Vircon32

32-BIT VIRTUAL CONSOLE

System specification

Part 7: Other console components

Document date 2022.12.27

Written by Carra

What is this?

This document is part number 7 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 order 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 7 of the specification defines the remaining internal components of the console that were not specified so far. For each of them this document will describe its behavior, communications, internal variables and the general process of its operation.

Just like part 6, to group the remaining chips into this single document, each section for a particular chip will instead be done as a subsection in this document.
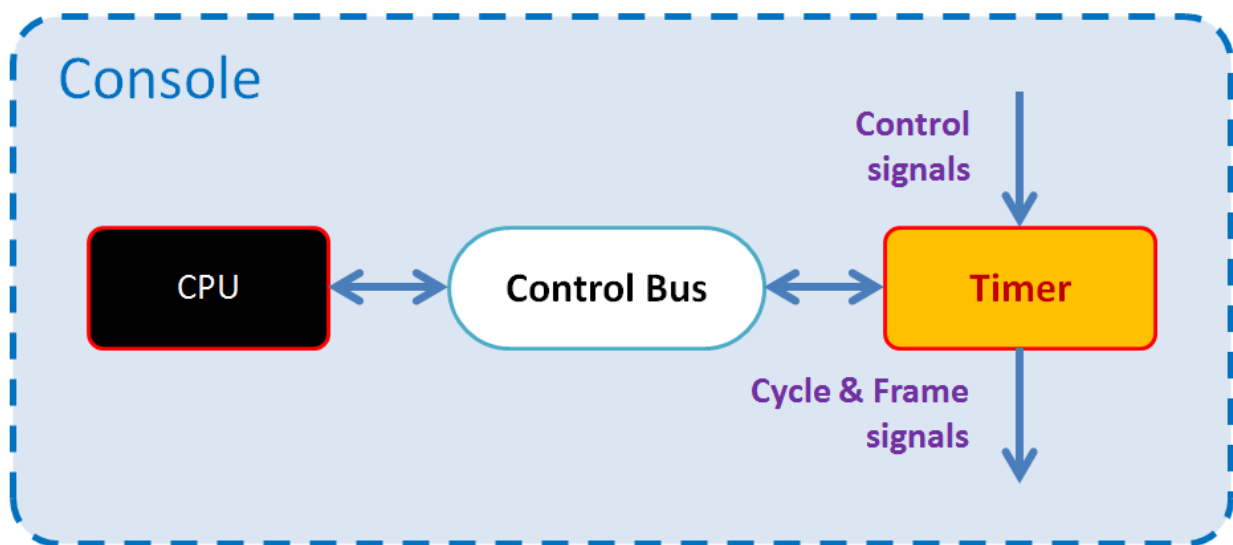
1 Timer

The timer is the chip in charge of controlling the global timing of the console. It does so by producing the new cycle and new frame signals that are then sent to all components in the console. In addition to this, the timer can also provide the CPU with information about the current date and time, via control port requests.

1.1 External connections

Since the timer is just one of the chips forming the console, it cannot operate in isolation. This figure shows all of its communications with other components.

Each of these connections will be explained individually in the sections below.

1.1.1 New cycle and new frame signals
The timer internally generates the new cycle and new frame signals. It will then output these signals so that other console elements can receive them. This is described in section 1.6 of this document.

1.1.2 Control signals
As all console components, the timer receives the signal for reset. The new frame and new cycle signals can also be considered to be "received" here as they would be in any other console component, even if they are being sent by the timer itself. The responses to control signals are detailed in section 1.7 of this document.

1.1.3 Control Bus
The timer is connected as slave device to the Control Bus, with device ID = 0. This allows the bus master (the CPU) to request read or write operations on the control ports exposed

by the timer. The list of its ports as well as their properties will be detailed in later sections.

1.2 Current date and time

The timer has an internal clock that stays updated with the current date and time. It is up to the implementation to define how this managed: hardware implementations could feature a clock with an internal battery that keeps functioning even when the console is off. Instead, a software implementation could just update date and time from external sources every time the console is powered on.

1.2.1 Internal date format

The timer internally represents the current date as a 32-bit word. This internal format groups 2 fields, each encoded as a 16-bit unsigned integer, as follows:
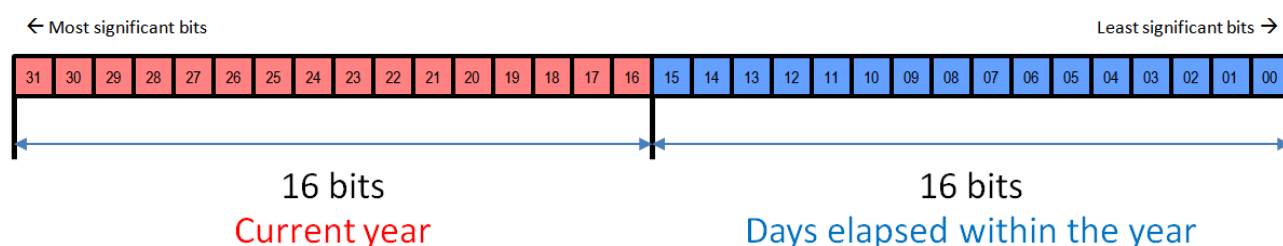
The upper 16 bits are simply the number for the current year (e.g: 2022). Lower 16 bits represent the number of days currently elapsed for that year. So if for instance 3 days are elapsed, it will now be January 4th.

The valid ranges for these 2 integer fields in the date format are as follows:

- **CurrentYear:** From 0 to 65535
- **ElapsedYearDays:** From 0 = January 1st to 364 = December 31st (*)

(*): If the present year is considered a leap year, upper range for ElapsedYearDays extends to 365 to include February 29th as a new date in the year.

1.2.2 Internal time format

Within a given date, the timer also provides the current time within the day. This time is represented internally as a regular 32-bit integer value, that stores the number of elapsed seconds within the current day.

A day has 24 x 60 x 60 = 86400 seconds, so this format to represent elapsed seconds within the day will have a valid range from 0 = 00:00:00 to 86399 = 23:59:59.

1.3 Process for updating time

As time passes, the timer needs to keep its internal current date and time updated. The internal time format has a resolution of 1 second, so updates will only be needed each time a full second has passed.

The closest reference the timer has to determine this is the new frame signal. Console frames happen exactly at 60 Hz, so to update date and time the timer will trigger this process each time it produces a new frame signal. The logic used to add the new frame into the current date and time is to propagate any range overflows in the timer's internal counters, as shown in this algorithm:
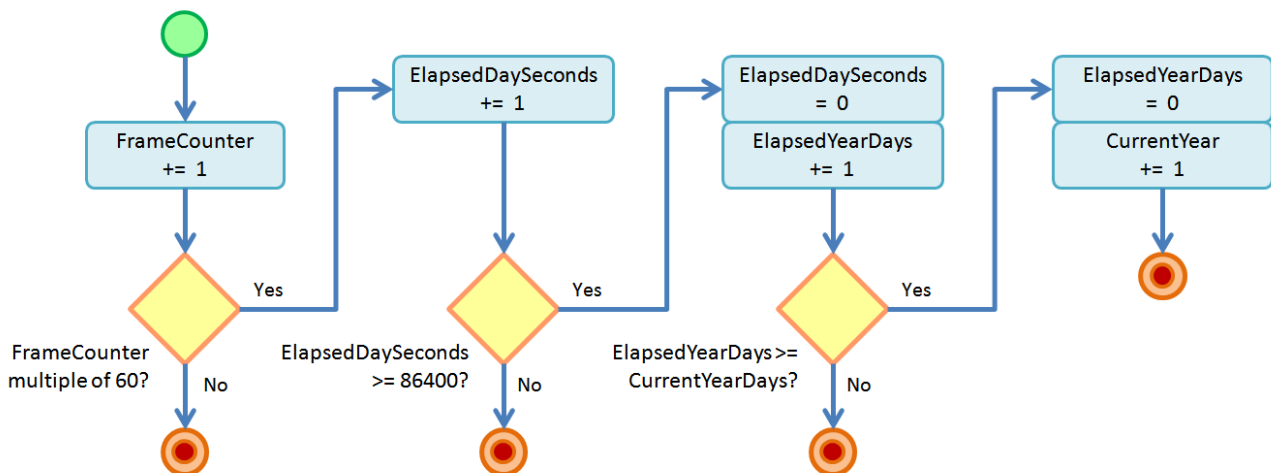
Note that this logic needs to determine whether the current year is a leap year. That will determine if the current year's days need to be 365 or 366.

1.4 Internal variables

The timer features a set of variables that store different aspects of its internal state. These variables are each stored as a 32-bit value, and they are all interpreted using the same data formats (integer, boolean, etc) described in part 2 of the specification, except for the current date format seen before. Here we will list and detail all internal variables.

Current date	**Initial value:** (*)
Format: Date	**Valid range:** January 1st, 0 to December 31st, 65535

This value indicates the current date, in the date format described in section 1.2.1.

(*) Its initial value is determined by the current date at the moment of console startup.

Current time	Initial value: (*)
Format: Integer	Valid range: 0 to 86399

This value indicates the current time, encoded as described in section 1.2.2.

(*) Its initial value is determined by the current time at the moment of console startup.

Frame counter	Initial value: 0
Format: Integer	Valid range: Any non-negative integer value

This value indicates the number of frames elapsed since last console power on or reset.

Cycle counter	Initial value: 0
Format: Integer	Valid range: 0 to 149999

This value indicates the number of CPU cycles elapsed within the current frame.

1.5 Control ports

This section details the set of control ports exposed by the timer via its connection to the CPU control bus as a slave device. All exposed ports, along with their basic properties are listed in the following table:

List of exposed control ports			
External address	Internal address	Port name	R/W access
000h	00h	Current Date	Read Only
001h	01h	Current Time	Read Only
002h	02h	Frame Counter	Read Only
003h	03h	Cycle Counter	Read Only

1.5.1 Behavior on port read/write requests

Unlike other chips, timer ports can actually be modeled as read-only registers. The effect of reading these ports is only obtaining their related value, as detailed in this section.

Note that, in addition to the actions performed, a success/failure response will need to be provided to the request as part of the control bus communication. This response will

always be success on read requests, and failure on write requests. In this last case the timer will perform no further actions and the CPU will trigger a HW error.

Current Date port

On read requests:
The timer will provide the current value of the internal variable "Current date".

Current Time port

On read requests:
The timer will provide the current value of the internal variable "Current time".

Frame Counter port

On read requests:
The timer will provide the current value of the internal variable "Frame counter".

Cycle Counter port

On read requests:
The timer will provide the current value of the internal variable "Cycle counter".

1.6 Producing timing signals

To control the global console timing, the timer needs to be able to accurately measure time. The specific mechanism used for this is to be defined by the implementation: hardware implementations could use an oscillator, while a software version might use system timers with enough resolution.

While the console is on, the timer will produce new frame signals at regular intervals as time passes, at a frequency of 60 Hz. This frequency should be as precise as possible to avoid synchronization problems when the console keeps running for a long time.

Between each 2 consecutive new frame signals, the timer must also produce 150,000 new cycle signals. Note that, unlike new frame signals, the new cycle signals are not required to be produced at regular intervals. They can be produced at any rate within its containing frame, even irregularly. Implementations are free to decide the specific timing for new cycle signals within each given frame.

1.7 Responses to control signals

As with all components in the console, whenever a control signal is triggered the timer will receive it and produce a response to process that event. For each of the control signals, the timer will respond by performing the following actions:

Reset signal:
- Internal variables "Frame counter" and "Cycle counter" are set to their initial values.
- Internal variables "Current date" and "Current time" are set to represent current date and time respectively. If needed, the timer will access the implementation's date and time information and convert it to the internal formats described in section 1.2.
- The timer generates a new frame signal to begin the initial frame.

Frame signal:
- The timer triggers the time update process, as described in section 1.3.
- Internal variable "Cycle counter" is set to 0.
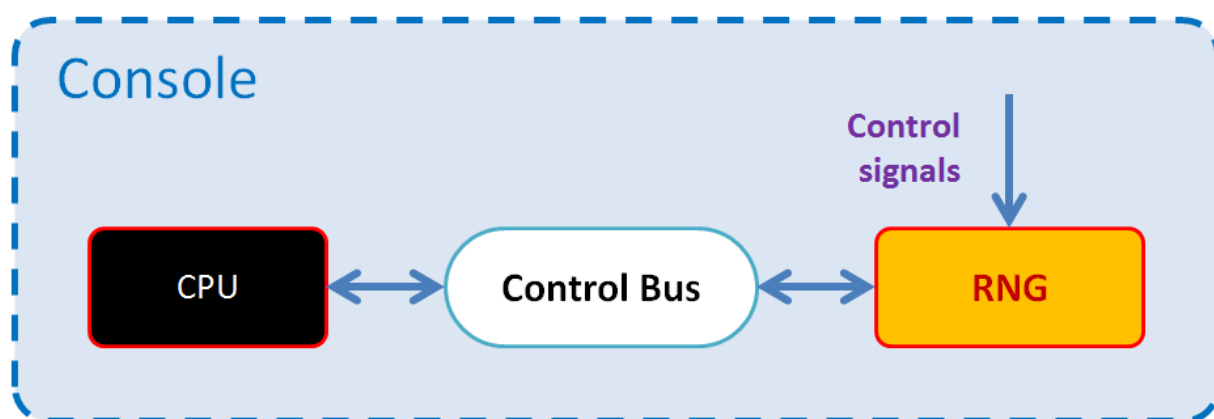
Cycle signal:
- Internal variable "Cycle counter" is incremented by 1.

2 Random Number Generator (RNG)

The RNG chip has only one function: generating sequences of pseudo-random numbers. This is useful in games that need to simulate luck or random behaviors. While this kind of feature is quite easy to implement in software, having a chip dedicated to it is still valuable because it provides all Vircon32 systems with a standard generation algorithm that all implementations are required to agree with.

2.1 External connections

Since the RNG is just one of the chips forming the console, it cannot operate in isolation. This figure shows all of its communications with other components..

Each of these connections will be explained individually in the sections below.

2.1.1 Control signals
As all console components, the RNG receives the signals for reset, new frame and new cycle. The responses to those signals are detailed in section 2.5 of this document.

2.1.2 Control Bus
The RNG is connected as slave device to the Control Bus, with device ID = 1. This allows the bus master (the CPU) to request read or write operations on the control ports exposed by the RNG. The list of its ports as well as their properties will be detailed in later sections.

2.2 Internal variables

The RNG features a single variable that stores its internal state. This variable is stored as a 32-bit value, and it is interpreted using the same data formats (integer, boolean, etc) described in part 2 of the specification. Here we will list and detail this internal variable.

Current value	Initial value: 1
Format: Integer	Valid range: From 1 to 7FFFFFFEh

This value represents the current seed used by the pseudo-random generation algorithm. It is also the next value that will be provided as part of the sequence.

2.3 Control ports

This section details the set of control ports exposed by the RNG via its connection to the CPU control bus as a slave device. The single exposed port, along with its basic properties, is listed in the following table:

List of exposed control ports			
External address	Internal address	Port name	R/W access
100h	00h	Current Value	Read & Write

2.3.1 Behavior on port read/write requests

The RNG's control port is not simply a hardware register. The effects triggered by a read/write request to this port can be different than reading or writing values. This section details those behaviors.

In addition to the actions performed, the response provided to the control bus for any read and write requests to this port will always be success.

Current Value port

On read requests:
The RNG will provide the current value of the internal variable "Current value". After that, the RNG will trigger the sequence generation process decribed in section 2.4 to update said variable with the next value in the sequence.

On write requests:
The RNG will check if the received value is out of range for the internal variable "Current value". In case it is, the request will be ignored. For valid values, the RNG will overwrite the internal variable "Current value" with the received value. This will make the next read request to this port to provide the new written value.

2.4 Sequence generation process

Algorithms to generate pseudo-random numbers are usually based on an initial seed value. The sequence of values obtained is determined completely from that specific initial seed value. The seed used by the RNG is its internal variable "Current value". For each generation step, that variable gets overwritten with the next value in the sequence. Then, that updated value will in turn serve as seed for the next generation step, and so on.

Every Vircon32 implementation is required to use an equivalent generation algorithm. This means every system must generate the same sequence for a same initial seed value. In the case of the RNG, the reference algorithm is the following:

Note that this algorithm makes use of a temporary 64-bit integer variable that we will call Value64. The reason for this is that, if we use only 32-bit values, there may be inconsistencies in the way that different platforms handle the overflow bits produced by a multiplication. By using a 64-bit variable to perform that multiplication we can ensure a consistent result.

The numeric values used for these algorithm are chosen to be the same as C++11's minstd_rand implementation.

2.5 Responses to control signals

As with all components in the console, whenever a control signal is triggered the RNG will receive it and produce a response to process that event. For each of the control signals, the RNG will respond by performing the following actions:

Reset signal:
- Internal variable "Current value" is set to its initial value.
- Any additional effects associated to that change in the internal variable are immediately applied, as described in the control port write behaviors.

Frame and Cycle signals:
- The RNG does not need to react to these signals, unless specific implementation details require it.
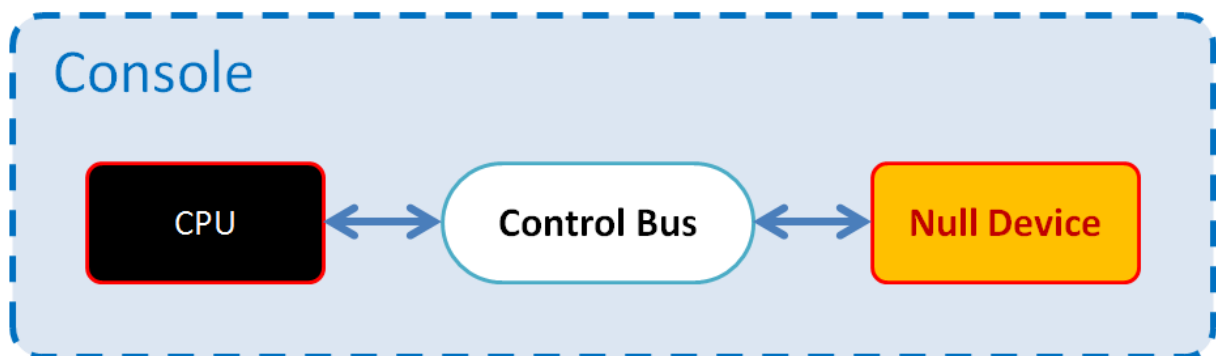
3 Null device

This device was not mentioned in any document so far, because it is not a real console chip. Instead, it is an optional dummy chip that may be useful in some implementations to help ensure that any requests to access non-existent control ports are always responded with failure.

As discussed in part 2 of the specification, control bus global addresses are composed by 3 bits (device ID) + 8 bits (port's local address in that device). This means that the control bus address space is effectively split into 8 separate device IDs. However, since the control bus only has 7 devices connected to it (IDs 0 to 6), device ID = 7 is left unused and does not correspond to any slave device.

By connecting the null device into the last existing device ID, it becomes possible for implementations to treat all port read/write requests the same way, and just send every request to the corresponding device for it to be processed.

3.1 External connections

The null device's only function is to be connected to the control bus as slave device, with ID = 7, and respond to any received requests with failure. As a dummy device it does not communicate with any of the chips that perform actual console functions. It does not need to receive any control signals either.


3.2 Control ports

The null device does not actually expose any ports. Instead, it automatically rejects any request within its full port range (external addresses from 700h to 7FFh) by providing a failure response to the control bus.

4 RAM

The RAM chip memory is mainly a passive component: it does not perform any functions other than managing the memory it contains and processing read/write requests for it. As such, the RAM chip does not feature any internal variables or control ports.

4.1 External connections

Since the RAM is just one of the chips forming the console, it cannot operate in isolation. This figure shows all of its communications with other components.
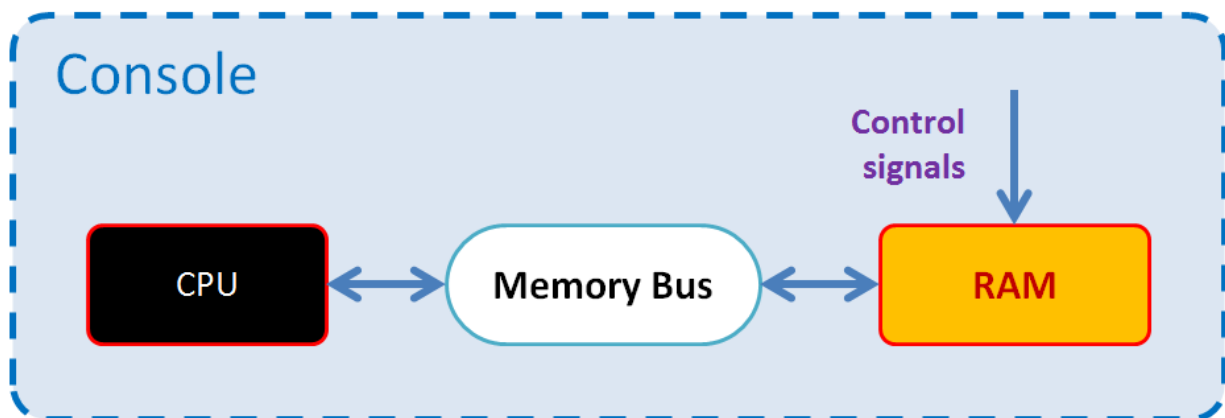
Each of these connections will be explained individually in the sections below.

4.1.1 Control signals

As all console components, the RAM chip receives the signals for reset, new frame and new cycle. The responses to those signals are detailed in section 4.3 of this document.

4.1.2 Memory Bus

The RAM chip is connected as slave device to the Memory Bus, with device ID = 0. This allows the bus master (the CPU) to request read or write operations on the memory addresses exposed by the RAM chip. The range and properties of RAM memory addresses will be discussed in later sections.

4.2 RAM Memory

The RAM memory is a numbered sequence of 32-bit words. It has a size of 4 MWords = 4 x 1024 x 1024 words. RAM stands for Random Access Memory: each of these individual words can be independently read and modified.

RAM memory is not persistent: it can only store its contents while the console is powered on. Its contained data will get deleted at console power off.

4.2.1 Memory connection

The RAM chip is connected to the memory bus to expose the contents of its memory to the CPU. RAM uses device ID = 0 within the memory bus, so the range of addresses for RAM memory will be:

> Internal addresses → From 0 to 3FFFFFh (= 4 x 1024 x 1024 − 1)
> External addresses → From 00000000h to 003FFFFFh.

4.2.2 Memory performance

While no specific performance levels will be stated, RAM memory is assumed to be fast enough so that every cycle, it can respond to all of the CPU's requests in time for it to finish the current instruction within that same cycle.

4.3 Responses to control signals

As with all components in the console, whenever a control signal is triggered the RAM chip will receive it and produce a response to process that event. For each of the control signals, the RAM chip will respond by performing the following actions:

Reset signal:

- All RAM memory addresses will be set to value 0.

Frame and Cycle signals:

- The RAM chip does not need to react to these signals, unless specific implementation details require it.

5 BIOS

The BIOS chip, just like RAM, is mainly a passive component that manages access to the BIOS ROM. As such, the BIOS chip does not feature any internal variables or control ports. It has no need to receive control signals either, unless specific implementation details require it.

BIOS stands for Basic Input/Output System, and is a common way of calling, in many electronic devices, a set of installed routines in ROM that take charge of managing low level operations such as startup or error handling. In the case of Vircon32 a BIOS ROM is similar in structure to a cartridge ROM, containing the same 3 parts (program ROM, video ROM and audio ROM), but subject to some additional restrictions.

5.1 External connections

Since the BIOS is just one of the chips forming the console, it cannot operate in isolation. This figure shows all of its communications with other components.
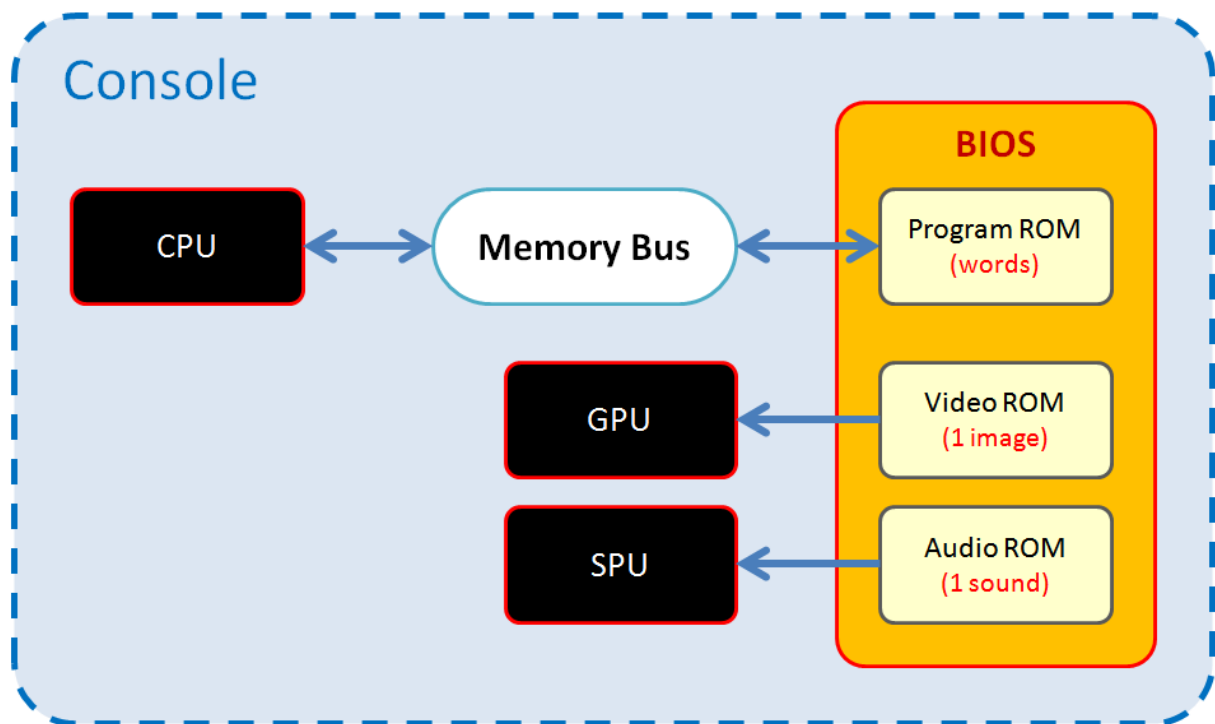
Each of these connections will be explained individually in the sections below.

5.1.1 Memory Bus
The BIOS chip is connected as slave device to the Memory Bus, with device ID = 1. This allows the bus master (the CPU) to request read or write operations on the memory

addresses exposed by the BIOS chip. The range and properties of BIOS memory addresses will be discussed in later sections.

5.1.2 GPU and SPU

The GPU and SPU need to access the BIOS's video and audio ROM respectively. Connections from GPU and SPU are conceived as a direct access: each chip is able to read those contents freely. Still, the actual mechanisms to set up and manage this access are to be defined by the implementation.

5.2 BIOS memory

To run the BIOS routines, the CPU needs to read the memory words stored in the BIOS by connecting to its program ROM. A program ROM is a read-only memory region that contains a sequence of 32-bit words.

The BIOS chip is connected to the memory bus to allow it to expose the contents of its program ROM to the CPU. The BIOS chip uses device ID = 1 within the memory bus so, for a BIOS program ROM containing N words, the range of addresses for BIOS memory will be:

> Internal addresses → From 0 to N − 1
> External addresses → From 10000000h to (10000000h + N − 1).

The size of BIOS program ROM will vary for each different BIOS. It can contain any number of words from 1 up to the BIOS program ROM size limit of 1024 x 1024.

5.2.1 Memory performance

While no specific performance levels will be stated, the 3 components of the BIOS ROM are assumed to be fast enough to be capable of the following:

- Every cycle, program ROM memory can provide the CPU with all words it requests in time for it to finish the current instruction within that same cycle.
- Every frame, the video ROM memory can provide the GPU with all pixels it needs in time for it to finish all drawing operations within that same frame.
- Every frame, the audio ROM memory can provide the SPU with all samples it needs in time for it to finish all sound generation within that same frame.

5.3 Requirements of a BIOS

In addition to the general requirements applicable for all program, video and audio ROMs, the contents of a Vircon32 BIOS are subjected to several, more restrictive requirements. They are listed below, grouped into a subsection for each type:

5.3.1 Requirements for BIOS contents

- Its video ROM must exist and contain exactly 1 texture.
- Its audio ROM must exist and contain exactly 1 sound.
- Its audio ROM has a size limit of 1024 x 1024 samples.
- Its program ROM has a size limit of 1024 x 1024 words.
- Its program ROM must contain a startup routine to be called immediately after each console power on or reset. The entry point for this routine must be located at address 4 in the program ROM.
- Its program ROM must contain an error handler routine to be called immediately after the CPU triggers any hardware error. The entry point for this routine must be located at address 0 in the program ROM.

5.3.2 Requirements for BIOS startup routine

- It must define a text font in the BIOS texture's regions 0 to 255. See section 5.4 for more detail about this font.
- It must define BIOS texture's region 256 to be a single pixel of full-opacity white, this is: its 4 RGBA components will be 255.
- Displaying some kind of intro screen with image and sound is optional, but recommended. It can help discard malfunctions in specific implementations without needing to use a cartridge.
- As its last step it must check if a cartridge is present and, if it is, it will transfer execution to the beginning of its program ROM.
- If the cartridge is instead found to be absent, it will report it on screen.

5.3.3 Requirements for BIOS error handler

- In must report on screen, at least, the type of hardware error that happened.
- Displaying extended information about the error (for instance, the value of the Instruction Pointer register) is optional, but recommended.
- After all its processing is done, it must stop execution by halting the CPU.

5.4 The BIOS text font

The BIOS text font consists of 256 characters, defined in BIOS texture's regions 0 to 255. Each of these regions must be 10 pixels wide x 20 pixels high, and have its hotspot located at its top-left pixel. Since a BIOS is always installed, all programs can safely assume that the BIOS text font is available for them to write text.

Character mapping for the BIOS font is mainly based on Windows code page 1252 (also known as Latin-1), as well as its subset ISO 8859-1. The following table indicates which character must be defined at each region, from 00h to FFh. The 4 digits in blue at each cell indicate the Unicode code point of the character at that position.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	not used	not used	not used	not used	not used	not used	not used	not used	not used	(HT) 0009	(LF) 000A	not used	not used	(CR) 000D	⬚ 2B1A	▯ 25AF
10	(empty)	░ 2591	▒ 2592	▓ 2593	█ 2588	─ 2500	│ 2502	┘ 2518	┐ 2510	┌ 250C	└ 2514	├ 251C	┤ 2524	┴ 2534	┬ 252C	┼ 253C
20	(SP) 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
30	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
40	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
50	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
60	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
70	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	not used
80	€ 20AC	not used	‚ 201A	ƒ 0192	„ 201E	… 2026	† 2020	‡ 2021	ˆ 02C6	‰ 2030	Š 0160	‹ 2039	Œ 0152	not used	Ž 017D	not used
90	not used	` 2018	´ 2019	" 201C	" 201D	• 2022	– 2013	— 2014	˜ 02DC	TM 2122	š 0161	› 203A	œ 0153	not used	ž 017E	Ÿ 0178
A0	not used	¡ 00A1	¢ 00A2	£ 00A3	¤ 00A4	¥ 00A5	¦ 00A6	§ 00A7	¨ 00A8	© 00A9	ª 00AA	« 00AB	¬ 00AC	- 00AD	® 00AE	¯ 00AF
B0	° 00B0	± 00B1	² 00B2	³ 00B3	´ 00B4	µ 00B5	¶ 00B6	· 00B7	¸ 00B8	¹ 00B9	º 00BA	» 00BB	¼ 00BC	½ 00BD	¾ 00BE	¿ 00BF
C0	À 00C0	Á 00C1	Â 00C2	Ã 00C3	Ä 00C4	Å 00C5	Æ 00C6	Ç 00C7	È 00C8	É 00C9	Ê 00CA	Ë 00CB	Ì 00CC	Í 00CD	Î 00CE	Ï 00CF
D0	Ð 00D0	Ñ 00D1	Ò 00D2	Ó 00D3	Ô 00D4	Õ 00D5	Ö 00D6	× 00D7	Ø 00D8	Ù 00D9	Ú 00DA	Û 00DB	Ü 00DC	Ý 00DD	Þ 00DE	ß 00DF
E0	à 00E0	á 00E1	â 00E2	ã 00E3	ä 00E4	å 00E5	æ 00E6	ç 00E7	è 00E8	é 00E9	ê 00EA	ë 00EB	ì 00EC	í 00ED	î 00EE	ï 00EF
F0	ð 00F0	ñ 00F1	ò 00F2	ó 00F3	ô 00F4	õ 00F5	ö 00F6	÷ 00F7	ø 00F8	ù 00F9	ú 00FA	û 00FB	ü 00FC	ý 00FD	þ 00FE	ÿ 00FF

The colors used across this table are to be interpreted as follows:

	White: Regular printable characters. Depicted as their respective code points.
	Red: Unused characters. They should all be empty.
	Yellow: Whitespace characters. Space should be fully empty; the rest can also be empty, or depicted as chosen by the implementation.
	Green: These positions correspond to non-printable control codes in code page 1252, but the BIOS font replaces them with a set of characters designed for drawing rectangular frames and grids using only text.

5.5 The standard BIOS

There can be any number of BIOS implementations, created by different authors. Each of them can be considered as a valid BIOS as long as it is compliant with the requirements stated in this document. However there is only one standard BIOS, which is named as "Vircon32 standard BIOS" in the ROM file. Other BIOS implementations are required to use different names so that implementations and users can distinguish them from the standard BIOS.

It is recommended that all Vircon32 implementations, even if they use a custom BIOS by default, include the standard BIOS as an option as well. That way all implementations will have the means to agree in the way hardware errors are handled, in case developers need to rely on this for testing or debugging.

(End of part 7)