VBXE FX and GTIA cores

v1.27 Programmers Manual

Manual revision 10 · 31 October 2026

Table of contents

1. Introduction **2**
Document conventions 2

2. Compatibility and hardware notes **4**
EXTSEL pull-up 4
Cartridges and cartridge port devices 4
Turbo systems (6502 replacements) 4

3. Detection of VBXE FX/GTIA core I/O page, core type and core version **5**
Detection procedure and code example 5

4. FX core **8**
FX core I/O REGISTERS 9

5. GTIA emulation core **10**
GTIA core I/O REGISTERS 11

6. Common FX/GTIA core features **12**
Soft and hard RESET 12
Common I/O registers 12
Modification of RGB palettes 15

7. MEMAC, VBXE VRAM bridge **17**
RAMBO compatible shared extended memory 18
MEMAC-A 18
MEMAC-B 20

8. XDL **21**
XDLC control word 22
Associated data 27

9. Overlay modes **28**
Pixel modes 28
Text mode 29

10. Overlay-PF/PMG priority **31**
P0, P1, P2, P3 32

11. Overlay-PF/PMG collision detection **33**
COLMASK, COLDETECT and COLCLR registers 33

12. Attribute Map **35**
Characteristics 35
Attribute Map cell data 35
Resolution swap (RES bit) 38

13. Blitter **39**
BCB structure 40
Extended BCB containers 43
Blitter modes of operation 47
Blitter and constant source data 51
Blitter collision detection 51
Running and controlling the Blitter 53
Blitter cycle usage 55

HISTORY **57**

1. Introduction

This document is a programmer's guide for FPGA cores for VBXE 1.x/2.x extension boards.

> **NOTE**
> This document does not describe the VBXE hardware itself, the detection of the VBXE hardware, or the communication with the VBXE core controller. It describes only the FX and GTIA cores. Therefore, except where it explicitly refers to the hardware (such as the VBXE 1.x/2.x boards), "VBXE" in this document means the functionality of these cores.

Available cores:

Core	VBXE 1.x	VBXE 2.x
FX v1.27.2 PAL	v1f127a1.xbf	v2f127a1.xbf
GTIA v1.27.2 PAL	v1g127a1.xbf	v2g127a1.xbf
FX v1.27.2 NTSC	v1f127a2.xbf	v2f127a2.xbf
GTIA v1.27.2 NTSC	v1g127a2.xbf	v2g127a2.xbf

Cores with a name ending in "r" contain emulation for RAMBO 256 KB extended memory - see RAMBO.

> **NOTE**
> Currently v1.27 cores do not have this function.

Cores named "FX" - see FX core. Cores named "GTIA" - see GTIA emulation core.

> **NOTE**
> Neither VBXE nor its cores detect the TV system the computer is running on by themselves; the correct core (PAL or NTSC) has to be selected and loaded explicitly. Both cores will, however, operate independently of the computer's own TV system - the only differences between them being the default palette 0 and, in GTIA cores, the default state of the BLEND bit (see FEATURE_CONTROL).

To recognize the type of core that is actually running on the FPGA please refer to the section Detection of VBXE FX/GTIA core I/O page, core type and core version.

Document conventions

I/O registers access and RESET state flags

"W" -- write-only
"R" -- read-only
"RW" -- read/write
"-0" -- "0" after RESET / core reboot
"-1" -- "1" after RESET / core reboot
"-x" -- undefined after RESET / core reboot

In these flags, "RESET" means a soft or a hard reset (see Soft and hard RESET), and "core reboot" means loading the core into the FPGA, which is the same as power up and is always followed by a hard reset.

Unused I/O registers are marked "-". Writing to them is forbidden, their read state is UNDEFINED (usually 255).

Unused bitfields in I/O registers are marked "-". Zeros should be always written there. Their read state is UNDEFINED, except if explicitly stated.

Reading write-only registers and bitfields results in UNDEFINED value except if explicitly defined.

The same convention is used for VRAM data and structures (XDL, BCB list, Attribute Map).

Other conventions

Unless stated otherwise, a field that does not occupy whole bytes is right-aligned within the byte(s) carrying it.

Throughout this document, "hires" refers to the ANTIC/GTIA high resolution mode (GR.8), while "HR" refers to the Overlay High Resolution pixel mode.

2. Compatibility and hardware notes

Compatibility has to be provided by all the devices connected to the computer at the same time. There are situations in which VBXE works correctly on its own, but does not in combination with another device, either internal or connected externally to the bus of the computer.

This chapter collects notes on how the cores work together with the computer and with other hardware in it.

EXTSEL pull-up

VBXE pulls EXTSEL low during a MEMAC access and then releases it; the line goes high again through the pull-up resistor in the computer (usually 3 kΩ). With the capacitance of the bus this rise is slow and can take more than 100 ns. If VBXE does not cooperate well with devices on the parallel bus (PBI/ECI), replacing this resistor with 1.5 kΩ can help: EXTSEL then rises about twice as fast. Every device that pulls EXTSEL low then has to sink about 3 mA. Use 1 kΩ (about 5 mA) only if all of them can sink that current. The change makes sense only when such problems occur.

Cartridges and cartridge port devices

> CAUTION
>
> **SUBcart with its 1 MB PORTB memory extension enabled is not compatible with VBXE** (checked on SUBcart firmware v012 and v013 beta 1).

In this mode SUBcart puts its own memory in place of the computer's RAM at $A000-$BFFF in every CPU access (it pulls the REF line low, so the computer disables its memory there). MEMAC windows placed over $A000-$BFFF then do not work: reads return SUBcart's memory and writes go there, not to VRAM. This is a limitation of SUBcart, not a VBXE malfunction: VBXE gives the bus to the device that took it, exactly as it does for ROM and cartridges. The smaller PORTB modes of SUBcart (128, 192, 320 and 576 KB), and SUBcart with the extension off, work with VBXE.

Limited compatibility with VBXE is also suspected when some of the cartridge emulation modes of SUBcart are used (running programs from .car files).

Turbo systems (6502 replacements)

VBXE, a device designed about 20 years ago, is made for the architecture of the Atari 800XL/XE, and its software and cores assume, as they cannot do otherwise, that all the signals of the computer that they use work according to their original specifications.

The 6502 replacements available today (turbo systems) change the architecture of the computer in various, often quite radical ways, which may have different consequences for mutual compatibility, hard to predict and different for each device. For this reason VBXE works in such an environment on a best-effort basis, and the compatibility has to be accepted as it is.

3. Detection of VBXE FX/GTIA core I/O page, core type and core version

The type of the core loaded can be determined by reading:

CORE_TYPE

bits [7:0]
TYPE
R

CORE_TYPE for FX core has value of $10.
CORE_TYPE for GTIA core has value of $11.

Other values are RESERVED for further core types. The value, if VBXE is not present, is undefined.

The exact version of the core loaded can be read from:

CORE_REVISION

bit 7	bits [6:0]
RAMBO	VERSION
R	R

RAMBO field is set only for cores featuring 256 KB RAMBO extended memory (shared with VBXE VRAM). These cores can be also distinguished by "r" letter in the core name. In most cases this information can be discarded, thus the $7F AND mask should be applied to extract core version.

It is assumed, that compatible cores have bits [6:4] of CORE_REVISION (three most significant bits of VERSION field) equal, so software for core v1.20 should run with core v1.27. Cores aren't downward compatible, however: an application that uses features specific to core v1.27 will not run on a v1.26 or older core, so such an application should check the core revision strictly. As for now, however, no older cores than v1.26 should be used at all.

VERSION for FX core v1.27 has value $27.
VERSION for GTIA core v1.27 has value $27.

> **WARNING**
> With core not loaded these registers do not exist despite VBXE hardware being still present and detectable for example by FC.XEX utility.

Detection procedure and code example

The VBXE registers, listed throughout this document with the placeholder address "Dx", are mapped either to the $D600-$D6FF page (base address $D600) or to the $D700-$D7FF page (base address $D700), depending on the board's configuration. Since there is no separate way to ask the VBXE which page it is decoded on, the programmer has to probe each candidate page in turn

and treat the page that reports plausible values as the correct one. While probing, only reads should be performed - no register should be written to until the correct page (and thus the presence of the VBXE) has been confirmed, since the probed addresses may belong to some other, unknown hardware, and writing to them could have unintended side effects. Basing this check on CORE_TYPE alone is possible, but checking both CORE_TYPE and CORE_REVISION together is more reliable, as it is far less likely for both offsets to accidentally read back plausible values from an unrelated device or open bus.

Within either page, CORE_TYPE sits at offset $40 and CORE_REVISION at offset $41. The recommended approach is to prepare a zero page vector holding the VBXE base address (with the low byte always 0), set its high byte to $D6, then $D7 in turn, and on each attempt read both registers through it. The page is accepted once CORE_TYPE is a recognized value ($10 or $11) and CORE_REVISION, masked with $7F to filter out bit 7 (RAMBO, distinguishing an "a" core from an "r" core), falls in a plausible range; the example below looks for a core v1.26 or newer v1.2x. The very same zero page vector can then be reused for 6502 indirect indexed (with Y) addressing of any VBXE register for the rest of the program, unless the programmer decides to use some other addressing method instead.

code example:

```
vbxe_io_ptr = $80            ; zero page vector to the VBXE register page (2 bytes: vbxe_io_ptr,
vbxe_io_ptr+1)

CORE_TYPE_OFS      = $40     ; offset of CORE_TYPE (read) within the register page
VIDEO_CONTROL_OFS  = $40     ; offset of VIDEO_CONTROL (write) within the register page
CORE_REVISION_OFS  = $41     ; offset of CORE_REVISION / XDL_ADR0 within the register page

detect_vbxe:

  lda   #0
  sta   vbxe_io_ptr

  lda   #$D6
  sta   vbxe_io_ptr+1
  jsr   probe_vbxe
  bcs   page_found

  lda   #$D7
  sta   vbxe_io_ptr+1
  jsr   probe_vbxe
  bcs   page_found

  jmp   error                  ; no VBXE found on either page

probe_vbxe:

  ldy   #CORE_TYPE_OFS
  lda   (vbxe_io_ptr),y
  cmp   #$10
  beq   probe_check_revision   ; $10 = FX core, tentatively
  cmp   #$11
  beq   probe_check_revision   ; $11 = GTIA core, tentatively
  clc
  rts

probe_check_revision:
```

```
  pha                         ; keep CORE_TYPE for later
  ldy   #CORE_REVISION_OFS
  lda   (vbxe_io_ptr),y
  and   #$7F                  ; to filter out "r" core flag
  cmp   #$26
  bcc   probe_fail            ; revision too low to be a real / acceptable core
  cmp   #$30
  bcs   probe_fail            ; revision too high to be a real / acceptable core
  sta   exact_core_revision   ; #$27 = v1.27 a/r - determined once, for either core type
  pla                         ; restore CORE_TYPE into A
  sec
  rts

probe_fail:
  pla
  clc
  rts

page_found:

  ; vbxe_io_ptr+1 now holds $D6 or $D7 - the confirmed VBXE register page.
  ; exact_core_revision now holds CORE_REVISION (masked with $7F).
  ; A (from the successful probe) still holds CORE_TYPE ($10 or $11).

  cmp   #$10
  beq   fx_core
  jmp   gtia_core

fx_core:

  ; FX 1.2x core application here

  rts

gtia_core:

  ; GTIA 1.2x core application here

  rts
```

From this point on, any VBXE register at offset Y within the confirmed page can be accessed with indirect indexed addressing through the vbxe_io_ptr zero page vector, for example:

```
  ldy   #CORE_TYPE_OFS
  lda   (vbxe_io_ptr),y      ; read CORE_TYPE

  ldy   #VIDEO_CONTROL_OFS
  lda   #...
  sta   (vbxe_io_ptr),y      ; write VIDEO_CONTROL
```

4. FX core

FX core offers all VBXE functionality and clear RGB output.

Available features:

- GTIA emulation with 21-bit RGB output,
- MEMAC, 512 KB VRAM bridge system,
- XDL (eXtended Display List),
- Text and graphic Overlay,
- Attribute Map,
- Four modifiable 256-color RGB Palettes,
- Blitter.

FX core I/O REGISTERS

Address	Write	Read
$Dx40	VIDEO_CONTROL	CORE_TYPE
$Dx41	XDL_ADR0 (XDL_ADR[7:0])	CORE_REVISION
$Dx42	XDL_ADR1 (XDL_ADR[15:8])	-
$Dx43	XDL_ADR2 (XDL_ADR[18:16])	-
$Dx44	CSEL	-
$Dx45	PSEL	-
$Dx46	CR	-
$Dx47	CG	-
$Dx48	CB	-
$Dx49	COLMASK	-
$Dx4A	COLCLR	COLDETECT
$Dx4B	FEATURE_CONTROL	FEATURE_CONTROL
$Dx4C	-	-
$Dx4D	-	-
$Dx4E	-	-
$Dx4F	-	-
$Dx50	BL_ADR0 (BL_ADR[7:0])	BLT_COLLISION_CODE
$Dx51	BL_ADR1 (BL_ADR[15:8])	-
$Dx52	BL_ADR2 (BL_ADR[18:16])	-
$Dx53	BLITTER_START	BLITTER_BUSY
$Dx54	IRQ_CONTROL	IRQ_STATUS
$Dx55	P0	-
$Dx56	P1	-
$Dx57	P2	-
$Dx58	P3	-
$Dx59	-	-
$Dx5A	-	-
$Dx5B	-	-
$Dx5C	-	-
$Dx5D	MEMAC_B_CONTROL	-
$Dx5E	MEMAC_CONTROL	MEMAC_CONTROL
$Dx5F	MEMAC_BANK_SEL	MEMAC_BANK_SEL

x = 6 or 7, depending on where the VBXE is decoded in the Atari memory.

> **NOTE**
> The registers are decoded without regard to address bit 5, so the whole block is mirrored in the $Dx60-$Dx7F range (for example $Dx60 = $Dx40, $Dx6B = $Dx4B).

5. GTIA emulation core

The GTIA core just emulates GTIA chip with 21-bit RGB output.

There are a few enhancements available:

- PAL blending - averages colors vertically, emulating PAL system color encoding behavior: each displayed color is averaged with the color made of the chroma (hue) part of the pixel at the same position in the previous scanline and the luma of the current pixel. Required mainly by some PAL demos to enhance the number of colors visible on screen. FX core has no such function, being focused on producing as clear and sharp an RGB picture as possible.
- Four palettes as in the case of FX core. Palette 0 isn't modifiable, though.
- FEATURE_CONTROL register.
- VIDEO_CONTROL register (limited to XCOLOR).

GTIA core I/O REGISTERS

Address	Write	Read
$Dx40	VIDEO_CONTROL	CORE_TYPE
$Dx41	-	CORE_REVISION
$Dx42	-	-
$Dx43	-	-
$Dx44	CSEL	-
$Dx45	PSEL	-
$Dx46	CR	-
$Dx47	CG	-
$Dx48	CB	-
$Dx49	-	-
$Dx4A	-	-
$Dx4B	FEATURE_CONTROL	FEATURE_CONTROL
$Dx4C	-	-
$Dx4D	-	-
$Dx4E	-	-
$Dx4F	-	-
$Dx50	-	-
$Dx51	-	-
$Dx52	-	-
$Dx53	-	-
$Dx54	-	-
$Dx55	-	-
$Dx56	-	-
$Dx57	-	-
$Dx58	-	-
$Dx59	-	-
$Dx5A	-	-
$Dx5B	-	-
$Dx5C	-	-
$Dx5D	-	-
$Dx5E	-	-
$Dx5F	-	-

x = 6 or 7, depending on where the VBXE is decoded in the Atari memory.

> **NOTE**
> The registers are decoded without regard to address bit 5, so the whole block is mirrored in the $Dx60-$Dx7F range (for example $Dx60 = $Dx40, $Dx6B = $Dx4B).

6. Common FX/GTIA core features

Soft and hard RESET

Soft reset occurs after any write to $D080..$D0FF.

- reinitializes all (see note) registers to their default state,
- immediately switches off XDL processing, Overlay and Attribute Map,
- stops the blitter.

It does not blank out the screen leaving pure ANTIC/GTIA layer visible.

Hard reset means the RST line (RESET key) reset, or the reset that always follows power up / core reboot (loading the core into the FPGA).

It involves all soft reset actions plus:

- resynchronizes the core to the computer bus,
- blanks out the screen until next VBLANK.

> NOTE
> "all registers" mean all registers and fields with a DEFINED reset state.

Common I/O registers

VIDEO_CONTROL

bits [7:4]	bit 3	bit 2	bit 1	bit 0
-	TRANS15	NO_TRANS	XCOLOR	XDL_ENABLE
-	W-0	W-0	W-NOTE	W-0

Fields available in both FX and GTIA cores:

XCOLOR: Legacy field. It directly controls XCOLOR[1], and, when written, always clears XCOLOR[0] bit - see FEATURE_CONTROL.

> CAUTION
> Because writing VIDEO_CONTROL always clears XCOLOR[0], it may be necessary to write to FEATURE_CONTROL again after every write to VIDEO_CONTROL, in order to precisely set or preserve the desired XCOLOR value - keep this in mind.

The reset state of the XCOLOR field is the same as that of the XCOLOR field of FEATURE_CONTROL: it is cleared to 0 by a reset, unless PXGF is set - then the value is preserved.

Fields available only in FX core:

XDL_ENABLE: 1 = enable the XDL processing after the nearest VSYNC (the end of the vertical synchronization).
0 = disable the XDL processing after the nearest VSYNC (the end of the vertical synchronization).

NO_TRANS: 0 = in the Overlay modes the color index 0 will be treated as transparent and the ANTIC/GTIA display will be visible in its place.
1 = the Overlay has no transparent colors. This bit allows to use all 256 palette indices as solid, opaque colors (but still, Overlay-PF/PMG priority apply).

> **NOTE**
> The NO_TRANS bit has no influence on the Blitter, which in most of its modes will consider color index 0 as transparent.

TRANS15: This bit is only taken into account, when NO_TRANS = 0. This allows to define additional transparent colors: see Overlay modes for details.

FEATURE_CONTROL

bit 7	bit 6	bit 5	bits [4:3]	bit 2	bits [1:0]
PXGF	BLEND	CLIP	XCOLOR	-	GTIA_PALETTE
RW-0	GTIA cores: RW-PAL:1/NTSC:0 FX cores: R-0	RW-0	RW-0	R-0	RW-0

PXGF: Persistent GTIA Features - if set all other fields in this register do not change their values during soft and hard RESET (except power up / core reboot, as VBXE does not have nonvolatile configuration memory). PXGF itself is initialized only after power up / core reboot.

BLEND: GTIA cores: controls PAL blending feature enable.

CLIP: Hard clipping enable. Limits width of visible screen to 336 ANTIC/GTIA hires pixels (standard 320 plus 8 pixels from each side) thus cutting off most if not all unwanted overscan garbage. Must be noted, however, that there is always an enabled 360-pixel clip, leaving visible only 20 pixels from each side of standard width PF. The purpose is to symmetrically cut off ANTIC DMA garbage in hscrolled lines.

XCOLOR: see XCOLOR below.

GTIA_PALETTE: Select palette used by GTIA core and palette used by FX core for the GTIA layer (PF palette) from the top of the screen until changed by the XDL.

XCOLOR

XCOLOR controls the processing of the video data in the ANTIC/GTIA hires modes and the width of the GTIA color registers.

In the description below, a "set" pixel (foreground, lit) is a pixel whose bit is 1, and a "clear" pixel (background, unlit) is a pixel whose bit is 0.

XCOLOR	meaning
00	XCOLOR off, standard ANTIC/GTIA hires luma (COLPF1[3:1]) bypass: bPF1 (see below), 7-bit color regs (bit 0 of the color register luminance is ignored)
01	full COLPF1 hires bypass enabled: PF1 (forced) after the GTIA PRIOR logic, 8-bit color regs
1x	hires PF1 ANx decoder level extension (before GTIA PRIOR logic), 8-bit color regs
x1	PF2 hires mod swaps PF2 to PF0 instead of PF3 (after the GTIA PRIOR logic)

The XCOLOR bits work independently: XCOLOR = 11 has both the 1x and the x1 behavior, and PF1 (forced) applies only to XCOLOR = 01. XCOLOR operates in the same way both for the global ANTIC/GTIA hires and for hires forced locally by an Attribute Map cell.

Layer seen by the PRIOR logic depending on XCOLOR and the ANx decoding

What the GTIA PRIOR logic itself sees after the decoding of the ANx stream from ANTIC depends only on XCOLOR[1]:

XCOLOR[1]	Set pixel seen by PRIOR as	Clear pixel seen by PRIOR as
0 (XCOLOR = 00, 01)	PF2	PF2
1 (XCOLOR = 10, 11)	PF1	PF2

For XCOLOR[1] = 0 the information that the pixel is set is invisible to the PRIOR logic (set and clear pixels look the same to it) and bypasses it as bPF1 for XCOLOR = 00 and as PF1 (forced) for XCOLOR = 01 - see the notation below.

Layer and color selected depending on XCOLOR and the PRIOR logic

Notation used in the tables below:

- **PF2HM** - a pixel to which the PF2 hires mod applies (see Cell Byte 1).
- **sPF0-sPF3** - the playfield layer at the GTIA output, after the PRIOR logic: sPFx means PFx, or "no layer" if the GTIA PRIOR logic blocked it because of a conflict with a PMG or another playfield.
- **PF3 (from sPF2)** - after the PRIOR logic, sPF2 (if present) becomes PF3 and the PF2 layer is removed.
- **PF0 (from sPF2)** - after the PRIOR logic, sPF2 (if present) becomes PF0 and the PF2 layer is removed.
- **bPF1** - (XCOLOR = 00) the luminance-only bypass of the PRIOR logic: the luminance (bits [3:1]) of the displayed color is replaced with the one of COLPF1, also when the pixel is covered by a PMG (the PMG color then gets the luminance of COLPF1). For the Overlay-PF/PMG priority and collision logic it is additionally a PF1 layer.
- **PF1 (forced)** - (XCOLOR = 01) the full PF1 layer forced after PRIOR: the result of the PRIOR logic is ignored and no PMG is displayed.

The first table gives the layers and the color selected by the GTIA after the PRIOR logic and the XCOLOR-dependent processing that follows it. These layers are used in the collision detection and in the Overlay priority decision, and the color is displayed unless the Overlay takes priority over it:

XCOLOR	Set pixel	Clear pixel	Set pixel, PF2HM	Clear pixel, PF2HM
00	sPF2 + bPF1	sPF2	PF3 (from sPF2) + bPF1	PF3 (from sPF2)
01	PF1 (forced)	sPF2	PF1 (forced)	PF0 (from sPF2)
10	sPF1	sPF2	sPF1	PF3 (from sPF2)
11	sPF1	sPF2	sPF1	PF0 (from sPF2)

In the Overlay priority and in the collision detection the layers specific to XCOLOR count as follows:

Layer	XCOLOR	Priority bit / COLDETECT bit
PF3 (from sPF2)	x0	6 (PF2 or PF3)
PF0 (from sPF2)	x1	4 (PF0)
bPF1	00	5 (PF1)
PF1 (forced)	01	5 (PF1)

NOTE

Whether a PMG is displayed over a PF2HM pixel is decided by the GTIA PRIOR logic as for a normal PF2 pixel. A PMG that has priority over PF2 covers the pixel (sPF2 is then "no layer"), so no swap takes place there and the PMG is visible. The swapped layer is displayed with the COLPF3 or COLPF0 color and counts as PF3 or PF0 in the Overlay priority and in the collision detection, but not in the PRIOR logic.

NOTE

Wherever luminance is mentioned above, it means the luminance part of the color index (bits [3:0]), not the contents of the RGB palette, which is completely independent of it.

Modification of RGB palettes

Palette modification is done by selecting palette through PSEL, color index through CSEL and then writing component values through CR, CG and CB.

VBXE allows for simultaneous display of 1024 colors (out of a 21-bit hardware palette of 2097152 colors; the least significant bit of the component color registers is discarded). There are 4 sets (palettes) with 256 user-predefined colors each.

The RGB palette for each of those 1024 (4 x 256) colors is updated as follows:

- write palette number (0-3) into PSEL register,
- write color index (0-255) into CSEL register,
- write R component color value into CR register,
- write G component color value into CG register,
- write B component color value into CB register.

NOTE

In GTIA cores palette 0 (default system colors) isn't modifiable.

CAUTION

Write to CB (but not to CR nor CG) will automatically increase CSEL, so one can write next color value without direct CSEL modification; however, if CSEL overflows from 255 to 0, PSEL will be unaffected, so if you want to modify the next palette, you will have to write the proper value into PSEL - otherwise, you will be modifying the same palette, only starting with color index 0.

Writes to each of the CR, CB and CG registers make an instant change on the screen, in the form of changing the color (if the given palette and color are currently being displayed) – the component color values are not buffered.

By default, after power on / core reboot, palette 0 contains default colors for GTIA layer PF/PMG. Palettes 1-3 have all components zeroed out (all colors black).

> CAUTION
> After modifying palette 0, the programmer is responsible for resetting it back to default values when the program exits. Since it is impossible to read back the current contents of a palette, palette 0 should only be modified if the program needs to use and modify all four of the VBXE core's palettes; otherwise, one of the palettes 1-3 should be used instead.

CSEL

Color SELect register.

bits [7:0]
Starting color index for RGB components modification
W-x

Before modifying RGB color components, one needs to write the color index (the one to modify, or the one from which to start modification) to CSEL register.

PSEL

Palette SELect register.

bits [7:2]	bits [1:0]
-	Palette number to modify
-	W-x (GTIA cores: W-0)

Before modifying RGB color components, one needs to write the palette number to PSEL register. PSEL, unlike CSEL, never increments automatically.

> CAUTION
> In GTIA cores PSEL is reset to 0 by every reset (soft or hard, regardless of PXGF), which blocks palette modification (palette 0 is not modifiable in these cores) until a non-zero palette number is written to PSEL again.

CR, CG and CB

RGB component registers.

bits [7:1]	bit 0
7-bit R/G/B component	-
W-x	-

Component value change is taken into account instantly after its register write. See the CAUTION note in Modification of RGB palettes regarding automatic CSEL increment on CB write.

7. MEMAC, VBXE VRAM bridge

The MEMAC is a part of the VBXE core responsible for allowing the system (CPU and ANTIC) access to the 512 KB VRAM installed on the VBXE.

The VBXE VRAM memory access can be accomplished through two independent "MEMAC windows" in 6502/ANTIC address space:

MEMAC-A - window of definable base address and size.
MEMAC-B - fixed size window at $4000 - $7FFF (a 16 KB window).

> **WARNING**
> In case of a MEMAC window in $4000-$7FFF address range, the programmer must disable extended PORTB memory in order to access the VBXE VRAM memory. This is because some memory extensions will have priority over MEMAC accesses (and some not - but this isn't a thing VBXE core is able to control). Such programs should enable one kind of memory window at a time.

An access to the VRAM made through the MEMAC costs VBXE 1 cycle of the PCLK clock (14.18/14.31 MHz) per byte being read or written. Looking from the side of the CPU or ANTIC, there is no difference between this access and any other access to the memory or I/O registers. Technically the VBXE disables the standard RAM and substitutes it by its own memory using the EXTSEL signal. Additionally the core is checking the CASINH signal from the MMU to be able to avoid conflicts with OS, BASIC or the CARTRIDGE ROM memories (these memories, if enabled, always have priority over MEMAC).

> **NOTE**
> If the MEMAC-A window is defined so it overlaps the MEMAC-B window, then MEMAC-A has priority over MEMAC-B. There is no conflict if MEMAC-A(B) is defined as ANTIC only accessible, and MEMAC-B(A) as 6502 only accessible - in this case windows can overlap partially or as a whole.

> **NOTE**
> OS, BASIC, CARTRIDGE ROM (if enabled) and hardware registers always have priority over enabled MEMAC windows.

> **NOTE**
> Devices that take over memory cycles (for example SUBcart) and the EXTSEL pull-up resistor: see Compatibility and hardware notes.

> **CAUTION**
> VBXE oriented applications should disable both MEMAC windows at exit.

RAMBO compatible shared extended memory

In case of cores with 'r' suffix (for example "fx v1.26r") additional function of the MEMAC is to emulate standard 256 KB RAM expansion ("RAMBO" - 256 KB of extended RAM with no separate access for ANTIC and CPU). Expanded memory is mapped from upper half of VBXE VRAM memory. One has to remember that this memory is shared with VRAM, and can be altered by VBXE oriented application! Careful and separate XRAM / VRAM access policy apply here too.

Cores with 'a' suffix or with no suffix (for example "fx v1.26a" or "fx 1.27") have no extended memory emulation, and should be used if standard memory expansion unit is already installed in the computer.

MEMAC-A

The MEMAC-A window can be placed at an arbitrary base address with granularity of $1000 bytes and can have sizes from 4 KB up to 32 KB:

```
Possible MEMAC-A window base addresses:

$0000, $1000, $2000, $3000,
$4000, $5000, $6000, $7000,
$8000, $9000, $A000, $B000,
$C000, $D000, $E000, $F000

Possible MEMAC-A window sizes:

$1000 (4 KB)
$2000 (8 KB)
$4000 (16 KB)
$8000 (32 KB)
```

> **NOTE**
> There is no window wrap to address $0000 in case of placing it at the end of address space. If window does not fit in 64 KB, then it is truncated.

The MEMAC-A window can be:

- disabled (standard RAM will be there)
- enabled for CPU (CPU "sees" VRAM, ANTIC "sees" Atari RAM)
- enabled for ANTIC (ANTIC "sees" VRAM, CPU "sees" Atari RAM)
- enabled for both ANTIC and the CPU

> **NOTE**
> ANTIC and the CPU always see the same fragment of VRAM (they have the same bank register - MEMS).

MEMAC-A is controlled by two I/O registers:

MEMAC_CONTROL

Alias MEMC. This register selects MEMAC-A window base address (in 6502 space) and size. After the RESET 0 is written to the MCE and MAE bits (and to the MGE bit of MEMAC_BANK_SEL, as well as to the MEMAC-B enable bits) so the window is disabled, remaining bits are left untouched.

bits [7:4]	bit 3	bit 2	bits [1:0]
MA_BASE	MCE	MAE	MA_SIZE
RW-x	RW-0	RW-0	RW-x

MA_BASE: Base address of MEMAC-A window in 6502 address space

MCE: MEMAC CPU ENABLE - when set, the MEMAC-A window will be accessible by 6502. This bit is reset on RESET.

MAE: MEMAC ANTIC ENABLE - when set, the MEMAC-A window will be accessible by ANTIC. This bit is also reset on RESET.

> NOTE
> The MEMAC-A window will stay disabled if the MGE bit (MEMAC GLOBAL ENABLE) in MEMAC_BANK_SEL register is reset, despite of MCE and MAE state.

MA_SIZE: size of MEMAC-A bank and window:

MA_SIZE	bank/window size
00	4 KB
01	8 KB
10	16 KB
11	32 KB

MEMAC_BANK_SEL

Alias MEMS. This register decides of which part of the VRAM is available through the MEMAC-A window. Additionally there is MGE bit which globally enables the window.

MA_SIZE	bit 7	bits [6:0]
4 KB	MGE	MA_BANK <0 ... 127>
8 KB	MGE	MA_BANK [6:1] <0 ... 63>, bits [0] unused
16 KB	MGE	MA_BANK [6:2] <0 ... 31>, bits [1:0] unused
32 KB	MGE	MA_BANK [6:3] <0 ... 15>, bits [2:0] unused
-	RW-0	RW-x

MA_BANK selects the VRAM area visible in the MEMAC-A window. Each bank size is equal to MA_SIZE selected. MA_BANK selection is always aligned to msb of this field, leaving lower bits unused for 8, 16 and 32 KB sizes.

MGE: MEMAC GLOBAL ENABLE - when set enables MEMAC-A window as previously defined in MEMAC_CONTROL register. This global enable bit allows to use MEMAC_CONTROL register as configuration only, so the programmer can set it once at the beginning of the code, and use only

MEMAC_BANK_SEL register during the run.

MEMAC-B

The MEMAC-B window is located at fixed $4000 address and its size is also fixed to $4000 (16 KB).

The MEMAC-B window can be:

- disabled (standard RAM will be there)
- enabled for CPU (CPU "sees" VRAM, ANTIC "sees" Atari RAM)
- enabled for ANTIC (ANTIC "sees" VRAM, CPU "sees" Atari RAM)
- enabled for both ANTIC and the CPU

MEMAC_B_CONTROL

Alias MEMB. MEMAC-B window enable and bank selection register.

bit 7	bit 6	bit 5	bits [4:0]
MBCE	MBAE	-	MB_BANK <0 ... 31>
W-0	W-0	-	W-x

MBCE - MEMAC-B CPU ENABLE - when set 6502 can access VRAM.
MBAE - MEMAC-B ANTIC ENABLE - when set ANTIC can access VRAM.
MB_BANK - 32 banks, 16 KB size each gives access to whole 512 KB of VRAM.

> **NOTE**
> Unlike MEMAC-A registers, MEMAC_B_CONTROL register is write-only.

8. XDL

The XDL (eXtended Display List) is a list of commands, that controls the Overlay display and the Attribute Map of the VBXE. The XDL is loaded to the VBXE VRAM through the MEMAC buffers (see the MEMAC description). It may be loaded to any location inside the 512 KB VBXE VRAM, and it is pointed to by the XDL_ADR (XDL_ADR0/1/2). The XDL processing starts, when the bit XDL_ENABLE in the VIDEO_CONTROL register is set to 1. There are no limitations on the XDL's size, and its structure is vertical: as with the ANTIC DL, XDL processing begins at the top of the display.

> **NOTE**
> Unlike the ANTIC DL, the XDL has no jump instruction.

The XDL is structured as follows:

```
XDLC (2 bytes)
associated data (0-20 bytes)
XDLC (2 bytes)
associated data (0-20 bytes)
...
...
XDLC with the XDLC_END marker (2 bytes)
associated data (0-20 bytes)
```

XDL_ADR (XDL_ADR0/1/2)

XDL_ADR0 bits [7:0]	XDL_ADR1 bits [7:0]	XDL_ADR2 bits [7:3]	XDL_ADR2 bits [2:0]
XDL_ADR[7:0]	XDL_ADR[15:8]	-	XDL_ADR[18:16]
W-x	W-x	-	W-x

Bits [7:0], [15:8] and [18:16] (respectively) of the XDL address in the VBXE VRAM. The XDL address in the VRAM consists of 19 bits. The XDL can be located anywhere in the VRAM and start at a single byte boundary. XDL_ADR0, XDL_ADR1 and XDL_ADR2 have to be loaded before the XDL processing is enabled via the XDL_ENABLE bit in the VIDEO_CONTROL register.

> **CAUTION**
> A change of XDL_ADR0, XDL_ADR1 or XDL_ADR2 is only taken into account at the end of VSYNC.

> **NOTE**
> The XDL is processed only in scanlines that are not blanked at the start of the Overlay line - blanked scanlines (e.g. during the vertical blanking) do not consume XDL records nor repeat counts.

XDLC control word

The "XDLC" stands for the XDL Control word. This word always occupies two bytes. Every bit of the control word has different meaning. A part of the bits enables or disables display functions, and another part of them carries an information, whether the XDL Controller should fetch associated data, and what data it would be. The bits do not depend on each other, any combination of them can be used at any time; it is, for example, possible to load the Overlay video memory address leaving the Overlay switched off. The XDLC word gets processed before the line starts to be displayed.

Bytes of XDLC are numbered starting from 1 (LSB). Bits inside them are numbered starting from 0 (lsb), we use byte.bit notation below.

```
1.0 XDLC_OV_TM_ON - enable Overlay text mode.
1.1 XDLC_OV_GM_ON - enable Overlay graphic (pixel) mode.
1.2 XDLC_OV_OFF - disable Overlay.
```

Normally, no more than one of these bits should be set at once. Setting more than one of the above bits in one XDLC will cause the Overlay to be switched off. The Overlay is disabled by default at the top of the screen. Leaving all of these bits zeroed will preserve the current state of the Overlay. It can be useful when you only want to change the font or scrolling values. See also XDLC_OV_HR and XDLC_OV_LR flags.

```
1.3 XDLC_MAP_ON - enable color attributes (Attribute Map).
1.4 XDLC_MAP_OFF - disable color attributes (Attribute Map).
```

Setting more than one of the above bits will disable the Attribute Map. It is always disabled by default, i.e. at the top of the screen. Leaving all of these bits zeroed will preserve the current state of the map.

The Attribute Map data is (re)loaded:

- when the map gets enabled by XDLC_MAP_ON (while it was disabled),
- when a new address is set by XDLC_MAP_ADR while the map is enabled,
- automatically, when the map is enabled and the last scanline of the current row of cells has been displayed - the map address is then advanced by MAP_STEP and the data of the next row of cells is loaded.

XDLC_MAP_ON while the map is already enabled does not reload the map data by itself. Setting XDLC_MAP_ADR while the map is disabled (and without XDLC_MAP_ON) only stores the new address.

```
1.5 XDLC_RPT_LN - continue display for next REPEAT scanlines.

  associated data byte:

  1. REPEAT[7:0]
```

After the current line was displayed, there will be REPEAT (<0 ... 255>) consecutive scanlines to use the same settings, and the XDL will not be processed for them. A REPEAT value of 0 is equivalent to not setting XDLC_RPT_LN at all. For example, if you want to display a line of text, enable the text mode with XDLC_OV_TM_ON, and set XDLC_RPT_LN giving 7 as REPEAT. As a result 8 scanlines will be produced forming a line of the text mode.

```
1.6 XDLC_OV_ADR - set the address and step of the Overlay display memory.

  associated data bytes:

  1. OV_ADR[7:0]
  2. OV_ADR[15:8]
  3. OV_ADR[18:16]
  4. OV_STEP[7:0]
  5. OV_STEP[11:8]
```

The address of the Overlay display memory is a 19-bit value initialized here by OV_ADR. The unsigned OV_STEP parameter defines, how many bytes should be added to the address every mode line, so that it would point to the data for the next line - in other words, OV_STEP is the difference between the addresses of the start (left edge) of consecutive lines. The OV_STEP may be an unsigned value from 0 to 4095. For text mode, standard width (unscrolled) Overlay and linear screen buffer OV_STEP may be 160 as 80 chars use 160 bytes of VRAM, two bytes for every char. For pixel standard width SR mode that would be usually 256. In a pixel mode the OV_STEP gets added to the OV_ADR after every scanline. In the text mode the OV_STEP is added to the OV_ADR after last mode scanline (text mode line occupies eight scanlines).

```
1.7 XDLC_OV_SCRL - set scrolling values for the text mode.

  associated data bytes:

  1. OV_HSCROLL[2:0]
  2. OV_VSCROLL[2:0]
```

- OV_HSCROLL is a value ranged <0 ... 7>, where 0 is a not scrolled line, and 7 is a line scrolled 7 VBXE HR pixels to the left.
- OV_VSCROLL is a value ranged <0 ... 7>, where 0 is not a vertically scrolled line, and 7 is a line scrolled 7 scanlines up (i.e. starting from last text mode scanline).

By default, at the top of the screen, OV_HSCROLL = OV_VSCROLL = 0. Scrolling values can be changed in every scanline. Setting the XDLC_OV_SCRL does not enable the scroll (which is always enabled), but only sets the values of the scrolling registers. The OV_HSCROLL value will be used for every consecutive scanline until the XDL changes it. OV_VSCROLL, however, is only the STARTING value of the vertical scroll counter, which selects the scanline (font row) of the text mode line being displayed: the counter is incremented after every text mode scanline, and the OV_STEP is added to the Overlay address when it reaches 7 (after which it wraps to 0). As a result, the first text mode line displayed after setting OV_VSCROLL = v has only 8 - v scanlines, and the following lines have the full 8 scanlines again. The counter does not advance while the text mode is not displayed.

```
2.0 XDLC_OV_CHBASE - set text mode font base.

  associated data byte:

  1. OV_CHBASE[7:0]
```

The font set contains 256 characters, 8x8 pixels each, and resides in the VBXE VRAM memory. Every font set (pointed by OV_CHBASE) must start at 2K boundary (i.e. font set effective VRAM address is OV_CHBASE * 2048). Therefore up to 256 font sets can be loaded to the 512 KB VRAM. As everything else, the font is stored in the VBXE VRAM through the MEMAC buffers.

```
2.1 XDLC_MAP_ADR - set the address and step of the Attribute Map.

  associated data bytes:

  1. MAP_ADR[7:0]
  2. MAP_ADR[15:8]
  3. MAP_ADR[18:16]
  4. MAP_STEP[7:0]
  5. MAP_STEP[11:8]
```

The Attribute Map may start at any location (single byte boundary) in the VBXE VRAM.

The MAP_STEP value is automatically added to the Attribute Map address when the Attribute Map cell has been completely displayed vertically (i.e. after displaying the last – bottom – scanline of the cell), unless the settings get explicitly changed by the XDL - in other words, MAP_STEP is the difference between the addresses of the start (left edge) of consecutive rows of Attribute Map cells.

> **NOTE**
> MAP_STEP bits [1:0] are ignored by the hardware - only multiples of 4 bytes are actually usable as a step value. As every Attribute Map cell is 4 bytes long, this is not a practical limitation.

```
2.2 XDLC_MAP_PAR - set scrolling values, width and height of an Attribute Map cell.

  associated data bytes:

  1. MAP_HSCROLL[4:0]
  2. MAP_VSCROLL[4:0]
  3. MAP_WIDTH[4:0]
  4. MAP_HEIGHT[4:0]
```

- MAP_HSCROLL is a value of range <0 ... 31>, where 0 means that the line is not scrolled, and 31 – that the line is scrolled 31 SR/hires pixels to the left.
- MAP_VSCROLL is a value of range <0 ... 31>, where 0 means that the line is not scrolled, and 31 – that the line is scrolled 31 scanlines up.
- MAP_WIDTH – the width of the Attribute Map cell in pixels, range <7 ... 31> == 8 to 32 SR/hires pixels.
- MAP_HEIGHT – the height of the Attribute Map cell in scanlines <0 ... 31> == 1 to 32 lines.

Default values (at the top of the screen) are MAP_HSCROLL = MAP_VSCROLL = 0, MAP_HEIGHT = MAP_WIDTH = 7 (the Attribute Map cell size 8x8).

The Attribute Map cell size and scrolling values may be changed in any scanline. The hscroll unit for the map is 1 pixel (SR/hires).

MAP_VSCROLL is only the STARTING value of the counter of the current scanline within the cell. The counter is incremented after every scanline while the Attribute Map is enabled; after the last (bottom) scanline of the cell, it wraps to 0 and MAP_STEP is applied. As a result, the first row of cells displayed after setting MAP_VSCROLL = v is only (MAP_HEIGHT + 1 - v) scanlines tall, and the following rows have the full height again.

> **NOTE**
>
> MAP_HSCROLL and MAP_VSCROLL should never get greater than the respective values of MAP_WIDTH and MAP_HEIGHT. Greater values are not rejected by the hardware, and the result is well defined, although certainly not useful. The in-cell pixel counter and the in-cell scanline counter are 5-bit counters that are compared for equality with MAP_WIDTH / MAP_HEIGHT; a counter that starts above its limit therefore does not stop at the limit, but counts up to 31, wraps to 0 and only then counts up to MAP_WIDTH / MAP_HEIGHT. As a result:
>
> - with MAP_HSCROLL > MAP_WIDTH, the first cell of every line (cell 0 of the row, shown with its own attributes all the way) is (32 - MAP_HSCROLL) + MAP_WIDTH + 1 pixels wide, instead of MAP_WIDTH + 1 - MAP_HSCROLL; the following cells have the normal width again;
> - with MAP_VSCROLL > MAP_HEIGHT, the first row of cells is (32 - MAP_VSCROLL) + MAP_HEIGHT + 1 scanlines tall, instead of MAP_HEIGHT + 1 - MAP_VSCROLL; the following rows have the normal height again.

> **NOTE**
>
> Cells narrower than 8 pixels (MAP_WIDTH < 7) are outside of the specified range, because the internal Attribute Map buffer holds the attributes of only 43 cells of one row (43 x 4 bytes, loaded as described in Blitter cycle usage). 43 cells of the minimum specified width of 8 pixels cover 344 pixels, which is enough for the widest Overlay (WIDE = 336 SR pixels). With narrower cells a line may need more than 43 cells; the buffer then wraps around: the cell counter goes from cell 42 back to cell 0, so the 44th cell of the line is displayed with the attributes of cell 0 again, the 45th with those of cell 1, and so on, until the end of the line.

Setting XDLC_MAP_PAR does not enable the map to scroll (this function is always on), it only loads the scrolling registers. The values loaded will be used in consecutive scanlines until they are explicitly changed with the XDL.

```
2.3 XDLC_OV_ATT - set the display width, palettes and Overlay priority.

  associated data bytes:

  1. ATT_1 - Overlay / Attribute Map width + palette change
  2. ATT_2 - main Overlay / GTIA priority
```

ATT_1:

bits [7:6]	bits [5:4]	bits [3:2]	bits [1:0]
XDL_PF_PALETTE	XDL_OV_PALETTE	-	OV_MAP_WIDTH

OV_MAP_WIDTH: Overlay and Attribute Map width:

OV_MAP_WIDTH	meaning
0	NARROW (256 pixels, as ANTIC narrow)
1	NORMAL (320 pixels, as ANTIC normal)
2	WIDE (336 pixels; in this mode the display is 8 pixels wider at both sides, than NORMAL)
3	reserved

In the NARROW and NORMAL modes not only the width but also the horizontal position of the Overlay and the Attribute Map coincides exactly with the ANTIC playfield of the same width.

The default (at the top of the screen) width is NORMAL (320 pixels).

XDL_OV_PALETTE: the palette for Overlay active from this screen line. By default, Overlay uses palette 1 from the top of the screen.

XDL_PF_PALETTE: the palette for PLAYFIELD and PMG active from this screen line. By default it's palette 0 from the top of the screen; this default can be changed with GTIA_PALETTE in FEATURE_CONTROL.

The "default from GTIA_PALETTE" behavior only lasts until the first XDLC_OV_ATT of the frame is processed. From that point on, the XDL_PF_PALETTE value loaded by the XDL takes over and remains in effect for the PLAYFIELD/PMG palette for the rest of the frame (i.e. until the next VSYNC / top of screen), even on scanlines where a new XDLC_OV_ATT is not issued. There is no way to make a later part of the same frame revert to using GTIA_PALETTE again - once XDL_PF_PALETTE has been used, only another XDLC_OV_ATT can change the active PLAYFIELD/PMG palette until the next frame.

> **NOTE**
> If the Attribute Map is turned on, the palettes on the area it covers are selected by the Attribute Map cells - see Cell Byte 4.

ATT_2:

Main priority. See Overlay-PF/PMG priority for the meaning of each bit — the same rules apply here.

The default value (at the top of the screen) of the priority is 255. The main priority is not taken into account, when the Attribute Map is enabled. In this case it is the map, that decides, which one of the 4 predefined priorities P0 ... P3 will be used for the particular part of the screen (Attribute Map cell).

```
2.4 XDLC_OV_HR - enable the HR (High Resolution) pixel mode.
```

This bit is only taken into account, when XDLC_OV_GM_ON == 1. The HR mode (see Pixel modes) has a resolution of 640 pixels horizontally for the NORMAL display width and can display 16 colors, from $00 to $0F, in the current Overlay palette. Each pixel is represented by a nibble of data (4 bits) in the VBXE VRAM. Each data byte contains 2 nibbles: the most significant nibble represents the leftmost pixel.

```
2.5 XDLC_OV_LR - enable the Low Resolution (LR) mode.
```

This bit is only taken into account, when XDLC_OV_GM_ON == 1. The LR mode (see Pixel modes) has a resolution of 160 pixels horizontally for the NORMAL display width. The number of displayable colors is the same as in the standard display mode (256).

```
2.6 - reserved (=0).
```

```
2.7 XDLC_END - XDL end (the last XDL record), wait for VSYNC.
```

XDLC_END tells the XDL controller that after processing of this XDLC is finished, it has to wait for the vertical sync pulse, and then start processing the XDL from the beginning. Once all lines described by this XDLC have been displayed, the Overlay and the Attribute Map are switched off, until re-enabled by XDL/XDLC after VSYNC.

Associated data

The associated data bytes of an XDLC record lie in the XDL immediately after its two XDLC bytes, before the next XDLC record. Only the bits listed above as requiring associated data (XDLC_RPT_LN, XDLC_OV_ADR, XDLC_OV_SCRL, XDLC_OV_CHBASE, XDLC_MAP_ADR, XDLC_MAP_PAR and XDLC_OV_ATT) contribute bytes; a bit that is 0, or a bit that carries no data of its own (such as XDLC_OV_TM_ON, XDLC_OV_GM_ON, XDLC_OV_OFF, XDLC_MAP_ON, XDLC_MAP_OFF, XDLC_OV_HR, XDLC_OV_LR or XDLC_END), contributes none.

When more than one data-carrying bit is set in the same XDLC, their associated data bytes are concatenated in the order of the bits within the XDLC word, from the lowest-numbered bit of byte 1 to the highest-numbered bit of byte 2 (i.e. XDLC_RPT_LN, then XDLC_OV_ADR, then XDLC_OV_SCRL, then XDLC_OV_CHBASE, then XDLC_MAP_ADR, then XDLC_MAP_PAR, then XDLC_OV_ATT). With every data-carrying bit set at once, this amounts to the maximum of 20 associated data bytes mentioned in the XDL structure above.

> **CAUTION**
> The XDL parameters that have no default value listed (OV_ADR, OV_STEP, OV_CHBASE, MAP_ADR and MAP_STEP) are not reset at the top of the screen - they keep the values from the previous frame (and are undefined after RESET).

9. Overlay modes

This section describes the display modes of the Overlay.

The bit combinations of the XDLC that enable the Overlay display modes of the VBXE FX core:

All column headers below (except "mode") name bits of the XDLC control word.

OV_TM_ON	OV_GM_ON	OV_HR	OV_LR	mode
0	1	0	0	Pixel SR
0	1	1	0	Pixel HR
0	1	0	1	Pixel LR
0	1	1	1	Forbidden, HR mode with LR data rate
1	0	X	X	"80 column" HR text mode
1	1	X	X	Forbidden, works as XDLC_OV_OFF

> **NOTE**
> In the forbidden combination of OV_HR and OV_LR, the Overlay data is fetched at the LR rate (one byte per two SR pixels, i.e. 128 / 160 / 168 bytes per line for NARROW / NORMAL / WIDE), while the pixels are shown at the HR rate. Each byte is therefore displayed as four HR pixels: its high nibble, low nibble, high nibble and low nibble again - the pixel pair is repeated, not each pixel doubled. The colors are selected as in the HR mode.

The address of the displayed Overlay data (used both by the pixel modes and the text mode) is initialized via OV_ADR. After displaying a mode line, this address is automatically increased by the OV_STEP value programmed in the XDL. A mode line is not the same as a scanline: in the pixel modes a mode line is a single scanline, while in the text mode a mode line spans 8 scanlines.

Pixel modes

The vertical resolution of every pixel mode is defined by the XDL structure. The horizontal resolution depends on the display width selected via the XDL:

mode	colors	pixels per byte	NARROW	NORMAL	WIDE
SR (Standard Resolution)	256	1	256	320	336
LR (Low Resolution)	256	1	128	160	168
HR (High Resolution)	16	2	512	640	672

> **TIP**
> The number of colors available in a given mode can be increased by locally changing the Overlay palette by means of the Attribute Map.

The SR mode (Standard Resolution) and the LR mode (Low Resolution)

In both the SR and LR modes, every pixel is represented by a byte (color index) in VBXE VRAM.

SR corresponds to ANTIC/GTIA hires resolution, and LR corresponds to ANTIC/GTIA CCR (Color Clock Resolution).

The HR mode (High Resolution)

A byte of the video memory contains information about 2 pixels, 4 bits each:

bits [7:4]	bits [3:0]
Leftmost pixel	Rightmost pixel

Each pixel selects the color index 0 ... 15 from the currently selected (locally or globally) Overlay palette.

Transparent colors

When the NO_TRANS bit in the VIDEO_CONTROL register is set to 1, no color is transparent in either pixel mode, regardless of the TRANS15 bit state.

When the NO_TRANS bit is 0:

- TRANS15 = 0: SR / LR mode - color index 0 is transparent; HR mode - the nibble value 0 is transparent.
- TRANS15 = 1: additionally to the above, SR / LR mode - each color whose index [3:0] == 15 (i.e. $0F, $1F, $2F and so on up to $FF) is transparent; HR mode - the nibble value $F is transparent.

The TRANS15 bit allows to create invisible objects, which can be used, for example, as fields to detect collisions with other Overlay objects - the collisions here being those detected by the Blitter during its operation (see blt_collision_mask), not the raster collision detection described in Overlay-PF/PMG collision detection.

Text mode

This is a text mode with horizontal resolution of 64/80/84 characters (the width is selected via the XDL, the vertical resolution is defined by the XDL structure) with 128 foreground colors and, depending on the attribute, a transparent or colored background (see the attribute table below). The text mode uses a 256-character 8x8 pixel font set located in the VRAM, at an address selected via the XDLC_OV_CHBASE field of the XDL. The video memory structure is as follows:

char (1 byte), attribute (1 byte), char, attribute, char, attribute, ... and so on.

The full line of the text mode occupies the number of bytes in the memory equal to 2x line width in characters. If a non-zero hscroll value is used, remember to lengthen the line in memory by at least one additional character (i.e. 2 bytes).

The char is a value 0-255 and defines which character of the 256-character font will be displayed.

The attribute (AT) has the following structure: bit 7 decides, whether the character's background is transparent or it has a color; bits [6:0] hold the color index, i.e. select a color from the active (locally or globally) Overlay palette. The resulting foreground and background colors also depend on the NO_TRANS and TRANS15 bits in the VIDEO_CONTROL register:

AT bit 7	NO_TRANS	TRANS15	foreground color index	background color index
0	0	0	AT bits [6:0]	transparent
0	0	1	AT bits [6:0]; transparent if AT bits [3:0] == 15	transparent
0	1	X	AT bits [6:0]	128
1	0	0	AT bits [6:0]	AT bits [6:0] + 128
1	0	1	AT bits [6:0]; transparent if AT bits [3:0] == 15	AT bits [6:0] + 128; transparent if AT bits [3:0] == 15
1	1	X	AT bits [6:0]	AT bits [6:0] + 128

Text mode scroll

Horizontal and vertical scrolling of the text mode is set via XDLC_OV_SCRL: OV_HSCROLL scrolls the line 0 ... 7 VBXE HR pixels to the left, and OV_VSCROLL scrolls it 0 ... 7 scanlines up (i.e. starting from the last text mode scanline). Both default to 0 at the top of the screen and can be changed in any scanline. OV_HSCROLL applies to every consecutive scanline until the XDL changes it, while OV_VSCROLL only sets the starting scanline of the current text mode line (see XDLC_OV_SCRL).

10. Overlay-PF/PMG priority

There are several priority registers defining the order data of PF/PMG and Overlay content are drawn:

- main priority register - by default, at the top of the screen, set to 255 (which means that the Overlay data has higher priority than any of PF/PMG colors), and later changed by the XDL as the image is displayed.
- P0-P3 registers - used INSTEAD of the main priority register when the Attribute Map is enabled. Their meaning is the same as for the main priority register. The content of the 4th byte of Attribute Map cell decides which (P0, P1, P2, or P3) register is active for a given cell. See the register description for the further details.

Priority bits

The meaning of the bits in the main priority register (and in P0-P3 registers) is as follows:

	Value = 1	**Value = 0**
bit 0	Overlay over PM0	Overlay covered by PM0
bit 1	Overlay over PM1	Overlay covered by PM1
bit 2	Overlay over PM2	Overlay covered by PM2
bit 3	Overlay over PM3	Overlay covered by PM3
bit 4	Overlay over PF0	Overlay covered by PF0
bit 5	Overlay over PF1	Overlay covered by PF1
bit 6	Overlay over PF2 and PF3	Overlay covered by PF2 and PF3
bit 7	Overlay over COLBAK	Overlay covered by COLBAK

These bits are evaluated independently and it is enough for a single one of them to indicate that the Overlay should be shown for the Overlay pixel to actually be displayed - even if more than one of the PF/PMG colors are active at the same time and the resulting per-color priorities would otherwise conflict. Only the PF/PMG colors that are actually visible after the ANTIC/GTIA's own priority resolution (GTIA PRIOR register) are taken into account - a color hidden by the GTIA priority does not take part in this evaluation.

In the ANTIC/GTIA hires modes, the way set hires pixels take part in this evaluation depends on XCOLOR - see Layer and color selected depending on XCOLOR and the PRIOR logic.

In GTIA modes (9,11) (16 shades / 16 colors) the ANTIC / GTIA generated pixels are all treated as COLBAK color, so, if not using the PMG, only bit 7 of the priority register controls if the Overlay is visible or not.

P0, P1, P2, P3

Overlay - PF/PMG Priority select I/O registers used when Attribute Map is enabled (on the area covered by the Attribute Map).

bits [7:0]
PRIORITY
W-x

> **CAUTION**
> Since these registers have no defined state after RESET / core reboot, when enabling the Overlay and the Attribute Map, the programmer must remember to initialize at least the P0-P3 register(s) that will actually be used.

11. Overlay-PF/PMG collision detection

It is possible to detect a collision between the displayed Overlay data (both text mode and graphics mode) and any of the PF/PMG colors (except COLBAK) and/or Attribute Map cell with CATT bit set. Collision detection is automatic, and takes place while VBXE draws the display.

This is done using the COLMASK, COLCLR and COLDETECT registers, described below.

> **NOTE**
> Overlay-PF/PMG collisions are not detected with transparent Overlay colors, so both the NO_TRANS and TRANS15 bits in the VIDEO_CONTROL register matter here, in the same way as they decide transparency for display purposes (see Overlay modes).

Overlay-PF/PMG collisions are detected independently of the display priority established by the Overlay-PF/PMG priority mechanism (see Overlay-PF/PMG priority) - a collision is reported regardless of whether the Overlay or the PF/PMG color would actually end up being displayed. However, only the PF/PMG colors that are visible after the ANTIC/GTIA's own priority resolution (GTIA PRIOR register) are taken into account - a color hidden by the GTIA priority does not produce a collision with the Overlay. In the ANTIC/GTIA hires modes the way set hires pixels are detected depends on XCOLOR - see Layer and color selected depending on XCOLOR and the PRIOR logic.

COLMASK, COLDETECT and COLCLR registers

COLMASK

bits [7:0]
Collision detection mask
W-x

The mask that configures the 'sensitivity' of the raster collision detector, allowing to detect collisions between the Overlay and the PF/PMG colors of the ANTIC/GTIA. Any combination of bits is allowed.

COLMASK bit	A collision will be detected if the Overlay color index (regardless of the Overlay mode) falls within the following range: (the palette number is insignificant)
0	0-31
1	32-63
2	64-95
3	96-127
4	128-159
5	160-191
6	192-223
7	224-255

COLDETECT

bits [7:0]
Collision code
R-x

The latch register holding the collision code. As the display is being generated, collisions are detected in the process and latched here; a bit set to 1 means that a collision has been detected. Any combination of bits is possible - software collision checking should be done with the appropriate AND mask. The register has no defined state after RESET - write to COLCLR to clear it before use.

COLDETECT bit	If set, the collision is detected between Overlay objects and:
0	PM0 color
1	PM1 color
2	PM2 color
3	PM3 color
4	PF0 color
5	PF1 color
6	Colors PF2 or PF3
7	Attribute Map cell with CATT bit set (for an Overlay pixel selected by COLMASK)

COLCLR

bits [7:0]
Any value (the value written is not used)
W-x

Writing any value here invalidates the detected collision, by clearing the COLDETECT register. This does not, however, stop collision detection itself, so new collisions will get latched again as soon as all the conditions for them are met.

12. Attribute Map

The Attribute Map allows to locally (i.e. within an Attribute Map cell of 8x1 up to 32x32 SR/hires pixels) change the colors PF0, PF1 and PF2, override the main Overlay priority over the ANTIC/GTIA to one of four predefined priorities, change the local color palette for both ANTIC and Overlay screen modes, and change the resolution of the display generated by the ANTIC/GTIA from ANTIC/GTIA hires (GR.8) to CCR (Color Clock Resolution = GR.15) or vice versa. Additionally, in ANTIC/GTIA hires mode, it can locally swap PF2 for COLPF0 or COLPF3 in an independently patterned way, via byte 1 of the cell data - see below.

Consequently, the Attribute Map greatly extends the graphic capabilities of your Atari machine even if the proper Overlay modes are not in use.

Characteristics

- the Attribute Map cell size (X x Y): 8x1 up to 32x32 SR/hires pixels.
- Attribute Map address in the VBXE VRAM: no limitations.
- automatic update of the address after displaying a full line of the map: programmable in the range <0 ... 4092> bytes, in multiples of 4.
- horizontal and vertical scrolling by 1 pixel of ANTIC/GTIA hires, controlled by the XDL. It is possible to change the register values in any scanline.
- every Attribute Map cell is defined by a set of 4 bytes stored consecutively in the VRAM (see Attribute Map cell data below).

The Attribute Map is completely controlled by the XDL, i.e. its VRAM address, scrolling registers, Attribute Map cell size etc. are defined inside the XDL list.

Attribute Map cell data

Every Attribute Map cell is defined in the VBXE VRAM by four consecutive bytes:

Cell Byte 1: PF0 color / PF2 hires mod

bits [7:0]
CCR mode: local substitute of the GTIA COLPF0 register. ANTIC/GTIA hires mode: PF2 hires mod, see below.

In **CCR** mode byte 1 is simply a local substitute for the COLPF0 register, exactly the same way bytes 2 and 3 below are local substitutes for COLPF1 and COLPF2.

In **ANTIC/GTIA hires** mode (whether the globally selected ANTIC display mode, or a mode forced locally by the RES bit of byte 4 below), byte 1 is instead used as an 8-bit **PF2 hires mod**: a bit pattern laid over the Attribute Map cell, starting from its left edge and scrolled together with it by MAP_HSCROLL. Each bit controls one horizontal segment of the cell (leftmost segment = PF2 hires mod bit 7, the next one = bit 6, and so on down to bit 0); a set bit marks the segment as modified. The segment width depends on the configured MAP_WIDTH, see the table below.

Example: the PF2 hires mod `%10110010` (bit 7 = leftmost segment) laid over cells of different widths, with MAP_HSCROLL = 0. In the last column every pixel of the cell is shown as the PF2 hires mod bit of the segment it falls in (1 = the PF2 hires mod is applied to that pixel, 0 = it is not):

Cell width (pixels)	Segment width (pixels)	Segments used	PF2 hires mod bits used	PF2 hires mod over the cell
8	1 (widths up to 8)	8	7...0	`10110010`
12	2 (widths 9...16)	6	7...2	`110011110000`
16	2 (widths 9...16)	8	7...0	`1100111100001100`
20	4 (widths 17...32)	5	7...3	`11110000111111110000`
32	4 (widths 17...32)	8	7...0	`11110000111111110000000011110000`

For every hires PF2 pixel that falls inside a segment whose PF2 hires mod bit is set, the pixel is displayed using the global COLPF3 register (XCOLOR[0] = 0, default) or COLPF0 register (XCOLOR[0] = 1) instead of PF2's normal (locally substituted, see byte 3 below) color, see XCOLOR - effectively adding an independently patterned hires color to the standard ones. Pixels outside the PF2 hires mod (bit = 0) keep displaying PF2 normally. The swap is done after the GTIA PRIOR logic, on the resulting sPF2 layer, so the PMG priority is the one of PF2 (a PMG with priority over PF2 stays visible over such pixels) - see Layer and color selected depending on XCOLOR and the PRIOR logic.

Cell Byte 2: PF1 color

bits [7:0]
Local substitute of the GTIA COLPF1 register

Cell Byte 3: PF2 color

bits [7:0]
Local substitute of the GTIA COLPF2 register

> NOTE
> COLPF3 has no matching field in the Attribute Map cell data - it always keeps using the global register value, in both CCR and hires mode.

Cell Byte 4: control bits

bits [7:6]	bits [5:4]	bit 3	bit 2	bits [1:0]
MAP_PF_PALETTE	MAP_OV_PALETTE	CATT	RES	PS

MAP_PF_PALETTE – local palette selection for normal PF/PMG objects. It may be any of the four palettes.

MAP_OV_PALETTE – local palette selection for the Overlay. It may be any of the four palettes.

> **CAUTION**
> The Color Palette assigned here has priority over the one selected in the XDL (XDL PF/OV PALETTE), meaning that, if the Attribute Map is active, then on the area it covers, pixels are colored with palettes assigned to the Attribute Map cells, rather than the default palette, or whichever palette the XDL has assigned.

CATT - collision detection auxiliary bit. If set, then it is possible to detect collisions of Overlay pixels with the Attribute Map cells during the scanline - see COLDETECT bit 7 in Overlay-PF/PMG collision detection.

RES – local change ANTIC/GTIA hires <-> CCR (1 = enabled). This bit "reverses" the resolution selected by the standard ANTIC DL, changing the mode locally (within the particular Attribute Map cell) from ANTIC/GTIA hires into CCR or vice versa.

PS – selection of one of the 4 predefined Overlay-PF/PMG priorities.

PS	register selected
00	Priority register P0
01	Priority register P1
10	Priority register P2
11	Priority register P3

Within the active Attribute Map the global GTIA color registers COLPF0, COLPF1 and COLPF2, and the global priority register, are not used - with one exception: in ANTIC/GTIA hires mode, byte 1's PF2 hires mod (see above) selectively substitutes the global COLPF0 or COLPF3 register for PF2 pixels with the PF2 hires mod applied.

> **NOTE**
> The width of the Attribute Map may be forced to correspond to the wide/normal/narrow ANTIC display. This is accomplished using the XDL, see XDLC_OV_ATT.

Resolution swap (RES bit)

Normally ANTIC decides (at the beginning of the scanline) how GTIA should interpret the data it sends:

- as ANTIC/GTIA hires mode (modes 2, 3 and F from ANTIC's DL), or
- as ANTIC/GTIA CCR mode (remaining text and graphics modes)

The RES bit forces a local reversal of this interpretation for the Attribute Map cell it belongs to - part of the image can be displayed in the altered resolution this way.

Conversion of CCR to artificial ANTIC/GTIA hires mode:

Each pixel in CCR resolution has width of 2 pixels of ANTIC/GTIA hires mode, and is converted into two pixels according to the rules in the table below:

color of the pixel in CCR mode	hires, left pixel	hires, right pixel
BACKGROUND	hires background	hires background
COLPF0	hires background	PF1 color
COLPF1	PF1 color	hires background
COLPF2	PF1 color	PF1 color
COLPF3	PF1 color	PF1 color

"Hires background" is the color generated by the PF2 hires mod rules described under Cell Byte 1 above: depending on the corresponding bit of the 1st byte of the Attribute Map cell, it is either the PF2 color (from the 3rd byte of the cell), or - PF2 hires mod bit set - COLPF0/COLPF3 as selected by XCOLOR[0]. "PF1 color" is a simplification along the same lines - it means the 2nd byte of the Attribute Map cell, subject to the same ANTIC/GTIA hires and XCOLOR rules as a standard PF1 pixel.

Conversion of ANTIC/GTIA hires mode into artificial CCR mode:

Pairs of adjacent bits are converted to pixels according to the rules in the table below:

hires, left pixel	hires, right pixel	Resulting color
0	0	PF0 (1st byte of Attribute Map cell)
0	1	PF1 (2nd byte of Attribute Map cell)
1	0	PF2 (3rd byte of Attribute Map cell)
1	1	COLPF3

13. Blitter

The Blitter built into the VBXE core is a hardware data-mover: it copies data from a source area (source rectangle) to a destination area (destination rectangle), both located in the VBXE VRAM, optionally performing simple logical or arithmetic operations on the data along the way (see Blitter modes of operation). This allows it to copy, fill and blend VRAM areas of any size, freeing the Atari CPU from these tasks. The copying is organized line by line: copying within a line is referred to as copying along the X axis, while copying successive lines is referred to as copying along the Y axis.

The Blitter is controlled by the BlitterList – a sequence of data loaded into the VRAM by the Atari CPU. The general structure of the BlitterList looks as follows:

```
BCB
BCB
BCB
...
BCB with NEXT-marker cleared
```

The BCB stands for "Blitter Command Block". The BlitterList consists of one or more BCBs. The BCB is a set of information for the Blitter. Each BCB defines one blitter operation. The standard BCB is 21 bytes long.

BCB structure

Byte	Name	Description
1	source_adr_0	bits [7:0], source address
2	source_adr_1	bits [15:8], source address
3	source_adr_2	bits [18:16], source address
4	source_step_y_0	bits [7:0], source step y (signed)
5	source_step_y_1	bits [14:8], source step y (signed) (bits [14:13] are significant only while the raycasting feature is active)
6	source_step_x	bits [7:0], source step (signed)
7	dest_adr_0	bits [7:0], destination address
8	dest_adr_1	bits [15:8], destination address
9	dest_adr_2	bits [18:16], destination address
10	dest_step_y_0	bits [7:0], destination step y (signed)
11	dest_step_y_1	bits [12:8], destination step y (signed)
12	dest_step_x	bits [7:0], destination step x (signed)
13	blt_width_0	bits [7:0], object width (in bytes)
14	blt_width_1	bit 8, object width (in bytes)
15	blt_height	bits [7:0], object height (in lines)
16	blt_and_mask	AND-mask for source data
17	blt_xor_mask	XOR-mask for source data
18	blt_collision_mask	AND-mask for collision detection
19	blt_zoom	X- and Y- axis zoom of the object being copied
20	pattern_feature	pattern fill
21	blt_control	additional information (see below)

source_adr

The source data for the Blitter operation may be located at any address inside the VBXE VRAM.

> **NOTE**
> While the raycasting feature is active, source_adr is the base address of the texture, with alignment requirements - see raycasting feature.

source_step_y

This parameter defines, how many bytes to add to, or subtract from the source_adr after the horizontal line of the blt_width width has been copied. Range (signed): <-4096 ... 4095>.

The step is applied to the address the line started at, not to the address reached at the end of the line.

Unlike source_step_x, source_step_y already occupies a full 15-bit signed field in the standard BCB - no raycasting container byte is needed to extend it. In the standard addressing mode only its lower 13 bits are used (matching the <-4096 ... 4095> range above); while the raycasting

feature is active, however, the Y accumulator is incremented by the full 15-bit signed, fixed-point value, so the two extra bits become significant.

source_step_x

The distance (in bytes) between consecutive source data bytes within a line. Range (signed): <-128 ... 127>.

A raycasting container extends source_step_x to a full 15-bit signed, fixed-point value - see raycasting feature.

dest_adr

The destination data for the Blitter operation may be located at any address inside the VBXE VRAM.

dest_step_y

This parameter defines, how many bytes to add to, or subtract from the dest_adr after the horizontal line of the blt_width width has been copied. Range (signed): <-4096 ... 4095>.

The step is applied to the address the line started at, not to the address reached at the end of the line.

dest_step_x

The distance (in bytes) between consecutive destination data bytes within a line. Range (signed): <-128 ... 127>.

blt_width

The width of the source object being copied (measured in BYTES), less 1. blt_width = <0 ... 511>. This corresponds to the width of 1 ... 512 bytes, and in the Overlay modes SR and LR this means 1 ... 512 pixels. In the HR mode this is 2 ... 1024 pixels.

If the object is enlarged with blt_zoom, the width of the destination object is (blt_width + 1) * (BLT_ZOOMX + 1) bytes.

blt_height

The height of the source object being copied (measured in lines), less 1. blt_height = <0 ... 255>, i.e. 1 ... 256 lines.

If the object is enlarged with blt_zoom, the height of the destination object is (blt_height + 1) * (BLT_ZOOMY + 1) lines.

blt_and_mask

Clearing bits in the source data:

```
source' = source AND blt_and_mask
```

Every byte of the source data undergoes this operation. The "source" in the equation above means the source data byte having been fetched by the Blitter. See also Blitter and constant source data.

blt_xor_mask

Reverting bits in the source data:

```
source'' = source' XOR blt_xor_mask
```

Every byte of the source data undergoes this operation.

Both the blt_and_mask and blt_xor_mask operations are performed in phase 0, right after the source data is read - see Blitter data flow.

blt_collision_mask

Collision detection mask. Any combination of bits is allowed. Collision detection is available in blitter modes 1, 2, 3, 4, 5 and 6 (see Blitter modes of operation).

See Blitter collision detection below (after the description of the blitter modes) for details on how the mask is interpreted in each mode.

> **NOTE**
> In MODE 7, blt_collision_mask is instead used as part of the lookup table address (see Blitter modes of operation); no collision detection is performed in this mode.

blt_zoom

It is possible to resize the object horizontally and vertically. This is accomplished by multiplying its width and height by a constant 1 ... 8.

bit 7	bits [6:4]	bit 3	bits [2:0]
-	BLT_ZOOMY	-	BLT_ZOOMX

The source data remains unchanged (blt_width and blt_height refer to the source object size in bytes), it is the destination area that gets enlarged. The enlargement is equal to:

```
ZOOMX(Y) = BLT_ZOOMX(Y) + 1
```

pattern_feature

bit 7	bit 6	bits [5:0]
IN_USE	-	PATTERN_WIDTH

The pattern_feature allows to "replicate" the source data within a horizontal line. If the IN_USE bit is cleared, then this function is switched off and the source data is never replicated.

If the IN_USE bit is 1, then, when PATTERN_WIDTH+1 (1 ... 64) bytes have been copied, the source address value is restored to its initial state for the line, and, next to this, the PATTERN_WIDTH+1 bytes will be copied again, and the source address will be restored again etc. until blt_width+1 bytes are copied. If blt_width+1 is not a multiple of PATTERN_WIDTH+1, the last repetition is cut short, because the line always ends after blt_width+1 bytes - blt_width has the higher priority.

> **NOTE**
> When the raycasting feature is active, PATTERN_WIDTH selects the texture size and IN_USE is not used - see the description there.

blt_control

The byte controlling the operation and general behavior of the blitter:

bit 7	bits [6:4]	bit 3	bits [2:0]
EXTENDED	AUX_CONTROL	NEXT	MODE

EXTENDED – if this bit is set (1), one or more extended BCB containers are appended to the current BCB - see Extended BCB containers.

AUX_CONTROL – the meaning of these bits depends on the Blitter's mode of operation - see Blitter modes of operation. Currently, AUX_CONTROL is used in MODE 7 (source table lookup mode), and as the sub-function selector in MODE 2.

NEXT – if this bit is cleared (0), the current BCB is the last one in the BlitterList, and after its execution the Blitter will end processing, clear the BUSY flag in the BLITTER_BUSY register, and trigger an IRQ if allowed by the IRQ_CONTROL register. If the NEXT bit is set (1), after performing the requested blit operation the Blitter will load and execute the next BCB, located in memory right after the current one (or at the address given by a jump container, if the BCB has one). If EXTENDED is set, the appended containers will be loaded even if NEXT is cleared.

MODE – see Blitter modes of operation below.

Extended BCB containers

Since version 1.27, the size of the BCB is no longer fixed - additional containers, carrying extra data and functions, can be appended to any BCB. Whether any containers are appended is decided by the EXTENDED bit in blt_control; appending a container automatically enables the function associated with it. Containers can be chained together, and their number is not formally limited. Every container starts with a container_control byte:

bit 7	bits [6:5]	bits [4:0]
NEXT	CONTAINER_TYPE	CONTAINER_CTRL

NEXT – if set (1), another container follows and should be loaded next; if cleared (0), this is the last container (last appended to current BCB).

CONTAINER_TYPE – selects the type of this container: %00 selects the jump container, %01 selects the raycasting container. %10 and %11 are not valid container types (reserved). When one is encountered, the Blitter stops loading at its container_control byte: the BCB is not executed, the Blitter ends processing (BLITTER_BUSY = 0) and no IRQ is generated, even if enabled in IRQ_CONTROL. The BCBs that precede it in the BlitterList have already been executed.

CONTAINER_CTRL – auxiliary data specific to the container type; see the corresponding container's own description.

All containers are loaded before the Blitter starts working.

BCB jump container structure

Byte	Name	Description
1	container_control	container header byte, see Extended BCB containers above; CONTAINER_TYPE = %00 selects this container. CONTAINER_CTRL bit 0 selects an absolute (0) or relative (1) jump (see jump feature below); the remaining CONTAINER_CTRL bits are unused.
2	jump_adr_0	bits [7:0] of the jump address (absolute jump) or of the jump offset (relative jump; the offset is relative to the address of byte 4 of this container)
3	jump_adr_1	bits [15:8] of the jump address (absolute jump) or of the jump offset (relative jump; the offset is relative to the address of byte 4 of this container)
4	jump_adr_2	bits [18:16], the remaining, most significant bits of the jump address (absolute jump only); this byte is always present in the container and is always fetched, but for a relative jump it is unused/reserved.

Jump feature

Appending a jump container to a BCB redirects the address the Blitter is loading its BCB/container data from - either to an absolute 19-bit VRAM address, or by adding a signed 16-bit offset to the address of the jump container's last byte (byte 4, jump_adr_2) - so an offset of +1 simply continues with the byte following the container, and an offset of 0 would load that last byte again. The redirection takes effect the moment the jump container itself has been fully loaded: whatever comes right after it - another container, if the jump container's own container_control byte had its NEXT bit set, or otherwise the next BCB, if the current BCB's blt_control NEXT bit is set - is then fetched from the new address, instead of from the address immediately following the jump container in memory.

This makes it possible, for example, to share a common sequence of containers between different BCBs, or to arrange a self-contained loop of BCBs, without the CPU having to retrigger the Blitter or rewrite any BlitterList addresses.

> **NOTE**
> A jump only redirects the Blitter's internal, current BCB/container fetch pointer - it does not change BL_ADR (BL_ADR0/1/2) itself. BL_ADR is only read once, when the Blitter is started (see Running and controlling the Blitter); the next time the Blitter is started, fetching will again begin from the (unchanged) BL_ADR address, regardless of any jumps performed during a previous run.

BCB raycasting container structure

Byte	Name	Description
1	container_control	container header byte, see Extended BCB containers above; CONTAINER_TYPE = %01 selects this container. CONTAINER_CTRL is unused.
2	rc_src_x_0	bits [7:0], initial value of the source X accumulator's fractional part (see raycasting feature below)
3	rc_src_x_1	bits [6:0], initial value of the source X accumulator's integer part, bits [14:8] (bit 7 unused)
4	rc_src_y_0	bits [7:0], initial value of the source Y accumulator's fractional part
5	rc_src_y_1	bits [6:0], initial value of the source Y accumulator's integer part, bits [14:8] (bit 7 unused)
6	source_step_x_1	bits [6:0], extends source_step_x to a 15-bit signed value, bits [14:8] (bit 7 unused)

Raycasting feature

Since version 1.27, appending a raycasting container to a BCB switches the Blitter to an alternative source-addressing mode, meant for texture-mapped, per-column rendering (such as a raycasting-style pseudo-3D view). This mode is only active for the BCB the container was appended to - every new BCB starts with it switched off, and a raycasting container has to be appended again to re-enable it, if needed.

This addressing mode only changes how the source data is addressed - it works independently of, and can be combined with, any of the Blitter's modes of operation (MODE 0 through 7).

While this mode is active, the Blitter treats the source data as a two-dimensional texture, addressed by two accumulators: X (the texture column) and Y (the texture row). Both are 15-bit fixed-point values: bits [14:8] form the integer (texel) part, and bits [7:0] the fractional part, which allows sub-texel positioning and stepping. Both are initialized from the raycasting container.

source_step_x (extended by the container to a full 15-bit signed value) and source_step_y (15-bit signed) use the same format, so they are added to the accumulators directly.

source_adr is the base address of the texture and is not modified during the operation. blt_width and blt_height describe only the destination area.

During the blit, the accumulators are updated after every destination byte - whether written, or skipped because of source transparency. With the X axis blt_zoom, the accumulators stay unchanged for the repeated bytes, and the update takes place once, together with the last repetition:

```
x_acc = (x_acc + source_step_x) & 0x7FFF;
y_acc = (y_acc + source_step_y) & 0x7FFF;
```

The accumulators are never reloaded during the blit: they are not reset at the end of a destination line, and lines repeated because of the Y axis blt_zoom do not repeat the source either. The first byte of every destination line continues with the accumulator values left by the last byte of the previous line - only the destination area has a line structure. For per-column rendering, blt_width = 0 (one byte per line) is used, so source_step_x and source_step_y become the texture step per destination line.

The size of the source texture (its width and height, in texels) is configured through the pattern_feature register: PATTERN_WIDTH bits [5:3] select how many integer bits are taken from the Y accumulator (and so, the texture's height), and PATTERN_WIDTH bits [2:0] do the same for the X accumulator (the texture's width). Both fields are encoded the same way:

PATTERN_WIDTH bits [5:3] (height) / [2:0] (width)	Texture size (texels)
0	8
1	16
2	32
3	64
4	128
5-7	reserved (currently the same as 4: 128)

Only the selected number of low-order integer bits of each accumulator is used, so the texture repeats (wraps) when an accumulator goes beyond the texture size.

The IN_USE bit of pattern_feature is not used while the raycasting mode is active.

The texture's width and height are powers of two, so the texel row and column, taken from the integer part of the Y and X accumulators respectively, are combined into a single address by concatenation, not by a multiplication.

Together, the integer parts of the two accumulators form a 14-bit value, texture_adr, as follows (MSB to LSB, left to right):

unused, remaining high-order bits	Y bits (as many as the selected texture height requires - 3 to 7, see above)	X bits (as many as the selected texture width requires - 3 to 7, see above)
0	low-order integer bits of the Y accumulator (the texture row)	low-order integer bits of the X accumulator (the texture column)

texture_adr is then combined with source_adr to form the actual VRAM address, as follows:

VRAM address bits	Formed from
[18:14]	source_adr bits [18:14]
[13:6]	source_adr bits [13:6] OR-ed with texture_adr bits [13:6]
[5:0]	texture_adr bits [5:0] (replacing source_adr bits [5:0] entirely)

> **CAUTION**
>
> The texture data must be located in VRAM at an address aligned to the total size of the texture in use (width × height texels, i.e. 64 ... 16384 bytes), and source_adr must point to its beginning. In this mode the bits [5:0] of source_adr are ignored, and its bits [13:6] covered by the texture size are OR-ed with the texture address, so they must be zero - otherwise the texture is read from wrong (folded) addresses.

Blitter modes of operation

MODE 0 – The so-called "COPY MODE". Every byte of the source data is copied to the destination, without any regard to transparency (values of 0) and without collision detection.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
dest' = source'';
WriteDest(dest');
```

MODE 1 – The main blitter mode. The source" data is copied to the destination area, IF (source" != 0). With collision detection active (see Blitter collision detection), the destination byte is fetched first and checked against the mask. Set blt_collision_mask to 0 if the detection is not needed - the blitter works faster then.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
if (source'' != 0)
{
  // cen: the dest is fetched only until the first collision
  if (blt_collision_mask != 0 && !BLT_COLLISION_CODE)
  {
    dest = ReadDest();
    if (CollisionSr(dest)) BLT_COLLISION_CODE = dest;
  }
  dest' = source'';
  WriteDest(dest');
}
```

MODE 2 – The written out data dest' is an arithmetical sum of source" and the dest. AUX_CONTROL in the BCB's blt_control field allows selecting the standard, carry or the saturation mode. The combination with both AUX_CONTROL[0] and AUX_CONTROL[1] set is reserved (not allowed).

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
if (source'' != 0 || AUX_CONTROL[0])
{
  dest = ReadDest();
  if (CollisionSr(dest) && !BLT_COLLISION_CODE) BLT_COLLISION_CODE = dest;
  sat = dest + source'' > 255 ? 255 : dest + source'';
  carry = dest + source'' > 255 ? 255 : 0;
  dest' = AUX_CONTROL[0] ? carry : AUX_CONTROL[1] ? sat : dest + source'';
  WriteDest(dest');
}
```

MODE 3 – The written out data dest' is a result of a bitwise OR of source" and dest.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
if (source'' != 0)
{
  dest = ReadDest();
  if (CollisionSr(dest) && !BLT_COLLISION_CODE) BLT_COLLISION_CODE = dest;
  dest' = dest | source'';
  WriteDest(dest');
}
```

MODE 4 – The written out data dest' is a result of a bitwise AND of source" and dest.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
if (source'' != 0)
{
  dest = ReadDest();
  if (CollisionSr(dest) && !BLT_COLLISION_CODE) BLT_COLLISION_CODE = dest;
  dest' = dest & source'';
}
else
{
  dest' = 0;
}
WriteDest(dest');
```

MODE 5 – The written out data dest' is a result of a bitwise XOR of source" and dest.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
if (source'' != 0)
{
  dest = ReadDest();
  if (CollisionSr(dest) && !BLT_COLLISION_CODE) BLT_COLLISION_CODE = dest;
  dest' = dest ^ source'';
  WriteDest(dest');
}
```

MODE 6 – HR Overlay support. It is basically the mode 1, except that transparency analysis and collision detection is done by nibbles rather than by bytes.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
if (source'' != 0)
{
  dest = ReadDest();
  no_collision_yet = !BLT_COLLISION_CODE;   // sampled once, before this byte
  if (source''[3:0] != 0)
  {
    if (CollisionHr(dest[3:0]) && no_collision_yet) BLT_COLLISION_CODE[3:0] = dest[3:0];
    dest'[3:0] = source''[3:0];
  }
  else dest'[3:0] = dest[3:0];
  if (source''[7:4] != 0)
  {
    if (CollisionHr(dest[7:4]) && no_collision_yet) BLT_COLLISION_CODE[7:4] = dest[7:4];
    dest'[7:4] = source''[7:4];
  }
  else dest'[7:4] = dest[7:4];
  WriteDest(dest');
}
```

MODE 7 – table lookup mode. The written out data dest' is read from a lookup table. The 19-bit table address is formed as in the table below. Because of the way the address is formed, the lookup table can be located anywhere in the VBXE VRAM, aligned to a 256-byte boundary, and its length is always 256 bytes.

```
source = ReadSource();
source' = source & blt_and_mask;
source'' = source' ^ blt_xor_mask;
dest' = Read({AUX_CONTROL, blt_collision_mask, source''});
WriteDest(dest');
```

table address bits	[18:16]	[15:8]	[7:0]
formed from	AUX_CONTROL	blt_collision_mask	source"

Blitter data flow

The following describes the internal data flow of the Blitter as a sequence of phases, using these abbreviations:

- **cen** - collisions enabled (blt_collision_mask != 0), but not yet detected
- **acm** - add carry mode (AUX_CONTROL[0] == 1, see blt_control)
- **const_src** - constant source data (blt_and_mask == 0)
- **zoomx** - source data X repetition due to blt_zoom (BLT_ZOOMX); internally realized as a downcounter, loaded with BLT_ZOOMX and counting down on every repeated source byte

The Blitter processes data in three phases (every source / destination rectangle byte):

- **phase 0** - source (src) read (modes 0-6) / table index (src) fetch (mode 7)
- **phase 1** - destination (dst) read (modes 0-6) / table read (mode 7)
- **phase 2** - last phase - destination (dst) write

Phase transitions (any transition not listed below - e.g. phase 0 -> 0 or phase 2 -> 2 - takes place whenever the conditions for the listed transitions are not met):

- phase 0 -> 1:
  - mode 1 with src != 0 and cen
  - mode 2 with src != 0 or acm
  - modes 3, 4, 5, 6 with src != 0
  - mode 7
- phase 0 -> 2:
  - mode 0
  - mode 1 with src != 0 and !cen
  - mode 4 with src == 0
- phase 1 -> 2: unconditional
- phase 2 -> 0: !const_src and !zoomx
- phase 2 -> 1:
  - mode 1 with cen and (const_src or zoomx)
  - modes 2, 3, 5, 6 with (const_src or zoomx)
  - mode 4 with src != 0 and (const_src or zoomx)

The destination rectangle progress is triggered on a phase 0 -> 0 transition (skip) or on phase 2 (destination written).

Address progression

Both the source and the destination address advance together, in lock-step with the destination rectangle - every time this progress is triggered (see above), both are updated according to the rules below. Their behavior differs, however, whenever blt_zoom is repeating a destination byte or line: the destination address always keeps advancing (every repeated byte/line still occupies a new destination position), while the source address is held - the same source byte/line is simply reused for every repeat.

Destination address:

- dest_adr only ever holds the address of the first byte of the current destination line; a separate, internal counter is what actually advances byte by byte and gets used for every access.
- This counter is incremented by dest_step_x after every destination byte - whether it has been written, or skipped (when the source data is transparent in the modes that do not write it).
- Once blt_width+1 bytes have been processed (the destination line is complete), the counter is instead reloaded from dest_adr advanced by dest_step_y; dest_adr itself is updated the same way, becoming the address of the next line's first byte.

Source address:

- As with the destination, source_adr only ever holds the address of the first byte of the current source line, with a separate counter advancing byte by byte - but, unlike the destination, this counter's advance is paused during a blt_zoom repeat, since the same source byte/line is meant to be reused for every repeated destination byte/line.

- The counter is incremented by source_step_x after every destination byte (written or skipped), except while that byte is being repeated due to blt_zoom BLT_ZOOMX - in which case the same source byte is simply reused, without being re-addressed. (When the source data is constant, see Blitter and constant source data below, the source byte is not fetched again for the following destination bytes, although the counter is still incremented.)
- If pattern_feature IN_USE is set, the counter is additionally restored to source_adr every time PATTERN_WIDTH+1 bytes have been copied, unless the destination line has also just been completed - in which case the end-of-line handling below takes over instead.
- Once blt_width+1 bytes have been copied (the destination line is complete), the counter is reloaded from source_adr advanced by source_step_y; source_adr itself is updated the same way, becoming the address of the next source line's first byte. Unlike the destination address, this is skipped while the destination line is being repeated due to blt_zoom BLT_ZOOMY - the same source line is then read again from its unchanged starting address.

> NOTE
> While the raycasting feature is active, none of the source-address handling above applies - the source is instead addressed through the raycasting accumulators, as described there.

Blitter and constant source data

If **const_src** is true (see Blitter data flow):

```
const_src = (blt_and_mask == 0)
```

then the source data is CONSTANT – it is independent from the source area and its value is equal to blt_xor_mask. The Blitter will skip the phase of fetching the source data, and the entire operation will be performed quicker. Filling VRAM with a constant value is twice as fast as copying.

Since the Blitter always starts working from phase 0, the source data will be fetched at least once even with const_src set - this first, fetched value is however ignored.

Zoomx (source data X repetition due to a blt_zoom BLT_ZOOMX value greater than 0, i.e. a ZOOMX magnification greater than 1) is a special case of temporarily constant source data - while a source byte is being repeated horizontally, it is likewise not re-fetched, and this too is accelerated by the Blitter in the same way as const_src.

Blitter collision detection

Collision detection is available in blitter modes 1, 2, 3, 4, 5 and 6 (see Blitter modes of operation), and is configured through the blt_collision_mask field.

> TIP
> The TRANS15 bit in the VIDEO_CONTROL register can be used to create invisible Overlay objects, serving as dedicated fields for this Overlay-Overlay collision detection.

> TIP
> If collision detection is not of interest, blt_collision_mask should be set to 0 - this disables collision detection and allows the Blitter to work faster in MODE 1.

Blitter modes 1, 2, 3, 4 and 5:

Each collision mask bit corresponds to one segment of the palette. Each segment of the palette has 32 colors (0-31, 32-63, 64-95 and so forth). Thus, by using the palette in a proper way, we can allow for collision detection in different groups of objects on the screen.

Collision is detected if the following condition is true: `(source'' != 0 && CollisionSr(dest))` - except for MODE 2 with AUX_CONTROL[0] set (add-carry mode), where the destination is read (and so checked) also for a zero source, so the condition is `((source'' != 0 || AUX_CONTROL[0]) && CollisionSr(dest))`, where:

`source''` – source data (0-255) after AND and XOR operations

```
uint8_t CollisionSr(uint8_t dest)
{
  return (((blt_collision_mask & 1) && dest >=1 && dest <= 31) ||
          ((blt_collision_mask & 2) && dest >=32 && dest <= 63) ||
          ((blt_collision_mask & 4) && dest >=64 && dest <= 95) ||
          ((blt_collision_mask & 8) && dest >=96 && dest <= 127) ||
          ((blt_collision_mask & 16) && dest >=128 && dest <= 159) ||
          ((blt_collision_mask & 32) && dest >=160 && dest <= 191) ||
          ((blt_collision_mask & 64) && dest >=192 && dest <= 223) ||
          ((blt_collision_mask & 128) && dest >=224 && dest <= 255)) ? 1 : 0;
}
```

"dest" is the retrieved target data (before blitter writes).

Example: color 0 never causes a collision (bit 0 covers colors 1-31 only). If the background image uses colors 1-31 and the static screen elements use colors 32-63 and 96-127, setting blt_collision_mask to `%00001010` (bits 1 and 3) makes the Blitter ignore the background and latch in BLT_COLLISION_CODE the color of the first static element it overwrites. A zero code means no collision, so the CPU needs no collision check of its own.

Blitter mode 6 (VBXE in HR mode):

Palette is only 16 colors long, so each mask bit controls 2 consecutive colors. Both nibbles are treated separately. Whether a collision has already been detected is checked once per byte, before the byte is processed, so if both nibbles of the same destination byte collide, the codes of both nibbles are latched in BLT_COLLISION_CODE.

Collision is detected if the following condition is met: `(source'' != 0 && CollisionHr(dest))`, where:

`source''` – source data with value of 0 to 15 (more significant nibble or less significant nibble – treated separately) after AND and XOR operations.

```
uint8_t CollisionHr(uint8_t dest)
{
  return (((blt_collision_mask & 1) && dest ==1) ||
          ((blt_collision_mask & 2) && dest >=2 && dest <= 3) ||
          ((blt_collision_mask & 4) && dest >=4 && dest <= 5) ||
          ((blt_collision_mask & 8) && dest >=6 && dest <= 7) ||
          ((blt_collision_mask & 16) && dest >=8 && dest <= 9) ||
          ((blt_collision_mask & 32) && dest >=10 && dest <= 11) ||
          ((blt_collision_mask & 64) && dest >=12 && dest <= 13) ||
          ((blt_collision_mask & 128) && dest >=14 && dest <= 15)) ? 1 : 0;
}
```

"dest" – retrieved target data (before blitter writes). More significant nibble or less significant nibble (treated separately).

When a collision is detected, the code of the non-zero pixel overwritten by the Blitter is written to the BLT_COLLISION_CODE register. A collision was detected when BLT_COLLISION_CODE != 0.

Only the first collision encountered while executing a single BCB is detected - once BLT_COLLISION_CODE becomes non-zero, further collisions are no longer checked for the remainder of that BCB. This allows the Blitter to speed up its work in MODE 1.

BLT_COLLISION_CODE

bits [7:0]
COLLISION_CODE
R-x

The register has no defined state after RESET.

The code of the collision detected, when the Blitter was running. The code corresponds to the non-zero value of the pixel overwritten by the Blitter.

BLT_COLLISION_CODE is automatically cleared to 0 whenever the Blitter starts executing a BCB - including every BCB chained via the blt_control NEXT bit, not just the first one in the BlitterList. A collision detected while executing one BCB is therefore never carried over into the next; software does not need to (and cannot) clear BLT_COLLISION_CODE itself before starting the Blitter.

Running and controlling the Blitter

To run the Blitter, the address of the BlitterList (a chain of one or more BCBs) has to be written to the BL_ADR (BL_ADR0/1/2), and then the Blitter has to be started by writing 1 to the START bit of the BLITTER_START register. The BLITTER_BUSY register can then be polled, or an IRQ can be requested via IRQ_CONTROL, to detect when the Blitter has finished processing the BlitterList.

BL_ADR (BL_ADR0/1/2)

BL_ADR0 bits [7:0]	BL_ADR1 bits [7:0]	BL_ADR2 bits [7:3]	BL_ADR2 bits [2:0]
BL_ADR[7:0]	BL_ADR[15:8]	-	BL_ADR[18:16]
W-x	W-x	-	W-x

Bits [7:0], [15:8] and [18:16] (respectively) of the BlitterList address in the VBXE VRAM. The BlitterList address in the VRAM consists of 19 bits. The BlitterList can be located anywhere in the VRAM and start at a single byte boundary. There are no limits to the length of the BlitterList. When the address of the BlitterList has been written to the BL_ADR, the Blitter may be started. After it has finished, the contents of the BL_ADR remains unchanged. The BL_ADR has to be loaded before starting the Blitter. As the BL_ADR registers are internally buffered, they may be changed immediately after the Blitter operation has been started, without affecting the BlitterList address already in use by the Blitter.

BLITTER_START

bits [7:1]	bit 0
-	1=START 0=STOP
-	W-0

Setting bit 0 to 1 causes the Blitter to start. The start takes effect only if the Blitter is idle (BLITTER_BUSY = 0); writing 1 while the Blitter is working is ignored. At the start the Blitter takes over the current value of BL_ADR, loads the first BCB (with its containers, if it has any) from that address and executes the blit it describes.

While the Blitter is working, writing 0 to bit 0 stops it immediately, which allows to abort a running Blitter operation. Normally there should be no need for this, but it is available if required.

BLITTER_BUSY

bits [7:2]	bit 1	bit 0
-	BUSY	BCB_LOAD
R-0	R-0	R-0

This register contains a non-zero value, while the Blitter is running, i.e. is processing the BlitterList (BCB_LOAD = 1) or it performs the actual operation (BUSY = 1). In the IDLE state, this register contains a value of 0, and the Blitter is ready to be set up for another task.

IRQ_CONTROL

bits [7:1]	bit 0
-	IRQE
-	W-0

IRQE = 0 – Blitter IRQ disabled.
IRQE = 1 – Blitter IRQ enabled.

IRQE, when set, enables the Blitter IRQ. Blitter IRQ is triggered when the Blitter has finished the BlitterList, i.e. after the last BCB (with NEXT = 0 in blt_control) has been executed.

Writing any value of IRQE acknowledges pending Blitter IRQ (and clears IRQF). Writing to IRQE is necessary to end the generation of the IRQ signal from the VBXE. Additionally, any core reset will also clear IRQE and IRQF and end the IRQ request.

IRQ_STATUS

bits [7:1]	bit 0
-	IRQF
R-0	R-0

IRQF = 0 – no IRQ has been requested by the VBXE.
IRQF = 1 – the Blitter IRQ has been requested. Acknowledge by a write to IRQ_CONTROL.

IRQF is set only if IRQE in IRQ_CONTROL is 1 at the moment the Blitter finishes the BlitterList. A BlitterList finished with IRQE = 0 does not set IRQF, and setting IRQE afterwards does not generate a delayed IRQ. No IRQ is generated either when the Blitter is stopped by writing 0 to BLITTER_START, or when a BCB load is aborted by an invalid container (see Extended BCB containers). To check whether the Blitter has finished regardless of IRQE, read BLITTER_BUSY.

Blitter cycle usage

The Blitter performs one VRAM access (one byte read or written) in every PCLK cycle (14.18/14.31 MHz) that is not used by another user of the VRAM. There are 912 PCLK cycles in a scanline (8 per CPU cycle). No cycles are lost between the phases of an operation, between the BCBs of a BlitterList, or at the start of an operation. The Blitter never delays any of the other VRAM users listed below.

Cycles needed by the Blitter

Loading the BlitterList (1 cycle per byte):

Element	Cycles
BCB	21
jump container	4
raycasting container	6

The number of destination bytes of a BCB is (blt_width+1) x (BLT_ZOOMX+1) x (blt_height+1) x (BLT_ZOOMY+1), see blt_zoom. Each destination byte costs the following number of cycles, depending on the Blitter mode and on source":

Mode	Fetched byte: source" != 0	Fetched byte: source" = 0	Repeated byte: source" != 0	Repeated byte: source" = 0
0	2	2	1	1
1	2 (3 if collision detection is active)	1	1 (2 if collision detection is active)	1
2	3	1 (3 if AUX_CONTROL[0] = 1)	2	1 (2 if AUX_CONTROL[0] = 1)
3, 5, 6	3	1	2	1
4	3	2	2	1
7	3	3	1	1

- Fetched byte: a byte whose source data is read from the VRAM (1 of the cycles is the source read). It is every destination byte with a variable source, except the repeats caused by BLT_ZOOMX; with a constant source, it is only the very first byte of the operation.
- Repeated byte: any other destination byte - the source data is not read again. These are the repeats caused by BLT_ZOOMX and, with a constant source, all bytes except the first one.
- Collision detection is active in mode 1 when blt_collision_mask is non-zero and no collision has been detected yet in the current operation.
- The cost does not depend on the addresses, the steps, the pattern, the raycasting or BLT_ZOOMY (other than through the number of destination bytes).

Cycles used by other VRAM users

All the accesses below have priority over the Blitter:

VRAM user	Cycles taken from the Blitter	When
MEMAC (CPU or ANTIC access to the VRAM window)	1 per byte read or written	at every such access
Overlay, SR and HR graphics mode	1 per 2 PCLK	during the Overlay part of the scanline: 256 / 320 / 336 bytes per scanline (NARROW / NORMAL / WIDE)
Overlay, LR graphics mode	1 per 4 PCLK	during the Overlay part of the scanline: 128 / 160 / 168 bytes per scanline
Overlay, text mode	3 per character (character, attribute and font byte), i.e. 3 per 8 PCLK	during the Overlay part of the scanline: 64 / 80 / 84 characters (one more if OV_HSCROLL is non-zero)
XDL	1 per XDL byte read: 2 per XDLC + its associated data bytes (22 at most)	after every scanline that finishes an XDLC (see The XDL), and right after VSYNC for the first XDLC
Attribute Map	172 (43 cells x 4 bytes)	when the map gets enabled, when a new XDLC_MAP_ADR is set, and after the last scanline of every row of cells

Number of cycles left for the Blitter in one scanline (912 minus the video data of the Overlay, before subtracting the XDL, Attribute Map and MEMAC cycles):

Overlay mode	NARROW	NORMAL	WIDE
off	912	912	912
SR, HR	656	592	576
LR	784	752	744
text	720	672	660

In the text mode with a non-zero OV_HSCROLL there are 3 cycles fewer.

HISTORY

Version 1.27.2

- Internal improvements: more reliable synchronisation with the computer after RESET and better cooperation of the MEMAC with other devices on the bus,
- The cores cannot be told apart from v1.27 by software (the same VERSION value) and their functionality is exactly the same.

Version 1.27

- Major rebuild and refactor for future enhancements and compatibility,
- GTIA implemented from scratch, closer to the original (for example PMG locking works),
- GTIA core has four palettes as FX, but palette 0 isn't modifiable there; PAL blending can be turned on or off,
- New register FEATURE_CONTROL (BLEND, CLIP, XCOLOR, GTIA_PALETTE); its features can be made persistent through RESET (PXGF), but are still volatile regarding power up / core reboot, as VBXE has no EEPROM for config,
- Strict (hard) border clipping to 336 pixels added (CLIP), in addition to the always present, less strict clipping that symmetrically cuts off wide PF garbage,
- XCOLOR extended to two bits (three variants); GTIA core has XCOLOR functionality similar to FX core,
- Attribute Map PF2 hires mod can impose an additional hires layer by swapping PF2 to COLPF0 or COLPF3, XCOLOR dependent (the feature is not new, but was limited to the PF2->COLPF3 swap),
- AVG/SUBcart compatibility - no more PMG glitches,
- NTSC vertical alignment bug fixed,
- Blitter: extended BCB containers added, with the jump feature and the raycasting feature (fractional source steps),
- Blitter: MODE 7 (table lookup) added; MODE 2 carry and saturation variants added (AUX_CONTROL),
- Internal timing improvements, bus sync after RESET improved.

Version 1.26

- bugfix in blitter: ZOOM'ing while copying in X axis didn't work properly.
- Fixed emulation of BASIC GR.11 (GTIA MODE) – now the luminance of the %0000 pixel value is correct.
- Change in XCOLOR bit (VIDEO_CONTROL register of FX core) operation – now when SET, ANTIC/GTIA hires pixels not only have background independent colors, but also have priority of PF1 color as in any other ANTIC GFX mode (GTIA priority registers can be used as normal).
- Change in handling of priority of Overlay-GTIA. Now, if any of GTIA generated colors overlaps, and for any of them PRIORITY was set to override with Overlay, Overlay pixel will be displayed, compared to Overlay logically OR'ed with GTIA as it was in 1.24 core and older. Situation like this can have place if GTIA PRIOR register will have value 0, and some PF/PMG colors will overlap.

- Change in priority registers – main one and these loaded with XDL/color map. Now 6th bit chooses priority of colors PF2 and PF3, and 7th bit changes priority for COLBAK (ie to hide Overlay behind COLBAK).
- In GTIA-only cores PAL-BLENDING was added.
- In GTIA-only cores is now possible to change palette to user defined one – no more issues with personal preferences or dislikes to default palette (LAOO.ACT). Please refer to the FX section on how to change palette (Palette 0). (Superseded in 1.27: palette 0 is no longer modifiable in GTIA cores.)

Version 1.25

Unofficial version, never released to GP.

Version 1.24

- minor internal changes
- bugfix in XDL width change from narrow to normal or wide and from normal to wide: error was introducing additional (erratic) fetch from XDLC from VRAM memory what resulted in vertical shift of following Overlay lines
- added VGA output handling

Version 1.23

- changes of internal timings and EXTSEL handling - not important from programmer's point of view

Version 1.22

- Changes improving stability of the hardware

Version 1.21

- bugfix in GTIA emulation - PRIOR register behavior ($D01B)
- since now, cores with "r" suffix will have bit 7 set in MINOR_REVISION register (now documented as CORE_REVISION)

Version 1.20

- new MEMAC-A and MEMAC-B memory windows.
- WARNING: since now CASINH is required to be connected to pin 9 (AUX4) of J3 connector on VBXE v1.x. For VBXE v2.0 CASINH signal should be connected to J3 connector pin 4. CASINH is available on pin #16 of MMU, or pin #4 of FREDDIE chip.
- Changes from 1.09 core were translated by Candle'O'Sin (Sebastian Bartkowicz)
- Proofreading and grammar check by Dan Winslow

Version 1.10

- Attribute Map: fixed RES bit functionality
- Attribute Map: added 1 bit Overlay to the Attribute Map cell
- Blitter: fixed zoom behavior, extended BCB to 21 bytes.

Version 1.09

- removal of OV_COLOR_SHIFT
- XDL can now change palettes for Overlay and PF/PMG (the same bits are used that were previously used for OV_COLOR_SHIFT)

- new blitter collision detection model - see "blt_collision_mask"
- new raster collision detection model
- Overlay/Attribute Map cell collision detection (see raster collision detection and Attribute Map – CATT bit)
- MSEL/RGB mechanism removed. - new color modification registers available (CSEL,PSEL,CR,CG,CB)
- MSEL/PRIOMAP mechanism removed - P0, P1, P2 and P3 registers have now dedicated IO addresses.
- XDLC_OVATT bit renamed to XDLC_OV_ATT
- Changes from 1.07 core were translated by Mikey (Michał Szwaczko)

Version 1.08

- security update (highly recommended) in the MEMAC A/B area handler. Now, VBXE will honour the fact that the external expansion can force this area available for memory access (via EXTSEL signal, like the CAS INHIBIT signal from the computer) VBXE will not map its own memory in this area. One needs to remember to write $FF to $D301 before any MEMAC B usage (or any other value that will shutdown a possible external or internal RAM expansion). Otherwise, the RAM expansion could get higher priority than the MEMAC accessible VRAM, via EXTSEL assertion.
- Writing of any value into any of the $D080 - $D0FF addresses (GTIA registers copy) will result in a software VBXE reset (similar to RESET keypress, but without blanking the screen and resynchronizing the core to the computer bus - see Soft and hard RESET) resulting in termination of XDL processing (Overlay and Attribute Map become unavailable) and some bits of the control registers will be reset to the default values. CAUTION: No RESET is able to restore the default VBXE palette if it has been already modified by a program. Restoring of the palette is possible only by re-loading the core. This function allows for restoring of the original screen after cold/warm start of the system, by OS ROM jump, not by RESET keypress.
- New register MINOR_REVISION (read only; now documented as CORE_REVISION) for easy identification of the loaded core.

Version 1.0 beta 7

- First English version of the programmer's manual. Translated by Drac030 (Konrad M. Kokoszkiewicz).