Vircon32

32-BIT VIRTUAL CONSOLE

System specification

Part 2: Console architecture

Document date 2022.12.15 Written by Carra

What is this?

This document is part number 2 of the Vircon32 system specification. This series of documents defines the Vircon32 system, and provides a full specification describing its features and behavior in detail.

The main goal of this specification is to define a standard for what a Vircon32 system is, and how a gaming system needs to be implemented in orded to be considered compliant. Also, since Vircon32 is a virtual system, an important second goal of these documents is to provide anyone with the knowledge to create their own Vircon32 emulators.

About Vircon32

The Vircon32 project was created independently by Carra. The Vircon32 system and its associated materials (including documents, software, source code, art and any other related elements) are owned by the original author.

Vircon32 is a free, open source project in an effort to promote that anyone can play the console and create software for it. For more detailed information on this, read the license texts included in each of the available software.

About this document

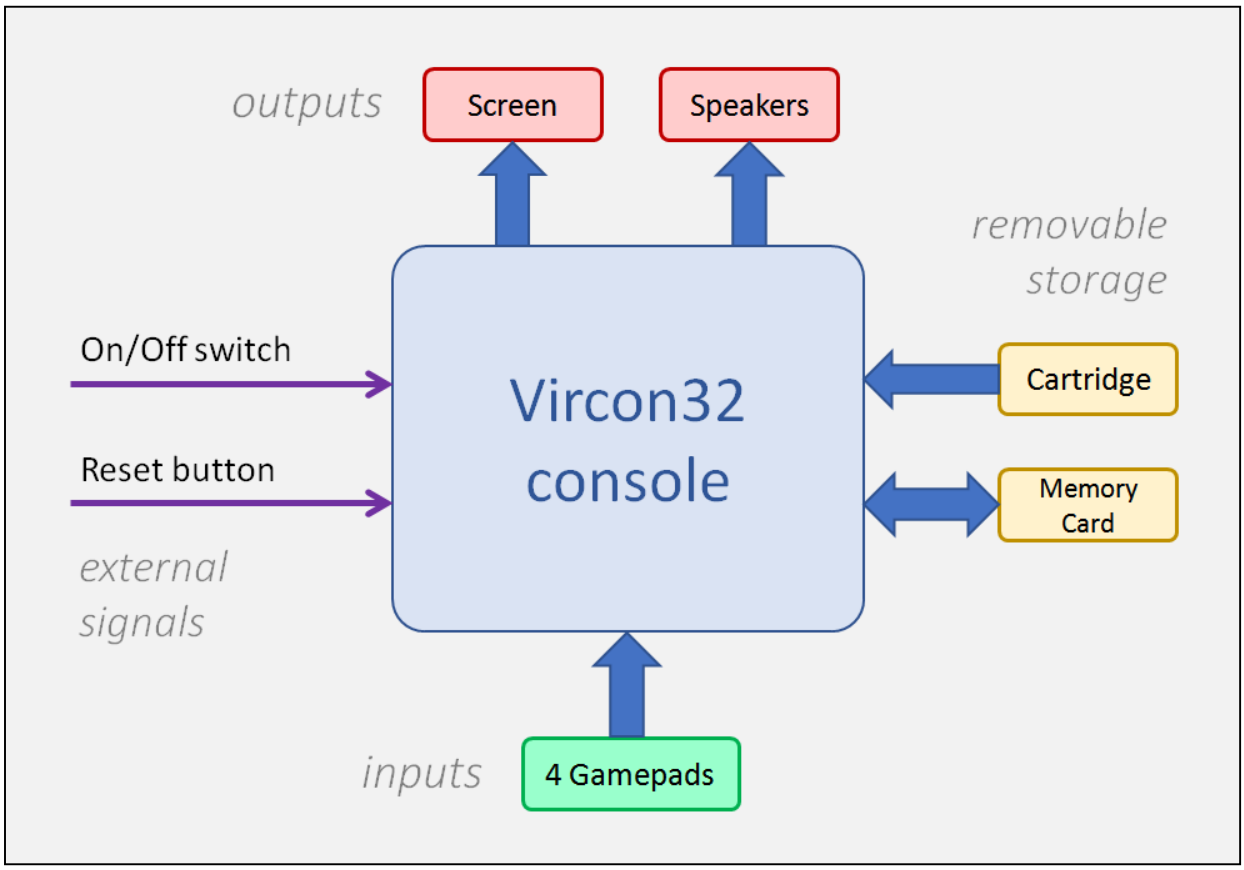
Summary

Part 2 of the specification first identifies all console components and their assigned tasks. Then, it explains the ways in which the different chips can communicate and interact so that they integrate to work together as a console.

Here we will also cover some mechanisms and data properties that affect the functioning of all console components.

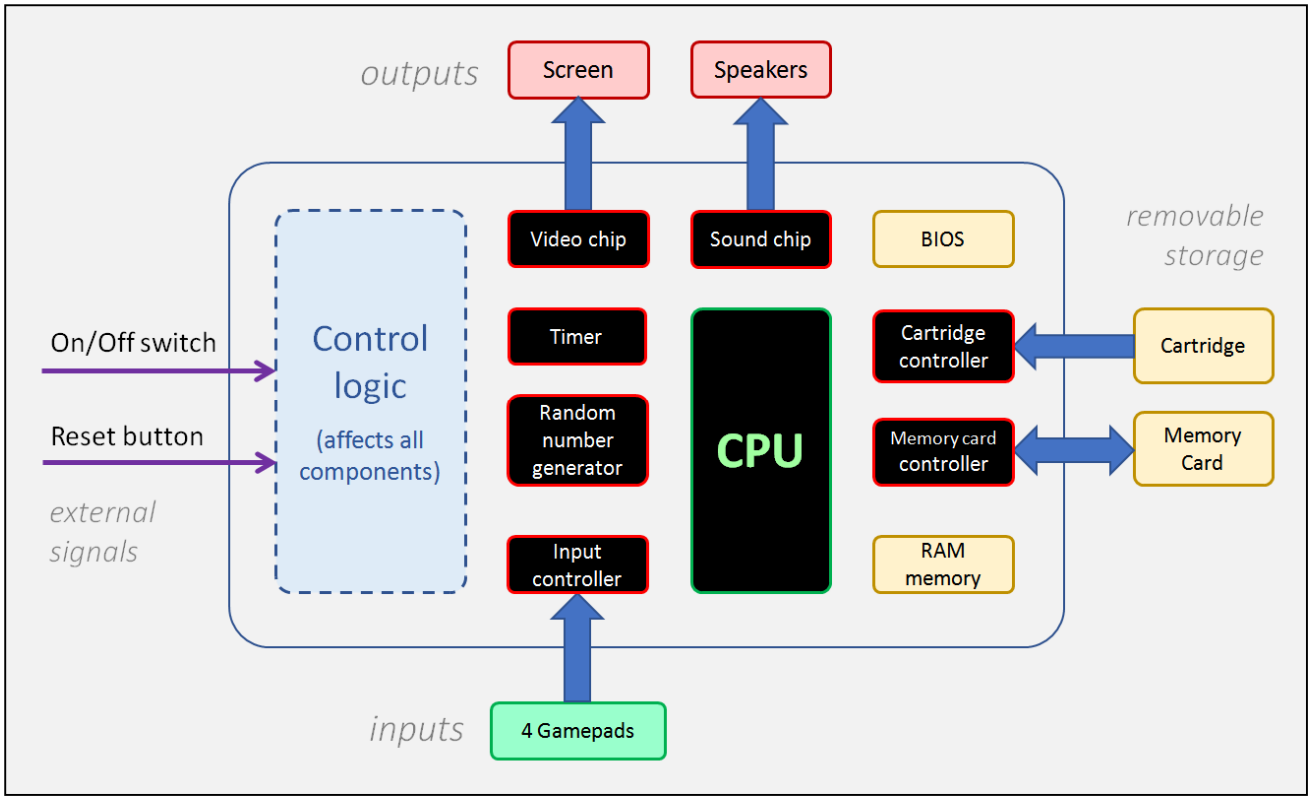
1 Console internals

After reading the introductory sections in part 1, we can already deduce that the basic, system-level architecture of Vircon32 follows this diagram:



The diagram uses a simple color code to identify each type of external components that may be connected to the console: there are devices for input and output, and devices that connect external memory. Any of these devices can be connected to the console or be absent, since all of them are optional. When they are not present the console will still work, although it will obviously lack their added functions.

Part 1 already provided a general overview of all those external devices, so we will now temporarily omit the console's environment and have a look into the console itself. To step into the next level of detail we can take that same previous diagram and expand it so that the console is no longer a black box. In this second diagram we can distinguish its internal components.



For clarity this diagram is omitting the data buses that communicate these internal console chips (which are presented in later sections). As for the "Control logic" block shown, it is a simpler way of taking into account mechanisms that span the whole console. In this case said mechanisms are control signals that are received by all components of the console, and transmit events related to power, reset and timing.

1.1 Console chips

The main components within the console are the different chips responsible for performing the console functions. The other console components are secondary, being present mostly to provide support for these chips. The chips in the Vircon32 console are:

- **Processor (CPU):** It is the main chip and the one running the program. The processor executes program 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 of the 4 gamepads are connected at each moment. It allows programs to query the state of their d-pads and buttons.

- **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. When present it enables the CPU to access that card's memory.

- **Graphics chip (GPU):** This chip accesses the images present in the cartridge's video 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.

- **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. Its size is 16MB or, in 32-bit words, 4MW.

- **BIOS:** It's a read-only memory, similar in structure to a cartridge. It contains an internal software that controls the console startup and reacts to possible hardware errors. It also includes a text font to allow writing on screen.

1.2 Communication buses

The described console chips cannot properly function in isolation: they must be able to communicate and interact. The detailed architecture diagram purposely omitted the 2 communication buses connecting the different chips, which are:

- **Memory Bus:** This bus allows different devices to make their internal memory accessible to the CPU. It joins all of their local address ranges into a map of global memory addresses that can be read from and written to in the same manner.

- **Control Bus:** The purpose of this bus is to let the CPU access a set of "control ports" exposed by each chip. By reading and writing from those ports, the running program can control the funcions of those chips.

These communication buses and their connection configurations will be described in detail in chapter 3.

1.3 Control signals

There are some global, console-level events that must be made available to all of the console chips, to ensure that all of them can react to the event when needed. To this end, the console uses the following control signals:

- **Power:** This is not really a signal in itself. A physical console would need to transmit power to all components, controlling it with a button or switch. In the case of Vircon32 this can be reduced to simple On/Off methods of the emulator itself.

- **Reset:** The Reset signal can be triggered directly by the user, but it also happens every time the console is powered on. It is transmitted to all components in order to make them revert into a known initial state.

- **New frame:** This signal is triggered every time the Vircon32 video signal is refreshed, which happens exactly at 60 fps. Due to the way Vircon32 handles timing (see next section), this also needs to be transmitted to some chips in order for them to take actions.

- **New cycle:** Cycles are the minimum time unit within Vircon32. The cycle signal is meant to act as clock signal for the CPU and other components. Vircon32 cycles happen at a speed of 9 MHz.

2 Timing control

All operation of the Vircon32 system follows a global timing scheme. The Timer is responsible for triggering "new frame" signals at 60Hz. Then, within each frame, it will trigger 150,000 "new cycle" signals. This results in 9,000,000 cycles per second.

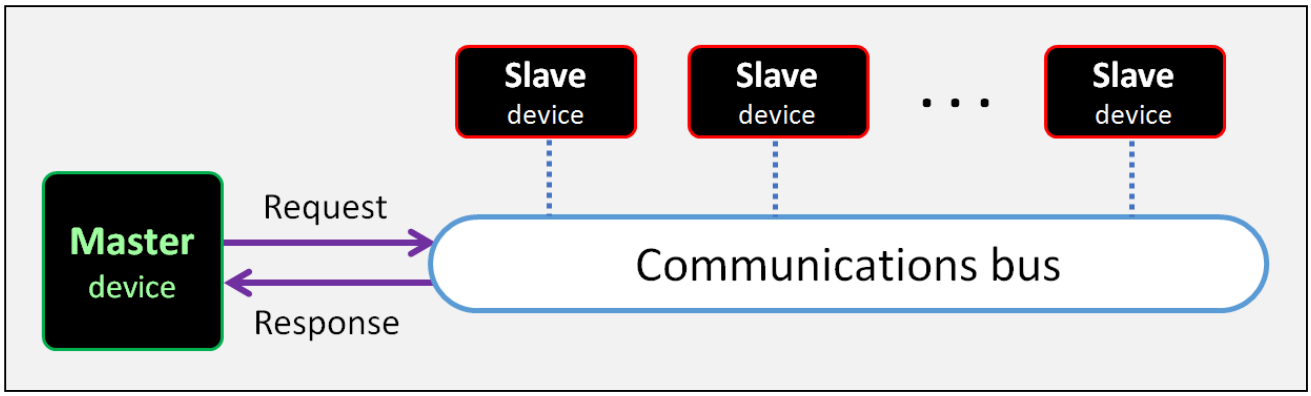
Every action performed by Vircon32 components is always (directly or indirectly) triggered as a reaction to a signal from the Timer. Since these same timing signals are shared by all components, overall timing is greatly simplified because no inter-component coordination is needed.

2.1 Timing within a frame

A Vircon32 system is required to guarantee that frames begin exactly at 60Hz intervals. However, cycles within a frame do not have to happen at regular intervals in real time. Any particular implementation can time cycles within the frame as it sees fit, as long as order is preserved and all 150,000 cycles happen before the next frame. It is valid, for instance, to run all cycles as fast as possible and then wait. As a result, programs cannot assume that elapsed frame cycles are a way to accurately measure time.

3 Communication buses

Chips within the Vircon32 console use buses as their mechanism for communication. These buses follow a master-slave model, in which there is a unique master device that controls communication and sends requests. The rest of connected components (the slaves) behave passively and are restricted to answering the requests they receive. In both console 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.

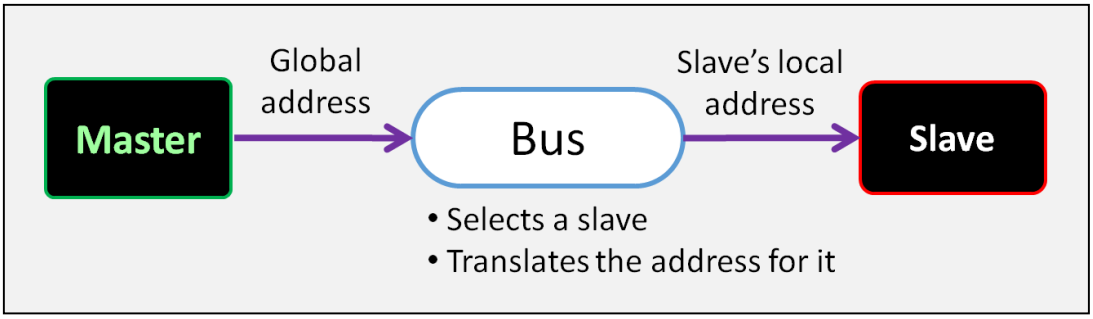


3.1 Address space

Each of the connected slaves will expose a series of 32-bit registers, that are all made available to the master device. Each of those registers is identified within their host device with an integer value (their local address).

When multiple slaves are connected, the bus is in charge of translating between local adresses for each particular slave, and global addresses that the master can use as part of a unified global address space.

To make a request, the bus master has to select a target register by providing the bus with the desired target address. The bus will then select the correct slave device to which the request must be forwarded, and do the same process backwards when said slave sends its response.



3.2 Request mechanism

Fot any of the buses, there are the same 2 types of requests the CPU can make as master:

- **Read request:**
  The CPU requests to read the value stored at a specific global address. When the request is successful, the requested value is provided.

- **Write request:**
  The CPU requests to write a given value at a specific global address. When the request is successful, the value is written at the requested address.

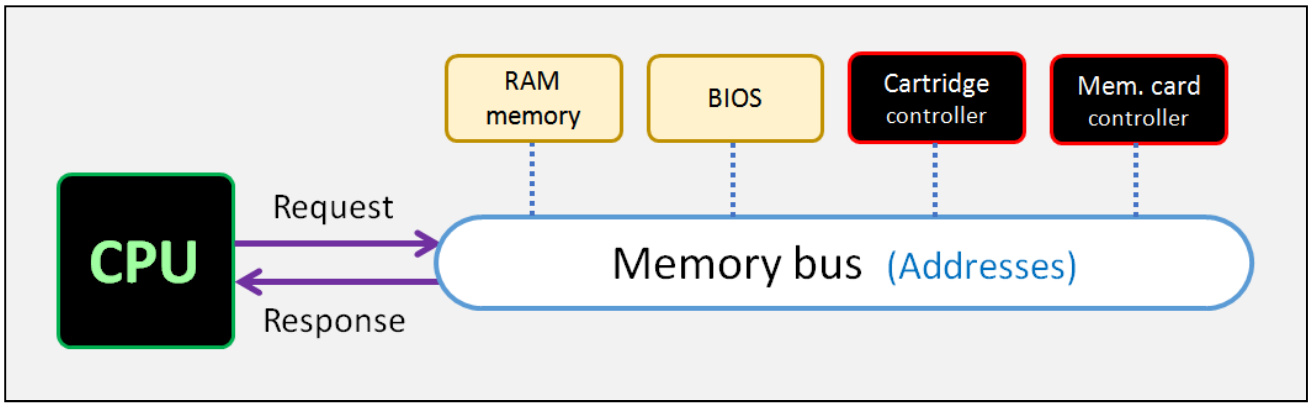
In both cases the response to the CPU must include a boolean value indicating whether the request was successful, or failed.

A request can fail for one of 2 reasons. The first is that the request may refer to a non-existent address (that is not exposed by any of the currently connected devices). If the address does exist, a second cause for failure happens if a slave device is unable to perform that particular request. A typical example is trying to write on a read-only memory such as a cartridge.

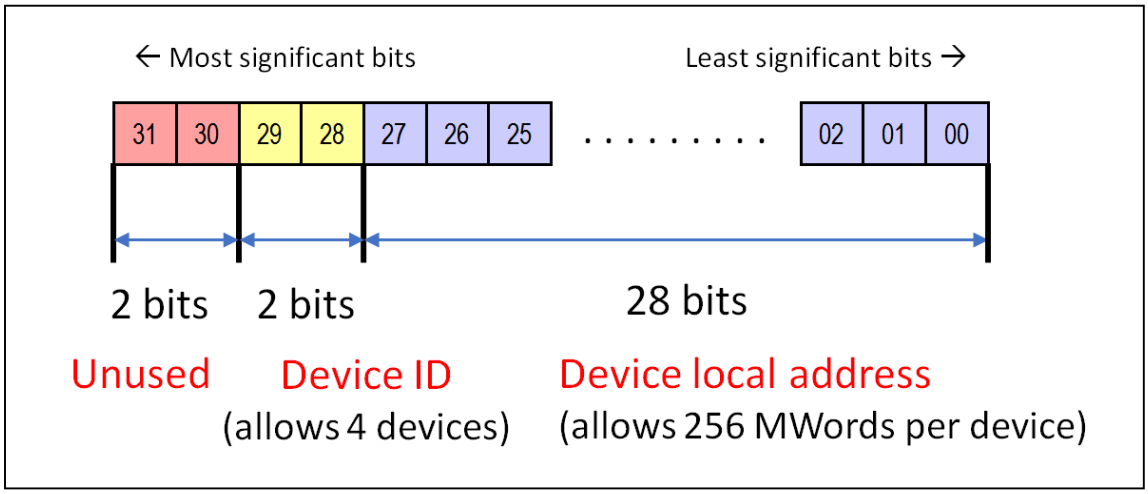
4 Memory bus

This is the bus to which all devices that have memory (be it RAM or ROM) will connect. By connecting to this bus, its memory space receives a global address range and its contents become accessible by the CPU. Requests made by the CPU to read or write an address in this bus, when successful, have the effect of reading or writing that particular word in the target device's memory.

The connection schematic for the memory bus is as shown in this diagram. Note that, even though all 4 components are always connected, there may be no accessible memory in the controller chips if the cartridge or memory card are not connected to the console.



The memory bus uses 32-bit addresses, and the process it uses to convert between global and local address spaces is to break the address in fields as shown here:



The 4 connected devices are mapped to memory addresses using the following device IDs:

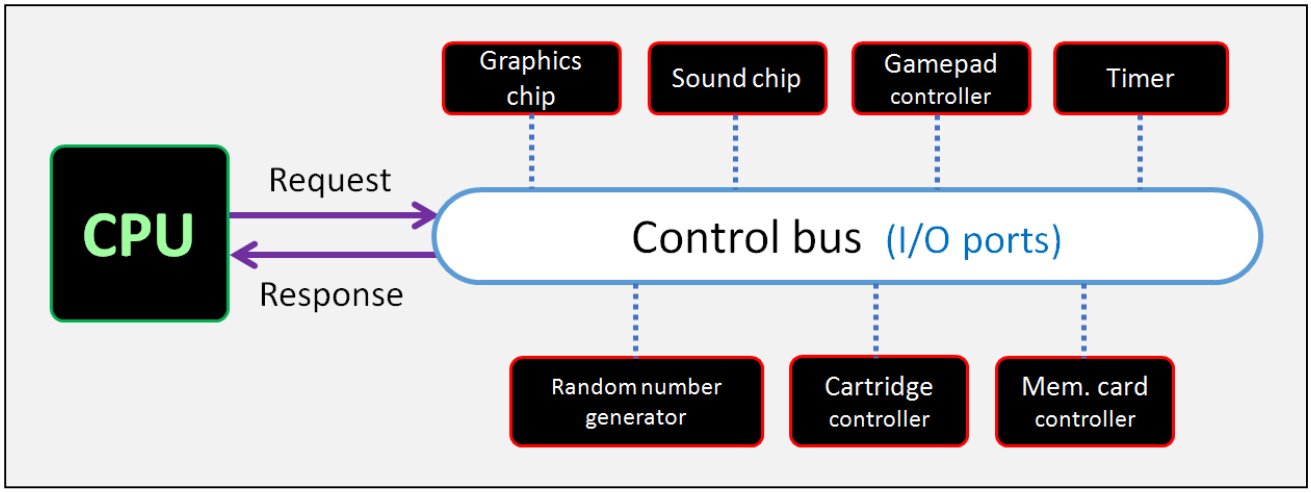
Device ID	Connected device
0	RAM memory
1	BIOS program ROM
2	Cartridge program ROM
3	Memory card RAM

5 Control bus

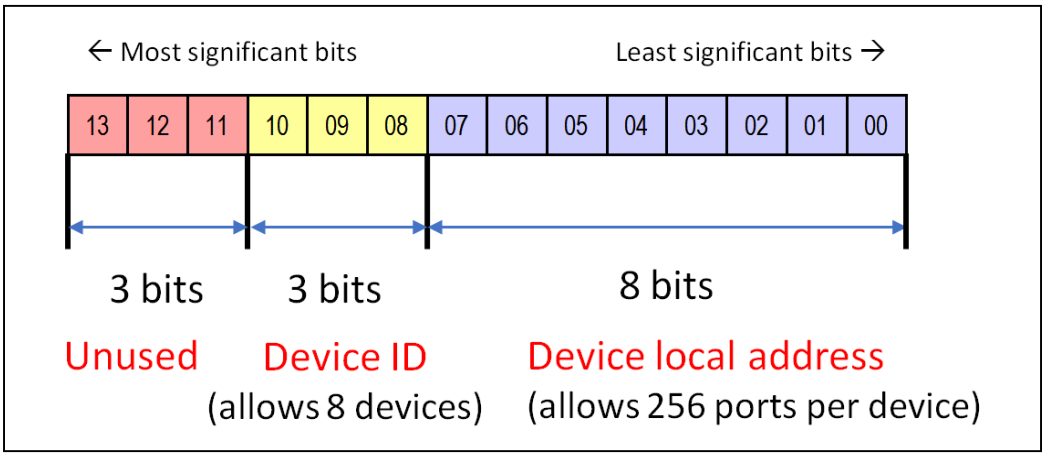
This bus allows the processor to access a series of I/O ports in each of the connected chips. Each of these ports is a single-word control register that the chip exposes. By reading and writing to these ports, the CPU can request operations to the different console chips and receive information about their state.

When the CPU successfully reads or writes an I/O port, the effects in general may be very different from the expected result of just "reading or writing a value". Ports can have side effects when they are read from or written to. Some write requests may not even write a value at all. To know about the behavior of a specific port, please read the list of control ports in the specification of each chip.

The connection schematic for the control bus is as shown in this diagram. In this case, since all of these chips are always connected, their control ports are always accessible.



The control bus uses addresses of only 14 bits because they are extracted from program instructions (see their format in chapter 7). The process it uses to convert between global and local address spaces is to break the address in fields as shown here:



The 7 connected devices are mapped to the 8 available device IDs as listed in this table:

Device ID	Connected device
0	Timer
1	Random number generator
2	Graphic chip (GPU)
3	Sound chip (SPU)
4	Input controller
5	Cartridge controller
6	Memory card controller
7	*(No device connected)*

6 Chip commands

As stated before, during program execution the CPU will need to make use of the control bus to invoke functions from other chips. The mechanism used by console chips to allow for that is to expose a command port in which the CPU can write certain command codes to trigger specific actions in that chip.

Command parameters

Some chip functions need to be specified certain parameters. For instance, if a program wishes to draw a texture region on the screen it is needed to know what region to draw, and where on screen to place it. A logical structure for that command could be this one:

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

Invoking a chip function by writing in its command does not allow the command to receive any parameters, so this type of structure cannot be used. Instead, each chip will feature a series of internal variables that control the behavior of invoked commands. Those variables are exposed through control ports so that they can be read or modified. That way the process for drawing a region would follow a structure such as this instead:

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

The CPU will first adjust the value of the needed variables, and only then write to the chip's command port. This will ensure the chip functions work in the desired way.

7 Internal data formats

All data processed within the console components is always in the form of 32-bit words. However, the same 32 bits will be interpreted in different ways by different console components depending on the context. In this section we will describe the available bit formats and interpretations within a Vircon32 system:

7.1 Integer

The word is interpreted as a 32-bit signed integer. It uses two's-complement notation. This is equivalent to the C data type "int32_t".

Note that this is the only integer data type available: there are no unsigned integers, and there are no variants for different sizes.

7.2 Boolean

The word is first interpreted as an integer number as seen before. This number is then taken as a boolean true/false value. If the integer value is 0 (i.e. all bits are 0), the word is interpreted as false. Any other value will be interpreted as true. This is equivalent to the C data type “bool”.

When producing a boolean, Vircon32 always uses the values 1 for “true” and 0 for “false”.

7.3 Float

The word is interpreted as a 32-bit floating point number. It follows the usual exponential notation as defined by IEEE 754. This is equivalent to the C data type “float”.

Note that floats can use special values such as NaN (not a number), infinities and indeterminates. Vircon32 systems are not required to implement any kind of support or error detection for these non-numeric values, so using them in a program will result in undetermined behavior.

7.4 Binary

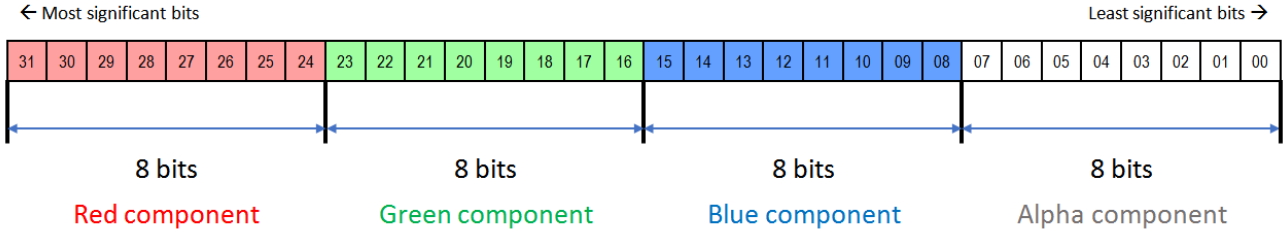
The word is taken to be just a sequence of 32 bits, independent from each other and without any further meaning as a group.

This interpretation is usually only used by binary operations such as AND/OR. This is equivalent to the way C treats integers in such operations.

7.5 GPU color

If the GPU uses a word to produce a screen color, it will interpret that word as a set of 4 8-bit fields representing the color components in RGBA space. This is equivalent to the RGBA8 format used by many graphic cards.

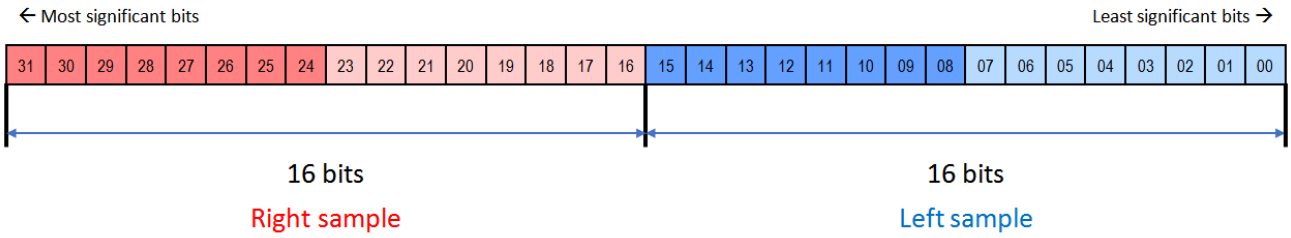
In some situations the alpha component might not be applicable, and in those cases the value for Alpha field will simply be ignored, and treated as “full opacity”.



7.6 SPU sample

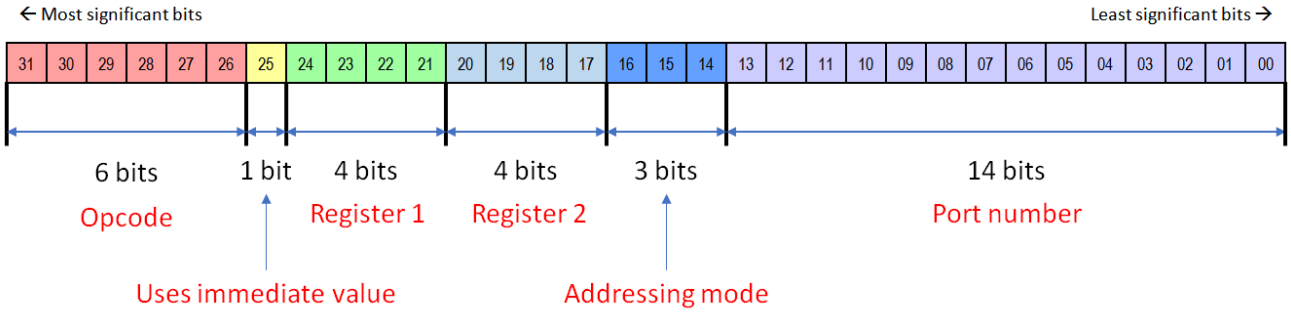
If the SPU uses a word to output a sound sample, it will interpret it as a pair of 16-bit signed integers: one for the left speaker and one for the right speaker. The two of them together form a single 32-bit stereo sample.

In this diagram the darker color bits for each sample represent the most significant byte within that 16-bit integer. This is equivalent to the sample format used by WAV files.



7.7 CPU instruction

If the CPU attempts to execute a word as an instruction, it will interpret it as an instruction opcode with a series of complementary fields. Among those fields is the "Port number" that is used by the control bus.



As for the use and interpretation of other fields, it will be covered in the CPU specification.

8 Endianness

In all of the Vircon32 system the minimum (and only) data size is not the byte, but the 32-bit word. This same principle also applies to storage in memory, so addresses are indexed as word offsets and individual bytes are not accessible.

Since words are not taken as a sequence of bytes, in most situations there is no concept of endianness that needs to apply. However, most computers store and access data in

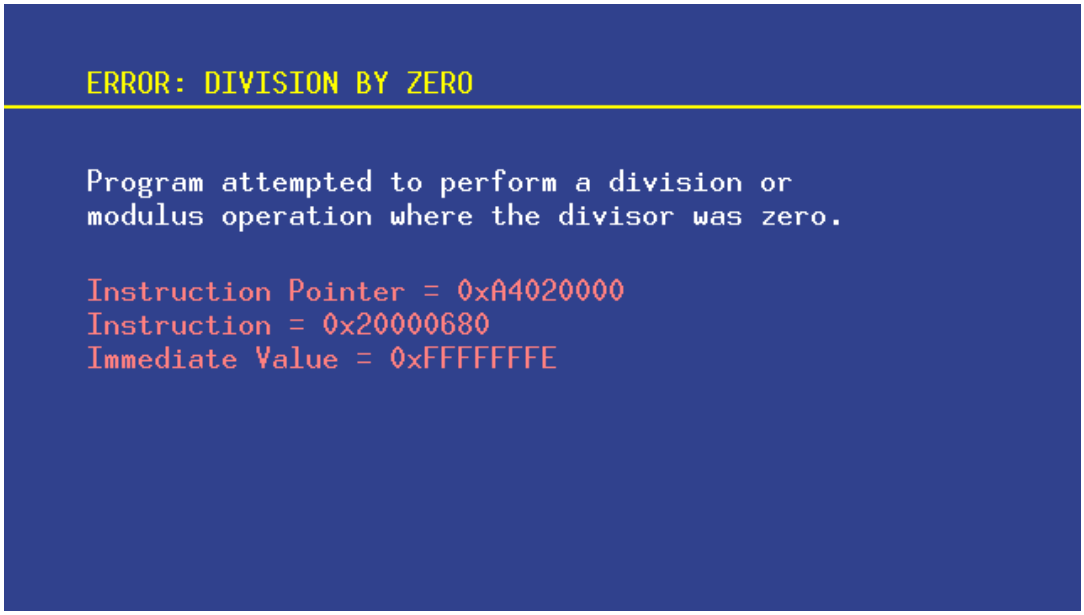
bytes. Because of this some implementation details may need to take endianness into account in order to ensure binary compatibility.

For this reason Vircon32 system is defined to represent words using a **little-endian** system. This is the representation used by most of the current computer systems.

9 Hardware errors

During the execution of a program, some hardware components may be requested to perform operations that they are unable to complete. Common examples are accessing non-existent memory, and mathematical errors such as divisions by zero. When the console detects any of these situations it will trigger a hardware error. When a hardware errors is triggered it will cause the console to stop execution of the program and transfer control to the BIOS error routines to handle the error.

The BIOS error handlers are very simple, since they are only required to display basic information about the error and halt all execution. An example of the error presentation on screen can be seen in this screenshot:



All hardware errors are triggered by processing CPU instructions, so the list of errors and their specific processing will be covered in specification part 3, dedicated to the CPU.

(End of part 2)