RETROHACK

TECHNICAL INFORMATION

TABLE OF CONTENTS

OVERVIEW

The RetroHack machine is composed of three principal parts — the CPU, the blitter and the sound generator.

The CPU is used for all logic calculations and for controlling the blitter and the sound generator.

The blitter is the CPU's interface to all video functions. It has access to two framebuffers with hardware page flipping and 64 kB of sprite memory. It can transfer graphics from sprite memory to the framebuffer through a highly flexible blitting operation.

The sound generator is an industry-standard AY-3-8910 and offers three tone generators, a noise generator and 16 different sound envelopes.

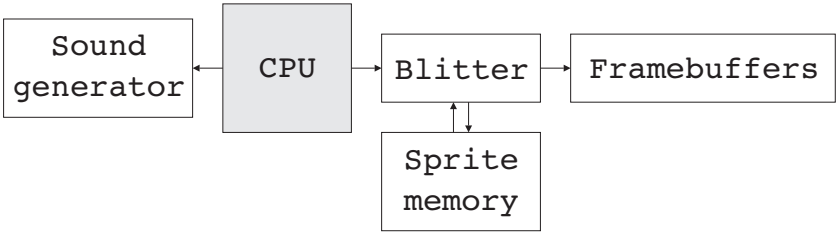


Figure 1: hardware overview

THE BLITTER

All drawing operations are performed by the blitter. It has three primary functions:

- Transferring graphics from the CPU to sprite memory
- High speed blitting of graphics from sprite memory to the frame buffer
- Flipping the frame buffers for smooth animation

THE FRAMEBUFFER

The RetroHack machine has a 320×240 pixel, 256 colour framebuffer which stores colours in the YCbCr colour space. Four bits are allocated to storage of Y, and two bits each to Cb and Cr.

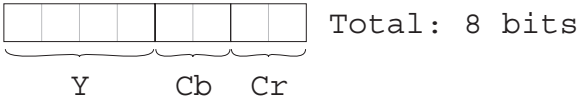


Figure 2: framebuffer format

SPRITE MEMORY

Graphics are stored in sprite memory for later transfer to the framebuffer by the blitter. The CPU has no direct access to sprite memory, but can instruct the blitter to move data there.

Images are stored in sprite memory in 16 colour, 4 bit format. The blitter fills in the other 4 bits necessary to make 8 bit images while blitting.

BLITTING

The blitter allows transfer of graphics from sprite memory to the framebuffer, with arbitrary scaling. Advanced sprite effects can be achieved through its support for progressive per-scanline modification of position and scaling on the x-axis.

PAGE FLIPPING

The blitter is connected to two hardware framebuffers. These are set up to provide hardware page flipping, and are always organised so that one is providing visible information and the other is being drawn to.

FUNCTION REFERENCE

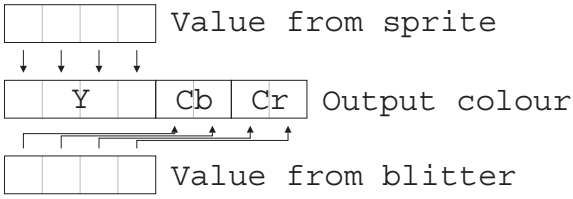
void ****rt_UploadGraphic****(enum rt_TargetFormats* ***TargetFormat****, unsigned char* ****Data****, int* ***SourcePixelsPerByte****, int* ***Width****, int* ***Height****, int* ***MaskCol****)*

Uploads a graphic to sprite memory and returns an opaque pointer that can be used to identify the graphic in subsequent graphics calls. If there is not enough sprite memory left to store the graphic, this function returns **NULL**.

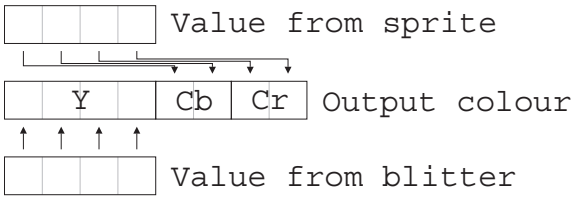
All graphics are stored in sprite memory in a 4 bit format. They are expanded to 8 bits when they are blit using 4 additional bits provided directly by the blitter.

TargetFormat defines how the two sets of 4 bits will be combined:

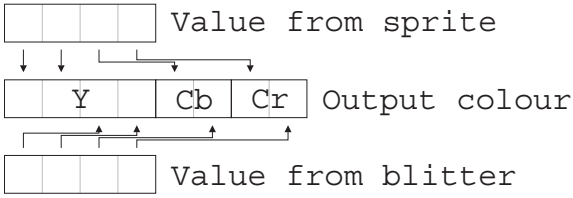
RT_Y – the 4 bits from the sprite are the Y component of each pixel. The 4 bits from the blitter are the Cb and Cr channels. i.e.



RT_CbCr – the 4 bits from the sprite are the Cb and Cr components of each pixel. The 4 bits from the blitter are the Y channel. i.e.



RT_MIX – The 4 bits from the sprite and the 4 bits from the blitter are interleaved across all three channels as:



Data is a pointer to the data that should be uploaded to sprite memory.

SourcePixelsPerByte tells the blitter how many pixels are stored in each byte of **Data**. It may be either '1' or '2'. '1' indicates that each unique byte of **Data** holds a single pixel of the sprite, stored in its low 4 bits. '2' indicates that each unique byte of **Data** holds two pixels of the sprite, a left pixel in the high 4 bits and a right pixel in the low 4 bits.

The RetroHack machine requires that individual scanlines start on byte boundaries. So if you want to upload a 2 pixels per byte graphic with a width that is an odd number of pixels then you must remember to leave an empty nibble on the end of each scanline of data.

Width and **Height** inform the blitter of the dimensions of the sprite being uploaded.

MaskCol indicates which of the 16 sprite colours will be transparent if the sprite is later blitted with masking.

void ***rt_FreeGraphic****(void* ****graphic****)*

Frees the memory being used to store a graphic and removes the graphic from sprite memory.

void ***rt_Blit****(void **graphic*, *int* ***OtherNibble***, *int* ***x***, *int* ***y***, *int* ***ScaleX***, *int* ***ScaleY***, *int* ***Skew***, *int* ***ScaleChange***)

Blits **graphic** from sprite memory to the framebuffer. The sprite's mask colour is ignored.

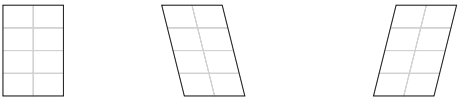
The low 4 bits of **OtherNibble** are used by the blitter to convert **graphic** from the 4 bit version stored in sprite memory to an 8 bit version suitable for the framebuffer. How exactly these bits are used depends on the **TargetFormat** passed to **rt_UploadGraphic**.

x and **y** set the position of the sprite on screen. These are **16:16 fixed point** numbers.

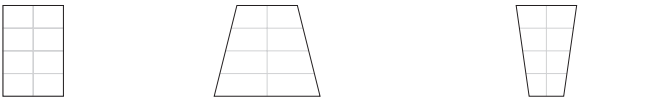
ScaleX and **ScaleY** set the amount of scaling that is applied to the image. They are **16:16 fixed point** numbers. Passing the fixed point value '1' for each causes the sprite to be reproduced so that one source pixel covers one framebuffer pixel. Passing '2' for **ScaleX** causes the sprite to be blitted twice as wide as normal.

The **Skew** value is an amount added to **x** after every scanline. Passing a value of 0 will result in a normal blit operation, passing a positive value causes the bottom of the sprite to be skewed to the right and passing a negative value causes the bottom of the sprite to be skewed to the left.

i.e.



The **ScaleChange** value is an amount added to **ScaleX** after every scanline. Passing a value of 0 will result in a normal blit operation, passing a positive value will cause the bottom of the sprite to be fatter than the top of the sprite and passing a negative value will cause the bottom of the sprite to be thinner than the top. i.e.



*void **rt_MaskedBlit**(void ***graphic**, int **OtherNibble**, int **x**, int **y**, int **ScaleX**, int **ScaleY**, int **Skew**, int **ScaleChange**)*

rt_MaskedBlit works in exactly the same was as **rt_Blit** except that any pixels from the sprite that match the sprite's mask colour are not written to the framebuffer.

void ***rt_Flip****(void)*

Instructs the blitter to flip the two hardware framebuffers, causing the one that was being displayed to be the one that is drawn to and the one that was being drawn to, to be displayed.

Flipping can only occur during vertical retrace, so this function halts the CPU until vsync.

TIMING

The blitter is clocked at 21,272,400 Hz — a little over 21 Mhz. The RetroHack machine produces 50 frames per second, so that gives 425,448 cycles per frame.

rt_UploadGraphic costs 60 + 8**total number of pixels* cycles.

rt_FreeGraphic costs 1 cycle.

rt_Blit costs 25 + *total number of possible framebuffer pixels calculated* + 15**number of possible scanlines calculated* cycles. All sprite clipping is done at the end of the blitting process, so off-screen pixels cost the same as on-screen pixels.

rt_MaskedBlit costs the same amount as an equivalent **rt_Blit** of the same sprite would have cost.

rt_Flip costs 1 cycle + however many cycles it has to wait for the next end of frame.

OTHER BLITTER NOTES

The blitter is able to flip sprites both horizontally and vertically. This is achieved by passing negative values as the **ScaleX** and **ScaleY** parameters during a blit.

Sprites are always scaled around their upper left hand corner. So a sprite that is flipped over the vertical axis will fill out to the left of its designated screen position and a sprite that is flipped over the horizontal axis will fill out upwards from its designated screen position.

Sprites that are flipped across the horizontal axis are drawn from bottom to top. So **Skew** and **ScaleChange** still affect drawing as it moves from the top of the stored sprite image to the bottom.

Sprites may not be more than 512 pixels wide or tall.

For scaling purposes, the RetroHack machine treats the centre of each pixel as the position for colour sampling.

A complete output frame contains 311 horizontal lines once the border and vertical sync periods are factored in.

The values used by the video circuits for Y map onto analogue values starting at 0.0625 and incrementing by 0.05346679688 with each discrete step. The values used for Cb and Cr start at -0.4375 and increment by 0.21875.

READING INPUT

The RetroHack machine is equipped with a 62 key memory mapped keyboard. By reading the value of different keylines a program can query the status of different keys. Keylines are read using the function:

Uint8 ***ay_ReadKeyboard****(Uint16* ***Line****)*
Returns the current value of keyline **Line**.

The keyboard is mapped to memory as follows:

	Bit						
Line	**7**	**6**	**5**	**4**	**3**	**2**	**1**
0	6	5	4	3	2	1	ESC
1	7	8	9	0	-	=	BACK SPACE
2	y	t	r	e	w	q	TAB
3	u	i	o	p	[	]	ENTER
4	h	g	f	d	s	a	
5	j	k	l	;	'	#	
6	b	v	c	x	z	\	LEFT SHIFT
7	n	m	,	.	/	RIGHT SHIFT	
8	SPACE	LEFT ALT	LEFT CTRL				
9	RIGHT ALT	RIGHT CTRL					
10	RIGHT	UP	DOWN	LEFT			

A **0** in any bit indicates that the corresponding key **is pressed**.

Most RetroHack machines have a standard QWERTY keyboard, but some minor variations can exist from country to country.

Reading the keyboard costs 1 cycle on the 21 Mhz bus.

THE SOUND GENERATOR

The sound generator is an industry-standard AY-3-8910. It provides:

- Three channel audio
- Tone and noise generators
- 16 hardware envelopes

Full details of how to program the AY are given on the included datasheet. Only information about how to access the AY is given below.

ACCESSING THE SOUND GENERATOR

All sound generator access is performed through just two functions:

void ***ay_Write****(Uint16* ***Addr****, Uint8* ***Value****)*
Writes **Value** to the register at **Addr**.
Possible values for **Addr** are:

AY_REGSELECT — AY register select
AY_REGVALUE — AY register value

Uint8 ***ay_Read****(void)*
Returns the value of the currently selected AY register.

AY MUSIC LIBRARY

The RetroHack library also includes support for playback of PSG format music files. PSG files store AY register changes, accurate to the nearest display frame.

Three functions are provided for playing back PSG files:

int ***psg_OpenFile****(const char* ****name****, int* ***loop****)*
Attempts to open the file stored as **name**. Returns **TRUE** if the file was successfully opened, **FALSE** otherwise.

You can specify how you want the music to repeat using the loop parameter. Pass TRUE if you want the sound file to loop forever, FALSE otherwise.

int ***psg_Update****(void)*
This function should be called once per frame, and allows the PSG player to update the music output.

It returns **TRUE** if the sound was successfully updated, **FALSE** if no PSG file is open or if it has played past the end of a PSG file for which the user requested no looping.

void ***psg_CloseFile****(void)*
Closes the open PSG file if one is open, and silences the audio.

TIMING

The AY-3-8910 is clocked at 3.5454 Mhz, which is equal to the blitter clock divided by 6. All writes to the AY-3-8910 are synchronised with that bus and cost 1 cycle on the 3.5454 Mhz bus.

ADDITIONAL ROUTINES

The RetroHack simulation library provides the additional support routines listed below. Unless specified otherwise, these calls cost nothing to the simulated RetroHack machine.

INITIALISATION/DE-INITIALISATION & MISCELLANEOUS

int ***rt_Init****(void)*
Attempts to initialise the RetroHack simulation library. This should be called at the start of all RetroHack programs.

Returns **TRUE** if successful, **FALSE** otherwise.

void ***rt_Exit****(void)*
Shuts down the RetroHack simulation library. This should be called at the end of all RetroHack programs.

END_OF_MAIN*()*
A helper macro required by some of the libraries that the RetroHack simulation library may rely on. You should include the text 'END_OF_MAIN()' immediately after your main function.

TIMING

void ***rt_WaitCycles****(int* ***Cycles****)*
Causes the CPU to wait for **Cycles** ticks of the 21.2 Mhz bus.

void ***rt_WaitEvent****(int* ***Event****)*
Causes the CPU to wait until the event specified in **Event**, which may be one of:

RT_HSYNC — Wait for the next horizontal sync
RT_VSYNC — Wait for the next vertical sync

FIXED POINT HELPERS

ftofix*(float* ***Number****)*
Evaluates to the fixed point version of **Number**.

itofix*(int* ***Number****)*
Evaluates to the fixed point version of **Number**.

fixtoi*(int* ***Number****)*
Evaluates to the integer version of the fixed point value stored in **Number**.

fixtof*(int* ***Number****)*
Evaluates to the floating point version of the fixed point value stored in **Number**.

fixmul*(int* ***A****, int* ***B****)*
Evaluates to the fixed point result of A×B.

fixdiv*(int* ***A****, int* ***B****)*
Evaluates to the fixed point result of A÷B.

HOST OS INTERACTION

int ***rtSim_QuitWanted****()*
Returns **TRUE** if the OS running this RetroHack simulation is requesting that your application should quit, **FALSE** otherwise.

void ***rtSim_SetWindowTitle****(const char* ****name****)*
Communicates the title of your program to the host OS for purposes such as window titling.