Vircon32: How the console works

Document date 2023.01.22 Written by Carra

What is this?

This document is a quick guide to learn how the Vircon32 virtual console works. It does not intend to be a specification or to cover all the features in detail: it is only intended to support those who want to create programs for the console. This guide is oriented to program Vircon32 in C language. To create programs in assembler it is necessary to know more internal details of the console operation that are beyond the scope of this document.

Vircon32

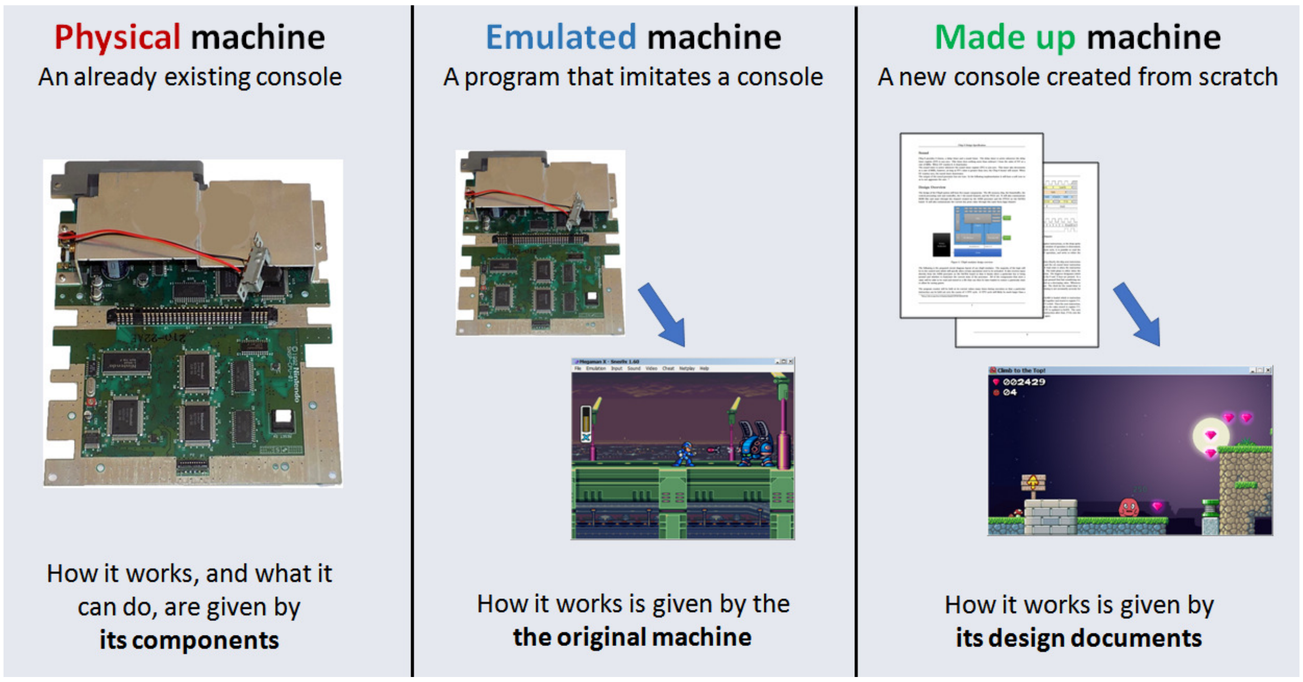
Summary

The first part of this document explains the general features of the console and its basic architecture. The second part covers the operation of each of the console components.

PART 1: KNOWING THE CONSOLE

Introduction

Vircon32 is a 32-bit virtual console. This is a made up machine: it's not based on any previously existing console. The design documents and its compliant implementations ARE the console.


Physical machine
An already existing console
How it works, and what it can do, are given by its components
Emulated machine
A program that imitates a console
How it works is given by the original machine
Made up machine
A new console created from scratch
How it works is given by its design documents

The full project covers some other areas as well. Together with the emulator and its games, there will also exist development tools, guides such as this one, test programs, etc.

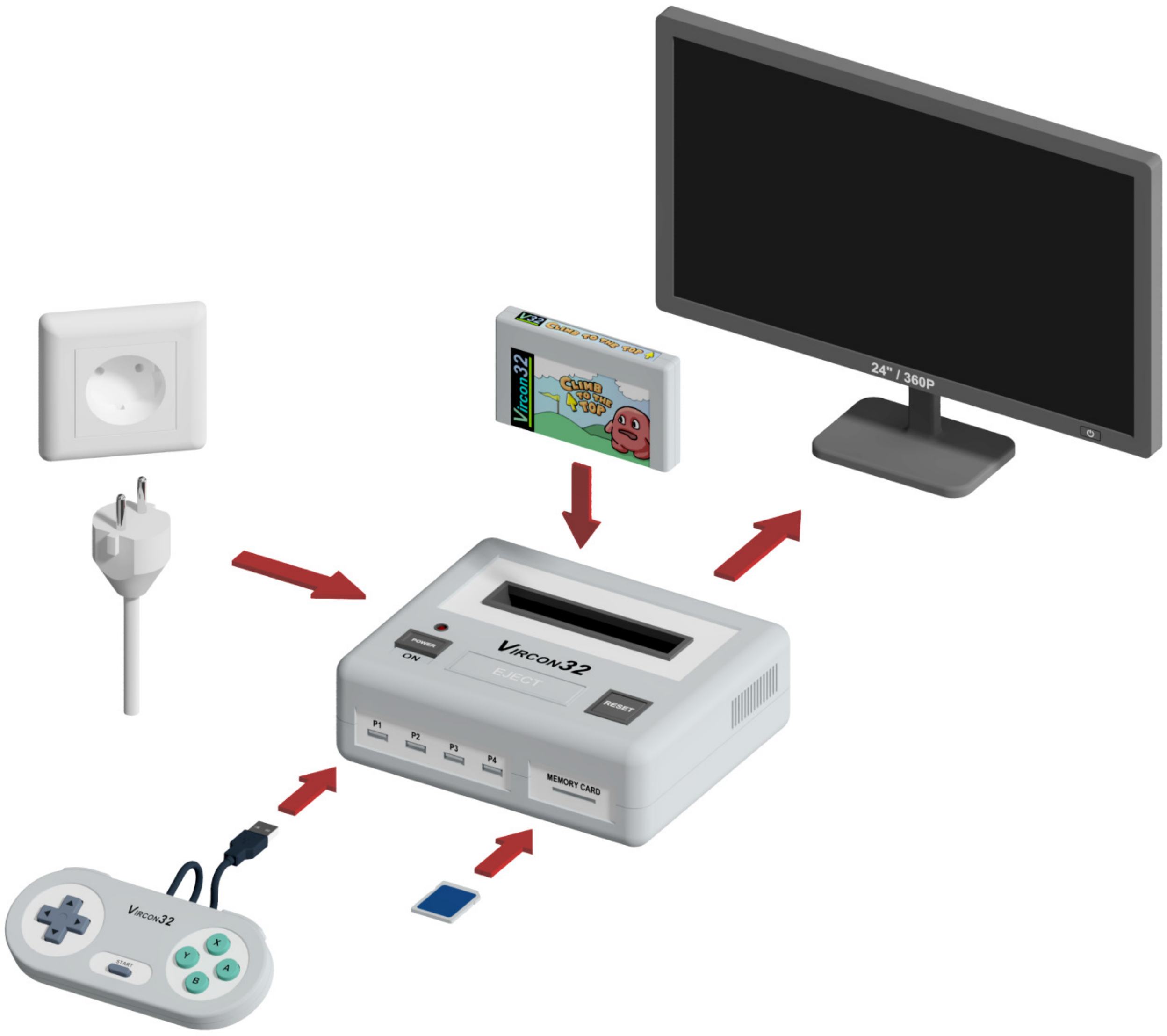
How is Vircon32?

Vircon32 has been designed to be as simple as possible. It is modeled as a real machine (meaning: it has a processor, graphics chip, controllers, buses...) but its components are simplified to make it easier to program games, as well as creating emulators and other tools.

The second goal when designing this machine has been to ensure that the console permits the creation of elaborate games that go beyond simple technical demos or minigames. This console is based on the 32-bit generation of consoles, and its capabilities are in line with that generation (although without 3D capabilities). Thus, it allows to go beyond the typical "8-bit style" that nowadays has become way too limited.

Parts of the console

Vircon32 does not model just the console: Elements such as the gamepads, the display or the cartridge also have to be included in the design, and implementations must take them into account.



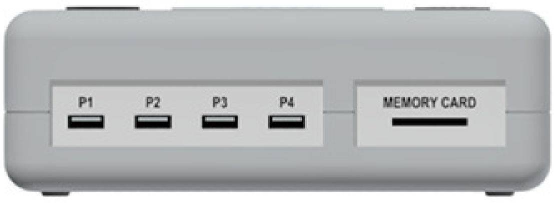
What does this mean? Let's take for instance the controller. The player may actually be playing with a different type of controller than Vircon32's, or even a keyboard. But for anyone programming a game, the controller that it's being played with is always the Vircon32 gamepad. It will be the emulator's task to adapt the real controller inputs, and players will be interested in trying to map their buttons in a similar way to the Vircon32 controller layout (since the games can take for granted that placement in their controls).

The Vircon32 system in itself is not very different from the classical 32-bit consoles: aside from the console we have up to 4 gamepads, a cartridge and optionally a memory card. The console’s output is some kind of video and audio display.

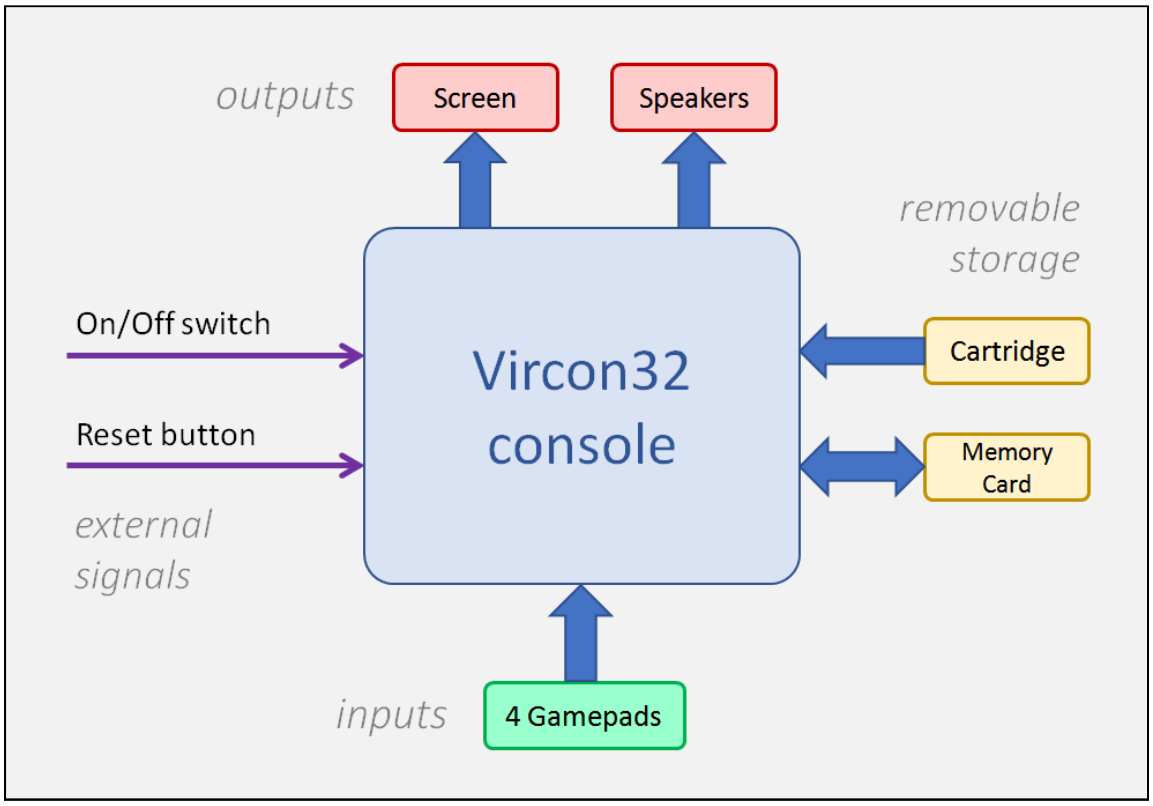
The console

The console is the core component since it runs the games. But we won't explain its inner workings until later sections. At the external connectivity level we have the cartridge input, 4 gamepad ports, a slot for a memory card and the connection to the display.

VIRCON32
POWER
ON
EJECT
RESET

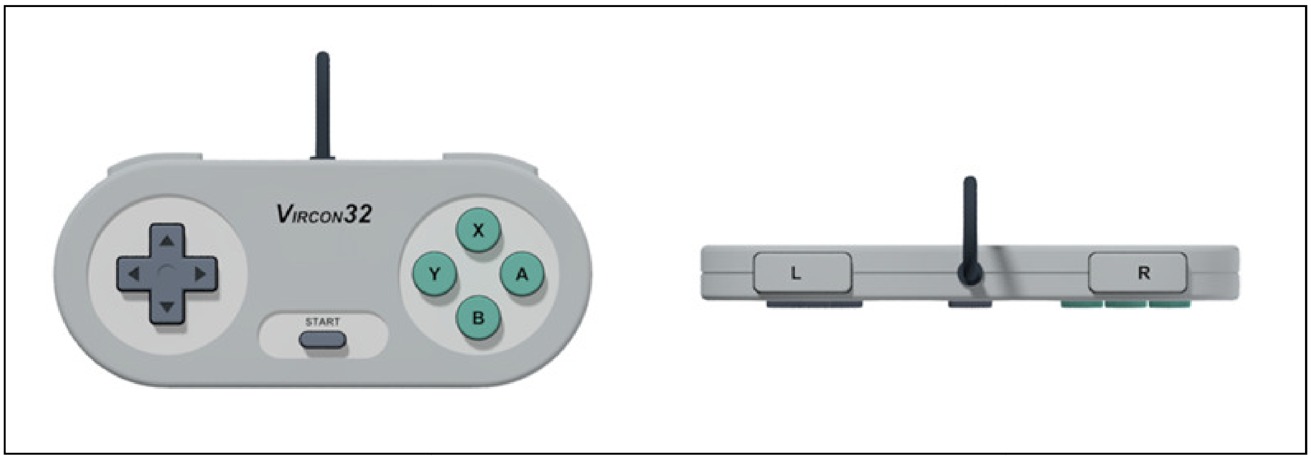


In addition to connecting or removing devices from the console, we (the users) can control its operation using the power switch and the reset button.



Gamepads

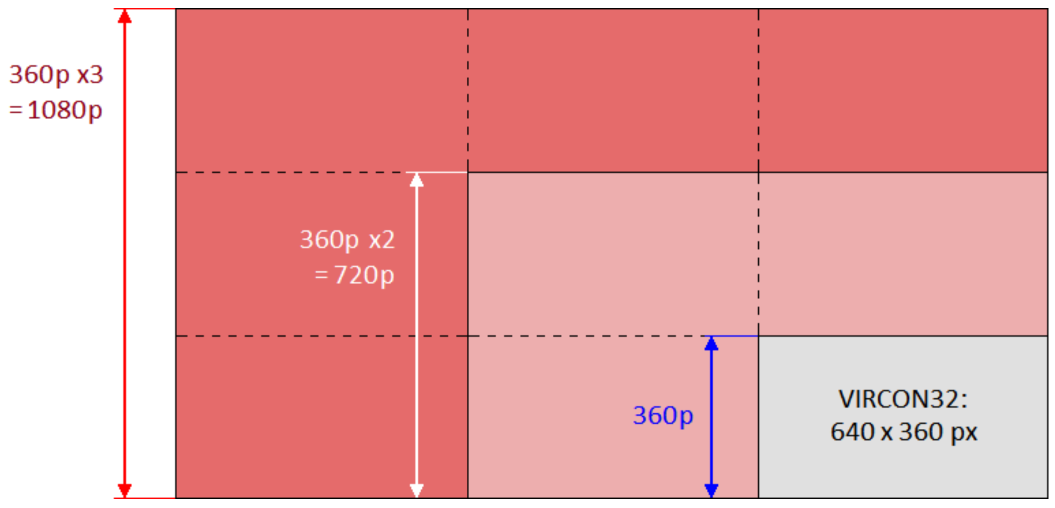
Vircon32 gamepads feature a d-pad, 4 front buttons, 2 shoulder buttons and a central button for Start. The layout of these elements is as seen in this image.



All controls in the gamepad are digital: they only distinguish between being pressed or not. The d-pad uses a tilting mechanism, so in each axis the 2 opposite directions can never be pressed at the same time.

The screen

Vircon32 screen has a resolution of 640x360 pixels at 16:9 aspect ratio, and it works at 60 frames per second. So, in terms of modern displays, it is a 360p resolution. That resolution allows us to adapt Vircon32 image output to the most common screens used as of today (720p, 1080p, 1440p, 4K) using integer scaling to prevent deforming the image.



In terms of color depth, the screen can display true color (RGB color with 24 bits per pixel).

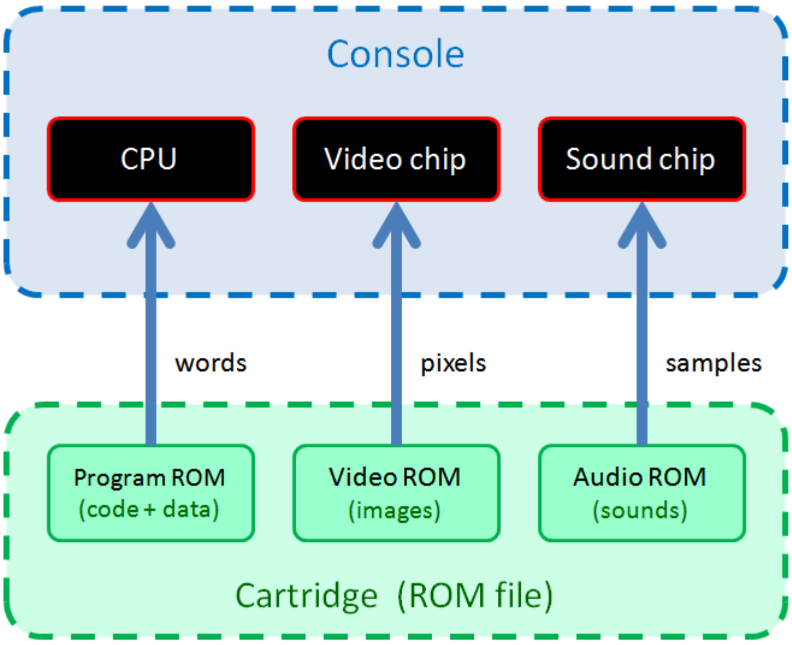
The cartridge

Vircon32 cartridges are made from 3 read-only memories (ROMs) that are independent from each other. When the cartridge is inserted each of those memories becomes connected to a component within the console:

- **The program ROM:** Contains the program to be executed and all data needed by that program. It gets connected to the memory bus, from where the CPU can access its contents.

- **The video ROM:** Contains a collection of all the images the game can show. It gets connected directly to the graphics chip (CPU).

- **The audio ROM:** Contains a collection of all the sounds the game can play. It gets connected directly to the sound chip (SPU).

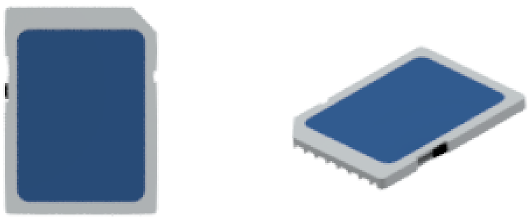


The console can read those 3 types of content from cartridges with immediate access, so loading times don't exist and there is no need to copy anything in memory. This is possible because, for the sake of simplicity, all included images and sounds are stored uncompressed.

However, since there is no compression, games need to be able to take up more space. To ensure that this does not prevent the creation of elaborate games, cartridge limits have been defined generously. Audio and video ROMs can reach a maximum of 1 GB each, while the program ROM has a limit of 512 MB. Therefore a Vircon32 cartridge can store up to 2.5 GB in total.

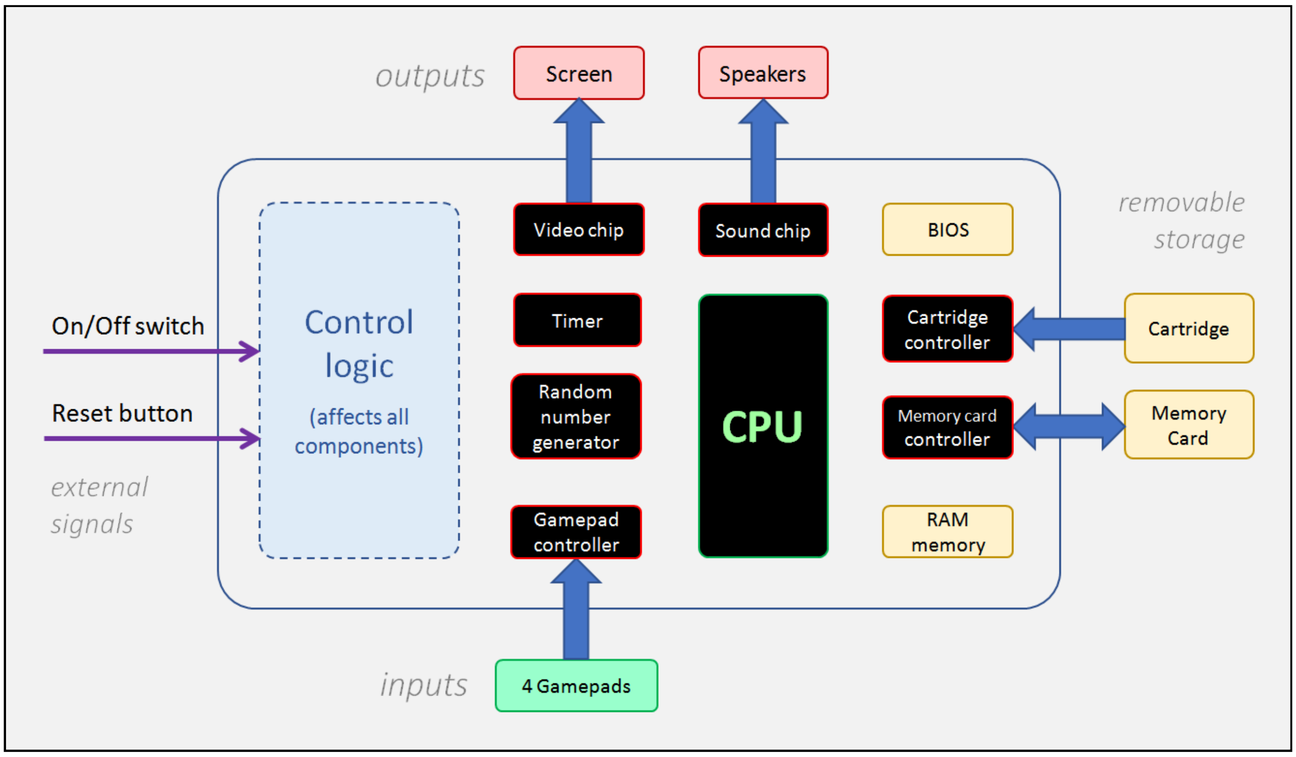
Memory card

Since cartridges only contain ROMs (this is, they are read-only), the console allows the use of memory cards as a permanent storage. With this we can support the creation of longer or more complex games, in which the game can be saved. These memory cards have a capacity of 1 MB.



Console architecture

We have already explained the basics about all devices that can be connected to the console. Now we will see the console itself in greater detail. We can take the previous connection diagram for the whole system, and extend it to see what is inside the console. At a basic level this is what we would see:



There is no need to know all this detail in order to program games in C. The most important thing is knowing that the CPU is the main chip from which we run the game. The CPU is able to communicate with the rest of chips to request them to perform their specific functions. For instance, when our game's program calls functions that draw on screen, the CPU will send commands to the graphics chip that indicate what should be drawn and where.

Console inner components

To begin knowing the console components we will now give a general idea about each one of them and their functions.

- **Processor (CPU):** It is the main chip and the one running the game. The processor executes our program's instructions one by one and interacts with the other chips when it is required.

- **Timer:** Controls when each CPU execution cycle must happen. Also, 60 times per second, it inits a new frame. This triggers the screen update, but it also controls some of the functions of other chips.

- **Input controller:** It will detect which gamepads are connected at each moment. It allows to query the state of their d-pads and buttons.

- **Graphics chip (GPU):** This chip accesses the images present in the cartridge's vidro ROM and uses them to draw on screen. It can apply some effects to them such as rotation and scaling.

- **Sound chip (SPU):** The SPU has direct access to the sounds in the cartridge's audio ROM. It can play up to 16 of them at the same time, and apply some effects like speed changes and loops.

- **Cartridge controller:** Detects when a cartridge is connected, and can provide basic information on its contents. Through it the CPU can read the program memory in the cartridge.

- **Memory card controller:** Allows us to know if there is a memory card connected. It enables the CPU to access the card's memory.

- **Random number generator:** It uses an algorithm to produce pseudo-random numbers each time we request it, allowing us to simulate randomness.

- **RAM memory:** It's the working memory for the program running in the CPU.

- **The BIOS:** It's an internal software similar to a cartridge (program, video and audio). It controls the console startup and reacts to possible hardware errors. It also includes a text font to allow writing on screen.

In part 2 of this document we will see every console component in detail, to understand how they work and know what our programs can do.

Comunication between components

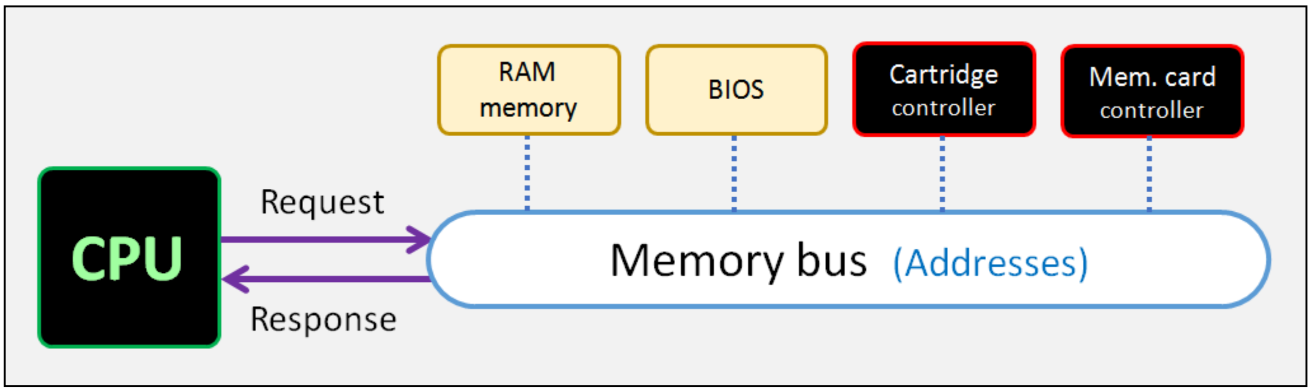
The console components that we described above cannot function in isolation. They must be capable of communicating to be able to interact and make the console work. For this reason there are 2 communication channels or buses, that are used for different purposes.

Vircon32 buses use a master-slave scheme, in which there is a unique master device that controls communication and sends commands. The rest of connected components (the slaves) behave passively and are restricted to answering the requests they receive. In both buses the CPU is the device controlling communication (the master), and the mission of all other components connected to the bus is to service the processor.

Memory bus

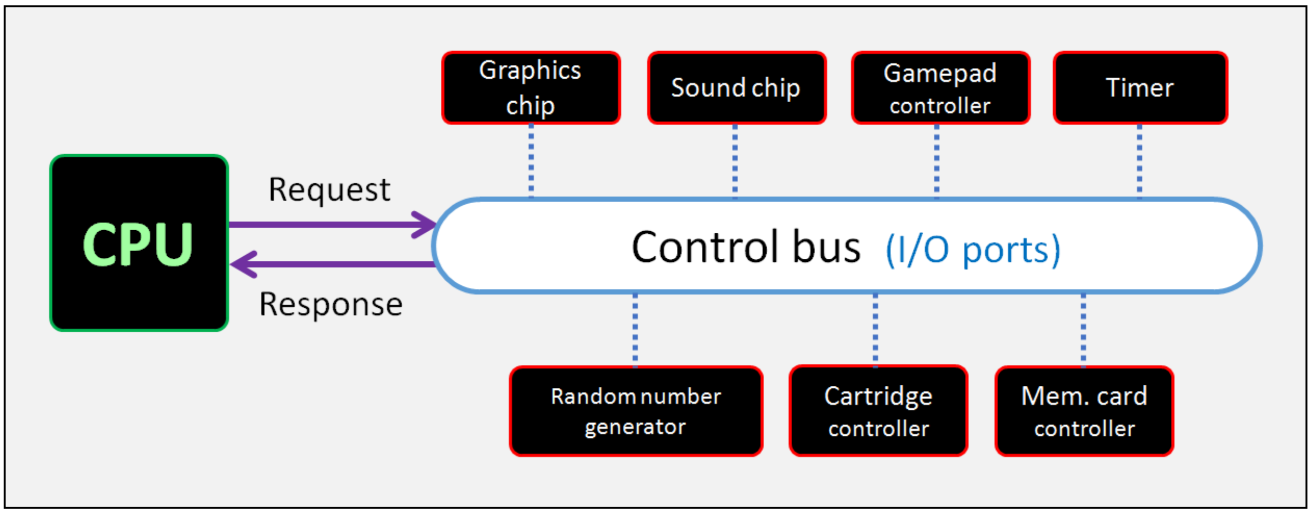
This is the bus to which all devices that have memory connect (be it RAM or ROM memory). By connecting to this bus, its memory space receives an address range and its contents become accessible by the CPU.

The CPU, as master, can request the bus to perform a read or write operation in some specific memory address. If the operation is performed, the CPU will receive a response with the result. If it cannot be performed, a hardware error will be raised and our program will be stopped. This can happen either because said memory address does not exist, or because the operation is not valid in that address (for instance: a write is attempted on the cartridge memory, which is read-only).



Control bus

This bus allows the processor to control the functions of other chips and send them commands to take action. The control mechanism is the following: each chip that can be controlled exposes externally a set of input/output ports in which a value can be read and/or written. When connected to the control bus, each chip is assigned a range of port numbers and the CPU can send and receive data from them.



For all ports the value that can be sent or received is a single 32-bit word. This means that, at bus level, handling a port is very similar to handling a memory address. And, same as in the address bus, if the CPU requests an operation that cannot be performed, our program will be stopped by a hardware error.

However, the behavior of each of these ports can be very different for every specific case. There are read/write ports, read-only ports and write-only ports. And while some ports act as simple registers, in others the operations of reading or writing at the port can trigger other actions.

PART 2: CONSOLE COMPONENTS

In this second part we will see each component of the console more in depth: what it can do and how we can use it. However, before that, we will mention an important detail that affects the whole console in general: to simplify the CPU and memory, data in Vircon32 can only be handled in 32-bit units. This means that the minimum unit in this console is not the byte, but the 32-bit word. When creating programs this will affect us in several different ways:

- CPU registers are always used as a whole: there are no variants of the instructions to handle only 8 bits or 16 bits as in other CPUs.
- Memory addresses and their offsets are always expressed in words, not bytes.
- The same is true for data sizes, because in Vircon32 it's not possible to have individual bytes.

However, this can also mean certain advantages for us. When programming other systems we must take into account the alignment of bytes in memory, and if bytes are being grouped in big-endian or little-endian system. In Vircon32 we are not affected by this.

Processor (CPU)

The CPU is the central component of the console, since it is the one that runs our program. To create games in Vircon32 we don't need to know much about the CPU itself, unless we are going to write assembly code, so we are only going to mention some general details about it.

CPU Control

From our programs in C we can use some functions to control the execution of the program. We can use `end_frame()` to pause the CPU until the next frame begins. This allows us to control the speed of the game.

```
// game's main loop
while( true )
{
    // process thr current frame
    // (...)

    // wait for the next one
    end_frame();
}
```

On the other hand, if our game has ended and we want to stop the program completely we have 2 options. The first of them is to just exit the main function. The second is to use function `exit()`, that can be called at any point in the code.

```
void ShowGameEnding()
{
    // show ending sequence
    // (...)
    
    // the game has ended and we halt it
    exit();
}
```

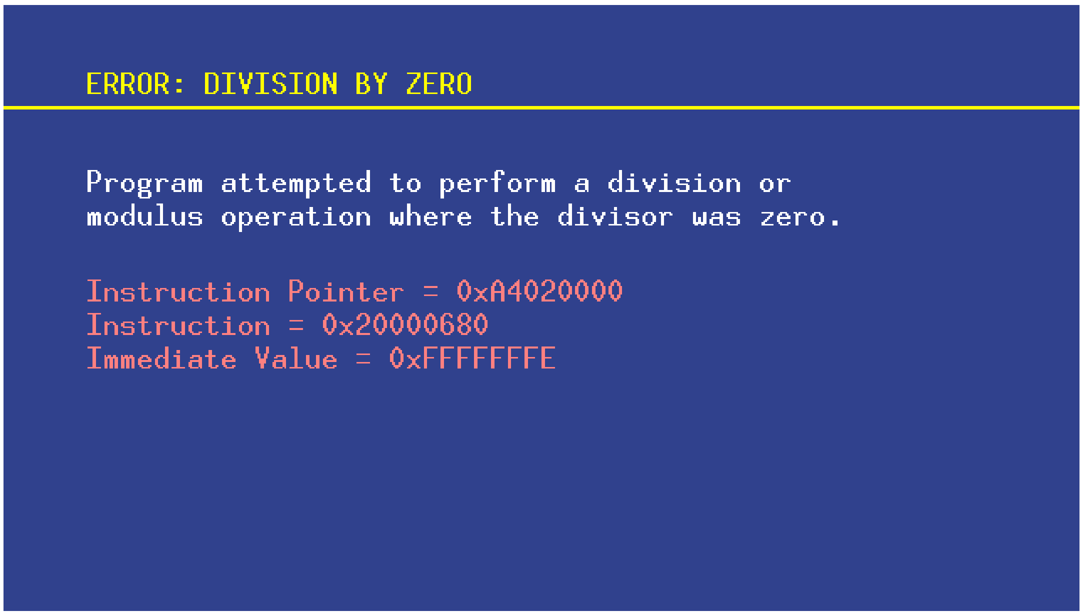
Performance

The CPU runs at 15 MHz. For simplicity this CPU will always execute any instruction in 1 clock cycle. In Vircon32 many aspects of a game are going to be processed for each frame, and running at the typical 60 fps will mean that we are capable of running up to 250,000 instructions per frame.

In theory a game can also control execution in such a way that it uses more processing power and does not update every frame but for example every 2 frames (at 30 fps). However other functions such as screen refresh or controller update are independent of the program, and will continue to function at 60 fps.

Hardware errors

There are some situations when running a program that can cause a hardware error, such as dividing by zero. Any of them will cause our program to stop, and the console will display an error message like the following.



```
ERROR: DIVISION BY ZERO

Program attempted to perform a division or
modulus operation where the divisor was zero.

Instruction Pointer = 0xA4020000
Instruction = 0x20000680
Immediate Value = 0xFFFFFFFE
```

Timer

The timer keeps track of the elapsed time, and takes care of sending other components the appropriate signals when they must perform some action. Time measurement is done at 3 different levels:

Cycle counter

This counter measures the CPU cycles that have passed since the start of the frame. That is, each frame counts from 0 to 249,999. We can query it from C using the `get_cycle_counter()` function. It is not common for a game to use it, but it can have some uses. For example you can read the value before the end of the frame to internally know the CPU load and display it on screen in a debug mode

```
// calculate the % of CPU used
int UsedCycles = get_cycle_counter();
float CPUPercentage = 100.0 * UsedCycles / 250000;

// wait for the next frame
end_frame();
```

However we must keep in mind that the cycle counter cannot be used to time events. A Vircon32 emulator is required to guarantee that the start of each frame happens when it should (at regular intervals, 60 times per second). However it is free to emulate the cycles of that frame at any rate that it considers appropriate. Even if our program reads a cycle counter equal to 125,000, we should not think that exactly 1/120th of a second has passed in real time.

Frame counter

The frame counter starts at 0 when the console is turned on, and measures the number of frames that have passed since that moment. This counter is the one most commonly used in programs, since it is our main way of measuring time. In this case the function `get_frame_counter()` is used. For example, in the main loop we could do:

```
// save current time at an initial event
if( Event.Happened )
  StartingFrames = get_frame_counter();

// (...)
// activate a response to that event after 1 second
if( Event.Happened )
  if( get_frame_counter() - StartingFrames >= 60 )
    TriggerResponse( &Event );

// (...)
// wait for the next frame
end_frame();
```

Additionally we can also use the sleep() function, which relays on the frame counter to wait for a certain time. It is useful for example in scene transitions in which we want to include a small wait

```
// fade the scene to black
FadeToBlack( &Scene1 );

// keep the screen black for 0.5 seconds
sleep( 30 );

// make the transition to the next scene
FadeFromBlack( &Scene2 );
```

Date and time

The timer does not only control the time relative to the console itself. It also features an internal clock with the current date and time. These can be queried with get_date() and get_time(). These functions will provide the information in the native Vircon32 binary format (an int for each), but we can convert them to the normal human format with the following code:

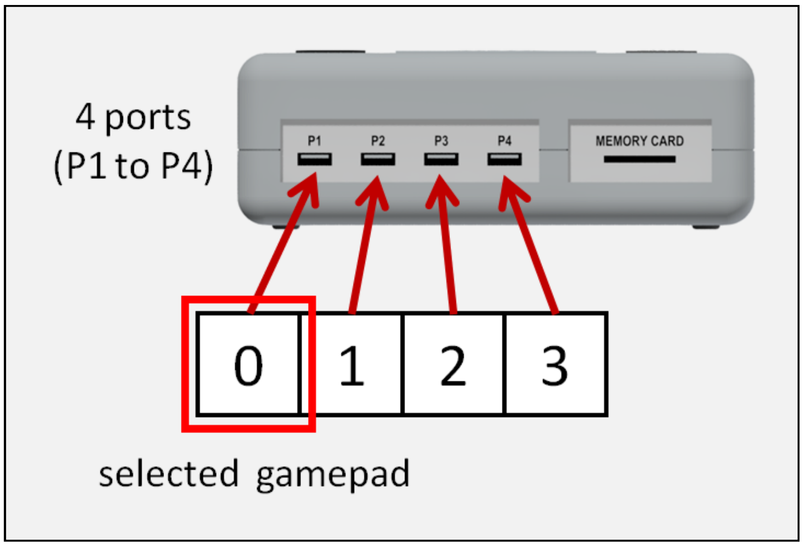
```
// read current date and time, in Vircon's native format
int VirconTime = get_time();
int VirconDate = get_date();

// translate date to human format
// (structure with year, month and day)
date_info HumanDate;
translate_date( VirconDate, &HumanDate );

// translate time to human format
// (structure with hours, minutes and seconds)
time_info HumanTime;
translate_time( VirconTime, &HumanTime );
```

A game could use these values to store the date and time in which a game was saved to a memory card, and show them to the player when the game is loaded.

Input controller

This controller constantly monitors the state of the gamepads connected to each of the 4 ports of the console. In addition to providing us with information about the d-pad and buttons, it will also tells us which gamepads are connected (as you can connect and disconnect gamepads at any time).

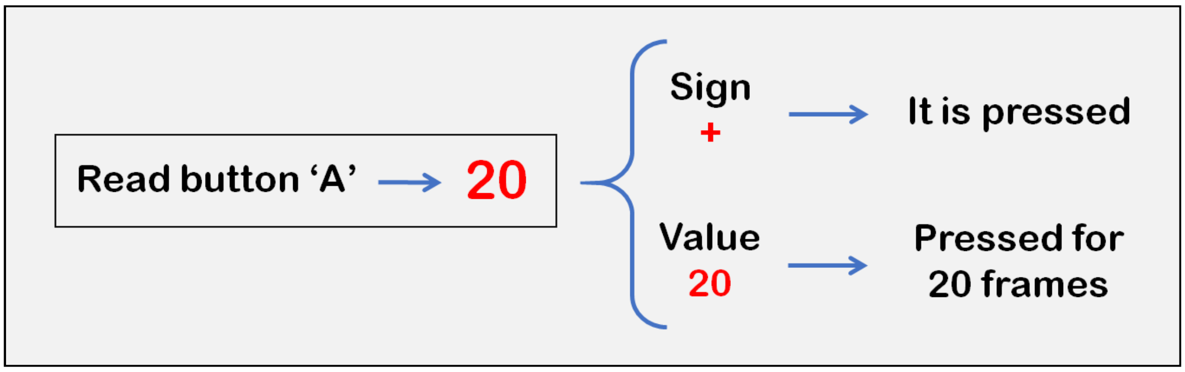
A concept that is important to know is that, although there are 4 ports (with gamepads connected or not), there is always only one of them that is selected in the gamepad controller. The 4 gamepads are always kept up to date, but our queries are always made on the currently selected gamepad. The gamepad selected by default is the first one.



From our program we can use select_gamepad() to change the selected gamepad. Once we choose a gamepad we can check what is being pressed on it with the gamepad_xxx functions, depending on the specific direction or button we want to read:

```
// we will read states for player 2's gamepad
select_gamepad( 1 );

// when pressing button A, shoot
if( gamepad_button_a() == 1 )
  Shoot( &Player2 );

// if left is currently pressed, advance backwards
if( gamepad_left() > 0 )
  Player2.x -= 3;
```

In the Vircon32 gamepad itself, the states of buttons and directions are a simple yes or no (pressed or released). But as you can see in the code, the query functions do not return a boolean, but an integer. It is that way because the gamepad controller gives us more information: we can use only the sign to know if a control is pressed, but also the value itself tells us how long it has been in the current state.



That is why in the example, we shoot only when the value is 1: the button has just been pressed this very frame. We don't keep firing for each pressed frame. On the other hand, when looking at the direction we are not interested in the time and only check its sign, since we do move the player every frame.

The state we receive from a button or direction always has a sign: the gamepad controller will guarantee that it can never be zero. This makes it easier for us when programming.

Updates of the state of gamepads

The controller does not work using events but, at the beginning of each frame, it automatically checks the presses of all the gamepads and updates their state. The program doesn’t need to do anything for this: only to make queries when needed.

Graphics chip (GPU)

The GPU is, apart from the processor, the most complex component of Vircon32. It is however a very simplified graphics chip. In fact, there are only 2 types of commands that the GPU can execute if the processor requests it:

- It can clear the screen with a constant color.
- It can draw on screen a region from some of the textures in the cartridge.

Commands with no parameters

The drawing commands that the GPU can do need some parameters. For example, if we are going to draw part of a texture on the screen we will need to know what region we are talking about and where on the screen we are going to place it. If these commands were to be used as a function, the processor would invoke a command by passing its parameters to it in a way similar to this:

```
DrawRegion( Region, X, Y )
```

However the GPU receives these commands as simple values on one of its ports, so the commands cannot receive parameters. What happens is that in the GPU there are internally a series of configurable parameters (which we will see) that the commands use when executing. Programs must set these values before sending a command. Thus, the above function would actually have these phases:

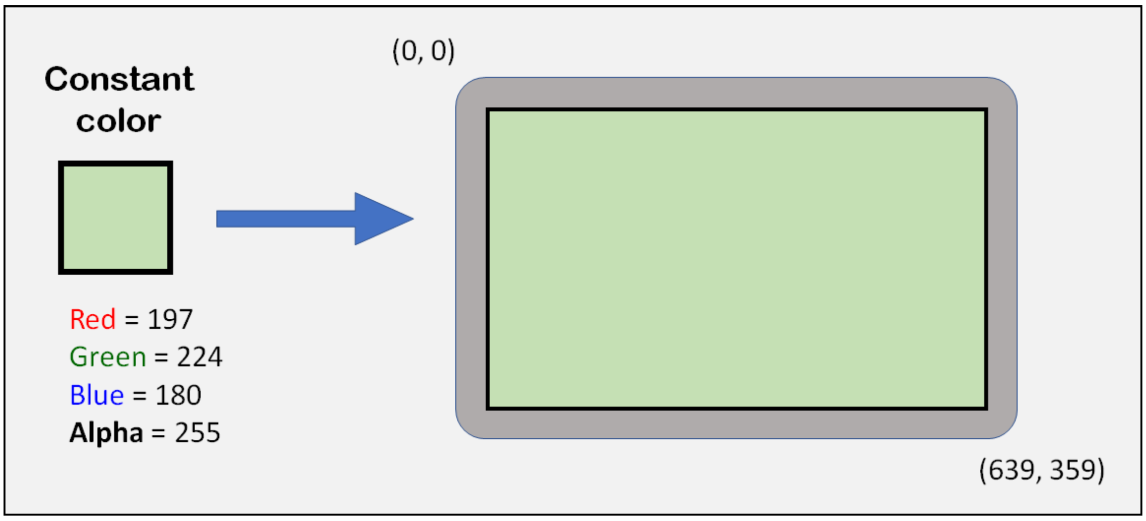
```
SelectedRegion = Region
DrawingPointX = X
DrawingPointY = Y
DrawRegion( )
```

In some cases the standard C libraries try to make our work easier and will allow us to use commands from the GPU (or other chips) as if they were functions. For example, to draw a region in the center of the screen the libraries allow us these 2 alternatives:

```
// option 1: with some parameters already integrated in the command
select_region( RegionCharacter );
draw_region_at( 320, 240 );

// option 2: command with all of its parameters separately
select_region( RegionCharacter );
set_drawing_point( 320, 240 );
draw_region();
```

Clearing the screen

This command is very simple: it only depends on one internal parameter, which is the clear color. The command will apply this color uniformly to the whole screen.



The color is a 32-bit value in RGBA format. Some predefined colors already exist in the C libraries. For example, to erase the screen in blue color we would do this:

```
clear_screen( color_blue );
```

In addition to using predefined colors we can also write their numerical value. But we must take into account that in that case its written representation is inverted (ABGR) since Vircon32 stores the bytes of each word in little-endian system. For example in a program the blue color would be written as FFFF0000 (and not 0000FFFFF). An easier option is to use functions from the C library to create colors:

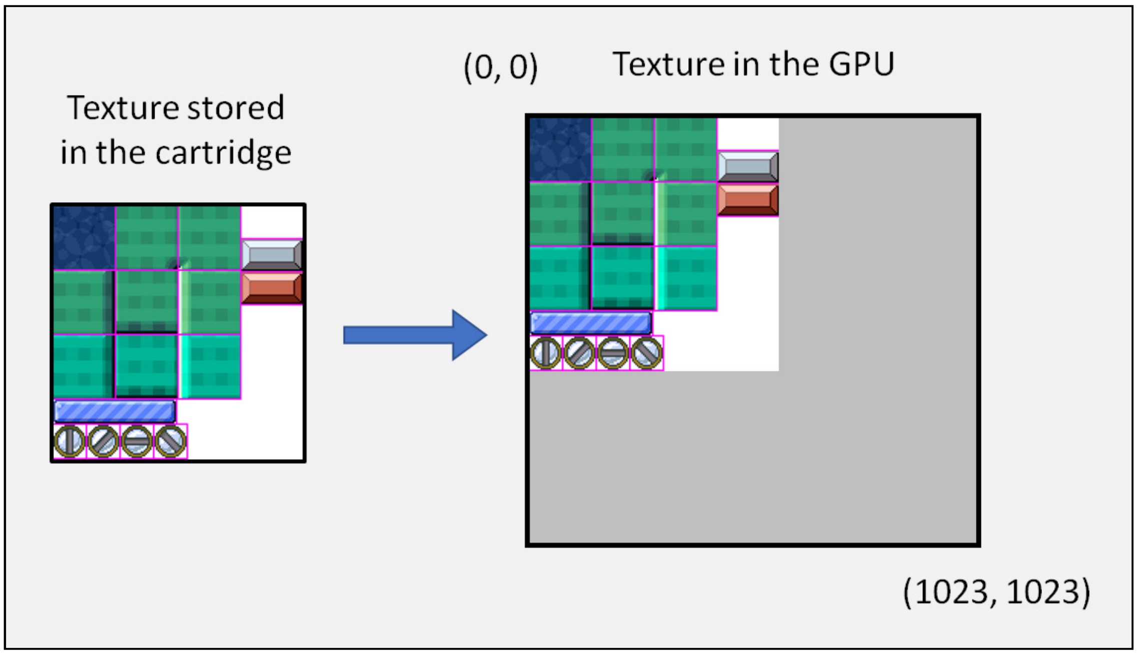
```
// we can also use make_color_rgba if we want to use alpha
clear_screen( make_color_rgb( 0, 0, 255 ) );
```

When clearing the screen, the alpha channel (opacity) can be used to make a partial clearing. If we make a clearing in black but with alpha = 128, we will darken the screen image but without making it completely black.

GPU textures

To describe the second type of GPU commands we need to know the concept of texture. As we know, the GPU works with a series of images stored on the cartridge. The GPU can only work with images of a standard size of 1024x1024 pixels, while each image on the cartridge can have a different size (up to that limit).

To compensate for this, when the cartridge is connected to the GPU, the cartridge images are stored in the video memory and positioned as shown in this image. Since both of them have a same first pixel (0,0), the coordinates of all pixels are preserved. Each of these images, already stored and adapted to the GPU, is what we call a texture.

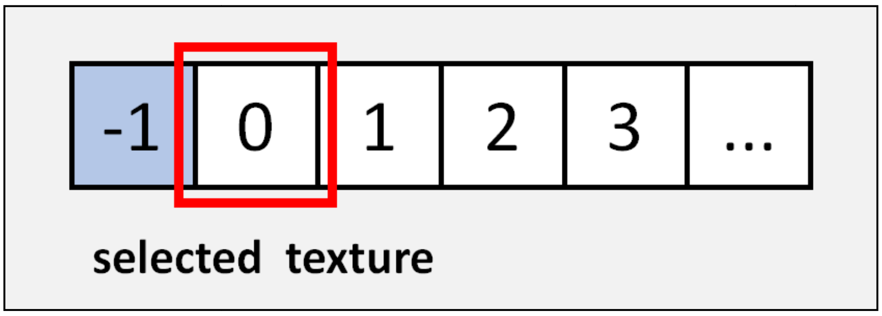


The texture list

The GPU has enough space for up to 256 textures on the cartridge, plus 1 additional texture for the BIOS. All these textures are handled as a numbered list: the BIOS texture (which always exists) is given the special ID -1. Textures coming from the cartridge are numbered with IDs from 0 to 255.

Numbering follows the same order as the images in the cartridge: the first one that exists will be ID 0, the second one ID 1, and so on. IDs that are not used by the cartridge, up to 255, will be left without an associated texture and the program will not be able to use them.

Among all the textures that exist, the commands are always applied on the texture that is currently selected on the GPU. By default the selected texture is the BIOS texture, as it is the only one guaranteed to always exist (a cartridge may not contain any textures).

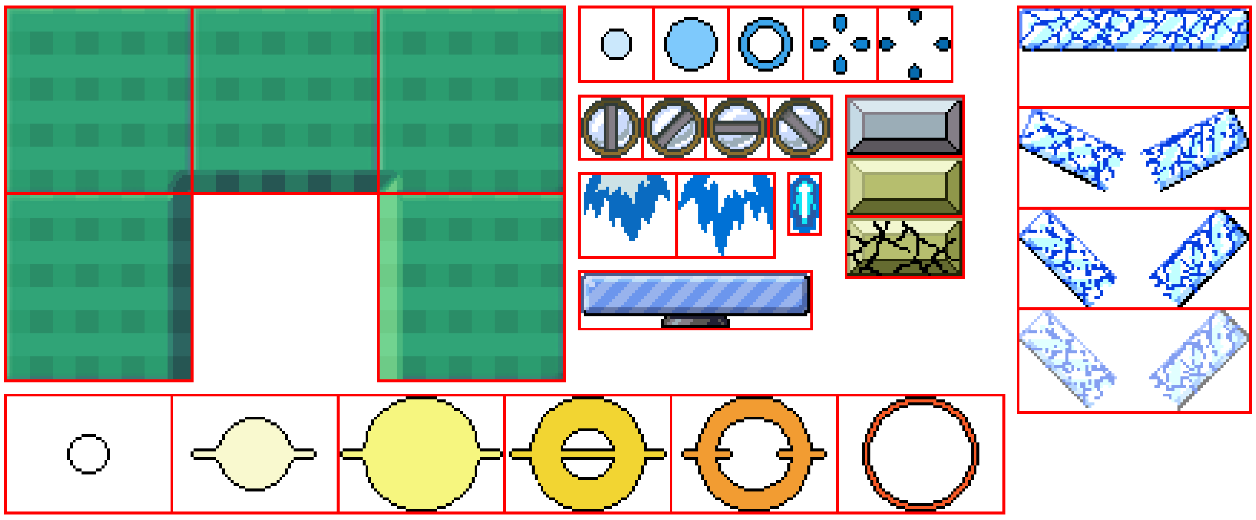


In our programs we can use select_texture() to change it. If our program has more than one texture it is usually recommended that we name them.

```
#define TextureLevels  0
#define TextureMenus   1

// (...)
select_texture( TextureLevels );
```

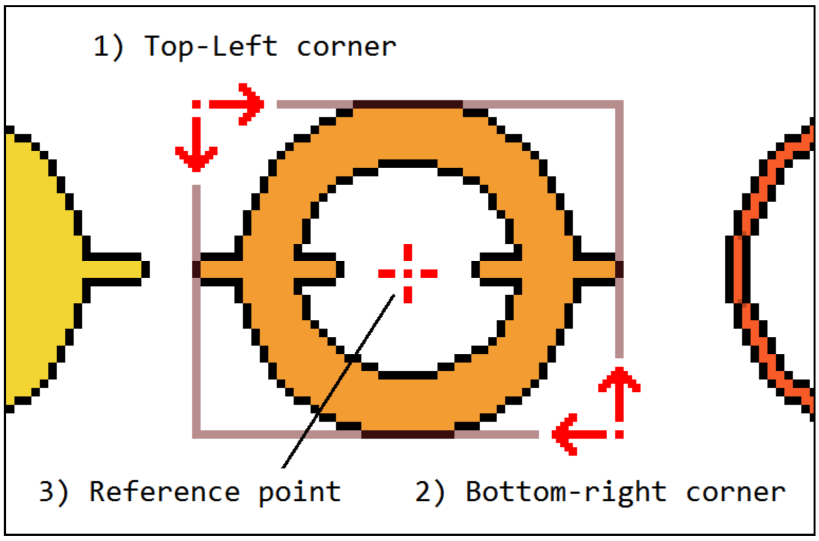
Regions of each texture

The textures used by the GPU are larger than the screen itself. For this reason the GPU will not driectly draw a texture but only regions of that texture. Usually several images from the game are stored together in the same texture, as in this example:

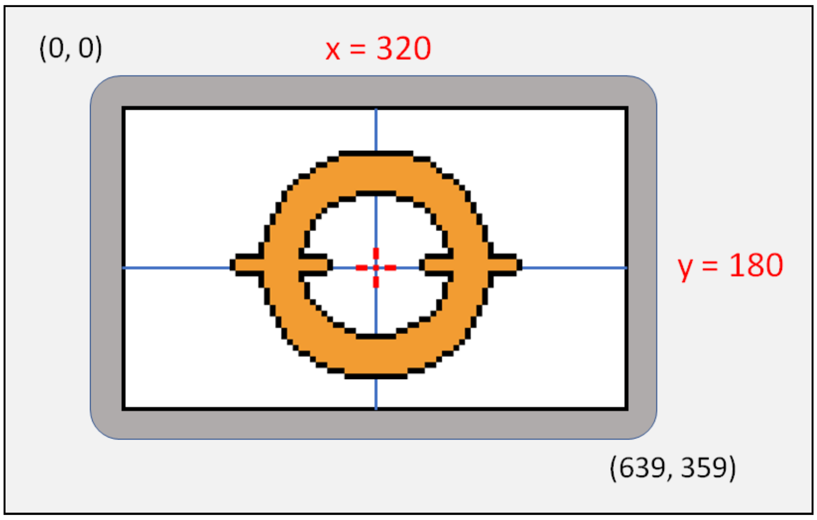


The borders separating regions are not really necessary for the console. But they make things easier for the programmer when creating and using textures.

In each GPU texture there exist 4096 regions, with IDs from 0 to 4095. The way to define a region is to use 3 points of the texture, as shown in the following image:



With the first 2 points we delimit a rectangle with the area to be drawn. The third point will be taken as a reference to position the image on the screen. For example, if we are drawing in the center of the screen (point 320,180), the reference point of our region will be made to match these coordinates.



Just as there is always a selected texture ID, the GPU also stores a selected region ID, which will be used by the commands. We can change the selected region using `select_region()`. However, we should note that the selected region ID is a global parameter of the GPU. There is not a different selected region ID saved for each individual texture.

Our C functions allow us to define regions in several ways, but the most basic one is `define_region()`, which applies the 3 points that define the region.

```
#define TextureLevels    0
#define RegionPlatform  50

// define a platform region
select_texture( TextureLevels );
select_region( RegionPlatform );
define_region( 0,0,  324,49,  112,49 );  // reference at bottom central point
```

As we can see here, unlike the IDs for textures, we can freely choose the region IDs we want to use. In each texture its 4096 regions always exist, and we simply configure them. We don't have to start at ID = 0, and the IDs don't need to be consecutive.

Commands to draw regions

As we have already seen, when the GPU draws a region on the screen it always does so according to its internal parameters. This means that it will use the texture and region that are currently selected, and to place the image it will also use other 2 parameters (coordinates x,y) that are the current drawing point. We saw in the code from previous examples that we can change it with `set_drawing_point()`, or use functions that already integrate it.

However, it's not only possible to draw the regions on screen as they are in their textures. The GPU can also apply transformations to them, in this case rotation and/or scaling. When drawing a region we can choose to apply or not each of these 2 transformations, and because of this we have a total of 4 commands. In this table we can see which C functions are used in each case:

Function	Scaling	Rotation
`draw_region()`	no	no
`draw_region_zoomed()`	yes	no
`draw_region_rotated()`	no	yes
`draw_region_rotozoomed()`	yes	yes

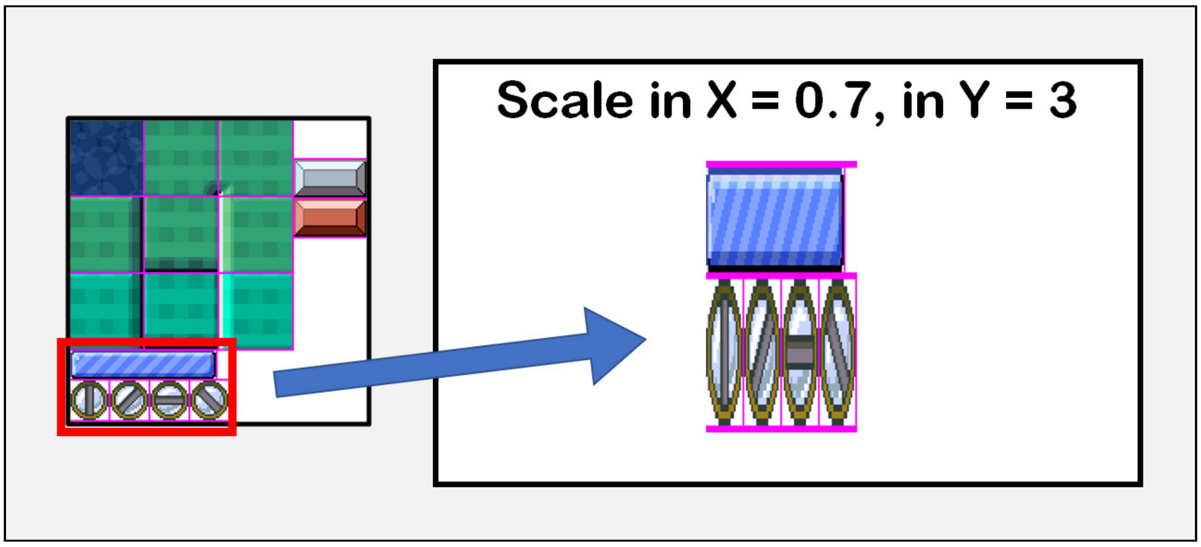
When we use a command that doesn't apply some transformation, the GPU will simply ignore the parameters associated to it.

Drawing with scaling

When we draw a region applying scaling to it, the command will use two additional parameters which are the X and Y scaling factors. These factors, in addition to being different from each other, can also be negative. The effect of a negative scaling is that the image is inverted along this axis. For example, we could write this:

```
// different scaling in X and in Y
select_region( RegionPlatform );
set_drawing_scale( 0.7, 3 );
draw_region_zoomed_at( 320, 180 );
```

And the result shown by the GPU would be like this:

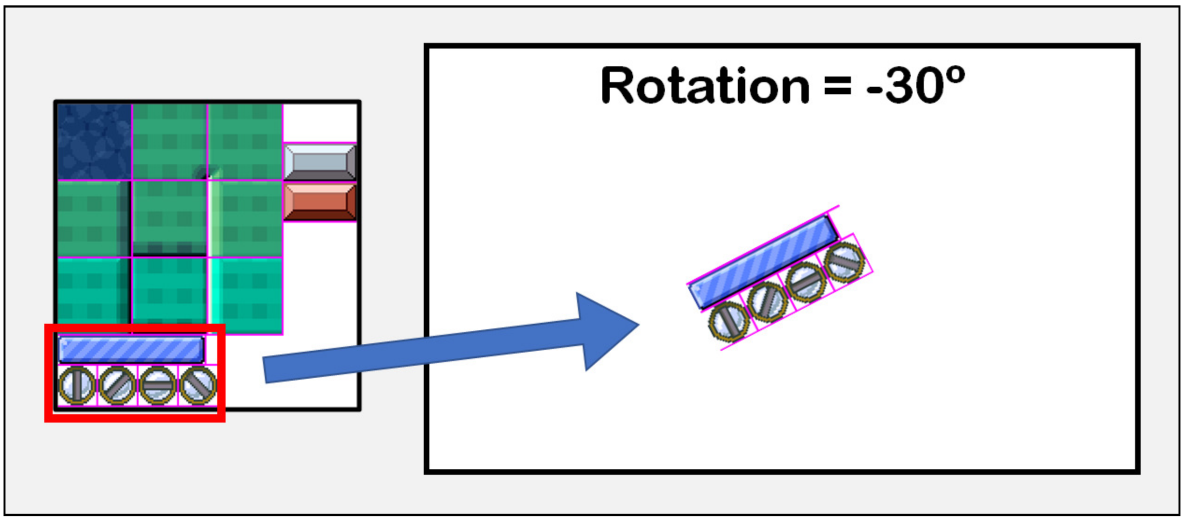


Drawing with rotation

If we draw a region using rotation, the command will use another parameter which is the drawing angle. Angles in Vircon32 are always measured in radians (180 degrees are equivalent to pi radians). Also, since the screen coordinates start at the top, the Y-axis is inverted. This makes angles grow clockwise and not counterclockwise. An example would be as follows:

```
// to rotate in anticlockwise direction the angle is negative
select_region( RegionPlatform );
set_drawing_angle( -pi / 6 );   // pi = 180º, luego -pi/6 = -30º
draw_region_rotated_at( 320, 180 );
```

And in that case the following would be displayed on the screen. The center of rotation is always the reference point of the region. That means: even with transformations, the reference point is always drawn matching with the GPU drawing point.



Modifying colors

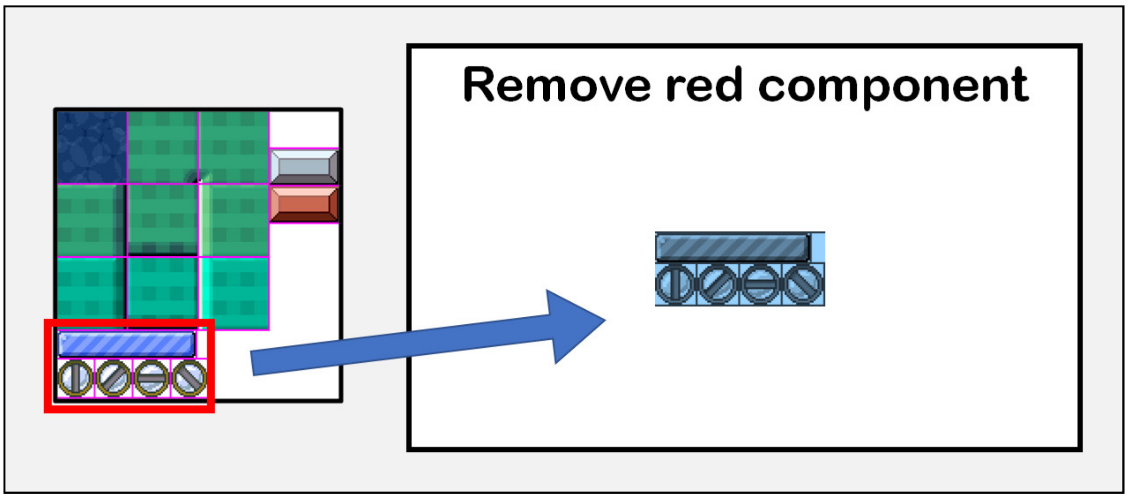
When drawing regions we can also do it applying a transformation to the colors. There is a parameter in the GPU called the "multiply color", which modifies the colors that are drawn with a multiplying effect over their RGBA components.

For those who know OpenGL, the effect is the same as glColor. That is, the neutral color is white (the default multiply color) and as the RGB components of the multiply color are darkened, the components of the region being drawn will be darkened, acting as a filter. The Alpha component also gets multiplied, which allows us to draw our region with different levels of opacity.

For instance, we could write the following code to eliminate the red component from the colors of our region.

```
// multiplying color with green and blue, but no red (in this case, cyan color)
select_region( RegionPlatform );
set_multiply_color( make_color_rgb( 0, 255, 255 ) );
draw_region_at( 320, 180 );
```

And visually, the effect we would see on screen with that code is this:



The multiply color is applied on all commands thatdraw regions (all of the 4 variants), but does not affect the clear screen command.

Color blending modes

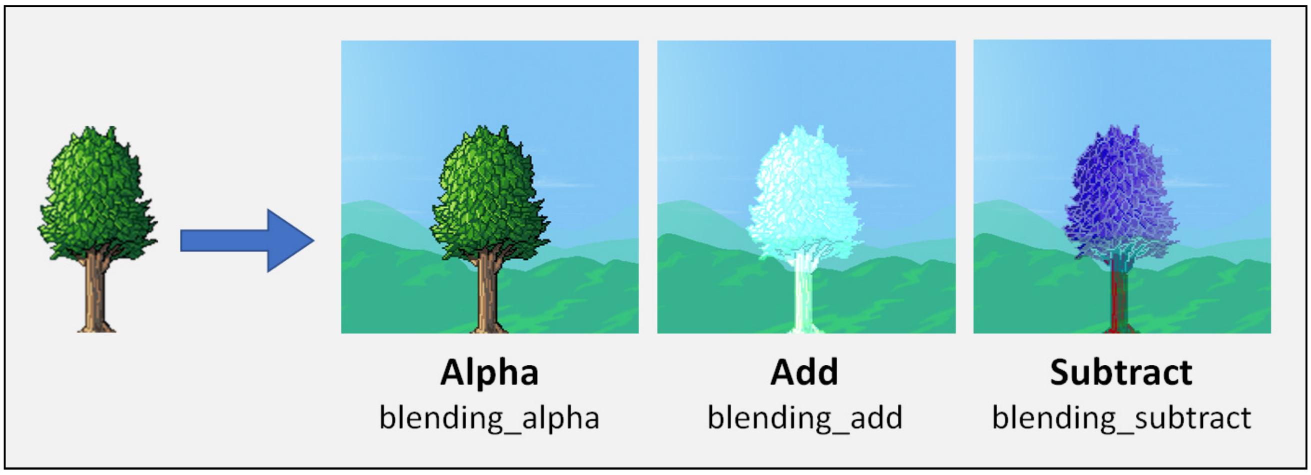
Color blending is the way an image is drawn on a background. To understand the effects that are achieved by varying the blending, the easiest way is to look at different types of layers in a drawing program like Photoshop.

The GPU in Vircon32 has 3 different color blending modes available .

- **Alpha mode:** This is the default blending mode, and it draws the region in the intuitive way: the colors are not modified, and it uses the Alpha component of the region as the degree of opacity. The effect is that we can have semi-transparent images that blend with the background on which we draw.

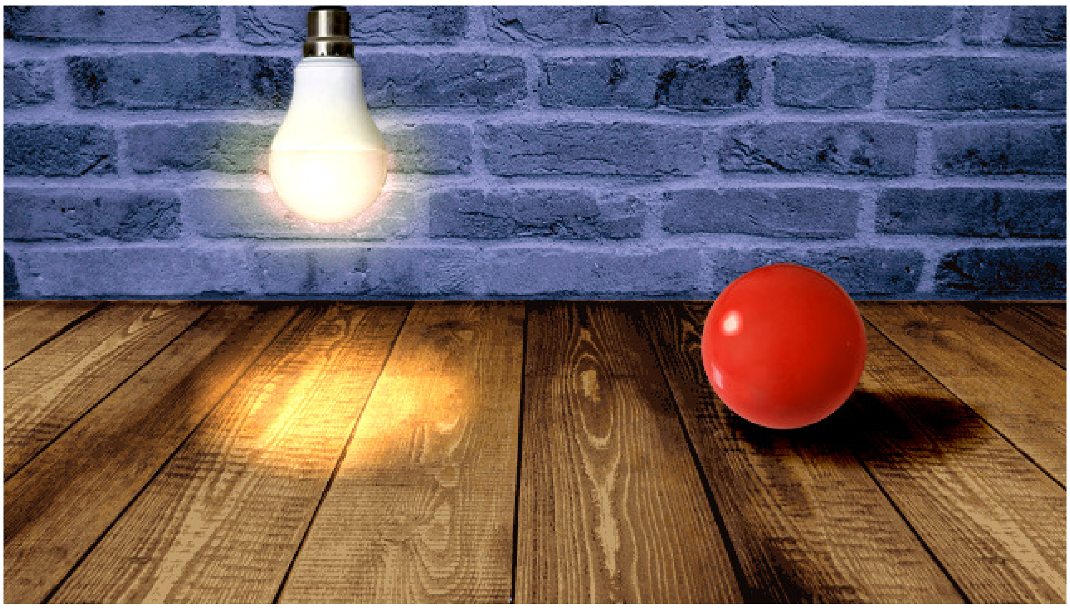
- **Add mode:** Also known as "Linear Dodge" in drawing programs. The colors of the region are added to those of the background, thus making a lighting effect. In this mode, black is the neutral color.

- **Subtract mode:** It is also known as "Difference" in drawing programs. The colors of the drawn region are subtracted from those of the background, allowing for shading effects. This makes black also be the neutral color in this case.



We can change the blending mode using `set_blending_mode()`, using the mode names shown in the image above.

If we combine the different modes we can get environmental effects like these:



The different blending modes, unlike color modifications, are not applied when drawing regions but also when clearing the screen.

GPU performance

The graphics chip would not be well defined if it didn't have a determined performance limit. For this GPU performance is based on the number of pixels that are drawn in each frame.

Each frame the GPU starts with a counter of remaining pixels equal to 9 times the full screen (which is equivalent to 1 full screen at 1080p resolution). Every time any command is performed, the GPU will calculate a very basic approximation of how many pixels will be consumed. If at any time the GPU does not have enough capacity in the current frame, it will ignore any request to perform commands until the next frame.

The calculation of the pixels consumed by each command is intended to be fast and simple, even if it is not very accurate. It's not needed to know this process to create games, but it may be necessary to get the most out of the GPU. This is the process:

- The "effective width" is calculated: it is the width in screen pixels of what will be drawn, applying the scaling in X (without sign) if applicable. If it is larger than the screen width, it is limited to 640 pixels.

- The "effective height" is calculated: analogous to the previous point. If it is greater than the height of the screen, it is limited to 360 pixels.

- The consumed pixels are calculated as: "effective width" x "effective height".

- Not all operations are equally expensive, so depending on the operation performed, the consumed pixels are modified by these factors:

  - When clearing the screen: -50%
  - When drawing regions with scaling applied: +15%
  - When drawing regions with rotation applied: +25%
    (if both effects are applied the factor is +40%)

As can be seen this calculation ignores the drawing position. This means that any parts of a region that fall outside the screen will also count as used pixels. Also, the current blending mode doesn't affect performance. They are all considered to be equally expensive.

Sound chip (SPU)

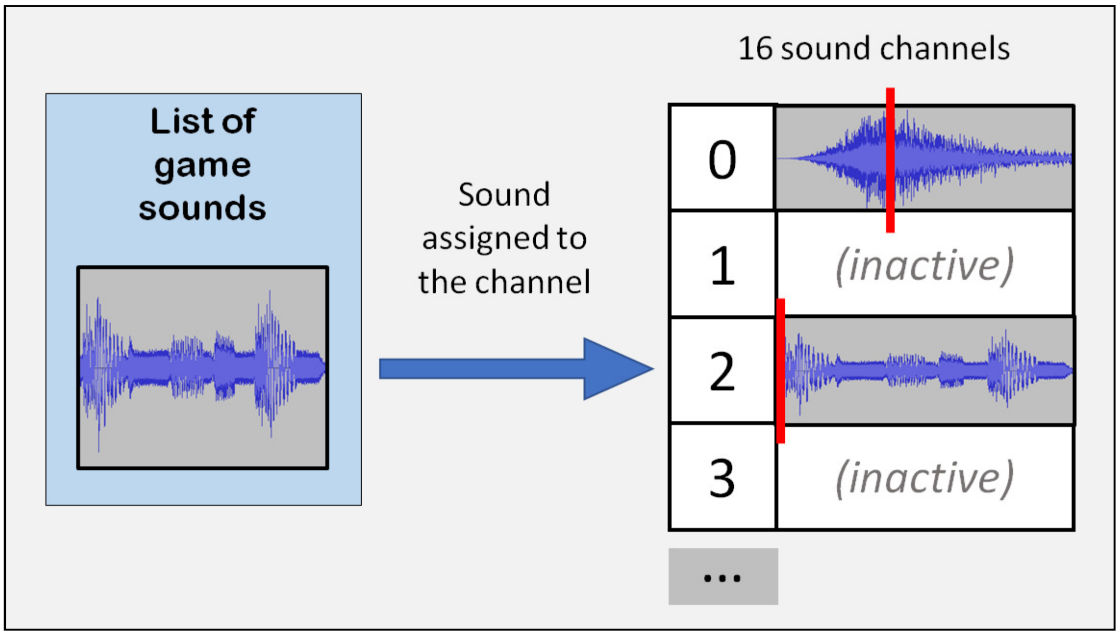
The sound chip has some features in common with the GPU, although it is simpler. Like the GPU, it has a numbered list of up to 1024 sounds for the cartridge (with their IDs in the same order, from 0 to 1023), plus ID = -1 reserved for the BIOS sound.

In that list of sounds there is always a selected sound that we can change using `select_sound()` with the corresponding ID. The default sound selected is the BIOS sound, as it is the only one that always exists (a cartridge may not have any sound).

The difference with the graphics chip is that sounds are used as they are. They don't need to be of a standard size, and we don't play them using parts of the sound as it happened with texture regions. In many cases we won't need to configure anything in sounds.

Playback channels

The sound chip has 16 channels available in which sounds can be played. They are numbered with IDs from 0 to 15. There is always a channel selected in the SPU (by default ID 0), which we can change with `select_channel()`. When a channel is selected we can assign to it a sound from the list (using its ID) and start playing.



Each sound channel is independent from the others, and all of them can be playing any of the sounds at the same time. But we must keep in mind that if many sounds are playing at the same time, they will saturate the speaker signal unless we reduce their playback volumes.

A sound channel can be in any of these 3 states: playing, paused and stopped. When a channel has no sound assigned or has finished playing, its state is always stopped. The state of a channel can be queried with `get_channel_state()`:

```
// wait until a sound finishes playing
while( get_channel_state( ChannelSoundEffects ) != channel_stopped )
  end_frame();
```

Sound commands

There are 6 sound commands in the Vircon32 SPU. On one hand we can operate on a given channel, with commands to play, pause or stop. With the other 3 commands, the SPU allows us to stop, pause or resume all channels at the same time.

We have more than one option to play a sound from C. We can simply use play_sound() and let it search for a free channel for us. However in some cases (for example, for background music) we may want to choose a particular channel to play. In code, the 2 ways are used like this:

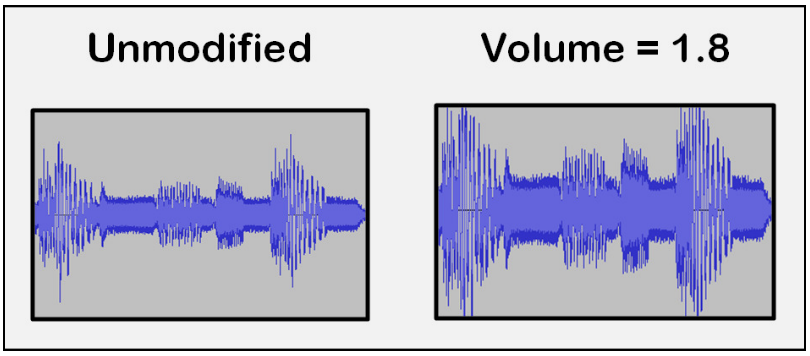
```
// reserve channel 15 for background music
play_sound_in_channel( Level1Music, 15 );

// play a sound effect in any other channel
// (but we need to know which one to change its volume)
int UsedChannel = play_sound( Level1Music );
set_channel_volume( UsedChannel, 0.25 );
```

The commands affecting all channels can be useful in situations where a game changes context. For example, we can pause all sounds in a pause menu, and resume them when we exit the pause menu like this:

```
// when entering the pause menu
pause_all_channels();

// show the menu itself
ShowPauseMenu();

// when going back to the game
resume_all_channels();
```

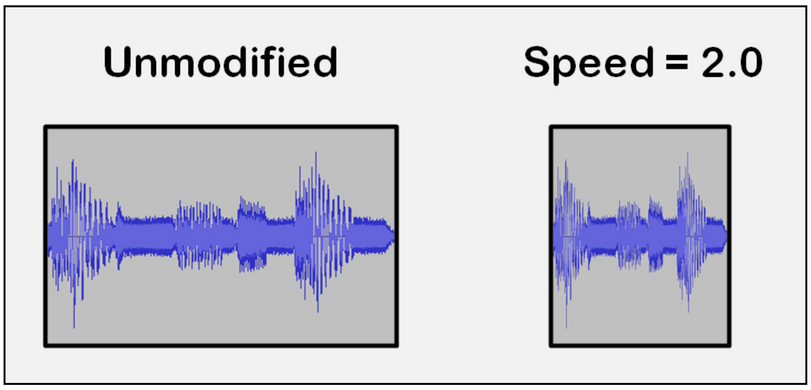
Playback volume

In sound playback channels the volume can be modified with set_channel_volume(), in range from 0 to 10. The default volume is 0.5. The volume of a channel does not change unless we change it, and it will remain the same even if it plays different sounds.



Playback speed

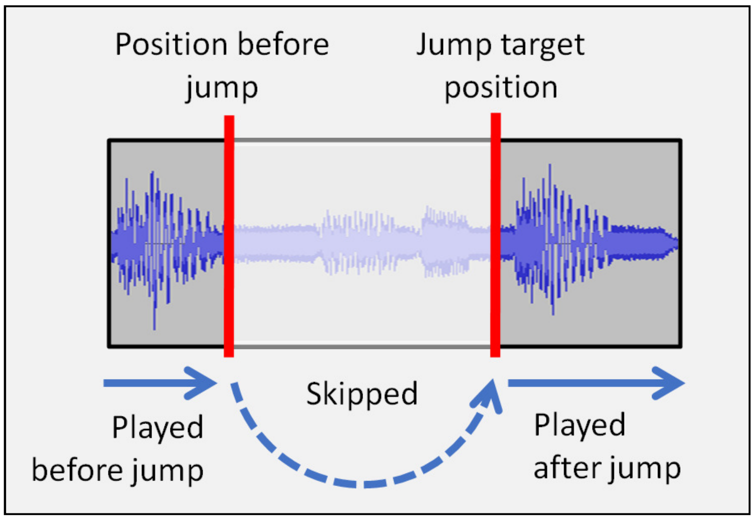
Channels also allow changing the playback speed, using the `set_channel_speed()` function. Playing at a different speed also changes the pitch of the sound. The range is from 0 to 100, and the default speed is 1. This effect can be useful to simulate the different notes of an instrument.



Same as with volume, the speed of a channel will remain the same if we don't change it.

Playback jumps

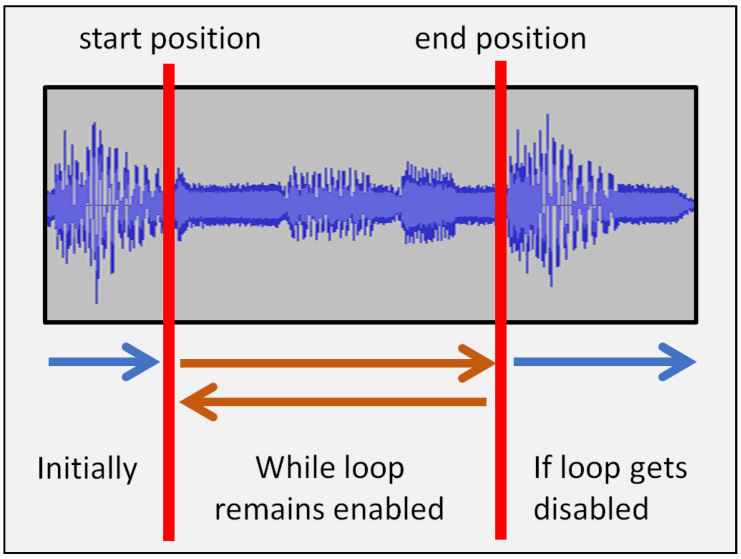
Channels play their assigned sound by continually advancing their playback position along the samples of that sound. Channels allow this playback position to be modified using the `set_channel_position()` function, resulting in jumps. For example, in the case of a jump forward, playback of a sound would happen as shown in the image below.



Playing in loop

SPU channels allow us to play sounds in a loop. This option can be enabled at the channel level, with `set_channel_loop()`, but also each of the GPU sounds themselves has a parameter to determine if they should be played looped. This is controlled with set_sound_loop(). Whenever a channel starts playing any sound, the channel will automatically turn playback loop on or off to adapt to that particular sound. By default all sounds and channels have looping disabled.

By default the loop is made for the whole sound, but we can make partial loops by marking start and end points. The playback will then be done as in this image:



To achieve this we will need to delimit within that sound the specific region that should be looped. We can do this as in the following example:

```
// configure this sound to be played in loop
select_sound( SoundCarEngine );
set_sound_loop( true );

// configure its looped region (positions given in samples)
set_sound_loop_start( 1348 );
set_sound_loop( 12507 );

// upon play command, channel loop is enabled automatically
int EngineChannel = play_sound( SoundCarEngine );

// (...)

// disable loop to allow the sound to end
select_channel( EngineChannel );
set_channel_loop( false );
```

RAM memory

RAM memory is a passive component, so there is not much to know about it. Its size is 16 MB, although since the minimum unit is the 32-bit word, it is more appropriate to measure its size as 4 MW.

The BIOS

The BIOS is an internal software that always exists in the console. When the console starts it will show the Vircon32 logo, and then check if there is a cartridge connected. If there is, it will transfer control to it, and if not it will show a warning screen.

If at any time a hardware error occurs, the BIOS will receive the control again to handle it and when finished it will stop execution.

A Vircon32 BIOS always contains exactly 1 texture and 1 sound, which have reserved ID number -1 on both GPU and SPU. One of the elements included in that texture is a 10x20 pixels text font to allow writing text on the screen.

Cartridge controller

This controller allows the console to access the contents stored in the cartridge (program, textures and sounds). All of this happens automatically, and we don't need to do anything related to it in our program.

In addition, the controller can also be queried for basic information about the cartridge in the console: whether it is connected, how many textures and sounds it contains, and the size of the program ROM. This information is not normally needed from a game program, which already knows the elements it uses. So, it would only make sense to ask the cartridge controller for this information from test programs.

The BIOS does indeed need to check if a cartridge is connected during console startup. After that no more checks are necessary: on Vircon32, when the console is turned on, the cartridge input is locked and no cartridge can be inserted or removed without first turning off the console.

Memory card controller

A memory card, when it is connected, is accessible to the CPU as a read/write memory region. Its size is 1 MB, although since the minimum unit is the 32-bit word, it is more appropriate to measure its size as 256 KW. In our C programs we can check if there is a card in the console by calling the function `card_is_connected()`. It is important to make this check, since trying to access the card will cause a hardware error if there is none connected.

The signature of each game

A program can access the memory card just like any other memory region, but since it may contain game saves (even from other games), it is recommended to make sure of what the card actually contains before attempting to use it.

For this, the common practice is that the first 20 words of the card are used as a "signature" of the game, that is, a set of known values that identifies whether it is our game that saved the data (if the signature matches the expected one) or it was another game. Thus, if when checking the card our game detects an unknown signature, we prevent it from trying to load a game that will be incompatible and may cause errors.

We can work with signatures using `card_read_signature()`, `card_write_signature()` and `card_signature_matches()`, as shown in the following example:

```
// we create a signature for our game
game_signature GameSignature;
memset( GameSignature, 0, sizeof(game_signature) );
strcpy( GameSignature, "MY TEST GAME" );

// (...)

// when saving our game
card_write_signature( &GameSignature );
SaveGameData();


// (...)

// when loading our game
if( card_signature_matches( &GameSignature ) )
  LoadGameData();
```

A memory card can also be empty, if it has not yet been used after being created. A program can assume that a card is empty if its signature contains only zeros. In the standard library we have `card_is_empty()` to check this.

Once we have verified that we can use the card, we can read and write data to it using functions `card_read_data()` and `card_write_data()`. For example, the load and save functions in the previous example might look like this:

```
// the data in our game
game_signature MySignature;    // type already defined in "memcard.h"
GameState CurrentState;        // a structure defined by us

// (later in our program)
void SaveGameData()
{
    // our data is written just after the signature
    int OffsetInCard = sizeof( game_signature );
    card_write_data( &CurrentState, OffsetInCard, sizeof( CurrentState ) );
}
```

```
void LoadGameData()
{
    // our data is read just after the signature
    int OffsetInCard = sizeof( game_signature );
    card_read_data( &CurrentState, OffsetInCard, sizeof( CurrentState ) );
}
```

Random number generator

This generator starts from an initial number (a "seed"), and from it it's able to produce a sequence of pseudo-random numbers every time the CPU requests it. From the same seed, the generators will always create the same sequence, even on different Vircon32 consoles.

We can change the seed with `srand()`, and extract numbers from the sequence with repeated calls to `rand()`. In C programs it is common to create a seed using the current time to make the generated sequence different for each game session.

```
// we initialize a new sequence of numbers
srand( get_time() );

// simulate the throw of a dice: from 1 to 6
int DiceThrow = 1 + rand() % 6;
```