exactly 2 times the size, when `KT_VIDEO_FRAME_FULLSCREEN` is passed, the OSW will cover the television/monitor screen (to exit full screen mode the user can press the *Esc* key on the keyboard). Finally, the frame's title can be set with `kt_VideoTitleSet()`.

2.1 Defining the video output

Standard Katsu will always draw a 424x240 pixel image called the *Katsu frame buffer* (KFB). By default, the whole image will be shown on the OSW, with `kt_VideoOutputSet()` you can crop the KFB to only show a certain width and height in pixels (the minimum values for both are 32 pixels), this is known as the *video output*. This way, you can easily change between different video resolutions.



2.2 Fill modes

Most likely the video output will be smaller than the frame of the OSW. With `kt_VideoFillModeSet()` you can specify the way the video output fills the OSW. The following figure shows how all the preset options act:

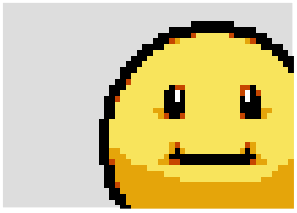
KFB

KT_VIDEO_FILL_SCALE

KT_VIDEO_FILL_STRECH

KT_VIDEO_FILL_INTSCALE

KT_VIDEO_FILL_OUT_SCALE



KT_VIDEO_FILL_OUT_STRECH

KT_VIDEO_FILL_OUT_INTSCALE

The normal `KT_VIDEO_FILL_*` options center the video output inside the KFB and uses its dimensions to fill the screen, pixels outside the video output are not displayed. For `KT_VIDEO_FILL_OUT_*` options the dimentions used are those of the video output.

2.3 Filters

TO DO

3 Joypad

A *joypad* is the standard device for controlling the application, it supports 14 digital buttons and two analogue sticks. A joypad will connect to one of an expected maximum of 4 *ports*, this is an artificial maximum and can be changed in **katsu/joypads.h** when compiling the library. Since Katsu has no real controller, the button layout can be set to be whatever you want. The suggested use is the following:

Before reading the state of the joypad you must first poll them with `kt_Poll()`, this function is also used to listen to other events, so this should be made at the start of the main application loop. The `kt_JoyIsActive()` function returns a non-zero value if there is a controller connected at that port. By using the `kt_JoyButton*()` functions you can retrieve the state of the buttons in three conditions (whether they are held in, just been released or just been pressed), pressed buttons can be tested by doing following:

Code 3.1

```
kt_Poll(); /* Poll the joypad state */
/* Check if A and START buttons in joypad 0 have been pressed at the same time */
if (kt_JoyButtonDown() & (JOY_A | JOY_STR)) {
    /* Do something */
}
```

The state of the joysticks can be read by using `kt_JoyStick()` with the stick ID being an axis of the right or left stick. The value of the axis will be returned as a signed 8-bit value (the values will go from -128 to 127). Of course, not all controllers have joysticks, because of this, the input from the directional arrows can be copied to the joysticks with the XXX function.

3.1 Keyboard

The keyboard behavior for keyboard input is the same on startup: the keyboard is assigned to the first available joypad slot, this means that if only one controller is connected then the keyboard will be assigned to the second joypad automatically. The following is the keymapping when the keyboard:

Keyboard Key	Joypad Button
Left Arrow	Left
Right Arrow	Right
Up Arrow	Up
Down Arrow	Down
Z Key	A
X Key	B
A Key	X
S Key	Y

Q Key	L
W Key	R
E Key	TL
D Key	TR
Return Key	Start
Right Shift Key	Select

You can use the XXX function to switch the keyboard behavior from joypad to standard keyboard and back again.

3.2 Mouse

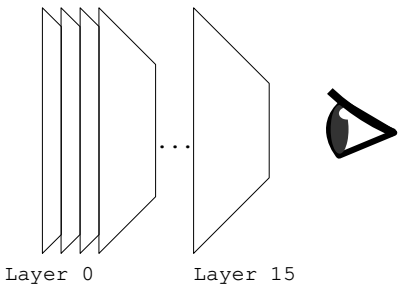
TO DO

4 Graphics

Currently Katsu's graphics subsystem is the most complex, it's designed to be simple to use and understand yet very feature rich, while a lot of effects are possible by clever use of the rendering model, some can be absent to keep things simple, so emulating them with another feature is required.

4.1 How an image is drawn

The output image is composed of what is referred to as *layers*, think of a layer as a 2D image that can be placed in-between other layers to produce a final image. There are a maximum of 16 layers that can be displayed at once, they are numbered from 0 to 15, layers with a higher number will be shown on top of layers with a lower number. After all layers are drawn, the final color of each pixel is manipulated to obtain the final image.



4.1.1 Back Color

Think of the final image as a canvas were you paste each layer in sequence from front to back, the back color would be the canvas' initial color, this means that if a given pixel of the image has all of the layers transparent at that location then the back color is used to fill that pixel's color. At startup the back color is black, this can be changed by using the `kt_BackColor()` function.

4.1.2 Offset Color

Before showing the final output image you can offset or "tint" the color of all the pixels on screen by the same amount, this can be useful to make fade to black/white effects, basic night color filtering, flashes, etc. This is a per-component offset so there are a lot of possibilities. At startup there will be no offset applied (meaning that 0 is added to all color components), to change the offset color use `kt_OffsetColor()`.

4.2 Graphics Formats

To keep Katsu simple, it uses very specific formats for all it's graphics data. All of these are composed of a sequence of bytes to keep the code as portable as possible. Katsu does no conversion from other popular formats by itself so previous format conversion is recommended, if you need to convert images on the fly there's code in the *kt-img* tool can help to accomplish that.

4.2.1 Tiles and Tile Memory

A *tile* refers to a single 8x8 pixel image. Katsu supports tiles that are 4 bits per pixel (4bpp), which is a paletted format where each pixel references one of 16 colors. As such, we need 32 contiguous bytes to store a tile:

0	0	8	8	9	9	0	0
0	8	9	A	C	B	9	0
8	8	A	A	C	D	B	9
7	8	A	A	A	B	B	9
7	8	8	9	A	A	9	8
1	9	8	8	9	9	8	7
0	1	9	9	9	8	7	0
0	0	1	1	1	1	0	0

```
0x00, 0x88, 0x99, 0x00,
0x80, 0xA9, 0xBC, 0x09,
0x88, 0xAA, 0xDC, 0x9B,
0x87, 0xAA, 0xBA, 0x9B,
0x87, 0x98, 0xAA, 0x89,
0x91, 0x88, 0x99, 0x78,
0x10, 0x99, 0x89, 0x07,
0x00, 0x11, 0x11, 0x00
```

Each byte holds two pixels, the lower 4 bits correspond to the leftmost pixel, and the higher 4 bits correspond to the rightmost pixel. Since each pixel holds an index to a 16 entry color palette, the tiles color depends on the current state of the Color Memory. Katsu's Tile Memory can store up to 16,384 tiles at once, each tile has a corresponding ID assigned, which is a 14 bit value. You can load a set of contiguous tiles to the Tile Memory by using `kt_TilesetLoad()` starting from an initial tile ID. When loading, the tiles are copied to Tile Memory so changes to loaded tile graphics must be updated by loading them again.

4.2.2 Tilemaps and Characters

A *tilemap* is a planar grid composed of 64x64 *characters*, a character informs how a tile is drawn and includes parameters such as a palette, if flipping is performed, etc. The first 64 characters of a tilemap refer to it's first row, there are a total of 16 tilemaps. One character is 4 bytes long and stores it's information in the following order:

7	6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
–	pal	–	fthi	tlo

You should set the unused bits to 0 so that future updates to the format are still compatible with previously created assets. The `pal` bits refer to the number of the palette used for the tile (from 0 to 128). The `fthi` byte stores the following information:

Bits	Name	Description
0–5	thi	Highest 5 bits of the tile ID.
6	hf	Flips the tile horizontally when set.
7	vf	Flips the tile vertically when set.

The value of `thi` (highest 5 bits) and `tlo` (lowest 8 bits) are joined to produce the 14 bit tile ID. A single character of a tilemap can be set by using `kt_TilemapSetChr()`, for loading a larger amount of tilemap data at once use `kt_TilemapLoad()`, this function allows for loading a sub-rectangle of a larger map to the Tilemap Memory.

4.2.3 Colors and Color Memory

Individual *colors* are stored in RGBA format, where each component is 8 bits. The last byte of a color (also called the alpha component) is ignored and only there to have colors 32-bit aligned:

7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
red	green	blue	alpha

A maximum of 2048 colors can be stored in Color Memory, giving a total of 128 palettes of 16 colors each when using the 4bpp tile format. You can update a set of colors starting from a given color number with `kt_PaletteLoad()`, to update just a single color use the `kt_PaletteSetColor()` function.

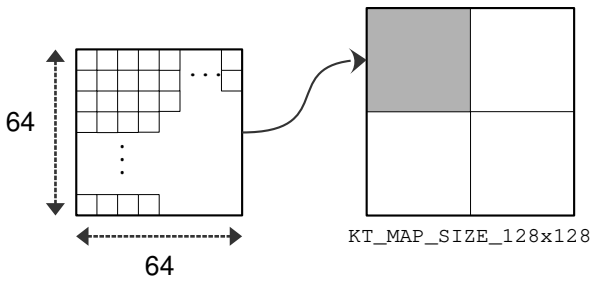
4.3 Layers

Katsu has a maximum of 16 layers, each layer can be modified independently. What a layer draws on screen depends on it's type, there are a total of 4 types: *Normal Map Layer*, *Line Map Layer*, *Rotation Map Layer* and *Sprite Layer*. Before changing the type of a layer by using `kt_LayerSetType()` you should first use `kt_LayerClear()` to zero out said layer's parameters, when a layer has type `KT_LAYER_NONE` then the layer won't be drawn on screen.

Each layer holds the same amount of data, but the data is interpreted differently depending on the type, having said this, there are two parameters that are shared between layers: the layer's blend mode, the window activation bits. Also, every layer has an *application data pointer*, the layer uses this pointer to access a portion of memory directly from the application to aid in drawing, since a poiner to memory is used you must not free the memory when a layer has it's adress attached, to be extra sure that this not happen you can use `kt_LayerClearAll()` to zero out all 16 layers at once. Generally, the `kt_LayerInitMap()` and `kt_LayerInitSprite()` helper functions can be used initialize a specified layer directly as a certain type.

4.3.1 Normal Map Layers

Everything in this subsection applies to the other types of map layers. Make a distinction between a tilemap and a *map layer*, a map layer can draw up to four different tilemaps, this depends on it's size, a map can have four sizes: 64x64, 128x64, 64x128, 128x128. The top left tilemap will correspond to the tilemap number that is specified by the layer, both of these parameters can be set with the `kt_LayerSetMapSize()` function. The following example shows the use of a 128x128 map:



Normal map layers (with type specified as `KT_LAYER_MAP_NORMAL`) are layers that render maps with no distortion applied to them, so the tilemaps are drawn as is. Since a single tilemap is generally larger than Katsu's

maximum video output, the map can be scrolled by a certain number of pixels, this pixel offset can be set by using `kt_LayerSetMapOffset()`, when the screen reaches the border of the map, the map will wrap around from the start (basically repeating the map indefinitely), with this you can update tiles in the tilemap when it is off screen to make the map appear larger.

For certain effects or game modes you would want to show just a portion of the map on screen, by using `kt_LayerSetMapRect()` you can set a rectangle on screen space were the map will be drawn (be aware that this rectangle only crops the layer, so the scrolling offset will still start from the top-left corner of the screen).

Maps can have a transparency effect applied to them, use the `kt_LayerSetMapBlend()` function to activate/deactivate the blending and to specify the alpha value used to blend, the blending mode will only be considered if blending is active.

Finally, most of the time tile/color data will be separated per map layer, you can offset the tile ID and palette of all characters in a map layer by a fixed amount by using `kt_LayerSetMapChrOffset()`. With this, you can change the tiles/colors shown by the same map without modifying either Tile Memory or Color Memory.

4.3.2 Line Map Layers

TO DO

4.3.3 Rotation Map Layers

TO DO

4.3.4 Sprite Layers

Sprite layers draw a number of specified sprites on screen from back to front, this means that the first sprite of a sprite layer will have the least priority. Sprite layers are very simple to construct, after specifying the layer type as `KT_LAYER_SPRITE` you must pass the address to an array of sprites to the application data pointer of the layer by using `kt_LayerSetAppData()` with the number of sprites to draw as the second argument. The layer's blending mode applies to all sprites in the same way, each sprite can say if blending is activated or not.

4.4 Sprites

Sprites are individual images made out of joined tiles that can move around the screen independently from each other. There is a hard limit of 2048 sprites drawn in a single pass for all layers (for example, if one layer draws 2047 sprites, then another layer can only draw 1 more sprite).

Each sprite has it's own individual attributes which are specified in 16 bytes divided across four 32-bit values, meaning that sprite data is not endian independent so be mindful of endianness if you intend to store sprite data, the `KTSpr` struct is defined as a way for the application to create sprites easily. The following code would be used to set up a sprite's values:

Code 4.1

```
KTSpr sprite = {0}; /* Create sprite (inits to zero)*/
/* POS: Set position as a (x, y) = (obj_x, obj_y) */
sprite.pos = KT_SPR_POS(obj_x, obj_y);
/* CHR: Has a size of 16x24 and is horizontally flipped */
sprite.chr = KT_SPR_CHR(tile, KT_FLIP_X, KT_SIZE_16, KT_SIZE_24, pal);
```

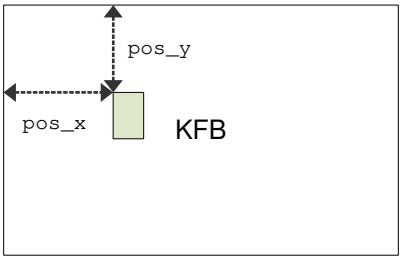
```c
/* SFX: Add a hue with 0x23 intensity and set transparency with 0x8A intensity */
sprite.sfx = KT_SPR_HUE(0xf102, 0x23) | KT_SPR_BLEND(0x8A);
/* MTX: Use the 31st matrix */
sprite.mtx = KT_SPR_MTX(31);
```

4.4.1 POS

The POS value specifies where in the screen will the sprite be drawn, the coordinates dictate the top-left corner of the sprite, the coordinates are signed values and go from -32768 to 32767, this value can be constructed with the `KT_SPR_POS()` macro:

1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10	F E D C B A 9 8 7 6 5 4 3 2 1 0
pos_y	pos_x

Bits	Name	Description
0–15	pos_x	X coordinate of the sprite.
16–31	pos_y	Y coordinate of the sprite.

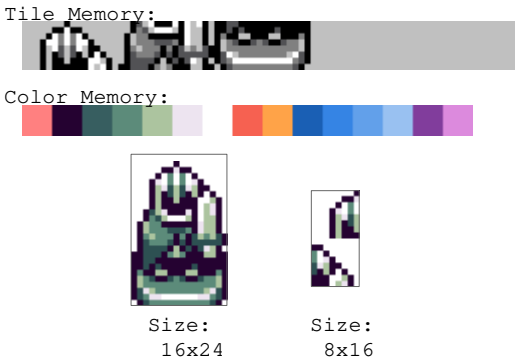


4.4.2 CHR

The CHR value specifies the set of tiles that will be drawn, the palette used and if any mirroring is to be applied, this value can be constructed with the `KT_SPR_CHR()` macro:

1F	1E 1D 1C 1B 1A 19 18	17 16 15 14	13 12 11 10	F	E	D C B A 9 8 7 6 5 4 3 2 1 0
–	pal	vsize	hsize	vf	hf	tid

Bits	Name	Description
0–13	tid	The tile ID of the top-left corner.
14	hf	Horizontal flipping
15	vf	Vertical flipping
16–19	hsize	Width of sprite (divided by 8).
20–23	vsize	Height of sprite (divided by 8).
24–30	pal	Palette number used.
31	–	Unused (should be set to 0 to avoid future compatibility issues.



4.4.3 SFX

The SFX value specifies special effects that are to be applied to the sprite: hue mixing and transparency. This value can be constructed by OR'ing the `KT_SPR_HUE()` and `KT_SPR_BLEND()` macros (these can be omitted if you don't want these effects):

1F	1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10	F E D C B A 9 8	7 6 5 4 3 2 1 0
bl_act	hue	hue_alpha	alpha

Bits	Name	Description
0-7	alpha	The alpha value of the sprite.
8-15	hue_alpha	Amount of hue mixing
16-30	hue	Hue to be mixed to the sprite
31	bl_act	Transparency active toggle.

4.4.4 MAT

The MAT value specifies the index of the matrix loaded in Matrix Memory that will be applied to the sprite, the index 0 refers to the identity matrix, this matrix is a constant in Matrix Memory so it cannot be overwritten. This value can be constructed by using the `KT_SPR_MTX()` macro:

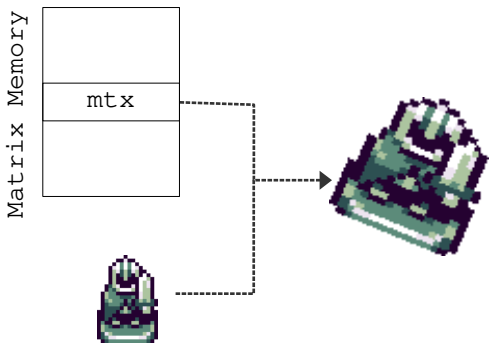
1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F E D C B A 9 8	7 6 5 4 3 2 1 0
-	mat_idx

Bits	Name	Description
0-7	mat_idx	Index of the matrix to apply.
31	-	Unused (should be set to 0 to avoid future compatibility issues).

4.4.5 Matrix Memory

Matrix Memory is a section of memory that is reserved for sprites to reference at all times, it can store 256 2x2 floating point matrices at once. Since there are less matrices than sprites you should optimize the matrix

usage. The matrix with index 0 is constant and refers to the identity matrix (use the constant `KT_MTX_IDENTITY`), the other 255 entries can be set by using `kt_MtxSet()`.



Matrices are generally used for rotation and scaling so the `kt_MtxSetRotoscale()` function can be used to set these transformations easily, note that the angle passed to this function is in radians.

4.5 Blending

Transparency effects can be done via color blending. Each layer has a blending mode that is applied when the map/sprite has it's blending bit active. The blending equation is the following:

$$pix\_color = Clamp(src\_color \times src\_factor \pm dst\_color \times dst\_factor)$$

By using `kt_LayerSetBlendMode()` you can specify whether an addition or subtraction is done (`KT_BL_FUNC_ADD` and `KT_BL_FUNC_SUB` respectively). Values src_factor and the dst_factor are one of the following constants:

Blending factor	Description
`KT_BL_ONE`	Constant of 1
`KT_BL_ZERO`	Constant of 0
`KT_BL_SRC_ALPHA`	Alpha from the map/sprite
`KT_BL_INV_SRC_ALPHA`	1 - Alpha from the map/sprite
`KT_BL_DST_ALPHA`	Alpha from the screen's pixel
`KT_BL_INV_DST_ALPHA`	1 - Alpha from the screen's pixel

4.6 Windows

TO DO

4 AUDIO

TO DO