Altirra Hardware Reference Manual

by

Avery Lee

Version: 07/05/15

Table of Contents

Index of Tables

Index of Figures

1.1 Introduction

This document describes the hardware programming model used by Altirra, an emulator for the Atari 8-bit series of home computers, including the 400, 800, 600XL, 800XL, 1200XL, 130XE, and XEGS models. Although the emulator provides a virtual programming environment, it is intended to mimic the actual hardware. This document attempts to describe the hardware in detail as the target to which the emulator aspires to imitate. Some of this information has been collected from both official and unofficial sources, and some of it has been determined by hand through testing on a real, still functioning Atari 800XL.

While I've spent a lot of time tracking down details myself, I have to acknowledge the substantial amount of literature already available which provided background for this document. First and foremost, I'm indebted to the technical staff behind the *Atari Home Computer System Hardware Manual*, which did a very good job of describing the behavior and programming specifications for the official functionality in the Atari hardware, and which should be considered required reading prior to this document. Similar shout-outs go to the authors of Atari's *OS Manual*, which similarly documents the software side, and to Ian Chadwick and his *Mapping the Atari, Revised Edition*, which contains the most detailed and complete memory map of the Atari I know of.

If you have the time and inclination, please check out my Altirra emulator, available at the following web address:

http://www.virtualdub.org/altirra.html

-- Avery Lee

1.2 What's new in this edition

This release

- System: Added information about floating PIA port B bits.
- CPU: Added new sections on new 65C816 functionality, undocumented 6502 opcodes, and opcode tables.
- ANTIC: New sections on display timing, effects of extending the height of mode lines.
- POKEY: Added info about keyboard conflicts.
- GTIA: Added info about color generation.
- New chapter on cartridges: AtariMax, SIC!, SIDE, Corina, R-Time 8, Veronica.
- New chapter on Parallel Bus Interface devices: Black Box, Multi I/O.
- Additional device information: R-Verter, MidiMate, Ultimate1MB, VideoBoard XE.
- Additional XEP-80 commands.
- New appendices on polynomial counters and physical floppy disk formats.

04/27/2014 release

- CPU: Added section on 65C02 and 65C816 compatibility issues.
- System Control: Added information on Parallel Bus Interface IRQs.
- POKEY: Added keyboard scan code table.
- GTIA: Updated with new table of player/missile/playfield priority conflicts and information about priority conflicts in GTIA modes.
- Serial I/O: Now has its own chapter, including information about type 0-4 polling and device-provided relocatable loaders.
- 850: Corrected errors in the description of the Write command, expanded description of the Stream command, and added sections on the 850 bootstrap process.
- Disk: Added more details on 810 FDC controller status and command error conditions, and a new section about disk anomalies used by protection mechanisms.
- New section on XEP80 device.
- Reference: Updated to note guarantees on PAL register bits, and fixed errors in PACTL listing and register quick reference.

05/14/2013 release

- ANTIC updates:
  - Bus activity during WSYNC.
  - Abnormal playfield DMA.
- GTIA updates:
  - Border behavior in mode 10.

    - Player/missile shift details and lockup state.
- POKEY updates:
    - Polynomial counter patterns and timing behaviors.

09/15/2012 release

- Cycle numbers have been readjusted back so that cycle 0 is once again the missile DMA fetch.
- PIA corrections and interrupt behavior.
- CPU interrupt acknowledge timing.
- Parallel Bus Interface (PBI) information.
- XEGS game ROM selection and keyboard sense.
- ANTIC updates:
    - Virtual playfield DMA
    - Vertically scrolled jump instructions
    - VSCROL vs. DLI timing
- POKEY updates:
    - Additional serial port initializing and timing information
- GTIA updates:
    - Lo-res mode 10 anomaly
- Additional peripheral documentation:
    - CX-85 numerical keypad
    - 850 Interface Module
    - 1030 Modem
    - 810, 1050, and XF551 Disk Drives
    - Generic SIO protocol
- Fixed backwards serial port and keyboard overrun bits in SKCTL reference.
- Fixed swapped Control and Shift bits in KBCODE reference.
- Removed incorrect location of international character set from memory map; this is an OS convention anyway, not inherent in hardware.

11/23/2010 release

- 5200 SuperSystem documentation.
- BRK anomalies, decimal mode, and I flag timing.
- ANTIC horizontal scrolling bug.
- NMIST timing.
- Temperature sensitive POKEY and GTIA behaviors.

- Keyboard scan behavior.
- All scan line cycle numbers have been corrected to match the horizontal position counter (one less than previous).

1.3 Conventions in this manual

Number format

Unless specified, numbers without a prefix are given in base 10 (decimal). Numbers prefixed by $ are given in base 16 (hexadecimal).

Scan line timing

A significant number of hardware events with interesting timing occur relative to a particular offset within the timing of a *scan line*, which is one horizontal sweep of the display CRT beam. Many activities within the hardware occur at specific positions within a scan line and it is frequently useful to synchronize the CPU to scan line timing. There are 114 machine cycles for each scan line.

There is no program visible horizontal position counter in the Atari hardware. To make it easier to refer to specific offsets within a scan line, the cycles within a scan line are numbered from 0-113 in this manual, where cycle 0 corresponds to the missile DMA at the beginning of a scan line. This is also approximately the beginning of horizontal sync in the output video signal. Altirra also uses this convention in its debugger.

Deadlines

Sometimes it is necessary for the CPU to write to a hardware register before or after a particular deadline to produce a desired behavior. For purposes here, A CPU write to a register *on cycle N* satisfies a requirement to write *by cycle N*, *before cycle N+1*, and *after cycle N-1*. The cycle number is always in terms of the actual write cycle from the CPU and not the write instruction. For instance, an INC NMIRES instruction that begins execution on cycle 90 writes to NMIRES at cycles 95 and cycle 96, assuming no DMA contention.

Event timing

An event observable by a register is said to occur on a particular cycle when that is the first cycle in which a read of that register reflects the event. For instance, if an interrupt bit activates in IRQST on cycle 95 of a scan line, it means that reading the register on or prior to cycle 94 will not show the interrupt and reading it on or after cycle 95 will.

In most cases, event timing is described in this manual in terms of when it becomes visible to program execution. For instance, interrupts are described according to when the 6502 can either sense a change in interrupt status or begins executing an interrupt routine, and not when the IRQ signal on 6502 is asserted. An exception is externally visible outputs, such as video, audio, and I/O.

Active low and active high signals

In hardware designs, the signals may be designated as either *active low* or *active high* depending on the interpretation of the circuit design. The $\overline{\text{IRQ}}$ line on the CPU, for instance, is an active low signal and is activated by pulling the signal line to the low state. On the other hand, the RD5 signal from the cartridge that maps $A000-BFFF is active high, and is pulled up to +5V to signal that cartridge ROM is present.

To avoid confusion, this manual uses the terms **asserted** and **negated** to indicate the state of a signal line. An active low signal is asserted in the low state, and negated in the high state; an active high signal is asserted in the high state and negated in the low state.

1.4 Basic characteristics

Program visible behavior

A behavior or effect in the hardware which can be detected by a running program is *program visible*. Most of the hardware behavior described in this manual is program visible. For instance, the serialization behavior of the player/missile registers in GTIA is program visible because it can be detected through the collision registers. Any program-visible behavior is detectable by program code and can therefore be checked to detect incomplete emulation or broken hardware.

In contrast, a non program visible behavior cannot be detected by a running program: there is no way for an Atari program to detect the colors produced by the GTIA priority logic unless external hardware provides a loopback path.

Byte order (endian)

The 6502 is a little endian processor and therefore writes words with the lower order byte at the lower address of the byte pair. The hardware follows the same convention: in the few cases where word registers exist or words are fetched, the byte with the lower address is the lower order byte.

Bit order

Within a byte, bit 7 is the most significant bit (MSB), and bit 0 is the least significant bit (LSB). A left shift moves bits toward the MSB from the LSB, and is equivalent to multiplying by a power of two.

Whenever data in a byte represents graphics patterns, the left-most (MSB) pixel is displayed on the left side on screen. Wider two-bit and four-bit pixels are stored with the same bit ordering within a pixel, allowing arithmetic operations to function on those pixels.

Address alignment

The timing of certain CPU operations and the behavior of DMA by ANTIC can depend on the addresses of bytes within a block of memory. The start of a block of memory is said to be aligned to a particular boundary if it is a multiple of that value. For instance, the address $0800 is aligned to a 1K boundary because $0800 is divisible by a 1K block size ($0400 bytes). The address $0A00, however, is not.

A memory block crosses an alignment boundary if the addresses of the first and last bytes result in different values when divided by the alignment block size. A 40 byte block at $090A-0931 is contained within a 1K boundary, whereas $07FF-0826 crosses the 1K boundary at $0800. There are two specific behaviors associated with crossing such a boundary. One is that the 6502 sometimes requires an extra cycle when boundary is crossed; another is that the 6502 or ANTIC may fail to cross an alignment boundary and wrap addresses within the alignment block instead.

A *page* is a 256 byte block of memory aligned on a 256 byte boundary. Many operations in the 6502 require accesses to specific pages or require extra cycles when indexing causes address arithmetic to produce a final address in a different page. Two 16-bit addresses have the same page if their first two hex digits are the same, i.e. $A900 and $A947.

Read-only and write-only registers

Most registers in the hardware are either read-only or write-only: you cannot read a write-only register or write to a read-only register. The address locations are also often shared between different read-only and write-only registers, meaning that an attempt to use an unsupported memory operation will actually access the wrong register. The OS maintains a number of *shadow registers* in the kernel database in order to support reading of write only registers, with the caveat being that the shadow must be manually updated along with the hardware

register.

There are a few notable exceptions where registers are read/write, such as CONSOL in the GTIA and the direction register in the PIA.

Strobe registers

Some hardware registers, such as POTGO and WSYNC, are strobe registers. These registers trigger an action in the hardware when written by the CPU. The value written to the register is irrelevant and ignored, and the strobe is activated even if the same value is written multiple times.

There are also registers that will trigger changes on a read cycle. The PIA data registers are examples, as reading them clears pending interrupts. Similarly, some cartridge banking hardware only decodes addresses without checking the read/write line and thus respond to a read by switching cartridge banks.

Latched (sticky) bits

Latched bits are activated when an event occurs and stay in that state until reset. Most of the interrupt status bits in IRQST work that way, asserting IRQ on the CPU until the interrupt is acknowledged.

Incomplete address decoding

Address decoding is the hardware process of determining if a memory address corresponds to a particular device. A device with full address decoding responds only to the specific addresses it is designed. For efficiency reasons, many hardware devices on the Atari only partially decode addresses by checking a subset of address bits. An example is the PIA, which only contains four addressable locations but is assigned a 256 byte region at $D300-D3FF. Because bits 2-7 of the address are ignored, the PIA is mirrored 64 times within this address space.

Although all of the mirror addresses of a hardware register are equivalent, there is typically still a canonical address associated with that register, the address intended to be used. Using the canonical address of a register is less likely to run into problems in expanded configurations. For instance, while $D3C0 is a valid address to access the PORTA register on stock hardware, it may be overlaid and repurposed by expansion hardware.

Machine cycles (clocks)

Although most of the system actually runs at a faster rate, the smallest atomic unit of time for CPU execution is a single cycle at approximately 1.8MHz. All CPU instructions must begin and end on a cycle boundary; all reads and writes to registers must take place on a particular cycle. Unless otherwise specified, all cycles in this document refer to machine cycles.

Color clock

Much of the graphics system in the Atari runs at the speed of the *color clock*, which for NTSC machines runs at the color subcarrier (3.579545MHz). A *color cycle* is completed every time the color clock advances. The highest resolution possible for most graphics is determined by this clock, which produces 160 low resolution pixels across at standard playfield width. High resolution displays run at twice this frequency, for a dot clock of 7MHz, but only luminance effects are possible at this rate. Playfield and sprite positioning also occur at color clock rate.

There are two color cycles for every machine cycle. On PAL machines, where the color subcarrier is at a much higher frequency, most of the faster processes within GTIA still occur at twice the machine cycle rate.

Machine-specific behavior

There are unfortunately a few cases in which marginal timing causes systems to differ in behavior. Examples are the interrupt delay between POKEY and the 6502 and the behavior of the GTIA fifth player bit. In some cases

this can even manifest as temperature sensitivity, where a system will change behavior once a certain involved chip has warmed up and display erratic behavior during the transition. It is best that code be written to avoid dependency on such cases and to tolerate variance between systems.

Chapter 2
CPU

The 6502 chip is the CPU of the Atari. Used in many computers of the time and still in use as a microcontroller in enhanced forms, both the official and unofficial behaviors of the 6502 are well known. While the 6502 was later superseded by chips such as the 65C02 and the 65C816, the Atari 8-bit line continued using the original 6502 until the very end.

Note that there is some confusion as to the precise chip used in the Atari 8-bit series. The original 400/800 use the NMOS 6502, along with a handful of extra circuitry to provide the ability to halt the CPU for ANTIC DMA; this was later replaced with the 6502C, a custom version that contains the HALT logic built-in. This should not be confused with the CMOS 65C02, which is an enhanced 6502 with additional instructions and which was never used in the Atari 8-bit line.

The 6502 contains many nuances and unusual undocumented behaviors which are crucial to understand when programming to the metal on the Atari 8-bit series. For the sake of brevity, the basic architecture of the 6502 will be omitted here to allow more space for documenting these corner cases.

2.1 Registers

Unused flag

The 6502 does not use bit 5 of the P register. It can't be cleared and always reads as a 1.

On the 65C816, bit 5 is reused as the (M)ode bit in native mode.

Break (B) flag

Bit 4 of the processor status register is the (B)reak bit and is used to indicate whether an IRQ or a BRK instruction caused the IRQ routine to be run. It is set if the trigger was an BRK and cleared if it was a IRQ.

Contrary to both official and unofficial documentation, the B bit does not actually exist in the P register. Attempting to clear bit 4 of P and reading the result back always gives a 1 bit. The only time the B flag is visible is when the 6502 pushes the P register on the stack as part of interrupt handling. In that case, the P value pushed onto the stack will have bit 4 cleared for a BRK.

Decimal (D) flag

The D bit (bit 3) in the processor status register activates decimal mode in the 6502. When set to 1, the ADC and SBC instructions perform BCD correction. CMP, CPX, CPY, INC, and DEC are not affected.

NMOS 6502s do not clear the D flag automatically, so it must be cleared on reset. It should also be cleared in an interrupt handler if the interrupt code uses ADC or SBC and mainline code may use decimal mode.

2.2 Decimal mode

Decimal correction

Decimal arithmetic in the 6502 works by correcting each nibble after addition or subtraction. For addition, 6 is added if the nibble result exceeds 10; for subtraction, 6 is subtracted if the result is negative. The carry between the low and high nibbles is computed before this correction, so the correction can never cause a double carry. For instance, for $0F + $0F, an intermediate result of $1E is computed, and the correction then produces $14.

Flags computation

All flags are computed after carries are propagated between nibbles but before decimal correction occurs.[1]

For addition, the C flag is set whenever there is a carry out from the high nibble, allowing for extended precision decimal arithmetic. For instance, $99 + $01 = $00 with carry set. For subtraction, it is cleared for a borrow.

The Z flag is set when the intermediate result is $00, before decimal correction. Example: $FF + $01 = $66, with Z set.

The N flag is also set according to the intermediate result, to match bit 7. Example: $99 + $01 = $00, with N set.

The V flag is set when the carry between bit 6 and bit 7 is different than the result carry, or alternatively, when there is a signed overflow in binary arithmetic.

65C02 behavior

ADC and SBC take an additional cycle in decimal mode on the 65C02.

1 [IJO10]

The 65C02 computes the N, V, and Z flags differently in decimal mode. All three are computed the same way as if the same result were achieved in binary mode. That is, N is set if bit 7 of the result is set; Z is set if the result is $00; V is set if the carry from bit 6 to bit 7 is different than the carry flag.

ADC produces the same results for invalid BCD encodings on the 65C02 as it does on the 6502, but SBC can produce different results.[2]

65C816 behavior

The 65C816 computes decimal flags and results the same way as the 65C02, regardless of the state of the E flag. This means that the flags can be tested to distinguish a 6502 from a 65C816 in the same way. No extra cycle is taken as with the 65C02.

Unlike the 65C02, the 65C816 produces the same accumulator results as the 6502 for an SBC instruction with invalid opcodes.

2.3 Cycle timing

Clock speed

On an NTSC machine, the 6502 runs at exactly half the speed of the color clock, or 1.789773MHz. There are exactly 114 cycles per scan line and 29,868 cycles per frame. On a PAL machine, the 6502 runs at 2/5ths the color subcarrier frequency, or 1.773447MHz; there are still 114 cycles per scan line, but 35,568 cycles per frame.

DMA contention

On occasion the Atari's custom chips must fetch data from memory. This is known as Direct Memory Access (DMA), and when it occurs, the 6502 is blocked from the memory bus while ANTIC does a read cycle. This phenomenon slows down execution of code on the CPU and is known as DMA contention. All DMA in the Atari is related to the display and therefore the graphics setup determines the reduction in CPU performance. For NTSC, the highest rate at which the CPU can run is 92% (1.65Mcycles/sec); the standard Graphics 0 display reduces this to 64% (1.14Mcycles/sec). PAL runs noticeably faster since all display related DMA runs only 5/6ths as often.

Dead memory cycles

The 6502 uses the memory bus on every cycle without exception. Most of the time this is for useful work and therefore leads to very efficient bus utilization. There are cases, however, when these memory cycles are wasted cycles, such as:

- The second cycle of an implied mode instruction. (TXA)
- The ALU cycle of a read-modify-write instruction. (INC abs)
- The second-to-last cycle of a zero page indexed read or write. (LDA zp,X)
- The second-to-last cycle of an absolute or indirect indexed write. (STA abs, X)
- The second-to-last cycle of an absolute or indirect indexed read that crosses a page boundary (AND abs, Y).
- Conditional branches that cross a page boundary (BNE).

A memory transaction is issued during these dummy cycles and therefore these dead cycles cannot be overlapped by DMA – the CPU must still be halted. For the most part these cycles are harmless, as the Atari is a

2[6502Dec]

fairly safe platform where reads to hardware registers seldom have side effects. There are a few cases in which this does matter and indexing should be used with care:

- Accessing the PIA ($D300-D3FF), because reads from the data registers will clear pending interrupts.
- Accessing the cartridge control region ($D500-D5FF). Some cartridges use this region to switch banks and will respond to both reads and writes.
- Accessing PBI devices ($D100-D1FE and $D600-D7FF), which may also have read-sensitive regions.
- Any access with a read-modify-write instruction, since the extra cycle is a **write** cycle (except on the 65C02/65C816).

Crossing page boundaries

The 6502 attempts to optimize indexed reads by issuing a speculative read before it has adjusted for a possible carry in the high byte. If no carry is required, a cycle is saved. Otherwise, if a carry is required, it will retry the read with the correct address. For example, given the following sequence:

```
LDX     #$80
LDA     $20F0,X
```

...the 6502 will read $2070 first, and then retry with the correct address $2170. The only modes that have this behavior are: abs,X, abs,Y, and (zp),Y. The zp,X, zp,Y, and (zp,X) modes do not need to index outside of zero page and wrap from $00FF to $0000 without an extra cycle; (zp),Y does not incur an extra cycle for using $FF as the zero-page address. The (abs) mode, unique to JMP, also lacks the extra clock due to the well-known bug on the NMOS 6502 of accessing $xxFF and $xx00.

Writes, on the other hand, cannot be done speculatively as a wrong guess would trash an unrelated memory location. Therefore, stores using the abs,X, abs,Y, and (zp),Y modes always take the extra clock cycle. The first clock cycle is a speculative read and the second clock cycle is a write with the correct address. Read-modify-write instructions also always take an extra clock cycle, indexed or not, except that the dummy cycle is a **write** cycle.

Branches that cross a page boundary also have this behavior, doing a read with an incorrect address high byte first, and taking four clock cycles instead of three. No additional cycle is taken to cross a page boundary for a non-taken branch, a JMP, JSR, RTI, or RTS instruction, or any other non-branch execution.

2.4 Interrupts

Level-based vs. edge-based interrupts

IRQs on the 6502 are level triggered interrupts, which means that the interrupt request is a continuing condition that is active as long as the IRQ line is asserted. This facilitates delayed response to the IRQ as the 6502 will eventually respond to the IRQ as long as the device continues to assert the IRQ line. It also allows for multiplexing as multiple devices can assert IRQ and the 6502 will execute the IRQ handler repeatedly until all interrupts are handled. However, this also means that the interrupt condition must be cleared on the device or else the IRQ handler will continue to execute. It also means there is no memory of an interrupt event – if an interrupt request occurs while IRQs are masked in the 6502 and is revoked before they are unmasked, the IRQ handler will not execute.

NMIs, on the other hand, are edge triggered and are one-time event rather than a condition. Once the NMI signal is asserted, the 6502 will execute the NMI handler at the next opportunity. If a second NMI is requested before the first one is acknowledged, the NMI handler will only run once and the other NMI is lost.

Interrupt timing

The 6502 does not abort or resume instructions and can only respond to an interrupt on instruction boundaries. This means that longer instructions can increase interrupt response delay. The longest standard instruction possible on the 6502 is seven clocks, which can be due to a (zp),Y access crossing a page boundary, a read-modify-write instruction using abs,X mode, or a BRK/interrupt. A delay of 8 cycles is possible with undocumented read-modify-write instructions that use indirect indexed or indexed indirect mode, such as opcode $13. However, much longer delays can occur if a store to WSYNC [D40A] is performed, which can lengthen an instruction by as much as a hundred clock cycles. Use of WSYNC should be avoided if display list interrupts or other time-critical interrupts are active.

Clearing I with an interrupt pending

If an interrupt is already pending but is blocked by the I flag, clearing the I flag with a CLI or PLP instruction will result in the interrupt occurring at the end of the next instruction, and not immediately after the clearing instruction. For instance, given the following code:

```
CLI
NOP
```

The pending interrupt will not be serviced until the end of the NOP instruction. This does not happen with the RTI instruction; an IRQ can be serviced immediately after an RTI that clears the I flag.

Setting the I flag with an interrupt pending

Because of pipelining within the 6502, it is possible for the last cycle of a SEI or PLP instruction to execute immediately after the 6502 begins to acknowledge an IRQ. When this happens, the IRQ routine begins executing before the next instruction, and the curious result is that an IRQ executes with the pushed flags on the stack having the I bit set. The most common way to hit this behavior is using the following sequence to dispatch pending IRQs at a well-defined time:

```
CLI
SEI
```

This does not happen with the RTI instruction, which changes the flags earlier in the instruction.

Taken branch delay

A taken relative branch delays interrupt acknowledgment by one cycle: a case in which the earliest opportunity to respond to an interrupt is immediately after the branch instead is delayed to the next instruction. This occurs for any Bcc instruction which does not cross a page boundary. The effect does not occur if the branch instruction crosses a page (4 cycles), or for any other control flow instruction such as JMP, JSR, RTS, or RTI.

Overlapping interrupts

It is possible for the 6502 to first begin executing the seven-cycle interrupt sequence for an IRQ and then jump to the NMI vector instead if an NMI occurs quickly enough.

For IRQ+NMI conflicts, this behavior simply leads to faster acknowledgment of the NMI. However, it also has unfortunate consequences for the BRK ($00) instruction. The BRK instruction is essentially the same as an IRQ except that the flags byte pushed on the stack has the B flag set. Because of this, it is possible for an NMI to hijack the BRK sequence in the same way. When this occurs, the NMI vector is invoked with the B flag set on the flags byte on the stack. Thus, robust handling of BRK instructions requires it to be checked for in both the IRQ and NMI handlers.[3]

3 This effect is covered in detail in [VIC09], under 6510 Instruction Timing. The effect of an IRQ on a BRK is arguably not a bug, as I can find no program-visible effects: the BRK executes as expected, and the IRQ is then acknowledged afterward assuming that the IRQ line is still asserted. This does require that the IRQ handler check BRK first, though,

There are no issues with an overlapping IRQ and BRK instruction. However, when multiplexing the IRQ vector for both IRQ and BRK, the BRK instruction must be serviced before the handler exits. For multiplexed IRQs, the handler can service one IRQ at a time, relying on the hardware to keep IRQ asserted as causing the handler to re-execute until all IRQs are serviced. This is not true for BRK, which will be lost if not serviced.

On the Atari, this effect occurs if a BRK instruction begins execution at between cycles 4-8 of a scan line where either the DLI or VBI is activated.

which usually doesn't happen.

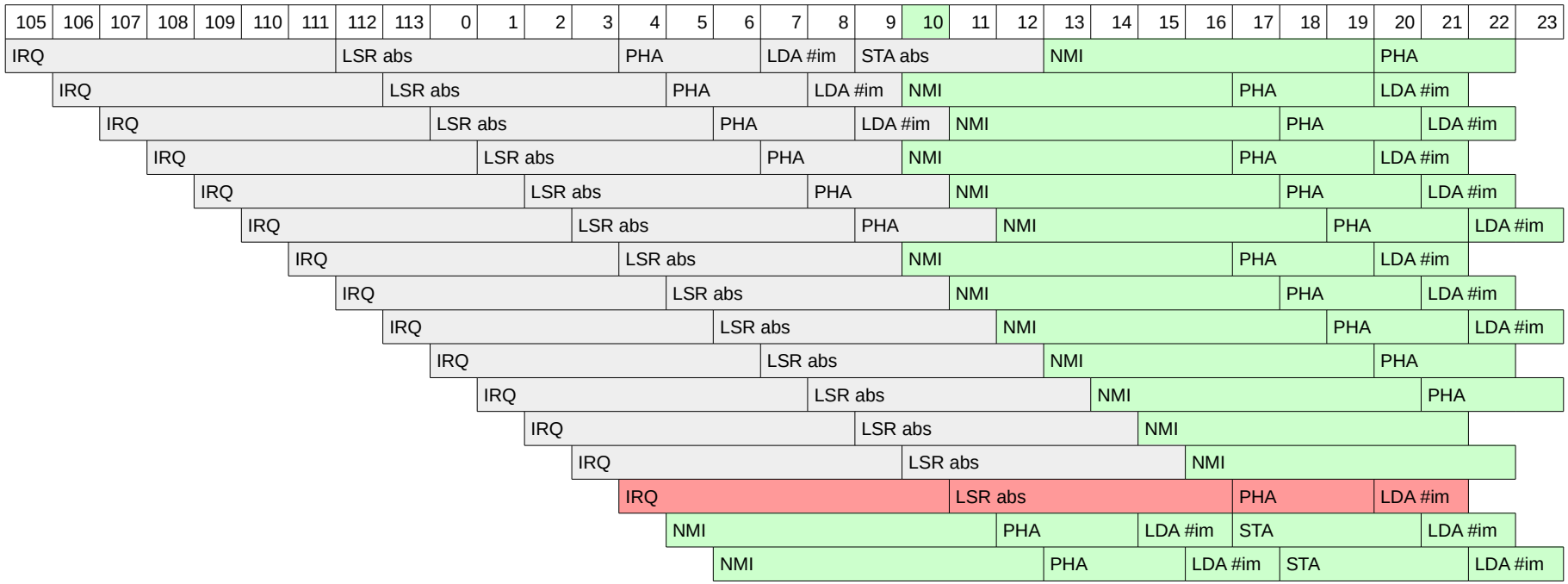


Figure 1: Effects of overlapping IRQ/NMI timing

The table above shows how the 6502 responds to IRQ and NMI being asserted at varying offsets from each other. When the IRQ occurs sufficiently before the NMI, the 6502 completes the pending interrupt sequence or current instruction before beginning the interrupt sequence for the NMI. This always entails a minimum of at least 7 cycles for the interrupt sequence and 6 cycles for the first instruction of the IRQ handler (LSR abs, 6 cycles). Unusual behavior starts when the IRQ sequence begins on cycle 4, which causes the NMI to be lost entirely. Afterward, the IRQ sequence that would begin at cycle 5 or later is taken over by the NMI, resulting in the NMI handler executing earlier than usual. The exact same timing occurs with BRK instead of IRQ.

Consecutive interrupts

The 6502 cannot acknowledge an interrupt immediately after executing an interrupt sequence. This includes BRK, IRQ, and NMI. The first instruction of the IRQ or NMI handler is always executed, regardless of any pending interrupt. The one case where interrupt sequences will execute back-to-back is if the first instruction of the interrupt handler is a BRK instruction. Because the BRK instruction is piggybacked on top of the interrupt logic, a pending interrupt can hijack the BRK instruction to run the interrupt handler instead.

2.5 Undocumented instructions

Out of the 256 possible 8-bit opcode encodings, 151 correspond to defined instructions. Due the way that the 6502 decodes instructions, some of the other 101 opcodes activate strange internal behaviors instead of being ignored or raising an interrupt.

Table 1 shows the complete opcode table for the 6502. Opcodes in gray are undocumented instructions that appear to have stable behavior; opcodes in yellow are undocumented instructions that appear to be unstable. Opcodes in red lock up the 6502 until reset.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA (zp,X)	KIL	SLO (zp,X)	NOP zp	ORA zp	ASL zp	SLO zp	PHP	ORA #imm	ASL	ANC #imm	NOP abs	ORA abs	ASL abs	SLO abs
1x	BPL rel	ORA (zp),Y	KIL	SLO (zp),Y	NOP zp,X	ORA zp,X	ASL zp,X	SLO zp,X	CLC	ORA abs,Y	NOP	SLO abs,Y	NOP abs,X	ORA abs,X	ASL abs,X	SLO abs,X
2x	JSR abs	AND (zp,X)	KIL	RLA (zp,X)	BIT zp	AND zp	ROL zp	RLA zp	PLP	AND #imm	ROL	ANC #imm	BIT abs	AND abs	ROL abs	RLA abs
3x	BMI rel	AND (zp),Y	KIL	RLA (zp),Y	NOP zp,X	AND zp,X	ROL zp,X	RLA zp,X	SEC	AND abs,Y	NOP	RLA abs,Y	NOP abs,X	AND abs,X	ROL abs,X	RLA abs,X
4x	RTI	EOR (zp,X)	KIL	SRE (zp,X)	NOP zp	EOR zp	LSR zp	SRE zp	PHA	EOR #imm	LSR	ASR #imm	JMP abs	EOR abs	LSR abs	SRE abs
5x	BVC rel	EOR (zp),Y	KIL	SRE (zp),Y	NOP zp,X	EOR zp,X	LSR zp,X	SRE zp,X	CLI	EOR abs,Y	NOP	SRE abs,Y	NOP abs,X	EOR abs,X	LSR abs,X	SRE abs,X
6x	RTS	ADC (zp,X)	KIL	RRA (zp,X)	NOP zp	ADC zp	ROR zp	RRA zp	PLA	ADC #imm	ROR	ARR #imm	JMP (abs)	ADC abs	ROR abs	RRA abs
7x	BVS rel	ADC (zp,Y)	KIL	RRA (zp),Y	NOP zp,X	ADC zp,X	ROR zp,X	RRA zp,X	SEI	ADC abs,Y	NOP	RRA abs,Y	NOP abs,X	ADC abs,X	ROR abs,X	RRA abs,X
8x	NOP #imm	STA (zp,X)	NOP #imm	SAX (zp,X)	STY zp	STA zp	STX zp	SAX zp	DEY	NOP #imm	TXA	ANE #imm	STY abs	STA abs	STX abs	SAX abs
9x	BCC rel	STA (zp),Y	KIL	SHA (zp),Y	STY zp,X	STA zp,X	STX zp,Y	SAX zp,X	TYA	STA abs,Y	TXS	SHS abs,Y	SHY abs,X	STA abs,X	SHX abs,Y	SAX abs,X
Ax	LDY #imm	LDA (zp,X)	LDX #imm	LAX (zp,X)	LDY zp	LDA zp	LDX zp	LAX zp	TAY	LDA #imm	TAX	LXA #imm	LDY abs	LDA abs	LDX abs	LAX abs
Bx	BCS rel	LDA (zp),Y	KIL	LAX (zp),Y	LDY zp,X	LDA zp,X	LDX zp,Y	LAX zp,Y	CLV	LDA abs,Y	TSX	LAS abs	LDY abs,X	LDA abs,X	LDX abs,Y	LAX abs,X
Cx	CPY #imm	CMP (zp,X)	NOP #imm	DCP (zp,X)	CPY zp	CMP zp	DEC zp	DCP zp	INY	CMP #imm	DEX	SBX #imm	CPY abs	CMP abs	DEC abs	DCP abs
Dx	BNE rel	CMP (zp),Y	KIL	DCP (zp),Y	NOP zp,X	CMP zp,X	DEC zp,X	DCP zp,X	CLD	CMP abs,Y	NOP	DCP abs,Y	NOP abs,X	CMP abs,X	DEC abs,X	DCP abs,X
Ex	CPX #imm	SBC (zp,X)	NOP #imm	ISB (zp,X)	CPX zp	SBC zp	INC zp	ISB zp	INX	SBC #imm	NOP	SBC #imm	CPX abs	SBC abs	INC abs	ISB abs
Fx	BEQ rel	SBC (zp),Y	KIL	ISB (zp),Y	NOP zp,X	SBC zp,X	INC zp,X	ISB zp,Y	SED	SBC abs,Y	NOP	ISB abs,Y	NOP abs,X	SBC abs,X	INC abs,X	ISB abs,X

Table 1: NMOS 6502 opcode table

Note on opcode names

Because the additional instructions were neither supported nor documented, there are no official names for the instructions. As such, emulators, assemblers, and disassemblers vary widely in the names used. The names

used here match a popularly used assembler, but they are by no means definitive.[4]

KIL

Opcodes: $02, 12, 22, 32, 42, 52, 62, 72, 92, B2, D2, F2.

The KIL opcodes permanently lock up the 6502 such that it stops executing instructions and no longer responds to interrupts. Only a reset will restart execution.

NOP

Opcodes: $04, 0C, 14, 1A, 1C, 34, 3C, 44, 54, 5A, 4C, 64, 74, 7A, 7C, 80, 82, 89, D4, DA, DC, F4, FA, FC.

NOP opcodes may execute addressing modes but do not change registers, flags, or control flow. Opcode $EA is the only official NOP instruction.

Note that these opcodes proceed similarly to ALU operations, so they will read operands similarly as to an LDA instruction. This includes executing an additional cycle when indexing across a page boundary.

Merged read-modify-write and read-modify instructions

Many of the illegal instructions are a result of combining read-modify-write instructions such as INC/DEC with ALU instructions like ADC and SBC. The combinations are:

- DCP = DEC + CMP
- ISB = INC + SBC
- SLO = ASL + ORA
- RLA = ROL + AND
- SRE = LSR + EOR
- RRA = ROR + ADC

The read-modify-write portion proceeds in the same manner, but the result of the RMW instruction is then used as the argument of the ALU instruction, changing the flags and potentially A. Cycle count is the same as the RMW instruction.

The ISB and RRA instructions are sensitive to the decimal mode flag due to incoporation of the SBC and ADC functions.

LAX (LDA + LDX)

Opcodes: $A3, A7, AF, B3, B7, BF

LAX instructions load the same value into both A and X, setting the N and Z flags.

SAX (STA + STX)

Opcodes: $87, 8F, 97, 9F

Stores the bitwise AND of A and X to memory. No flags are changed.

SHA

Opcodes: $93

4 For more information on undocumented opcodes and alternative mnemonics: [VIC09] [IllOpc]

Stores the bitwise AND of A, X, and the high byte read from the base address. Note that this is the high byte of the *base* address as read from page zero, not the high byte after Y has been added.

In addition, if a page crossing occurs during indexing with Y, the result of the bitwise AND also replaces the high address byte.

> **Warning**
>
> The $93 opcode has been reported to be unstable – the interaction between the high byte and bitwise AND operation does not reliably occur on all CPUs.

SHX

Opcodes: $9E

Stores the bitwise AND of X and the high byte + 1 of the base address. If a page crossing occurs during indexing with Y, the bitwise AND result also replaces the high address byte.

ANC

Opcodes: $0B

Same as AND, except with the result bit 7 also being copied into the carry flag.

ASR (AND + LSR)

Opcodes: $4B

Same as an AND instruction followed by and LSR A instruction.

ARR (ADC + AND + ROR)

Opcodes: $6B

Performs a complex operation involving a rotate right and possible decimal correction, changing the A register and the N, V, Z, and C flags.

ANE

Opcodes: $8B

Bitwise AND with accumulator, X, and immediate data, written back to accumulator.

> **Warning**
>
> The $8B opcode is not stable and may produce varying results where not all bits in the above formula participate in the bitwise AND instruction.[5]

SHS (TXS + STA abs,Y)

Opcodes: $9B

The stack pointer (S) is set to the bitwise AND of X and A, and the data written to abs,Y is this result bitwise ANDed with the high byte + 1.

5 See http://visual6502.org/wiki/index.php?title=6502_Opcode_8B_%28XAA,_ANE%29 for an extended discussion of this opcode.

LXA (LDA + TAX)

Stores the bitwise AND of A and the argument to both A and X, setting the N and Z flags.

> **Warning**
>
> The $AB opcode is not stable. It has been reported to load the immediate argument to A and X without the bitwise AND on an Atari 800.

LAS (LDA + TSX)

A, X, and S are set to the bitwise AND of the read data and S, with the N and Z flags set as usual.

SBX

AND A into the X register, then CMP with data.

2.6 65C02 compatibility

The 65C02 is an enhanced version of the 6502 implemented in CMOS and with additional instructions added. While it is mostly compatible with the 6502, there are a few differences in both documented and undocumented behavior.

Note that the 65C02 is not the same as a 6502C. Some Atari computers had a custom CPU called the 6502C (Sally) that had integrated HALT logic. This chip uses the same NMOS 6502 core and lacks the additional instructions or behavior of the newer 65C02.

Opcode table

None of the undocumented instructions of the 6502 work on the 65C02. All previously unassigned opcodes are reassigned to new opcodes or defined as NOPs with specific behavior. Table 2 shows the new opcodes in green and the defined NOPs in gray. Bit change/branch opcodes in purple are only supported by some 65C02 variants; other 65C02 makes and the 65C816 do not support bit opcodes.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA (zp,X)	NOP	NOP	TSB zp	ORA zp	ASL zp	RMB0 zp	PHP	ORA #imm	ASL	NOP	TSB abs	ORA abs	ASL abs	BBR0 zp,rel
1x	BPL rel	ORA (zp),Y	ORA (zp)	NOP	TRB zp	ORA zp,X	ASL zp,X	RMB1 zp	CLC	ORA abs,Y	INC	NOP	TRB abs	ORA abs,X	ASL abs,X	BBR1 zp,rel
2x	JSR abs	AND (zp,X)	NOP	NOP	BIT zp	AND zp	ROL zp	RMB2 zp	PLP	AND #imm	ROL	NOP	BIT abs	AND abs	ROL abs	BBR2 zp,rel
3x	BMI rel	AND (zp),Y	AND (zp)	NOP	BIT zp,X	AND zp,X	ROL zp,X	RMB3 zp	SEC	AND abs,Y	DEC	NOP	BIT abs,X	AND abs,X	ROL abs,X	BBR3 zp,rel
4x	RTI	EOR (zp,X)	NOP	NOP	NOP	EOR zp	LSR zp	RMB4 zp	PHA	EOR #imm	LSR	NOP	JMP abs	EOR abs	LSR abs	BBR4 zp,rel
5x	BVC rel	EOR (zp),Y	EOR (zp)	NOP	NOP	EOR zp,X	LSR zp,X	RMB5 zp	CLI	EOR abs,Y	PHY	NOP	NOP	EOR abs,X	LSR abs,X	BBR5 zp,rel
6x	RTS	ADC (zp,X)	NOP	NOP	STZ zp	ADC zp	ROR zp	RMB6 zp	PLA	ADC #imm	ROR	NOP	JMP (abs)	ADC abs	ROR abs	BBR6 zp,rel
7x	BVS rel	ADC (zp,Y)	ADC (zp)	NOP	STZ zp,X	ADC zp,X	ROR zp,X	RMB7 zp	SEI	ADC abs,Y	PLY	NOP	JMP (abs,X)	ADC abs,X	ROR abs,X	BBR7 zp,rel
8x	BRA rel	STA (zp,X)	NOP	NOP	STY zp	STA zp	STX zp	SMB0 zp	DEY	BIT #imm	TXA	NOP	STY abs	STA abs	STX abs	BBS0 zp,rel
9x	BCC rel	STA (zp),Y	STA (zp)	NOP	STY zp,X	STA zp,X	STX zp,Y	SMB1 zp	TYA	STA abs,Y	TXS	NOP	STZ abs	STA abs,X	STZ abs,X	BBS1 zp,rel
Ax	LDY #imm	LDA (zp,X)	LDX #imm	NOP	LDY zp	LDA zp	LDX zp	SMB2 zp	TAY	LDA #imm	TAX	NOP	LDY abs	LDA abs	LDX abs	BBS2 zp,rel
Bx	BCS rel	LDA (zp),Y	LDA (zp)	NOP	LDY zp,X	LDA zp,X	LDX zp,Y	SMB3 zp	CLV	LDA abs,Y	TSX	NOP	LDY abs,X	LDA abs,X	LDX abs,Y	BBS3 zp,rel
Cx	CPY #imm	CMP (zp,X)	NOP	NOP	CPY zp	CMP zp	DEC zp	SMB4 zp	INY	CMP #imm	DEX	WAI	CPY abs	CMP abs	DEC abs	BBS4 zp,rel
Dx	BNE rel	CMP (zp),Y	CMP (zp)	NOP	NOP	CMP zp,X	DEC zp,X	SMB5 zp	CLD	CMP abs,Y	PHX	STP	NOP	CMP abs,X	DEC abs,X	BBS5 zp,rel
Ex	CPX #imm	SBC (zp,X)	NOP	NOP	CPX zp	SBC zp	INC zp	SMB6 zp	INX	SBC #imm	NOP	NOP	CPX abs	SBC abs	INC abs	BBS6 zp,rel
Fx	BEQ rel	SBC (zp),Y	SBC (zp)	NOP	NOP	SBC zp,X	INC zp,X	SMB7 zp	SED	SBC abs,Y	PLX	NOP	NOP	SBC abs,X	INC abs,X	BBS7 zp,rel

Table 2: 65C02 opcode table

Absolute indirect addressing bug

The JMP (abs) instruction ($6C) no longer wraps within a page on the 65C02: a JMP ($02FF) instruction will access $2FF and $300 instead of $2FF and $200, and take an additional cycle when doing so.

Decimal mode

ADC and SBC instructions take one additional cycle in decimal mode on the 65C02. This is to compute proper flag results.

The 65C02 automatically clears the decimal flag on reset or on entry to an interrupt. On the 6502, it was undefined on power-up and left at the previous state on interrupt.

Read-modify-write instructions

Instructions that do read-modify-write cycles – INC, DEC, ASL, LSR, ROL, and ROR – behave differently during the modify cycle. On the original 6502, the sequence is read-write-write, where the second cycle is a write cycle that just rewrites the data that was just read. On the 65C02, the second cycle is a read cycle to that address. This alters the timing of RMW instructions to WSYNC and breaks fast IRQ acknowledgment hacks involving RMW cycles on IRQEN/IRQST.

Read-modify-write with absolute indexing

The abs,X mode versions of read-modify-write instructions only take 6 cycles on the 65C02 when indexing within

a page, instead of 7 as on the 6502.

2.7 65C816 compatibility

The 65C816 is a further enhanced version of the 65C02 with even more instructions and addressing modes as well as new native execution mode. It is actually slightly more compatible with the original 6502 than the 65C02 due to some corrections in emulation mode. Because of its greatly increased power, the 65C816 is more common of an addition to Atari computers than the 65C02.

Opcode table

The 65C816 doesn't support any of the 6502's undocumented instructions either, but it has even more of the previously unused opcodes filled with valid instructions, including ones that were NOPs on the 65C02. There are no unassigned opcodes on the 65C816. New opcodes are shown in blue in Table 3.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	ORA (dp,X)	COP #imm	ORA d,S	TSB dp	ORA dp	ASL dp	ORA [dp]	PHP	ORA #imm	ASL	PHD	TSB abs	ORA abs	ASL abs	ORA al
1x	BPL rel	ORA (dp),Y	ORA (dp)	ORA (d,S),Y	TRB dp	ORA dp,X	ASL dp,X	ORA [dp],Y	CLC	ORA abs,Y	INC	TCS	TRB abs	ORA abs,X	ASL abs,X	ORA al,X
2x	JSR abs	AND (dp,X)	JSR al	AND d,S	BIT dp	AND dp	ROL dp	AND [dp]	PLP	AND #imm	ROL	PLD	BIT abs	AND abs	ROL abs	AND al
3x	BMI rel	AND (dp),Y	AND (dp)	AND (d,S),Y	BIT dp,X	AND dp,X	ROL dp,X	AND [dp],Y	SEC	AND abs,Y	DEC	TSC	BIT abs,X	AND abs,X	ROL abs,X	AND al,X
4x	RTI	EOR (dp,X)	WDM	EOR d,S	MVP b,b	EOR dp	LSR dp	EOR [dp]	PHA	EOR #imm	LSR	PHK	JMP abs	EOR abs	LSR abs	EOR al
5x	BVC rel	EOR (dp),Y	EOR (dp)	EOR (d,S),Y	MVN b,b	EOR dp,X	LSR dp,X	EOR [dp],Y	CLI	EOR abs,Y	PHY	TCD	JMP al	EOR abs,X	LSR abs,X	EOR al,X
6x	RTS	ADC (dp,X)	PER rel16	ADC d,S	STZ dp	ADC dp	ROR dp	ADC [dp]	PLA	ADC #imm	ROR	RTL	JMP (abs)	ADC abs	ROR abs	ADC al
7x	BVS rel	ADC (dp,Y)	ADC (dp)	ADC (d,S),Y	STZ dp,X	ADC dp,X	ROR dp,X	ADC [dp],Y	SEI	ADC abs,Y	PLY	TDC	JMP (abs,X)	ADC abs,X	ROR abs,X	ADC al,X
8x	BRA rel	STA (dp,X)	BRL rel16	STA d,S	STY dp	STA dp	STX dp	STA [dp]	DEY	BIT #imm	TXA	PHB	STY abs	STA abs	STX abs	STA al
9x	BCC rel	STA (dp),Y	STA (dp)	STA (d,S),Y	STY dp,X	STA dp,X	STX dp,Y	STA [dp],Y	TYA	STA abs,Y	TXS	TXY	STZ abs	STA abs,X	STZ abs,X	STA al,X
Ax	LDY #imm	LDA (dp,X)	LDX #imm	LDA d,S	LDY dp	LDA dp	LDX dp	LDA [dp]	TAY	LDA #imm	TAX	PLB	LDY abs	LDA abs	LDX abs	LDA al
Bx	BCS rel	LDA (dp),Y	LDA (dp)	LDA (d,S),Y	LDY dp,X	LDA dp,X	LDX dp,Y	LDA [dp],Y	CLV	LDA abs,Y	TSX	TYX	LDY abs,X	LDA abs,X	LDX abs,Y	LDA al,X
Cx	CPY #imm	CMP (dp,X)	REP #imm	CMP d,S	CPY dp	CMP dp	DEC dp	CMP [dp]	INY	CMP #imm	DEX	WAI	CPY abs	CMP abs	DEC abs	CMP al
Dx	BNE rel	CMP (dp),Y	CMP (dp)	CMP (d,S),Y	PEI (dp)	CMP dp,X	DEC dp,X	CMP [dp],Y	CLD	CMP abs,Y	PHX	STP	JMP [abs]	CMP abs,X	DEC abs,X	CMP al,X
Ex	CPX #imm	SBC (dp,X)	SEP #imm	SBC d,S	CPX dp	SBC dp	INC dp	SBC [dp]	INX	SBC #imm	NOP	XBA	CPX abs	SBC abs	INC abs	SBC al
Fx	BEQ rel	SBC (dp),Y	SBC (dp)	SBC (d,S),Y	PEA abs	SBC dp,X	INC dp,X	SBC [dp],Y	SED	SBC abs,Y	PLX	XCE	JSR (abs,X)	SBC abs,X	INC abs,X	SBC al,X

Table 3: 65C816 opcode table

Decimal mode

The 65C816 computes “correct” flags for ADC and SBC in decimal mode like the 65C02, but doesn't take an additional cycle to do so, fixing the timing incompatibility.

The decimal flag is cleared on entry to the reset or interrupt handlers in the same way.

Absolute indirect addressing bug

Like the 65C02, the 65C816 indexes correctly across pages when reading the address for a JMP (abs) instruction. However, it does so without an additional cycle.

Read-modify-write instructions

Unlike the 65C02, the 65C816 preserves the 6502's read/write/write cycle pattern for RMW instructions in emulation mode. In native mode, the sequence is read/read/write as for the 65C02. The 65C816 also executes the abs,X versions in 7 cycles like the 6502.

Cross-bank indexing

Absolute indexed and indirect indexed address modes can cross banks on the 65C816 on an attempt to wrap around from $FFFF to $0000, even in emulation mode. This is a rare case where the 65C816 is less compatible in emulation mode than the 65C02 and affects the abs,X, abs,Y, and (zp),Y addressing modes. The access instead crosses over into bank $01.

The most common way to accidentally trigger this is by attempting to index using the Y register and a negative offset on a page zero symbol, i.e. LDA ICHIDZ-$F0,Y. The zp,Y addressing mode is only available on the STX and LDX instructions, so assemblers will commonly promote this to the abs,Y addressing mode. The resulting code then wraps around the 64K address space and fails on a 65C816 with 24-bit addressing.

Depending on the address wrapping pattern, affected code may still work if there is RAM in bank $01 and the data stored there is only accessed by wrapping around the 64K address space. The affected code will access bank $01 instead of bank $00 as originally intended, but still work, The code will also work if the 65C816 is only connected to a 16-bit address bus, in which case banks $00 and $01 are equivalent anyway.

Program-bank and hardwired bank 0 reads never cross bank boundaries and wrap within the same bank, in either emulation or native mode. This includes instruction fetches, relative branches, absolute indirect and absolute indexed indirect addressing modes, stack operations, and direct page addressing mode reads.

2.8 65C816 native mode

New to the 65C816 is the ability to switch into *native mode*, which unlocks the full power of the 65C816 including 16-bit memory access, arithmetic, and indexing, extended addressing, and extended interrupt handling.

M and X flags

The formerly unused bits 5 and 4 of the P register are re-purposed in native mode as the M and X flags, respectively. The M flag selects the width of memory and accumulator operations, whereas the X flag selects the width of operations involving the X and Y index registers. Indexed addressing and memory accesses from X/Y based instructions like PHX and CPY use the X flag. In both cases, a flag value of 1 selects 8-bit width, and 0 selects 16-bit width. Both M and X flags are forced to 1 upon entering emulation mode and cannot be changed until native mode is re-entered.

Whenever the X flag is set to 1 for any reason, the high bytes of the X and Y registers are cleared to $00 and their previous contents are lost. This happens both with an explicit change to the X flag and implicitly when switching to emulation mode. However, setting the M flag to 1 does *not* clear the high byte of the accumulator register, which can still be accessed by the XBA, TCS, TSC, TCD, TDC, TAX, TXA, TAY, and TYA instructions.

Some memory access and accumulator-based operations are always 16-bit regardless of state of the M flag, because they involve registers or values that are inherently 16-bit wide. These include accesses to the D register (PHD, PLD, TCD, and TDC), accesses to the S register (TCS, TSC, TXS, TSX), push effective address instructions (PEA, PEI, PER), and indirect addressing modes ((dp), (dp,X), (dp),Y, etc).

Extended direct page addressing

In emulation mode, the dp,X and dp,Y addressing modes wrap within a page by default to emulate the behavior of the 6502's zp,X and zp,Y addressing modes. This occurs whenever the low byte of the D register is $00, which is the default as D is set to $0000 on reset. If the D register is modified to a value where the low byte is not $00, then direct page indexing will cross pages, but at the cost of one additional cycle per direct page indexed instruction.

An exception to the above is that new instructions introduced with the 65C816 that read words from the direct page will cross pages even if the low byte of D is 0. This includes PEI (dp) and all instructions that use the [dp] and [dp],Y addressing modes, which will cross over from $00FF to $0100 and $0101 with D=0. This differs from instructions using ($FF,X) and ($FF),Y, which would read the base address argument from $FF and $00 instead.

In native mode, all direct page accesses cross page boundaries regardless of the value of D. Indexing will cross pages freely, and 16-bit accesses starting at $xxFF will continue to $xx00 on the next page. No additional clock cycles are taken when doing so. However, direct page accesses always wrap within bank 0, and if the low byte of D is not $00, all direct page indexed addressing will take an additional cycle.

Extended stack addressing in native mode

The stack pointer is 16 bits wide in native mode and thus the stack can be of any length and start at any address. Like direct page accesses, stack-relative accesses are always constrained to be within bank 0, even when wrapping from $FFFF to $0000.

> **Warning**
>
> In emulation mode, the high byte of the stack pointer is constrained to $01, so setting the stack pointer via TXS places the stack in the $01xx page as it does on the 6502. However, in native mode, executing TXS with 8-bit indexing (X flag set) sets the stack pointer to $00xx, which is typically undesirable. This means that setting the stack in native mode usually requires either 16-bit indexing mode or using TCS instead.

Similarly to when the X flag is set, whenever emulation mode is entered, the high byte of S is reset to $01 and the previous contents are lost.

Extended stack addressing in emulation mode

During emulation mode, stack operations performed by all 6502 and 65C02 instructions are constrained to page one. However, almost all new instructions introduced on the 65C816 that access the stack will temporarily index and write outside of page 1 into page zero when pushing or read from page two when popping.[6]

Instructions that have this behavior: PHD, PLD, PLB, PEA, PEI, PER, JSL, JSR (a,X), RTL, LDA d,S, STA d,S, LDA (d,S),Y.

Instructions that don't have this behavior: PLX, COP, PHB, PHK. The latter two instructions, although new to the 65C816, can't differ in behavior because they only push a single byte, which is always within page one regardless.

The stack pointer is readjusted to be within page 1 again after the instruction executes. For instance, executing PHD twice with S=0 will write to $0100 and $00FF, then $01FE and $01FD. Similarly, RTL with S=$FF will read from $0200-0202 and then finish with S=$02.

Interrupt vectors

In native mode, a different set of interrupt vectors is used: ($FFEE) for IRQ, ($FFEA) for NMI, ($FFE6) for BRK, ($FFE4) for COP, and ($FFE8) for ABORT. The dedicated BRK vector means that it is no longer necessary to check for it in IRQ and NMI handlers.

6 See also [ObWrap].

There is no native RESET vector, because the 65C816 switches to emulation mode on reset. Thus, ($FFFC) is always used.

2.9 Examples

Pole Position

The decrementing counters seen at the end of a race rely on the undocumented behavior of the N flag in decimal mode. If the N flag is not emulated correctly, the counters may underflow and count indefinitely.

2.10 Further reading

For a witty introduction to 6502 assembly language programming, read [LAN84].

Everyone knows about the official 6502 instruction set and about the JMP indirect bug, but sources giving exact corner-case behavior in other areas are scarcer. For cycle-level operation of the 6502, [MOS76] and [MOS76a] give details that can be difficult to find elsewhere, such as precise timing for acknowledging non-maskable interrupts. The datasheet in [EYE86] gives similar information for the 65C816 and has valuable information about differences between the NMOS 6502, 65C02, and 65C816.

For undocumented instruction details, consult [VIC09] for a thorough overview and for functionality and timing details. Note, however, that there are some errors in compared to the actual 6502 and the VICE emulator in the BCD correction algorithm.

Chapter 3
System control

3.1 System Reset button

On the original 400/800, the [SYSTEM RESET] key is connected to the $\overline{\text{RNMI}}$ line on ANTIC, which then causes an NMI to be issued to the 6502. The system NMI routine detects this condition via bit 5 of NMIST and invokes warm start behavior.

Starting with the 1200XL, this behavior was changed to use real reset logic instead. On the XL/XE models, pressing the Reset button causes the reset lines to be pulled on the 6502, ANTIC, FREDDIE, and PIA. This causes NMIs to be masked, memory banking to be reset to default, and the 6502 to restart execution at the reset vector. The $\overline{\text{RNMI}}$ line is permanently wired with a pullup to +5V and thus ANTIC will never signal a system reset NMI on these models.

3.2 Peripheral Interface Adapter (PIA)

The 6520 PIA chip controls several miscellaneous functions within the Atari.

Addressing

The PIA occupies the $D3xx block of address space and exposes four register locations from $D300-D303. Only the low two address bits are decoded, so each register is repeated 64 times.

> **Caution**
>
> Ultimate1MB overlays the $D380-D3FF half of the PIA region with its own registers.

I/O ports

The PIA contains two 8-bit data ports, port A and port B. Each contains eight bits which are individually switchable between input mode or output mode by a data direction register. Port A is controlled by control register PACTL [$D302] and data register PORTA [$D300]; port B uses control register PBCTL [$D303] and data register PORTB [$D301].

The data direction register and input/output registers share the same address. In order to read or write the data direction register, bit 2 of the control register must be set to 0, and to read or write the data port, bit 2 must be set to 1.

Port A is connected to the direction lines of joystick ports 1 and 2. Port B is connected to ports 3 and 4 on the 400/800. The XL/XE models do not have these joystick ports, so port B is used for memory banking and LED control instead.

I/O direction

Each bit in the data direction register controls whether a bit is in input or output mode. A zero bit sets the bit to input mode, while a one bit enables output for that bit. A bit in the output register is ignored when that bit is set to input, but all bits in the input register are valid even for output bits. This behavior differs between port A and port B. For port A, a bit set to output will read back as the logical AND of the output and external state. This is sometimes used to mask off incoming bits; a bit will read as zero if either the PIA or an external device is pulling the line low. For port B, any bit set to output always reads back the output state regardless of external influence.

Control lines

The interrupt and proceed lines of the SIO bus are connected to control lines CB1 and CA1 of the PIA, respectively. These are generally unused and disabled by setting bits 0 and 1 of PACTL and PBCTL to zero. They are used by a few devices, though, most notably the 1030 Direct Connect Modem.

Control lines CB2 and CA2, however, are connected to the SIO command and motor control lines, respectively. Bits 3-5 of PACTL/PBCTL are used to control the line state and should be set to 110 for a low state or 111 for a high state.[7] The command line is pulled low by the Atari while a command is being sent to an SIO device; the motor line is pulled low when a cassette tape deck should begin recording or playback.

The control lines can be used to issue an IRQ to the CPU, but this is seldom useful unless an external SIO device is specially made to take advantage of this ability.

Typically the values $34 and $3C are written to PACTL/PBCTL; this disables interrupts, raises or lowers the CA2/CB2 line, and keeps the PORTA/PORTB register in data mode so the OS VBI routine can read the joystick ports.

Interrupt status/enable bits

Bits 7 and 6 of PACTL and PBCTL indicate interrupt status of CA1/CB1 and CA2/CB2, respectively. They are read-only and their values are ignored on write. A set bit indicates a pending interrupt, and if the interrupt is enabled, an IRQ is also issued to the CPU.

Reading the input register resets both interrupt bits for the corresponding port. This must be the input register; reading the data direction register has no effect on interrupt status.

Unlike with POKEY, disabling interrupts does not clear the pending interrupt bit, and interrupts can be flagged by edge detection even if interrupts are disabled. However, switching CA2/CB2 to output mode (1xx) does clear the corresponding interrupt status (bit 6).

Reset behavior

The PIA is reset only on power on on the 800; it is also reset by the Reset button on XL/XE models. When the PIA is reset, all registers are cleared to $00. This disables all interrupts, switches PORTA/PORTB to the data direction register, and sets all peripheral port bits to input mode.

Floating inputs

On the XL/XE series, unused signal lines on PIA port B are not tied to ground or +5V and are therefore left floating. This creates a condition where the value read on those bits via the PORTB register can drift over time. Specifically, if unused port B bits are switched from outputting a 1 to input mode, they will read as 1 for a while before eventually stabilizing at 0. If the last output value was a 0, the read bit in input mode will immediately be a 0 with no delay.

While this can cause port B to return random data, it is not usually a problem in practice because only unused port B bits are affected and it only occurs for bits in input mode. On XL/XE systems, PIA port B is usually set to output mode on all bits early in initialization and kept that way during normal operation.

The unused, floating port B bits for unmodified hardware are as follows:

- 1200XL: bits 1-6
- 600XL, 800XL, 65XE: bits 2-6
- 130XE: bit 6
- XEGS: bits 2-5

The approximate time delay for the 1-to-0 transition, based on measurements on real hardware, is in the range of 100-500 ms. Delays vary between individual bits, between systems and can even vary widely on the same

7 [ATA82] III.20 indicates that bits 4-5 should be set to 1. While this is the most useful setting, bits 3-5 can also be set to other values to access six more control modes for the CA2 line. For instance, a value of 000 will reconfigure the pin for input, resulting in it being passively pulled up to the true state.

system. For instance, one system may show fairly consistent 160-190ms delays among its bits, whereas another may show 300-500ms. In any case, it is slow enough that it can even be detected from BASIC.

The 400/800 has pull-ups on all port B lines and leaves none floating. Port A is not susceptible to this issue either as it has internal pull-ups within the PIA.

For systems that have add-on extended memory, the additional bits used by the memory expansion are expected to be connected to additional hardware such that they would always be pulled up, preventing those bits from floating. This is notably not true for Ultimate1MB, though, since it implements extended memory by shadowing writes to the PIA instead of physically connecting to the PIA's port B. Therefore, on a U1MB system, it is possible to have bits that both float in input mode and control extended memory.

Spurious interrupts

Switching from output to input mode on the CA2/CB2 control lines can cause spurious interrupts to be flagged in the control register. For CA2, this happens when positive edge detection is enabled (PACTL[3:5] = 010 or 011) after the output has been pulled low recently (110). For CB2, an output low-to-high transition must be followed by any input mode (PBCTL[3:5] = 110 to 111, then 0xx). When the input mode is selected, bit 6 will become set and an IRQ will be requested from the CPU if the PIA interrupt is enabled (PACTL/PBCTL[3] = 1).

The CB2 case is particularly nasty as it corresponds to the SIO command line and the required transition is part of the normal SIO protocol. Merely writing $08 into PBCTL can cause an infinite series of interrupts if an appropriate IRQ routine is not registered to clear the unexpected PIA interrupt.

3.3 Memory system

Initial memory contents

The contents of memory upon power-up are undefined and should be treated as such. However, in some circumstances they are deterministic or almost deterministic.

The first case is when the computer is powered up after being turned off for a long time. In this case, the RAM will contain block patterns related to the internal organization of the DRAM memory chips. One possible pattern is alternating $00 and $FF bytes.

The second case is if the computer is only turned off for a short period of time before being turned back on. When the power is turned off, the DRAM contents will begin to degrade as the lack of regular refresh causes the memory cells to lose state. This can take anywhere from seconds to minutes, and if power is restored in between, the result will be a random mix of data from the last powered state and bits that have decayed to the base state.

Floating data bus

Some addresses are not decoded and responded to by any hardware device, leaving the data bus in an undriven state. These include $D100-D1FF and $D600-D7FF with no PBI devices installed and $D500-D5FF with no cartridge.

Depending on the model, this may either result in a pulled up or floating bus. On an XL and some XE machines, there are pull-up resistors on the data bus which will force the bus to $FF for an unhandled read. On the 400/800 and other XE machines, these pull-ups are missing and the result is a floating data bus. The floating data bus will tend to return the byte that was on the data bus from the previous cycle.

RAM does not drive the data bus during a refresh cycle, so the value on the floating data bus is not changed. However, the floating data bus will reflect the value read by ANTIC if the last cycle was a DMA cycle from a driven location.

When the CPU is suspended by a write to WSYNC, it repeats its current read cycle until the WSYNC condition is

cleared by ANTIC. During this time, the bus will repeatedly reflect the data at the location the CPU is trying to read. This can be in turn picked up by ANTIC if one of its DMA channels is reading from an undriven location.

3.4 Bank switching

Bank switching allows the CPU to access more memory than would ordinarily be reachable via the 64K address space dictated by its 16 address lines by multiplexing address regions based on bank switching registers. On the XL series, this allows ROM to be selectively disabled, permitting access to 62K of memory.

800XL banking

PIA port B is re-purposed in the 800XL to control the memory map instead of the third and fourth controller ports. Bit 0 = 1 enables the kernel ROM at $C000-CFFF and $D800-FFFF; bit 1 = 0 enables the BASIC ROM. BASIC takes precedence over a cartridge, if present. Bit 7 disables the self test ROM at $5000-57FF. By setting PORTB to $82, it is possible to access 62K of memory. The 2K block of hardware registers at $D000-D7FF cannot be disabled.

The self-test ROM at $5000-57FF only appears if the kernel ROM is enabled. In other words, the values $7E and $FE have the same effect, disabling both the kernel and self-test ROMs.

The remapping of PORTB means that one must be careful when clearing hardware registers – carelessly clearing memory across the $D3xx range can cause the kernel ROM to be swapped out, resulting in a system crash. This is one reason for older games failing on XL/XE computers. Whether or not this happens depends on how the port B control register [$D303] is set, as clearing bit 2 of that register causes the direction register to be accessed at $D301 instead.

Writes to ROM

The MMU logic maps addresses to circuitry solely based on address. This means that any writes to addresses that are currently assigned to kernel ROM, BASIC ROM, or cartridge ROM are ignored and do not affect the underlying RAM. It is not possible to “write through” the ROM as on some other platforms.

BASIC ROM overlap (XL/XE only)

The priority in the $A000-BFFF address space is cartridge ROM, then BASIC ROM, and then RAM. If both the cartridge and BASIC ROM are enabled in that area, the cartridge is visible.

Game ROM (XEGS only)

On the XEGS, setting bit 6 of PIA port B to 0 enables the Missile Command game ROM at $A000-BFFF. This has lower priority than the BASIC ROM and will therefore be overridden by BASIC if port B bit 1 is also set to 0.

3.5 Extended memory

130XE banking

The 130XE additionally uses bits 2-5 to control access to an additional 64K of memory through a window at 4000-7FFF. Bits 4 and 5 enable CPU and ANTIC access to extra memory when *cleared*. Bits 2 and 3 control the memory bank, selecting one of four extra 16K banks. The CPU and ANTIC must access the same bank of memory if both are using extended memory. There is no way to redirect the 4000-7FFF window to any memory in the primary 64K.

The self-test ROM has priority over the extended memory window if both are enabled.

320K modification (RAMBO)

Bits 4-7 of PORTB are used to access 16 extended banks of 16K at 4000-7FFF. This results in 64K + 256K = 320K of memory. Because bit 5 is used, ANTIC external memory access is not available if this mod is used on 130XE hardware. Instead, both CPU and ANTIC are switched at the same time.

576K modification

Reusing the BASIC bit (bit 1) raises the number of selectable banks to 32, for a total of 64K + 512K = 576K. Bits 0, 4, and 7 still control the kernel ROM, CPU access, and the self-test ROM on XL/XE hardware.

1088K modification

Overloading the self-test bit (bit 7) gives six bits for bank selection, bits 1-3 and 5-7. 64 banks of 16K plus the main 64K bank gives 64K + 1024K = 1088K. Note that the self-test ROM is still accessible if the extended RAM access bit (bit 4) is disabled.

3.6 Miscellaneous connections

Cartridge sense (XL/XE only)

On the XL/XE series, the RD5 cartridge line is connected to the trigger 3 input (T3) of GTIA. The RD5 line signals when the cartridge is supplying data in the $A000-BFFF range and therefore built-in memory should be suppressed. Because RD5 is active high, the TRIG3 register in GTIA reads as a 1 (button not pressed) when cartridge ROM is present and 0 (button pressed) when it is absent. This is used as a cartridge sense mechanism by the XL/XE OS.

When a cartridge is disabled via bank switching and no longer presenting anything at $A000-BFFF, TRIG3 reads as a 0.

The internal BASIC ROM does not affect TRIG3.

On a SECAM system with an FGTIA, the triggers are gated and only updated once each horizontal blank. This causes delays in TRIG3 updating to match cartridge state changes and is a source of cartridge compatibility problems. The TRIG3 cartridge sense can also be affected by the GTIA trigger latch function.

Keyboard sense (XEGS only)

On the XEGS, the trigger 2 input (T2) of GTIA is used to sense whether a keyboard is connected. If a keyboard is connected, TRIG2 reads $01 (trigger not pressed), while it reads $00 otherwise. This is consistent with the XL/XE series which has T2 disconnected and also reads $01.

1200XL option jumpers

The 1200XL has four option jumpers which are connected to unused pot lines. Option jumper J1 is connected to POT4 and causes a self-test on startup if installed.[8]

3.7 Examples

Caverns of Mars

This game configures the upper four bits of port A as output in order to force them to zero, and fails to read the joystick if this is not reflected in the values read.

MidiTrack III

Monitors the CA1 (SIO proceed) input of the PIA for synchronization pulses without having IRQA1 enabled.

R-Verter handler software

Monitors CA1 (SIO proceed) and CB1 (SIO interrupt) inputs to the PIA without either IRQA1 or IRQB1 enabled.

3.8 Further reading

The definitive resource for anything involving the Atari memory map is [CHA85]. Appendix 16 provides information on the new PORTB assignments for the 130XE.

[ATAXL] describes numerous modifications to the hardware and kernel in the 1200XL, such as the option jumpers.

8 [ATAXL] p.15

[ATA82] contains both functional and detailed schematics of the Atari 400/800 and is useful in tracing signal flow between the custom chips.

For detailed programming information for the 6520 PIA chip, particularly modes not covered by the Hardware Manual, consult [MOS76].

Chapter 4
ANTIC

ANTIC is the master chip of the Atari 8-bit chipset, controlling frame timing and doing all direct memory access (DMA).

4.1 Basic operation

Addressing

ANTIC occupies the $D4xx block of address space. Only the low four bits are decoded, so any address of the form $D4XY will address mirror X of register Y. The canonical registers are at $D400-D40F.

Unassigned addresses within the ANTIC address range read as $FF. This is true even on hardware models that have a floating data bus for unassigned addresses, as ANTIC actually drives $FF onto the bus for addresses in its range that don't have registers assigned.

Reset behavior

On power-on or reset, ANTIC automatically clears the following items:

- Horizontal and vertical counters
- Refresh row address counter
- NMIEN
- DMACTL
- Playfield DMA clock

The following items are **not** reset:

- WSYNC
- HSCROL/VSCROL
- PMBASE
- CHBASE
- PENH/PENV
- CHACTL
- DLISTL/H
- NMIST
- Memory scan counter

Typically a warm reset routine will clear all registers in order to reset ANTIC to a known state.

Note that on 400/800 hardware, ANTIC is only reset on power-on. On XL/XE hardware, the Reset button also resets ANTIC.

4.2 Display timing

As the main display processor in the system, ANTIC is responsible for overall display timing. The ideal display timings produced by ANTIC are as follows (ignoring component variation):

Parameter	NTSC	PAL
Master clock	14.31818MHz	14.18757MHz
Machine clock	1.790772MHz (14.31818MHz ÷ 8)	1.773447MHz (14.18757MHz ÷ 8)
Horizontal scan rate (scan line rate)	15.69975KHz (1.789772MHz ÷ 114)	15.55655KHz (1.773447MHz ÷ 114)
Vertical scan rate (frame rate)	59.92271Hz (15.69975KHz ÷ 262)	49.86074Hz (15.55655KHz ÷ 312)

Table 4: ANTIC display timing

Importantly, the horizontal and vertical scan rates deviate from ideal NTSC and PAL broadcast timing. For NTSC, the machine clock runs at exactly half the color subcarrier rate (3.58MHz), but the scan line is 114 machine cycles instead of 113.75 cycles and the frame has 262 scan lines instead of 262.5. This prevents the color subcarrier from inverting phase on each scan line and produces a non-interlaced display with 15.700KHz / 59.92Hz timing instead of an interlaced one with 15.735KHz / 59.94Hz timing. Similarly, the PAL ANTIC produces 312 scan lines instead of 312.5 and also produces a non-interlaced display.

Mixed PAL/NTSC systems

While standard systems have matched ANTIC and GTIA chips, it is possible to combine an NTSC ANTIC with a PAL GTIA or vice versa. This results in either a 50Hz NTSC display or a 60Hz PAL display. The NTSC-50 case is the more interesting of the two as the 50Hz frame rate avoids many compatibility issues with software written for PAL. In such a mixed system, the ANTIC type determines the frame timing and the GTIA type determines the value read from the PAL register.

Although ANTIC does not directly indicate its type via a readable register like GTIA does, an NTSC ANTIC can readily be distinguished from a PAL ANTIC by polling the VCOUNT register.

Pixel aspect ratios

The display timings used by ANTIC also determine the aspect ratio of pixels on screen. These pixels are not square, and furthermore, differ between NTSC and PAL.

For NTSC, a dot clock of 12.2727Hz corresponds to square pixels.[9] However, this is for interlaced video (~480 visible scan lines), so the equivalent rate for non-interlaced video is half the rate, 6.1364MHz. The dot clock produced by NTSC ANTIC+GTIA at hires mode is faster at 7.159MHz, giving a noticeably narrow pixel at 0.857:1. Player/missile graphics and higher-resolution but non-hires playfields typically use 160 clock resolution, however, so their pixels will be doubly wide at 1.714:1.

For PAL, a dot clock of 14.75MHz is used for square pixels in interlaced video, giving 7.375MHz for non-interlaced video. The PAL ANTIC+GTIA in hires mode outputs pixels at 7.094MHz, giving a slightly wide hires pixel at 1.04:1. Although not square, this is close enough for many purposes.

Many other computers of the era used a similar technique of generating pixels with a dot clock derived from the color subcarrier frequency and have comparable pixel aspect ratios, particularly the Apple II and the Amiga.

4.3 Playfield

The main display produced by ANTIC is known as the playfield.

9 [TIVideoDec] p.2-7.

Playfield width

Three playfield widths are supported: narrow, normal, and wide. The normal playfield width is 160 color clocks wide (320 hires pixels), and is used by all OS graphics modes. Narrow playfields are 128 color clocks wide; these are useful when the extra width is not needed, as narrow playfields have less data to set and also allow the CPU to run slightly faster. Wide playfields are 192 color clocks wide and even cover the overscan regions on the sides.

All three playfield widths share the same center, so a normal playfield adds 16 color clocks on each side of the narrow playfield, and a wide playfield adds another 16 color clocks on each side. However, the wide playfield is so wide that it is truncated: 12 color clocks are hidden on the left side and two are cut off by horizontal blank on the right. As a result, only 178 color clocks out of 192 are visible.[10]

DMACTL bits 0-1 control the width of the playfield, and can also disable the playfield entirely, causing the background color to be displayed.

Playfield colors

The playfield is composed of up to four colors, PF0-PF3, overlaid on top of the background (BAK). ANTIC tells the GTIA when each playfield color is used, and five independent color registers in GTIA are used to produce the final playfield. Depending on the display mode, there are four different color configurations:

- **Two colors**. These bitmap modes display either BAK or PF0.
- **Four colors**. These bitmap modes display BAK or PF0-PF2.
- **Five colors**. These character modes display BAK or PF0-PF3.
- **One color in two luminances**. These are special high-resolution modes where pixels are so narrow that they are only a half color clock wide. In these modes, the entire playfield is a single hue as specified by PF2, but the graphics data is used to conditionally substitute in the luminance from PF1.

The fourth playfield color, PF3, is seldom used by the playfield. Therefore, the GTIA contains a bit to reuse this color as a fifth color for player/missile graphics instead.

Playfield modes

ANTIC supports fourteen playfield display modes, selected by the display list. Each playfield covers the entire width of the screen for some vertical distance, controlled by the display list; it is possible to vertically stack different playfield modes on the same screen. Six of the display modes are character modes, while the other eight are mapped (bitmap) modes.

Playfield data ordering

All playfield data, including bitmap data and character font data, is stored such that bit 7 represents the left-mode pixel on screen and bit 0 is the right-most pixel. In multicolor modes where a pair or group of four bits is used to represent a pixel, the bits are ordered as for CPU integers. For instance, the color PF1 in the second pixel of a four-color bitmap or character map mode would be represented by the pattern xx10xxxx.

4.4 Character modes

The playfield can be configured to display text through character modes, which use a layer of indirection to produce output. In these modes, two separate memory regions are used:

10 The displayable width for a wide playfield is given as 176 color clocks in some references. The discrepancy is because in a wide unscrolled IR mode 2-5/D-F playfield, the last two color clocks are garbage due to suppressed DMA cycles. They are part of the playfield, however, as they can cause player-playfield collisions.

- **Character names**. These are fetched first, and indicate which characters to display within the mode line.
- **Character set data**. The character names are then used to index into the current row of the character set to fetch the actual data to display.

Character modes allow text displays to be produced with minimal data manipulation, since the CPU need only modify one byte per character rather than copy the data for each character.

Some character modes display characters as monochrome, whereas others display characters as multicolor. The multicolor modes are often used to quickly display graphical tiles rather than text.

Mode list

These are the character modes supported by ANTIC:

Mode	Scan lines	Colors	Bytes (normal width)	Resolution	Color mode	Pixel size
2	8	1.5	40	40	Hi-res	8x8
3	10	1.5	40	40	Hi-res	8x8
4	8	5	40	40	Lo-res	8x8
5	16	5	40	40	Lo-res	8x16
6	8	5	20	20	Lo-res	16x8
7	16	5	20	20	Lo-res	16x16

Modes 2 and 3: High-resolution monochrome text

Mode 2 is the standard 40-column screen seen on startup. Each playfield byte selects an 8x8 character from an array of 128 pointed to by CHBASE; bit 7 controls inversion or blinking, based on modes in CHACTL.

The character set requires 1K of memory and must be aligned to a 1K boundary. Each of the 128 characters is described by 8 contiguous bytes, where the first byte corresponds to the data for the first scan line. With each byte, each bit corresponds to a pixel on screen, where bit 7 is the left-most pixel. Because mode 2 is a hi-res mode, the entire playfield uses the PF2 color, and each bit indicates whether luminance comes from PF2 (0 bit), or PF1 (1 bit).

Although it is not exposed as a standard OS mode, it is possible to enable the GTIA modes with a mode 2 or 3 playfield, thus giving a 9 or 16 color tiled playfield.

Mode 3 is similar to mode 2, except that each mode line is 10 scan lines tall instead of 8. The extra two scan lines reuse the same data from the first two, but only one of the pairs displays valid data. Characters 00-5F display data for scan lines 0-7 and display $00 data for rows 8-9, while characters 60-7F display on rows 2-9 instead and display $00 data for scan lines 0-1. This permits one-quarter of the character set to have descenders. For descenders to display properly, the character data must be stored out of order since rows 2-7 are displayed above rows 0-1.

Modes 4 and 5: Multicolor text

Mode 4 is another character mode that produces 40 characters across in normal width, but unlike modes 2 and 3, mode 4 is a lo-res mode that produces up to five colors. Instead of each character producing monochrome characters in an 8x8 block, each character is instead 4x8 with pixels twice as wide. Normally each pair of bits produces either the background color (00) or PF0-PF2 (01-11). If bit 7 is set, however, the 11 pair produces PF3

instead of PF2.

Mode 5 is the same as mode 4, except that scan lines are repeated once and each character is 16 scan lines tall instead of 8.

Modes 6 and 7: Single color text in five colors

Mode 6 is the familiar single-color, double-wide signature character mode of the Atari. At normal width, it produces 20 8x8 characters per row, where each pixel is one color clock wide. The character set is half the size in mode 6, requiring only 512 bytes and 512 byte alignment. Only 64 characters are available in the mode because the upper two bits are used to select the foreground color used by 1 bits, with 00-11 producing PF0-PF3. 0 bits in the character data always produce the background color.

Mode 7 is the same as mode 6, except that scan lines are doubled and each character is 16 scan lines tall.

Character set storage

All character modes require image data for each character. For modes 2, 3, 6, and 7, the character set is stored as 128 characters within a 1K block, aligned to a 1K boundary; for modes 4 and 5, it contains 64 characters within a 512 byte block, aligned on a 512 byte boundary. The low three bits of the address specify the row so that each contiguous block of 8 bytes represents a character.

The top 6 or 7 bits of the CHBASE register specify the base address of the character set. It can be dynamically changed on the fly, but the change will not take effect until two cycles past when the register is changed. While bit 1 is not used in modes that use 1K of character data, it is still stored on write and that latent bit will become active should a 0.5K character data mode activate.

Blinking and inversion

In the high-resolution modes (modes 2 and 3), bit 7 of the character name is used as an extra attribute bit to indicate reverse video or blinking. For this to happen, bits 0 and 1 of CHACTL must be used. When bit 1 is set, character cells with name bit 7 set are displayed inverted. When bit 0 is set, those cells are blanked as if the character font data were all zero bits. This means that in order for text to blink, software must periodically toggle the state of bit 0. Setting both bits 0 and 1 results in inverted space characters.

If display DMA is temporarily disabled when character name fetch would occur, ANTIC reuses the character names stored in the line buffer, but the invert/blink state that normally comes from bit 7 is reused from the last character rather than the bit 7 value from the line buffer.

Bits 0 and 1 of CHACTL have no effect in modes 4-7.

Vertical reflection

Setting bit 2 of CHACTL flips all characters upside-down, displaying row 7 of the character set first. Unlike the blink and inversion features, this affects all character modes.

Vertical reflection works exactly as if the row bytes in the character set were reversed in order. This means that it produces nonsensical results for characters with descenders in mode 3 (60-7F), as the reflection causes rows 6-7 to appear in the descender area.

4.5 Mapped (bitmap) modes

The playfield can also display data from memory directly in bitmap modes, which simply map single bits or pairs of bits to color. This allows every pixel to be completely independent at the cost of often requiring much more memory, as much as 8K per frame buffer. ANTIC always displays bitmap data with the first byte of each row and the most significant bit of each byte corresponding to the leftmost pixel.

The supported modes are as follows:

Mode	Scan lines	Colors	Bytes (normal width)	Resolution	Color mode	Pixel size
8	8	4	10	40	Lo-res	8x8
9	4	2	10	80	Lo-res	4x4
A	4	4	20	80	Lo-res	4x4
B	2	2	20	160	Lo-res	2x2
C	1	2	20	160	Lo-res	2x1
D	2	4	40	160	Lo-res	2x2
E	1	4	40	160	Lo-res	2x1
F	1	1.5	40	320	Hi-res	1x1

Mode 8: Four color bitmap at lowest resolution (4x8 pixels)

Mode 8 is the lowest resolution graphics mode, producing 40 pixels across with one of four colors. Bits 7 and 6 of a byte correspond to the left-most pixel; 00 selects the background color while 01-11 produces PF0-PF2. Each pixel is 4 color clocks wide and 8 scan lines tall.

Modes 9 and A: Bitmap modes with 2x4 pixels

Mode 9 is double the horizontal and vertical resolution of mode 8, with each pixel being 2 color clocks wide and 4 scan lines tall. However, it is only a two-color mode, with each bit selecting the background (0) or PF0 (1). Bit 7 is the left-most pixel in each byte.

Mode A is the four-color version of mode 9. Each pixel selects the background (00) or PF0-PF2 (01-11).

Modes B and D: Bitmap modes with 1x2 pixels

Mode B increases resolution further to 1 color clock and 2 scan lines per pixel, with two colors per pixel (background and PF0).

Mode D is the same as mode B, except that each pixel is two bits and selects from one of four colors.

Modes C and E: Bitmap modes with 1x1 pixels

Mode C is the same as mode B, except that mode lines are only one scan line high. It is the highest resolution two color bitmap mode available.

Mode E is the same as mode C, except that each pixel is two bits and selects from one of four colors. It is the highest resolution four color bitmap mode available.

Mode F: High resolution bitmap mode

Mode F produces 320 pixels across at normal width, with each bit corresponding to a pixel one-half color clock wide and one scan line tall. It is a high-resolution mode, meaning that the whole playfield uses the PF2 color and the luminance from either PF2 (0) or PF1 (1).

This mode is also the mode that serves as the basis for the three new modes added with the GTIA; the only difference in setup is that bits 6 and 7 of PRIOR on the GTIA are set to a value other than 00.

4.6 Display list

The display list determines how and when ANTIC fetches playfield data for display through GTIA. It is composed of a series of one-byte or three-byte instructions, each of which controls the display of at least one scan line on screen, and is normally repeated for every frame.

Instruction pointer

DLISTL/DLISTH set the instruction pointer used to fetch the display list. It can be placed anywhere in the 64K address space, but cannot cross a 1K boundary without an explicit jump instruction as only the lower 10 bits increment.[11] This includes single instructions – a 3-byte LMS or jump instruction that crosses a 1K boundary will wrap addresses in the middle of the instruction.

Any write to DLISTL/DLISTH will immediately change the memory pointer used for the next display list fetch. Because of the possibility of display list interrupts, it is dangerous to do this in the middle of a display list, as changing only one of the address bytes may cause ANTIC to execute random memory as a display list and therefore issue spurious DLIs. A $C1 instruction is particularly dangerous as it will cause a DLI to activate every scan line until vertical blank and can easily cause a crash. Therefore, the display list pointer should normally only be updated when either display list DMA is disabled or during vertical blank.

Instruction format

A display list instruction is described in a single byte as follows:

DLI	LMS	VS	HS	Mode

D7 Display list interrupt

0	No interrupt
1	Interrupt CPU at beginning of last scan line

D6 Load memory scan counter (LMS operation)

0	Normal
1	Load memory scan counter with new 16-bit address

D5 Vertical scroll

0	Disable vertical scrolling
1	Enable vertical scrolling

D4 Horizontal scroll

0	Disable horizontal scrolling
1	Enable horizontal scrolling

D0:D3 Mode

0000	Blank
0001	Jump
other	Non-blank mode line

Instruction bytes are read into the Instruction Register (IR) within ANTIC.

Playfield mode lines

Modes 2-F select a playfield mode line for display.

11 Hardware II.10

Load Memory Scan (LMS) commands

Setting bit 6 on a non-blank mode line causes the playfield memory scan pointer to be reloaded with a new address from the two following bytes, LSB first. This can be done on any such mode line and as frequently or infrequently as required; no blank line is incurred and the display appears uninterrupted. Normally one LMS is required at the beginning of the display list to reset the playfield address to the beginning of the screen memory.

Screen modes that require more than 4K of memory require at least one other LMS command in the middle of the screen to hop the 4K boundary. LMS commands may also be used in order to store rows of the display in discontiguous memory or with address spacing other than the default for the current playfield width, which is useful for large scrolling playfields.

> **Warning**
>
> An LMS alone is not enough to correctly display a playfield that requires more than 4K of data. If a scan line crosses a 4K boundary, it will wrap around to the beginning of the 4K block in the middle of the scan line. This cannot be fixed with LMS as that can only affect the beginning of the scan line. The OS avoids this problem while still maintaining contiguous addressing by adjusting the offset of the playfield buffer so that the 4K boundary occurs exactly between scan lines.

Blank mode lines (IR mode 0)

A blank mode line is specified by an instruction byte whose lowest four bits are 0000. In this case, bits 4-6 specify a scan line count instead, where 000-111 specify 1-8 scan lines. A blank mode line is never considered to have scrolling enabled or to initiate an LMS operation.

Jump command (IR mode 1)

Instruction bytes with a mode of 0001 are jump commands and are always followed by two bytes indicating the new instruction pointer for the display list. This produces a three-byte instruction similar to a 6502 JMP instruction, where the new 16-bit address is specified as low-byte first. Because the jump instruction occupies a display list slot, a blank line is displayed during its execution.

Like blank line instructions, jump instructions are never interpreted as having scrolling enabled, regardless of the values of bits 4 and 5, which are ignored for jump instructions. However, if the jump instruction follows a vertically scrolled mode line, it can be extended due to ending a vertical scrolling region the same way that blank lines can. When this occurs, ANTIC repeatedly fetches a new display list address at the beginning of each subsequent scan line. This has the effect of following a chain of indirect 16-bit addresses and is typically undesirable.

It is, however, possible to activate a DLI on a jump command.

Jump and wait for Vertical Blank (IR mode 1 + bit 6)

A jump instruction with bit 6 set ($41) also suspends the display list until vertical blank. This is usually used to terminate the display list and restart it for the next frame. When using a display list that loops using such an instruction, it is not necessary to write DLISTL/DLISTH per frame as ANTIC will autonomously repeat the display list every frame.

The internal execution of a JVB instruction is the same as if display DMA were disabled immediately after a jump instruction. No instruction or address bytes are fetched again, and the jump instruction is replayed over and over. If the previous instruction had vertical scrolling enabled, then the JVB instruction will initially have its height modified appropriately, and then replay subsequently with one scan line high as usual. Similarly, if the DLI bit is set on the JVB instruction ($C1), ANTIC will fire a DLI each and every time it is replayed, up to once per scan line.

Like any other instruction, JVB requires a scan line to execute. This means that attempting to create a display list

with 240 visible scan lines and ending with a JVB will fail, since the JVB makes the display list 241 scan lines tall. Unless DLISTL/DLISTH is rewritten in the VBI to manually restart the display list each frame, this will result in a flickering display where even frames display the intended 240 line display and odd frames are blank frames consisting solely of the JVB instruction.

The display list pointer is reset when the address bytes are fetched on the first scan line of the JVB instruction. Writes to DLISTL/DLISTH afterward will replace the address that was loaded with JVB, even if they occur before vertical blank.

Once display list DMA has been suspended with a JVB instruction, there is no way to restart it other than to wait for vertical blank.

Valid display list range

The display list starts at scan line 8 and ends no later than scan line 248. The maximum height of a display list is thus always 240 scan lines. This is true even in PAL, which has 50 more scan lines than NTSC.

If a display list is too long, ANTIC automatically suspends the display list at the beginning of vertical blank at scan line 248 and resumes it at the end of vertical blank on scan line 8 of the next frame. This means that if a display list were exactly 480 scan lines tall and looped with a jump ($01) instruction, it would alternate perfectly between two images. Typically this doesn't happen, though, because the vertical blank routine reloads DLISTL/DLISTH. Otherwise, however, ANTIC will happily keep fetching instructions, wrapping around within 1K of memory over and over.

The vertical scroll bit (bit 5) is still tracked across vertical blank. This means that if the vertical scroll bit is always on for all displayed mode lines, no vertical scrolling actually occurs, because none of the mode lines is either the start or end of a vertical scrolling region.

Any mode line which extends partially over the vertical blank is truncated.

Suspended display list DMA

If display list DMA is turned off, ANTIC reuses the last fetched instruction byte. This essentially repeats the last mode line until display list DMA is re-enabled. Jump and Load Memory Scan (LMS) commands simply act when repeated as though the LMS bit (bit 6) were not set. The last mode line continues to be repeated even across vertical blank.

Turning off display list DMA has no effect if a wait for vertical blank ($41) instruction is executing, as no fetches occur anyway while the instruction loops.

Bit 6 of the instruction register is cleared across vertical blank. This makes no difference except in the extremely rare case where display list DMA is enabled on cycles 0 or 1, late enough for the instruction byte fetch to be suppressed but early enough for the address fetches to occur.

Display list DMA enable/disable timing

Display list instructions are fetched on cycle 1 of a scan line, between missiles and players. However, display list DMA must be enabled by cycle 113 of the previous line in order for it to take effect at the beginning of the next line. If DMA is enabled on cycle 0, it still doesn't occur on the immediately following cycle.

Hi-res last scan line bug

Under normal circumstances, a display list should not be constructed such that scan line 247 is a hi-res scan line. This is not ordinarily possible with a normal display list, only with one that is too long or by repeating mode lines by disabling display list DMA. If scan line 247 is a hi-res line, then ANTIC will fail to properly activate vertical blank or vertical sync in the active playfield display region whenever bits 0-1 of DMACTL[3:2] are other than 00. This can result in severe display distortion if vertical sync on scan lines 251-253 (NTSC) or 275-278 (PAL) is

disturbed. Another side effect is that GTIA will continue to process player/missile graphics and P/M collisions in the non-blanked regions.

4.7 Scrolling

Normally, a playfield can only be scrolled by changing the memory scan pointer used to begin fetching data. This restricts scrolling to byte granularity, which is fairly large on-screen for most display modes. ANTIC has support for both fine horizontal and vertical scrolling, which allows playfields to be scrolled more finely than by LMS instructions.

Enabling horizontal scrolling

Bit 4 of a display list instruction enables horizontal scrolling for that mode line. This enables the fetch of extra playfield data and then shifts the playfield by the value specified in the HSCROL register, specified as the number of color clocks to shift the playfield right from 0-15. For a scroll value of 0, the visible playfield image is aligned as if the wider playfield were simply windowed to the requested width.

The same number of color clocks is displayed as without scrolling, so there are no visible scroll artifacts on the sides with horizontally scrolled narrow or normal width playfields. A wide playfield will shift in background color on the left with increasing scroll values, and also show a few color clocks of garbage on the rightmost border.

Effects on playfield DMA

Enabling horizontal scrolling increases the fetch width by one level, so a narrow playfield fetches the same data as a normal playfield, and a normal playfield fetches a wide playfield's worth of data. This increases the number of bytes per scan line accordingly, which must be taken into account when laying out playfield data. It also results in more playfield DMA cycles, impacting CPU speed and DLI timing. There is no change in fetch width for wide playfields.

Playfield DMA is delayed by one cycle for each increase by two in the HSCROL value. Even and odd scroll values have the same DMA timing and are differentiated by an optional single color cycle delay within ANTIC. With normal or wide playfields, the shift in DMA timing results in some DMA cycles being dropped near the end of the scan line. While ANTIC doesn't halt the CPU during these cycles, it does still fetch data from the bus into internal memory and increment the memory scan counter.

Scrolling high-resolution modes

High resolution modes cannot be scrolled with single pixel accuracy. It is only possible to scroll by pairs of pixels at a time because HSCROL only has color clock precision.

Scrolling GTIA modes

In GTIA modes, data from adjacent color clocks are paired together by GTIA to form 4-bit pixels. The pairing is determined relative to horizontal blank and is not affected by horizontal scrolling. This means that for proper scrolling of these modes HSCROL should be set to even values only. If odd values are used, ANTIC will delay the playfield data by a color clock unbeknownst to GTIA, resulting in the wrong pairs of bits being merged together into pixels.

Changes to HSCROL between rows of a mode line

For mode lines that are more than one scan line tall, it is possible to change HSCROL between scan lines within that mode line. This makes it possible to shear the mode line. The internally buffered data is replayed relative to the start of each scan line, so it moves as expected.

Changes to HSCROL in the middle of a scan line

The horizontal scroll value can also be changed in the middle of a scan line, but the effects are less intuitive. The LSB of HSCROL which controls the internal color clock delay can be changed at any time for immediate effect, shifting following displayed data by a color clock. Changes to bits 1-3, however, will not result in a visible change at the point of change since they change the starting and stopping cycles for playfield DMA. For instance, changing HSCROL from 0 to 4 would have no visible effect, but changing it from 0 to 5 would.

There are two artifacts that can occur at the end of the scan line, however, when changing bits 1-3. The first is the change in the playfield DMA end position can change the number of bytes that the memory scan counter is advanced, resulting in playfield data for the next scan line being displaced. For instance, changing HSCROL from 0 to 8 in the middle of a horizontally scrolled, narrow width mode 7 line will result in the memory scan counter being advanced by 21 bytes instead of 20. A more serious artifact occurs if the playfield DMA pattern for the new scroll value no longer aligns with the pattern that was established when DMA started; this happens if bit 1 is changed in modes 2-5/D-F, bits 1-2 in modes 6-7/A-C, or bits 1-3 in modes 8-9. Doing so changes the cycle at which ANTIC attempts to stop playfield DMA, and if it fails, playfield DMA continues through horizontal blank and into the next scan line.

Artifacts with wide playfields

With some combinations of IR mode and horizontal scroll values, it is possible for garbage to appear on the right side of a wide playfield. This garbage appears very far right and off the visible areas of most televisions, although some can display it. The garbage data is not random: it corresponds to activity on the data bus during playfield fetches blocked due to occurring too late in the scan line (see DMA timing charts). This is usually limited to 1-2 color clocks and is more likely to happen in character modes due to character data being fetched one cycle later relative to display than bitmap data. The effect can extend farther left if HSCROL is changed in the middle of a mode line to shift display of data in ANTIC's internal buffer.

Most of the time, the garbage is simply an unwanted artifact. However, because this data is sent to GTIA, it can be detected by player/missile collisions against the playfield and can be a source of unwanted collisions.

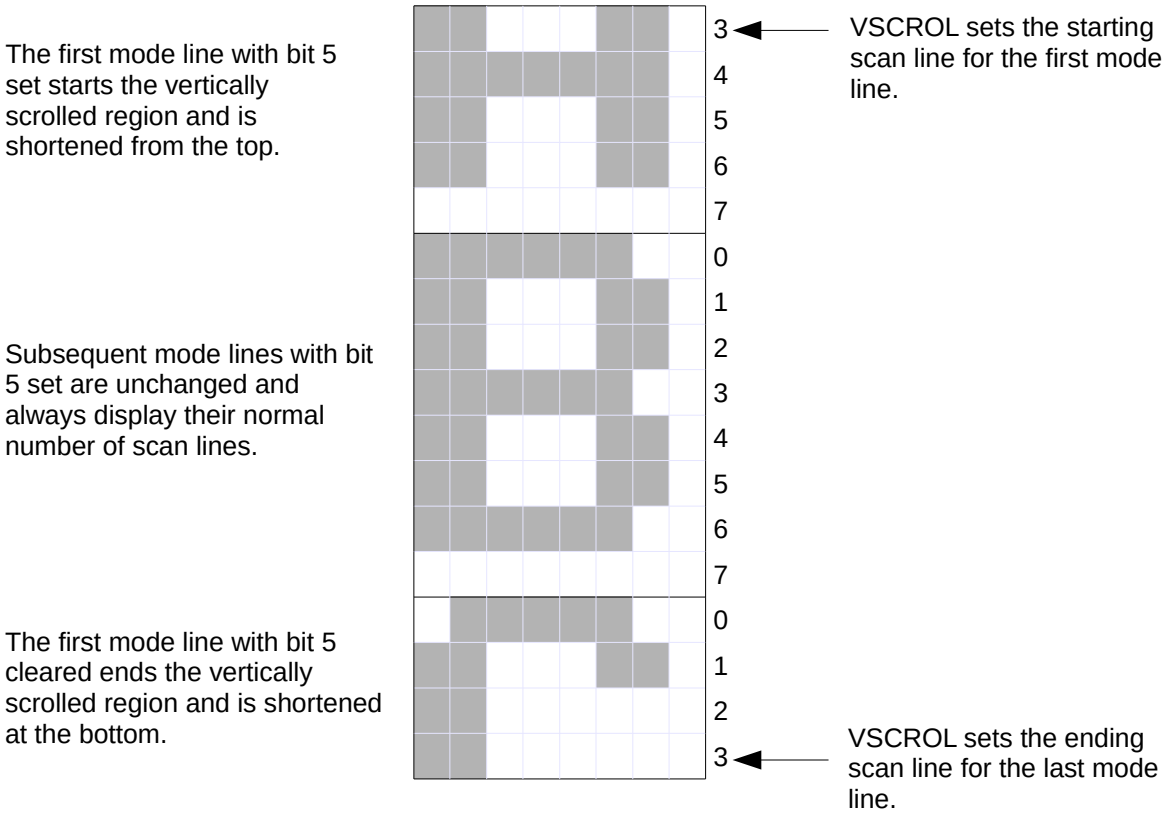


Figure 2: Effect of vertical scrolling on mode lines

Vertical scrolling

Vertical scrolling in ANTIC is controlled by bit 5 of a display list instruction. When bit 5 is set, the VSCROL [D405] register modifies the height of selected mode lines in the display list to allow portions of the display to be scrolled on a scan line basis. When the vertical scrolling bit changes from a 0 to a 1 on adjacent mode lines, the first line for which it is set is shortened by starting it at the scan line specified by VSCROL. Similarly, when it changes from a 1 to a 0, the first line for which that bit is reset is also shortened by ending it at that scan line. This means that a vertically scrolled region consisting of three mode 2 lines will have bit 5 set on the first two lines and occupy (8-VSCROL) + 8 + (VSCROL+1) = 17 scan lines instead of the usual 24.

VSCROL and the row counter are both 4-bit counters regardless of mode, and odd effects can be created by setting them to out of range values. For instance, a mode F scan line is only one scan line high and ordinarily vertical scrolling doesn't make sense. However, if VSCROL is set to 13 upon entering such a scan line, the row counter will count from 13 to 0, creating a mode F region where each pixel is four scan lines tall, but the DMA overhead is still only for one scan line. This is similarly possible when exiting the vertically scrolled region by setting VSCROL to 3 so that the row counter runs from 0 to 3. This creates the so-called “GTIA 9++” mode where GTIA modes can be run with lower vertical resolution with much lower DMA overhead than if LMS lines were used to produce the same effect.

There are different deadlines for VSCROL changes depending on what specifically is affected. For determining the initial row counter when entering a vertical scrolling region, VSCROL must be written by cycle 0, and for determining the final row for the end of a scrolled region, it must be written by cycle 108. The six clock window between these deadlines can be abused in order to halve the number of DLIs required to implement a turbo mode. This is done by writing VSCROL twice in quick succession, with the first value terminating the current

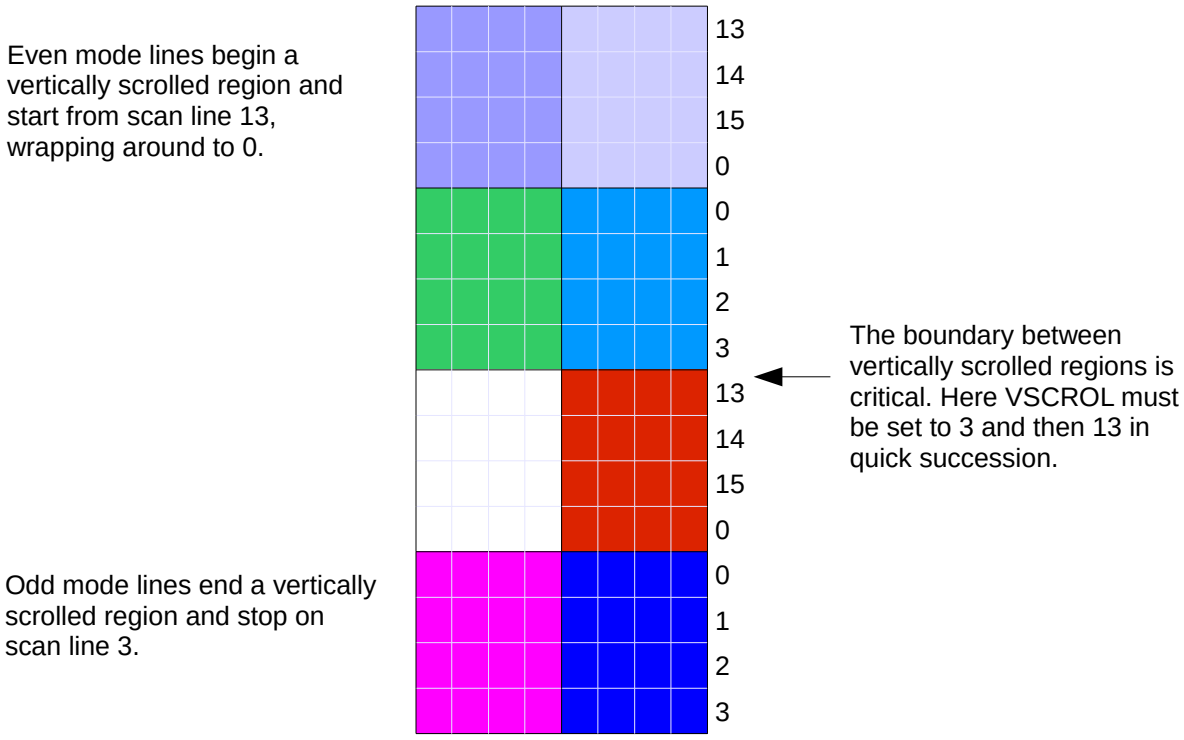


Figure 3: Abusing vertical scrolling in the “GTIA 9++” mode

mode line and the second value setting the height of the next. Finally, VSCROL must be written by cycle 5 to affect DLIs.

Vertical scrolling regions do not have to exclusively use the same mode, as the vertical scrolling functionality only affects the starting and ending mode lines via row count.

Blank mode lines ($x0) are always considered to have the vertical scroll bit cleared since the scroll bits are used for a blank line count instead. The blank line is still subject to height changes if it ends a vertically scrolled region, however. Jump instructions ($x1) can also have their height modified in the same way.

Mode lines with unusual height

All mode lines can be extended beyond their normal height up to 16 scan lines through vertical scrolling.

IR mode 0 lines are always blank, no matter how high.

IR mode 1 lines are always blank, but when extended beyond one scan line, re-fetch DLISTL and DLISTH on each scan line.

For IR mode 2, rows 8-9 are blanked for characters $00-5F and $80-BF the same way that they are for IR mode 3. Rows 10-15 are the same as rows 2-7.

For IR mode 3, rows 10-15 are the same as rows 2-7.

For IR modes 4 and 6, rows 8-15 are the same as 0-7.

For all bitmap modes (IR modes 8-15), all rows are the same. Regardless of how high the bitmap mode line is, the data fetch still always only occurs on the first scan line.

Mode 8/9 horizontal scrolling bug

IR 8 and 9 mode lines can be corrupted if they follow a horizontally scrolled mode line at normal or wide width. This occurs when the prior line uses IR modes 2-5 or D-F with HSCROL >= 10, or modes 6-7 or A-C with HSCROL >= 14. When this happens, the memory scan counter is incorrect unless reset with an LMS instruction, pixels are shifted out at incorrect rates, and scan lines within the mode 8-9 line are not aligned properly. This bug can occur regardless of whether the mode 8/9 line is horizontally scrolled, although the artifacts are different.

The effects can also carry over into subsequent mode 8/9 lines:

- **Non-scrolled IR mode 8/9 line:**
  - **Following mode 2-5/D-F, HSCROL=A-B or E-F:** Corruption carries over to subsequent scan lines.
  - **Following mode 2-5/D-F, HSCROL=C-D:** Resolves itself within two scan lines.
  - **Following mode 6-7/A-C, HSCROL=E-F**: Corruption carries over to subsequent scan lines.
- **Scrolled mode 8/9 line:**
  - **Following mode 2-5/D-F:** Resolves itself within three scan lines.
  - **Following mode 6-7/A-C:** Resolves itself within two scan lines.

The effect does not occur with narrow playfield width. The cause of this bug is the playfield DMA clock failing to stop properly; see Abnormal playfield DMA for details.

4.8 Non-maskable interrupts

ANTIC can assert two types of non-maskable interrupts to synchronize the CPU to the display. Vertical blank interrupts (VBIs) occur at the end of the displayable region and are used to synchronize to frames. Display list interrupts (DLIs) occur in the middle of the displayable region and are used to effect mid-screen changes that are not possible through the display list alone.

Enabling interrupts

Setting bits 6 and 7 of NMIEN enable DLIs and VBIs, respectively. Once an interrupt is enabled, ANTIC will then assert an NMI on the CPU at the beginning of scan line 248 for VBIs, or the last scan line of a DLI-enabled mode line. The NMI handler will then begin execution on the next instruction boundary at cycle 10 or later.

NMIEN must be written by cycle 7 to enable an interrupt and by cycle 8 in order to disable it.

Triggering a DLI

To trigger a DLI, bit 7 should be set on a display list instruction. This causes ANTIC to fire an NMI at the *start* of the *last* scan line for that mode line. Typically the DLI interrupt handler will then issue an STA WSYNC in order to synchronize to the end of the scan line, enabling it to write to hardware registers just prior to the next mode line at a time where the user will not see artifacts from the changes.

You can set the DLI bit on *any* mode line, including a blank mode line. The strangest use is when the DLI bit is set on a wait for vertical blank instruction ($C1); this causes a DLI to be issued on *every* scan line until vertical blank begins at scan line 248. Obviously, the DLI must be very short to run reliably in this situation, but it is possible.

If vertical scrolling causes a mode line with a DLI to be shortened, the DLI will still fire at the end of the shortened mode line, and just prior to the next mode line. This can cause surprises if a DLI is attempted at the start or end of a vertically scrolled region, because this can cause the DLI to occur on the more strongly contended first scan line.

Reading interrupt status

Since all NMIs from ANTIC route through a single vector on the CPU, the NMIST register is used to determine the interrupt source. Bit 7 indicates a DLI, bit 6 indicates a VBI, and bit 5 indicates that the system reset button was pressed (400/800 only). The status bits in NMIST are independent of the enable bits in NMIEN: interrupt status is reported even for disable interrupts.

The reset bit stays latched until cleared by NMIRES, but the VBI and DLI bits are mutually exclusive: the DLI bit is cleared at scan line 248, and the VBI bit is cleared whenever a DLI occurs. This means that it is generally unnecessary to test the VBI bit or write to NMIRES past boot – the NMI routine can test bit 7 for a DLI, bit 5 for reset, and then assume a VBI otherwise.

NMIST bits 6 and 7 are set starting on cycle 7 of a scan line where a VBI or DLI is active. Clearing those bits by writing NMIRES does not prevent the interrupt from firing, but can confuse an NMI dispatch routine.

Interrupt dispatch timing

The earliest that the CPU can normally begin execution of the seven-cycle sequence to enter the NMI handler is cycle 10, with additional delays as needed to finish the current instruction. However, if an IRQ triggers starting at cycles 5-9, its interrupt sequence can be co-opted by the NMI, allowing the NMI to execute correspondingly earlier.

If an interrupt is enabled on exactly cycle 7 of a scan line, NMI timing is delayed by one cycle to cycle 11.

DLI timing

Display list interrupts have extremely critical timing for two reasons: they have to change hardware registers within a very narrow window of time (usually horizontal blank), and they need to execute quickly to avoid conflicting with each other or stealing too much CPU from mainline and IRQ routines. As such, it is very useful to count exact cycles for DLI execution.

DLI execution proceeds as follows[12]:

- ANTIC pulls NMI at cycle 8 at the beginning of a scan line, right after display list and P/M DMA.
- The 6502 requires two cycles to acknowledge the NMI[13].
- 0-6 cycles pass as the 6502 finishes the currently executing instruction.
- Interrupt entry takes 7 cycles.

Thus, if you are writing a custom NMI handler, the earliest that the handler will run is cycle 17. Note that DMA contention will slow down this sequence, and it's virtually guaranteed that at least refresh DMA will interfere starting with cycle 25.

If the OS handler is used, then the OS will execute a BIT NMIST / BPL not taken / JMP (VDSLST) sequence before executing your handler, resulting in an additional 11 cycles of delay. Including refresh DMA, your handler will execute starting on cycle 28-36.

At this point, the normal procedure is to save registers as needed, load up registers with needed values, STA WSYNC, and then write values to hardware registers as quickly as possible during horizontal blank. Afterward, the exit path will frequently spill into the middle of the next scan line, but that is not nearly as critical.

Note that these timings assume that the DLI is occurring on a blank mode line. Any non-blank line will require playfield DMA cycles that will significantly delay interrupt routine execution: a normal-width mode 0 line will shift

12 De Re Atari also has a good description of DLI timing and explains how to break a DLI routine into phases by timing requirements.

13 [MOS76] 38

the entry window for the OS case to cycles 36-44, and horizontal scrolling or wide playfields makes this worse. Extra care is required when using DLIs around vertical scrolling, because it can shorten a mode line to only the first scan line, causing a DLI to fire on a scan line where the active region is blocked by solid playfield DMA. The extreme case occurs if the next mode line is also a character mode line, which can result in so much DMA contention that *two entire scan lines* pass before the 6502 can even enter the DLI handler.

Missed NMIs

If the 6502 responds to an IRQ starting at exactly cycle 4, any NMI that ANTIC would have triggered on cycle 8 will be lost.[14] This happens whenever the IRQ acknowledgment sequence occurs over cycles 4-10 and includes DLIs, VBIs, and on the 400/800, the SYSTEM RESET interrupt. NMIST is still updated as usual, however. The most visible artifact caused by this problem is glitching on screen if you attempt to use DLIs while an SIO transfer is in progress. However, it can happen with *any* IRQ source, including POKEY timers and the keyboard. It can also occur with an exactly timed BRK instruction. It cannot, however, occur with a regular instruction, not even one that takes seven cycles (INC abs,X).

DLIs and writes to VSCROL

A vertically scrolled region ends when the row counter matches the value in VSCROL. Normally, this happens shortly before the display list fetch at the end of the scan line. However, when a DLI is requested on the ending mode line, ANTIC must determine the end of the mode line much earlier in a scan line. Specifically, this happens shortly before the DLI would occur. A write to VSCROL that affects whether a DLI occurs on a scan line must occur by cycle 5. Writes after that point will be too late to block or trigger the DLI, but will still affect the height of the mode line.

4.9 WSYNC

A write to WSYNC [D40A] halts CPU execution until the end of a scan line, allowing the CPU to synchronize to the display. One more cycle elapses before the CPU is halted until cycle 105, when execution resumes around the start of horizontal blank. Because the CPU usually gets to execute the first cycle of the next instruction, this appears as if the instruction started on cycle 104. There are, however, three circumstances that can change this behavior:

- **If the cycle immediately after writing WSYNC is blocked.**

  In this case, the CPU doesn't get to execute the first cycle of the next instruction, and that instruction starts from the beginning as usual on cycle 105.

- **If playfield DMA extends to cycle 105.**

  Wide playfields, normal playfields with horizontal scrolling, and narrow playfields with high horizontal scroll values can encroach on cycle 105. This causes a one-cycle delay in the CPU restart.

- **If refresh DMA extends to cycle 105 or 106.**

  The first scan line of a character mode line can incur solid playfield DMA during the active region such that refresh DMA is pushed all the way to the end of the scan line. This can cause refresh DMA to occupy cycle 105, resulting in a one-cycle delay for the CPU. If playfield DMA is already occupying cycle 105, however, then it will push refresh DMA to cycle 106, resulting in a two-cycle delay.

These factors mean that there can be up to a three cycle variance in when an instruction following a write to WSYNC finishes execution, not counting interrupts. Therefore, if you are attempting to use a write to WSYNC to establish an event at an exact time on a scan line, your best bet is to write to WSYNC twice during vertical blank

14 Speculation on the AtariAge forums is that this is caused by a bug in ANTIC, which does not assert the NMI line long enough for the CPU to reliably acknowledge the interrupt.

or during blank mode lines, ensuring that no DMA interference occurs. You should also ensure that a DLI or VBI does not take place on the scan line as otherwise the interrupt is guaranteed to fire immediately after the instruction that writes to WSYNC.

Because the 6502 can only respond to interrupts at the end of an instruction, a write to WSYNC can cause long delays in interrupt response time. This is particularly problematic for DLIs, which can be pushed down by an entire scan line. Therefore, STA WSYNC should be avoided in main code when time-critical DLIs are in use. A loop on VCOUNT is a popular alternative:

```
        LDA VCOUNT
LOOP    CMP VCOUNT
        BEQ LOOP
```

Execution resumes anywhere between cycles 0-6 of the next scan line.

Deadline for writes to WSYNC

Writes to WSYNC up to cycle 103 wait until the start of horizontal blank on the current scan line. A write to cycle 104 or later is too late and causes a wait until the start of horizontal blank on the next scan line.

Read-modify-write instructions

Using a read-modify-write instruction such as INC or DEC to write to WSYNC causes special behavior because this is the only case where the cycle immediately following the write to WSYNC is another write cycle.[15] The 6502 does not respond to RDY during a write cycle and therefore always performs this write on the next available cycle regardless. As a result, an INC WSYNC instruction has the useful behavior of ignoring whether the next cycle is occupied by DMA, with the next instruction starting on cycle 105.

The deadline for the last cycle of a RMW instruction to write to WSYNC is still cycle 103. If the instruction executes one cycle later such that two write cycles occur on cycle 103 and 104, the behavior is slightly different: the next instruction will still start on cycle 105, but the second cycle of that instruction will be delayed until cycle 105 on the next scan line.

The 65C02 and 65C816 have different behavior when RDY is asserted during writes, so it is best to avoid relying on this behavior if compatibility with CPU accelerators is desired.

Bus activity during WSYNC

Because WSYNC works by asserting the RDY signal to the CPU, it effectively causes the CPU to retry its current read cycle repeatedly until RDY is negated. This will ordinarily be either the first or the second instruction byte of the next instruction after the write to WSYNC. Ordinarily this is of no consequence unless the address corresponds to a read-sensitive hardware device or the WSYNC wait occurs during a period when phantom DMA is occurring (see Scan line timing and Player/missile graphics).

4.10 VCOUNT

The VCOUNT [D40B] register reflects bits 1-8 of the vertical scan counter. Bit 0 is not connected, so this only permits two-line resolution. VCOUNT maintains its value up through cycle 109 and increments on cycle 110 of a scan line. For an NTSC machine, VCOUNT counts from $00 to $82; for PAL, it counts to $9B.

If you are using VCOUNT to check for a scan line near the top of the screen, consider using a greater-equal check rather than an equality check, as otherwise the test can lock up if the VBI handler takes too long to execute. This is a common cause of lockup when programs meant for PAL are run under NTSC, where there is

15 The 6502 can actually run up to three write cycles back to back if you include the interrupt acknowledgment sequence, where PCH/PCL/P are pushed onto the stack. However, since this is always to stack locations in page 1, WSYNC cannot be involved.

much less vertical blank time.

End-of-frame anomaly

ANTIC requires one additional cycle to detect that the vertical counter has hit the end of frame value and to reset it to $00. This means that reading VCOUNT on exactly cycle 110 of the very last scan line will give $83 (NTSC) or $9C (PAL), which correspond to scan lines 262 and 312, respectively; starting with cycle 111, it reads $00. This is the only cycle in the frame where this highest value can be seen and is thus extremely rare, but it could be a surprise to a DLI handler using VCOUNT to index tables.

4.11 Playfield DMA

Fetch rates

ANTIC supports three different fetch rates for playfield DMA. The slowest rate is one fetch per eight cycles and is used for modes 8 and 9. The medium rate of one fetch per four cycles is used for modes 6-7 and A-C. The fastest rate of one fetch per two cycles is used for modes 2-5 and D-F.

During the first scan line of a character mode, ANTIC fetches both character names and character data. The data fetch occurs three cycles after the corresponding name fetch. For modes 2-5, this causes ANTIC to occupy the bus with playfield DMA continuously with name and data fetches for a large portion of the scan line.

Line buffering

A 48 byte buffer within ANTIC is used to store graphic data for a single scan line. Its purpose is to buffer data for use on repeated scan lines, reducing DMA overhead. For bitmap modes, it allows ANTIC to only read graphics data for a mode line once, during the first scan line. For character modes, it holds the character name data which is then repeatedly used to fetch each scan line of character data from the character set.

Because only character names are buffered in character modes and not character data, the two text modes that have double-height characters – modes 5 and 7 – must still fetch character data on every scan line even though half of the fetches are redundant.

Loading the line buffer

The line buffer is loaded during playfield DMA on the first scan line of a mode line during character name or bitmap graphics fetches. Character data fetches are not loaded into the line buffer. During normal operation, this loads 8, 16, or 32 bytes for a narrow playfield, 10, 20, or 40 bytes for a normal-width playfield, or 12, 24, or 48 bytes for a wide playfield.

If playfield DMA is disabled during portions of the first scan line, the DMA cycles are disabled but the loads still occur at the standard times, loading the current values of the bus as bitmap or character data. The internal address counter also continues to advance as usual, so if playfield DMA is re-enabled later in the scan line loads into the buffer will resume with the correct internal address for each horizontal location. However, if playfield DMA is disabled early enough so that the playfield never starts on the first scan line, no loads will occur and the line buffer will not be modified at all.

The line buffer is never cleared. Narrow or normal width playfield loads preserve the unused contents at the end of the line buffer. It is not changed by a blank mode line or a jump and the contents also persist across vertical blank. By carefully toggling playfield DMA and stretching mode lines through abuse of vertical scrolling, it is possible to fill the screen with playfield with reduced or even total absence of playfield DMA cycles.

Line buffer addressing

The line buffer is addressed such that the first location is always accessed at the playfield start position. This means that if the same data is replayed with different start positions – either through varying HSCROL with

horizontal scrolling or by varying playfield fetch in DMACTL – the displayed graphics will shift to follow the change in the left playfield border.

If the mode line is changed, causing a change in interpretation or in data rate, the buffered data is replayed just as if it were fetched from memory. For instance, if the line buffer is loaded with a normal mode E line and then replayed in mode 8, the first 10 bytes of the mode E line will be reinterpreted as mode 8 data.

Dynamic changes to playfield width

The playfield width bits in DMACTL[1:0], and the horizontal scroll position bits in HSCROL[3:1], determine the start and stop positions of the playfield on each scan line. Normally, ANTIC starts the playfield at the start position and stops the playfield at the stop position. Moving the timing of the start and stop positions dynamically can cause unusual playfield widths.

For the playfield start position, the deadlines for setting the playfield start position are cycles 24, 16, and 8 for narrow, normal, and wide fetch widths. Bits 1 and 0 of DMACTL must be set to the desired value by these cycle numbers to take effect. When horizontal scrolling is active, the deadline is delayed by one additional cycle for every increase by two in the HSCROL value. Various writes then have the following effects:

- Moving the start later (narrower fetch width or greater scroll value) takes effect as expected if done by the deadline, and is ignored for the current scan line if done too late.
- Moving the start earlier will still take effect if written by the deadline for the *new* width (earlier deadline). If the start is moved earlier by the deadline of the old position and after the deadline of the new position, the playfield will not start at all since the start has been moved back behind the current position.

The playfield stop position acts similarly, with corresponding deadlines of cycles 88, 96, and 104. Moving the stop later by the earlier deadline extends the playfield to the farther stop position. Moving the stop earlier behind the current position extends the playfield to the wide stop position, which is always active.

By changing the width and horizontal scroll values on the fly, it is possible to start and stop the playfield at mismatched positions. For instance, changing the playfield width from narrow to normal in the middle of the scan line with mode E will extend the playfield on the right side and cause additional bytes to be fetched. The resulting playfield is 144 color clocks wide and advances the memory scan counter by 36 bytes.

> **Warning**
>
> It is easy to accidentally hit one of these corner cases when changing DMACTL from a DLI handler, since the window for cleanly changing the playfield width is very narrow. If you are using WSYNC to synchronize, you only have a few instructions afterward to write DMACTL before you are in the danger zone. Timing for changing DMA parameters is much tighter than those for display parameters, so change DMACTL before modifying color registers. Symptoms that you are hitting DMACTL too late include losing a line when trying to enable DMA, gaining an extra line when trying to disable it, or having subsequent playfield addressing screwed up unless LMS instructions are added to the display list.

Disabling playfield DMA

Setting DMACTL bits 1-0 to %00 disables the playfield, shutting off both DMA cycles and the display. The playfield is always absent (background color) whenever playfield DMA is disabled. If it is disabled in the middle of an active playfield, it vanishes until re-enabled. This is true even in high-resolution modes: background is displayed, not PF2.

If playfield DMA is disabled before the playfield starts, the memory scan counter and line buffer are not updated. However, if disabled after playfield DMA starts, the memory scan counter continues to count and the line buffer is still loaded according to the current DMA pattern.

Mid scan line changes to playfield DMA

Changing the playfield DMA mode via the low two bits of DMACTL in the middle of a scan line has a number of interesting effects. Much of this is related to the scan line buffer within ANTIC, which buffers some but not all of the data between scan lines. Specific cases:

- In IR modes 2 and 3, the invert state is also not updated while DMA is disabled, resulting in the blanked scan line from the previous case displaying either PF2 or PF2L1 depending on the last seen invert state. This only occurs on the affected scan line; subsequent scan lines will once again show the correct invert state according to the buffered character names in the line buffer as long as DMA is re-enabled.
- For mode lines that span multiple scan lines, suspending playfield DMA for a portion of the first scan line results in portions of the line buffer not being updated. Previously written data in those portions are reused in display for subsequent scan lines. In character mode, this results in old character names being used.

4.12 Abnormal playfield DMA

Under certain circumstances, ANTIC can lose track of playfield DMA such that it begins fetching playfield data with an abnormal pattern, producing a garbled playfield. This can also scramble the display list, which can in turn crash the CPU by issuing bogus DLIs. As these effects are very difficult to control, typically this condition is simply an unwanted artifact to avoid.

DMA clock

There are two clocks within ANTIC that control playfield display, the DMA clock and the shift clock. Both are constructed as shift rings with taps to read cycling bits and extra gates to inject or clear bits in the cycling pattern. The first of these, the DMA clock, controls the timing of DMA cycles and the incrementing of the memory scan counter. It runs at machine cycle rate and is either two, four, or eight cycles long depending on the fetch rate required for the current playfield mode. Three taps off this clock produce the requests for character name, bitmap data, and character data at 0, +2, and +3 cycle offsets, respectively.

A single bit is injected into the DMA clock at playfield start, and that single bit position is cleared at playfield stop. The DMA clock is also unconditionally cleared whenever the current IR mode corresponds to a blank line or jump, or during vertical blank.

Shift clock

The shift clock, on the other hand, controls the shifting of graphics data out of the graphic shift register. It is a four-bit ring and runs at color clock rate, twice as fast as the DMA clock. There are taps at all four bits and either one, two, or all four of them are enabled depending on the required shift rate for the graphic shift register, which shifts either one or two bits per interval.

ANTIC clears both the shift clock and the shift register during special DMA time (cycles 0-7). The shift clock starts running when bits are injected into it from the DMA clock by means of the bitmap or character data fetch, synchronizing it to the arrival of the first graphics byte from either the bus or line buffer RAM. It is not stopped at playfield stop, simply continuing to run to clear out the shift register.

Table 5 gives the rates for both clocks for each mode.

IR Mode	DMA rate	Shift rate	Shift mode
2, 3, 4, 5	Fast (every two cycles)	Fast (1/cc)	2-bit
6, 7	Medium (every four cycles)	Fast (1/cc)	1-bit
8	Slow (every eight cycles)	Slow (1/4cc)	2-bit
9	Slow (every eight cycles)	Medium (1/2cc)	1-bit
A	Medium (every four cycles)	Medium (1/2cc)	2-bit
B, C	Medium (every four cycles)	Fast (1/cc)	1-bit
D, E, F	Fast (every two cycles)	Fast (1/cc)	2-bit

Table 5: DMA and shift clock rates by mode

Disrupting the DMA and shift clocks

As noted earlier, ANTIC stops the DMA clock by resetting a single bit in it at playfield stop time. Changing registers mid-scanline in a way that shifts the playfield stop position can cause ANTIC to clear the wrong bit and prevent it from stopping the DMA clock properly. When this happens, the DMA clock continues to run through horizontal blank and into the next scan line. This causes several undesirable results:

- Playfield DMA continues across horizontal blank and into the next scan line. This also advances the memory scan counter by additional steps, resulting in skipped playfield bytes. Note that playfield DMA cycles are still suppressed during cycles 105-111 and 0, so any extra cycles during that window are still virtual DMA cycles.
- DMA fetches can overlap. This can occur between playfield DMA itself – character name and character data fetch – or with special DMA such as display list and player graphics fetches. When this happens, the address used is the logical AND of all fetches involved and the fetched data is used for all of the DMA requests. A refresh DMA cycle cannot overlap, however, as it is only triggered by the absence of other DMA requests.
- The clocks can run at faster than normal rate or with erratic timing. ANTIC can fetch continuously at one fetch/cycle even in graphics modes if the DMA clock is disrupted. When the shift clock is disrupted separately, pixels are shifted out to GTIA faster than normal for the mode line and 00 pixels are shifted out whenever the 8-bit shift register runs out of data bits.

Disrupting the DMA clock with HSCROL

Once the DMA clock is running, ANTIC attempts to reset a single bit in the DMA clock at exactly two points: the playfield stop position for the current width setting, and the playfield stop position for a wide playfield. The stop positions for all playfield widths are multiples of eight cycles apart and thus the wide playfield stop aligns with the DMA pattern started at any playfield width. Therefore, it is not possible to disrupt the DMA clock with width changes alone as ANTIC will stop the clock on its second attempt and the playfield will only be extended to the wide playfield stop position.

Horizontal scrolling is another story, as for every two color clocks in horizontal scroll the playfield start and stop positions are shifted by one cycle. The cycle pattern for the ending HSCROL value must match the cycle pattern of the starting HSCROL pattern for the DMA clock to stop properly. For instance, in mode 2 the DMA clock runs at a rate of one fetch per two cycles, so the HSCROL bit 1 must match up for the start and stop patterns to line up with even or odd cycle timing. Similarly, in mode 8, the clock is running at a rate of one fetch per eight cycles, so HSCROL bits 1-3 must match exactly. When this occurs, playfield DMA will stop cleanly, although the scan line may be an unusual number of pixels long.

When the start and stop patterns do not line up, the DMA clock will continue running. ANTIC will continue to set

and unset bits in the DMA clock on subsequent mode lines. Therefore, it is possible to build up or drop additional fetch cycles, leading to progressively more or less screwy DMA patterns.

What makes this bug especially problematic is that the DMA clock runs rather late into horizontal blank when horizontally scrolling at wide fetch width. This means that it is easy to accidentally trigger it by changing HSCROL on the fly in a DLI handler right after writing to WSYNC. The deadlines for affecting this behavior with HSCROL are the same as for moving the playfield stop with DMACTL: the write must occur three cycles before where the next character name fetch would occur in the pattern, or in a bitmap mode, five cycles prior to the next graphics fetch. For a normal character mode playfield, this is on or before cycle 95 + HSCROL/2. ANTIC always tries again at the equivalent wide stop, for which the write must happen on or before cycle 103 + HSCROL/2. This means that in order for a horizontally scrolled normal or wide width line to display correctly, HSCROL should not be rewritten before cycle 111, three cycles before missile DMA fetch.

Disrupting the DMA clock with mode switching

Abnormal DMA patterns can also occur simply with specific orders of mode lines where the DMA clock slows down between the two mode lines. This happens because the DMA clock is always eight bits long even though the ring part is restricted to four or two bits for medium and fast shift rates, and thus it takes four or six clocks for any bits left in the clock to completely shift out. The DMA clock runs so late into horizontal blank when horizontal scrolling is active at normal or wide playfield width that these latent bits can be recaptured when the ring part of the clock is suddenly extended at the switch to the slowest speed. These extra bits then cause an abnormal DMA condition.

For this problem to occur, a playfield character name fetch must have been scheduled within cycles 109-111 for a character mode, or a graphics fetch within cycles 111-113 for a bitmap mode. The only conditions that can cause this are:

- Horizontally scrolled normal or wide width mode line at fast DMA fetch rate (modes 2-5 or D-F), with HSCROL >= 10.
- Horizontally scrolled normal or wide width mode line at medium DMA fetch rate (modes 6-7 or A-C), with HSCROL >= 14.
- Existing abnormal DMA condition including those fetch cycles.

These fetches do not have to be actual DMA cycles as the DMA clock still runs during subsequent mode lines to fetch from the internal line buffer. The bits captured during these 1-3 cycles then become extraneous fetches in the 4-bit or 8-bit playfield DMA pattern for the next scan line.

Abnormal DMA patterns across scan lines

An abnormal DMA condition will persist across multiple scan lines as long as errant bits continue to cycle around the DMA clock and it is not stopped by a blank line or other clearing condition. However, because the scan line is 114 cycles long and not evenly divisible by the length of the DMA clock, the abnormal DMA pattern will change on each scan line when the DMA clock is operating in slow or medium speed modes where it is eight or four cycles long. This can result in the abnormal pattern resolving itself after a few scan lines as ANTIC “sweeps” over the abnormal pattern at different offsets, removing one or more errant bits each time.

As an example, changing HSCROL from $00 to $04 in the middle of a horizontally scrolled mode 8 line will shift the offset of playfield DMA cycles from %10000000 to %00100000 after the start bit has been injected into the clock, preventing the stop from occurring and causing the former pattern to stay in the DMA clock. However, because 114 mod 8 = 2, the errant pattern will have shifted by two clocks on the next scan line, resulting in subsequent extra DMA patterns of %00000010, %00001000, and %00100000. The last pattern lines up with the normal pattern of HSCROL=$04, so the errant bit will be cleared by the playfield stop, ending the abnormal DMA.

Similarly, if HSCROL is instead changed from $00 to $02, a four-line cycle of patterns %01000010, %01001000,

%01100000, and %11000000 will result.

Abnormal shift patterns

The shift clock is reset at the beginning of each scan line and initialized based on the pattern of DMA cycles produced by the DMA clock, which means that the shift clock can only run abnormally if the DMA clock is abnormal. However, the shift clock runs double speed at color clock rate and is only four bits long, which means only two bits can be affected by the even and odd fetches from the DMA clock. Furthermore, mode 8 is the only mode in which the shift clock can be disrupted because every other mode already requires the playfield shift register to shift at least once per machine cycle anyway.

In mode 8, the shift pattern is abnormal if the DMA pattern includes both even and odd cycles. When this happens, the shift clock then runs at double normal speed, producing pixels at two color clock resolution (80 across) instead of four color clock (40 across) resolution. If this causes the shift register to empty before it is reloaded again, the background is produced (pixel code 00).[16]

In all modes, the additional DMA cycles will also result in extra loads into the shift register. The extra data is ORed into the contents of the shift register. In character modes, this happens prior to the effects triggered by character name bits 6 and 7, such as inversion/blinking in IR modes 2 and 3 and the color changes in IR modes 4-7. This means that the next time a character name is read, the new values of bits 6 or 7 will immediately take effect, even for bits that have yet to shift out of the playfield shift register.

Abnormal line buffer addressing

Ordinarily, ANTIC never advances beyond the 48th location in the line buffer. An abnormal DMA clock, however, can advance the line buffer address faster at up to double normal speed, causing the line buffer address to exceed that limit or even wrap. The internal address counter is a 6-bit maximal length polynomial counter and has a sequence of 63 addresses. The first 48 addresses correspond the internal RAM and there is no response to the last 15 addresses. This means that when the line buffer is loaded, the entire 48 byte RAM is loaded before 15 fetches are discarded, and then the RAM is reloaded again. Similarly, during display, the 48 byte buffer is displayed and then the last 15 locations result in $FF data.

The second anomaly that can occur is that ANTIC can skip addresses in the line buffer when reading from it on back-to-back cycles in a bitmap mode. Specifically, whenever there are back-to-back cycles, all but the last fetch of the sequence will use the data from one later position. As a result, the value that should have been fetched first will be dropped and the last value will be used twice. This happens even on the first line where DMA fetches occur, because the data is first written to and then read from the line buffer. Only the reads from the line buffer are affected; the writes occur to the expected addresses and the buffered data will be normal if replayed on a subsequent mode line with a normal DMA clock.

Overlapping DMA

Abnormal DMA patterns can cause DMA cycles to overlap. In a character mode, character name and data fetches can occur at the same time when the DMA clock causes both even and odd fetches. When this occurs, the bitwise AND of the two addresses is used as the fetch address and the returned data is used for both fetches.

A DMA conflict can also occur between special DMA at cycles 0-7 and playfield DMA. As with playfield-playfield DMA conflicts, the bitwise AND of all addresses is used and the fetched data goes to all requests. However, this can occur even if playfield DMA is disabled in DMACTL. Display list DMA, missile DMA, and player DMA can be affected by this conflict.

16 The reason this can happen, despite the DMA clock also running at double rate, is that the extra bits in the DMA clock may not be evenly spaced. A second fetch can partially overlap the first in the shift register, leaving a gap.

> **Warning**
>
> The potential for overlap with display list DMA is what makes the abnormal playfield DMA bug a serious one. If it just affected the playfield, then the only problem would be visual glitching. When abnormal playfield DMA overlaps display list DMA, however, it can send the display list execution off into the weeds. This can then cause wild display list interrupts to fire and the program to crash.

Resetting the playfield clocks

Whenever an appropriate playfield stop position is reached, ANTIC clears bits from the DMA clock. If there are no other bits left flying around in the clock, the abnormal condition is ended. Entering vertical blank or executing blank mode display list instructions ($x0 or $x1) will also unconditionally clear the DMA clock and end any abnormal DMA pattern.

Switching to a mode line with a faster shift rate will shorten the recirculating portion of the DMA clock. Once this happens, any extraneous bits in the non-circulating portion will shift out and no longer contribute to abnormal DMA.

Since the shift clock is reset by ANTIC at the beginning of each scan line, clearing an abnormal condition in the DMA clock will automatically fix the shift clock.

4.13 Player/missile DMA

ANTIC can fetch graphics data for players and missiles on behalf of GTIA. Bit 3 of DMACTL enables player DMA, and bit 2 of DMACTL enables missile DMA. Missile DMA is forced on if player DMA is enabled in order to preserve proper timing against GTIA.

Vertical resolution

Bit 4 of DMACTL switches between two-line and one-line resolution. This simply changes the addressing that ANTIC uses to fetch player data. If one-line resolution is selected (bit 4 = 1), each player/missile occupies 256 bytes of memory and unique data is fetched per scan line. If two-line resolution is selected, each player/missile occupies 128 bytes of memory and each byte is fetched twice on adjacent scan lines.

P/M graphics memory layout

The address of player/missile data is specified by PMBASE [$D407]. In two-line resolution mode, player/missile data must be aligned on a 1K boundary and the upper six bits of the address are specified by bits 2-7 of PMBASE. In one-line resolution mode, P/M data must be aligned on a 2K boundary and the upper five bits of the address are specified by bits 3-7 of PMBASE, with bit 2 being ignored. However, bit 2 of PMBASE is still stored and becomes active if resolution is switched back to two-line without writing to PMBASE again.

The P/M graphics memory is in turn split into 8 sections of 128 or 256 bytes each. The first three sections are unused. The fourth section, starting at offset $0180 or $0300 from PMBASE, contains the four missiles; bits 7-6 correspond to missile 3 and bits 0-1 correspond to missile 0. The last four sections starting at $0200 or $0400 contain the graphics for players 0-3. Within each section, bits 0-7 or bits 1-7 of the vertical scan counter are used as the offset for fetching graphics data.

P/M DMA timing

When enabled in DMACTL, player and missile data is fetched on each scan line within the visible region (8-247). This means that in one-line resolution mode, the first and last 8 bytes of each section are always unused. Missile data is fetched during cycle 0 while player data is fetched during cycles 2-5.

In two-line resolution mode, bit 0 of the vertical resolution counter is ignored and each byte is fetched twice and sent to GTIA on consecutive scan lines. This means that the P/M graphics can still change on each scan line if

the data is modified in between. The only difference between one-line and two-line resolution is in addressing.[17]\

P/M DMA enable timing

Player/missile DMA must be enabled or disabled in DMACTL at least two cycles in advance to take effect. In particular, disabling missile DMA only one cycle earlier at cycle 113 will not prevent missile DMA from immediately occurring on the following cycle 0.

4.14 Scan line timing

Memory refresh DMA

Nine cycles of refresh DMA occur on every scan line in order to refresh DRAM, starting at cycle 25 and occurring every four cycles after that. These refresh cycles occur even in vertical blank. Refresh DMA can be blocked by playfield DMA, in which case the refresh cycle occurs on the next free cycle. Only one such cycle can be deferred at a time and any additional blocked refresh cycles in a row are simply dropped. This only occurs in the first scan line of modes 2-5, where memory is so contended that only 1-2 refreshes can fit.

In wide character modes, the final refresh cycle can be pushed all the way to the end of playfield DMA at cycle 105 or 106, resulting in an additional cycle of delay for a WSYNC on that scan line.

Data output from the RAMs is not enabled during refresh cycles and the data bus is undriven during refresh cycles. This leads to either a pulled up or floating data bus condition, depending on the memory configuration.

Display list DMA

The display list requires one DMA cycle for each mode byte, which occurs at cycle 1, between players and missiles. Mode lines that perform an LMS or a jump also fetch an additional address word at cycles 6 and 7. This fetch occurs at the beginning of the scan line where the mode line takes effect visually.

For modes that span multiple scan lines, the display list fetch only occurs on the first scan line. The jump and wait for vertical blank (JVB) instruction is also only fetched once regardless of the number of scan lines until vertical blank.

Playfield DMA

Three playfield widths are available: narrow, normal, and wide. Normal playfields are 80 cycles wide, while narrow playfields are 64 cycles and wide playfields are 96 cycles long. All fetch windows have the same center, with each wider setting adding 8 clocks on each side. There is a hardware stop that prevents playfield DMA from going beyond cycle 105. Any fetch cycles that would occur on cycle 106 or later are suppressed, although the playfield memory address is still incremented.

Enabling horizontal scrolling automatically causes narrow playfields to use the normal fetch window and normal width playfields to use the wide fetch window. No additional data is fetched for wide scrolled playfields. Horizontal scrolling causes the playfield fetch window to be delayed by one cycle for every two color clocks of scroll. The additional color clock delay required by odd scroll values is given by internal buffering.

Mapped mode playfield DMA

The mapped graphics modes have three horizontal densities, resulting in fetches every eight clock cycles (modes 8-9), four cycles (modes A-C), or two cycles (D-F). These occur on the first scan line of the mode. ANTIC internally buffers the data so that modes that span more than one scan line do not need to fetch any data on subsequent scan lines. This is used to powerful effect in the so called “GTIA 9++” modes, where mode F lines

17 [AHS00] p.45 contains a couple of errors. Each fetched missile or player consumes 240 bytes per frame, not 226, and two-line resolution mode takes the same number of cycles as one-line mode, not half.

are extended to four scan lines by vertical scroll trickery, resulting in one-fourth vertical resolution with one-fourth the bandwidth requirements.

Mapped playfield DMA begins at clock 26, 18, or 10 depending on width.

Character mode playfield DMA

Character modes have two horizontal densities, resulting in name fetches every two clock cycles (modes 2-5) or every four clocks (modes 6-7). The character names are fetched with the same timing as for mapped mode data, at clocks 28, 20, and 12 for the various widths.

Additionally, in these modes the character data itself must be fetched. The data fetch occurs three clocks later than the name fetch. Although the names are buffered internally by ANTIC, the character data isn't, and is always fetched for each scan line regardless of whether double-height modes are used (modes 5 and 7).

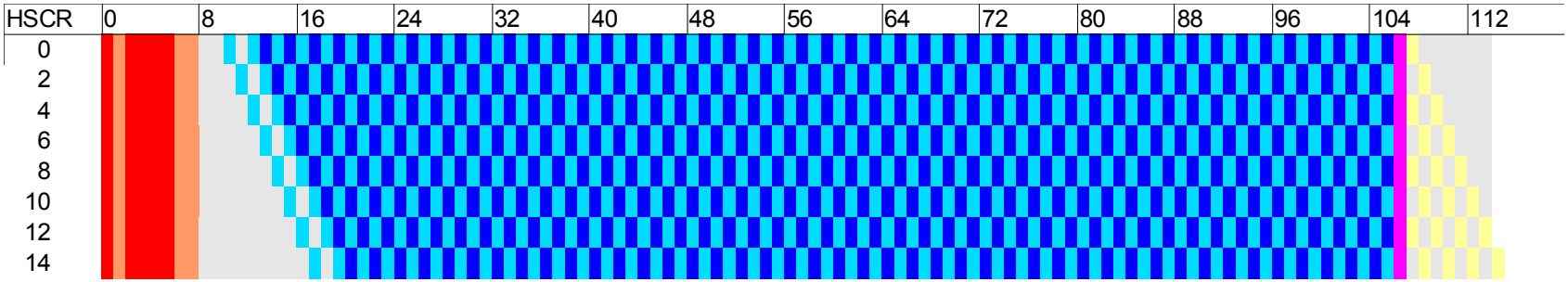
Virtual DMA cycles

Playfield DMA cycles that would occur on cycle 106 or later are blocked by the hardware and do not occupy the bus or stop the 6502. However, ANTIC still reads the data bus and stores or interprets the data on those cycles. This usually results in 6502 bus activity being loaded as playfield data. In rare cases, it is possible for a refresh cycle to overlap with a virtual DMA cycle, resulting in floating bus data being used.

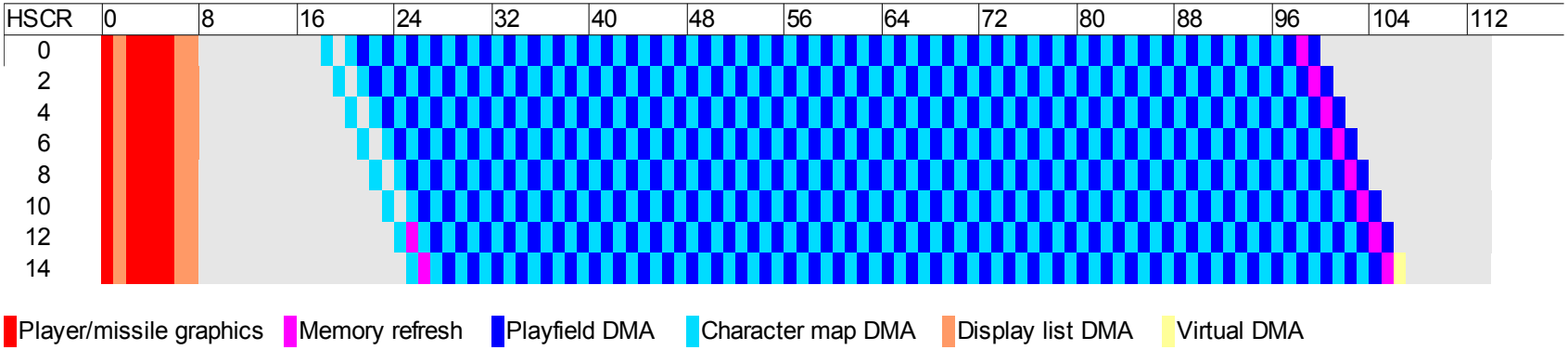
DMA timing charts

The following charts show the timing of per scan line DMA, based on various modes and settings. IR mode, playfield width, P/M graphics, LMS instructions, and horizontal scrolling all affect DMA timing. Note that the charts are arranged by fetch width, so a narrow playfield with horizontal scrolling is actually described by the normal playfield chart. There are no charts for subsequent scan lines for mapped modes, as no playfield DMA occurs in that case. HSCR refers to the HSCROL value, if horizontal scrolling is enabled; odd values have the same DMA pattern as the next lower even value.

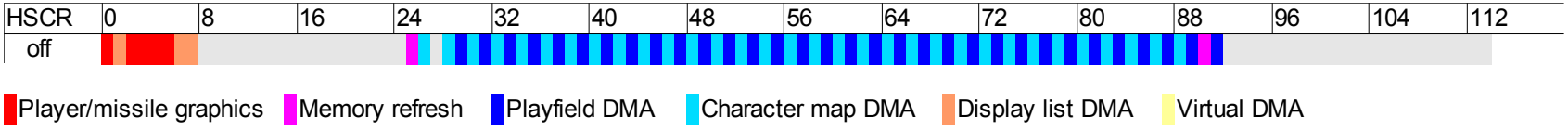
ANTIC modes 2-5, mode line, wide playfield



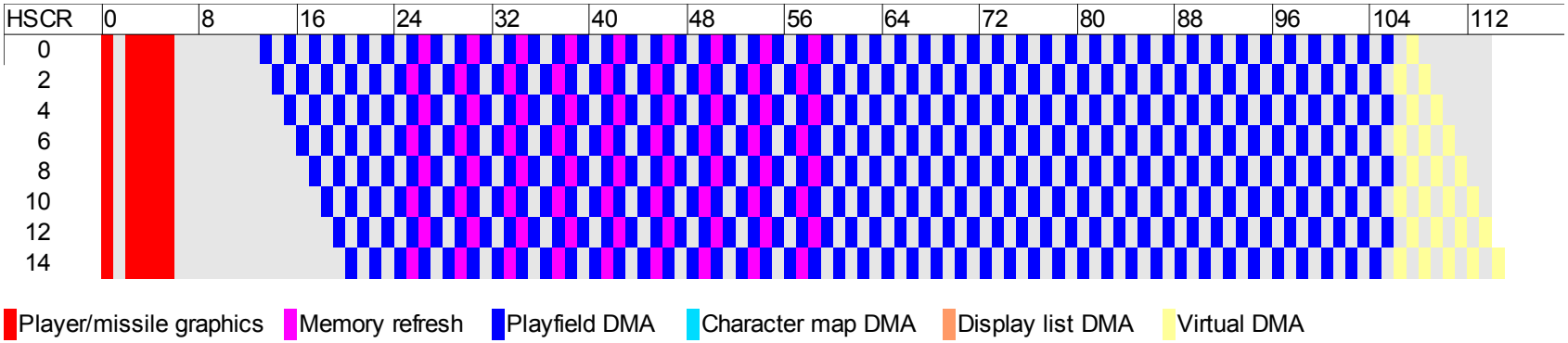
ANTIC modes 2-5, mode line, normal playfield



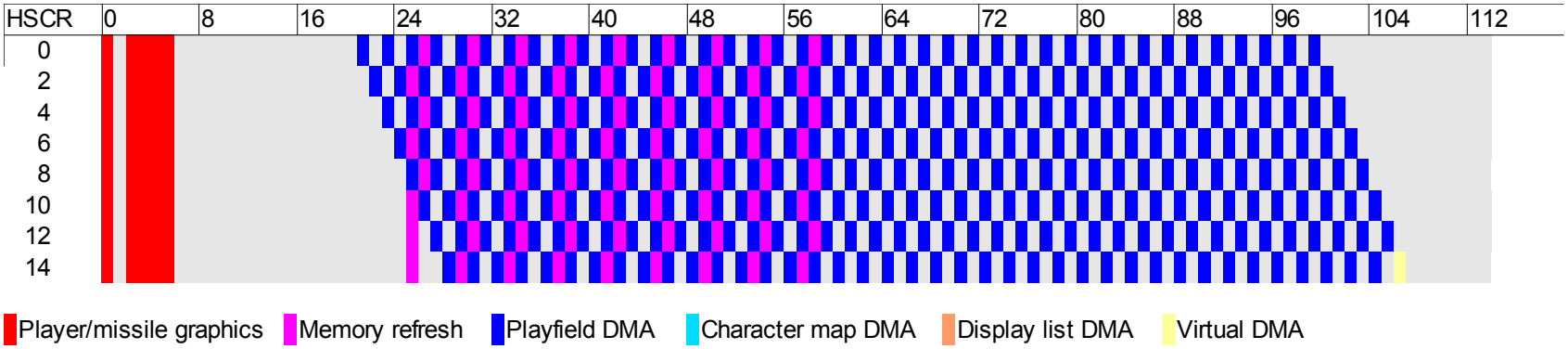
ANTIC mode 2-5, mode line, narrow playfield



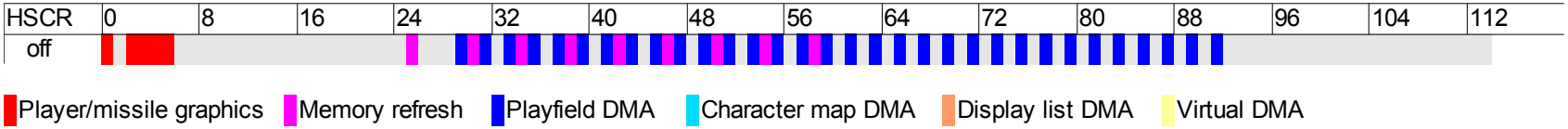
ANTIC modes 2-5, subsequent lines, wide playfield



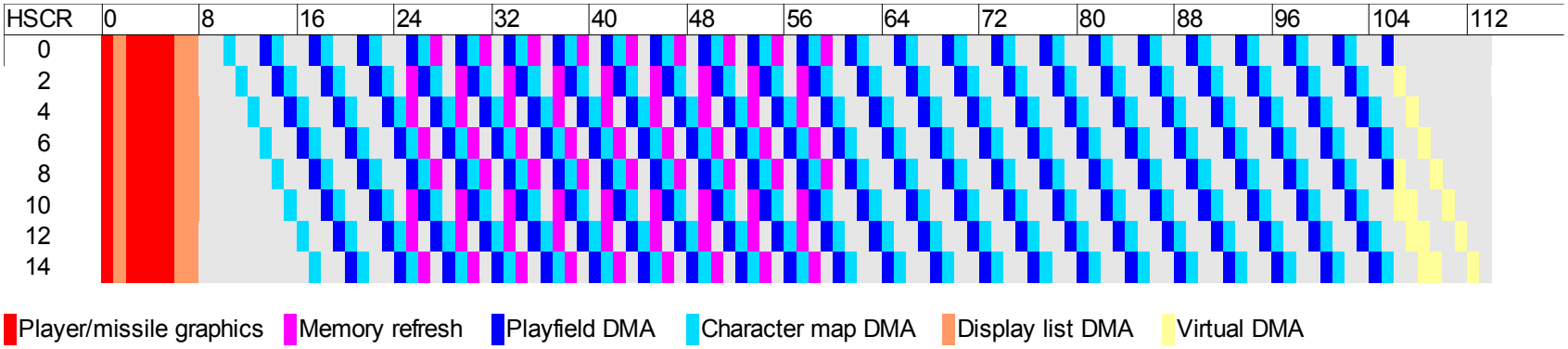
ANTIC modes 2-5, subsequent lines, normal playfield



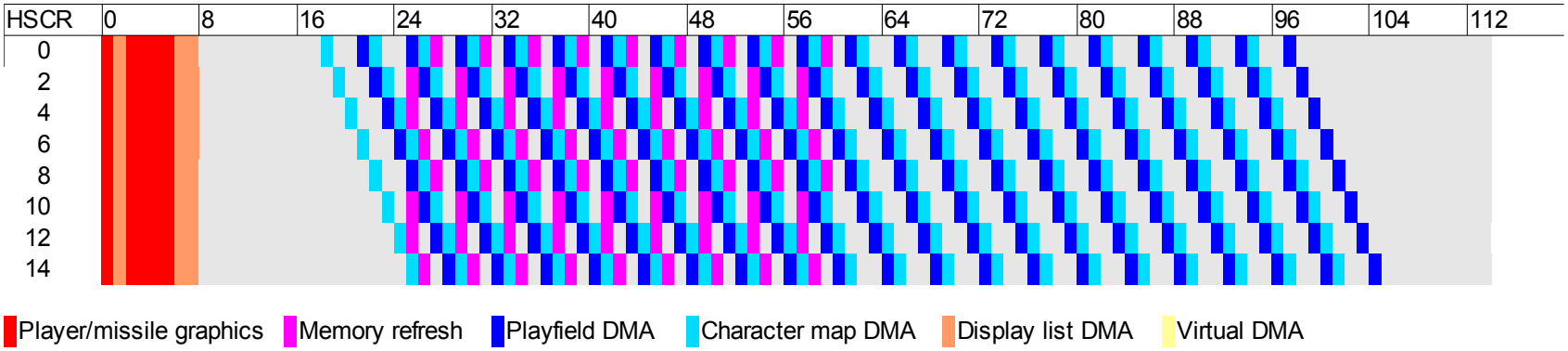
ANTIC modes 2-5, subsequent lines, narrow playfield



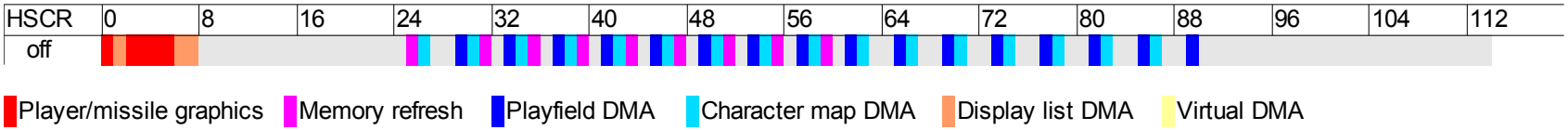
ANTIC modes 6 and 7, mode line, wide playfield



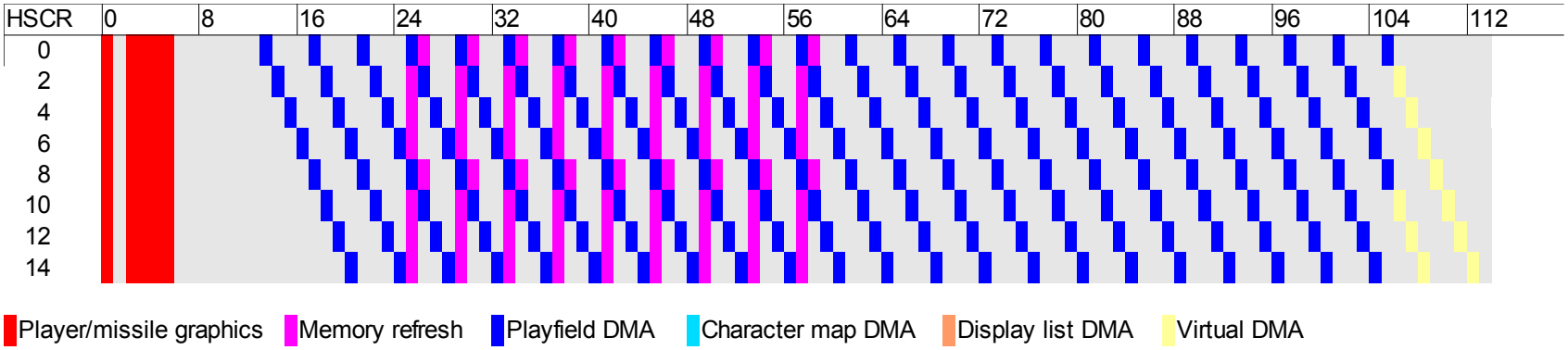
ANTIC modes 6 and 7, mode line, normal playfield



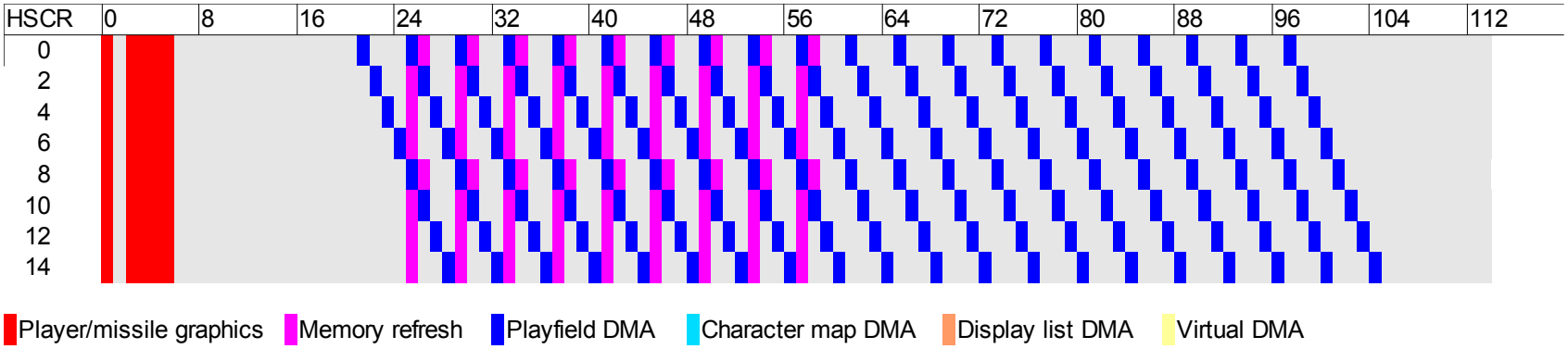
ANTIC modes 6 and 7, mode line, narrow playfield



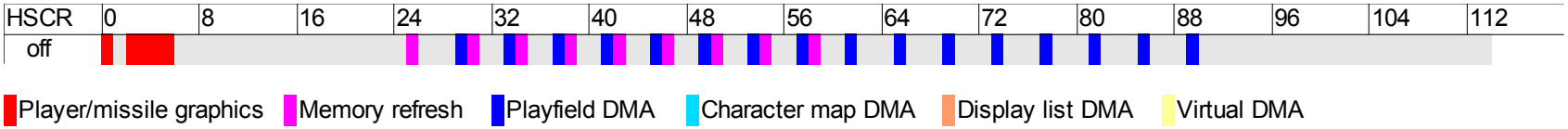
ANTIC modes 6 and 7, subsequent lines, wide playfield



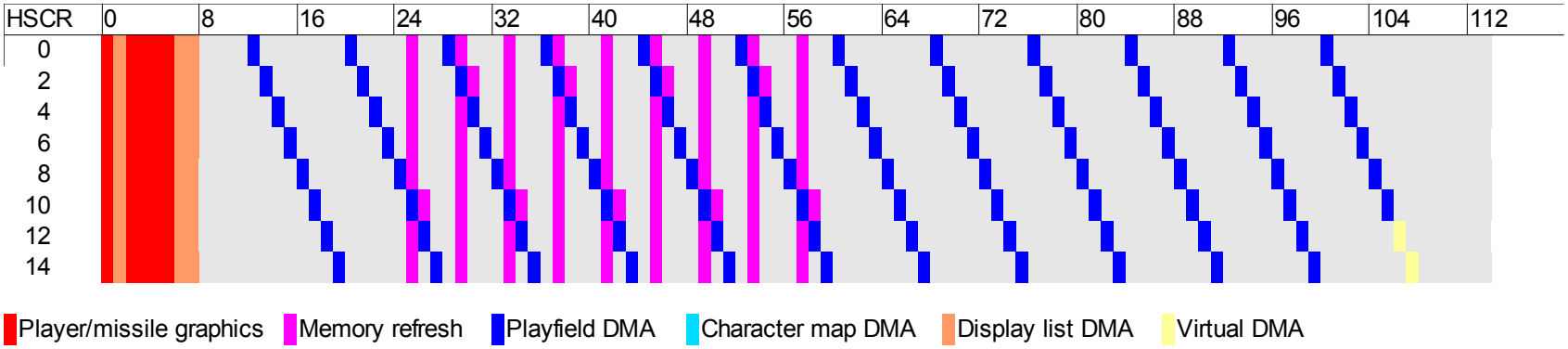
ANTIC modes 6 and 7, subsequent lines, normal playfield



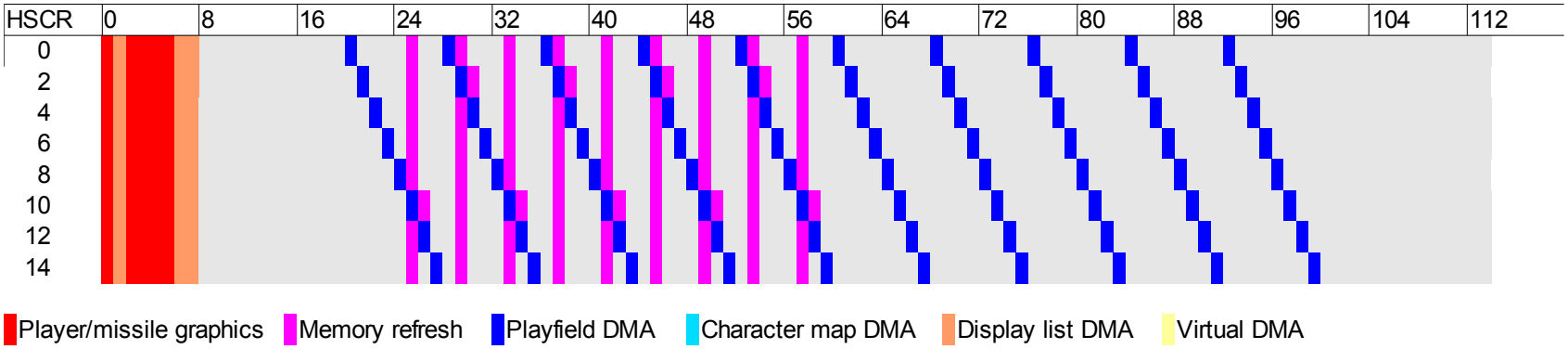
ANTIC modes 6 and 7, subsequent lines, narrow playfield



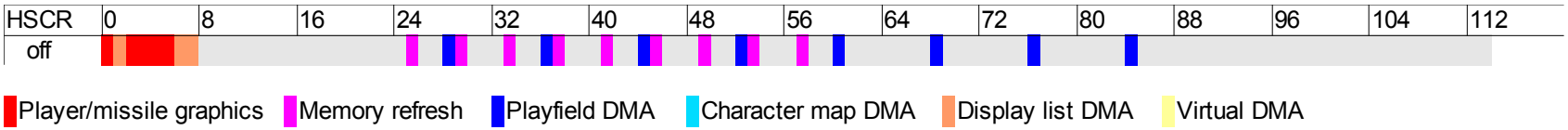
ANTIC modes 8 and 9, mode line, wide playfield



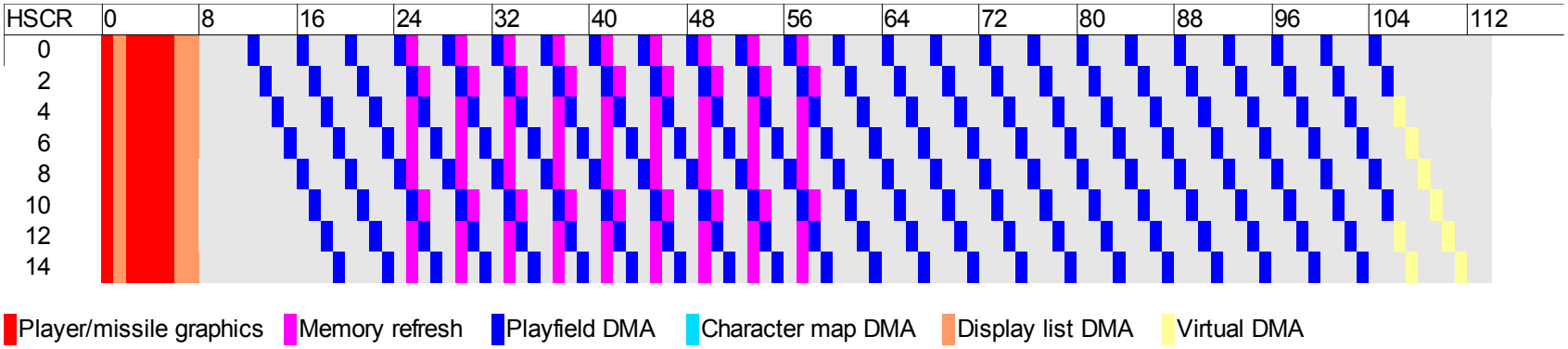
ANTIC modes 8 and 9, mode line, normal playfield



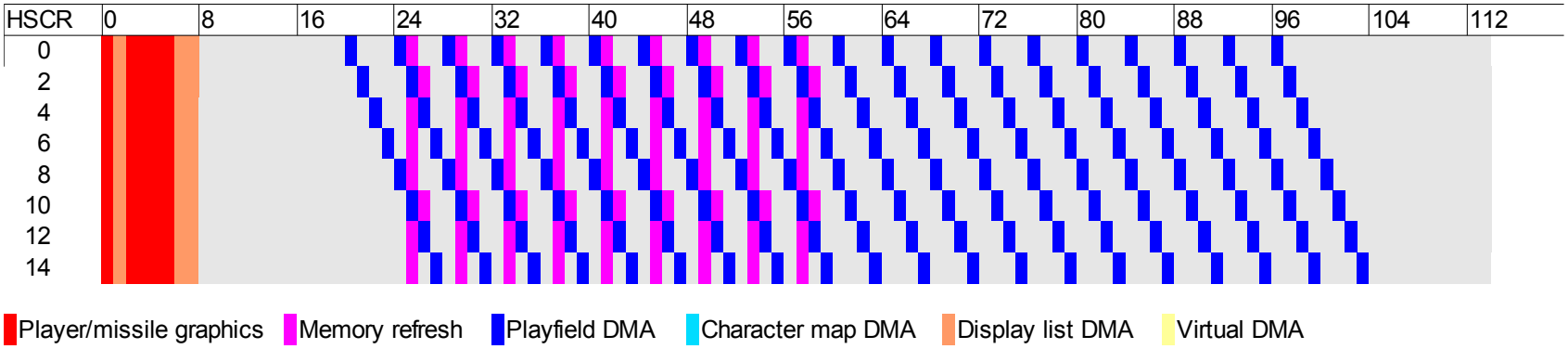
ANTIC modes 8 and 9, mode line, narrow playfield



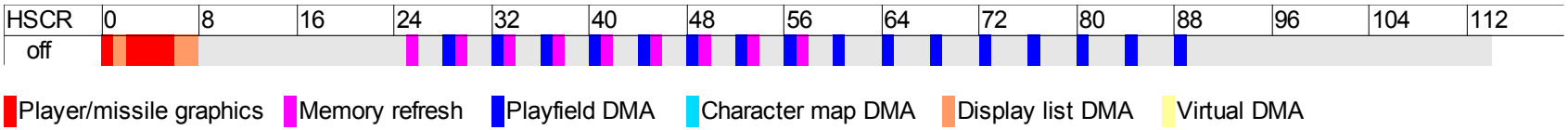
ANTIC modes A-C, mode line, wide playfield



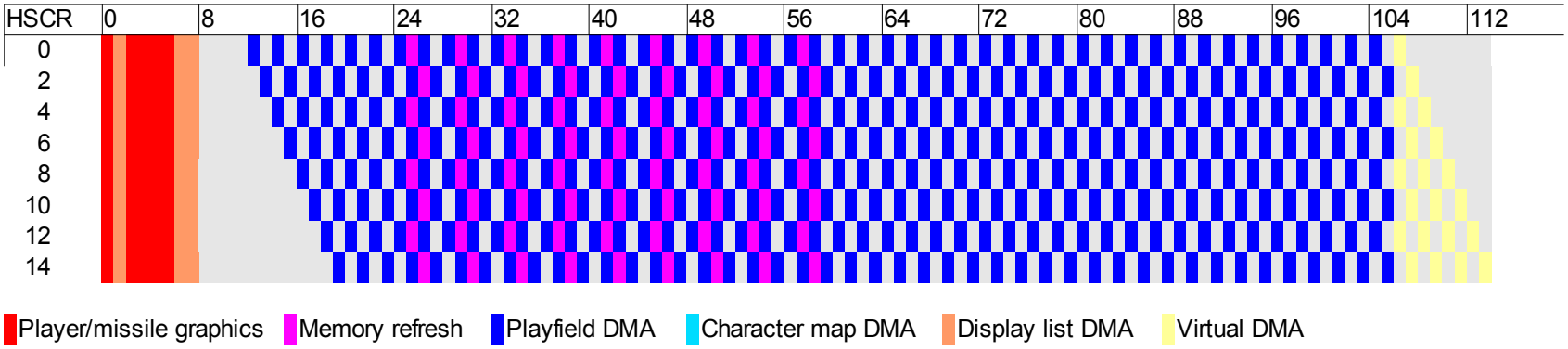
ANTIC modes A-C, mode line, normal playfield



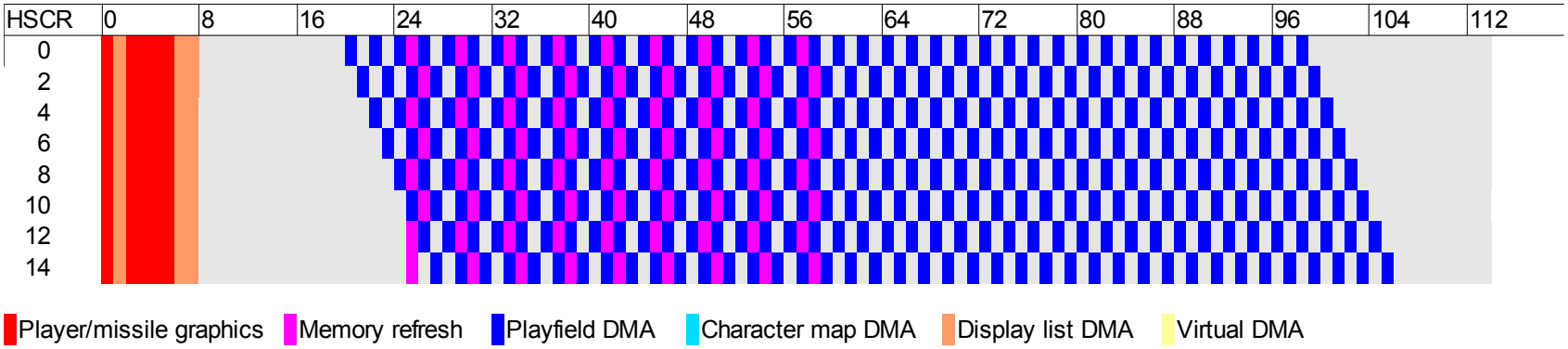
ANTIC modes A-C, mode line, narrow playfield



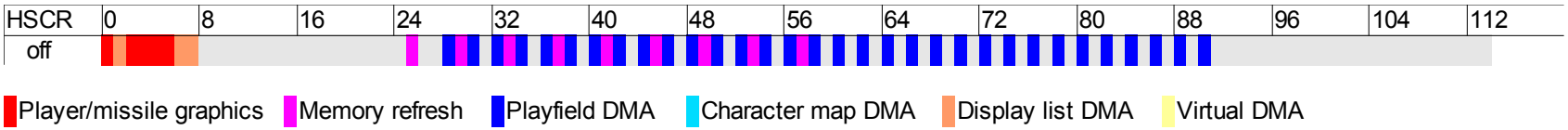
ANTIC modes D-F, mode line, wide playfield



ANTIC modes D-F, mode line, normal playfield



ANTIC modes D-F, mode line, narrow playfield



Event timing chart

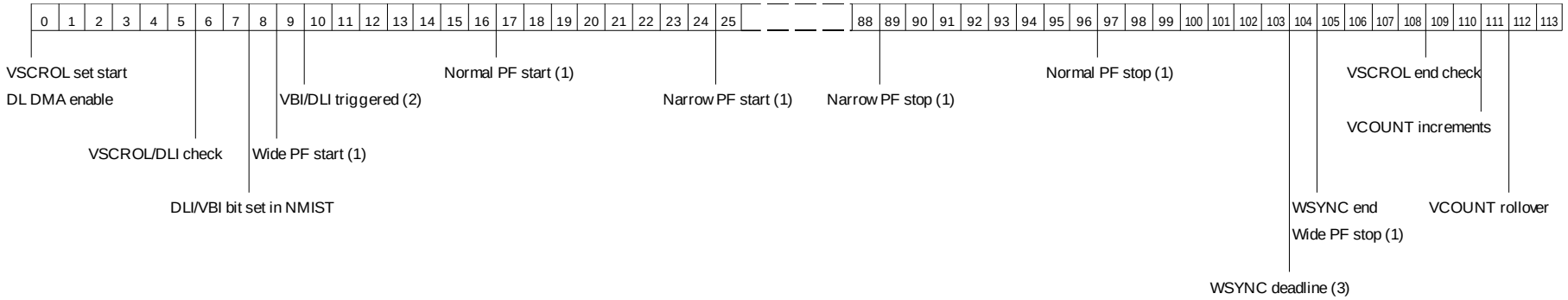


Figure 4: ANTIC event timing

The above figure shows the timing of various events within ANTIC and the available cycle times at which the CPU can read or write values in response. These are marked on machine cycle boundaries, so only writes before the boundary will affect the event and only reads after the boundary will reflect it. For instance, the narrow width playfield start boundary is between cycles 24 and 25, so a write to DMACTL to turn on the narrow playfield must occur on cycle 24 or earlier. Similarly, the VCOUNT increment on a scan line will only be reflected in reads on cycle 100 or later.

(1) PF start/stop events are delayed by one cycle for every two increase in HSCROL when horizontal scrolling.
(2) 7-cycle NMI sequence normally starts at first instruction boundary on cycle 10 or later, unless overlapping an earlier IRQ.
(3) If read/modify/write instruction on 6502 or 65C816 (emulation mode), both write cycles must occur before this deadline.

4.15 Cycle counting example

Let's assume that we want to schedule a series of palette color changes between lines of 40-column text (ANTIC mode 2). To do this, we use the following DLI routine:

```
PHA
TXA
PHA
LDX    NEWCL1
LDA    NEWCL2
STA    WSYNC
STX    COLPF1
STA    COLPF2
PLA
TAX
PLA
RTI
```

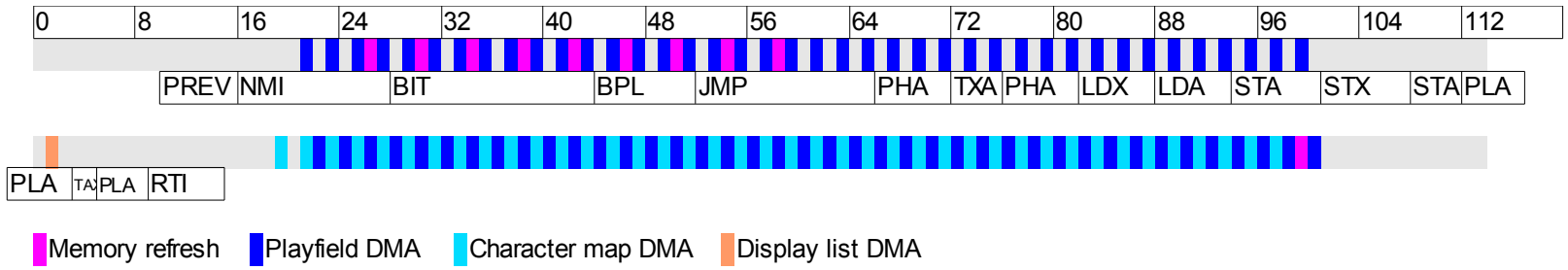


Figure 5: DMA and CPU timing for DLI handler.

Cycle counting breakdown

Figure 5 shows the DMA and instruction timing for the DLI handler. First, after receiving the NMI request at cycle 8 and acknowledging it at cycle 10, the 6502 has to finish the previous instruction. The worst case of six clocks is shown here. Afterward, it takes seven clocks for the 6502 to push PC and P onto the stack and to fetch the NMI vector. At this point playfield DMA starts, which slows down the CPU; the first instruction doesn't execute until cycle 28. From there, it takes 11 CPU cycles to execute the OS NMI handler, which actually takes 36 machine cycles with DMA contention, meaning that the user DLI handler isn't entered until cycle 66.

Once in the DLI handler, it takes 8 CPU cycles (16 machine cycles) to save X and A and 6 CPU cycles (12 machine cycles) to preload two colors. That's as much that can be done while still in the visible region, so on cycle 94, an STA WSYNC is executed. The first cycle of the next instruction is executed before the CPU is halted until cycle 105, after which X and A are pushed into the PF0 and PF1 color registers at cycles 107 and 111, respectively. Finally, the epilogue begins at cycle 112, where it takes 10 CPU cycles (11 machine cycles) to restore A and X and another six cycles to exit the DLI handler.

There are a few aspects to note about this DLI handler. First, it doesn't write NMIRES; that is generally unnecessary for DLIs. Second, the horizontal blank region before the line to be modified is critical timing-wise. In this case there would have been enough CPU time to STA WSYNC first and then both load and store the color values in HBLANK, but that's not always the case, especially with P/M DMA enabled or when the background color is involved. Second, the DLI handler consumes an entire scan line worth of CPU time despite only changing two registers and not setting up a subsequent DLI handler. In practice, this means that any large region that requires many per-scan-line register changes is better done with a kernel started by one DLI rather than with multiple smaller DLIs.

Examples

Zaxxon II

This game uses a display list interrupt (DLI) on a scan line that is highly contended, with a scrolled normal width playfield and P/M graphics active. As a result, the 6502 is unable to read NMIST until past the standard interrupt cycle on the next scan line, and the DLI bit must remain active for more than a full scan line for Zaxxon to work correctly.

Race in Space

Unusually, the interrupt flag is set on the wait for VBL instruction at the end of the display list for the title screen. The game relies on the high number of interrupts that this generates; failing to generate an interrupt per scan line results in the title screen scrolling very slowly or never completing.

Race in Space also uses player collisions against a hi-res (mode F) playfield.

Numen

A lot of tricks are used in this demo, but it almost immediately goes into the “GTIA 9++” mode where VSCROL is alternated to generate mode F with four scan lines per row and one-quarter the DMA overhead.

Bounty Bob Strikes Back!

This game loops on an alias of the VCOUNT register, $D47B, and jams on startup if address mirroring is not supported.

Chicken

The display list for *Chicken* contains a vertical scrolling region that ends on a blank mode line. The vertical scroll interaction causes this mode line to be variably extended beyond its usual one-scan-line height.

Tarzan of the Apes

The mid-screen DLI routine for the title screen of this game expects VCOUNT to roll over prior to P/M DMA at the start of the next scan line.

Atomix Plus!

There is a buggy loop in this game for copying memory below the kernel ROM ($D800-FFFF) that enables ANTIC interrupts before re-enabling the kernel ROM. It relies on a DLI or VBI never interrupting the following sequence:

```
LDA #$40
STA NMIEN
LDA #$01
STA PORTB
```

Pacem in Terris

One of the DLI handlers for the title screen attempts to change playfield width from narrow to normal by writing to DMACTL, but misses the deadline doing so. The result is that the scan line is blank and the “Quasimodo” bitmap is shifted one scan line lower than the display list would indicate.

4.16 Further reading

Consult [ATA82] for a overviews and register descriptions for ANTIC. Surprisingly, there is little, if any, additional information in the formerly confidential chip document [AHS99]. A bit more information can be found in [AHS00] , but the accuracy of the additional information appears questionable.

[CRA82] notes a number of nuances about programming ANTIC, most notably the tricky timing in display list interrupts. Note that there appear to be some slight timing discrepancies compared to the real Atari.

Chapter 5
POKEY

5.1 Addressing

POKEY occupies the $D2xx block of memory. Only the lowest four bits are significant, so any access of the form $D2xy accesses mirror x of register y. The canonical registers are at $D200-D20F.

One popular modification involves piggybacking a second POKEY onto the system and using address line A4 to select between them. In that case, the even mirrors select the main POKEY, and the odd mirrors select the secondary one. The secondary POKEY has less functionality available due to missing interrupt and I/O connections.

5.2 Initialization

POKEY does not have a RESET line and therefore powers up in indeterminate state. IRQEN must be reset prior to clearing the I bit on the CPU to avoid stray IRQs.

Bits 0 and 1 of SKCTL normally control the keyboard scan and debounce features. However, clearing both of those bits also activates another initialization function, which causes the 15KHz clock, 64KHz clock, serial port hardware, and polynomial noise generators to be reset.

The initialization function can be used to reset the 15KHz and 64KHz clocks to known offsets in the scan line. However, both clocks will have significant offsets from when initialization mode ends, due to the counters being reset to the middle of their cycles. Initialization mode must be asserted for at least seven cycles to ensure that both clocks are fully reset; afterward, the 15KHz and 64KHz clocks will first fire approximately 81 and 22 cycles from when initialization mode ends.

Setting the serial clock selection bits SKCTL (bits 3-5) to 0 resets the serial port circuitry. Therefore, SKCTL should be set to $00 to initialize all POKEY functions.

5.3 Sound generation

POKEY has four audio channels with individual timers and audio output circuitry. Each channel has an associated frequency register (AUDF1-4) and control register (AUDC1-4). In addition, there is a shared control register (AUDCTL) for common functions.

Countdown timers

Each channel has an 8-bit countdown timer associated with it to produce clocking pulses. The period for each timer is set by the AUDFx register, specifying a divisor from 1 ($00) to 256 ($FF). The countdown timer produces a pulse each time it underflows and resets, which can then be used to drive an interrupt, the serial port, or sound generation.

By default, timers use the default audio clock, which is selected by AUDCTL bit 0. Setting this bit to 0 selects the 64KHz clock, while setting it to 1 selects the 15KHz clock. It is not possible to use both the 15KHz and 64KHz clocks at the same time. In addition, timers 1 and 3 can be switched to the fast 1.79MHz clock through AUDCTL bits 6 and 5.

When a timer underflows, it takes three cycles to reload. This means that for a timer running at 1.79MHz, the actual period for a AUDF1/3 value of N is N+4 cycles. For timers using the 15KHz or 64KHz clock, the period is N+1 ticks, where each tick is 114 cycles with the 15KHz clock and 28 cycles with the 64KHz clock. Because the audio clock runs independently of the timers, the three cycle reload delay is absorbed in this case and does not affect the timer period.

Linked timers

Setting bit 4 of AUDCTL links timers 1 and 2 so that timer 2 is clocked using the output of timer 1, and similarly,

bit 3 links timers 3 and 4 together. This merges the pair of counters into a 16-bit counter. This is typically used with the 1.79MHz clock on the low timer in order to achieve higher precision, but linked timers can also be used with the 15KHz and 64KHz clocks. The high timer – timer 2 or 4 – is the one that has the desired period and is the one that should be enabled for audio, IRQs, or serial port clocking. Linking occurs prior to the audio circuitry and thus the waveform settings for the low channel have no effect on the clocking of the high channel.

When timers are linked, the delay for reloading the timer pair is increased because the low timer must underflow first before the high timer can underflow. This increases the reload delay to 6 cycles and therefore the period for a 1.79MHz paired counter is N+7 cycles instead of N+4 cycles. Since linking is the only way that the 1.79MHz clock can be used with the serial port, this effectively limits the serial port to a bit rate of 128 kilobaud and a byte rate of 12.8 kilobytes per second.

While linked timers are intended to be used as a single high-precision timer, both channels are still active. Normally, timers 1 or 3 should be muted in a linked scenario, but it is possible to use them and exploit the irregular timing of the low channel. For instance, when the 16-bit period is $0140, the low timer will underflow after 65 ticks first and then run a full 256 ticks before the counter pair resets. One use for this is as a one-shot timer for creating variable delays within a 256 tick period or less.

Waveform selection

Bits 5-7 of AUDCx control the waveform used by the audio circuitry for a channel. This allows each channel to produce a flat level (no output), a square wave, or a more complex wave driven by the polynomial noise generators.

Bit 5 selects either noise (0) or a square wave (1). When the square wave is enabled, each time the timer expires and the output circuitry is clocked, the output toggles, resulting in a square wave with a frequency half that of the timer. When noise is enabled, bit 6 selects either the 9/17-bit generator (0) or the 4-bit generator (1).

Bit 7 controls the sampling mode. If it is set, the timer output directly clocks the output waveform. If it is cleared, however, the 5-bit generator masks out some of the clock pulses, giving a rougher sound.

Due to the short periods of most of the pseudorandom noise generators, it is possible to have undesirable interactions between the period of the countdown timer and the period of the noise generator. For instance, a channel using the 64KHz clock and an AUDFx value of $CC has a period of 5740 clocks. When used with the 4-bit noise generator, five different sounds can result because the 4-bit generator has a period of 15 and the timer period is divisible by 5, meaning that only three bits of the pattern are used. Exactly which three are used depends on when the sound is started. In a more extreme case, $D1 would produce no noise at all, because the period is 5880 clocks, which is divisible by 15 – meaning that it will always sample the same bit from the noise pattern.

Volume control

Bits 0-3 of AUDCx control the volume level for a channel, from 0 (silent) to 15 (maximum volume). The volume level only matters if the channel output is currently a 1; if it is a 0, then there will be no output from the channel regardless of the volume level.

Volume output from POKEY is non-linear in that adding two channels of equal volume doesn't produce output with twice the amplitude, but somewhat less. Instead, two channels at volume 15 will only be about 50% louder as each one individually. This has the effect of compressing the output, amplifying quieter sounds and attenuating louder ones.

Volume-only mode

Bit 4 of AUDC1-4 activates volume-only mode for a channel. This causes the channel output to be forced to a 1, ignoring the output of the timer, noise generators, and high-pass logic, and only producing sound based on the volume set by bits 0-3 of AUDCx. This is often used for playback of digital sound effects at 4-bit/sample precision.

Note that because the volume-only mode is enforced after the high-pass logic, the normal inversion of channels 1 and 2 relative to 3 and 4 doesn't apply to this mode; volume-only channels will add in any combination.

High-pass filter

Channels 1 and 2 have a high-pass filter which is enabled by bits 2 and 1 of AUDCTL, respectively. The filter works by XORing the signal against a point-sampled version of itself, which crudely blocks lower frequencies. Channels 3 and 4 control the rate at which the flip-flop is updated and thus the sampling rate for the XOR source.

When the high-pass filter is disabled, the high-pass flip-flop is forced to a 1, but the XOR still takes place. This causes the signal from channels 1 and 2 to be inverted. Normally this isn't noticeable, but it can show up when two channels play synchronized sound. If channels 1 and 2 are set to the same frequency and to pure tone mode, they will add, but if the same is done with channels 1 and 3, they will cancel. This doesn't happen in volume-only mode, however, as the gates that force volume-only mode are after the high-pass circuitry and therefore volume-only channels always add in any combination.

Resetting the timers

Writing to the STIMER register causes all of the timers to reload and sets the output flip-flops. When high-pass filters are disabled, this turns off the output of channels 1 and 2 and turns on the output of channels 3 and 4. This is useful to synchronize the sound channels.

There is a four cycle delay from the time that STIMER is strobed to when the timers are reset. With timer 1 set to 1.79MHz, 8-bit mode and with an AUDF1 value of N, IRQST bit 1 is set N+8 cycles after STIMER is written. This holds even if timers 1+2 are linked, although the bit for timer 2 (bit 1) is set three cycles later.

STIMER has no effect on the phase offset of the 15KHz and 64KHz clocks. Regardless of when it is strobed, any timers that are using those clocks will still only decrement and underflow according to the timing of those clocks, and if such a timer hasn't decremented since the last time it was reset, there will be no effect on that timer. This can be exploited by using STIMER to reset 1.79MHz clocked timers without affecting the slowly clocked ones.

5.4 Serial port

The serial port is used to transfer data to and from the SIO bus. This allows for communication with disk drives, printers, cassette tape recorders, and other SIO-supporting peripherals.

Shift registers

The main programmatic interfaces to the serial port are the SERIN and SEROUT registers, which hold the last byte received or the next byte to transmit on the serial bus. POKEY automatically exchanges these registers with input and output shift registers and handles start and stop bits, requiring interaction with the CPU only on a byte basis and allowing the Atari to read and write back-to-back bytes on the serial bus.

Three IRQs notify the CPU when the serial port needs attention. The serial input data ready interrupt (IRQEN/IRQST bit 5) is asserted when a byte has been assembled and transferred to SERIN; the CPU can then store this byte while a new byte is shifting in. The serial output needed interrupt (bit 4) fires when SEROUT has been transferred to the output shift register and a new byte can be queued. Finally, the serial transmission complete interrupt (bit 3) is asserted when the last byte has finished shifting out and transmission is complete.

Framing errors

SKSTAT bit 7 reports if a framing error occurs on the serial input port. A framing error occurs when the stop bit (bit 9 after the start bit) is not a 1, indicating that the byte was not received correctly.

Overrun errors

SKSTAT bit 5 indicates whether an overrun has occurred.[18] An overrun occurs when a serial byte is not read before the next byte is received; when this occurs, the new byte replaces the previous byte and the previous byte is lost.

In order to acknowledge receipt of a byte from SERIN, the serial input interrupt (IRQST bit 5) must be cleared. Reading a data byte from SERIN by itself has no bearing on whether an overrun is detected, only the interrupt status. The interrupt should also be cleared before the start of a receive operation to clear any previously received stray data.

> **Warning**
>
> The design of the serial port makes it impossible to completely reliably detect overrun errors since the serial input ready IRQ must be temporarily disabled to acknowledge it, during which time an overrun can be missed.

Polled operation

It is possible to drive the serial port in polled mode by enabling serial interrupts on POKEY, disabling interrupts on the CPU, and then polling IRQST. This can be useful if the data rate is too high to use interrupts. The interrupt must both be enabled and masked since the interrupt status bit is required to detect the reception of a new data byte.

Direct input

Bit 4 of SKSTAT directly reads the state of the serial input port. This is used by the kernel to measure baud rate prior to reading a block from cassette tape, since the serial input shift register cannot be used until the baud rate has been set.

Clock selection

Bits 4-6 of SKCTL control the clocks used during serial port operation. These three bits affect a number of switches and gates and interact in complex manners. For instance, bit 4 generally enables asynchronous receive using timer 4, but it also sometimes changes the output clock as well. Each setting specifies a different combination of signals to use for both the input and output clocks, as well as whether to configure the bidirectional clock line as an input or an output. Here are all of the modes:[19]

18 Credit to HiassofT for noting that the SKSTAT reference on [ATA82] III.18 has D5 and D6 swapped.

19 [ATA82] II.27 has the official mode chart; see also unnumbered page with serial/audio diagram for exact switch and gate layout.

Setting	Input clock	Output clock	Bidirectional clock
000	External clock	External clock	Input
001	Channel 3+4 (async)	External clock	Input
010	Channel 4	Channel 4	Output channel 4
011	Channel 3+4 (async)	Channel 4 (async)	Input
100	External clock	Channel 4	Input
101	Channel 3+4 (async)	Channel 4 (async)	Input
110	Channel 4[20]	Channel 2	Output channel 4
111	Channel 3+4 (async)	Channel 2	Input

Table 6: Serial port timing modes

The modes for standard half-duplex SIO operation are 001 for reception and 010 for transmission. The external clock is not normally used; for instance, the 810 disk drive ignores the clock lines and uses timing loops for both transmission and reception.

Serial port clocks are produced by divide-by-two flip flops driven off of the counter outputs. They are not affected by any of the audio control bits in the AUDC1-4 registers. However, the clock select and linking bits in AUDCTL – bit 0 and bits 3-6 – do affect serial port operation since they affect the countdown timers themselves.

When using timer channels to clock the serial port, the timer frequency should be set to twice the baud rate.[21] Channels 3+4 should also be linked together and driven by the 1.79MHz clock for highest precision. For cassette operation at 600 baud, the divisor setting is $05CC; for disk operation at 19200 baud, it is $0028. Remember that there is a six cycle delay in reloading a 16-bit, 1.79MHz timer. Due to imprecision in the timer divisor at high frequencies, the actual transmission rate for the SIO bus is 19040 baud.

Serial clock reset

Setting bits 4-6 of SKCTL to 000, thus selecting an external clock for both input and output, also resets the serial input and output clock flip-flops to a known state. The serial output updates on the next output clock cycle, whereas the serial input updates after the next two input clock cycles.

Timer usage during serial port operation

The serial port and audio circuitry both share the countdown timers and thus timers used for controlling the serial port are not available for audio generation. Usually channels 3 and 4 are used for clock generation; when using two-tone mode for recording to cassette, channels 1 and 2 are also occupied for FSK output.

Note that while the serial port uses the output of the counters, the audio circuitry is still active. This means that the occupied channels should normally be silenced by setting their volume to zero and the corresponding interrupt enables in IRQEN should also be disabled. However, the audio channels can be enabled for effect. The SIO library in the kernel ROM normally enables audio from channel 4 during transfers, producing the characteristic beep-beep-beep of Atari disk loads.

The asynchronous receive mode tone

Asynchronous receive mode can be enabled by setting bit 4 of SKCTL ($D20F). Enabling asynchronous mode

20 [ATA82] II.27 and [AHS03] p.21 appear to have the same error of showing channel 2 as the input clock for the 110 setting. This is not possible, as only channel 4 or the bidirectional clock line can be routed to the serial input shift register. The description text correctly indicates channel 4.

21 [ATA82] II.25. The output clock toggles level each time the timer expires, so the frequency of the clock is half the frequency of the timer.

causes timer counters 3 and 4 to be reset either whenever the serial logic is waiting for a start bit or when a zero is received, resynchronizing the receive clock to the incoming bit stream. With the standard SIO receive routine, this disruption introduces an additional audible tone into the channel 4 output caused by the output flipping every byte. At 19200 baud, this produces a 960Hz tone during the read of each disk sector.

A side effect of this behavior is that channels 3 and 4 can be locked in reset state if asynchronous receive mode is enabled. Therefore, bit 4 of SKCTL should be cleared before attempting to use those channels for audio.[22]

Shift timing

As stated earlier, the serial port logic shifts bits in or out at half the rate of the controlling timer. The serial port shift registers are also only loaded or unloaded on this clock, which means that the interrupt bit latches are only activated on clock edges. This leads to unintuitive behavior when SEROUT is loaded for the first byte of an output stream, as the serial output shift register is only loaded at the next clock edge. First, SEROUT cannot be written twice back-to-back at the start because of the delay – it is necessary to wait for the serial output ready IRQ (bit 4). Second, if the output shift register is initially idle, the serial output complete IRQ (bit 3) will not clear until the load occurs. This means that the complete IRQ should not be enabled or polled until the output register is known to be shifting, or else a transmit routine may fail to wait for the last byte to complete and truncate the transmission.

Two-tone mode

Two-tone mode is enabled by setting bit 3 of SKCTL and replaces the normal 1 and 0 bits output to the SIO bus with tones clocked by timers 1 and 2, respectively. This is used when writing data to tape, where the timers are programmed in 64KHz mode with divisors $05 (5327Hz) and $07 (3995Hz) to do FSK encoding. The timer output is tapped prior to the output circuitry and so the serial output is unaffected by either AUDC1 or AUDC2; the serial port has its own divide-by-two circuit, independent of the audio dividers.

The switching between timer 1 and 2 based on serial data is done only in the serial logic and is therefore inaudible; both audio channels will play during transmission if their control registers are set appropriately. There is still an audible effect from two-tone mode, however, due to resynchronization between the timers: whenever a timer pulse toggles the serial output, both timers are reset. The purpose of this is to align the timer phases to avoid runt pulses in the output. This does not send a pulse to the audio logic, so the channel whose timer did not underflow can either be silenced or lowered in pitch.

In order for two-tone mode to function as intended, timer 2 must have a lower frequency (longer period) than timer 1.[23] The reason is that while timer 1 pulses only toggle the serial output when the serial bit is a 1, timer 2 pulses always toggle the output regardless.[24] When timer 2 has a longer period, this works because on a 1 bit the pulses from timer 1 will always preempt timer 2 before it can underflow and fire. If timer 2 has a shorter period, however, it will affect timer 1 regardless of the bit being output. With the standard 5327Hz/3995Hz tones, this means that a 1 bit results in timer 1 playing 5327Hz and a 0 bit results in timers 1 and 2 playing both tones.

The force break bit (SKCTL bit 7) can be used to enforce a known 0 output so that timer 2 is always used to reset timer 1.

5.5 Clock generation

There are three clocks that can be used to drive the counters:

- Channels 1 and 3 can use 1.78979MHz (NTSC) if bits 6 and 5 of AUDCTL ($D208) are set, respectively.

22 [ATA82] II.26 states a slightly different rule, that the start bit resets channels 3+4. This must be interpreted as waiting for the start bit and not the actual reception of the start bit in order to explain why those channels become silent when asynchronous mode is enabled even when no serial data is being received.

23 [ATA82] II.26

24 This means that the audio and serial port block diagram in [ATA82] is incorrect; it should show ((chan 1 AND serial) OR chan2) instead of a switch between chan 1 and chan 2 leading into the div-by-2 block in the two tones path.

- Otherwise, channels use a 63.9210KHz clock by default. This is exactly 1/28th of the main clock.
- If bit 0 of AUDCTL ($D208) is set, then channels use a 15.7KHz clock instead. This is exactly 1/114th of the main clock.

Both the 64KHz and 15KHz clocks are generated by polynomial counters internal to POKEY, driven off the main clock, and have no guaranteed phase relation to other clocks in the system. In particular, the 15.7KHz clock signal is labeled as HSYNC in the schematic, presumably because it counts at the same rate as ANTIC's scan lines, but there is no connection to synchronize the two. The phase relationship between the 15.7KHz clock and horizontal scan timing is determined by when initialization mode is ended.

5.6 Pseudo-random number generators

POKEY contains three pseudo-random number generators, all composed of maximal-length linear feedback shift registers (LFSRs, or polynomial counters) that run at 1.79MHz. These are used both for generating audio noise as well as random numbers for the CPU.

4-bit and 5-bit noise generators

The 4- and 5-bit generators within POKEY are linear feedback shift registers with the polynomials $1+x^3+x^4$ and $1+x^3+x^5$, respectively. They are only used for noise output and are not accessible to the CPU.

The 4-bit generator has the pattern: 000111011001010.

The 5-bit generator has the pattern: 10000011100100010101111101101001.

9/17-bit noise generator

POKEY also has a third shift register which is either 9 or 17 bits long, depending on bit 7 of AUDCTL. When in 9-bit (short) mode, the polynomial is $1+x^4+x^9$; when in 17-bit (long) mode, an additional eight bits are added to the shift register and the polynomial is $1+x^{12}+x^{17}$. Eight bits of the shift register are visible to the CPU via RANDOM ($D20A); this is most commonly used for random numbers, but it can also be used to test cycle counting hypotheses. The CPU sees the top bits of the shift register such that bit 7 is closer to where bits are being shifted in and bit 0 is where bits are being shifted out.

If the main LFSR is in 9-bit mode and samples are taken from RANDOM ($D20A) every scan line by STA WSYNC + LDA RANDOM, part of the sequence is as follows: 00 DF EE 16 B9.

Audio channel noise delays

The outputs of the noise generators are delayed to each audio channel by one clock apart to prevent the channels from receiving the exact same noise. A given pattern bit arrives at channels 1, 2, 3, and then 4, in that order.

Initialization behavior

The polynomial counters must be reset on startup in case they power up in a lock-up state, of which there is always exactly one state: either all 1s or 0s, depending on the exact formulation. Initialization mode forces bits into the register until it is reset to the opposite of the lock-up state so that it is guaranteed to count normally when the initialization state ends. Initialization mode need not be asserted for a long period of time for the polynomial counters to work, as a single bit of the right polarity is enough to prevent lock-up.

Once the 17-bit polynomial counter is fully initialized, RANDOM reads a constant $FF until initialization mode ends.

5.7 Interrupts

POKEY can issue interrupts to notify the CPU of events such as timer expiration and changes in serial port state. All interrupts from POKEY are IRQs.

Interrupt enable/status

The IRQEN register selectively enables or disables interrupts; a 1 bit enables an interrupt. When an interrupt is enabled and becomes active, the corresponding bit in IRQST is set to a 1 and the IRQ line to the 6502 CPU is asserted. POKEY will keep the IRQ line asserted until all pending interrupts are cleared by resetting the corresponding IRQEN bit; this ensures that the CPU will continue to execute its IRQ routine until all interrupts are serviced, even if an NMI intervenes temporarily.

Note that the serial transmission complete interrupt (bit 3) is special – it is not latched, so it is simply active whenever the serial output shift register is idle and automatically deasserts when a new byte begins to shift out. The interrupt status bit and corresponding interrupt will be set in that case even if bit 3 of IRQEN is cleared. This can be useful to assert an IRQ on the CPU on demand.

Interrupt timing

There is a minimum 2-3 unhalted cycle delay from the time that an interrupt is signaled in the IRQST register to the first time that the 6502 will begin the seven cycle interrupt acknowledge sequence. This delay is extended if the 6502 is in the middle of executing an instruction when the three cycles have elapsed or if ANTIC halts the CPU for DMA.

> **Machine-specific Behavior Warning**
>
> The IRQ delay can vary between systems or based on temperature. A 3 cycle delay appears to be more common, but some systems can consistently show 2 cycles.[25]

Enable/disable timing

A write to IRQEN that enables an interrupt must occur at least four cycles before the interrupt source activates, or else the interrupt will not be latched in IRQST and an IRQ will not occur. For instance, if a timer is configured such that the IRQ handler would trigger on cycle 16 of a scan line, the latest that the write to IRQEN can occur is cycle 12.

For disabling an interrupt, the write to IRQEN must occur at least two cycles in advance. In other words, for an IRQ on cycle 16, the write must occur on cycle 14 or earlier to block the interrupt in time. This means that there is a one-cycle window where an IRQ can still occur after its source has been shut off via IRQEN.

> **Warning**
>
> The fact that a previously signaled IRQ can happen immediately after a write to IRQEN means that caution must be taken when attempting to shut off POKEY interrupts. Simply attempting to write $00 to IRQEN can fail if an IRQ occurs afterward and re-enables interrupts, leading to a rare crash. To be safe, mask interrupts with an SEI instruction before clearing IRQEN; this ensures that the 6502 cannot service the interrupt before noticing that the IRQ line has been negated.

Initial interrupt state

Because POKEY has no reset pin, IRQEN state is indeterminate on start-up. IRQEN should be cleared before the 6502 I flag is cleared.

25 Credit goes to HiassofT for discovering this innovative method of measuring temperature with an Atari computer.

5.8 Keyboard scan

The keyboard is automatically scanned by POKEY, which detects any pressed keys and notifies the CPU of new key presses.

Key press detection

When a key is pressed, the key code is placed into bits 0-5 of the KBCODE [D209] register. Bits 6 and 7 are also set to indicate the state of the shift and control keys, respectively. The keyboard interrupt (IRQST/EN bit 6) is also activated if it is enabled. At the same time, SKSTAT bit 2 is set to indicate that a key is depressed and stays asserted as long as the key is held down, allowing software to implement key repetition.

If the same key is pressed multiple times in a row, KBCODE does not change. Therefore, the only way to detect manually repeated key presses is through the keyboard IRQ or by polling SKSTAT. Key releases never change KBCODE or interrupt status and can only be detected by polling.

Key codes

The key codes that appear in KBCODE are scan codes, which are different than ATASCII or INTERNAL codes for characters. Table 7 lists the base key codes returned for each key, before the Shift and Ctrl bits are set.

	+0/8	**+1/9**	**+2/A**	**+3/B**	**+4/C**	**+5/D**	**+6/E**	**+7/F**
$00	L	J	; :	F1	F2	K	+ \	* ^
$08	O		P	U	Enter	I	- _	= |
$10	V	Help	C	F3	F4	B	X	Z
$18	4 $		3 #	6 &	Esc	5 %	2 "	1 !
$20	, [	Space	.]	N		M	/ ?	Invert
$28	R		E	Y	Tab	T	W	Q
$30	9 (		0)	7 \	Bksp	8 @	<	>
$38	F	H	D		Caps	D	S	A

Table 7: Key codes

Keyboard overruns

If a new key is pressed and detected while the keyboard IRQ is still active (IRQST bit 6), a keyboard overrun is signaled by clearing SKSTAT bit 6, and the new key replaces the old one. The overrun condition is cleared by writing to SKRES.

Scan timing

The keyboard scan is triggered by the 15KHz clock. This means that keyboard IRQs occur relative to when the 15KHz clock is initialized. This typically means that the keyboard IRQ never hits the magic cycle on a scan line that can block NMIs, but just about every key can hit that cycle if POKEY is initialized at just the wrong offset. This happens if initialization mode is cleared at around cycle 32 on a scan line. The timing will vary somewhat due to variance in when the 6502 is able to acknowledge the interrupt.

Scan algorithm

The keyboard scanning hardware consists of a 6-bit counter, a 6-bit latched compare register, and a state machine with four states. One key out of 64 total is checked per cycle at 15KHz, so a full scan takes 4ms. The state hardware functions as follows[26]:

- **State 0**:
  - If a key is down, latch the counter in the compare register and go to state 1.
- **State 1**:
  - If the counter matches the compare register, and the current key is not down, go to state 0.
  - If the counter matches the compare register, and the current key is down, assert the keyboard IRQ, clear bit 2 of SKSTAT, copy the counter value into KBCODE, and go to state 3.
  - If the counter does not match the compare register, and the current key is down, go to state 0.
- **State 3**:
  - If the counter matches the compare register, and the current key is not down, go to state 2.
- **State 2**:
  - If the counter matches the compare register, and the current key is not down, set bit 2 of SKSTAT and go to state 0.

This flow assumes that keyboard debounce (SKCTL bit 0) is enabled. If debounce is disabled, then comparisons against the compare register always pass.

The design of the keyboard state machine limits the maximum typing rate to approximately 60 characters per second, since key presses can only be registered once every four full keyboard scans (256 horizontal blanks).

Keyboard scan enable

Bit 1 of SKCTL enables keyboard scanning. If it is disabled, the state machine is forced to state 0 and the counter is held in reset state. KBCODE and any previously signaled keyboard IRQs are unaffected.

Keyboard debounce

Bit 0 of SKCTL controls the *debounce* function. When enabled, a key must be detected as pressed in two consecutive scan cycles before a key press is registered, and the key must be released for two consecutive scan cycles before the key is considered released. This is intended to filter out noise from mechanical switches, which produce noise output when pressed or released.

Unfortunately, this function is poorly named and has several side effects besides debouncing. When enabled, the keyboard will never register a key press when two keys are pressed simultaneously. When disabled, the keyboard is basically non-functional, as the keyboard state machine checks consecutive keys rather than the same key in consecutive cycles. In this mode, a key press will only register if two consecutive keys are held down, and even then the keyboard logic will be unable to detect held keys, reporting a rapid series of key presses instead. Debounce should therefore be enabled for normal keyboard operation.

Note that the 5200 keyboard is the opposite: it requires debounce to be disabled to function. See chapter 12 for details.

26 Flowchart versions of the keyboard state machine can also be found in [AHS03] and [AHS03a]. They do not, however, indicate the connection to SKSTAT.

Key conflicts

While the POKEY hardware views the keyboard as a linear set of 64 keys, it is actually physically arranged as a 2D matrix where the high three bits of the key scan code control the output lines and low three bits control the input lines, and a key is detected when it connects an output line to an input line. Because there are no diodes on the keys, pressing three or more keys can result in additional phantom keys appearing in the matrix. Ordinarily this isn't a problem, because the debounce logic prevents any keys from being registered when more than one key is down.

Where this causes a problem is when two or more of the Control, Shift, or Break keys are pressed in conjunction with another key. The Control key shares a control line with keys that have a $0x base scan code, Shift with $1x, and Break with $3x. Usually, pressing one of these keys at the same time as a regular key is OK because they have a dedicated input line. Pressing two of them, however, will cause phantom keys to appear on the regular key matrix. The most noticeable impact of this is that none of the Control+Shift+key combinations for scan codes $C0-C7 or $D0-D7 can be detected.

Note that this problem is caused by the keyboard matrix hooked up to POKEY. On the 5200, the upper trigger is fully independent from the keypad and causes no such conflicts.

Auto-repeat

There is no auto-repeat hardware in POKEY. Keyboard auto-repeat must be implemented in software.

5.9 Examples

Atari OS, up through XL/XE OS ver. 2

Most versions of the Atari OS have a race condition in the SIO first byte transmit routine where a byte is written to SEROUT before the CHKSUM variable is initialized, while IRQs are unmasked. The serial input ready IRQ, which fires one serial tick layer, can strike in between the writes to SEROUT and CHKSUM, updating CHKSUM with the second byte before it is initialized. The chances of this are greatly raised by the VBI being enabled, which can also strike in between and then extend the window for the IRQ to ~130 cycles.

The result of this race is a blown checksum calculation. A disk drive will send back a NAK in response, but due to another bug in SIO, the result is a long timeout delay before the command is retried. It was fixed in later versions by swapping the order so that CHKSUM is written first.

Ray of Hope, Numen

Both of these demos use channels 3+4 in 16-bit mode at 1.79MHz with the 4-bit polynomial noise generator selected. The channels are set to a high frequency and the demos rely on the pattern of the noise generator to alias the frequency down to a lower range. The cycle period is therefore critical for the high notes to sound correctly instead of squeaking.

SpartaDOS X

SDX uses its own SIO routines for disk access that use polling rather than interrupts, by disabling interrupts on the CPU and waiting for bits in IRQST to change state.

5.10 Further reading

Read the Hardware Manual [ATA82] or the POKEY datasheet [AHS03] for theory and register-level specifications for POKEY. The Hardware Manual is especially useful here as it has detailed descriptions of the serial port and audio paths that are undocumented elsewhere.

Chapter 6
CTIA/GTIA

6.1 Color encoding

Color registers

For the most part, colors are encoded in GTIA through a palette of color registers, where displayed data refers to a color register and that register provides the actual color used. Changing the color register changes the color of all objects using that color register.

There are nine write-only color registers on the GTIA. COLBK is the background/border color register, COLPF0-3 are the playfield color registers, and COLPM0-3 are the player/missile color registers.

Color encoding

The high four bits of each color register encodes the hue, with 0 being a special value indicating no color (grayscale). Bits 1-3 encode the luminance (brightness) of the color, with 000 being the darkest and 111 being the brightest. Note that the luminance does not affect the saturation of the color, so a luminance of 0 does not mean black if hue is non-zero. The two fields together allow for 128 distinct colors.

Bit 0 of any data written to a color register is ignored. Although the GTIA can display 256 colors, this is only possible through the special 16 luminance mode and not through the color registers. The lowest luminance bit is always 0 for any output from a color register.

A word on colors

The actual colors produced by GTIA differ for each computer, depending on the setting of a tuning pot inside the computer and also the display monitor hooked up to it. This has led to a lot of disagreement about what colors result from each hue value. Even official Atari documentation differs. For instance, the Atari BASIC Reference Manual and the Hardware Manual specify that hues 1 and 15 should have different colors, whereas the 400/800 Service Manual advises adjusting the SALT color bar test pattern so that they have the same color. As such, **there is no single authoritative, official answer on what colors each hue value should provide**. This must be kept in mind when choosing color values.

Another important issue is that the versions of the GTIA produced for the three main TV encoding standards – NTSC, PAL, and SECAM – all differ in the way they encode color.

NTSC color encoding

An NTSC GTIA produces color by phase shifting a square wave at the same frequency as the NTSC color subcarrier. This generates different, evenly-spaced hues. Because the strength of the color signal is independent of the brightness, colors with low brightness are much more saturated than ones with higher brightness. Hue value 0 does not produce any color signal and therefore produces pure grays.

Hue 1 is the same phase as the color subcarrier, as it is actually used to generate the color burst that synchronizes the TV's color circuits. It produces a light yellow-orange color, sometimes called “gold.” Each subsequent hue adds an additional delay of around 24°, adjustable by a trim pot in the hardware. Ascending hues, and therefore increasing delays, produce colors in the orange, red, purple, blue, cyan, green, lime, and finally light yellow-orange again.

The delay between the hue phases is adjustable by a trimpot on the motherboard. This affects each delay stage and therefore has greater effect on higher hue numbers. The last hue, hue 15, varies the most as it is at the end of all delay stages and therefore has the most sensitivity to the color adjustment. Depending on the adjustment, its output can range from green, to yellow, to even orange if it wraps around past hue 1. In contrast, hue 1 varies only due to the display, and low-numbered hues have less variation between systems.

PAL color encoding

PAL encodes color differently than NTSC, and thus the PAL GTIA uses a different strategy to generate colors. The main issue is that one of the color subcarrier axes reverses phases on every scan line, so different phases are required to produce the same color. Like the NTSC GTIA, the PAL GTIA uses a delay line to produce different phases, but different phases are used for even and odd scan lines, and the spacing between the hues is also not even.

The phases used by the PAL GTIA for the various colors are as follows, in terms of delays (angles are ideal given a 22.5° delay):

Hue	Even lines	Odd lines	Ideal UV angle
1	1	5	135.0°
2	0	6	112.5°
3	7 (inverted)	7	90.0°
4	6 (inverted)	0 (inverted)	67.5°
5	5 (inverted)	1 (inverted)	45.0°
6	4 (inverted)	2 (inverted)	22.5°
7	2 (inverted)	4 (inverted)	337.5°
8	1 (inverted)	5 (inverted)	315.0°
9	0 (inverted)	6 (inverted)	292.5°
10	7	7 (inverted)	270.0°
11	5	1	225.0°
12	4	2	202.5°
13	3	3	180.0°
14	2	4	157.5°
15	1	5	135.0°

Table 8: PAL GTIA color encodings

Hue 1 is used for the color burst, which uses an angle of 135° and 225° on alternating lines, the latter of which is converted back to 135° in UV space by the alternating line inversion. The reversal of the color subcarrier direction between scan lines means that colors can display different hues between even and off scan lines depending on the color adjustment.

The greater complexity of the encoding scheme means that encoded colors from a PAL system have less variance than an NTSC system, and the “correct” color adjustment for PAL is more apparent. Hues 1 and 15 are always the same, for instance, because they are hardcoded to the same delays. The offset due to inversion on hues 3-10 is always 180° regardless of the adjustment. Finally, while the stable reference color on NTSC is hue 1, on PAL it is hue 13.

Regarding the actual colors produced, the U-V color encoding space used by PAL is flipped and rotated 33° from the I-Q color encoding space used by NTSC. The hue 1 color burst emitted by NTSC systems lies at 180° in the U-V coordinate space. While NTSC systems nominally have their colors spaced by 23-26°, in the PAL encoding they are spaced by uneven multiples of 22.5°, leading to wider gaps between hues 6 and 7 and hues 10 and 11.

PAL color blending

To combat hue shifting problems that occur with NTSC, PAL reverses the phase direction of the color subcarrier on alternating scanlines. This has the effect of reversing the direction of phase errors as well. For instance, if a signal transmission issue caused color signal phase to advance on each scan line between the encoder and the decoder, this would result in alternating increasing and decreasing angles in U-V space. Decoders can take advantage of this by combining color from adjacent scan lines, canceling the phase error at the cost of decreased saturation. A common way is to average in the color from the previous scan line via a delay line.

This effect can be used to blend colors between scan lines. Alternating mode 9 and 11 lines, for instance, will mix the gray level from the mode 9 lines with the color from the mode 11 lines, producing a more pseudo-256 color mode. Note that the blending effect only pertains to chroma and not luma.

6.2 Player/missile graphics

GTIA supports display of eight sprites on top of the playfield. These sprites can have distinct colors and can be moved horizontally much more quickly than the playfield for fast action. Four of the sprites are 8-bit wide players and four are two-bit wide missiles. All sprites are the height of the screen and can be as tall as desired. It is also possible to reposition sprites horizontally in the middle of the screen in order to increase the number of visible objects on screen.

Player/missile colors

Four color registers are reserved for player/missile graphics, COLPM0-3. Each player shares its color with the missile of the same number.

Player/missile graphics DMA

The default method for GTIA to receive player/missile graphics data is for ANTIC DMA to read it on a scan line basis, thus relieving the CPU of the burden of spoon-feeding graphics data. In order for this to happen, either bits 2 or 3 of DMACTL in ANTIC must be set to enable DMA, and the corresponding bits 0 and 1 of GRACTL must be set in GTIA to receive data. The graphics data registers GRAFP0-P3 and GRAFM are then accordingly loaded automatically at the beginning of each scan line.

If player or missile DMA is only set in GRACTL and not in DMACTL, then two odd effects can occur. First, if only missile DMA is enabled on ANTIC, but player DMA is enabled in GTIA, then the players will be loaded with whatever bytes are active on the bus while the CPU is executing during cycles 2-5 of the scan line. Second, if P/M DMA is entirely disabled on ANTIC, it is possible for GTIA to mistake a display list fetch for the missile fetch, because the first halted cycle within horizontal blank is considered to be the missile fetch. This causes GTIA to read the display list instruction as missile data and to load players at cycles 3-7 instead of 2-5.

When P/M graphics DMA is stopped on the GTIA side, the graphics data registers retain the last value loaded into them. This results in full-height stripes on screen unless the objects are subsequently repositioned or have their data registers cleared.

Graphic data registers

The CPU can also load directly into the graphics data registers for players and missiles by writing to GRAFP0-3 and GRAFM directly. This allows the CPU to directly control P/M graphics data when ANTIC DMA is inconvenient. It also allows vertical bar patterns to be displayed without requiring data in memory, since the graphics latches can be loaded once and GTIA will reuse the same pattern for each scan line.

Vertical delay

Vertical delay is used to move a two-line resolution sprite with scan line resolution. Unlike the Atari 2600's TIA, the GTIA does not have a true vertical delay function with a delayed graphics latch. Instead, the “vertical delay”

function works by masking DMA fetches. Setting the bit for a sprite in the VDELAY register causes GTIA to load DMA data for that sprite only on odd scan lines. In two-line resolution mode, when ANTIC repeats the same data on pairs of scan lines, this effectively moves the sprite image down by one scan line. In one-line resolution mode, this effectively reduces the sprite to two-line resolution.

VDELAY has no effect on writes from the CPU to GRAFP0-3 or GRAFM.

Player/missile positioning

The eight P/M objects are positioned along their left side via registers HPOSP0-HPOSP3 [D400-D403] and HPOSM0-HPOSM3 [D404-D407]. Position registers have color clock resolution. A player or missile begins shifting its output to the video display when the horizontal position counter matches the position register; this happens even if the object is positioned in the horizontal blank region (pos < $22), as long as part of it is in the visible region.

The center of the playfield is at the pixel boundary between $7F and $80. This means that the narrow playfield spans $40-$BF, the normal playfield $30-$CF, and the wide playfield $2C-$DD (visible portion of $20-$DF).

Size control

Each of the players and missiles can be set to one of three widths, with each bit displaying as one color clock (single width), two color clocks (double width), or four color clocks (quadruple width). Player widths are set by SIZEP0-SIZEP3; missile widths are set by SIZEM. Objects are always positioned from their left edge, so increasing a object's width causes it to expand to the right.

Shift triggering and timing

An object's image is produced by a shift register that gradually shifts out bits to the left. The timing of this shifter is controlled by a horizontal position comparator and a state machine controlled by the size setting.

A player or missile's shift register is loaded and begins shifting when the horizontal position of the object matches the horizontal position counter. This is checked every color cycle, so changing the position in the middle of the scan line can result in missing or duplicated object images. Moving it to the left of the current position prevents the object from triggering, and moving it to the right sets it up to trigger at the new position. Repeatedly moving the object to the right will cause it to appear multiple times. Because only the trigger point at the left side of the object matters, changing the position in the middle of the object's image has no effect and the object will continue to shift out at the same position.

The player/missile shift registers are constantly running, even across horizontal and vertical blank. This means that unlike with the 2600's TIA, positioning a player partially off-screen horizontally will show a partial object within the display region and not wrap the image within it. It is possible, however, for overlap and lockup effects to be carried over from vertical blank into the display of the next frame.

Overlapping object images

When the horizontal comparator matches, the shift register is reloaded with the contents of the graphics data register. This is done by ORing the latch data into the shift register. Ordinarily the shift register will have long emptied and therefore the shift register contents afterward will be that of the data register. However, if the image has not yet completed shifted out, some of the old bits from the previous image will still be in the register and combined with the new image.

Shift state machine

The timing of the shift register is controlled by a two-bit state machine whose operation is directed by the object's size setting. This state machine effectively counts off the color clocks for each bit in the sprite image, starting at %00 and going up to %11 for a quadruple width register. A shift register occurs each time the state machine

Original size	New size	Pixels before size change				Pixels after size change			
1x	2x	00	00	00	00	01	00	01	00
1x	4x	00	00	00	00	01	10	11	00
2x	1x	01	00	01	00	00	00	00	00
		00	01	00	01	00	00	00	00
2x	4x	01	00	01	00	01	10	11	00
		00	01	00	01	10	11	00	01
4x	1x	01	10	11	00	00	00	00	00
		10	11	00	01	00	00	00	00
		11	00	01	10	00	00	00	00
		00	01	10	11	00	00	00	00
4x	2x	01	10	11	00	01	00	01	00
		10	11	00	01	00	01	00	01
		11	00	01	10	01	00	01	00
		00	01	10	11	00	01	00	01
2x	1x*	01	00	01	10	10	10	10	10
		00	01	00	00	00	00	00	00
4x	1x*	01	10	11	00	00	00	00	00
		10	11	00	01	10	10	10	10
		11	00	01	10	10	10	10	10
		00	01	10	11	00	00	00	00

Table 9: Results of various size changes in the middle of a player image

transitions to the %00 state, which is forced whenever the shift register is reloaded. The operation of this state machine can be expressed simply:

state' = (state + 1) AND size

Thus, for normal width (%00) the shifter stays in %00 state and shifts out at a rate of one color clock per bit, whereas with quadruple width (%11) the shifter counts from %00 to %11 and shifts at out four color clocks per bit.

Mid-image size changes

Changing the size of an object causes its shift register to immediately begin shifting with the new width, but using the existing shift state. For the most part, this causes the shift register to finish shifting out its current pixel at the new width, but this leads to some strange patterns when switching to and from double width. Table 9 shows the effects of various size changes.

Shift register lockup anomaly

The size code %10 produces a normal width sprite similarly to the %00 code. However, the state machine acts slightly differently than the %10 mode in that it has a lockup state not present with %00. Specifically, switching an

object to the %10 mode when it is in double or quadruple width and in the %01 or %10 state results in the shift register getting stuck in the %10 state and continuously outputting the same bit. These cases are shown in red in Table 9. This condition persists as long as the size is not changed again and object is not retriggered, even across horizontal and vertical blank into the next frame. Typically this does not cause problems unless the size is changed in the middle of an image, as otherwise the shift register will have emptied out anyway.

6.3 Collision detection

GTIA has 60 collision bits to indicate when players, missiles, and the playfield collide. This permits fast collision detection at pixel-exact level without the need for the CPU to do expensive bounding box or image comparison checks.

Collision detection mechanism

A collision is flagged between two objects when both objects are active at the same time during display. This means that a collision is not detected until the display logic actually processes the collision location on-screen, and the CPU must wait until the end of a frame or at least past the point of object display in order for collisions to be reliably detected.

Color registers do not play a part in collision detection – the collision logic can distinguish between two objects of the same color. This is sometimes used to establish hidden collision objects for gameplay purposes, such as an invisible wall or a trigger. The collision logic can also see collisions between two objects even if a third object is displayed on top. Collisions are reported for all pairs of colliding objects, so if three players overlap, six collisions are reported: P0P1, P0P2, P1P0, P1P2, P2P0, P2P1.

Playfield collisions

For collision detection purposes, the non-background playfield colors are each separate entities that can register collisions with players and missiles. 32 collision bits in eight registers, P0PF-P3PF and M0PF-M3PF, are devoted to registering P/M collisions against PF0-PF3. No collisions are detected against the background.

In high resolution mode (ANTIC modes 2, 3, and F), the areas corresponding to a 1 bit in the graphics data are considered to be PF2 for collision purposes. Each pair of high-resolution pixels is combined and a collision is detected if either pixel is set where a sprite is present. No collisions are registered against areas with a 0 bit even though those are displayed as non-background color.

No playfield collisions are detected in GTIA modes 9 and 11. In GTIA mode 10, a playfield collision will register whenever pixels using PF0-PF3 codes are present. No P/M collisions are reported for playfield pixels that use P/M color codes in a GTIA mode 10 screen.

Player/missile collisions

Twelve collision bits report collisions between players. A collision between player X and player Y sets two bits, one for player X in the P*y*PL register and another for player Y in the P*x*PL register. A player never registers a collision with itself: the bit for collision between a player and itself is always 0.

Sixteen collision bits in registers M0PL-M3PL report collisions between players and missiles. Each register indicates collisions between all four players against each missile.

There is no support for collision detection between missiles.

Horizontal and vertical blank

P/M collisions are only registered during the visible portions of the screen refresh and are ignored during horizontal and vertical blank. This means that only the portions of objects at horizontal positions 34-221 ($22-$DD) and in scan lines 8-247 ($08-$F7) can trigger collisions.

An object that is so far left or right that it is in partially in horizontal blank can still register collisions in the part that is in the visible region.

Note that if ANTIC fails to activate vertical blank due to having hi-res active on scan line 247, GTIA will process P/M graphics and can report collisions in scan lines in the 248-7 range when the playfield is enabled.

Resetting collision latches

The collision detection bits are latches and will stay set once a collision has been detected. Writing to HITCLR resets all collision latches to zero.

6.4 Priority control

Playfield/object priority

The GTIA uses a priority scheme to determine which objects to display when multiple objects overlap. Bits 0-3 of PRIOR control the relative priority between player/missiles and the playfields. The four official modes are as follows[27]:

PRIOR[3:0]	1000	0100	0010	0001
Top	PF0	PF0	P0	P0
	PF1	PF1	P1	P1
	P0	PF2	PF0	P2
	P1	PF3	PF1	P3
	P2	P0	PF2	PF0
	P3	P1	PF3	PF1
	PF2	P2	P2	PF2
	PF3	P3	P3	PF3
Bottom	BAK	BAK	BAK	BAK

Note that the official hardware manual lists the fifth player (P5) as having the same priority as PF3. This is only partially true, as P5 actually assumes the priority of the highest priority playfield; more on this later.

The exact logic used by GTIA for resolving playfield and player/missile priorities is as follows:

```
PRI01 = PRI0 + PRI1
PRI12 = PRI1 + PRI2
PRI23 = PRI2 + PRI3
PRI03 = PRI0 + PRI3
SP0 = P0 * /(PF01*PRI23) * /(PRI2*PF23)
SP1 = P1 * /(PF01*PRI23) * /(PRI2*PF23) * (/P0 + MULTI)
SP2 = P2 * /P01 * /(PF23*PRI12) * /(PF01*/PRI0)
SP3 = P3 * /P01 * /(PF23*PRI12) * /(PF01*/PRI0) * (/P2 + MULTI)
SF0 = PF0 * /(P23*PRI0) * /(P01*PRI01) * /SF3
SF1 = PF1 * /(P23*PRI0) * /(P01*PRI01) * /SF3
SF2 = PF2 * /(P23*PRI03) * /(P01*/PRI2) * /SF3
SF3 = PF3 * /(P23*PRI03) * /(P01*/PRI2)
SB  = /P01 * /P23 * /PF01 * /PF23
```

In this form, the priority bits enable specific signals that cause elements to suppress lower priority elements.

27 Hardware III.8

Priority mode 0

Clearing all four priority bits PRIOR[3:0] causes the all of the cross-disable signals in the priority logic to turn off, enabling some combinations to mix. The reduced logic for this mode is as follows:

```
SP0 = P0
SP1 = P1 * (/P0 + MULTI)
SP2 = P2 * /P01 * /PF01
SP3 = P3 * /P01 * /PF01 * (/P2 + MULTI)
SF0 = PF0 * /SF3
SF1 = PF1 * /SF3
SF2 = PF2 * /P01
SF3 = PF3 * /P01
```

The effect is to allow playfields 0 and 1 to mix with players 0 and 1, and playfields 2 and 3 to mix with players 2 and 3. The result of two colors mixing is the bitwise OR of their color register contents. PF0/PF1/P0/P1 still have priority over PF2/PF3/P2/P3.

Conflicting priority bits

If more than one priority bit is set, then the more of the cross-disable signals are activated than usual, and the result is that the priority logic turns off outputs more often. This leads to cases where no signals are output, including the background, and the output is black (color $00).

Active layers	PRIOR[3:0] bits										
	0011	0101	0110	0111	1001	1010	1011	1100	1101	1110	1111
PF01+P01	P01	black	black	black	black	black	black	PF01	black	black	black
PF01+P23	P23	P23	PF01	P23	P23	PF01	P23	PF01	P23	PF01	P23
PF01+P01+P23	P01	black	black	black	black	black	black	PF01	black	black	black
PF23+P01	P01	PF23	PF23	PF23	P01	P01	P01	PF23	PF23	PF23	PF23
PF23+P23	black	black	PF23	black	P23	black	black	black	black	black	black
PF23+P01+P23	P01	black	PF23	black	P01	P01	P01	black	black	black	black
P5+P01	P01	P5	P5	P5	P01	P01	P01	P5	P5	P5	P5
P5+P23	black	P23	P5	black	P23	black	black	P23	black	black	black
P5+P01+P23	P01	black	P5	black	P01	P01	P01	P01	black	black	black
P5+PF01+P01	P01	P5	P5	P5	black	black	black	P5	P5	P5	P5
P5+PF01+P23	black	black	P5	black	P23	black	black	black	black	black	black
P5+PF01+P01+P23	P01	black	P5	black	black	black	black	black	black	black	black
P5+PF23+P01	P01	P5	P5	P5	P01	P01	P01	P5	P5	P5	P5
P5+PF23+P23	black	black	P5	black	P23	black	black	black	black	black	black
P5+PF23+P01+P23	P01	black	P5	black	P01	P01	P01	black	black	black	black

Table 10: Priority logic outputs for unusual priority modes

In the above table, P01 is player 0 or 1, P23 is player 2 or 3, PF01 is playfield 0 or 1, PF23 is playfield 2 or 3, and P5 is the fifth player (missiles). If fifth player mode is disabled, P01 and P23 also include the missiles.

All conflicts that produce black are the result of combinations involving players and playfield, where the fifth

player counts as PF3. Combinations between players alone or playfields and the fifth player are always resolved and never produce black.

Fifth player enable

PRIOR bit 4 changes the color of all four missiles to that of PF3, thus allowing them to be used as a fifth player. No other change to the missiles occurs – in order to be used as a player they must be moved together manually. This means, however, that it is possible to take advantage of just the color change and still position the missiles in different places on screen.

For the purposes of priority versus players, the fifth player assumes the priority of playfield 3. It always wins against all other playfields. This leads to a contradiction in the priority mode set by PRIOR[3:0] = %1000, where the playfields are split by players in priority order. In this configuration, PF0-PF1 should cover P0-P3, which should in turn cover PF2-PF3. However, because PF3 actually overrides PF0-PF2 in order to accommodate the fifth player, this leads to the odd result that when all of the following are active:

- Either PF0 or PF1
- At least one of P0-P3
- The fifth player

...PF3 actually shows up from the fifth player in this case, because PF0/PF1 overrides the players, and then PF3 overrides PF0/PF1. However, if PF0/PF1 is taken away, then P0-P3 show up instead.

Enabling the fifth player does not affect collisions in any way. Even though it changes all missiles to use the PF3 color, each individual missile still registers collisions against playfields and players as usual, and no extra PF3 collisions result.

The fifth player has odd interactions with the 16 luma and 16 color modes. The logic that prevents the playfield values from being impressed onto the players only checks the inputs that contribute to player colors. The fifth player bypasses this such that when it is active in these modes, the result is the PF3 color impressed with the luminance or color specified by the playfield.

Multiple color player enable

By setting PRIOR bit 5, it is possible to blend players together in order to produce additional colors. The pairs that blend are P0+P1, P2+P3, M0+M1, and M2+M3. This works simply by disabling the priority logic between these pairs, thus allowing both colors to contribute to the output. The resultant color is the bitwise OR of the color registers involved.

Multiple color mode has no effect on collision detection.

6.5 High resolution mode (ANTIC modes 2, 3, and F)

At the beginning of horizontal blank, ANTIC signals to the GTIA whether high resolution mode is enabled. This mode is enabled for ANTIC modes 2, 3 and F and specifies whether the low two bits of playfield data for each color clock is to be interpreted as individual bits for high resolution mode. This produces pixels at each half color clock, or 320 pixels across for normal playfield width. However, as much of the logic in GTIA operates at color clock rate, this necessitates some logic bypassing and thus some unusual behavior.

When high resolution mode is active, the priority logic always sees PF2, and that is the color that is used unless that playfield is overlapped by players. The high resolution data bypasses the priority logic and conditionally impresses only the luminance from PF1 onto the output. This takes place regardless of whatever color register is used, so the change in luminance occurs on top of anything, including players, missiles, and the fifth player. The collision logic, however, sees a modified PF2C output that is the OR of the two pixels in each color clock, thus registering collisions against PF2 as expected.

Pseudo ANTIC mode E

High resolution mode is forced off whenever any of the GTIA special modes are active, thus preventing the PF1 luminance substitution or PF2C collision from interfering. This leads to a quirk of the GTIA whenever PRIOR[7:6] are set in the middle of a scan line. The high resolution flip-flop can only be set at horizontal blank, but it resets any time PRIOR[7:6] is activated and stays off for the rest of the scan line even if those bits are reset to 00. When this happens, ANTIC continues to encode data in high resolution mode while GTIA starts interpreting it as low-resolution data. Due to the differences in ANx bus encoding, this causes ANTIC mode F to revert to a pseudo mode E, where the bit pairs 00-11 encode PF0-PF3 instead of BAK + PF0-PF2.

Artifacting

In high-resolution mode, the pixel dot clock is high enough and just the right rate that an alternating stream of 0 and 1 bits can trick an NTSC receiver into interpreting the alternating bits as color. This is known as artifacting, and is the same trick used by the Apple II to create color. Unlike the Apple II, however, the Atari lacks the ability to do a 90° phase shift and thus only two phases are available. When the background color is black and the foreground is white, this commonly produces either green/purple colors or red/blue colors with the GTIA, depending on the system. The exact color depends on the relative delay between the chroma and luma paths. Combining artifacting patterns with a colored playfield results in a color that is a vector sum of the artifacting color and the playfield color.

Because the display produced by ANTIC and GTIA has an integral number of color clocks per scan line (228) and per frame (59736), the color subcarrier does not invert phase on successive scan lines or frames, and therefore the same pattern of bits produces the same, consistent color, i.e. $AA bytes are the same color on both even and odd scan lines or frames. This is different from broadcast NTSC where the color subcarrier inverts phase both on scan lines and fields, resulting in the crawling checkerboard pattern.

Artifacting produces different results for PAL or SECAM because the pixel dot clock doesn't match the color subcarrier frequency. This prevents using artifacting to produce a consistent color and generally relegates it to only an unwanted side effect.

6.6 GTIA special modes

Setting the top two bits of PRIOR to something other than 00 enables one of the three special GTIA modes. These three modes have several features in common:

- Each pixel is elongated to occupy two color clocks, giving a resolution across of 80 pixels at normal playfield width.
- The GTIA modes only work properly with the hi-res ANTIC modes 2, 3, and F.
- They allow access to more simultaneous colors per scan line than any other documented modes.

Normally ANTIC sends either one pixel per color clock with five different values (low resolution mode), or two pixels per color clock in monochrome (high resolution mode). When the special modes are active, however, GTIA pairs inputs on alternating clocks for each pixel. One side effect of this is that the GTIA modes can only be scrolled by two color clocks at a time. Attempting to scroll by one color clock causes garbled output as GTIA pairs the wrong sets of bits for each pixel without actually shifting the pixel boundaries.

This pairing also explains why only ANTIC modes 2, 3, and F work. In order to form a four-bit pixel, GTIA extracts the two lower bits from each three-bit value sent by ANTIC per color clock. In the high resolution modes 2, 3, and F, these values are encoded as follows:

- 00 → 100
- 01 → 101
- 10 → 110
- 11 → 111

Most of the low resolution ANTIC modes, however, can only send BAK + PF0-PF2, encoded as follows:

- 00 → 000
- 01 → 100
- 10 → 101
- 11 → 110

This prevents access to any value of the form 11xx or xx11 and thus only gives nine of the possible sixteen pixel values. A few of the low-resolution character modes can output PF3 with the right character index (modes 4-7) and thus can produce the missing 11 output, but not in a way that is generally useful here.

Mode 9 (16 luminances in one color) (PRIOR[7:6] = 01)

Setting PRIOR[7:6] = 01 produces a playfield with a single color, but using sixteen luminance values. As this occurs by bypassing the color registers, this is the only mode in which the lowest luminance bit can be set and therefore 256 distinct color values produced instead of the usual 128. The color of the playfield comes from the background color register.

For priority purposes, the mode 9 playfield is essentially background. No playfield collisions register, and P/M graphics always have priority over the playfield. The playfield drops out in the presence of any player, even for priority conflicts that produce black.

Missiles also have priority over the playfield like players, unless fifth player mode is enabled. When the fifth player is enabled, however, it will mix with the playfield. The result is the color of PF3 combined with the luminance of the playfield.

Mode 11 (16 colors in one luminance) (PRIOR[7:6] = 11)

With PRIOR[7:6] = 11, the playfield is instead a single luminance, but with any of all 16 colors specified by the playfield data. The luminance comes from the background color register, with the exception of pixel value %0000, which is always forced to black.

Mode 11 playfields interact with P/M graphics similarly as with mode 9. When the fifth player overlaps the playfield, the result is as if the background color is replaced with PF3: PF3's luminance with the playfield's color, except if the playfield is %0000 in which case the result is black.

Mode 10 (9 color mode) (PRIOR[7:6] = 10)

The nine color mode, activated by PRIOR[7:6] = 10, is more unusual than the other two special modes. All of the colors come from the color registers, giving more color flexibility, and causing more interaction with the priority and collision logic.

The four bit pixel values activate color registers as follows:

- 0000-0011: P0-P3
- x100-x111: PF0-PF3
- 10xx: Background

For priority purposes, the pixel values which correspond to player colors act as though that player/missile were active and are thus modified by the priority settings in PRIOR[0:3]. They do not, however, activate player collisions. The nine color mode, however, is able to activate playfield collisions via the PF0-PF3 codes.

The nine color mode is delayed by one color clock (one half pixel) and thus appears shifted slightly right relative to all other modes.

Border regions are rendered with a code of 0000 or player 0. This means that players and missiles 1-3 will generally be hidden in borders except for when multicolor P/M or fifth player mode allows them to overcome

player in priority.

This mode has a quirk when driven with a low-resolution ANTIC display mode that does not occur with the 16 color/luminance modes. Ordinarily, the BAK and PF0 signals from ANTIC produce the same result as they both send 00 over the AN0 and AN1 lines. However, in the 9 color mode, the BAK signal mutes the playfield signals for the entire two color clock pixel when sent as the second half. This leads the a 9 color mode anomaly where the four bit combination 1000 in ANTIC mode E results in the background color rather than the PF0 color that the resultant 0100 pixel would normally indicate.

6.7 Cycle timing

The following sections all assume that a write has taken place on cycle 65 of a scan line. In a normal width mode E line, this would be immediately before ANTIC reads data for positions $8C-$8F.

Color register changes

A write to a color register takes place one color clock later, so a write to COLPM0 at cycle 65 shows up on screen at $81.

P/M priority changes

A write to PRIOR bits 0-3 or 5 takes place two color clocks later, so a write at cycle 65 shows up on screen at $82.

The fifth player bit (PRIOR bit 4) normally also takes place two color clocks later at $82. However, on some systems this circuit is **temperature sensitive** and shows a one-cycle artifact until $83 when the system has warmed up.

P/M graphics changes

A write to a player/missile graphics register only takes effect when the sprite retriggers and its shift register is reloaded. The delay for this is three color clocks. A write to GRAFP0 at cycle 65 would only take effect for player 0 at $83 or later.

P/M position/size changes

A write to a player/missile position or size register must take place five color clocks in advance to take effect. This means that a write on cycle 65 can prevent display of a player at or right of $85, and reposition it to $85 or farther. Effectively, both the old and the new player image are clipped on the left side of $85.

Changes to the size register will take effect immediately, with the remaining bits in the shift register shifting out at the new width. However, due to the design of the stretching circuitry, switching between double and quadruple width is slightly erratic, with the double-to-quadruple change showing a slightly uneven relation and the quadruple-to-double change being slightly non-monotonic. Changes to and from normal width are always well behaved.

Changed	Timing
Color register	$81 (1 cclk)
PRIOR bits 0-3, 5	$82 (2 cclks)
PRIOR bit 4	$82-83 (2-3 cclks)
PRIOR bits 6-7	$83-85 (3-5 cclks)
Player/missile image	$83 (3 cclks)
Player/missile position	$85 (5 cclks)

Table 11: Timing for mid-screen writes to GTIA registers

GTIA mode changes

A change to bits 6-7 of PRIOR takes place between 3-5 color clocks after the write, primarily after 4 color clocks with a possible cycle of artifact on each side. For a write on cycle 65, the change takes place at positions $83-$85. The nature of the artifact on-screen depends on the exact transition:

- **Mode 8 to mode 9/11:** Clean transition after 4 color clocks.
- **Mode 8 to mode 10:** Clean transition after 3 color clocks.
- **Mode 9/11 to mode 8:** 1-2 color clock transition after 3 color clocks. At $83, the mode 9/11 pixel is cut in half and the playfield is absent, showing background color if there are no players or missiles. Pseudo mode E display begins at $84, but the data from $83 is displayed instead. (Presumably this is an artifact of timing sensitivity in disabling the mode 10 delay line.)
- **Mode 10 to mode 8:** One color clock transition after 4 color clocks.

> **Machine-specific Behavior Warning**
>
> On some systems, the artifact at $84 does not occur when switching from mode 9/11 to mode 8.

6.8 General purpose I/O

Console switches

The CONSOL register controls and senses the state of four uncommitted I/O lines, each of which can be used in either read or write mode. Setting bits 0-3 to 1 causes the corresponding line to be pulled down and to read as a 0; clearing a bit allows the line to be read normally. On the Atari, bit 3 is connected to the console speaker and bits 0-2 are connected to the Start, Select, and Option bits, respectively.

Trigger inputs

TRIG0-3 report the state of the trigger input lines. Bit 1-7 are always 0, while bit 0 reads 1 for an inactive trigger and 0 for an active trigger. These are normally connected to joystick triggers. On the XL/XE, TRIG2 is hardwired inactive while TRIG3 indicates cartridge mapping state, bit 0 = 1 for cartridge ROM present. The XEGS also maps TRIG2 to a keyboard presence line, bit 0 = 1 for keyboard present.

Trigger latching can be enabled by setting bit 2 of GRACTL. This causes the trigger registers to latch so that they continue to register activation even after a trigger is released, allowing trigger activation to be detected at any time regardless of how often the TRIG0-3 registers are polled. Latching can only be enabled for all triggers at the same time, however, so enabling it on an XL/XE machine will also affect cartridge map sensing.

The SECAM version of the GTIA, the FGTIA, has an additional quirk in that trigger inputs are only sensed at the

beginning of horizontal blank.

6.9 Further reading

The main source for functionality and register level descriptions for the GTIA is the Hardware Manual [ATA82] as usual, but it only covers CTIA level of functionality. Read the GTIA datasheet [AHS99a] for additional details on the GTIA modes and on communication between ANTIC and GTIA.

Chapter 7
Accessories

7.1 Joystick

The Atari 8-bit computer series uses the same digital joystick used by the 2600 VCS. The direction sensors are connected to four contiguous bits on the PIA. Ports 1 and 2 use port A, whereas ports 3 and 4 on the 600/800 use port B:

7							0
Port 2/4				Port 1/3			
right	left	down	up	right	left	down	up

All direction bits are inverted, so these ports register $FF either when no joysticks are attached or all connected joysticks are centered.

There are generally no circuits to prevent both the left and right or up and down signals from being activated at the same time. Although it normally does not occur due to the design of the joystick, both opposing signals can be active at the same time either due to noise or simply due to another type of controller being plugged into the joystick port.

The joystick button is attached to one of GTIA's TRIGx inputs. The trigger bit is also inverted, reading $00 when the button is depressed and $01 when released.

7.2 Paddle

Paddle controllers consist of a single rotation knob and a trigger button. Two paddle controllers connect to a single game controller port, so up to four paddles can be attached to an XL/XE and eight paddles to a 400/800.

Paddle knob

The rotating knob on each paddle sends a signal to the computer that allows it to read the angular position of the knob with fine accuracy. On a standard CX30 paddle, the angular range of each paddle is about 330º. The position of the knob is read through the POT0-POT7 registers in POKEY, which have a range of 1-228 ($01-$E4), where 1 is fully counterclockwise (left) and 228 is fully clockwise (right).

In order to read the paddles, the POTGO register must be written. This resets all counters and begins charging a capacitor for each paddle through the potentiometer attached to the knob, where the position of the knob controls the charge rate. Once a capacitor reaches the threshold, the corresponding bit in ALLPOT is set and a scan line count is latched into the corresponding POTn register. As these counts are latched from a counter running at scan line rate (15.7KHz), the count isn't actually latched until that number of scan lines has actually passed. Typically the POTn values are read and then POTGO strobed from the vertical blank interrrupt.

The exact timing and values produced by this process depend on a couple of variables, specifically the voltage threshold used by POKEY, the resistance range of the potentiometers in the paddles, and the value of the charging capacitor. The ideal formula relating a paddle position as a fraction of the rotational range and the voltage threshold is as follows:

$$scanlines = \ln\left(\frac{V_{cc}}{V_{cc} - V_{threshold}}\right) \times (1 - fraction) \times RC \times 15700$$

Vcc is 5V, Vthreshold is 1.9-2.6V[28], R is 1MΩ for CX30 paddles[29], C is 0.047µF, fraction is 0 for full left and 1 for full right. Values will vary, particularly due to the wide range in threshold voltage, but for a threshold mid-value of 2.25V, this gives a scan line range of 1-441. Since the scan line counter only counts up to 228, this means that

28 [AHS03] p. 22 (V_{T+} positive-going threshold voltage)

29 There is a 1.8KΩ resistor in the computer in series with the potentiometer, but it is small enough in comparison that it can be ignored.

only about half of the paddle range is used (mid way to full clockwise), the left side returning full 228. The above formula is linear in *fraction* and the potentiometer in the CX30 is also linear, so the relationship between angular position and the POTn values is also linear.

Paddle trigger

Each paddle also has a trigger button associated with it. The paddle trigger is connected to the PIA ports, as with the joystick direction inputs. The lower of the paddle pairs – corresponding to POT0/2/4/6 – activates the left direction (bits 2/6) and the higher of the paddle pairs activates the right direction (bits 3/7). As usual, the bits are inverted and read as 0 when the button is activated.

7.3 Mouse

A computer mouse consists of up to three buttons and a pair of motion detectors. There are two types of mice that can easily be connected to an 8-bit Atari, Atari ST and Amiga. The two types are similar, with minor differences in the motion encoding.

The horizontal and vertical axes are encoded using quadrature encoding on pairs of control lines, producing different cyclical patterns based on the direction of movement, either 00-01-11-10-00 or 00-10-11-01-00. The pattern repeats indefinitely as long as the mouse is moving and there is no limit to how far the mouse can move. The quadrature signals are connected to the joystick direction bits and are reflected in the PIA port, although the wiring differs between the mouse types:

	Bit 3/7	Bit 2/6	Bit 1/5	Bit 0/4
Joystick	Right	Left	Down	Up
ST mouse	YB	YA	XA	XB
Amiga mouse	XB	YB	XA	YA

The pattern 0/0, 1/0, 1/1, 0/1 signifies rightward motion for the XA/XB signals and downward motion for the YA/YB signals.

The quadrature inputs must be sampled at a high rate in order for the mouse to work, as each change must be detected for motion to be measured properly. For instance, if two changes were to occur between measurements, i.e. 00 to 11, it would be impossible to determine the direction of motion. For a 100 cpi (counts per inch) mouse, this requires a minimum sampling rate of 300Hz to support motion up to 3 inches/second, with higher rates needed for faster motion or higher resolution mice. Checking the mouse from a VBI handler is therefore unlikely to produce satisfactory results.

There are up to three buttons on a mouse. The left mouse button is connected to the joystick trigger input and can be read the same way; the right and middle mouse buttons, if present, are connected to the paddle A and B inputs. Unfortunately, the mouse connects these lines to ground instead of +5V as the paddle does, so the Atari hardware cannot read them – there is no circuitry hooked up in this configuration to charge the pot capacitors.

7.4 Light Pen/Gun

Light pen and light gun devices sense the electron beam of a cathode ray tube (CRT) monitor to report the screen position of the device to the computer. They only work with CRTs that do single scan – they do not work with CRTs that scan at 100/120Hz or with LCDs.

Sensing signal connection

As the light pen or gun senses the passing of the electron beam, it sends a pulse to the computer on the joystick trigger input on its connected joystick port. On the 400, the device must be connected to port 4, but it may be

connected to any of the available ports on the 800/XL/XE models. Any trigger on any of the wired ports will register a pen position, including a non-light-sensing device such as a joystick.

Position reporting mechanism

The appropriate trigger lines are connected to the light pen ($\overline{LP}$) input on ANTIC, which latches the current horizontal and vertical position counters into the PENH and PENV registers. This latching only occurs on the edge when the line is asserted; if the trigger line is held down, such as from a joystick, the latched position will reflect the time of depression.

The PENH register reports the horizontal position with color clock resolution, from 0-227, while PENV reports the vertical position with two-line resolution, from 0-130 or 0-155, similar to VCOUNT. Latching is not limited to the visible area of the screen; ANTIC will record a location in the border or even in the blanking intervals if a pulse arrives during that time.

PENH and PENV will continue to reflect the last known position if no further trigger pulses arrive. They are not cleared by vertical blank.

On-screen detection

There is no direct way to sense if a light-sensing device is aimed at the screen. However, since the timing signal is connected to the trigger inputs, it is possible to read the TRIG0-3 registers on GTIA to determine this, since an off-screen device will not send pulses. Typically bit 2 of GRACTL is set to enable latching on the trigger inputs, making it easier to detect the quick pulse from a VBI routine.

7.5 CX-85 Numerical Keypad

The Atari CX-85 Numerical Keypad is a 17 key pad that attaches via the joystick port. It sends six signals through the joystick direction, trigger, and paddle B lines. The corresponding four bits in PORTA are set for each key as follows:

ESCAPE 1100	**7** 0101	**8** 0110	**9** 0111	**-** 1111
NO 0100	**4** 0001	**5** 0010	**6** 0011	**+/ENTER** 1110
DELETE 0000	**1** 1001	**2** 1010	**3** 1011	
YES 1000	**0** 1100		**.** 1101	

Table 12: CX-85 keypad to PORTA bit pattern mapping

The paddle B input (POT1/3/5/7) is used to distinguish the ESCAPE key from the 0 key, which both share the 1100 encoding. When ESCAPE is pressed, the paddle line is negated and the POTx register reads 228; for any other key it is asserted and POTx reads 1.

The trigger is asserted (0) as long as any key is pressed; when this happens, the joystick direction bits in PORTA and the pot line indicate the key that was pressed. The PORTA and POTx values will persist after the key is released, or even if other keys are pressed while the first key is held down. If the first key is then released, the keypad may begin reporting one of the other keys that are still pressed, although this is not always the case.

7.6 XEP80 Interface Module

The XEP80 Interface Module is a device that plugs into joystick port 1 or 2 and provides a separate 80-column, monochrome text display. It also has limited graphics capability.

Communication protocol

Data is transferred to and from the XEP80 via a serial protocol at a baud rate of 15.625KHz. This is designed to be close to the horizontal scan rate of 15.7KHz on the host computer. Communication from the host to the XEP80 is by means of the joystick up line (bit 0 or 4 of PORTA/B) and communication from the XEP80 to the host is via the joystick down line (bit 1 or 5 of PORTA/B).

The data format is one start bit, followed by nine data bits starting with the LSB, and ending with one stop bit. Bytes sent with bit 8=0 are characters to print, while bytes sent with bit 8=1 are commands.

When sending data back to the host, the XEP80 actually uses two stop bits, giving the host one bit cell of time between the bytes.

Cursor updates

Whenever a character is read or written, the XEP80 sends back update bytes to tell the computer that the operation has completed and the new location of the cursor. All cursor update bytes have bit 8 set. The cursor update consists of one to three bytes of the following types:

- $100-150: New horizontal position, with no following vertical position byte.
- $180-1D0: New horizontal position, to be followed by a new vertical position byte.
- $1E0-1FF: New vertical position.

The horizontal position update only indicates positions 0-80, with 80 being returned for any positions to the right of that. A horizontal position query command must be issued to retrieve the true horizontal position beyond column 80.

If the cursor doesn't change, such as if an escape sequence is started ($1B), a dummy horizontal update is sent.

Burst mode

The XEP80 can be placed into a burst mode where cursor updates are suppressed for faster text output. Instead, the XEP80 simply pulls its output low while it is busy and raises it when it is done. This avoids the delay of waiting for the cursor update bytes, at the cost of the computer needing to manually query the cursor position when needed.

There is a short delay between when the XEP80 receives a character and when it can assert the busy output. As a result, the host must wait 90µs before checking busy state.[30] This is about 160 machine cycles.

Burst mode is automatically activated in pixel graphics or printer mode.

Left and right margins

During put or get character operations, the cursor is constrained to be within the left and right margins, inclusively. Whenever the cursor advances beyond the right margin, it is moved to the left margin on the next line. By default, the left and right margins are set to columns 0 and 79.

Note that while the cursor is restricted to within the margins, vertical scroll operations always move entire rows including text outside of the margins. Line clear operations, on the other hand, clear 80 columns starting at the scroll position.

30 [ATA87] p.11

Logical lines

The first 24 lines of the screen are organized as a series of logical lines, where each logical line contains one or more contiguous physical lines. Physical lines are grouped into a logical line when characters are printed past the right margin at the end of a logical line.

There are two differences between the logical line handling in the XL/XE OS's screen editor and the XEP80. First, the OS screen editor allows logical lines to contain a maximum of three physical lines (120 characters), while there is no limit in the XEP80 and the entire screen can be one big logical line. Second, instead of using an external bitfield to track logical line boundaries, the XEP80 tracks logical line groupings by means of EOLs in the frame buffer. The end of a logical line is marked by an EOL at the right margin column.

Status row

The 25th row (row 24) is special as it is the status row, for which much functionality is disabled. When the cursor is in the status row, only the escape and clear special characters are processed and all other characters are printed. Advancing past the right margin wraps back to the left margin within the status row.[31]

Overscan

The XEP80 is notorious for extreme amounts of overscan in text mode that can make the outer portions of the display invisible to the user. The primary reason for this is the use of a 10 row character cell with 25 character rows, giving 250 active display scan lines out of 262 total in a non-interlaced NTSC display. This exceeds the 243 scan lines per field normally used for a fully overscanned display and far exceeds the approximately 192 scan line region typically considered title-safe, making the top and bottom rows of the screen hard to see on regularly adjusted displays. The situation is not much better in PAL, where the character cell height is increased to 12 rows, giving 300 active scan lines.

Video memory layout

8KB of video memory is present in the XEP80 for text, graphics, and auxiliary data. In text mode, this is organized as 25 rows of 256 bytes each for easy addressing, from $0000-19FF. This allows for horizontally scrolling the 80x25 display window over a 256x25 virtual text screen. While each row is contiguous, the XEP80 will display them out of order as scrolling is performed by swapping display row pointers rather than moving data in memory.

In addition to the text display, video memory is also used for tracking tab stops and queued print data. Memory at $1A00-1AFF contains flags for tab stops at each column, and $1B00-1FFF is used for the print buffer.

Internal memory layout

64 bytes of internal memory are also contained within the NS405 processor and contain working registers, the stack, and variables. These bytes are normally managed for internal use by the XEP80, but may be written using command $E5.

Of the internal memory locations, the most interesting are addresses $20-38, which contain the high byte of the starting address for each display row. Bits 0-4 are used for memory addressing, while bit 5 selects one of the two ATASCII character sets in the external character ROM and bit 6 bypasses the external character ROM entirely for pixel graphics or the internal character set. The row pointers are only reinitialized by power-on or a master reset ($C2) command; afterward they are swapped around as needed during scroll and insert/delete operations.

31 [ATA87] p.5 has a warning about a lockup if the cursor is moved to the status row while BASIC is at its READY prompt. This is an issue with the handler software – it tries to read characters until it finds an EOL, and due to the special behavior in the status row, it can end up looping infinitely.

Character display attributes

Two attribute latch registers determine the display characteristics of characters on screen. Attribute latch 0 is used when character data bit 7 = 0 while attribute latch 1 is used when character data bit 7 = 1. This mostly corresponds to characters $00-7F and $80-FF, except when the ATASCII character sets are enabled in which case $9B (EOL) also uses attribute latch 0.[32] Normally both attribute latches are set to $FF, which disables all special attributes.

The attribute registers can be set by means of commands $F4 and $F5, which each set one of the attribute latches to the value of the last character written. All bits in the attribute byte have inverted behavior such that they must be set to 0 to enable the feature:

- **Bit 0 (Reverse video)**: Inverts the entire character cell.
- **Bit 1 (Half intensity)**: This bit sets characters to half-intensity. This feature is not hooked up in the XEP80, so it does nothing.
- **Bit 2 (Blink)**: Causes the character to blink on and off by alternately blanking character data. This happens at half the cursor blink rate, normally toggling every 32 frames. If reverse video is also enabled on this character and the reverse video blink field option is set in the VCR (bit 0), the entire character cell is inverted instead.
- **Bit 3 (Double height)**: Stretches a character vertically to double its normal height. When active, the blanking function is disabled and bit 6 is repurposed as the character half bit, where 0 selects the lower half and 1 selects the upper half. Double height mode is only functional with the internal character set or block graphics and does not work with the ATASCII character sets.
- **Bit 4 (Double width)**: Stretches a character horizontally to double its normal width, covering both the current and next character cells. The next character and its attribute are ignored.
- **Bit 5 (Underline)**: ORs an underline into the character graphic.
- **Bit 6 (Blank)**: Blanks out all character data.
- **Bit 7 (Block graphics)**: Replaces the character from the character set with block graphics instead, based on bits 0-6 of the character. This mode only works with the internal graphics set; it produces garbage with the ATASCII character sets due to the character graphic data being converted to block graphics instead of the original character.

The order of operations for attributes is block graphics, double width + height, blank + blink, underline, reverse video blink field, reverse video, and then finally global reverse video.

Character sets

Three character sets are available with the XEP80, two of which correspond to the standard ATASCII and international ATASCII character sets, while the third is an internal character set within the NS405. The ATASCII and international ATASCII character sets can be mixed on a line-by-line basis, although this is not normally exposed and only available by writing directly to internal memory to toggle bit 5 of character row address bytes.

The two ATASCII character sets are both 256 characters in size, with the $80-FF characters being inverted versions of $00-7F. Thus, $80-FF produce inverted character graphics even though the attribute latches are not set for reverse video. The exception is the inverted escape or EOL character $9B, which is blanked in both character sets to keep the EOLs in the framebuffer from showing up.

32 This bizarre EOL anomaly is due to the way external character sets are implemented in the NS405: attribute latch selection is based on bit 7 of the data coming into the NS405, and when the ATASCII character sets are enabled this actually comes from bit 7 of the character data and not the character name. The external character ROM is set up to emit bit 7 = 0 for $00-7F and $9B and bit 7 = 1 for $80-9A and $9C-FF. When the external character ROM is bypassed, the NS405 sees the actual characters and so the split between the latches is the more normal $00-7F / $80-FF.

The internal character set contains only 128 characters and so does not show inverse video unless the attribute latches are changed. Because it does not contain the hacked-in blank for the EOL character, enabling the internal character set causes blank areas of the display to show ä instead.

Block graphics

Clearing bit 7 of one of the attribute latches causes the corresponding half of the character set to display block graphics. This divides the character cell into a 3x3 grid with bits 0-6 of the character set lighting the sub-blocks. Since there are 9 sub-blocks and only 7 bits, bits 0 and 5 control two sub-blocks each:

0	1	0
2	3	4
5	6	5

Table 13: Character bit to block graphics mapping

Block graphics normally only work properly with the internal character set. The reason is that it requires the NS405 to directly see the original character bytes, and when the ATASCII character sets are enabled those bytes are translated through the character ROM. The result is that each row of ATASCII character graphics data is interpreted as block graphics per the layout above, resulting in garbled block graphics. Enabling the internal character set disables the external character ROM and allows block graphics to work correctly. It can also be made to work with the ATASCII character sets by writing into internal memory to set bit 6 on character row address bytes to bypass the external character ROM for those rows.

Initial state

The power-on or post-reset state of the XEP80 is as follows:

- 60Hz text mode
- Attribute latches set to $FF
- List mode disabled, escape not active
- Left margin at 0, right margin at 79
- RAM cleared to EOL ($9B)
- Tabs set every 8 characters starting at the 8th column (column 7), and also at column 2

Special characters

Move up ($1C)

Moves the cursor up one physical line, wrapping from row 0 to row 23.

Move down ($1D)

Moves the cursor down one physical line, wrapping from row 23 to row 0.

Move left ($1E)

Moves the cursor left, wrapping from the left margin to the right margin within the same physical line.

Move right ($1F)

Moves the cursor right, wrapping from the right margin to the left margin within the same physical line. An EOL is replaced with a space if it is the character under the cursor prior to moving right.

Backspace ($7A)

Moves left one character within the current logical line and replaces the character at the new position with a space. If the cursor is at the left margin, it will move to the right margin on the previous line if that is part of the same logical line (no EOL at right margin); otherwise, the backspace operation is ignored.

Tab ($7F)

Advances the cursor right one character until the next tab stop is reached, replacing EOLs with spaces in positions that it leaves. This will splice logical lines together without inserting physical lines if the end of a logical line is breached.

Clear tab ($9E) / Set tab ($9F)

Sets or clears the current horizontal position as a tab position. Neither the framebuffer nor the cursor position are modified.

Command set

Set Horizontal Cursor Position ($00-4F)

Moves the cursor to the specified horizontal position.

Set Horizontal Cursor Position High Nibble ($50-5F)

Modifies the high four bits of the horizontal cursor position to $0x-Fx. The lower four bits are not modified.

Set Left Margin ($60-6F)

Sets the left margin to positions 0-15.

Set Left Margin High Nibble ($70-7F)

Sets the high bits of the left margin position to $0x-Fx. The lower four bits are not modified.

Set Vertical Cursor Position ($80-97)

Moves the cursor to the specified vertical position.

Set Cursor to Status Row ($98)

Moves the cursor to row 24, the status row.

Set Graphics to 60Hz ($99)

Reinitializes the XEP80 in 320x200 pixel graphics mode at 60Hz refresh rate.

Modify Graphics to 50Hz ($9A)

Changes video timing parameters to display pixel graphics at 50Hz refresh. This only works properly if the XEP80 is already in graphics mode.

Set Right Margin ($A0-AF)

Sets the right margin to positions 64-79 ($40-4F).

Set Right Margin High Nibble ($B0-BF)

Sets the upper four bits of the right margin position to $0x-Fx. The lower four bits are not modified.

Read Char and Advance ($C0)

Reads and returns the character under the current cursor position and then advances to the next position. This will return EOLs without translating them to spaces. The cursor wraps within the margins and either stays in the status row or advances to the next row if not in the status row. If the cursor goes beyond row 23, the screen will scroll.

A cursor update follows the read byte.

Read Horizontal Position ($C1)

Returns the horizontal cursor position. Unlike the cursor update data, this returns the unmodified horizontal position over the full $00-FF range and is useful when the cursor update indicates $50+.

Master Reset ($C2)

Reinitializes the XEP80, resetting everything that the power-on path does except for UART parameters. This includes the system and video control registers, the entire timing chain, and all state, as well as filling RAM with EOLs.

An $01 byte is returned on completion.

Get Printer Port Status ($C3)

Returns $00 if the printer is busy and $01 if it is online and ready.

Fill Memory With Previous Character ($C4)

The entire 8K of memory is filled with the last character written. This is intended to be used with pixel graphics mode since the byte is written in reversed bit order and the entire 8K is overwritten, including memory that would be used by the tab array and print buffer in text mode.

An $01 byte is returned on completion.

Fill Memory With Space ($C5)

Fills all 8K of memory with spaces ($20). An $01 byte is returned on completion.

Fill Memory With EOL ($C6)

Fills all 8K of memory with EOLs ($9B). An $01 byte is returned on completion.

Read Character Without Advancing ($C7) (undocumented)

Returns the character at the current cursor position. No EOL translation occurs, the cursor is not moved, and no cursor update is sent.

Read Timer Counter Register ($CB) (undocumented)

Reads and returns the value of the 8048 T register. This register is set to $00 for text mode and $03 for graphics

mode.

Clear List Flag ($D0)

Turns off list mode, enabling normal escape processing.

Set List Flag ($D1)

Turns on list mode, which causes all characters except for EOL ($9B) to be escaped and printed.

Set Normal Transmit Mode ($D2)

Disables burst mode so that each character is followed up by a cursor update of one or more bytes. This also exits printer mode.

Set Burst Transmit Mode ($D3)

Turns on burst mode. In burst mode, the XEP80 lowers its transmit line while busy and raises it when ready. No cursor update is sent. This also exits printer mode.

Set ATASCII Character Set ($D4)

Changes the text display to use the standard ATASCII character set, including all text currently on screen. This is the same as the standard OS character set at $E000-E3FF except that $9B displays as blank.

Set International Character Set ($D5)

Changes the text display to use the international ATASCII character set, including all text currently on screen. This is the same as the alternate OS character set at $CC00-CFFF except that $9B displays as blank.

Set Internal Character Set ($D6)

Changes the text display to use the internal character set inside the NS-405.

Modify Text Display to 50Hz ($D7)

Changes the text display to to 50Hz and taller characters for a PAL display.

Cursor Off ($D8)

Hides the cursor.

Cursor On ($D9)

Shows the cursor and sets it to solid mode.

Cursor On, Blinking ($DA)

Shows the cursor and sets it to blink mode. The cursor blinks on and off with a period of 16 frames per state.

Move to Logical Start ($DB)

Moves the cursor vertically to the start of a logical line. A logical line is defined as a set of contiguous physical lines where all but the last physical line have a non-EOL character at the right margin.

Note that the horizontal position of the cursor is not changed by this command.

Set Scroll Window ($DC)

Horizontally scrolls the text window so that cursor is at the left-most column on screen.

Set Printer Output ($DD)

Redirects character output to the printer. This automatically turns on burst transmit mode.

Set White on Black ($DE)

Turns off reverse video mode.

Set Black on White ($DF)

Turns on reverse video mode.

Set Extra Byte ($E1, $E4, $E6, $EE, $F0, $F2, $F9) (undocumented)

Copies the value of the last character to the extra byte. The extra byte is used for debugging commands that require two bytes of input. This should be followed up immediately with another command to use the extra byte, as it can be overwritten by many commands as well as some text movement operations (insert/delete).

Write Internal Memory ($E5) (undocumented)

Writes an internal memory location using the address specified by the extra byte and the value of the last character.

Write Video Control Register ($ED) (undocumented)

Writes the value of the last character into the video control register (VCR) of the NS405.

D7:D6 Display mode

0X	Text, internal character set
10	Text, external character set
11	Pixel graphics

D5 Display enable

0	Disable display
1	Enable display

D4 Internal/external attribute mode

0	Internal attribute latches (XEP80 operating mode)
1	External attribute memory (not supported by XEP80)

D3 Reverse video

0	Reverse video display for entire screen (note that this stacks with reverse video on each character)
1	Normal display

D2 Cursor reverse video

0	Cursor inverts character cell
1	Cursor overwrites character cell

D1 Cursor blink

0	Cursor is solid
1	Cursor blinks

D0 Reverse video blink field/character

0	Character data blinks when reverse video is enabled in attributes (blink between inverted char and filled cell)
1	Whole character cell blinks when reverse video is enabled in attribute (alternately invert/don't invert character cell)

Set Attribute Latch 0/1 ($F4 / $F5)

Sets the one of the two attribute latches used to format text characters on screen. Attribute latch 0 is used for character data with bit 7=0 while attribute latch 1 is used for character data with bit 7=1. Both attribute latches are set to $FF by default. The attribute latch is set to the value of the last character.

Set Timing Control Pointer ($F6)

Sets the Timing Control Pointer (TCP) register from the value of the last character. This register sets the index of the next register to modify in the NS405 timing control chain. Only bits 0-3 are valid.

Set Timing Control Register ($F7)

Sets the register in the timing chain pointed to by the TCP to the value of the last character. Afterward, the TCP is advanced to the next register.

Set Vertical Interrupt Register ($F8)

Sets the NS405 VINT register to the value of the last character. The VINT register controls the NS405 equivalent of ANTIC's DLI, such that an interrupt is fired at the end of the specified row. The XEP80 initializes VINT to the second to last row in the main region (22) and uses the vertical interrupt to reset the current row counter to the last row (23).

Set Baud Rate ($FA)

Sets the NS405 PSR (prescaler) register to the value of the last character and the BAUD register to the value of the extra byte. Bits 4-7 of the PSR select the prescaling factor in half-factor increments from 3.5 to 11. Bits 0-2 of the PSR supply bits 8-10 of the divisor, while the BAUD register supplies bits 0-7. The resultant baud rate is as follows:

$$baud = \frac{750{,}000}{(3.5 + 0.5 \times PSR[7:4]) \times (PSR[2:0] : BAUD + 1)}$$

The defaults are PSR=$90 and BAUD=$05. This sets the prescaler to ÷8 and the divisor to ÷6, for a final baud rate of 750,000 ÷ 48 = 15,625 baud.

Set UART Control Register ($FB)

Sets the NS405 UCR register to the value of the last character.

Set UART Multiplex Register ($FC)

Sets the NS405 UMX register to the value of the last character. This register allows either the transmit or receive rate to be divided down from the other, giving asymmetric baud rates. Exactly one bit from bits 0-5 is set to select a divisor from ÷1 to ÷32, respectively. Bit 7 then selects the divided down rate for transmit (0) or receive (1).

On init, UMX is set to $01 to use the same baud rate for transmit and receive operations.

Set UART Transmit Register ($FD)

Sets the NS405 XMTR register to the value of the last character, retransmitting that character back to the computer.

Strobe Printer ($FF)

Sends the printer a strobe indicating a new byte is available, without actually setting a new byte.

Chapter 8
Cartridges

8.1 Cartridge port

Address regions

Cartridges are accessed through three memory address windows:

- Left cartridge: $A000-BFFF
- Right cartridge: $8000-9FFF
- Cartridge control (CCTL): $D500-D5FF

The left cartridge slot is common to all hardware models and can map to any of the three regions. The right cartridge slot is only present on the 800 and can only map to the right cartridge and cartridge control regions. All three regions can be read/write, if the cartridge supports it.

These hardware regions are decoded by the computer itself and are the only ones accessible to the cartridge port; the cartridge cannot map to any other memory regions. It does have access to the read/write line, though, and can handle writes as well as reads.

> **Warning**
>
> The main computer hardware typically is tolerant of false reads as there are only two hardware registers that have side effects on reads, PORTA and PORTB on the PIA. Cartridges, on the other hand, often have banking registers in the $D500-D5FF cartridge control region that can trigger a bank switch on a read. This includes false reads from indexed addressing modes and DMA.
>
> Care should therefore be used when accessing registers in the CCTL or PBI ranges using abs,X, abs,Y, (zp,X), and (zp),Y addressing modes. For instance, using LDA $D5FF,X with X=$08 to access a PBI register at $D607 can trip a cartridge bank switch due to a false read from $D507.
>
> Similarly, display lists should be managed properly to avoid accidentally having ANTIC DMA from the cartridge bank registers. Overwriting an active display list with $D5 bytes, for instance, can cause playfield DMA to read from $D5D5 and crash the program by switching cartridge banks.

Power-on and reset behavior

Cartridges may or may not have a known state on a cold start, depending on whether they have circuitry to ensure a reset on power-up. Those that don't and have hardware registers essentially power up in indeterminate state and must be programmed accordingly. For instance, a banked cartridge without reset circuitry can power-up in any bank, so all banks must contain startup code to jump to the proper startup bank.

The cartridge port does not have the computer reset signal exposed, and so cartridges are not normally able to detect a warm reset by the System Reset button. Therefore, even cartridges that have power-up reset circuitry may not be able to reset themselves on a warm reset or a software cold reset and still need startup code in all banks. It is theoretically possible to do this based on detecting when header addresses are read from the cartridge, but doing so is not common. More typically, a hardware button is included to allow the user to manually reset the cartridge.

Late hardware reset

Some cartridges can also experience hardware delays in power-on reset. When these delays are long enough, they can cause the cartridge to change behavior after the 6502 has already begun executing the OS cold start initialization code. One symptom that this can cause is a cartridge that fails to reliably run a software image configured a diagnostic cartridge to the OS, which is checked very early in OS boot, but works when there is enough delay between power-on and the cartridge boot attempt. This can include configuring the cartridge

software as a non-diagnostic cartridge to the OS, on a software cold reset.

> **Warning**
>
> An Ultimate1MB-equipped system can fail to exhibit this issue due to the additional delays caused by the Ultimate1MB's BIOS code, which executes before the regular OS. This additional delay then gives the cartridge hardware enough time to fully reset, hiding the issue.

8.2 Atarimax flash cartridges

MaxFlash 1Mbit cartridge

The MaxFlash 1Mbit cartridge maps one megabit (128KB) of flash memory, 8KB at a time, through $A000-BFFF. Bank switching is performed by either read or write accesses to $D500-D51F, where address bits 0-3 control the bank and bit 4 disables the cartridge when set. Written data is ignored.

The flash ROM can be programmed in-place from the computer through standard flash ROM unlock and programming sequences.

Flash types seen in the wild:

- AMD Am29F010 ($01/$20)
- Micron M29F010B ($20/$20)

MaxFlash 8Mbit cartridge

The MaxFlash 8Mbit cartridge maps eight megabits (1MB) of flash memory, 8KB at a time, through $A000-BFFF.

Bank switching is performed by either read or write accesses to $D500-D5FF, where address bits 0-7 control the bank and bit 7 disables the cartridge when set. Written data is ignored.

Two 4Mbit flash chips are present in this cartridge. Like the 1Mbit cartridge, the 8Mbit cartridge can also be programmed in-place.

Flash types seen in the wild:

- AMD Am29F040B ($01/$A4)
- Bright BM29F040 ($AD/$40)

MaxFlash 1Mbit + MyIDE cartridge

The MaxFlash 1Mbit + MyIDE cartridge is similar to the 1Mbit cartridge, except with the banking address range moved and a MyIDE interface added. Banking is controlled by a read or write access to $D520-D53F instead of $D500-D51F.

The MyIDE interface maps the CompactFlash ATA registers at $D500-D507. Only the lower 8 bits of the data bus are exposed, so the CF device must be driven in 8-bit transfer mode.

8.3 Atarimax MyIDE-II

The Atarimax MyIDE-II cartridge is an advanced cartridge that contains 512KB of flash ROM, 512KB of RAM, and a CompactFlash interface.

For official programming information, consult the MyIDE-II Programming Information document on the Atarimax website: [MyIDE-II]

CompactFlash interface

The ATA-compatible registers of the CompactFlash interface are exposed by the MyIDE-II at $D500-D507. Only an 8-bit interface is provided. However, the alternate register set is also exposed, allowing access to the software reset facility of the CF device.

It is also possible to control power to the CF device, as well as sense when the CF device is changed. A green LED lights up whenever the CF device is powered.

Banking mechanism

Two independently controllable access windows are provided at $8000-9FFF and $A000-BFFF. The $A000-BFFF left cartridge bank is controlled by $D508 and the $8000-9FFF right-cartridge window by $D50A. Both registers are write-only. Bits 0-5 of the value written select an 8K bank, with bits 6-7 being ignored.

Unusually, the MyIDE-II also provides a third “keyhole” window at $D580-D5FF. It is mapped in 128 byte banks, selected by a pair of banking registers at $D50C (low byte) and $D50D (high byte).

An additional write-only control register at $D50F controls the mode for each of the banking windows.

Valid address ranges

The CF/IDE address range at $D500-D507 is only driven on read if the CompactFlash device is powered and active. If it is unpowered or held in reset state, this range will be undriven.

$D508-D50F return status information in bits 5-7, but bits 0-4 are undriven.

$D510-D57F is undriven.

$D580-D5FF is undriven if the keyhole window is disabled.

Registers

$D500-D507 CompactFlash control register window (read/write)

Exposes the main ATA control register set, or if bit 0 of $D50E is cleared, the alternate control register set.

$D508-D50F CompactFlash device status (read only)

CFP	CFR	CFP	Undriven

D7 CompactFlash device present sense

- 0 Not present
- 1 Present

D6 CompactFlash reset state

- 0 /RESET asserted
- 1 /RESET negated

D5 CompactFlash power state

- 0 Unpowered
- 1 Powered

Indicates whether a CompactFlash device is present and whether it is powered or in reset state. Note that bits 0-4 of the data bus are not driven when reading this address.

$D508 Left cartridge window banking register (write only)

Ignored	Bank

D5:D0 Left cartridge bank

Selects an 8K region for the left cartridge window out of 512K, from the memory type selected for the left cartridge window.

$D50A Right cartridge window banking register (write only)

Ignored	Bank

D5:D0 Right cartridge bank

Selects an 8K region for the right cartridge window out of 512K, from the memory type selected for the right cartridge window.

$D50C Keyhole window low banking register (write only)

Keyhole bank, bits 7-0

Controls bits 7-0 of the keyhole window bank, in 128 byte half-pages.

$D50D Keyhole window high banking register (write only)

Ignored	Keyhole bank, bits 11-8

Controls bits 11-8 of the keyhole window bank, in 128 byte half-pages.

$D50E CompactFlash control register (write only)

Ignored	CFP	CFA

D1 CompactFlash power control

- 0 Disable power and assert /RESET
- 1 Enable power

D0 CompactFlash alternate register select

- 0 Select alternate register set
- 1 Select main register set and deassert /RESET

Controls power to the CF device and which ATA/CF register set is active. Bits 0 and 1 also control the /RESET line to the CF device; it is automatically asserted when the device is powered on and deasserted when selecting the main register set.

$D50F Memory window control register (write only)

Left CTL	Right CTL	Ignored	Keyhole

D7:D6 Left cartridge window mode
D5:D4 Right cartridge window mode

- 00 Flash ROM
- 01 RAM, read/write
- 10 RAM, read-only

	11	Disabled
D1:D0	Keyhole window mode	
	00	RAM, read/write
	01	RAM, read-only
	10	Flash ROM
	11	Disabled

Note that the selection modes are encoded differently for the keyhole window than for the cartridge windows.

$D580-D5FF Keyhole window (read/write)

Accesses a 128-byte window of flash or on-board memory. If the keyhole window is disabled, reads from this region are not handled by the cartridge and return undriven bus data.

8.4 SIC!

The SIC!, or Super Inexpensive Cartridge!, is a flash ROM based cartridge which holds 128KB, 256KB, or 512KB.

Banking mechanism

The flash ROM is exposed via both the $8000-9FFF and $A000-BFFF windows, which are independently toggle-able but banked together. Banks are 16K with the $8000-9FFF window mapping the lower 8K of the bank.

The banking register is exposed at $D500-D51F, with the following contents:

- Bits 0-4: Selects 16K bank.
- Bit 5 = 1: Enables $8000-9FFF window.
- Bit 6 = 0: Enables $A000-BFFF window.
- Bit 7 = 1: Enables flash writes.

On power-up, the bank register is reset to $00.

Enable/disable switch

The enable/disable switch forces off the $A000-BFFF window when set to the disable side.

Flash types

- Winbond 29C020

8.5 SIDE 1 / SIDE 2

SIDE 1 is a cartridge with 512KB of flash memory and a CompactFlash interface. SIDE 2 revises the design with additional support for reading back banking register values, sensing CF removal, and signaling via an LED. The two versions are similar but not completely compatible.

Flash ROM

The flash ROM on the SIDE is used to emulate a pair of stacked cartridges, a pass-through SpartaDOS X (SDX) cartridge with a second (“top”) cartridge inserted above it. Both are independently bank switched through separate banking registers.

A physical switch on the cartridge enables or disables the SDX half, regardless of the SDX banking state.

The 4Mbit flash on the SIDE can be programmed in-place on the computer using standard flash ROM unlock and programming sequences.

CompactFlash interface

SIDE also includes a CompactFlash interface. The eight main parallel ATA compatible registers are exposed, as well as the CF reset signal.

Only the lower 8 bits of the CF data bus are exposed, so the CF device must be driven in 8-bit transfer mode.

Real-time clock

The RTC chip in the SIDE is a Maxim DS1305, which combines a real-time clock with battery backup and a 96 byte NVRAM. The DS1305 is accessed over an SPI bus through several control bits.

Register map

SIDE occupies a sparse set of addresses in the $D5E0-D5FF range. Table 14 shows the register layout. Grayed out entries indicate locations not handled by SIDE and ignored for reads or writes.

<table>
<tr><th></th><th colspan="2">SIDE 1</th><th colspan="2">SIDE 2</th></tr>
<tr><th>Address</th><th>Read</th><th>Write</th><th>Read</th><th>Write</th></tr>
<tr><td>D5E0</td><td rowspan="2"></td><td>SDX bank</td><td colspan="2"></td></tr>
<tr><td>D5E1</td><td colspan="2">SDX bank</td></tr>
<tr><td>D5E2</td><td>SPI sense</td><td>SPI control</td><td>SPI sense</td><td>SPI control</td></tr>
<tr><td rowspan="3">D5E4</td><td colspan="4" rowspan="3"></td></tr>
<tr><td>Cart bank</td><td colspan="2">Cart bank</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td>D5F0
D5F7</td><td colspan="2">IDE registers</td><td colspan="2">IDE registers</td></tr>
<tr><td>D5F8</td><td rowspan="3"></td><td rowspan="3">IDE reset</td><td>Signature</td><td>IDE reset</td></tr>
<tr><td>D5F9</td><td>Chg. sense</td><td>Reset + chg</td></tr>
<tr><td>D5FB</td><td></td><td>IDE reset</td></tr>
<tr><td>D5FC
D5FF</td><td>Signature</td><td></td><td>Signature</td><td></td></tr>
</table>

Table 14: SIDE 1/2 register map

SIDE 1 Registers

$D5E0 SDX banking register (SIDE 1 only, write only)

Controls banking for the SDX half of the cartridge.

Bits 0-5 select an 8K bank, while bit 7 disables the SDX half if set.

Bit 6 controls the top half of the cartridge, which is enabled if it is 0 and disabled if it is 1.

This register is set to $00 on power-up, enabling the SDX half and enabling the top cartridge. Pressing the menu button also does this.

$D5E2 SPI bus sense (read only)

Bit 3 reflects the state of the SPI bus input line from the RTC chip. Currently bits 0-2 and 4-7 are reserved, but are currently driven as 0 by the SIDE hardware and thus the register always reads as $00 or $08.

$D5E2 SPI bus control (write only)

Controls the three outgoing lines on the SPI bus to the RTC chip.

Bit 0 controls the chip enable and must be set to enable communication with the RTC chip.

Bit 1 controls the SPI clock.

Bit 2 controls the SPI outgoing data line.

$D5E4 Cartridge banking register (write only)

Controls banking for the top cartridge half.

Bits 0-5 select an 8K bank. The interpretation of bit 5 is inverted from the SDX half, so the halves of the flash ROM are flipped between the top half and the SDX half.

Bit 7 disables the top half if set. If cleared, the top half cartridge is enabled only if the pass through from the SDX half is also enabled via bit 6 of the SDX banking register.

This register is reset to $00 on startup, enabling the top half cartridge at bank 32.

$D5F0-D5F7 IDE hardware registers (read/write)

Parallel ATA register set, as exposed by the CompactFlash device.

$D5F8-D5FB IDE reset register (write only)

Bit 0 controls the reset signal to the CompactFlash device, where 0 resets the CF card, and 1 enables normal operation.

$D5FC-D5FF Signature bytes

These four locations hold the string “SIDE” ($53 49 44 45) if the SDX half is enabled by hardware switch and “ IDE” ($20 49 44 45) if it is not.

SIDE 2 Register Differences

$D5E1 SDX banking register (SIDE 2 only, read only)

This is the same as the SDX banking register in SIDE 1, except moved from $D5E0 to $D5E1 and made readable.

$D5E4 Top cartridge banking register (SIDE 2 only, read/write)

This register is read/write in SIDE 2, versus read-only in SIDE 1.

Also new to SIDE 2 is bit 6, which enables 16K banking if set. In that case, the top cartridge is mapped to $8000-BFFF in 16K banks instead of $A000-BFFF in 8K banks. Bit 0 of the bank number is ignored in 16K mode, although it is still stored and readable.

$D5F8 SIDE version detect (SIDE 2 only, read only)

Reads $32 to indicate a SIDE 2.

> **Warning**
>
> SIDE 1 does not respond to $D5F8 and reading that address will return bus data on systems with a floating data bus. This value can come from data read by ANTIC DMA, and therefore can read as $32 on a SIDE 1. This means that simply reading $D5F8 and comparing it to $32 can falsely detect a SIDE 1 as SIDE 2.
>
> One way to avoid this is ensuring that the floating bus data is a value other than $32. This can be done by preventing a DMA cycle prior to the read, thus making it likely that the last instruction byte will be returned instead ($D5), a value sufficiently different from $32. This can be ensured by reading in horizontal blank with interrupts or DMA off.

$D5F9 CompactFlash change status (SIDE 2 only, read only)

Bit 0 reads 0 when no change has been detected, and 1 when the CompactFlash card has been removed since the last time the change latch was reset. This latch stays set to 1, even after another CF card is inserted, until reset by a write to $D5F9.

$D5F9 CompactFlash change control/strobe (SIDE 2 only, write only)

Resets the CF change latch when written.

Bit 1 controls the status LED, which is normally turned on when bit 1 is cleared and turned off when it is set. The LED state is then inverted when the CF device is removed and the latch is set. The LED state is therefore determined by the XOR of the status LED bit and the change state.

Bit 0 of this register also controls the CF reset state, for compatibility with SIDE 1.

8.6 Corina

Corina is a hybrid cartridge with two configurations, 1MB of flash ROM or 512KB of flash ROM + 512KB of RAM. In addition, there is an EEPROM for persistent storage of small data.

Memory layout

Corina uses a 16K banking window at $8000-BFFF controlled by a single banking register at $D500. The banking register is write-only and values written into it are composed as follows:

7		0
DIS	Mode	Bank

D7 Enable/disable

0	$8000-BFFF window enabled
1	$8000-BFFF window disabled

D6:D5 Mapping mode

00	ROM banks 0-31
01	ROM banks 32-63 or RAM
10	EEPROM
11	Reserved

D4:D0 16K bank select (RAM/ROM only)

EEPROM

The EEPROM module provides 8KB of non-volatile storage, accessible when bits 5-6 of the banking register are set to %10. The NVRAM is directly writable; there is no special protocol necessary to access or unlock the EEPROM.

8.7 R-Time 8

The R-Time 8 is cartridge that adds a real-time clock to the computer. It has no firmware on-board and writes a software driver to be loaded externally. The real-time clock is provided by an M3002-16PI chip.

Register mapping

The R-Time 8 has 16 internal 8-bit locations mapped to a single read/write port, located at $D5B8-D5BF. The read/write port is only 4 bits wide, bits 0-3. Accesses are carried out by patterns of accesses:

- **Check status**: Read from initial state.
- **Read memory**: Write address, read high nibble data, read low nibble data.
- **Write memory**: Write address, write high nibble data, write low nibble data.

When the current step in the sequence is unknown, the M3002 can be reset to initial state by issuing two dummy reads.

> **Warning**
>
> The R-Time 8 only drives the lower four bits of the data bus. This means that depending on the computer, the upper four bits will either be 1111 or data from the floating data bus. All reads from the R-Time 8 should be masked to ignore the upper four bits.

Internal memory

The internal memory locations of the RTC chip are documented in the M3002 datasheet, but the important ones are as follows:

Location	Contents
$0	Seconds (0-59)
$1	Minutes (0-59)
$2	Hours (0-23)
$3	Day (0-31)
$4	Month (1-12)
$5	Year (0-99)
$6	Weekday (1-7)
$7	Week number (1-53)

All of these locations are read/write and stored as binary coded decimal (BCD).

8.8 Veronica

The Veronica cartridge adds a 65C816 coprocessor with 128K of RAM to the computer.[33]

Programming model

The 65C816 CPU runs in a dedicated memory address space, communicating with the 6502 solely by a semaphore bit and a 16K shared memory window. The 65C816 uses only 16-bit addressing with the bank address ignored, so it effectively runs as a 65C802; clock speed is 14MHz, 8x that of the main computer. The 6502 can reset the 65C816 at any time, but there is no support for interrupts.

Because of this design, the 65C816 cannot run programs directly on the computer; it is dependent upon the 6502 for bootstrap and for communication with main memory and hardware, including all of the custom chips (ANTIC, GTIA, PIA, and POKEY). However, there is no DMA contention either, so the 65C816 always runs at full speed.

There is no persistent storage on Veronica, and the cartridge powers up with both memory windows disabled. Therefore, bootstrap must occur from an external source, like disk.

Memory layout

64K of the memory is reserved exclusively for the 65816, of which nearly all is mapped directly, except for 16 inaccessible bytes shadowed by the hardware registers at $0200-020F in the 65C816's address space. The remaining 64K of memory is shared between the 65C816 and 6502 for communications purposes. It is split into two banks of 32K, of which the 65C816 sees one and the 6502 sees the other, and which can be swapped between them. In addition, each 32K bank is split into 16K halves, where both the 6502 and 65C816 can independently choose whether to access the bottom or top half.

On the 65C816 side, the 16K shared memory window can be placed at either $4000-7FFF or $C000-FFFF. The window is always enabled at one of the two possible locations. The location of the window and which 16K half of the 32K bank is selected are controlled by the hardware register at $0200-020F.

The 6502 can similarly map a 16K half of its 32K bank to the cartridge windows at $8000-9FFF and $A000-BFFF. One 16K half is selected at a time and each window is mapped to a fixed 8K section of it, but the two 8K windows can be independently enabled or disabled. Bank, half, and window selection are controlled by a hardware register at $D5C0.

33 The author would like to thank the Veronica team for permission and assistance in publishing technical information about the Veronica cartridge.

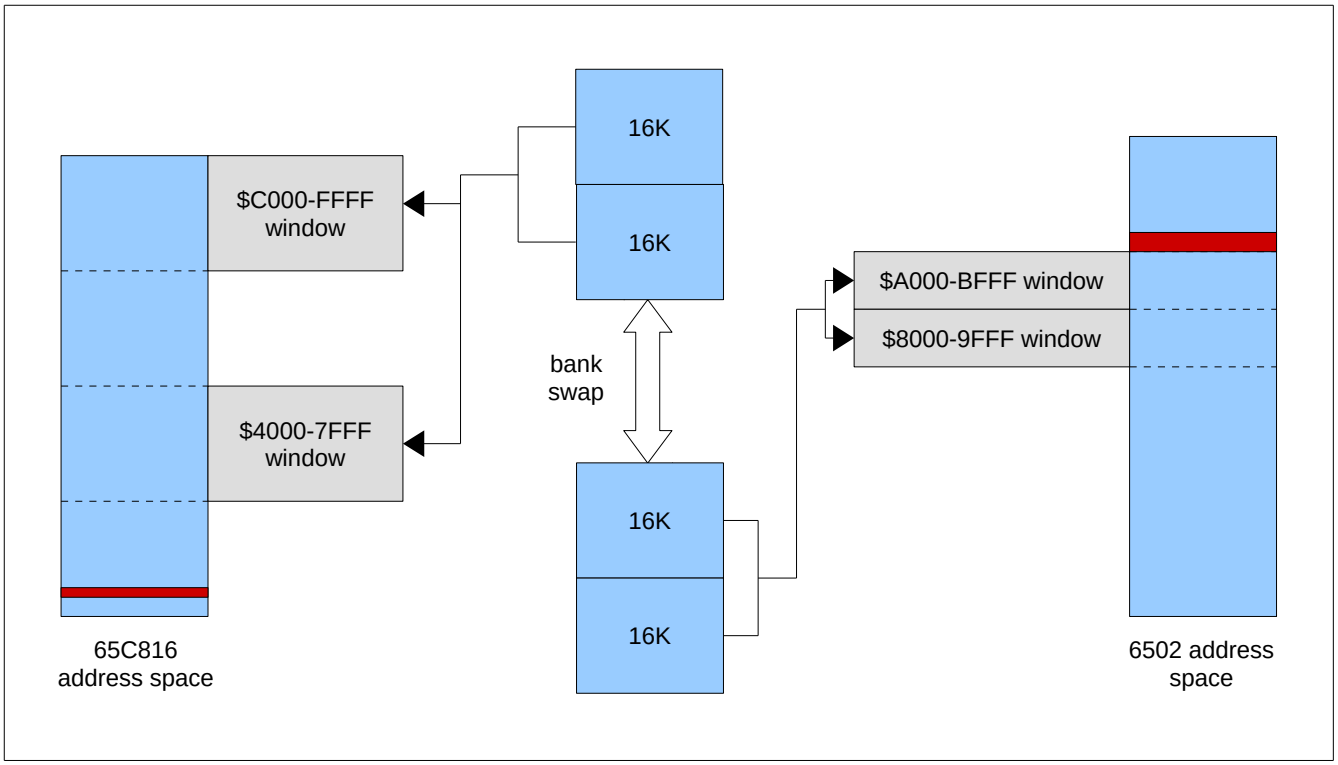


Figure 6: Veronica memory layout

A significant aspect of this design is that the 65C816 and 6502 never see the same memory at the same time. Both can access all of the memory, but each 32K bank is always exclusively accessed by one side and the two halves can only be swapped. Furthermore, while 16K halves can be independently chosen on each side, the 32K bank swap can only be triggered by the 6502.

Semaphore signaling

In addition to the shared memory windows, Veronica also contains a single shared semaphore bit between the two CPUs, exposed as the high bit (bit 7) of the respective hardware control registers. A change to the semaphore bit on one side is immediately reflected on the other side.

> **Warning**
>
> The multiprocessor environment created by Veronica poses some unique challenges for synchronization and requires special care when writing communications code between the CPUs. In particular, read-modify-write instructions like INC/DEC that would ordinarily be safe with interrupts on the 6502 are **not** safe when communicating between the 6502 and the 65816, because one CPU can swap memory banks or read the semaphore in the middle of an instruction being executed by the other. The 6502 does not support atomic memory primitives at all and the 65C816 on Veronica is not hooked up for doing so, so communications protocols must be written with this in mind.

Hardware registers

$0200-020F Veronica control register (65C816 side; read/write)

7							0
SEM	WIN	HLF	Reserved				

D7 Shared semaphore
- 0 (default)

D6 Window address
- 0 $C000-FFFF (default)
- 1 $4000-7FFF

D5 Bank half select
- 0 Use 16K half A of 32K bank
- 1 Use 16K half B of 32K bank (default)

D0-D4 Reserved (reads as 1)

The 65C816 controls its portion of the Veronica hardware by a single register, mirrored at $0200-020F. This register exposes the shared semaphore bit as well as banking window selection and bank half selection. Its value is reset to $3F on power-up or soft reset.

$D5C0 Host control register (6502 side; read/write)

7							0
SEM		WNA	WN8	BNK		SWP	RES

D7 Shared semaphore
- 1 (default)

D6 Reserved (reads as 1)

D5 $A000-BFFF window enable
- 0 Disable $A000-BFFF window (default)
- 1 Enable $A000-BFFF window

D4 $8000-9FFF window enable
- 0 Disable $8000-9FFF window (default)
- 1 Enable $8000-9FFF window

D3 Bank half select
- 0 Use 16K half A of 32K bank
- 1 Use 16K half B of 32K bank (default)

D2 Reserved (reads as 1)

D1 Bank swapping
- 0 6502 uses bank A, 65816 uses bank B (default)
- 1 6502 uses bank B, 65816 uses bank A

D0 Soft reset
- 0 Disable 65816 – hold in reset state (default)
- 1 Enable 65816

On the host side, Veronica is controlled via this single register at $D5C0. Three of the bits affect the 65C816 – bit 0 (reset), bit 1 (swap banks), and bit 7 (semaphore); the remainder control memory mapping on the 6502 side. At power-up this register is set to $CC.

By default, Veronica powers up with the 65C816 held in reset. This also resets the memory configuration on the 65C816 side so that the memory window is at $C000-FFFF and viewing the upper 16K half of the memory bank. The 6502 must upload bootstrap code into its window $A000-BFFF and then swap memory banks before or while turning off soft reset, so that the 65C816 can begin executing the bootstrap code at $E000-FFFF in its memory space through the reset vector at ($FFFC).

> **Caution**
>
> The semaphore bit is inverted between the 6502 and the 65816. A 0 on the 6502 side is reflected as a 1 on the 65816 side, and vice versa.

Hardware versions

There are two versions of the Veronica hardware. Version 1 use a three RAM chip design, where one chip supplies the main 64K of memory while the other two chips have the 32K swappable bank memory, and runs the 65C816 asynchronously to the main computer clock. Version 2 uses a single multiplexed 128K RAM and runs the 65C816 synchronously at 8x the main clock.

Both versions of the hardware have the same hardware registers and programming model. However, there is one significant difference. The V1 hardware swaps window banks without synchronization with the 65C816 side, meaning that a memory access to the banking window in progress during the swap can be corrupted. This means that reliable operation on V1 hardware requires excluding the 65C816 from the banking window whenever the 6502 swaps banks. V2 hardware does not have this limitation since it runs the 65C816 and swaps banks synchronously.

Chapter 9
Serial I/O (SIO) Bus

The serial I/O (SIO) bus is the main data bus for peripherals and supports cassette tape decks, disk drives, printers, communication devices.

9.1 Basic SIO protocol

Data bus connection

The serial data input and output lines are connected to the serial port lines on POKEY, and therefore all data transfers require manipulating POKEY's serial port. This also results in a standard data format of one start bit, eight data bits, and one stop bit with no parity. The normal communication rate is 19,200 baud, although this varies for device-specific commands.

SIO control lines

The CB2 control line on the PIA is connected to the command line on the SIO bus and is used to tell peripherals that a command frame is being sent. It is active low, so it is normally high and then dropped low during the command frame. The high-to-low transition tells the device that a command frame is starting; the low-to-high transition signals to the device that the computer is ready to receive the response.

The CA2 control line connects to the motor control line on the SIO bus to activate a cassette tape recorder. It is also active low and enables the cassette motor when lowered.

SIO interrupt lines

Two control lines on the SIO bus are rarely used but allow peripherals to interrupt the main computer CPU. The Proceed line is connected to the PIA's CA1 input, whereas the Interrupt line is connected to the CB1 input. Enabling these interrupts in the PIA control registers will cause the IRQ handler to be invoked on demand. The 1030 Direct Connect Modem is a peripheral that uses this functionality.

Command Frame

Because multiple devices can be connected to the SIO bus, a standard command sequence is necessary to address a specific device. This is done by lowering the command line and sending a five byte frame at 19200 baud. The five bytes are:

- Device ID
- Command
- Auxiliary byte 1
- Auxiliary byte 2
- Checksum

The device ID indicates the device being addressed. Table 15 lists some device IDs used.

$31-3F	Disk drive (D:)
$40-43	820 printer (P:)
$45	Atari Peripheral Emulator (APE)
$46	AspeQt
$4F	Type 3/4 poll
$50-53	850 serial device (R:)
$5F	Cassette tape (C:) (virtual OS device)
$6F	DOS2DOS (PCLink)

Table 15: SIO device IDs

Device $5F is special as it is a virtual device used by the OS to represent the cassette device, intercepted by the SIO routines and routed to special cassette code. DDEVIC=$5F is what is intercepted, not the final SIO bus device ID of $5F. The actual ID $5F is therefore unused, but unavailable for use by a physical device.

The command byte indicates the command being issued, with command codes specific to each device. The auxiliary bytes provide command parameters.

At the end of the frame is the checksum, which is a simple 8-bit carry wrap-around checksum. It can be computed by initially clearing the 6502 carry bit and then adding each data byte in sequence with ADC, followed by folding the carry bit back in with ADC #0. The command frame is valid if the carry wrap-around checksum of the four command bytes is equal to the checksum byte.[34]

Not all devices use command frames. Cassette tape recorders are dumb devices and only use the motor control line, not interpreting any commands on the bus; the 1030 modem uses simple command bytes at 300 baud.

Command Protocol

In addition to interpreting command frames, a standard intelligent SIO device also follows a specific protocol for command execution. All command and data transfers are normally at 19200 baud with no parity. This proceeds as follows[35]:

1. Command frame
    - The host lowers the command line to indicate the start of a command.
    - A delay of 750µs-1600µs is introduced for the peripheral to notice the command line state.
    - The five-byte command frame is sent.
    - Another delay of 650µs-950µs is introduced for the peripheral to finish receiving the command. (Note that the minimum bound on this is violated by the OS; peripherals should assume no minimum delay.)
    - The command line is raised.
    - The peripheral checks the command frame, and ignores it if the command is intended for another peripheral or if a framing or checksum error has occurred.
2. Command acknowledgment
    - The peripheral initially checks the command code and command parameters for validity. This may

34 This is also known as a one's complement sum and there are interesting properties that can be used to accelerate its computation, such as associativity. See RFC1071 for an extended discussion of optimization opportunities.

35 See [ATAXL] section 9 for the official SIO protocol description.

take up to 16ms.

- If the command is valid, it quickly sends back a $41 ('A') byte to acknowledge a valid command. If it is invalid, a $4E ('N') or NAK byte is sent back.

3. Data frame from computer
   - If the command requires a data frame to be sent from the computer, it is now sent at this time at 19200 baud between 10-18ms after the ACK byte is received from the device. The length is command dependent. A carry wrap-around checksum byte is included at the end.
   - The peripheral has 850µs-16ms to process and validate the data frame.
   - If the data frame is received correctly, a $41 ('A') or ACK byte is sent by the peripheral. Otherwise, a $4E ('N') or NAK byte is sent and the command is aborted. No complete/error byte is sent if a NAK is sent.
4. Command execution
   - The peripheral now executes the command. The amount of time taken varies and can be significant, anywhere from milliseconds for a status command to several hundred milliseconds for a read sector command and over a minute for formatting a disk.
5. Command result
   - A delay of at least 250µs is required after the ACK byte before indicating the command result. Note that this delay is dead time, i.e. from the *end* of the ACK byte to the *beginning* of the result byte.
   - If the command completed successfully, a $43 ('C') or Complete byte is sent by the peripheral. Otherwise, a $45 ('E') or Error byte is sent.
6. Data frame from peripheral
   - For commands that send data back to the computer, the peripheral now sends a data frame. A carry wrap-around checksum is sent at the end. Note that for commands that return data, this frame is sent even if an error status ($45) was returned.
7. End of command
   - The command is now completed and another command may be issued on the bus.

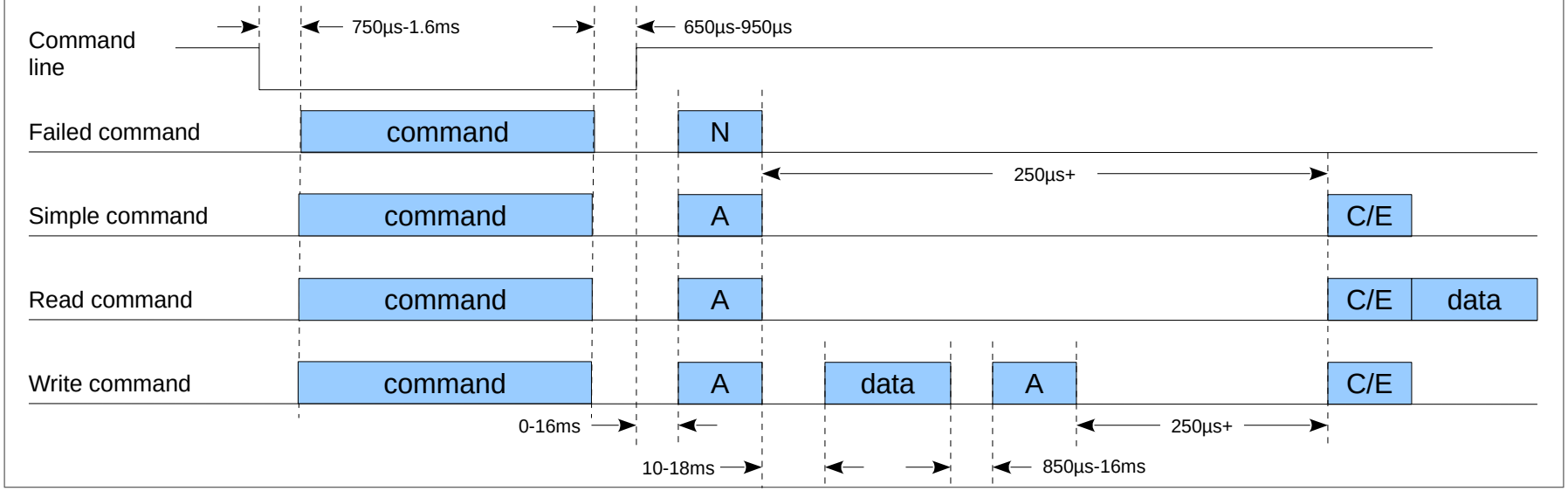


Figure 7: SIO command timing

Parameter	Source of delay	Timing
Assert command line to start of command frame	Host	750µs-1.6ms
Command frame (5 bytes)	Host	Varies (~2.6ms)
End of command frame to deassert command line	Host	650µs-950µs
Deassert command line to start of ACK/NAK	Peripheral	0-16ms
ACK/NAK byte	Peripheral	Varies (~520µs)
End of ACK/NAK to start of write data	Host	10-18ms
Write data	Host	Varies (~68ms for 128b sector)
End of write data to start of ACK	Peripheral	850µs-16ms
ACK byte	Peripheral	Varies (~520µs)
Execute operation	Peripheral	250µs to device timeout
Complete/Error byte	Peripheral	Varies (~520µs)
End of C/E byte to read data	Peripheral	Not specified (may be zero)
Read data	Host	Varies (~68ms for 128b sector)

9.2 Polling

The host computer can poll the SIO bus to automatically discover and download handlers from each device. This is done by sending polling commands out onto the bus and checking if any devices respond.

Type 0 Poll

A Type 0 Poll is essentially a disk boot – a request for sector 1 on D1:, which is then interpreted as a disk boot sector and results in consecutive sectors being read from disk. A peripheral can emulate a “disk” drive in order to satisfy a Type 0 Poll.[36] A peripheral can delay responding to a sector read until a number of consecutive get status requests have been received in order to ensure that any real disk drive at the D1: address has a chance to respond first.

The 850 Interface Module responds to Type 0 Polls.

Type 1 Poll

Command $3F (?) is used to perform a Type 1 Poll.[37] AUX1 and AUX2 are not used, and the command returns 12 bytes if successful. These 12 bytes are the device control block (DCB) to be used with the OS SIO to read in the bootstrap loader starting at $0500, which is then invoked by a JSR to $0506.[38] This is similar enough to a disk boot that the same image can be used for both. The 850 Interface Module responds to this type of poll.

DOS 2.0S's default AUTORUN.SYS responds to a Type 1 Poll, but it has a quirk that requires the device bootstrap to follow a few rules:

- Only one device's handler can load, as only one successful poll is handled.
- As noted, the entry point to the bootstrap routine must be at $0506.
- The bootstrap routine must hook (DOSINI) such that the next call to it does *not* chain through to DOS.

Another shortcoming of a Type 1 Poll is that a device will only respond affirmatively to it once, and then never again until it is power-cycled.

Type 2 Poll

Command $3F is also used for Type 2 Polls.[39] However, no other information is available about them.

Type 3 Poll

Type 3 Polls are documented in the XL Addendum and use both address $4F and command $40 (@) together.[40] Unlike a Type 1 Poll, Type 3 Polls allow the host to restart the process and re-poll peripherals without requiring them to be power-cycled, and also stream the handler in a standardized relocatable form instead of needing the peripheral to handle relocation itself. The host is required to use address $4F and command $40, but devices are allowed to only check for the command. This means that command $40 is globally reserved across *all* device IDs.[41]

The Poll Reset command resets all peripherals and restarts the polling process. It is issued with AUX1 and AUX2

36 [AHS05] p.8
37 [AHS05] p.10
38 The $0506 start address is, in fact, hardcoded by DOS 2.0's AUTORUN.SYS.
39 [AHS05] p.10
40 [ATAXL] p.22
41 [AHS05] p.9

set to $4F. No peripherals are supposed to actually reply to a Poll Reset – it is simply sent blind.

AUX1/AUX2=$00 is the main poll command. Every peripheral that responds to a Type 3 Poll responds to a unique retry of this command, i.e. one device may respond to the initial command, while another device might only respond to the 4th. This requires that each device count off the number of consecutive times this command has been received. There are 26 slots available since that is the number of attempts from the OS.

On a successful poll, the device returns a four byte payload with the following data: handler size (low byte), handler size (high byte), device ID, and version ID. The device ID can then be used to address the device directly to load the handler. The handler size must be even.[42] Afterward, the device remembers the successful poll so that it doesn't respond to it again until a poll reset is issued.

A Null Poll is issued by setting AUX1/AUX2=$4E. This effectively serves as a no-op command that does nothing but restart a polling sequence, resetting the retry counters for each device.

The XL/XE OS issues a Type 3 Poll after boot and before run.

Type 4 Poll

A Type 4 Poll is similar to a Type 3 Poll except that it is triggered by a request for a specific device.[43] The address and command are the same – $4F/$40 – but AUX1 contains the device name and unit number together in ATASCII. For instance, a request for H: or H1: would have $48/$01 in the AUX bytes.

The XL/XE OS issues a Type 4 Poll on an attempt to access a nonexistent CIO device.

Type 3/4 Handler Loading

When a device has been successfully found via a Type 3 or Type 4 Poll, the handler is then loaded from the device directly using the download command $26, using the device ID from the poll. AUX1 is set to a block number, starting at zero and counting up as many blocks as necessary according to the handler size. On success, a 128 byte payload is returned. Invalid block numbers can result in either a NAK or lack of response.[44]

Type 3/4 Handler Format

The blocks of handler data returned by the download command, when reassembled, form a stream of data to load and relocate a 6502 machine program. This module consists of two main relocatable sections, a zero-page section and a non-zero-page section, as well as optional non-relocatable sections at absolute addresses. The program is assembled from records in the stream, with each record defined by a leading type byte. All records except for the end record are followed by a length byte, which includes everything except the type and length bytes.

Text records (record types $00, $01, and $0A)

A text record delivers up to 255 bytes of a section. $00 is for the non-zero page section, $01 is for the zero-page section, and $0A is for a non-relocatable section. The payload consists of a 16-bit offset from the beginning of the section, followed by the section data.

Relocation records (record types $02-09)

Relocation records are used to adjust references within sections to point to the final location of those or other sections. The references in the section data initially contain the offset from the beginning of the target section; the address of the target section is added to produce the final referenced address.

There are eight types of reference:

42 [AHS05] p.10
43 [ATAXL] p.23
44 [AHS05] p.11

Token	Referencing section type	Target section type	Reference type
$02	Non zero page	Non zero page	Low byte of word address
$03	Zero page		
$04	Non zero page	Zero page	Byte address
$05	Zero page		
$06	Non zero page	Non zero page	Word address
$07	Zero page		
$08	Non zero page		High byte of word address
$09	Zero page		

Table 16: Peripheral Handler Relocation Record Types

The locations of the references to be relocated are specified in the payload as byte offsets from the beginning of the last text record. This means that relocation records need to be interleaved with text records and that word addresses should not be split across text record boundaries.

Record types $08 and $09 are special as they adjust references consisting of only the high byte of the target address. This is an unusual relocation type and allows handlers to be relocated anywhere in memory, not just to page boundaries. For these types, the offset data consists of pairs of bytes instead of single bytes, where the first byte of each pair is the offset and the second byte is the low byte of the reference offset. The low byte in the relocation data is combined with the high byte in the text record to form the reference offset, which is then added to the section address to produce the target address. The high byte of this target address is then written back to the section.

End record (record type $0B)

The last record in the handler is type $0B, which signifies the end of the relocation stream. Unlike the other record types, no length byte is included. Instead, the $0B token is followed by three bytes:

- Self-start byte: $00 for no self start, $01 to automatically invoke an absolute address, and $02 to invoke a relocated address within the non-zero-page section.
- 16-bit start relative or absolute start address.

9.3 850 Interface Module

SIO addressing

The RS232 ports are addressed using the SIO addresses $50-53.

Status command

The status command ($53 = 'S') returns error and control state information from the 850 controller; AUX1 and AUX2 are ignored. Two bytes are returned, the first of which contains error state and the second of which contains control line state:

First status byte:

7							0
FRA				OPT	RDY	SER	CMD

D7 Framing error detected
D3 Invalid option
D2 Not ready (monitored control line inactive)
D1 Bad SIO data frame
D0 Invalid command

0 No error detected
1 Error detected

Second status byte:

7				0
DSR	CTS	CRX	0	RCV

D7:D6 DSR state
D5:D4 CTS state
D3:D2 CRX state

00 Always low since last check
01 Currently low, but was high at some point since last check
10 Currently high, but was low at some point since last check
11 Always high since last check

D0 RCV state

0 Space
1 Mark

Write command

The write command ($57 = 'W') is used to send data to the 850 controller for transmission. The AUX1 byte of the command frame specifies the number of bytes in the data payload from 0 to 64 bytes, while AUX2 is ignored. If AUX1 is zero, the data payload portion of the SIO sequence is skipped.

Regardless of the value in AUX1, the data frame is always padded to 64 bytes.

The 850 controller does not issue a (C)omplete response until the entire block has been sent.

Control command

The control command ($41 = 'A') corresponds to the R: handler's XIO 34 and is used to modify the outgoing control lines.

AUX1:

7							0
DTRc	DTR	RTSc	RTS			XMTc	XMT

D7 Enable DTR (Data Terminal Ready) change
D5 Enable RTS (Request To Send) change
D1 Enable XMT (Transmit) change

0 No change
1 Change state

D6 New DTR state (if D7 set)
D4 New RTS state (if D5 set)
D0 New XMT state (if D1 set)

0 Negate / space
1 Assert / mark

Stream command

Sending the command $58 ('X') switches the 850 controller into streaming mode, which corresponds to concurrent mode on the R: handler. AUX1 specifies the I/O direction, while AUX2 is ignored:

AUX1:

7							0
						R	W

D1 Read enable

0	Read from 850 direction disabled
1	Read from 850 direction enabled

D0 Write enable

0	Write to 850 direction disabled
1	Write to 850 direction enabled

If the current word size for the channel is anything other than 8-bit, the I/O direction must be input only and the baud rate must be 300 baud or less, or the command will fail.

The returned data payload for the stream command consists of nine bytes to be written to $D200-D208 (AUDF1-AUDCTL) to configure POKEY for the correct baud rate during the transfer. Afterward, the controller starts operating in streaming mode.

Streaming mode causes the controller to reflect between the Atari SIO bus and the serial port. During streaming, no commands can be sent to the controller, and in particular, it is not possible to read the control line status. The controller exits streaming mode the next time that the command line is asserted.

Configure command

The configure command ($42 = 'B') corresponds to the R: handler's XIO 36. It sets the baud rate, word size, stop bits, and control signal monitoring.

AUX1:

7			0
2SB		Word size	Baud rate

D7 Stop bits

0	1 stop bit
1	2 stop bits

D4:D5 Word size

00	5 bits
01	6 bits
10	7 bits
11	8 bits

D0:D3 Baud rate

0000	300 baud
0001	45.5 baud
0010	50 baud
0011	56.875 baud
0100	75 baud
0101	110 baud
0110	134.5 baud
0111	150 baud
1000	300 baud

1001	600 baud
1010	1200 baud
1011	1800 baud
1100	2400 baud
1101	4800 baud
1110	9600 baud
1111	19200 baud

AUX2:

7			0
	DSR	CTS	CRX

D2 Watch DSR (Data Set Ready) line
D1 Watch CTS (Clear To Send) line
D0 Watch CRX (Carrier Ready) line

0 Ignore control line
1 Block attempts to write block or start streaming when control line is negated

Type 1/2 Poll command

Command $3F polls the SIO bus for devices with automatically loadable handlers. The 850 responds to this command in one of two ways. For the standard poll with AUX1=$00, it responds once to the very last (26th) attempt. It always responds to AUX1=$01. The result of the command is a 12-byte DCB to use with the SIOV vector to retrieve the booter/relocator, which is then invoked at $0506.[45] This program then loads, relocates, and initializes the handler at MEMLO, after which MEMLO is raised to above the handler.

Booter/relocator download command

Command $21 (!) loads the booter/relocator from the device; AUX1/2 are ignored. The booter/relocator is returned in a single block, so the size must be known beforehand. This command is usually not issued directly, but according to the DCB returned by the poll command. One ROM version returns 342 ($0156) bytes from this command, meaning that $0500-0655 must be available to bootstrap the 850.

Handler download command

Command $26 (&) is used to load the peripheral handler from the device; AUX1/2 are ignored. Unlike the similar command used by Type 3/4 Polls, the 850 does not return the handler in blocks. Instead, it is returned as a single large block. This means that the handler size must be known beforehand. This command is usually not issued directly, but automatically by the booter/relocator. One ROM version returns 1496 ($0592) bytes from this command.

9.4 1030 Modem

The 1030 modem is an SIO bus peripheral that allows phone line based communications at 300 baud with Bell 103 modem compatible modulation. [46]

Data protocol

When a connection is active, the computer and the 1030 modem exchange data directly on the bus at 300 baud, without using command or data frames. This means that the SIO bus is exclusively dedicated to the modem during an online connection unless suspended using the $5A (“Z”) command.

45 As noted earlier, DOS II's AUTORUN.SYS hardcodes $0506 as the start address, so this can be relied upon.
46 The author would like to acknowledge the help of mr-atari and AtariGeezer in recording and analysis of the 1030 hardware, particularly the bootstrap process.

Interrupts

The 1030 modem is one of the rare devices that uses the SIO proceed and interrupt control lines. The proceed line is used to signal completion of a command, while the interrupt line indicates the carrier detect state. Both are intended to drive the PIA interrupt facility in order to assert IRQs on the 6502.

Command protocol

Unlike other SIO peripherals, the 1030 uses a non-standard protocol for communication commands on the SIO bus. All commands are sent as single characters at 300 baud with the command line asserted. Presumably, any other peripherals on the bus would ignore such commands as they would encounter framing errors attempting to interpret the sent data at 19200 baud.

Code	Command	Description
$48 ('H")	Send break signal	
$49 ("I")	Set originate mode	Switches to the originating modem set band (1270Hz/1070Hz).
$4A ("J")	Set answer mode	Switches to the answering modem set band (2225Hz/2025Hz).
$4B ("K")	Begin pulse dial	Take phone off hook and prepare to pulse dial. Bytes received in the range of $01-$0A are interpreted as the number of times to pulse the phone line.
$4C ("L")	Pick up phone (off hook)	
$4D ("M")	Hang up phone (on hook)	
$4F ("O")	Begin tone dial	Take phone off hook and connect POKEY audio to phone line.
$50 ("P")	Start 30 second timeout	Wait up to 30 seconds for carrier.
$51 ("Q")	Reset modem	
$57 ("W")	Set analog loopback test	Turns on analog echo so that transmitted data is received.
$58 ("X")	Clear analog loopback test	Turns off analog echo.
$59 ("Y")	Resume modem	Stops transmission of received data across SIO bus.
$5A ("Z")	Suspend modem	Resumes transmission of received data across SIO bus.

Table 17: 1030 Modem hardware commands

Tone dialing

There is no direct support on the 1030 modem itself for tone dialing. Instead, DTMF tones are generated using POKEY on the main computer and the audio output is then conducted onto the phone line.

Bootstrap support

The 1030 supports both disk emulation for a full bootstrap and a separate command for handler download only.

Disk emulation is done similarly to the 850, where the 1030 will emulate get status and read sector commands for D1: after a number of unanswered command retries. The 1030 responds with an 810-like status response of $00 00 E0 00 and a single boot sector. The code within this boot sector then does two things: it bails out silently (C=0) if a T: device is already present, and then it loads the ModemLink software from the modem. ModemLink is downloaded to $0C00-33FF using SIO address $58, command $3B.

The T: handler embedded within the on-board ModemLink software can also be downloaded by itself using

address $58, command $3C, AUX $00/$00, and a receive length of $B30. This retrieves a handler to be placed at $1D00-282F, with its initialization routine at $1D0C.

9.5 SX212 Modem

The SX212 modem is a Hayes-compatible 1200 baud modem with both SIO and RS-232 connections. Like the XM301 and unlike the 1030, it has no on-board software and a T: handler or communications program must be loaded externally.

Command set

The SX212's command interface is handled by a Sierra Semiconductor SC11008 Stand-Alone Modem Interface Controller. The following commands are supported:

A/	Repeat last command
ATA	Answer (go off hook)
ATB	Set Bell modulation mode*
ATC	Set transmit carrier
ATD	Dial
ATE	Set echo
ATF	Set full duplex
ATH	Set on/off hook
ATI	Information
ATL	Speaker loudness*
ATM	Speaker mode
ATO	Originate (go off hook)
ATP	Set pulse dial mode
ATQ	Set quiet mode
ATR	Set reverse mode
ATS	Set or query register
ATT	Set touch dialing
ATV	Set verbose reporting
ATX	Set connect/busy/dialtone reporting
ATY	Set long space disconnect enable*
ATZ	Reset modem

Table 18: SX212 supported commands

Commands marked with an asterisk (*) are ones that are documented in the SC11008 datasheet and supported by the SX212, but not documented in the Atari SX212 Modem Owner's Manual.

Escape guard

Like truly Hayes-compatible modems, the SX212 requires appropriate guard time before and after the +++

escape sequence to recognize it as such. This is set in register S12 in 1/50th second units and defaults to one second. Attempting to use the time-independent escape sequence (TIES) used by some modems without appropriate delays does not work.[47]

Version information

The ATI0 and ATI1 commands emit decimal numbers for the product code and firmware checksum of the modem, respectively. On at least one modem, these values are 134 and 103.

SIO motor control line

Since the standard SIO command frame protocol is not used by the SX212, the motor control line is used instead to indicate when communication with the modem is desired. Data is only received by and sent from the SX212 when the motor control line is asserted low.

Conflicts on the motor control line are avoided because the SX212, like the 410 and 1010 Program Recorders, only have one SIO connector and are designed to only be used at the end of the SIO chain.

SIO proceed/interrupt lines

The SX212 indicates the carrier detect and high speed states to the computer through the SIO proceed and interrupt signals, respectively. These match the indicator lights such that the proceed line is low whenever CD is on and the interrupt line is low when HS is on. Changes in these signals are then exposed to software via connections to the CA1 and CB1 inputs on the PIA.

Because the PIA only allows edge detection on the CA1/CB1 inputs, an initialization sequence is necessary for the handler software to ascertain the current state of the SIO proceed and interrupt lines before it can begin tracking changes in the CD/HS states.

Speed auto-switching

Commands can be sent to the modem at either 300 or 1200 baud. When the character is received in command mode, the modem will automatically switch to the correct speed. This flips both the state of the high speed (HS) indicator and the SIO interrupt line. The character that triggers this switch is still recognized and processed correctly.

9.6 R-Verter

The R-Verter is a small adapter cable that connects an RS-232 serial device to the SIO bus.

Motor control line

The standard SIO protocol is not used by the R-Verter; the motor control line is used instead to enable communications. When the motor control line is asserted (motor on), full-duplex transmit/receive and DTR sensing are enabled; when it is negated, the R-Verter disconnects from the bus.

Communication parameters

The R-Verter does not have a UART or UART-like controller and simply adapts the SIO bus to an RS-232 connector. This means that serial framing and timing are determined by POKEY, while parity and stop bit settings are driven by software.

47 Not only does Atari's manual not document this either, the Atari SX212 modem handler tries to emit +++AT as part of its initialization sequence without delays, even though the hardware is configured by default to require guard times.

Carrier Detect and Data Set Ready sensing

The Carrier Detect (CD) and Data Set Ready (DSR) pins of the RS-232 connector are hooked up to the Proceed and Interrupt lines on the SIO bus, respectively, with inversion. This means that an active CD/DSR line pulls CA1/CB1 low.

9.7 810 Disk Drive

The 810 disk drive adapts a 5.25” floppy disk drive to the Atari SIO interface. Disks are formatted as single-sided, single-density with 18 128-byte sectors on 40 tracks, for a total of 720 sectors and 90K of storage. The disk geometry is abstracted from the computer so that the disk appears as a linear store of sectors numbered from 1-720 with simple read/write commands. Motor control and seeking are handled automatically by the drive.

Commands

Commands are sent to the 810 using a standard SIO frame at 19200 baud, and follow usual conventions for ACK/NAK/Complete/Error.

Attempts to issue unknown commands or commands with bad arguments – a sector number of 0 or above 720 – results in a NAK being sent for the original command. A valid command that fails because of a disk I/O error returns an ACK for the original command followed by an ERROR code for the command result, and then a data frame if one is expected for the command.

Status ($53)

The status ('S' = $53) command is used to query the status of the 810 disk drive. The AUX1 and AUX2 bytes of the command frame are ignored. In response, the drive sends back a four byte status block:

- Drive status
  - Bit 4 = 1: Motor running
  - Bit 3 = 1: Failed due to write protected disk
  - Bit 2 = 1: Unsuccessful PUT operation
- Floppy drive controller status (inverted from FDC)
  - Bit 6 = 0: Write protect error
  - Bit 5 = 0: Deleted sector (sector marked as deleted in sector header)
  - Bit 4 = 0: Record not found (missing sector)
  - Bit 3 = 0: CRC error
  - Bit 2 = 0: Lost data
  - Bit 1 = 0: Data pending
- Default timeout ($E0 = 224 vertical blanks)
- Unused ($00)

Read ($52)

The read ('R' = $52) command reads a 128 byte sector from the disk. The AUX1 and AUX2 bytes of the command frame hold the LSB and MSB, respectively, of the sector to read. On completion, the drive returns 128 bytes of sector data. This occurs even in the event of a read error.

If the sector number in AUX1/2 is 0 or greater than 720, the command is immediately NAKed.

Put ($50)

The put ('P' = $50) command writes a 128 byte sector to the disk, without verification. The AUX1 and AUX2 bytes of the command frame hold the LSB and MSB, respectively, of the sector to read, and 128 bytes of sector data are sent by the computer following acknowledgment of the command frame.

If the sector number in AUX1/2 is 0 or greater than 720, the command is immediately NAKed.

Write ($57)

The write command ('W' = $57) command is the same as the put command, except that it also re-reads the sector afterward to verify a successful write.

If the sector number in AUX1/2 is 0 or greater than 720, the command is immediately NAKed.

Format ($21)

The format command ('!' = $21) command formats a disk, writing 40 tracks and then verifying all sectors. On completion, the drive returns a 128 byte buffer containing a list of 16-bit bad sector numbers, terminated by $FFFF.

Transmission timing

The 810 does not have dedicated serial port hardware and instead uses its CPU to bit-bang data in and out of the serial port using exact cycle timed code. Its CPU is a 6507 running at 500KHz. The transmit routine takes 26 cycles per bit, for a transmit rate of 19231 baud, and 265 cycles per byte, producing a read sector tone on the Atari of 943Hz and a read rate of 1887 bytes/second.

Track layout and timing

The disk rotates in an 810 drive at a rate of 288 revolutions per minute (RPM) or 4.8 revolutions per second. Each track is split into 18 sectors, so each sector arrives under the head at a rate of at least 86 sectors/second. The sectors are not necessarily distributed evenly, so they will tend to arrive a bit faster than that.

The floppy disk controller (FDC) in the 810 is clocked at 1MHz. Internally, this clock is divided by four to produce a 250KHz bit clock, which is then divided by two again for separate clock and data bits to produce a data rate of 125Kbits/second or 26,042 bits per track. For each sector, there are 128 data bytes, 28 bytes of header and CRC overhead, and then a 10 byte gap between each sector. This nominally places sectors 1328 bits (10.6 ms) apart. A 12 byte track header and an additional 256 pad bytes fill out the track.

Because of significant transfer delays, the 810 formats tracks with non-sequential sectors to reduce rotational latencies. This includes the time to read the sector from the disk, and more significantly, the time to transfer the sector to the computer at 19,200 baud. This takes 95ms, during which 9 sectors will pass under the disk head. If the sectors were written out in order, the next sector would already have been passed, requiring another disk rotation for the sector to arrive again. This is known as “blowing a rev” and reduces the disk read rate to less than one sector per revolution. Instead, the 810 formats tracks with all odd sectors first and then all even sectors so that the next sector soon arrives under the head when the computer is issuing back-to-back read or write sector requests. The result is that two sectors can be read in a bit more than one revolution instead of just one, for an effective read rate within a track of about 1,170 bytes/sec or 11,700 baud.

Reading or writing sectors on another track also incurs seek delays. The 810 seeks at a rate of 5.3 ms/track, followed by an additional 10ms of head settling time at the end of the seek.

Disk anomalies

Abnormal sectors on the disk will trigger unusual behavior from the 810 disk drive. These can result from a corrupted disk, or they can be intentional in order to make a disk harder to copy. Copy protection mechanisms depend on the 810 responding to abnormal sectors in specific ways.

CRC error

The data payload of each sector is protected by a Cyclic Redundancy Check (CRC), which is a 16-bit code that is computed from the data and is written along with it. On read, the CRC is recomputed and verified against the written version to check if the data was read successfully. A mismatch indicates data corruption.

When a CRC error occurs, the 810 returns Error status instead of Complete status, but still returns the sector data. Bit 3 is also cleared in FDC status to indicate a CRC error.

Missing sector

If a sector cannot be found on a track, the 810 will make a couple more attempts to find it before giving up and returning Error. A sector's worth of data is still returned. FDC status bit 4 is cleared to indicate the failure.

Deleted sector

Each sector includes a data address mark byte that indicates the start of the sector data. The DAM is normally a modified $FB byte with some clock pulses missing, but the FDC also supports a modified $F8 byte to indicate a “deleted” record. While the FDC considers this normal and reads the sector successfully, the 810 considers it an error and will retry the read. Upon failing out, it will send back the deleted sector's data with an Error status. FDC status bit 5 is cleared to indicate a deleted sector.

Long sector

The FDC used by the 810 supports 256, 512, and 1024 byte sectors as well as the 128 byte sector that the 810 uses. With a long sector, the 810 firmware will stop after reading 128 bytes even though the FDC continues to read remainder of the sector. When the firmware fails to read the remaining data, the FDC asserts the Lost Data status bit (bit 2). Bit 1 (DRQ) will also be asserted due to the pending data. These are reflected as bits 1 and 2 being cleared in the returned FDC status due to the inverted FDC bus. As usual, the first 128 bytes of the sector are returned along with the Error status.

Weak sector

Weakly recorded or unrecorded data regions will appear as noise to the FDC. This results in random sector data, which virtually guarantees a CRC error.

Phantom sectors

Multiple sectors within the same track can have duplicate sector IDs, in which case any of them may be found by the FDC. The phantom sector found depends on when the read command is issued and how long it takes for the firmware to issue the corresponding command to the FDC. Note that the firmware will attempt to retry up to two times on an error, which may mean that the first phantom sector found is not the one that is returned.

The delays in processing the command and reading a physical sector affect the rotational timing, which in turn determines the phantom sector that is read. Ideal timings for an 810 drive running revision E firmware at 500KHz, with a drive mechanism running exactly at 288 RPM, no seeks required, and motor already on are given in Table 19.

Event	Time	Rotation
Send command at 19200 baud	2.63 ms	4.54º
Deassert command line	?	?
Command line deasserted to ACK byte sent	0.59 ms	1.02º
ACK byte sent to FDC command issued	3.22 ms	5.56º
Rotational delay	0-208.3 ms	0-360º
Read physical sector	9.66 ms	16.69º
Compute checksum and return sector data	74.06 ms	127.98º

Table 19: Ideal 810 sector read timing

Not counting additional delays for host-side processing, it takes a minimum of 90.16 ms (155.8º) to read a sector. For this reason, sectors need to be placed at nearly opposite sides of a track for the 810 to read them back-to-back.

9.8 1050 Disk Drive

The 1050 disk drive is a double-density drive capable of storing 130K on a diskette instead of the 90K of the 810. This is done by using double-density MFM encoding, allowing 26 128-byte sectors to be stored per track instead of 18, for a total of 1040 sectors. The stock 1050 does not support 256 byte MFM sectors.

Status command

The status command in the 1050 returns one additional bit of information: bit 7 indicates the disk encoding. If bit 7 is cleared, the disk is formatted as single density (18 sectors per track), whereas if it is set, the disk is formatted as enhanced density (26 sectors per track).

Format Medium Density command

The format medium density ($22 = "") command formats a disk using double-density encoding instead of single density. It otherwise operates similarly to the original $21 format command.

Transmission timing

Like the 810, the 1050 also uses its CPU to bit-bang data across the serial port, but it has a faster 1MHz processor. The transmit routine takes 51 cycles per bit and 549 cycles per byte, producing a transmit rate of 19608 baud, a read sector tone of 911Hz, and a read rate of 1822 bytes/second. This results in the computer producing noticeably lower tones when reading from a 1050 versus an 810.

Seek timing

The 1050 steps by half tracks a time since it uses an 80 track mechanism. The step rate is 10ms, giving 20ms per track, followed by a 20ms head settling delay.

Seek anomaly

A strange behavior of the 1050 is that it has slightly longer seek times when seeking forward to higher-numbered tracks than when seeking backward. When seeking forward, the 1050 does an additional half-track step forward, followed by a half-track step backward. This ensures that the 1050 always finishes a seek with a backward step, for better consistency in head positioning.

Formatted track layout

Medium density disks are formatted by 1050 with a 13:1 interleave and about a two sector skew between tracks. This means on each track even and odd sectors are segregated, and in terms of angular position sector 1 on track 2 is approximately where sector 5 is on track 1.

9.9 XF551 Disk Drive

The XF551 disk drive adds true support for double-density disks, with 720 sectors of 256 bytes each. It also spins the disk at 300 RPM instead of 288 RPM, resulting in slightly lower rotational latencies.

High speed transfers

The high bit of a command byte can be set to request high speed transfers. When this is set, the initial ACK or NAK byte is sent at 19,200 baud, and then the transmission rate between the computer and the XF551 for the remainder of the command is raised to 38,400 baud. This includes any following ACK, Complete, Error, checksum, and data frame bytes in either direction, but it does not include the command frame itself which must always be sent at standard speed. Any command may be executed in high speed except for disk format commands.

Status command

The status command ($53) returns additional status on the XF551. The first byte indicates the following:

- Bit 7 is set if and only if the disk is formatted as enhanced density (MFM, 26 sectors of 128 bytes per track). It is cleared for SD and DD formats.
- Bit 6 is set if the disk is formatted as double sided.
- Bit 5 is set if the disk is formatted as double density (MFM, 256 bytes per sector).
- Bit 4 is set if the disk motor is active.
- Bit 3 is set if the disk was write protected on the last write operation.
- Bit 2 is set if the last write operation failed.
- Bit 1 is set if the last data frame was received incorrectly.
- Bit 0 is set if the last command was invalid.

The timeout field in the returned status is set to $FE on the XF551 instead of $E0.

Format With High-Speed Skew command

Issuing the format command with the high bit set ($A1) instructs the XF551 to do a format using a sector skew better suited to high-speed transfer rates, using a 9:1 interleave instead of a 15:1 interleave. Because of the reuse of bit 7, this is not a high-speed command and the data frame is sent at low speed. The high-speed skew only pertains to double density formats and does not affect single density or enhanced density formatting.

Read PERCOM Block command

Command $4E ('N') reads a PERCOM configuration block from the drive. This corresponds to either the last detected format or the last format selected by the Write PERCOM Block command, whichever is more recent. The XF551 always returns one of the following four configurations as an 12 byte payload:

Data	Index	Format			
		Single density	Enhanced density	SSDD	DSDD
Track count	**0**	40			
Step rate	**1**	$00 (6 ms/half track)			
Sectors per track	**2-3**	18	26	18	18
Sides minus one	**4**	0	0	0	1
Recording method	**5**	$00 (FM)	$04 (MFM)	$04 (MFM)	$04 (MFM)
Bytes per sector	**6-7**	128	128	256	256
Drive status	**8**	$01 (online)			
Reserved byte	**9**	$41			
Reserved bytes	**10-11**	$00			

Table 20: XF551 PERCOM configuration blocks

Note that all 16-bit quantities in the PERCOM block are stored in big-endian order with the high byte first, backwards from traditional 6502 convention.

Write PERCOM Block command

The PERCOM block can also be modified using command $4F ('O'), which receives the PERCOM block as a 12 byte payload. This is used to set the desired format for a subsequent format command. The XF551 does not validate or interpret the entire PERCOM block, however, and does the bare minimum of checks needed to distinguish its supported formats:

- If the sectors per track count is 26, extended density is selected.
- If the bytes per sector count is less than 256, single density is selected.
- If the sides minus one value is zero, single sided double density is selected.
- Otherwise, double sided double density is selected.

Track count, step rate, recording method, and drive status are always ignored.

Transmission timing

The XF551's 8040 CPU runs at 8.333MHz, giving a machine cycle rate of 555KHz. The transmission loop runs at a rate of 29 cycles per bit (19157 baud) and 290 cycles per byte (1915.7 bytes/second). In high-speed mode, this is accelerated to 14 cycles/bit (39683 baud) and 140 cycles/byte (3968.3 bytes/second).

9.10 410/1010 Program Recorder

The 410 and 1010 Program Recorders are cassette tape recorders with a connection to the SIO bus.

Motor control

The SIO motor control line enables the 410/1010 recorder motor under computer control, allowing the tape to be stopped on demand. The Play or Record button must also be depressed on the 410 for the motor to activate.

The motor control line also serves as the voltage source for the audio track, cutting out the audio when the motor is deactivated. A side effect of this is that rapid toggling of the motor control line will be reflected as audio on the computer when the 410/1010 is connected.

Data decoding and encoding

During playback, the data track is processed by two bandpass filters centered at ~4kHz and ~5KHz. The amplitudes of the two filters are compared and the result is sent across the SIO bus, with the 4KHz filter producing a 0 and the 5KHz filter producing a 1. This decodes the frequency shift keying (FSK) encoding used to record data onto the tape.

When recording, the SIO bus data is recorded directly onto the tape. This is normally done with two-tone mode with timers 1 and 2 clocked with the 64KHz clock with divisors of 6 and 8 (AUDF1=5 and AUDF2=7), giving 5327Hz and 3995Hz as the two tones.

Turbo modifications

A popular modification to Atari cassette tape recorders involves adding a turbo mode that bypasses the FSK bandpass filters. This allows for higher data rates to be recorded onto the tape, at the cost of additional complexity to manually modulate the signal. This is often activated by lowering the command line.

9.11 MidiMate

The MidiMate provides MIDI capability over the SIO bus.

MIDI communications

MIDI messages are sent at 31250 baud, which is not a rate that POKEY can hit normally with sufficient accuracy. Therefore, the MidiMate provides an external clock for POKEY to use, by dividing down a 4MHz crystal by 128. This is then selected by software using the external input/output clocking modes in the SKCTL register. Software is responsible for following the MIDI protocol for sent messages.

Sync input

An external source can be connected to the MidiMate for timing synchronization purposes. This is connected to the SIO bus interrupt line, which in turn connects to the CB1 input on the PIA.

Enable/disable

The SIO motor control line is used to enable the MidiMate. When asserted, the external serial clock and sync input are enabled; otherwise, they are disconnected from the SIO bus.

Chapter 10
Parallel Bus Interface

10.1 Introduction

The parallel bus interface (PBI) was added with the XL series of computers for greater expansion capabilities. Unlike the cartridge port, the PBI allows for wider address ranges and use of interrupts.

The XE series has a similar but different expansion port called the enhanced cartridge interface (ECI), which combines with the cartridge port to provide PBI-like capabilities. The capabilities of the ECI are similar enough to PBI that adapters can be used to make the same hardware work on both.

10.2 Common memory map

Address regions

The address pages $D1xx, D5xx, and D6xx are reserved for the currently active PBI device. In addition, $D1FF is used as a shared control location.These ranges are unused by the main computer, so when not driven by the PBI device, reads return undriven bus data.

The math pack region at $D800-DFFF is also used as a PBI firmware window.

Device select register

PDVS [$D1FF] is the hardware select register for PBI devices. Each bit corresponds to an individual device, where setting a single bit to 1 selects that device and writing $00 deselects all devices. Only one bit should be set at a time. The response to selection is device dependent but typically involves overlaying the math pack at $D800-DFFF with device-specific firmware ROM.

The presence of the selection register at $D1FF is by convention, encouraged by the support in the XL/XE OS; it is not directly implemented or decoded by the computer and must be implemented in the PBI device. This means that devices can vary in its implementation. The Black Box, for instance, only partially decodes its address and overloads it with other device-specific control bits.

Device IRQ status register

PDVI [$D1FF] is also the address of the shared IRQ status register. A '1' bit in this register indicates that a device is requesting an interrupt. Only the bits corresponding to present devices are pertinent and other bits may have undefined values.

The procedure for acknowledging a PBI device IRQ is device specific and must be done by the device firmware.

Like PDVS, devices connected to the PBI bus are not obligated to implement PDVI according to the PBI standard.

PBI address region

The address range $D600-D7FF is reserved for PBI device addressing and can be used for RAM, ROM, or I/O of an actively selected PBI device.

PBI memory map overlays

The MMU allows the PBI device to overlay RAM, but not any I/O, or cartridge address space. ROM also cannot be overlaid, except for the math pack region at $D800-DFFF which can be swapped out for PBI device ROM through a Math Pack Disable (MPD) signal.

10.3 ICD Multi I/O (MIO)

The Multi I/O (MIO) device from ICD, Inc. adds SCSI, parallel printing, and RS-232 port capability through the PBI.

Register map

The MIO occupies the $D100-D1FF and $D600-D6FF regions of PBI address space:

	Read	Write
D6FF D600	MIO RAM	
D1FF	Status 2	Control 2
D1FE	Status 1	Control 1
D1FD	SCSI/printer data latch	
D1FC	Reset SCSI bus strobe	RAM bank A8-A15
D1FB D1E0	Mirrors of $D1FC-D1FF	
D1DF D1C0	6551 ACIA	
D11F D100	Unused	

Table 21: MIO memory map

$D1FC Reset SCSI bus (read-only strobe)

Reading from this register asserts the reset line on the SCSI bus. The value read is undefined.

$D1FC RAM bank A8-A15 (write-only)

Controls the eight of the bank address bits for the MIO memory window at $D600-D6FF. The banking value cannot be read back.

$D1FD SCSI/printer data latch (read/write)

Reads or writes data on the SCSI bus. The data is inverted for SCSI and non-inverted for the parallel printer. Bus driver direction is controlled by the SCSI I/O line state.

Accesses to this register also cause the MIO hardware to automatically acknowledge the byte on the SCSI bus, if handshaking is in effect (+REQ > +ACK).

$D1FE Status register #1 (read-only)

REQ	PBS	BSY			I/O	MSG	C/D

D7 SCSI REQ state

0 Data transfer requested

D6 Printer BUSY state
- 0 Printer not busy
- 1 Printer busy

D5 SCSI BSY state
- 0 $D600-D6FF RAM window disabled
- 1 $D600-D6FF RAM window enabled

D2 SCSI I/O state
- 0 Input (transfer from target to initiator/host)
- 1 Output (transfer from initiator/host to target)

D1 SCSI MSG state
- 0 Message is being transmitted

D0 SCSI C/D state
- 0 Asserted: command, status, or message being transferred
- 1 Negated: data being transferred

Reading this register also clears the SCSI reset state, if it was triggered by a read from $D1FC.

$D1FE Control register #1 (write-only)

PIQ	PST	RAM	SEL	RAM bank A16-A19

D7 Printer BUSY IRQ enable
- 0 IRQ disabled
- 1 IRQ enabled

D6 Printer STROBE
- 0 Asserted: New data valid
- 1 Negated: End of strobe

D5 RAM enable
- 0 $D600-D6FF RAM window disabled
- 1 $D600-D6FF RAM window enabled

D4 SCSI MSG signal
- 0 Asserted
- 1 Negated

D0:D3 RAM bank A16-A19

$D1FF Status register #2 / PBI interrupt status (read-only)

			IRQ	ACI	DSR	CTS	DCD

D4 IRQ status
- 0 ACIA or printer IRQ pending

D3 Printer interrupt status
- 0 Printer IRQ pending

D2 RS232 Data Set Ready (DSR) state
- 1 DSR asserted

D1 RS232 Clear To Send (CTS) state
- 0 Negated: Device requesting hold-off
- 1 Asserted: Device allowing data

D0 RS232 Data Carrier Detect

0	Negated: No modem carrier detected
1	Asserted: Modem carrier detected

Bits 3 and 4 reflect the status of the printer and PBI IRQ lines, respectively. These are not latches, and disabling the respective IRQs will cause these bits to change to 1 immediately.

$D1FF Control register #2 / PBI select (write-only)

		ROM bank		

D5:D2 ROM bank select

0000	No bank selected (disable ROM)
0001	Select bank 0 (canonical)
0010	Select bank 1 (canonical)
0011	Select bank 0
0100	Select bank 2 (canonical)
0101	Select bank 0
0110	Select bank 0
0111	Select bank 0
1000	Select bank 3 (canonical)
1001	Select bank 0
1010	Select bank 1
1011	Select bank 0
1100	Select bank 2
1101	Select bank 0
1110	Select bank 0
1111	Select bank 0

This register controls the selected ROM bank as well as whether the I/O light is active – it is lit up whenever any ROM bank is selected. Only one bit is supposed to be set at a time, selecting one of four 2K banks for a total of 8K of firmware ROM.

SCSI data handshaking

SCSI requires handshaking through a pair of REQ/ACK lines to transfer data bytes over the bus. The target first asserts REQ to begin a transfer, the initator (host) asserts ACK in turn to indicate that it has read or written a byte, the target negates REQ to acknowledge the ACK, and the initator negates ACK to complete the transfer. This allows data transfers to be throttled appropriately to accommodate delays on either end.

Because this handshaking protocol is expensive to implement in software, the MIO does this automatically in hardware. A read or write to the data latch [$D1C1] automatically asserts ACK if REQ is asserted, and the hardware automatically deasserts ACK whenever REQ is deasserted.

Printer busy IRQ

The MIO also has support for interrupt-driven printer spooling. Bit 7 of $D1FE enables an IRQ whenever the printer is not busy and can accept another byte. Like the serial output complete IRQ in POKEY, this IRQ is not latched. If the printer reasserts BUSY due to receiving another byte while this IRQ is enabled, the IRQ will deassert by itself.

10.4 CSS Black Box

The Black Box by Computer Software Services (CSS) is a device that provides SCSI, parallel printer, RS-232 port, and screenshot functionality over the PBI bus.

Register map

The Black Box occupies the $D100-D1FF and $D600-D6FF regions of PBI address space:

D6FF D600	Black Box RAM

D1FF D1C0	Status register (read only) Control register (write only)
D1BF D180	6521 PIA
D17F D150	6522 VIA
D13F D120	6551 ACIA
D11F D100	Unused

Table 22: Black Box memory map

The control register is designed to be compatible with the OS definition of the PBI select register:

DCD	CTS	DSR	MSG	ROM bank

D7 RS232 data carrier detect (DCD) signal

0 Negated
1 Asserted

D6 RS232 clear to send (CTS) signal

0 Negated
1 Asserted

D5 RS232 data set ready (DSR) signal

0 Negated
1 Asserted

D4 SCSI MSG signal

0 Asserted
1 Negated

D0:D3 ROM bank

0000 None selected
other ROM selected

Similarly, the status register is compatible with the OS definition of the PBI interrupt status register:

Not used	MNU	SS	VIA	ACIA

D3 Menu button

0 Button depressed
1 Button released

D2 Screenshot button

0	Button depressed
1	Button released

D1 VIA interrupt status

0	No interrupt pending
1	VIA interrupt pending

D0 ACIA interrupt status

0	No interrupt pending
1	ACIA interrupt pending

6522 VIA connections

VIA port A is connected to both the SCSI and printer buses. Although the SCSI bus has inverted data compared to the standard 6502 bus, an inverting bus transceiver is used so that the stored data is non-inverted. This means that the printer output is inverted instead, however.

VIA port B is used for several miscellaneous signals:

- D0=0 (input/output): SCSI Input/Output signal asserted. This also controls the direction of the printer/SCSI data bus driver, where 1=output.
- D1=0 (input/output): SCSI Command/Data signal asserted
- D2=0 (output): SCSI SEL signal asserted
- D3=0 (output): SCSI RESET signal asserted
- D4=1 (input): Printer busy
- D5=0 (input): Printer fault (overridable by DIP switch #1)
- D6=0 (input/output): SCSI BUSY signal asserted
- D7=0 (input/output): SCSI REQ signal asserted

The VIA's CA1 and CA2 signals are used for handshaking on the SCSI bus, with REQ on CA1 and ACK on CA2.

VIA CB1 is used as a switch indicator and is pulled low when either the menu or screen dump buttons are depressed. It is normally configured to generate an IRQ on a negative transition.

VIA CB2 is used to drive the printer strobe line and is driven low at least 0.5µs to indicate that a new valid byte is on the printer bus.

6521 PIA connections

Port A is used in output mode to select the RAM bank that appears at $D600-D6FF. Bits 0-6 are used, for a total of 32K RAM addressable.

Port B bits 0 and 1 are used to read the graphics/text and hard drive write protect switches. Both are pulled down to 0 (grounded) when activated.

Port B bits 2-7 are connected to DIP switches #2-7, where a switch that is ON pulls the corresponding port line down to a 0. DIP switch #8 is unused. For 32K firmware, port B bit 2 is re-purposed as a high bank select bit and DIP switch #2 must be turned off for normal operation.

CA2 is connected to RS-232 RTS (Request To Send), while CB2 is connected to DTR (Data Terminal Ready). CA1 and CB1 are not connected.

> **Caution**
>
> The Black Box has its PIA wired differently than the base computer. The main PIA in the computer has the address lines swapped, whereas the Black Box wires it conventionally. This means that the order of the four registers is port A, control A, port B, and control B on the Black Box instead of port A, port B, control A, control B.

Firmware ROM

The original version of the Black Box has 16K of ROM mapped in 2K banks at $D800-DFFF. Banks 1, 2, 4, and 8 correspond to PBI devices seen as the OS, and therefore must contain valid entry points. In addition, banks 1 and 2 service the ACIA and VIA interrupts, respectively. Bank 0 disables the ROM.

The I/O light on the Black Box is also tied to the bank select and will light up whenever the ROM is active.

On units with 32K of ROM, there are an additional 7 banks of ROM selectable via PIA port B bit 2. Bank 8 is not accessible.

RAM

Base versions of the Black Box also have 8K of scratch RAM mapped in 256 byte banks through $D600-D6FF. PIA port A bits 0-5 is used to select the bank. With 32K RAM, bits 0-7 are used. Finally, 64K RAM adds PIA port B bit 1 as an 8th RAM bank select bit.

SCSI hard disk interface

The Black Box exposes its interface to SCSI hard disks through its 6522 VIA and status registers. The pertinent connections are as follows:

- VIA port A connects to the SCSI data bus. The hardware handles inversion of data written to or read from this bus, so the 6502 sees non-inverted data.
- VIA port B bits 0, 1, 6 and 7 are used in both input and output mode for I/O, C/D, BUSY and REQ. All are active low. Port B bit 0 (I/O) also controls the direction of the data bus transceiver, where 1 = output.
- VIA port B bits 2 and 3 are used in output mode to drive SEL and RESET.
- VIA control signals CA1 and CA2 are used for SCSI handshaking signals REQ and ACK.

All handshaking and control signals are connected to the VIA as active low. The 6502 largely has to drive the entire SCSI protocol in software, with the notable exception of the REQ and ACK signals. Those are connected to allow the VIA to be configured in hardware handshaking mode, automatically driving one of the signals whenever the 6502 reads data from or writes data to port A.

> **Caution**
>
> The Black Box's data latch is inverted from the MIO's. The MIO has SCSI data inverted and printer data non-inverted, whereas the Black Box has SCSI data non-inverted and printer data inverted.

Printer interface

The parallel printer port interface is driven entirely through the VIA:

- VIA port A is used to send inverted printer data.
- VIA port B bit 0 is pulled high in order to switch the SCSI/printer bus transceiver to output mode.
- VIA port B bit 4 is used in input mode to sense the printer BUSY line (1 = busy).
- VIA port B bit 5 is used in input mode to sense a printer fault (0 = fault). DIP switch #1 overrides this signal to 1 (no fault) if set. This is done directly in hardware; the switch cannot be read directly.

- VIA CB2 is connected to STROBE and is momentarily driven low to indicate to the printer that a new data byte is available.

Serial (RS-232) interface

Serial communication primarily uses the 6551 ACIA. It is connected with a 1.8432MHz crystal for standard baud rate generation, which means that the ÷16 external clock mode is inoperative. The ACIA can generate IRQs and status register bit 0 is set when this occurs.

None of the control lines are hooked up to the ACIA, so hardware handshaking is not possible. Instead, the DSR, CTS, and DTD are read through status port bits 5-7, and RTS and DTR are driven through PIA CA2 and CB2. All are active high.

Chapter 11
Internal devices

11.1 Introduction

Internal devices are ones that are installed inside of the computer instead of connecting via the peripheral ports. With direct access to the address/data buses as well as other internal signals, they can add functionality in ways not possible through even PBI/ECI.

In this chapter, discussion will be limited to add-ons that are not simply internal versions of devices, with the same behavior as the external version, i.e. internal SDX.

11.2 Covox

A “Covox” interface is a very simple way to get higher quality digital sound output than is possible through POKEY. It consists solely of a latch to capture data off of the data bus and an digital-analog converter (DAC) to convert it to an audio waveform.

Note that Covox interfaces are not standardized, so individual interfaces may differ slightly in implementation.

Programming interface

The Covox interface is typically assigned an address range such as $D600-D6FF, $D700-D7FF, or $D280-D2FF. Writes to any address in the range change the signal level, which is specified as an unsigned 8-bit sound sample. Reads are not handled, which means that it is not normally possible to detect a Covox interface.

The audio sampling rate is determined by the timing of writes from the CPU, which must write to the interface at regular intervals. The sample changes immediately upon a write, so writes that change the sample value must be spaced to reduce jitter.

Multi-channel output

A Covox interface can be extended to stereo or 4-channel output by including two or four latch+DAC pairs. The A0 or A1-A0 address lines are then used to select the channel. For the four channel case, the channel order is arranged to match the Amiga, so that channels 0+3 route to the left channel and 1+2 route to the right.[48]

11.3 Ultimate1MB

Ultimate1MB, or U1MB for short, is a multifunction device installed internally to the computer that provides a number of expansion functions, some of which are not possible externally:[49]

- Up to 1MB of extended memory
- Selectable OS, BASIC, and game ROM images
- Computer-flashable firmware
- SpartaDOS X cartridge emulation
- Soft-enable and address decoding for other expansion peripherals
- Parallel Bus Interface device emulation

These features are primarily achieved by taking over the sockets used for the MMU and OS ROM.

For official programming information: [U1MB].

48 It is also valid to route the other way, with 0+3 to right and 1+2 to left. Apparently, Commodore couldn't get this consistently right in either the Amiga hardware manual or case markings.

49 The author would like to thank Candle for providing Ultimate1MB technical information and hardware for testing.

CPLD revisions

There are two versions of the Complex Programmable Logic Device (CPLD) that drives the U1MB. Revision 1 contains the bulk of the functionality, but revision 2 adds support for Parallel Bus Interface (PBI) device emulation. The revision 1 CPLD is flash upgradable and can be updated to revision 2, although this cannot be done from the computer and requires an external CPLD programmer.

Flash ROM

Much of the U1MB's functionality is a 4MBit (512K) flash device, which substitutes for all ROM in the system. It is directly mappable in 8K banks through the 8K read/write cartridge window and indirectly mappable as read-only through the BASIC, OS, and PBI address ranges.

Five memory ranges within the flash ROM have designated functions in the hardware. These include the bootstrap ROM at $50000-53FFF, the OS ROM banks at $70000-7FFFF, the BASIC ROM banks at $60000-67FFF, the game ROM banks at $68000-6FFFF, and the PBI ROM banks at $58000-5FFFF. The remainder of the flash ROM has designated areas for mapping through the cartridge window, but these usages are not required by the hardware design.

The flash ROM is programmable in-place from the computer side as long as flash writing is enabled via bit 7 of UAUX [$D381]. When this is enabled, *all* memory windows that have flash ROM exposed also handle flash writes. This means, for instance, that it is possible to enter autoselect mode through writes to the OS ROM at $F555 and $FAAA. There is no reset facility for the flash ROM, so if the computer is reset while the flash ROM is in a programming or query state, the computer will fail to run the BIOS and a power cycle is required to recover.

When the flash ROM is write protected via bit 7 of UAUX [$D381], all writes to the flash ROM are blocked. This not only prevents inadvertent erasure or programming of the flash ROM, but also prevents entry into command or autoselect mode.

U1MB has shipped with multiple types of flash ROM which vary in significant ways. Early versions shipped with an Amic A29040 ($37/$86), whereas some newer models shipped with different devices such as an SST39SF040 ($BF/$B7). This distinction is important due to the variation in sector size (64K vs. 4K) and in sector programming sequences.

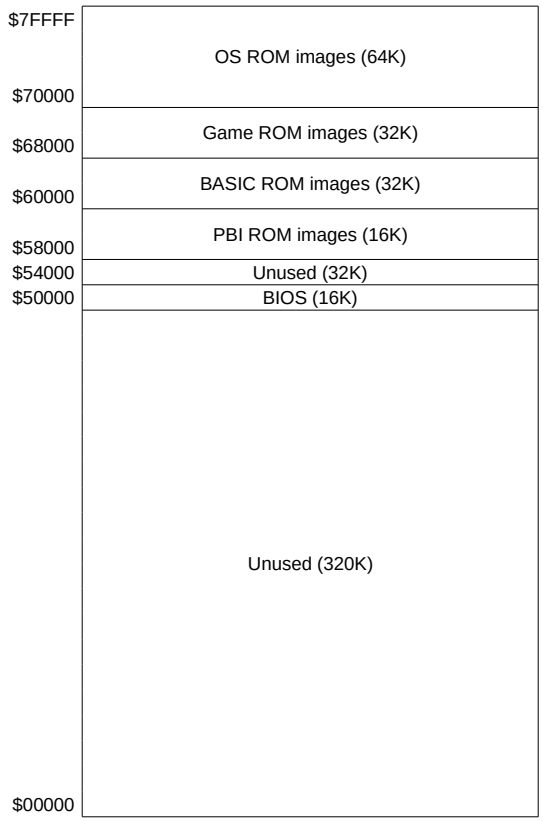


Figure 8: Ultimate1MB flash memory map

> **Warning**
>
> Because there is only one flash ROM chip in Ultimate1MB, **all** flash ROM based mappings visible to the 6502 will change when the flash ROM mode is changed. This includes the OS ROM, BASIC ROM, game ROM, self-test ROM, and internal cartridge ROM. Therefore, entering autoselect mode through writes to the cartridge window will cause the OS ROM image to vanish, as $C000-CFFF and $D800-FFFF will return manufacturer and device code data instead of flash array data.

BIOS ROM

On power-up, the BIOS ROM at $50000-53FFF in the flash ROM is mapped as the computer's bootstrap OS ROM. This is laid out in standard XL/XE OS ROM order, so $50000-50FFF is mapped to low OS ROM at $C000-CFFF, $51800-53FFF is mapped to high OS ROM at $D800-DFFF, and $51000-517FF supplies the self-test ROM at $5000-57FF when enabled. The BIOS ROM handles U1MB initialization and configuration before handing off control to one of the four OS ROM slots at $70000-7FFFF in flash. The BIOS ROM is re-enabled and regains control on a reset.

Normally, the XL/XE OS has to detect whether a reset sequence corresponds to a cold reset or a warm reset by the presence of signature bytes in RAM. The BIOS need not do this as the U1MB provides a hardware register bit to indicate a cold boot. Bit 7 of COLDF [$D383] is set on power-up and can be cleared under software control to indicate a warm reset the next time the reset vector is invoked.

OS, BASIC, and Game ROM

The flash ROM also contains four image slots each for the 16K OS and 8K BASIC ROMs present in a stock

XL/XE, as well as the 8K game ROM additionally present in an XEGS. One of the four images can be independently selected of each type by the BIOS.

The game ROM is enabled by PORTB bit 6. It is only enabled if the U1MB is configured in XEGS mode by hardware jumper.

Cartridge control

U1MB provides sufficient cartridge emulation facilities to run a version of SpartaDOS X (SDX). The flash ROM can be enabled in the left cartridge window at $A000-BFFF in 64 banks of 8K each, controlled by $D5E0. Although the lower half of flash ROM is assigned to cartridge emulation, this window actually allows all 512K of the flash ROM to be mapped and doubles as the window for updating flash ROM.

The upper two bits of the SDX control register allow toggling of both the internal and external cartridges: either can be enabled or both can be disabled. However, only the $8000-9FFF and $A000-BFFF windows of the external cartridge can be controlled; the cartridge control (CCTL) region at $D500-D5FF is always enabled. TRIG3 sensing is emulated so that it is asserted when either the internal or external cartridges are mapping $A000-BFFF.

Note that when enabled, the banking register overlaps with the CCTL region. The U1MB does not exclude this address from CCTL, meaning that a write to $D5E0 is handled **both** by U1MB and the external cartridge. A consequence of this is that U1MB's internal cartridge pass-through is incompatible with SIDE 1 as a write to this address will change the SIDE 1 bank even if the external cartridge is disabled. There is no issue if the internal cartridge is disabled by the BIOS, in which case U1MB's $D5E0 register is hidden.

External device control

Several bits in UAUX [$D381] are devoted to controlling external devices. Bits 0-3 control external signals and otherwise have no meaning to U1MB itself, although they are conventionally labeled for COVOX and stereo POKEY enable signals.

Bits 4-5 control VideoBoard XE addressing, allowing selection between $D640, $D740, or disabling VBXE entirely. U1MB will automatically decode the $D6xx or $D7xx pages for VBXE if enabled, and in addition, automatically disable VBXE when those ranges are needed for I/O RAM. These bits have no effect if the VBXE enable signal is not hooked up to anything.

Bit 6 controls SoundBoard decoding. Unlike the other bits, it has an effect even if no SoundBoard is present: it prevents POKEY from being accessed at $D210-D2FF.

Bit 7 enables or disables writes to the flash ROM.

Parallel Bus Interface device emulation (revision 2 CPLD only)

U1MB also allows for PBI device emulation through up to 8K of banked PBI ROM and an additional 0.8K of I/O RAM. The emulated PBI device can be configured to use bit 0, 2, 4, or 6 of the device select register at $D1FF; when selected, PBI ROM is enabled at $D800-DFFF as a math pack overlay. This overlays either RAM or OS ROM. Four banks of 2K are exposed from $59800, $5B800, $5D800, and $5F800 in the flash ROM, selectable through a bank switch register at $D1BF.

When PBI ROM is enabled, 895 bytes of I/O RAM are also exposed: 191 bytes at $D100-D1BE, 192 bytes at $D500-D5BF, and 512 bytes at $D600-D7FF. These bytes are from dedicated memory not otherwise accessible due to being shadowed by the I/O address region. When I/O RAM is active, these address ranges are blocked by the U1MB MMU so that they do not activate external cartridge control (CCTL) or VBXE accesses.

PBI device emulation is only available on U1MB devices with updated CPLD firmware; the original run lacks it.

> **Warning**
>
> The A29040 flash chip used in some devices has a 64K sector size, which means that it is not possible to reflash the PBI ROM without erasing the BIOS ROM at the same time. This is risky as a corrupted BIOS will brick the computer, since the computer cannot boot without the BIOS ROM. Furthermore, if the PBI ROM is enabled, the OS will attempt to call into it on any SIO operation. Flashing software that updates either the BIOS or PBI ROMs must be written with this in mind.

PBI button function (revision 2 CPLD only)

Although not required, the PBI device emulation functionality is intended to be used with the CompactFlash interface of a SIDE 2 cartridge. An additional “PBI button” feature can be enabled in U1MB to take advantage of the reset button of the SIDE 2, allowing it to be used as an input to drive PBI-based disk emulation.

When the PBI button function is enabled, the left cartridge window ($A000-BFFF) of the external cartridge is suppressed, but can still be sensed via bit 6 of [$D384]. To detect a press of the reset button, the PBI firmware disables the SIDE 2 $A000-BFFF banking window; a press of the reset button causes the cartridge to reset to bank 0 with the window enabled, which can then be sensed via $D384. While the button can be sensed regardless, the PBI button function prevents $A000-BFFF from suddenly being overlaid by the external cartridge.

The PBI button function must be disabled for external cartridges to work normally.

Real-time clock

A Maxim DS1305 real-time clock chip on the U1MB provides a clock and 96 bytes of non-volatile user storage. Both are battery backed up and thus persist across power cycles. The RTC is connected via Serial Peripheral Interface (SPI) bus and communication occurs serially through three bits in RTCIN/RTCOUT [$D3E2]. Since it is hooked up in 4-wire configuration, the clock may be driven with either polarity per the DS1305 specs.

The DS1305 has timing specs that are difficult but not impossible to violate using the stock 6502. One way to do so is to attempt to change CE (chip enable) or SDO (serial data out) in the same write as a change to SCLK (serial clock). This can violate the setup/hold timing requirements for the DS1305. The timing constraints on CE (chip enable) can also be exceeded by back-to-back stores. A 65C816 accelerator running in fast RAM can do so much more easily and more care is required when driving the DS1305 from an accelerator.

Config lock

When the BIOS is invoked on power-up or reset, the U1MB is initially in unlocked mode. This enables I/O RAM at $D100-D1BE, $D500-D5BF, and $D600-D7FF and the configuration registers. BASIC and GAME ROM are disabled and inaccessible. If enabled, these regions map to the same region as the internal cartridge window.

Once the BIOS is finished configuring the system, the configuration is locked by setting bit 7 of UCTL [$D380]. This swaps out the BIOS, enables the selected OS/GAME/BASIC ROMs and disables the configuration registers. Since the BIOS ROM image is swapped out in this process, a jump into RAM is usually necessary to trigger the config lock and then invoke the RESET vector on the OS ROM.

Config lock cannot be turned off in software once enabled and can only be reverted by a reset.

Memory mapping

The U1MB contains 1MB of extended memory, which can be enabled through UCTL[1:0]. This memory is only used for PORTB extended memory. Normal memory, and the I/O memoy enabled in PBI and config lock modes, still comes from the memory on the motherboard.

Non-canonical PIA access incompatibility

Normally, the entire address range $D300-D3FF is mapped to the Peripheral Interface Adapter (PIA). However,

in an U1MB system, $D380-D3FF is reserved for U1MB registers, and the PIA responds only to $D300-D37F. This is true at all times, regardless of configuration or config lock state. Memory locations that do not correspond to a readable U1MB register return undriven bus data.

Extended memory banking anomaly

Most memory expansions leverage the unused bits in PIA port B to select the extended memory bank. In the 576K and 1088K modes, this leads to a conflict with bits 1, 6, and 7, which control BASIC, XEGS Game, and self-test ROMs, respectively. Some memory expansions simply take over some of these bits, disabling the self-test ROM and/or requiring an external switch for BASIC.

Because U1MB shadows the PIA rather than using the output of PIA port B, it has unusual behavior here: in 576K and 1088K modes, it modifies BASIC / Game / self-test enables based on changes to bits 1, 6, and 7 only for writes that have the CPU window disabled (bit 4 = 1). This has three odd consequences: it means that PORTB writes are now sequence dependent as there are 11 bits of state being driven by 8 bits of written data, these ROMs can be enabled while the expanded memory window is active, and in 576K mode, these ROMs can be toggled while changing banking bits with only the ANTIC window enabled.

Cartridge sense anomaly

In XL/XE hardware, TRIG3 senses the state of the left cartridge window ($A000-BFFF) to provide a cartridge detection mechanism. This is emulated by U1MB so that when the internal cartridge is enabled, TRIG3 is active as expected. However, this is done by intercepting reads from TRIG3 in a way that is not sensitive to the trigger latching feature enabled by bit 2 of GRACTL. This means that if either the internal or external cartridge is unmapped from $A000-BFFF while trigger latching is enabled on a U1MB equipped system, TRIG3 will not stay 1 as ordinarily expected.

Registers

$D1BF UPBIBANK – PBI ROM bank select (write only; PBI ROM active only)

Ignored	BANK

D1:D0 PBI ROM bank

00	$59800-59FFF
01	$5B800-5BFFF
10	$5D800-5DFFF
11	$5F800-5FFFF

Controls the ROM bank mapped at $D800-DFFF when PBI emulation is active. This register is invisible and the bank is reset to 0 whenever the PBI device is deselected.

$D380 UCTL – Main configuration (write only; config unlocked only)

LCK	IOR	Ign.	SDX	OS	MEM

D7 Config lock

0	No change
1	Lock config

D6 I/O RAM enable

0	I/O RAM disabled
1	I/O RAM enabled

D4 SpartaDOS X (SDX) module enable

0	SDX module enabled

1 SDX module disabled

D3:D2 OS ROM select

00 $70000-73FFF
01 $74000-77FFF
10 $78000-7BFFF
11 $7C000-7FFFF

D1:D0 Memory configuration

00 64K – no extended RAM
01 320K Rambo – PORTB bits 2, 3, 5, and 6 control bank; bit 4 controls CPU+ANTIC access
10 576K Compy – PORTB bits 1, 2, 3, 6, and 7; bit 4 controls CPU access, bit 5 controls ANTIC access
11 1088K – PORTB bits 1, 2, 3, 5, 6, and 7; bit 4 controls CPU+ANTIC access

$D381 UAUX – Auxiliary configuration (write only; config unlocked only)

WE	SB	VBXE	S1	S0	M1	M0

D7 Flash write enable

0 Flash writes enabled
1 Flash writes disabled

D6 SoundBoard enable

0 $D210-D2FF assigned to SoundBoard
1 $D210-D2FF assigned to POKEY

D5:D4 VideoBoard XE (VBXE) address

00 $D640
01 $D740
1x Disabled

D3:D0 S1/S0/M1/M0 signal outputs

Controls flash and external device decoding. If flash writes are enabled, all memory windows that are mapped to the flash ROM accept writes; otherwise, writes to flash are blocked.

$D382 UPBI/UCAR – PBI/cartridge configuration (write only; config unlocked only)

GAME	BASIC	BTN	PBI	PBI_ID

D7:D6 Game ROM select

00 $68000-69FFF
01 $6A000-6BFFF
10 $6C000-6DFFF
11 $6E000-6FFFF

D5:D4 BASIC ROM select

00 $60000-61FFF
01 $62000-63FFF
10 $64000-65FFF
11 $66000-67FFF

D3 PBI button enable

0 PBI button disabled
1 PBI button enabled

D2 PBI emulation enable

0 PBI emulation disabled
1 PBI emulation enabled

D1:D0 PBI device ID select (PBI emulation enabled only)

00 PBI device ID 0 ($01)
01 PBI device ID 2 ($04)
10 PBI device ID 4 ($10)
11 PBI device ID 6 ($40)

$D383 COLDF – Cold reset flag (read/write; read-only once config locked)

CLD	0

D7 Cold reset flag

0 Last reset was warm reset
1 Last reset was cold reset

The cold reset flag is automatically set to 1 by the hardware on power-up, and can be set or cleared under software control. This is used to reliably distinguish between cold and warm resets, as its state persists across a warm reset. Once config lock is established, this register becomes read-only until the next reset.

$D384 PBI button status (read only)

BTN	RD5	0

D7 PBI button status

0 PBI button feature disabled
1 PBI button feature enabled

D6 External cartridge RD5 sense

0 External cartridge $A000-BFFF unmapped
1 External cartridge $A000-BFFF mapped

The PBI button status register is used to sense whether the PBI button feature is enabled and to check whether it has been pressed. Bit 6 indicates whether the external cartridge is attempting to map $A000-BFFF; this is used to sense the reset button on the SIDE 2 cartridge, since that button re-enables the banking window at bank 0.

This register is always visible even if the PBI emulation and PBI button features are disabled.

$D3E2 RTCIN – Real-time clock input (read only)

0	SDI	0

D3 Serial Data In (SDI) line

Senses the Serial Data In (SDI) line used to receive data from the DS1305 real time clock.

$D3E2 RTCOUT – Real-time clock output (write only)

Ignored	SDO	SCL.	CE

D2 Serial Data Out (SDO) signal
D1 Serial Clock (SCLK) signal
D0 DS1305 Chip Enable (CE) signal

0 DS1305 not selected
1 DS1305 selected

Used to drive the output signals on the Serial Peripheral Interface (SPI) bus to which the DS1305 RTC is connected. The chip must be enabled through the CE bit, and then SCLK toggled to either shift data into the chip a bit at a time through SDO or out a bit at a time through SDI.

Consult the DS1305 datasheet for required timing specifications. The general required precautions, for a DS1305 running at 2.0-3.3V and the standard 1.79MHz machine clock:

- Avoid writing to RTCOUT with read/modify/write instructions, due to the back-to-back writes with different values.
- Toggle SCLK no faster than every third cycle.
- Wait at least one cycle after a write to SCLK before reading SDI during reads.
- The first transition of SCLK must occur at least 7 cycles after CE is asserted.
- CE must be inactive for at least 7 cycles before being asserted again.

$D5E0 SDXCTL – SDX module control (write only; internal cartridge enabled only)

INT/EXT	SDXBANK

D7:D6 Internal/external cartridge enable

0x	Internal cartridge only enabled
10	External cartridge only enabled
11	Internal and external cartridge disabled

D5:D0 Internal cartridge bank

Controls both the internal cartridge, intended for SpartaDOS X (SDX), and the external cartridge. Both the $8000-9FFF and $A000-BFFF windows of the external cartridge are controlled together. If the PBI button feature is enabled, the $A000-BFFF window of the external cartridge is disabled even if the external cartridge is enabled. However, the $8000-9FFF window is unaffected.

This register is forced to $80 whenever the SDX module is disabled.

> **Warning**
>
> Writes to SDXCTL are not excluded from the cartridge control (CCTL) region, and so any writes to $D5E0 will be handled by *both* U1MB and the external cartridge.

11.4 VideoBoard XE

VideoBoard XE is an internal add-on that adds enhanced display capabilities, including higher horizontal resolution (640x), increased color depth of up 1024 colors per scan line out of a 21-bit color space, 80-column text, a hardware blitter, and RGB video output.

In addition, its FPGA core can be upgraded in software, allowing for bug fixes and additional features in the future. As of this writing, the current VBXE core is version 1.26.

For the official programming documentation for VBXE, see: [VBXE].

Architecture

VBXE acts parallel to the GTIA, interpreting ANTIC's output and replicating GTIA's behavior to reproduce the standard display. In parallel, the VBXE's extended display list (XDL) is used to drive new overlay and attribute map planes, which are combined with the ANTIC display to produce final output.

One significant point about this setup is that while VBXE shadows writes to GTIA, reads are still handled by GTIA itself. In particular, this means that collisions act the same as they normally do based solely on the ANTIC playfield and GTIA player/missile graphics, ignoring all of the new functionality. Display timing is also controlled by ANTIC; VBXE does not trigger vertical blank or display list interrupts, which must still be done through ANTIC's interrupt facilities.

Local memory

The VBXE contains 512K of high-speed local video memory, which can be used by the extended display and blitter. The local memory subsystem runs at 14MHz, providing 8x the memory bandwidth available to ANTIC. Data must be either created in local memory using the blitter or uploaded with the CPU before it can be used by VBXE.

Two memory access windows are provided to access local memory, MEMAC A and MEMAC B. MEMAC A is a flexible window that can be 4K, 8K, 16K, or 32K in size and placed anywhere in the 64K address space on 4K boundaries. MEMAC B is a fixed 16K window at $4000-7FFF. Both windows are read/write and can be enabled for either CPU/ANTIC access or both. MEMAC A has priority over MEMAC B if both are enabled.

Extended display list (XDL)

The extended display list (XDL) is the VBXE equivalent of the ANTIC display list. It runs in parallel to ANTIC's display list and controls the extended display functions on the VBXE side, including the overlay and attribute map layers.

To enable the XDL, bit 0 must be set in the VIDEO_CONTROL register. Once enabled, the XDL automatically repeats each frame starting at the local address specified by XDL_ADR0-2. Like the ANTIC display list, the XDL begins execution immediately after vertical blank starting at scan line 8.

Table 23 gives the layout of an XDL entry. Only two bytes are required for each entry; the remainder of the entry is composed of optional blocks depending on enable bits in the first two bytes. Optional parameters remain in effect until modified again.

	Data							
Control	OvScroll	OvAddress	Repeat	AtMapOff	AtMapOn	OvDisable	Graphics	Text
	End		Lores	Hires	OvMisc	AttrLayout	AttrAddr	ChBase
Repeat	Repeat count							
Overlay address	Overlay address, bits 7-0							
	Overlay address, bits 15-8							
						Overlay address, bits 18-16		
	Overlay stride, bits 7-0							
						Overlay stride, bits 11-8		
Overlay scroll						Horizontal scroll		
						Vertical scroll		
Character base	Character set base address, bits 18-11							
Attribute map address	Attribute map address, bits 7-0							
	Attribute map address, bits 15-8							
						Attribute map address, bits 18-16		
	Attribute map stride, bits 7-2							
					Attribute map stride, bits 11-8			
Attribute map layout				Attribute map horizontal scroll				
				Attribute map vertical scroll				
				Attribute map cell width, minus one				
				Attribute map cell height, minus one				
Overlay misc.	Playfield palette		Overlay palette				Overlay width	
	Overlay priority							

Table 23: VBXE extended display list (XDL) entry format

The first optional block is the repeat count, enabled by bit 5 of the first control byte. Unlike ANTIC, which requires mode bytes to be repeated, VBXE allows a repeat count for mode lines to compact the XDL. The repeat byte is the number of times to additionally repeat the mode line, so a repeat value of $FF causes 256 counts of the mode line. For simple displays, the repeat count allows the XDL to be more compact than the equivalent ANTIC display.

The next optional block is the overlay address, enabled by bit 6 of the first control byte. The first three bytes are the new starting address of the overlay row, and the last two bytes are the stride from the start of one row to the next, in bytes. This is the equivalent of the ANTIC LMS instruction, except that the stride is also controllable.

Bit 7 of the first control byte enables new horizontal and vertical scroll values, in pixels. These values only affect text modes.

Bit 0 of the second control byte enables a new base address for the text mode font.

Bit 1 of the second control byte sets a new base address and stride for the attribute map. Note that the attribute map stride can only be set as multiples of 4.

Bit 2 of the second control byte sets new scrolling and cell size parameters for the attribute map. Both scroll and cell size parameters are in VBXE standard resolution pixels, minus one.

Bit 3 of the second control byte loads miscellaneous parameters: the overlay/attribute map width, overlay and ANTIC playfield palettes, and the overlay priority bits. Overlay/attribute map width is %00 or %11 for narrow, %01 for normal, and %10 for wide. The overlay priority byte selects which layers the overlay has priority over, in the same format as the P0-P3 registers.

Bit 7 of the second control byte terminates the XDL when set.

XDL restart

Unlike the ANTIC DL, the XDL does not require a jump instruction at the end or the CPU to rewrite the XDL starting address each frame. It is instead automatically restarted from the XDL_ADR0-2 address automatically at the beginning of vertical sync. Note that this requires the XDL to be set up earlier than ANTIC, which allows the display list to be initialized as late as the end of vertical blank or even later.

Many parameters that can optionally be set in the XDL are reset to defaults at the beginning of each frame:

- Text mode scroll offsets are set to 0.
- Overlay width is set to normal width (%01).
- Overlay priority is set to $FF (priority over all other layers).
- The attribute map is disabled, attribute map scroll offsets are reset to 0, and the attribute cell size is set to 8x8 ($07, $07).
- Palette selections are reset to palette 0 for ANTIC's display and palette 1 for the overlay.

Notably, overlay and attribute map addressing is not automatically reset and should be initialized at the beginning of the XDL.

Overlay

The overlay is VBXE equivalent of ANTIC's playfield, displaying either text or bitmap graphics. It is so named because it normally displays on top of the playfield. Overlays can be displayed in the same three widths, narrow (128 color clocks), normal (160 color clocks), and wide (184 color clocks).

The overlay mode is selected by bits 0-1 of the first XDL control byte and bits 4-5 of the second:

- **Standard resolution (Text = 0, Graphics = 1, LR = 0, HR = 0)**: 320 pixels in normal width, same resolution as ANTIC hires, but with one byte per pixel (256 colors).
- **Low resolution (Text = 0, Graphics = 1, LR = 1, HR = 0)**: 160 pixels in normal width, same resolution as ANTIC lores, but with one byte per pixel (256 colors).
- **High resolution (Text = 0, Graphics = 1, LR = 0, HR = 1)**: 640 pixels in normal width, with one nibble per pixel (16 colors).
- **Text (Text = 1, Graphics = 0)**: 80-column text with attribute control bytes. Each pixel in the text font is rendered at hires resolution (640 across in normal width).

The wide overlay differs slightly in width from an ANTIC wide playfield. A wide overlay is displayed from horizontal positions $2C-D3, whereas an ANTIC wide playfield is displayed further right at $2C-DD.

VBXE's overlay does not require LMS instructions every mode line to accommodate non-standard strides between scan lines. The stride is set directly in the XDL and can accommodate address offsets from 0 to 2047 bytes.

Overlay priority

The priority of the overlay versus player/missile graphics is controlled via an 8-bit overlay priority mask. Bits 0-3 set priority versus player/missiles 0-3, bits 4-6 set priority vs. playfields 0-3, and bit 7 sets priority over the background color. A '1' specifies priority of the player/missile or playfield layer over the overlay.

Overlay priority is determined based on the result of player/missile and playfield priority. The priority bits corresponding to all color inputs are ORed together and the result is then used to mask the overlay. This means that the multicolor player bit affects overlay priority – if P0 and P1 are both active, this determines whether the priority bit for P0 alone or P0 and P1 together affect the overlay. If more than one layer is active, the overlay has priority if the priority bit for any of the layers is set.

Normally, the overlay priority is determined by the XDL. If the attribute map is active, it overrides the XDL and can select the priority register on a per-cell basis.

Overlay collision detection

Collisions are automatically detected between the overlay and other layers, including the playfield, player/missile graphics, and the attribute map. Like with GTIA, these collisions are detected during scan out and are registered for later inspection in the COLDETECT register. A write to COLCLR resets COLDETECT and prepares for another collision scan.

A major difference between GTIA collisions and VBXE overlay collisions is that the latter are affected by priority settings, specifically PRIOR bits 0-5. With GTIA, collisions are registered between all objects even if some are obscured, and neither the fifth player nor multicolor enables affect collisions. On the other hand, VBXE flags collisions based on which color registers contribute to the final output, and therefore will not flag overlay collisions with hidden objects and is sensitive to both the P5 and multicolor player bits. For instance, if players 0-2 and missile 3 are overlapping the overlay, VBXE will flag only P0+P1 collisions if PRIOR=$24, and only a PF3 collision if PRIOR=$14. The overlay priority settings, however, do not affect overlay collisions.

The COLMASK register selects which subset of overlay byte values can trigger a collision. In lores and standard res modes, this allows masking each of the eight groups of 32 color values out of the total of 256 to control collisions. In hires modes, this filtering is still by byte, so it filters based on the color of the left pixel of each pixel pair.

Text mode

In text mode, the overlay consists of pairs of bytes, a character name byte followed by an attribute byte. The image for each character is supplied as 8x8 bitmaps in a 2K block of local memory. This is similar to an ANTIC mode 2 character font except that the font contains a full set of 256 characters.

The attribute byte determines the colors used for the character cell. Bits 0-6 select the foreground color, from colors $00-7F. Bit 7 controls the background opacity. If set, the background is opaque and uses the foreground color with bit 7 set ($80-FF); if clear, the background is either transparent or color $80 depending on the overlay transparency mode.

Unlike ANTIC text modes, VBXE text mode lines are a single scan line tall and must be repeated 8 times to display a full character cell row. Vertical scrolling also works differently, changing offsets within the mode lines rather than changing mode line heights. The overlay memory pointer is advanced to the next row after whenever the last row (row 7) is displayed.

Attribute map

The attribute map allows the display to be altered on a per-cell basis, where the size of a cell varies from 8x1 to 32x32 VBXE standard resolution pixels. Palette selection and other rendering modes can be altered for each cell.

Each cell is controlled by a four-byte block:

	Data				
0	PF0 color or hires PF3 mask				
1	PF1 color				
2	PF2 color				
3	Playfield palette	Overlay palette	Collision	Rev	OvPriority

Table 24: VBXE attribute map block layout

The first three bytes override the PF0-PF2 playfield colors used for rendering the ANTIC playfield.

Control bits 0-1 override the overlay priority setting from one of the P0-P3 registers.

Control bit 2 reverses lores/hires interpretation of ANTIC data so that lores data is interpreted as hires and vice versa. Hires data is reinterpreted in bit pairs as PF0-PF3; lores data is reinterpreted as pairs of hires pixels.

Control bit 3 enables collisions between the attribute map and the overlay. A collision is signaled whenever an attribute map cell with this bit set overlaps non-transparent overlay pixels.

Control bits 4-5 override the palette used for the overlay.

Control bits 6-7 override the palette used for the playfield.

In ANTIC hires mode, byte 0 is repurposed as a PF3 mask instead of a PF0 override. A '1' bit replaces the PF2 background color with PF3. Bits are rendered in standard bitmap order from MSB to LSB. The resolution of the PF3 mask is determined by the width of the attribute cell: 1 pixel/bit at width 8, 2 pixels/bit at widths 9-16, and 4 pixels/bit at widths 17-32.

The horizontal position and width of the attribute map is controlled by the same width setting as the overlay width. This is true regardless of whether the overlay is enabled and of the width of the ANTIC playfield. Areas not covered the attribute map are rendered as if the attribute map were disabled.

Attribute map limitations

The attribute map is buffered in on-chip memory, which allows the map to be fetched only once per cell row and relaxes timing requirements. However, it also limits the width of the attribute map to 43 cells. This is just enough to cover a wide width overlay with horizontal scrolling with a cell width of 8 pixels. Below 8 pixels, the attribute map may run out of data as the hardware is constrained to stop fetching beyond 43 cells.

There is no limit on cell height; the attribute map can be used with single scan line resolution.

Blitter

The hardware blitter greatly accelerates data copying and transformation in VBXE space. It is a two-argument, src/dst blitter that can do simple arithmetic and logical operations between arbitrary 2D memory rectangles. The blitter can use the full 14MHz bandwidth of the VBXE local memory bus, but is limited to accessing local memory only; data must be copied to and from local memory by the CPU for the blitter to work with non-local data. The blitter cannot access main memory or hardware registers directly.

The blitter is driven by a blit list in local memory, which contains a linear array of 21-byte entries for each blit. Each blit entry contains all information required to set up the blit, including source and destination addresses, mode selectors, strides, and size information. The only CPU intervention required is to set up the beginning of the blit list and trigger the blitter. An IRQ can optionally be triggered at the end of the blit list to notify the CPU when all blits have completed.

Offset	Data
0	Source address bits 7-0
1	Source address bits 15-8
2	Source address bits 23-16
3	Source Y step bits 7-0
4	Source Y steps 12-8 (signed)
5	Source X step bits 7-0 (signed)
6	Destination address bits 7-0
7	Destination address bits 15-8
8	Destination address bits 23-16
9	Destination Y step bits 7-0
10	Destination Y step bits 12-8 (signed)
11	Destination X step bits 7-0 (signed)
12	Source width minus 1 bits 7-0
13	SW8
14	Source height minus 1 bits 7-0
15	Source data AND mask
16	Source data XOR mask
17	Collision mask
18	Source zoom Y minus 1 / Source zoom X minus 1
19	PatEnable / Source pattern width minus 1
20	Next / Mode

Table 25: VBXE blitter setup block

Bytes 0-11 specify the source and destination areas. The initial addresses are for the first byte to read/write, and step values are specified for both X and Y directions. The step values are signed and may also be zero, allowing for ascending and descending blits, strided blits, and pattern fill operations. Note that for a descending blit, the beginning of the last row or column must be specified and not one-after as for some copy interfaces; the supplied addresses are always the first ones used. Also, the Y step values are independent of the X step values and the copy width, so that each starting row address is the previous starting row address plus the Y step offset. This is often referred to as a pitch or stride value, versus a modulo value which is the distance from the *end* of one row to the *beginning* of the next.

Bytes 12-14 specify the blit size as width and height values. This is specified as the number of bytes or rows processed, minus 1, and allows for up to a 512x256 blit. This means that only 128K of memory can be modified per blit, but the X and Y step values mean that the range of addresses touched can be larger.

Bytes 15 and 16 allow modification of each source byte. Each source byte is bitwise ANDed with the AND mask and then XORed with the XOR mask. Using an AND mask of $00 allows use of constant source data without having to fill source memory.

Byte 18 controls source zoom. Each byte can be expanded to up to an 8x8 rectangle by replication. This is done by repeating the columns and rows in the blit with the same source byte; if zoom is set to 2x3, each source byte is repeated twice during each row, and then each row is repeated three times. The blit size is in terms of source area, so a 320x200 blit with 2x2 zoom reads a 320x200 source area and writes a 640x400 destination area. The X and Y destination step offsets are applied as usual between each repetition of a source byte.

Byte 19 controls source pattern mode. If enabled, 1-128 bytes at the beginning of each source row are repeated. This happens after zoom, so 4x zoom with a pattern length of 8 gives four copies of the first pattern byte, four copies of the second pattern byte, etc.

Byte 20 selects the blit mode. Table 26 lists the possible modes.

Mode	Operation	Description
0	Copy	Copy source to destination
1	Byte stencil copy	Copy source to destination if non-zero
2	Binary addition	Add source to destination
3	Bitwise OR	Compute bitwise OR of source with destination
4	Bitwise AND	Compute bitwise AND of source with destination
5	Bitwise XOR	Compute bitwise XOR of source with destination
6	Nibble stencil copy	Copy each nibble from source to destination if non-zero

Table 26: VBXE blit modes

Modes 1 and 6 are directly suited for blitting sprites with color 0 as a transparency value; mode 1 is for the byte oriented modes (LR/SR) and mode 6 is for the nibble oriented hires mode (HR).

Bit 3 in byte 20 indicates whether another blit follows in the blit list. If set, the blitter will automatically read in and perform the next blit in the blit list after the current one finishes. The blitter can therefore do a long series of heterogeneous blit operations without CPU involvement. This can be particularly advantageous given that the blitter's X/Y step offset capability allows the blitter to modify its own blit lists. However, a significant limitation is that the blit list must be contiguous in memory as there is no jump facility.

Blitter collision detection

The blitter can detect collisions between source data being merged with destination data. A collision is detected when a non-transparent source pixel is merged with a non-transparent source pixel. No collisions are detected in mode 0; collisions are detected per byte in modes 1-5, and per-nibble in mode 6.

The collision mask field in the blit block controls which destination pixel values trigger collisions. The range of palette indices is partitioned into 8 groups, where each bit enables collisions for a corresponding group. In modes 1-5, bit 0 enables collisions with $01-1F, bit 1 with $20-3F, bit 2 with $40-5F, etc. In mode 6, each group is two colors, so bit 0 enables $1, bit 1 enables $2-3, bit 2 enables $4-5, etc. A collision mask of $00 disables collision detection.

Upon the first collision within a blit, the destination pixel causing the collision is copied into the blitter collision code register. In mode 6, a collision in the high nibble or low nibble copies the destination pixel to the high or low nibble of the collision code register; only one nibble will be set per blit.

Collision detection is free for most modes, with the exception of stencil copies (mode 1). In mode 1, enabling collision detection reduces the speed of the blit because it requires a read from the destination that would otherwise be unnecessary.

Blitter speed

The blitter's speed depends on the available memory bandwidth, the operations selected, and sometimes the data involved. First, it uses all local memory cycles available, but at the lowest priority; any accesses to VRAM by the display, XDL, blitter, or MEMAC windows preempt the blitter. Higher-density displays therefore result in slower blits. Faster operations run at two cycles/byte, whereas slower ones run at three cycles/byte. Table 27 gives the speed of each blit mode.

Mode	Speed
Mode 0 (fill/copy)	2 cycles/byte
Mode 1 (bytewise stencil copy)	1 cycle/byte for $00 source 2 cycles/byte for non-$00 source w/o coll. detect 3 cycles/byte for non-$00 source w/coll. detect
Mode 2 (add)	1 cycle/byte for $00 source 3 cycles/byte for non-$00 source
Mode 3 (bitwise OR)	1 cycle/byte for $00 source 3 cycles/byte for non-$00 source
Mode 4 (bitwise AND)	2 cycles/byte for $00 source 3 cycles/byte for non-$00 source
Mode 5 (bitwise XOR)	1 cycle/byte for $00 source 3 cycles/byte for non-$00 source
Mode 6 (nibblewise stencil copy)	1 cycle/byte for $00 source 3 cycles/byte for non-$00 source

Table 27: VBXE blitter speeds

The $00 optimization check occurs after constant AND/XOR factors have been applied, and pertains to inputs into the mode operation such that either no change occurs in the destination or the existing value can be ignored. Note that there is no optimization for AND/OR with $FF.

Additionally, the blitter can skip source fetches if the source is known to be constant. If the source AND mask is $00, or for repetition due to X zoom, the blitter will skip source fetches after the first. For instance, a fill operation with mode 0 can run at 1 cycle/byte instead of 2 cycles/byte. However, a Y zoom is not accelerated in this manner and will re-read the source each time, so a 2x8 zoom blit can be done faster as a transposed 8x2 zoom blit.

In the event that both optimizations apply – source AND and XOR masks both $00, or X zoom of a $00 value – the blitter runs at 1 cycle/byte, re-reading the source.

For small blits, the time to read in the blit information from the blit list is also significant. Setting up each blit requires 21 free memory cycles for each blit.

DMA pattern

VBXE local memory accesses are primarily driven off of an 8-cycle sequence associated with machine cycles. MEMAC and overlay accesses have the highest priority; the former only occur on odd cycles and the latter on even cycles, so they never conflict with each other. The next priority are XDL/attribute map fetches, and the blitter has the lowest priority.

MEMAC accesses occur on cycle 1 for reads or cycle 3 for writes. Having ANTIC or the CPU accessing local memory through a MEMAC window can therefore consume up to one-eighth of the available local memory bandwidth. This will not affect the display, but may delay attribute map fetches or slow down the blitter.

The overlay only consumes memory cycles during the active region. Because the overlay uses only even cycles, it never collides with MEMAC accesses, which are on odd cycles. For graphics modes, lores modes fetch on cycles 0 and 4, while standard and hires modes fetch on 0, 2, 4, and 6. Text modes fetch character name, character attribute, and font data on even cycles every eight cycles.

XDL updates require 22 VBXE cycles just after the end of the overlay active region, regardless of how many bytes are actually read. MEMAC cycles preempt the XDL regardless of whether a fetch occurs, but any memory cycles skipped by the XDL are available for the blitter.

The attribute map is read immediately after the XDL is or would be processed. 172 bytes are read from local memory into an on-chip line buffer during horizontal blanking time, which is then used to display during active region. This load only occurs when a new row of the attribute map is encountered, either by rolling over the row counter, changing attribute map addressing, or restarting the attribute map. However, 172 bytes are always read regardless of the width of the playfield or attribute cells. Only MEMAC can preempt attribute map fetches, so this takes at most 26 machine cycles to complete.

The blitter has lowest priority and uses any spare cycles not otherwise needed by other DMA engines. Available blitter bandwidth may range from 236-912 cycles per scan line depending on other DMA requirements. The blitter never has idle cycles during a blit and is always reading and writing memory, so it only makes progress on a blit when a memory cycle is available.

No stolen cycles are needed for refreshing VBXE local memory, and none of the VBXE local accesses are slowed by ANTIC DMA cycles in main memory. ANTIC DMA only slows down VBXE operations when mapped to local memory through a MEMAC window.

Soft reset

Any write to $D080-D0FF in GTIA address space causes the VBXE to soft reset. This is used to ensure that VBXE resets properly on power-up. However, it can also cause compatibility problems if these addresses are written during normal operation. Accidentally writing $D080 every frame, for instance, will cause the VBXE display to blank out.

Register map

VBXE is controlled via a register bank of 32 bytes, normally decoded at either $D640-D65F or $D740-D75F. Most registers are write-only, although a couple of read-only and read-write registers exist. None of the read registers have side effects, so it is safe to use indexed loads and stores to VBXE.

	Read	Write
$Dx40	CORE_REVISION	VIDEO_CONTROL
$Dx41	MINOR_REVISION	XDL_ADR0
$Dx42		XDL_ADR1
$Dx43		XDL_ADR2
$Dx44		CSEL
$Dx45		PSEL
$Dx46		CR
$Dx47		CG
$Dx48		CB
$Dx49		COLMASK
$Dx4A	COLDETECT	COLCLR
$Dx4B		
$Dx4C		
$Dx4D		
$Dx4E		
$Dx4F		
$Dx50	BLT_COLLISION_CODE	BL_ADR0
$Dx51		BL_ADR1
$Dx52		BL_ADR2
$Dx53	BLITTER_BUSY	BLITTER_START
$Dx54	IRQ_STATUS	IRQ_CONTROL
$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

Table 28: VBXE registers

$Dx40 CORE_REVISION – VBXE major revision (read only)

Major version	GTIA emulation mode

D7:D4 Major version
D3:D0 GTIA emulation mode

- 0 Full FX core
- 1 GTIA-only core

Indicates the major version of the VBXE core and whether full FX functionality is enabled or only GTIA emulation. As of this writing, the latest FX core is version 1.26 (CORE_REVISION = $11, MINOR_REVISION = $26).

$Dx40 VIDEO_CONTROL – Video control register (write only)

Ign.	OvTrans	EXT	XDL

D3:D2 Overlay transparency mode

- x0 Disabled
- 01 Transparency enabled for color 0
- 11 Transparency enabled for colors 0 and 15

D1 Extended color mode

- 0 Disabled – GTIA color registers select 128 colors and ANTIC hires mode uses one hue
- 1 Enabled – GTIA color registers select 256 colors and ANTIC hires mode uses two hues

D0 Extended display list (XDL) enable

- 0 Disabled
- 1 Enabled

Bit 0 enables or disables the extended display list (XDL). If disabled, the overlay and attribute maps are turned off. It is buffered and only takes effect at the beginning of vertical sync.

Bit 1 enables extended color mode. This extends the standard GTIA color registers from 3 bits of luminance to 4, and switches ANTIC hires mode to use the full PF1 color for the foreground instead of just the luminance. If disabled, the LSB of all GTIA color registers are forced to 0 during color processing, although bit 0 is stored regardless.

Bit 2 enables transparency for the overlay layer, so that color 0 is transparent with regard to collisions and P/M or playfield layers placed behind the overlay.

Bit 3 enables color $F (hires) or $xF (lores/standard) as an additional transparency color for display and collisions. It is used to generate colors that are visually transparent but still detected as collisions during blitter operations.

VIDEO_CONTROL is forced to $00 on reset.

$Dx41 MINOR_REVISION – VBXE minor revision (read only)

SHA	Minor high	Minor low

D7 Shared memory capability

- 0 RAMBO 256K emulation disabled
- 1 RAMBO 256K emulation enabled

D6:D4 Minor version high digit
D3:D0 Minor version low digit.

Indicates whether the current VBXE core supports extended memory emulation and the minor version of the core. $26 indicates version x.26 without RAMBO emulation.

$Dx41-Dx43 XDL_ADR0-2 – XDL start address (write only)

Sets the 19-bit starting address in local memory for the XDL, with XDL_ADR0 supplying bits 0-7. Unlike DLISTL/DLISTH in ANTIC, this is not the actual address register, but a buffer register that is copied to the actual register during vertical sync (*not* vertical blank). Any writes to these registers will not take effect until then.

$Dx44 CSEL – Color register write select (write only)

Sets the color register to modify in the currently selected write palette.

CSEL is indeterminate on power-up and not affected by reset.

$Dx45 PSEL – Palette write select (write only)

Ignored.	Palette

D1:D0 Palette to modify

Sets the palette to modify with the CR/CG/CB registers.

PSEL is indeterminate on power-up and not affected by reset.

$Dx46-48 CR/CG/CB – Palette write latches (write only)

Color value	Ign.

D7:D1 Red, green, or blue color value

Sets the red, green, or blue value to be written to the color register selected by CSEL/PSEL. The new color value takes effect immediately. A write to $Dx48 (CB) also increments CSEL to the next color register.

Palette 0 is reset to GTIA colors on power-up. The palettes are not affected by reset.

$Dx49 COLMASK – Overlay collision mask (write only)

C7	C6	C5	C4	C3	C2	C1	C0

D7 Enable collisions with overlay values $E0-FF
D6 Enable collisions with overlay values $C0-DF
D5 Enable collisions with overlay values $A0-BF
D4 Enable collisions with overlay values $80-9F
D3 Enable collisions with overlay values $60-7F
D2 Enable collisions with overlay values $40-5F
D1 Enable collisions with overlay values $20-3F
D0 Enable collisions with overlay values $00-1F

Controls which overlay data bytes can trigger collisions. Each bit enables one-eighth of the possible values.

Note that the check is byte value based even in hires display mode, where each byte contains two pixels.

The value of COLMASK is indeterminate on startup.

$Dx4A COLCLR – Overlay collision clear strobe (write only)

A write to COLCLR clears the COLDETECT register.

$Dx4A COLDETECT – Overlay collision detect (read only)

ATT	PF2	PF1	PF0	P3	P2	P1	P0

D7 Collision detected with attribute map
D6 Collision detected with playfield 2 or 3, or the fifth player
D5 Collision detected with playfield 1
D4 Collision detected with playfield 0
D3 Collision detected with player/missile 3
D2 Collision detected with player/missile 2
D1 Collision detected with player/missile 1
D0 Collision detected with player/missile 0

Indicates which collisions have been detected between the overlay and the player/missile, playfield, or attribute map layers since the last time COLCLR was written.

$Dx50 BLT_COLLISION_CODE – Blitter collision code status register (read only)

Contains the destination byte that triggered the first enabled collision detected during a blit. This register is automatically cleared to $00 at the beginning of a blit.

BLT_COLLISION_CODE has indeterminate contents on power-up or after reset.

$Dx50-Dx52 BL_ADR0-2 – Blitter blit list start address (write only)

Sets the 19-bit starting address for the blitter blit list, with BL_ADR0 supplying bits 0-7. This address is only used when a blit is started; any writes to BL_ADR0-2 do not take effect until the blitter is restarted.

$Dx53 BLITTER_BUSY – Blitter status register (read only)

0	BSY	LOD

D1 Blitter busy
- 0 Blitter idle or loading from blit list
- 1 Blitter active

D0 Blitter control block load
- 0 Blitter idle or active
- 1 Blitter loading control block

BLITTER_BUSY indicates the current blitter status. Bit 0 is set while the blitter is loading a control block, while bit 1 is set while the blitter is processing data. The two bits are never set at the same time.

$Dx54 BLITTER_START – Blitter start/stop control register (write only)

0	RUN

D0 Blitter run control
- 0 Stop blitter
- 1 Start blitter

BLITTER_START is used to start or stop the blitter. Writing bit 0 = 0 will immediately stop the blitter. Writing bit 0 = 1, however, will only start the blitter if it is not already running; if it is already running, 0 must be written to stop

it before 1 can be written to restart the blitter at a new blit list address. If the blitter is known to have finished the last blit list, it is not necessary to write a 0 bit before writing a 1 bit to start the new blit list.

$Dx54 IRQ_STATUS – IRQ status register (read only)

0	BC

D0 Blitter complete IRQ status
- 0 Inactive
- 1 Active

IRQ_STATUS indicates whether the blitter complete IRQ is active. The IRQ_CONTROL register must be written to clear the interrupt.

$Dx54 IRQ_CONTROL – IRQ control register (write only)

Ignored	BC

D0 Blitter complete IRQ enable
- 0 Disabled
- 1 Enabled

Enables or disables the blitter complete IRQ. Any write to this register, regardless of the value written, also clears any pending IRQ.

$Dx55-Dx58 P0-P3 – Overlay priority registers (write only)

BAK	PF2	PF1	PF0	P3	P2	P1	P0

D7 Overlay has priority over background color
D6 Overlay has priority over playfield 2, 3, or the fifth player
D5 Overlay has priority over playfield 1
D4 Overlay has priority over playfield 0
D3 Overlay has priority over player/missile 3
D2 Overlay has priority over player/missile 2
D1 Overlay has priority over player/missile 1
D0 Overlay has priority over player/missile 0
- 0 Other layer has priority (underlay)
- 1 Overlay has priority

Determines which layers the overlay appears under or over, when the attribute map is active. If more than one layer is active, the overlay has priority if any of the priority bits for those layers indicate that the overlay has priority. The attribute map selects which priority register is in effect for a pixel; if the attribute map is disabled, these registers are ignored and the priority value comes from the XDL.

$Dx5D MEMAC_B_CONTROL – MEMAC B window control register (write only)

ANT	CPU	Ign.	Starting address

D7 ANTIC enable
D6 CPU enable
D4:D0 Starting address (bits 18-14)

Enables the MEMAC B window for either ANTIC access, CPU access, or both, and sets the starting address to one of 32 16K banks in the 512K local memory space. Note that unlike the MEMAC A control registers,

MEMAC_B_CONTROL is write only.

MEMAC_B_CONTROL is set to 0 on reset.

$Dx5E MEMAC_CONTROL – MEMAC A control register (read/write)

Base	CPU	ANT	Size

D7:D4 Window base address ($x000)
D3 CPU access enable
D2 ANTIC access enable
D1:D0 Window size

00	4K window
01	8K window
10	16K window
11	32K window

Enables or disables the MEMAC A window and sets its location in CPU/ANTIC address space. The window does *not* have to be aligned to its size in CPU address space, so it is possible to have a 32K window starting at $2000. If the window extends beyond the end of the 64K address space, it is truncated and does not wrap to the beginning.

MEMAC_CONTROL is set to $00 on reset.

$Dx5F MEMAC_BANK_SEL – MEMAC A bank control register (read/write)

ENA	Starting address

D7 Window enable
D6:D0 Starting address (bits 18-12)

Enables or disables the MEMAC A window and sets its starting address in local memory. The window is always aligned to its size, so up to three low order bits in MEMAC_BANK_SEL may be ignored depending on the window size. However, those bits are still stored and can become active if the window size is shrunk.

MEMAC_BANK_SEL is set to $00 on reset.

Chapter 12
5200 SuperSystem

12.1 Differences from the 8-bit computer line

Power control

Two cartridge lines are used as a power switching mechanism to cut power to the console whenever a cartridge has been removed. Therefore, the console is never running without a cartridge.

Memory space

The 5200 contains 16K of random access memory from $0000-3FFF.

Cartridges have a much larger 32K address space window at $4000-BFFF. The cartridge area is dedicated and does not overlay RAM. There is no cartridge control region.

ANTIC

ANTIC still exists at $D400-D4FF and works the same as on the XL/XE. Nothing is attached to the $\overline{\text{RNMI}}$ control line.

GTIA

The 5200's GTIA lies at $C000-CFFF instead of $D000-D0FF. The four switch lines controlled by the CONSOL register are used solely for output, specifically controller selection and analog stick control.

POKEY

POKEY exists at $E800-EFFF in the 5200 instead of $D2xx. The keyboard scanning logic is connected to the controllers rather than to a dedicated keyboard. The SIO port is not used in the base system but is still exposed via the expansion port and vectored in the OS.

Peripheral Interface Adapter

The 5200 does not have a PIA chip. Controllers are read through POKEY and GTIA instead, and there is no memory remapping ability.

Operating system

The OS ROM is only 2K in the 5200, from $F800-FFFF. The character font is at $F800, leaving only 1K of code space. There are no defined vectors within the OS.

The region from $F000-F7FF consists of an additional 2K of unused ROM.

12.2 Controller

The 5200 controller consists of an analog stick, a pair of top/bottom buttons, a 9-digit pad with # and * buttons, and Start/Reset/Pause buttons. Depending on the model, either two or four controllers can be attached to the system unit.

Multiplexing

The key pad and top button of all controllers are multiplexed and bits 0-1 of CONSOL select the controller to read. The bottom button and analog stick have dedicated inputs per controller and are not affected.

Analog stick

The analog stick is composed of a pair of potentiometers hooked up to pairs of POTx lines on POKEY. Even POT lines correspond to horizontal sticks, with lower values indicating left direction and higher values indicating right direction. Similarly, odd POT lines correspond to vertical, with lower to upper values meaning up to down placement.

Famously, the 5200 controller's analog stick does not auto-center, and thus the center position must be determined in software. Common techniques for doing this include periodically reading the joystick position between waves or levels and taking the average of min/max measured positions as the center. A correctly functioning controller is guaranteed to have a side-to-side range of at least 160 counts in the corresponding POTx register.[50]

Bit 2 of CONSOL must be set in order for the analog joystick to read properly. Clearing it cuts power to the potentiometers, causing the POTx registers to instead register the maximum value of 228 ($E4). This line also doubles as the calibration control line for the trackball and as a trackball detection mechanism.[51]

Keypad

The keypads for all four controllers are multiplexed onto the keyboard scanning lines of POKEY. The low two bits of CONSOL select the controller to read, with 00 selecting controller #1, 01 selecting #2, etc. From the selected controller, the twelve buttons and the three game control buttons are mapped onto KBCODE bits 1-4 as follows:

1 1111	**2** 1110	**3** 1101	**Start** 1100
4 1011	**5** 1010	**6** 1001	**Pause** 1000
7 0111	**8** 0110	**9** 0101	**Reset** 0100
*** 0011	**0** 0010	**#** 0001	

The $\overline{K0}$ and $\overline{K5}$ output lines are not used, causing POKEY to detect each pressed key two times during each keyboard scan. For instance, holding down the 0 key will cause key values $04, $05, $24 and $25 to be detected.

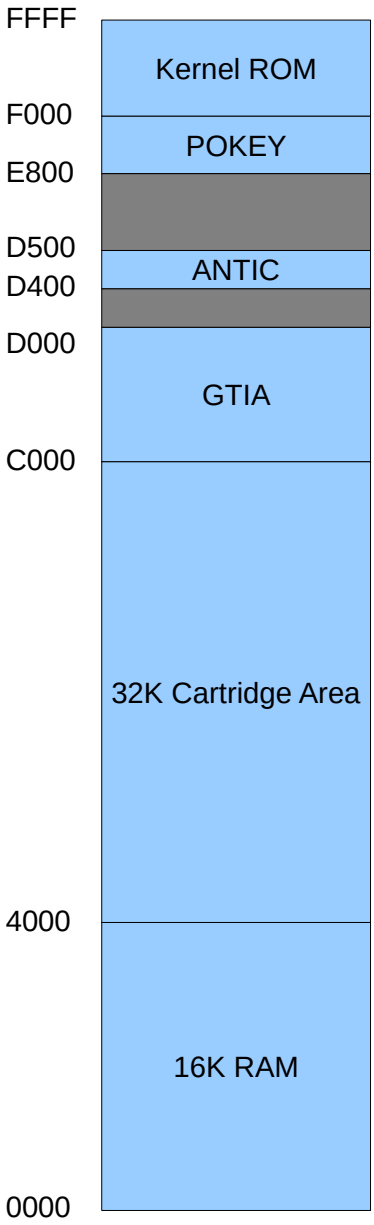
The keyboard debounce feature (SKCTL bit 0) must be disabled in order to detect a key press. If it is enabled, the keyboard logic will see the redundant keyboard mappings as multiple pressed keys and will never report a key press in SKSTAT, KBCODE, or IRQST. Disabling debounce also prevents POKEY from properly detecting a held key, however, and therefore in this mode each pressed key will be reported every 32 scan lines (~490Hz). If debounce is quickly enabled within one scan line after the key is reported, however, the keyboard logic will properly wait until the key is released before reporting any other key presses.

50 [AHS03a] p. 25.
51 [AHS03a] p. 21.

Triggers

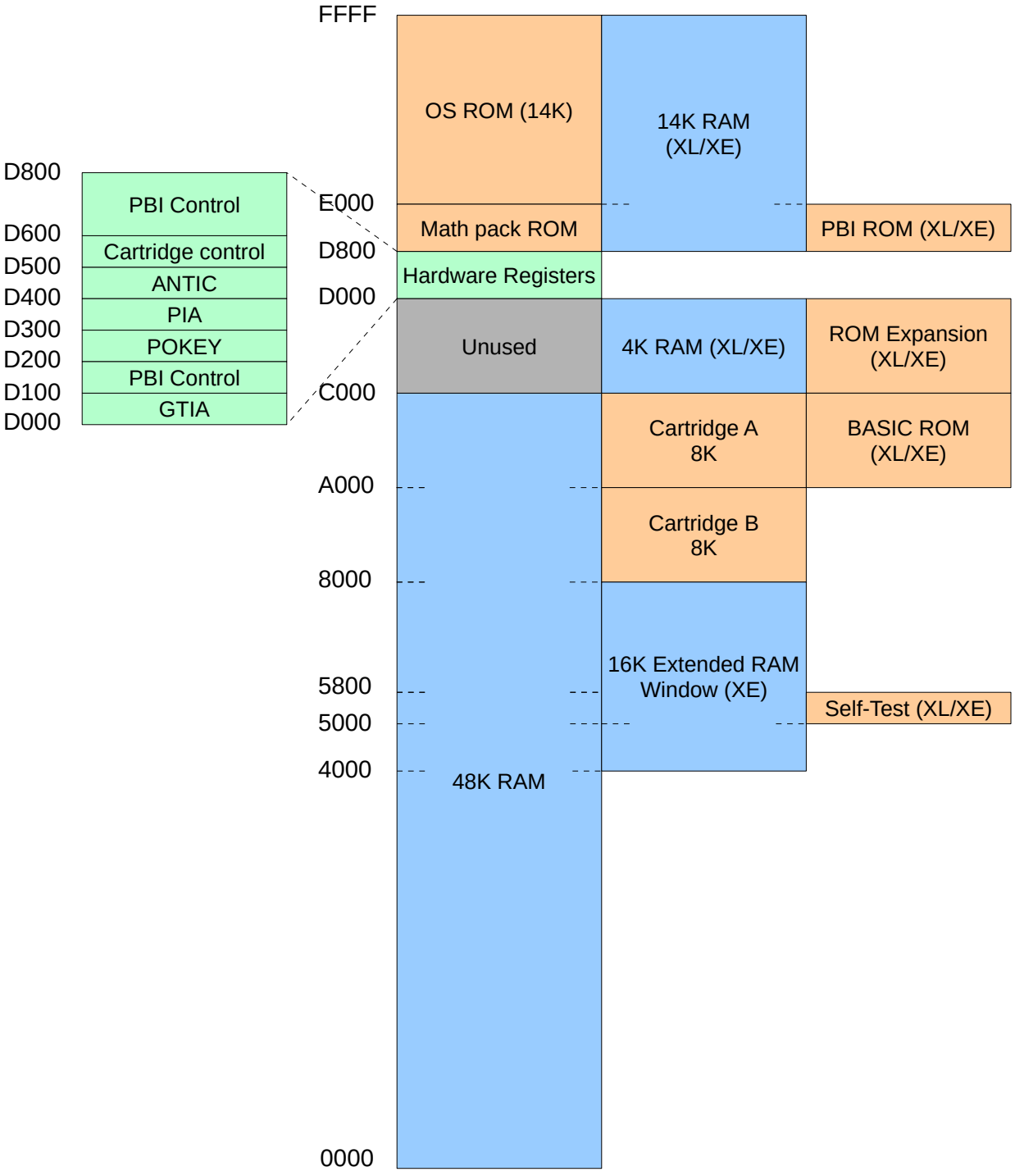
There are two trigger buttons on the 5200 controller, an upper trigger and a lower trigger. The bottom button of each controller is wired to TRIG0-TRIG3, depending on the controller, and functions the same as a joystick button on the 8-bit computer line. The top button is instead wired to the $\overline{KR2}$ line of POKEY, which causes it to register as the SHIFT, CONTROL, and BREAK keys on the scanned keyboard. This means that it will trigger a break IRQ (IRQST bit 7) as well as show up in the top two bits of KBCODE if a key pad button is pressed.

12.3 5200 Memory map



Chapter 13
Reference

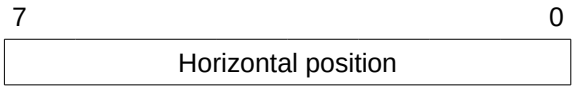
13.1 Memory map



13.2 Register list

Unit	Address	Description
GTIA	HPOSP0, HPOSP1, HPOSP2, HPOSP3 $D000-$D003	Player 0-3 horizontal position (Write Only)

Register layout



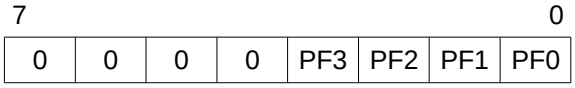
Description

HPOSP0-HPOSP3 control the position of the left edge of each of the four players, in color clocks. More precisely, they set the trigger point at which the shift register is loaded and begins shifting player graphics data through the collision and priority logic to the video output.

A position of $80 corresponds to the center of the playfield. The narrow playfield runs from $40-$BF, the normal playfield from $30-$CF, and the wide playfield from $22-$DD.

Unit	Address	Description
GTIA	M0PF, M1PF, M2PF, M3PF $D000-$D003	Missile-to-playfield collision registers (Read Only)

Register layout



D3:D0 Playfield 0-3 collision bits

0 No collision detected
1 Collision detected

Description

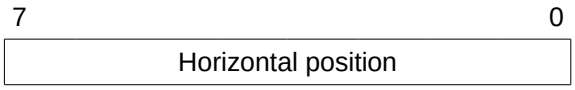
A bit is set in the M0PF, M1PF, M2PF, and M3PF registers whenever missiles 0-3 overlap a playfield in the visible region, but bit 0 being set for a collision with playfield 0. Overlaps in the horizontal or vertical blank region are not detected. Collisions are latched and stay flagged until HITCLR is written.

No playfield collisions are detected in GTIA modes 9 or 11. Playfield collisions are triggered normally for GTIA mode 10.

In high-resolution modes (ANTIC modes 2, 3, and F), the monochrome playfield is considered to be PF2. Either of the two pixels being set in the pair displayed during a color clock will signal a PF2 collision on that clock.

Unit	Address	Description
GTIA	HPOSM0, HPOSM1, HPOSM2, HPOSM3 $D004-$D007	Missile 0-3 horizontal position (Write Only)

Register layout



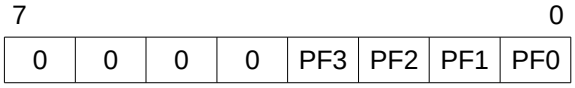
Description

HPOSM0-HPOSM3 control the position of the left edge of each of the four missiles, in color clocks. More precisely, they set the trigger point at which the shift register is loaded and begins shifting missile graphics data through the collision and priority logic to the video output.

A position of $80 corresponds to the center of the playfield. The narrow playfield runs from $40-$BF, the normal playfield from $30-$CF, and the wide playfield from $22-$DD.

Unit	Address	Description
GTIA	P0PF, P1PF, P2PF, P3PF $D004-$D007	Player-to-playfield collision registers (Read Only)

Register layout



D3:D0 Playfield 0-3 collision bits

0	No collision detected
1	Collision detected

Description

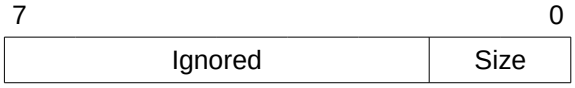
A bit is set in the P0PF, P1PF, P2PF, and P3PF registers whenever players 0-3 overlap a playfield in the visible region, but bit 0 being set for a collision with playfield 0. Overlaps in the horizontal or vertical blank region are not detected. Collisions are latched and stay flagged until HITCLR is written.

No playfield collisions are detected in GTIA modes 9 or 11. Playfield collisions are triggered normally for GTIA mode 10.

In high-resolution modes (ANTIC modes 2, 3, and F), the monochrome playfield is considered to be PF2. Either of the two pixels being set in the pair displayed during a color clock will signal a PF2 collision on that clock.

Unit	Address	Description
GTIA	SIZEP0, SIZEP1, SIZEP2, SIZEP3 $D008-$D00B	Player horizontal width control (Write Only)

Register layout



D1:D0 Player size

00	Normal width (1 color clock per bit)
01	Double width (2 color clocks per bit)
10	Normal width (1 color clock per bit)
11	Quadruple width (4 color clocks per bit)

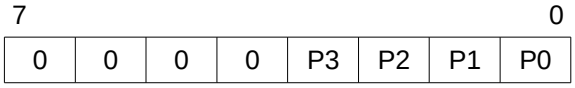
Description

SIZEP0-SIZEP3 control the horizontal width of each player by specifying how many color clocks to display each bit on screen. Since the horizontal position registers control the left side of each player, increasing the width causes players to expand to the right.

A change to SIZEPx while the corresponding player is being shifted out will take place immediately.

Unit	Address	Description
GTIA	M0PL, M1PL, M2PL, M3PL $D008-$D00B	Missile-to-player collision registers (Read Only)

Register layout



D3:D0 Player 0-3 collision bits

0	No collision detected
1	Collision detected

Description

A bit is set in the M0PL, M1PL, M2PL, and M3PL registers whenever missiles 0-3 overlap a player in the visible region, but bit 0 being set for a collision with player 0. Overlaps in the horizontal or vertical blank region are not detected. Collisions are latched and stay flagged until HITCLR is written.

Unit	Address	Description
GTIA	SIZEM $D00C	Missile horizontal width control (Write Only)

Register layout

7 0

Size 3	Size 2	Size 1	Size 0

D7:D6 Missile 3 size
D5:D4 Missile 2 size
D3:D2 Missile 1 size
D1:D0 Missile 0 size

00	Normal width (1 color clock per bit)
01	Double width (2 color clocks per bit)
10	Normal width (1 color clock per bit)
11	Quadruple width (4 color clocks per bit)

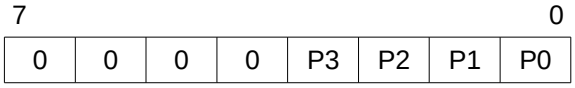
Description

SIZEM0-SIZEM3 control the horizontal width of each missile by specifying how many color clocks to display each bit on screen. Since the horizontal position registers control the left side of each missile, increasing the width causes missiles to expand to the right.

A change to SIZEM while the corresponding missile is being shifted out will take place immediately.

Unit	Address	Description
GTIA	P0PL, P1PL, P2PL, P3PL $D00C-$D00F	Player-to-player collision registers (Read Only)

Register layout



D3:D0 Player 0-3 collision bits

0	No collision detected
1	Collision detected

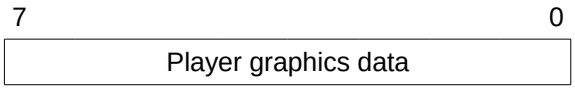
Description

A bit is set in the P0PL, P1PL, P2PL, and P3PL registers whenever two players overlap in the visible region, with bit 0 being set for a collision with player 0. Overlaps in the horizontal or vertical blank region are not detected. Collisions are latched and stay flagged until HITCLR is written.

A player never collides with itself and the corresponding collision bit is always 0.

Unit	Address	Description
GTIA	GRAFP0, GRAFP1, GRAFP2, GRAFP3 $D00D-$D010	Player graphics registers (Write Only)

Register layout



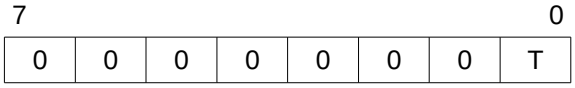
Description

GRAFP0-GRAFP3 hold the graphics data that is loaded into the shift register when each player is triggered by horizontal position. Normally player DMA is enabled on ANTIC when player graphics are used, which causes GRAFP0-GRAFP3 to be loaded automatically at the start of each scan line. When disabled, GTIA uses whatever data is in the internal latches. The latches can then be updated under CPU control, or simply left alone to display the same data on every scan line.

Data is displayed MSB to LSB, with the most significant bit being displayed on the left.

Unit	Address	Description
GTIA	TRIG0, TRIG1, TRIG2, TRIG3 $D010-$D013	Trigger registers (Read Only)

Register layout



D0 Trigger bit (inverted)

0 Trigger active
1 Trigger not active

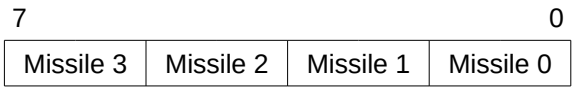
Description

TRIG0-3 reflect the state of the four joystick trigger inputs.

On the XL line, only two joystick ports are present and TRIG2 always reads as 1. TRIG3 is re-purposed as the cartridge detect line, reading 1 if cartridge ROM is mapped to $A000-BFFF and 0 otherwise.

Unit	Address	Description
GTIA	GRAFM $D011	Missile graphics register (Write Only)

Register layout



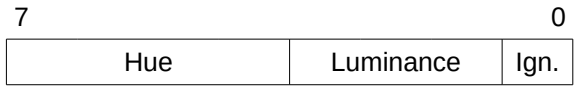
Description

GRAFM holds the graphics data that is loaded into the shift register when each missile is triggered by horizontal position. Normally missile DMA is enabled on ANTIC when missile graphics are used, which causes GRAFM to be loaded automatically at the start of each scan line. When disabled, GTIA uses whatever data is in the internal latch. The latch can then be updated under CPU control, or simply left alone to display the same data on every scan line.

Data is displayed MSB to LSB, with the most significant bit being displayed on the left.

Unit	Address	Description
GTIA	COLPM0, COLPM1, COLPM2, COLPM3 $D012-$D015	Player/missile 0-3 color register (Write Only)

Register layout

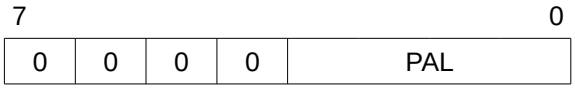


Description

These registers control the base colors used for players 0-3.

Unit	Address	Description
GTIA	PAL $D014	NTSC/PAL detect register (Read Only)

Register layout



D3:D0 NTSC/PAL detect

0001	PAL
1111	NTSC

Description

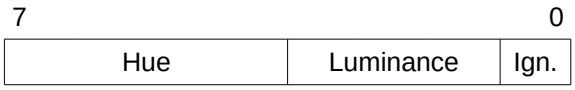
The PAL register indicates whether the GTIA is either the NTSC or PAL model.

Note that while the entire value read from the PAL register appears to be stable and consistent, only bits 1-3 are guaranteed to be set to a particular value according to the original specification.[52]

52 [ATA82] III.1

Unit	Address	Description
GTIA	COLPF0, COLPF1, COLPF2, COLPF3 $D016-$D019	Playfield 0-3 color register (Write Only)

Register layout



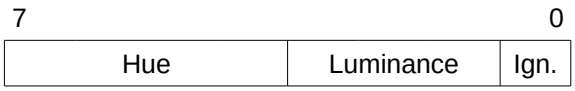
Description

These registers control the base colors used for playfields 0-3.

In ANTIC modes 2, 3, and F, COLPF2 controls the color of the playfield. A 1 bit in the graphics data replaces the luminance of a pixel with that from COLPF1.

Unit	Address	Description
GTIA	COLBK $D01A	Background color register (Write Only)

Register layout



Description

This register controls the color of the background, including the horizontal and vertical blank regions.

Unit	Address	Description
GTIA	PRIOR $D01B	Priority control (Write Only)

Register layout

7 0

GTIA	MC	P5	Priority mode

D3:D0 Playfield / P/M priority mode

1000	PF0 > PF1 > P0 > P1 > P2 > P3 > PF2 > PF3 > BAK
0100	PF0 > PF1 > PF2 > PF3 > P0 > P1 > P2 > P3 > BAK
0010	P0 > P1 > PF0 > PF1 > PF2 > PF3 > P2 > P3 > BAK
0001	P0 > P1 > P2 > P3 > PF0 > PF1 > PF2 > PF3 > BAK

D4 Fifth player enable

0	Missiles use player 0-3 colors
1	Missiles use playfield 3 color

D5 Multicolor player enable

0	Normal
1	Multicolor players enabled

D7:D6 GTIA mode enable

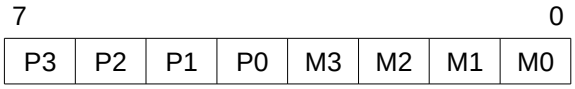
00	Normal
01	1 color / 16 luma mode
10	9 color mode
11	16 colors / 1 luma mode

Description

PRIOR controls a bunch of miscellaneous options, including player/missile priority relative to playfields. All of these options have complex interactions with the rest of the video display logic. See the CTIA/GTIA chapter for details.

Unit	Address	Description
GTIA	VDELAY $D01C	Vertical delay (Write Only)

Register layout



D7:D0 Vertical delay

0	Accept DMA data every scan line
1	Accept DMA data only on odd scan lines

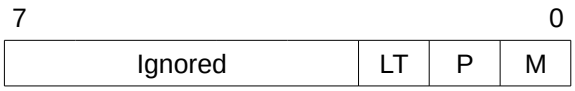
Description

VDELAY is used to vertically scroll players and missiles down by one scan line in two-line resolution mode. Contrary to its name, however, it doesn't actually delay anything. What it does is control whether GTIA loads the graphics latches from the data during DMA time on even scan lines. When a bit is set in VDELAY, the corresponding sprite only loads data on odd scan lines, which effectively moves the sprite down a scan line when two-line DMA mode is enabled. In single line mode, this has the effect of halving sprite resolution.

VDELAY has no effect on direct writes to the GRAFP0-3 or GRAFM registers.

Unit	Address	Description
GTIA	GRACTL $D01D	Graphics control (Write Only)

Register layout



D0 Missile DMA enable

0 Disabled
1 Missile DMA enabled

D1 Player DMA enable

0 Disabled
1 Player DMA enabled

D2 Trigger latch enable

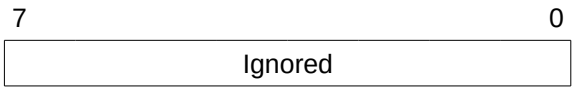
0 Trigger inputs are momentary
1 Trigger inputs are latched

Description

GRACTL controls player/missile DMA on the GTIA side. DMACTL in ANTIC must be set appropriately to enable P/M data to be fetched from memory, but GRACTL in GTIA must also be set for that data to be accepted into the GRAFP0-GRAFP3 and GRAFM registers.

Unit	Address	Description
GTIA	HITCLR $D01E	Collision control clear strobe (Write Only)

Register layout



Description

A write to HITCLR clears all of the collision registers.

Unit	Address	Description
GTIA	CONSOL $D01F	Console control (Read/Write)

Register layout

7							0
0	0	0	0	SPK	OPT	SEL	STA

D3 Loudspeaker

0 Source
1 Sink

D2 OPTION key

0 Asserted (read) / Source (write)
1 Inactive (read) / Sink (write)

D1 SELECT key

0 Asserted (read) / Source (write)
1 Inactive (read) / Sink (write)

D0 START key

0 Asserted (read) / Source (write)
1 Inactive (read) / Sink (write)

Description

CONSOL reads and writes the state of four bidirectional switch lines connected to GTIA. On the Atari, these are connected to the internal loudspeaker and the OPTION, SELECT, and START keys. Writing a 0 into a bit causes the corresponding switch line to be pulled up to +5V, and writing a 1 sinks it to ground.

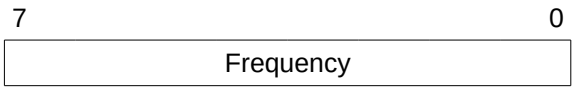
By default, the OS writes $08 into CONSOL during vertical blank.[53] This causes the CONSOL register to read $07 when no keys are pressed, with bits 0-2 going low when one of the console buttons is pressed. If a 1 is written into bits 0-2, the corresponding switch is grounded and always reads as a 0.

The XL series has no internal loudspeaker and thus the speaker output is routed to the TV instead.

53 [ATA82] III.15

Unit	Address	Description
POKEY	AUDF1, AUDF2, AUDF3, AUDF4 $D200, $D202, $D204, $D206	Audio channel 0-3 frequency (Write Only)

Register layout

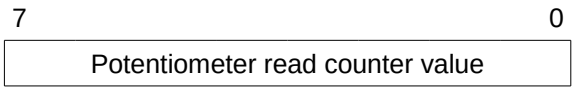


Description

AUDF1-AUDF4 control the frequency of the four audio channels.

Unit	Address	Description
POKEY	POT0-POT7 $D200-$D207	Potentiometer read counter (Read Only)

Register layout

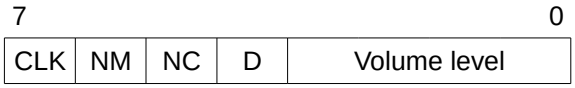


Description

POT0-POT7 indicate the value of each of the eight potentiometer read counters. When POTGO is written, each of the counters is reset to 0 and begins counting up until either the threshold or the value 228 has been hit. The corresponding bit in ALLPOT is then set to indicate that the pot counter value is valid.

Unit	Address	Description
POKEY	AUDC1, AUDC2, AUDC3, AUDC4 $D201, $D203, $D205, $D207	Audio channel 0-3 control (Write Only)

Register layout



D3:D0 Volume level

0000	Silent
0001	Lowest volume
1111	Highest volume

D4 Output disable

0	Normal operation
1	Volume-only mode

D5 Noise control

0	Sample noise source
1	Output pure tone (produce square wave by toggling output on clock pulse)

D6 Noise mode

0	Sample 9-bit or 17-bit polynomial generator (see AUDCTL bit 7)
1	Sample 4-bit polynomial generator

D7 Sampling clock mode

0	Mask out clock pulses using 5-bit polynomial generator
1	Use timer output directly as clock

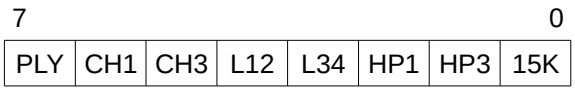
Description

AUDC1-AUDC4 control the volume and timbre of the four audio channels.

See the *Audio and Serial Port Block Diagram* page of the Hardware Manual [ATA82] for a logic diagram that shows precisely how the bits in AUDCx affect the output flow.

Unit	Address	Description
POKEY	AUDCTL $D208	Audio control (Write Only)

Register layout



D7 Polynomial select

- 0 RANDOM and audio channels use 17-bit polynomial generator
- 1 RANDOM and audio channels use 9-bit polynomial generator

D6 Channel 1 fast clock enable
D5 Channel 3 fast clock enable

- 0 Clock channel with 15KHz or 64KHz clock
- 1 Clock channel with 1.79MHz clock

D4 Channel 1+2 link enable
D3 Channel 3+4 link enable

- 0 Independent 8-bit counters
- 1 Linked 16-bit counter (clock 2 with 1 or 4 with 3).

D2 Channel 1 high pass filter enable
D1 Channel 2 high pass filter enable

- 0 Normal operation
- 1 High pass enabled (filter channel 1/2 with channel 3/4)

D0 Clock select

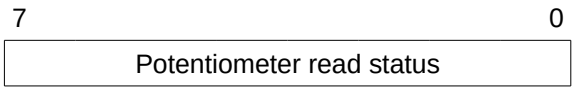
- 0 Use 64KHz as slow audio clock
- 1 Use 15KHz as slow audio clock

Description

AUDCTL controls a number of miscellaneous sound parameters.

Unit	Address	Description
POKEY	ALLPOT $D208	Potentiometer read status (Read Only)

Register layout



D7:D0 Pot 0-7 read status

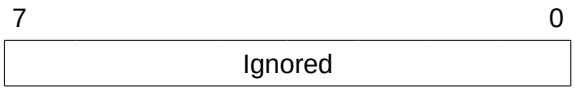
0	Potentiometer read complete
1	Potentiometer still being read

Description

ALLPOT indicates when each of the eight potentiometers have been read and the counter values are valid.

Unit	Address	Description
POKEY	STIMER $D209	Start timer strobe (Write Only)

Register layout



Description

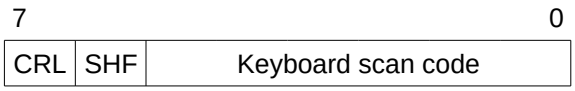
Writing to STIMER causes all timers to restart from their set period values, sets the output flip-flops for all channels to 0 (1 after inversion). When high-pass filters are disabled, this silences channels 1 and 2 and enables output for channels 3 and 4.

Errata

The POKEY datasheet [AHS03] states that STIMER forces channels 1 and 2 to logic high and channels 3 and 4 to logic low; this is backwards if logic high is the state that produces sound.

Unit	Address	Description
POKEY	KBCODE $D209	Keyboard code register (Read Only)

Register layout



D7 Control key state

- 1 Control key was down when key was pressed
- 0 Control key was not down when key was pressed

D6 Shift key state

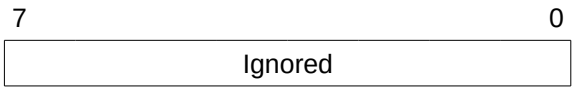
- 1 Shift key was down when key was pressed
- 0 Shift key was not down when key was pressed

Description

Contains the scan code of the most recently pressed key, along with the state of the Shift and Control keys when it was pressed. This register is only changed on a key press; it does not respond to a key release.

Unit	Address	Description
POKEY	SKRES $D20A	Serial/keyboard reset strobe (Write Only)

Register layout

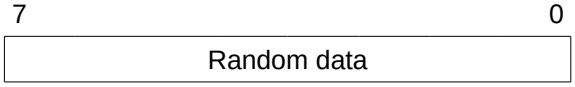


Description

Writing to SKRES resets the serial port and keyboard status bits in SKSTAT (bits 5-7).

Unit	Address	Description
POKEY	RANDOM $D20A	Random number generator (Read Only)

Register layout



Description

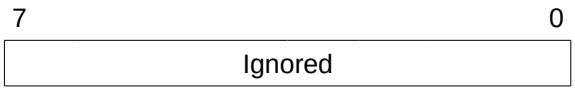
Reads the state of the top eight bits of the 17-bit polynomial noise generator. This generator counts at 1.79MHz and thus changes every cycle.

If bit 7 of AUDCTL is set, the 17-bit polynomial noise generator is shortened to 9 bits. This is reflected in the values read from RANDOM. Because the noise generator is a linear feedback shift register (LFSR) of the XOR variety, a state of all zeroes is invalid and therefore a RANDOM value of 00 is a unique LFSR state (all other values can be one of two states). From this state, the progression is as follows:

0: 00	43: 48	86: 94	129: ff	172: 70	215: e7	258: aa	301: a0	344: 45	387: cc	430: 3f	473: 86
1: 80	44: a4	87: 4a	130: ff	173: 38	216: 73	259: 55	302: d0	345: a2	388: 66	431: 9f	474: c3
2: 40	45: 52	88: 25	131: 7f	174: 9c	217: 39	260: aa	303: e8	346: d1	389: 33	432: 4f	475: 61
3: 20	46: a9	89: 12	132: 3f	175: ce	218: 1c	261: d5	304: 74	347: e8	390: 99	433: a7	476: b0
4: 10	47: 54	90: 09	133: 1f	176: 67	219: 0e	262: ea	305: ba	348: f4	391: 4c	434: d3	477: 58
5: 88	48: 2a	91: 04	134: 0f	177: 33	220: 07	263: f5	306: dd	349: fa	392: a6	435: 69	478: ac
6: 44	49: 15	92: 82	135: 87	178: 19	221: 03	264: fa	307: ee	350: fd	393: 53	436: b4	479: 56
7: 22	50: 8a	93: 41	136: c3	179: 0c	222: 81	265: 7d	308: f7	351: fe	394: a9	437: 5a	480: ab
8: 11	51: c5	94: 20	137: e1	180: 86	223: c0	266: be	309: fb	352: 7f	395: d4	438: ad	481: 55
9: 88	52: 62	95: 90	138: f0	181: 43	224: e0	267: 5f	310: 7d	353: bf	396: 6a	439: 56	482: 2a
10: c4	53: b1	96: c8	139: 78	182: 21	225: 70	268: af	311: 3e	354: 5f	397: 35	440: 2b	483: 95
11: 62	54: d8	97: 64	140: bc	183: 90	226: b8	269: d7	312: 1f	355: 2f	398: 9a	441: 15	484: ca
12: 31	55: 6c	98: 32	141: de	184: 48	227: dc	270: 6b	313: 8f	356: 97	399: 4d	442: 0a	485: e5
13: 98	56: 36	99: 99	142: ef	185: 24	228: ee	271: b5	314: c7	357: 4b	400: 26	443: 85	486: 72
14: 4c	57: 9b	100: cc	143: 77	186: 12	229: 77	272: 5a	315: e3	358: a5	401: 93	444: 42	487: 39
15: 26	58: cd	101: e6	144: 3b	187: 89	230: bb	273: 2d	316: f1	359: d2	402: c9	445: a1	488: 9c
16: 13	59: e6	102: 73	145: 1d	188: 44	231: 5d	274: 16	317: 78	360: 69	403: e4	446: 50	489: 4e
17: 89	60: f3	103: b9	146: 0e	189: a2	232: 2e	275: 0b	318: 3c	361: 34	404: f2	447: 28	490: 27
18: c4	61: f9	104: 5c	147: 87	190: 51	233: 97	276: 05	319: 9e	362: 1a	405: f9	448: 14	491: 13
19: e2	62: 7c	105: 2e	148: 43	191: a8	234: cb	277: 82	320: cf	363: 8d	406: fc	449: 8a	492: 09
20: 71	63: 3e	106: 17	149: a1	192: d4	235: e5	278: c1	321: 67	364: 46	407: 7e	450: 45	493: 84
21: b8	64: 9f	107: 8b	150: d0	193: ea	236: f2	279: 60	322: b3	365: a3	408: bf	451: 22	494: c2
22: 5c	65: cf	108: c5	151: 68	194: 75	237: 79	280: b0	323: 59	366: 51	409: df	452: 91	495: 61
23: ae	66: e7	109: e2	152: 34	195: ba	238: bc	281: d8	324: 2c	367: 28	410: 6f	453: c8	496: 30
24: 57	67: f3	110: f1	153: 9a	196: 5d	239: 5e	282: ec	325: 96	368: 94	411: b7	454: e4	497: 18
25: ab	68: 79	111: f8	154: cd	197: ae	240: af	283: 76	326: cb	369: ca	412: 5b	455: 72	498: 8c
26: d5	69: 3c	112: 7c	155: 66	198: d7	241: 57	284: bb	327: 65	370: 65	413: 2d	456: b9	499: 46
27: 6a	70: 1e	113: be	156: b3	199: eb	242: 2b	285: dd	328: b2	371: 32	414: 96	457: dc	500: 23
28: b5	71: 8f	114: df	157: d9	200: f5	243: 95	286: 6e	329: 59	372: 19	415: 4b	458: 6e	501: 11
29: da	72: 47	115: ef	158: 6c	201: 7a	244: 4a	287: b7	330: ac	373: 8c	416: 25	459: 37	502: 08
30: 6d	73: a3	116: f7	159: b6	202: 3d	245: a5	288: db	331: d6	374: c6	417: 92	460: 9b	503: 84
31: 36	74: d1	117: 7b	160: db	203: 9e	246: 52	289: 6d	332: eb	375: 63	418: 49	461: 4d	504: 42
32: 1b	75: 68	118: 3d	161: ed	204: 4f	247: 29	290: b6	333: 75	376: 31	419: 24	462: a6	505: 21
33: 8d	76: b4	119: 1e	162: f6	205: 27	248: 14	291: 5b	334: 3a	377: 18	420: 92	463: d3	506: 10
34: c6	77: da	120: 0f	163: 7b	206: 93	249: 0a	292: ad	335: 1d	378: 0c	421: c9	464: e9	507: 08
35: e3	78: ed	121: 07	164: bd	207: 49	250: 05	293: d6	336: 8e	379: 06	422: 64	465: f4	508: 04
36: 71	79: 76	122: 83	165: 5e	208: a4	251: 02	294: 6b	337: c7	380: 03	423: b2	466: 7a	509: 02
37: 38	80: 3b	123: c1	166: 2f	209: d2	252: 81	295: 35	338: 63	381: 01	424: d9	467: bd	510: 01
38: 1c	81: 9d	124: e0	167: 17	210: e9	253: 40	296: 1a	339: b1	382: 80	425: ec	468: de	
39: 8e	82: 4e	125: f0	168: 0b	211: 74	254: a0	297: 0d	340: 58	383: c0	426: f6	469: 6f	
40: 47	83: a7	126: f8	169: 85	212: 3a	255: 50	298: 06	341: 2c	384: 60	427: fb	470: 37	
41: 23	84: 53	127: fc	170: c2	213: 9d	256: a8	299: 83	342: 16	385: 30	428: fd	471: 1b	
42: 91	85: 29	128: fe	171: e1	214: ce	257: 54	300: 41	343: 8b	386: 98	429: 7e	472: 0d	

Unit	Address	Description
POKEY	POTGO $D20B	Potentiometer read start strobe (Write Only)

Register layout

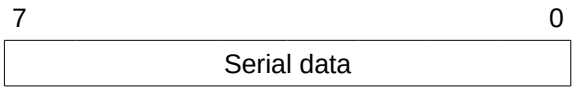


Description

Writing to POTGO dumps the potentiometer read capacitors and resets the pot counters, restarting the pot read process. This causes all POT0-POT7 registers to reset to 0 and ALLPOT becomes $FF until each pot is measured. In fast pot scan mode, the pots can be used as cycle timers, although the read values appear to be slightly unreliable while counting.

Unit	Address	Description
POKEY	SEROUT $D20D	Serial output register (Write Only)

Register layout

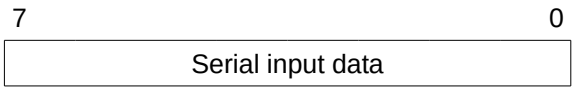


Description

SEROUT is written by the CPU to specify the data that should be copied to the serial output shift register and sent out to the SIO bus.

Unit	Address	Description
POKEY	SERIN $D20D	Serial input register (Read Only)

Register layout

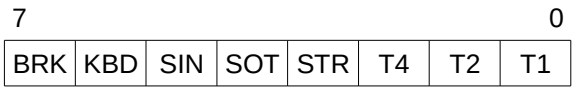


Description

Reads the data that was most recently shifted into POKEY from the SIO bus and clears the internal data-ready state. If two consecutive bytes are shifted in without SERIN being read in between, the second byte overwrites the first. This does not set the serial input overrun bit unless the serial input IRQ is still active – the overrun bit is based on the IRQ state and not whether SERIN is read.

Unit	Address	Description
POKEY	IRQEN $D20E	IRQ enable register (Write Only)

Register layout



D7 Break key interrupt
D6 Keyboard interrupt
D5 Serial input data ready interrupt
D4 Serial output data needed ready interrupt
D3 Serial output transmission completed interrupt
D2 Timer 4 expired interrupt
D1 Timer 2 expired interrupt
D0 Timer 1 expired interrupt

0 Disabled, reset associated status bit
1 Enabled

Description

IRQEN selectively enables or disables various IRQ sources within POKEY. Disabling an IRQ source via IRQEN also resets the associated status bit in IRQST and clears the interrupt if it is currently pending. The exception is the transmission complete bit in IRQST (bit 3), which is not reset by writes to IRQEN. As long as the serial output hardware is idle, setting bit 3 will immediately cause the serial output transmission interrupt to fire.

Unit	Address	Description
POKEY	IRQST $D20E	IRQ status register (Read Only)

Register layout

7							0
BRK	KBD	SIN	SOT	STR	T4	T2	T1

D7 Break key interrupt
D6 Keyboard interrupt
D5 Serial input data ready interrupt
D4 Serial output data needed ready interrupt
D2 Timer 4 expired interrupt
D1 Timer 2 expired interrupt
D0 Timer 1 expired interrupt

- 0 Interrupt pending
- 1 Not active or interrupt disabled

D3 Serial output transmission completed interrupt

- 0 Serial transmission completed
- 1 Serial transmission in progress

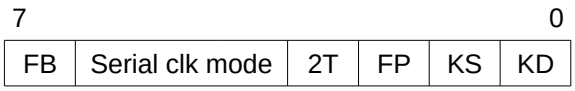
Description

IRQST indicates when various interrupts are pending from POKEY. These interrupts remain active and trigger at the end of the next instruction if the 6502 processor status bit I is cleared unless reset via IRQEN.

Most bits in IRQST are reset and stay low when the corresponding interrupt is cleared via IRQEN. The exception is the serial output transmission bit (bit 3), which is not latched and always indicates the current state.

Unit	Address	Description
POKEY	SKCTL $D20F	Serial/keyboard control (Write Only)

Register layout



D7 Force break

- 0 Serial data is output normally
- 1 Serial output line is forced to 0

D6:D5 Serial clock mode

D4 Asynchronous receive mode

- 0 Disabled
- 1 Enabled – use timer 4 as input clock and reset timers 3+4 when waiting for start bit or a zero is received

D3 Two-tone mode

- 0 Disabled – serial data is output directly on bus
- 1 Enabled – audio channels 1 and 2 output on bus for a 1 and a 0, respectively

D2 Fast pot scan

- 0 Slow pot scan: counters increment every 114 cycles
- 1 Fast pot scan: counters increment every cycle

D1 Enable keyboard scan

D0 Enable keyboard debounce

- 0 Disabled
- 1 Enabled
- 00* Special case – initialize

Description

SKCTL controls a number of miscellaneous serial port, keyboard, and pot scan functions in POKEY. See chapter 5.4, Serial port for more details.

Unit	Address	Description
POKEY	SKSTAT $D20F	Serial/keyboard status (Read Only)

Register layout

7							0
SF	SO	KO	SD	SH	KY	SI	1

D7 Serial input frame error
- 0 Framing error detected in serial data

D6 Keyboard overrun error
- 0 Keyboard overrun detected: new key pressed while keyboard interrupt (IRQST bit 6) active

D5 Serial input overrun error
- 0 Serial input overrun detected: new serial input byte received while serial input interrupt (IRQST bit 5) active

D4 Serial input data line state
- 0 Serial input data line low
- 1 Serial input data line high

D3 Keyboard SHIFT key state
- 0 A SHIFT key is depressed
- 1 No SHIFT keys are depressed

D2 Key depressed state
- 0 A non-modifier key is currently depressed
- 1 No non-modifier keys are depressed

D1 Serial input shift register busy
- 0 Serial byte currently being received

Description

SKSTAT reports the status of several keyboard and serial port functions. It is primarily used to determine if an error has occurred during serial reception. A write to SKRES resets the serial input frame, serial input overrun, and keyboard overrun bits.

Unit	Address	Description
PIA	PORTB $D301	Port B data/direction register (Read/Write)

Register layout

7 0

Direction bits							Direction (PBCTL bit 2 = 0)
Jack 4				Jack 3			400/800 only
S	Unused			L2	L1	B	K	1200XL only
S	Unused					B	K	600XL/800XL
S	Un.	A	C	Bank		B	K	130XE only

D7:D0 Direction bits (PBCTL bit 2 = 0)

0 Input
1 Output

D0 Kernel ROM enable (XL/XE)

0 Map RAM at $D800-FFFF
1 Map Kernel ROM at $D800-FFFF

D1 BASIC ROM enable (XL/XE)

0 BASIC ROM enabled at $A000-BFFF
1 BASIC ROM disabled

D3:D2 Console LED 1 and 2 states (1200XL only)

0 LED on
1 LED off

D3:D2 Extended bank select (130XE only)

00 Map $10000-$13FFF as extended bank
01 Map $14000-$17FFF as extended bank
10 Map $18000-$1BFFF as extended bank
11 Map $1C000-$1FFFF as extended bank

D4 CPU extended memory access enable (130XE only)
D5 ANTIC extended memory access enable (130XE only)

0 Extended bank at $4000-7FFF
1 Primary bank at $4000-7FFF

D7 Self-test ROM enabled (XL/XE)

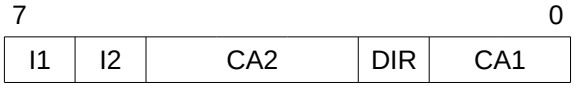
0 Map self-test ROM from $D000-$D7FF to $5000-57FF if kernel ROM is enabled
1 Disable self-test ROM

Description

PORTB originally accessed joystick ports 3 and 4 on the 800, but in later models with only two joystick ports it was re-purposed for various other features.

Unit	Address	Description
PIA	PACTL $D302	Port A control register (Read/Write)

Register layout



D7 IRQA1 status (read only)
D6 IRQA2 status (read only)

- 0 No interrupt pending
- 1 Interrupt pending

D5:D3 CA2 (SIO motor line) I/O mode

- 000 Input: set IRQA2 on negative transition, interrupt disabled
- 001 Input: set IRQA2 on negative transition, interrupt enabled
- 010 Input: set IRQA2 on positive transition, interrupt disabled
- 011 Input: set IRQA2 on positive transition, interrupt enabled
- 100 Output: lower on PORTA read until CA1 transition
- 101 Output: pulse low for one cycle on PORTA read
- 110 Output: assert (lower) motor line
- 111 Output: negate (raise) motor line

D2 Data direction register enable

- 0 PORTA [D300] accesses data direction register
- 1 PORTA [D300] accesses input and output registers

D1 CA1 (SIO proceed line) edge detection mode

- 0 Set IRQA1 on negative transition
- 1 Set IRQA1 on positive transition

D0 CA1 (SIO proceed line) interrupt enable

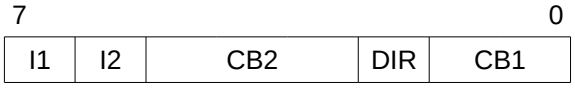
- 0 IRQA1 disabled
- 1 IRQA1 enabled

Description

PACTL controls the operation of port A and the PORTA register on the PIA. There are many more options supported by the PIA than documented here; consult [MOS76] for full details.

Unit	Address	Description
PIA	PBCTL $D303	Port B control register (Read/Write)

Register layout



D7 IRQB1 status (read only)
D6 IRQB2 status (read only)

- 0 No interrupt pending
- 1 Interrupt pending

D5:D3 CB2 (SIO command line) I/O mode

- 000 Input: set IRQB2 on negative transition, interrupt disabled
- 001 Input: set IRQB2 on negative transition, interrupt enabled
- 010 Input: set IRQB2 on positive transition, interrupt disabled
- 011 Input: set IRQB2 on positive transition, interrupt enabled
- 100 Output: lower on PORTB write until CB1 transition
- 101 Output: pulse low for one cycle on PORTB write
- 110 Output: assert (lower) command line
- 111 Output: negate (raise) command line

D2 Data direction register enable

- 0 PORTB [D301] accesses data direction register
- 1 PORTB [D301] accesses input and output registers

D1 CA1 (SIO interrupt line) edge detection mode

- 0 Set IRQB1 on negative transition
- 1 Set IRQB1 on positive transition

D0 CA1 (SIO interrupt line) interrupt enable

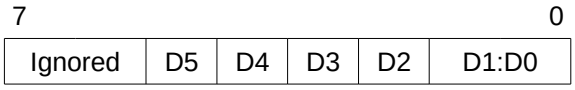
- 0 IRQB1 disabled
- 1 IRQB1 enabled

Description

PBCTL controls the operation of port B and the PORTB register on the PIA. There are many more options supported by the PIA than documented here; consult [MOS76] for full details.

Unit	Address	Description
ANTIC	DMACTL $D400	DMA control (Write Only)

Register layout



D1:D0 Playfield width

00	Disabled
01	Narrow playfield (128 color clocks)
10	Normal playfield (160 color clocks)
11	Wide playfield (192 color clocks)

D2 Missile DMA enable

0	Disabled (ignored if player DMA is enabled)
1	Enabled

D3 Player DMA enable

0	Disabled
1	Enabled

D4 Player/missile vertical resolution

0	Two-line resolution
1	One-line resolution

D5 Display list DMA enable

0	Disabled
1	Enabled

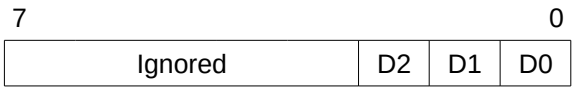
Description

The DMACTL register selectively enables DMA from ANTIC for various display items. For players and missiles, DMA mode must also be enabled in GTIA for it to take effect; otherwise, ANTIC will run DMA cycles but the object graphics will not be updated.

Missile DMA is enabled whenever player DMA is enabled, even if bit 2 is cleared. This is needed since GTIA interprets bus data depending on the number of cycles since the first time $\overline{\text{HALT}}$ is asserted during horizontal blank, and thus the timing of the missile DMA cycle determines which data is used for players.

Unit	Address	Description
ANTIC	CHACTL $D401	Character control (Write Only)

Register layout



D0 Character blink enable

- 0 Disabled
- 1 Hide characters with name bit 7 set

D1 Character invert

- 0 Disabled
- 1 Invert image of characters with name bit 7 set

D2 Vertical reflect

- 0 Display rows 0 through 7 (normal)
- 1 Display rows 7 through 0 (reflected)

Description

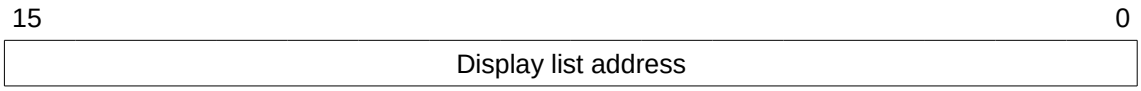
CHACTL controls various features of 40 column text modes (ANTIC modes 2 and 3).

The blink bit does not actually cause characters to blink – it only selectively hides or shows some characters. To actually blink text, the blink bit must be periodically toggled.

Vertical reflection is performed by inverting the bits of the row counter used to fetch character data. This means that reflection may not work as expected for ANTIC mode 3 since the special case mapping for the descendant rows is not affected.

Unit	Address	Description
ANTIC	DLISTL/DLISTH $D402/$D403	Display list address (Write Only)

Register layout



Description

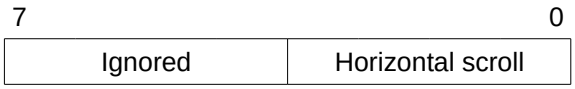
Set the current display list fetch address. Any writes to this register immediately redirect the display list, so it is recommended that it only be changed during vertical blank.[54]

The display list hardware only has a ten-bit counter. Display lists may be located anywhere in memory, but may not cross a 1K boundary without a jump instruction.[55]

54 [ATA82] III.6
55 [ATA82] III.5

Unit	Address	Description
ANTIC	HSCROL $D404	Horizontal scroll offset (Write Only)

Register layout



D3:D0 Horizontal delay in color clocks

Description

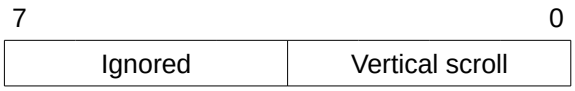
This register adjusts the horizontal scroll amount for mode lines that have display list mode bit 4 set. Data display can be delayed by up to 15 color clocks, scrolling the playfield to the right. This does not affect the timing of the displayed window, so the left and right displayed margins for narrow and normal width playfields are not affected.

Playfield fetch timing is delayed by one cycle for every two color clocks of scroll. Odd values have the same fetch timing as even values, with the additional delay coming from an internal one-clock delay.

Odd delay values will give unexpected results for GTIA modes since the boundaries of the pixels are not adjusted to match the fetch delay. This causes pairs of bits to be pulled from adjacent pixels to form the four-bit values used for display.

Unit	Address	Description
ANTIC	VSCROL $D405	Vertical scroll offset (Write Only)

Register layout



D3:D0 Vertical delay in color clocks

Description

This register adjusts the vertical scroll amount for mode lines in a vertical scroll region. This includes any mode line with display list instruction bit 5 set and the next mode line after that.

For the first mode line in a vertically scrolled region, the VSCROL register sets the index of the first row displayed in the mode line. Increasing the scroll amount therefore shortens the first mode line by removing scan lines from the top. For the last mode line in a vertically scrolled region, increasing scroll values extends the last mode line by adding scan lines from the bottom.

It is possible to set VSCROL such that the row counter counts through values not normally valid for a mode line. When this happens, the mode line is extended as the row counter counts up to 15 and wraps around to 0. For text modes, only the low three bits are used to fetch data and thus rows 8-15 display the same data as rows 0-7.

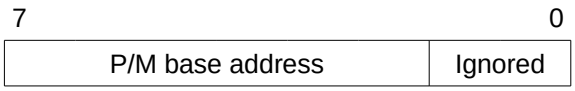
VSCROL must be written by cycle 0 at the beginning of a mode line to affect the start of a scrolling region and by cycle 109 to determine whether the next scan line is the last scan line of an scroll-ending mode line.

Errata

The hardware manual [ATA82] shows only the lowest three bits being significant for 8-line display modes, but all four bits are significant in all display modes.

Unit	Address	Description
ANTIC	PMBASE $D407	Player/missile base address (Write Only)

Register layout



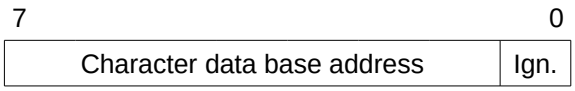
D7:D2 Bits 10-15 of P/M base address (two-line resolution)
D7:D3 Bits 11-15 of P/M base address (one-line resolution)

Description

PMBASE sets the base address for fetching player/missile graphics. For one-line resolution, only the top five bits can be set, and therefore the P/M data must be aligned to a 2K boundary. For two-line resolution, the top six bits are settable and 1K alignment is required.

Unit	Address	Description
ANTIC	CHBASE $D409	Character data base address (Write Only)

Register layout



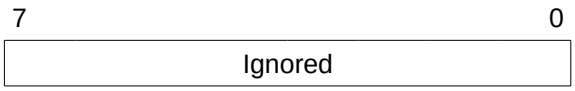
D7:D1 Bits 9-15 of character data base address (ANTIC modes 2, 3, 4 and 5)
D7:D2 Bits 10-15 of character data base address (ANTIC modes 6 and 7)

Description

CHBASE sets the base address for fetching character data. Each character consists of an 8x8 block of monochrome data and occupies eight contiguous bytes. For ANTIC modes 2-5, CHBASE points to 128 characters starting at a 1K boundary, and for ANTIC modes 6-7, 64 characters starting at a 512 byte boundary.

Unit	Address	Description
ANTIC	WSYNC $D40A	Wait for Horizontal Sync (Write Only)

Register layout



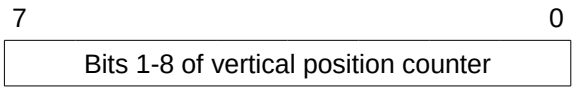
Description

A write to WSYNC causes the CPU to halt execution until the start of horizontal blank. One more cycle passes before the CPU is halted until cycle 105 on the current scan line. If the next cycle is free, the CPU executes the first cycle of the next instruction; otherwise, the next instruction starts at cycle 105. DMA contention at cycles 105 and 106 may cause the CPU restart to be delayed until as late as cycle 107.

Because the 6502 can only service an interrupt at the end of an instruction, use of WSYNC can cause excessively long delays in servicing interrupts. This is most serious with display list interrupts, where the delay can cause DLIs to occur on the wrong scan line or to be missed entirely.

Unit	Address	Description
ANTIC	VCOUNT $D40B	Vertical count (Read Only)

Register layout



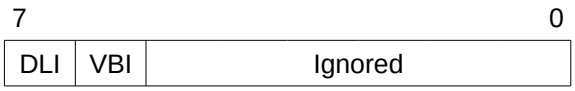
Description

VCOUNT allows the vertical position counter to be read to two-line resolution. For NTSC, VCOUNT runs from 0 to 131; for PAL, it runs from 0 to 156.

The VCOUNT register increments on cycle 110 of a scan line.

Unit	Address	Description
ANTIC	NMIEN $D40E	Non-maskable interrupt enable (Write Only)

Register layout



D7 Display list interrupt enable

0 Disabled
1 Enabled

D6 Vertical blank interrupt enable

0 Disabled
1 Enabled

Description

NMIEN enables and disables NMI interrupts issued by ANTIC. This is required since the 6502 itself does not allow masking the NMI. Both interrupts are disabled automatically on system reset.[56]

The reset interrupt cannot be masked through NMIEN.[57]

56 Hardware II.28
57 Hardware III.1

Unit	Address	Description
ANTIC	NMIST $D40F	Non-maskable interrupt status (Read Only)

Register layout

7							0
DLI	VBI	RES	1	1	1	1	1

D7 Display list interrupt status

0 Inactive
1 Active

D6 Vertical blank interrupt status

0 Inactive
1 Active

D5 System reset interrupt status (400/800 only)

0 Inactive
1 Active

Description

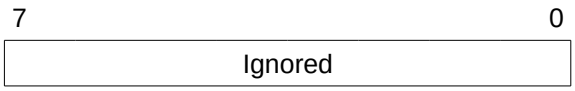
NMIST indicates which interrupt source in ANTIC triggered an NMI. The register layout is arranged so that a single BIT instruction can be used to very quickly check the DLI and VBI sources. A write to NMIRES is then used to clear status bits once the interrupt is serviced.

The DLI bit is automatically cleared when the VBI bit is set at scan line 248. Therefore, it is ordinarily never necessary to strobe NMIRES for either interrupt, as testing the DLI bit is sufficient to distinguish the two.

On the XL/XE series, the system reset button is hooked up to the RESET line rather than ANTIC's RNMI line, and thus the system reset NMI never occurs.

Unit	Address	Description
ANTIC	NMIRES $D40F	Non-maskable interrupt reset (Write Only)

Register layout



Description

A write to NMIRES resets the interrupt status bits in the NMIST register. This is only necessary for the NMI service routine to continue to identify the source of each interrupt – unlike for IRQs, the NMI is edge-triggered and therefore NMIRES does not need to be written to clear the interrupt itself.

Typically, NMIRES is only written when handling the vertical blank interrupt and not display list interrupts, because DLIs handlers are time critical. ANTIC assists this by automatically clearing the DLI bit in NMIST at the start of scan line 248.

13.7 Register listing

Unit	Address	Name	Desc	Bits							
GTIA	D000 (R)	M0PF	Missile/playfield collision	0			PF3	PF2	PF1	PF0	
	D001 (R)	M1PF									
	D002 (R)	M2PF									
	D003 (R)	M3PF									
	D004 (R)	P0PF	Player/playfield collision								
	D005 (R)	P1PF									
	D006 (R)	P2PF									
	D007 (R)	P3PF									
	D008 (R)	M0PL	Missile/player collision				P3	P2	P1	P0	
	D009 (R)	M1PL									
	D00A (R)	M2PL									
	D00B (R)	M3PL									
	D00C (R)	P0PL	Player/player collision							0	
	D00D (R)	P1PL							0	P0	
	D00E (R)	P2PL						0	P1		
	D00F (R)	P3PL					0	P2			
	D010 (R)	TRIG0	Joystick triggers	0						T0	
	D011 (R)	TRIG1								T1	
	D012 (R)	TRIG2								T2	
	D013 (R)	TRIG3								T3	
	D014 (R)	PAL	NTSC/PAL detect	$01 for PAL, $0F for NTSC							
	D000 (W)	HPOSP0	Player 0 position	Horizontal position in color clocks							
	D001 (W)	HPOSP1	Player 1 position								
	D002 (W)	HPOSP2	Player 2 position								
	D003 (W)	HPOSP3	Player 3 position								
	D004 (W)	HPOSM0	Missile 0 position								
	D005 (W)	HPOSM1	Missile 1 position								
	D006 (W)	HPOSM2	Missile 2 position								
	D007 (W)	HPOSM3	Missile 3 position								
	D008 (W)	SIZEP0	Player 0 size	Ignored					x0: Normal 01: Double 11: Quad		
	D009 (W)	SIZEP1	Player 1 size								
	D00A (W)	SIZEP2	Player 2 size								
	D00B (W)	SIZEP3	Player 3 size								
	D00C (W)	SIZEM	Missile sizes	M3		M2		M1		M0	
	D00D (W)	GRAFP0	Player graphics latch	Player graphic data							
	D00E (W)	GRAFP1									
	D00F (W)	GRAFP2									
	D010 (W)	GRAFP3									
	D011 (W)	GRAFM	Missile graphics latch	M3		M2		M1		M0	

Unit	Address	Name	Desc	Bits							
	D012 (W)	COLPM0	Player/missile colors	Hue				Luminance			Ign.
	D013 (W)	COLPM1									
	D014 (W)	COLPM2									
	D015 (W)	COLPM3									
	D016 (W)	COLPF0	Playfield colors								
	D017 (W)	COLPF1									
	D018 (W)	COLPF2									
	D019 (W)	COLPF3									
	D01A (W)	COLBK	Background color								
	D01B (W)	PRIOR	Priority control	GTIA mode		MC	PL5	Priority mode			
	D01C (W)	VDELAY	Vertical delay	P3	P2	P1	P0	M3	M2	M1	M0
	D01D (W)	GRACTL	Graphics control	Ignored					LT	P	M
	D01E (W)	HITCLR	Collision clear strobe	Ignored							
	D01F (R/W)	CONSOL	Console switches	0				SPK	$\overline{\text{OPT}}$	$\overline{\text{SEL}}$	$\overline{\text{STA}}$
PBI	D1FF (R)	PDVI	PBI device interrupt	each 1 bit = device interrupt pending							
	D1FF (W)	PDVS	PBI device select	$00 = none, single bit = select device							
POKEY	D200 (R)	POT0	Paddle (pot) positions	Paddle position (0-228)							
	D201 (R)	POT1									
	D202 (R)	POT2									
	D203 (R)	POT3									
	D204 (R)	POT4									
	D205 (R)	POT5									
	D206 (R)	POT6									
	D207 (R)	POT7									
	D208 (R)	ALLPOT	Direct pot. read	P7	P6	P5	P4	P3	P2	P1	P0
	D209 (R)	KBCODE	Keyboard code	CRL	SHF	Scan code					
	D200 (W)	AUDF1	Audio channel frequency	Period - 4 (8-bit) Period - 7 (16-bit)							
	D202 (W)	AUDF2									
	D204 (W)	AUDF3									
	D206 (W)	AUDF4									
	D201 (W)	AUDC1	Audio channel control	$\overline{\text{5-bit}}$	4-bit noise	$\overline{\text{Noise}}$	Vol. only	Volume			
	D203 (W)	AUDC2									
	D205 (W)	AUDC3									
	D207 (W)	AUDC4									
	D208 (W)	AUDCTL	Audio control	9-bit	Fast 1	Fast 3	1+2	3+4	Hi1	Hi2	15K
	D209 (W)	STIMER	Start timer strobe	Ignored							
	D20A (R)	RANDOM	Random number gen.	Random number							
	D20A (W)	SKRES	Serial/keyboard reset	Ignored							
	D20D (R)	SERIN	Serial input data	Received serial data							
	D20D (W)	SEROUT	Serial output data	Serial data to transmit							

Unit	Address	Name	Desc	Bits							
	D20E (R) D20E (W)	IRQST IRQEN	IRQ status IRQ enable	Brk	Key	Sin	Sout	Scmp	T#4	T#2	T#1
	D20F (R)	SKSTAT	Serial/keyboard status	$\overline{\text{Frm}}$	$\overline{\text{KOv}}$	$\overline{\text{SIOv}}$	SDir	Shift	KDwn	SIBs	1
	D20F (W)	SKCTL	Serial/keyboard control	FBrk	S.Clock		Asyn	2Tn	FPot	KScn	KDb
PIA	D300 (R/W)	PORTA	Port A data/direction	Joystick 2				Joystick 1			
	D301 (R/W)	PORTB	Port B data/direction	STst		$\overline{\text{CPU}}$	$\overline{\text{ANT}}$	ExtBank		BAS	$\overline{\text{OS}}$
	D302 (R/W)	PACTL	Port A control	IRQ1	IRQ2	CA2 (SIO motor)			DDR	CA1 (SIOInt)	
	D303 (R/W)	PBCTL	Port B control	IRQ1	IRQ2	CB2 (SIO cmd.)			DDR	CB1 (SIOPr)	
ANTIC	D400 (W)	DMACTL	DMA control	Ignored		DList	2Line	Plyr	Mssl	PF Width	
	D401 (W)	CHACTL	Character control	Ignored					Blink	Invert	Refl.
	D402 (W)	DLISTL	Display list addr low	Display list address bits 7-0							
	D403 (W)	DLISTH	Display list addr high	Display list address bits 15-8							
	D404 (W)	HSCROL	Horizontal scroll	Ignored				Horizontal scroll right			
	D405 (W)	VSCROL	Vertical scroll	Ignored				Vertical scroll down			
	D407 (W)	PMBASE	Player/missile base	Player/missile base address bits 15-10							
	D409 (W)	CHBASE	Character set base	Character set address bits 15-9							Ign.
	D40A (W)	WSYNC	Wait for horizontal sync	Ignored							
	D40B (R)	VCOUNT	Vertical count	Vertical counter bits 8-1							
	D40E (W)	NMIEN	NMI enable	DLI	VBI	Ignored					
	D40F (R)	NMIST	NMI status	DLI	VBI	RES	1				
	D40F (W)	NMIRES	NMI reset strobe	Ignored							

Chapter 14
Bibliography

[6502Dec] Clark, Bruce, Decimal Mode, 2009. Retrieved on June 28, 2009 from http://www.6502.org/tutorials/decimal_mode.html.

[AHS00] Atari, CGIA CO20577 (NTSC), 2000.

[AHS03] Atari, POKEY CO12294, 2003.

[AHS03a] Atari Historical Society, PAM Package, 2003. Retrieved on July 5, 2015 from http://www.atarimuseum.com/ahs_archives/archives/archives-techdocs-5200.htm.

[AHS05] Atari, Sweet 16 OS supplement 3, .

[AHS99] Atari, ANTIC CO12296 (NTSC) Rev. D, 1999.

[AHS99a] Atari, GTIA CO14805 (NTSC), 1999.

[ATA82] Atari, Atari Home Computer System Hardware Manual, 1982.

[ATA87] Atari, Product Specification for XEP80 80 Column and Parallel Printer Board, 1987.

[ATAXL] Atari, Atari Home Computer System Operating System Manual, XL Addendum, .

[CHA85] Chadwick, Ian, Mapping the Atari, Revised Edition, 1985.

[CRA82] Crawford, Chris, De Re Atari, 1982.

[EYE86] Eyes, David and Lichty, Ron, Programming the 65816, 1986.

[IJO10] ijor, Flags on decimal mode on the NMOS 6502, 2010. Retrieved on July 5, 2015 from http://www.atariage.com/forums/topic/163876-flags-on-decimal-mode-on-the-nmos-6502/.

[IllOpc] Offenga, Freddy, 6502 Undocumented Opcodes, 1999. Retrieved on May 5, 2015 from http://www.ataripreservation.org/websites/freddy.offenga/illopc31.txt.

[LAN84] Lancaster, Don, Assembly Cookbook for the Apple II/IIe, 1984.

[MOS76] MOS Technology, MCS6500 Microcomputer Family Hardware Manual, 1976.

[MOS76a] MOS Technology, MCS6500 Microcomputer Family Programming Manual, 1976.

[MyIDE-II] Tucker, Steven, myide_ii_hardware_registers.txt, 2012. Retrieved on July 5, 2015 from http://atarimax.com/flashcart/forum/viewtopic.php?f=17&t=1306.

[ObWrap] Anomie, Anomie's SNES OpenBus & Wrapping Doc, 2007. Retrieved on May 17, 2015 from https://github.com/gilligan/snesdev/blob/master/docs/ob-wrap.txt.

[TIVideoDec] Texas Instruments, TVP5020 NTSC/PAL Video Decoder Data Manual, 2000. Retrieved on July 5, 2015 from http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=slas186&fileType=pdf.

[U1MB] Bartkowicz, Sebastian, Ultimate1MB Programmer's Reference, 2015. Retrieved on July 5, 2015 from http://spiflash.org/block/20.html.

[VBXE] T. Piórek, VideoBoard XE FX Core version 1.26 Programmer's Manual, 2013. Retrieved on July 5, 2015 from http://spiflash.org/block/15.html.

[VIC09] VICE team, Documentation for the NMOS 65xx/85xx Instruction Set, 2009. Retrieved on July 19, 2009 from http://vice-emu.sourceforge.net/plain/64doc.txt.

Appendix A
Polynomial Counters

POKEY's noise generators use a form of polynomial counter called a *linear feedback shift register*. This is a type of pseudo-random number generator that is cheap to implement in hardware and generates a long sequence of bits with a short shift register.

Generated sequence

A maximal-length polynomial counter N bits wide has a cyclical sequence of 2^N-1 bits. The missing value is due to all-zeroes being a lock-up state, which otherwise is not generated in the sequence. This means that the sequence has one fewer 0 bit than 1 bits. The value of any N contiguous bits in the sequence is unique within the sequence.

The sequence length is of consequence when sampling the sequence as regular intervals, as POKEY does when using the polynomial counters for noise generation. Sampling periods that have common factors with the sequence length will reduce the effective sequence length. In particular, POKEY's 4-bit counter has a period of 15 = 5·3, and the 9-bit counter has a period 511 = 73·7.

Feedback polynomials

Each polynomial counter has a characteristic feedback polynomial associated with it that determines how new bits are generated from existing bits in the shift register. Each term in the polynomial corresponds to a specific delay. For instance, the polynomial for the 4-bit counter in POKEY is as follows:

$$x^4 + x^3 + 1$$

The “1” corresponds to the newly produced bit, whereas x^3 is the third bit back and x^4 is the fourth bit back. Also, addition is modulo-2 here, or equivalent to XOR. The tap x^N always exists for an N-bit counter, or else the top bit would not contribute to sequence length.

Polynomial counter simulation

The polynomial for POKEY's 9-bit generator, on the other hand, is:

$$x^9 + x^4 + 1$$

It is possible to simulate the values produced by the RANDOM register, given one other pieces of information, namely that it shifts right. Given this, the sequence can be generated by the following C code:

```
unsigned v = 0x80;

do {
      printf("%02x\n", v & 0xff);
      v = (v >> 1) + (((v << (9-1)) ^ (v << (4-1))) & 0x100);
} while(v != 0x80);
```

This produces the sequence in the order seen by the 6502, by XORing the bits at both taps together and shifting it in on the left.

Alternate configuration

Running the feedback in the opposite direction, shifting bits out and XORing them in at each tap, allows for an alternate method of simulation:

```
unsigned v = 1<<8;

do {
    printf("%02x\n", v & 0xff);

    v >>= 1;
```

```
    if (v & 0x80)
        v ^= (1 << (9+7)) + (1 << (4+7));
} while((v & (0x1ff << 8)) != 1);
```

This form is easier to simulate in software as the feedback is done via a scatter rather than gather operation. However, while it produces the same sequence of bits, the shift register state is different, and so an extra 8 bits of shift register are necessary to capture the same output values.

This is the same algorithm implemented in 6502 code:

```
        lda    #0
        sta    xrandom      ;initialize random output
        lda    #1           ;initialize shift register bits 0-7
        clc                 ;initialize shift register bit 8
loop:
        ror                 ;shift
        php                 ;save shift register bit 8
        ror    xrandom      ;contains newly generated byte
        plp                 ;restore shift register bit 8
        bcc    loop         ;skip if 0 bit shifted out
        eor    #$08         ;xor in x^4 tap (x^9 tap done via carry)
        bcs    loop         ;continue
```

Seeking to arbitrary positions

A shortcoming of the above algorithms is that they are limited to sequentially producing states. This is fine for noise generation or building tables, but can be a hassle when random access by position is needed. Fortunately, it is also possible to compute the shift register's state at any position in $O(\log^2 N)$ time.

The alternate simulation algorithm works by storing base-2 polynomials in binary numbers, where each bit i represents an element x^i in a polynomial. That means the algorithm computes the following in base-2 polynomial arithmetic:

$$x^i \bmod (x^9+x^4+1)$$

x^i, in turn, can be represented as the product of powers of two, i.e. x^0, x^1, x^2, x^4, x^8, etc. That leads to the following strategy:

- Precompute a table of $x^{2^i} \bmod M$.
- Given a position P, multiply together the appropriate values of $x^{2^i} \bmod M$ for each bit i set in P to determine the shift register state at position P.
- Run the shift register from that position to retrieve the desired output stream bits.

This is most useful with the 17-bit shift register ($x^{17}+x^{12}+1$), whose output sequence can be too impractically large to precompute. The following C code generates the register values seen by RANDOM after exiting initialization mode, with the 17-bit shift register enabled:

```
uint32_t powers[17];


uint32_t polymul(uint32_t x, uint32_t y, uint32_t base, uint32_t hibit) {
    uint32_t accum = 0;

    while(y) {
        if (y & 1)
            accum ^= x;

        y >>= 1;
```

```
        x <<= 1;

        if (x & hibit)
            x ^= base;
    }

    return accum;
}


uint32_t polyeval17(uint32_t idx, uint32_t initial_state) {
    uint32_t x = initial_state;

    for(int i=0; i<17; ++i) {
        if (idx & 1)
            x = polymul(x, powers[i], 0x21001, 0x20000);

        idx >>= 1;
    }

    return x;
}

int main() {
    // precompute powers of 2 mod polynomial
    powers[0] = 2;

    for(int i=1; i<17; ++i)
        powers[i] = polymul(powers[i-1], powers[i-1], 0x21001, 0x20000);

    // evaluate RANDOM at every possible position
    for(int i=0; i<131071; ++i) {
        uint32_t v = polyeval17((131071 - i + 4) % 131071, 0x1FFFF) << 8;

        for(int j=0; j<8; ++j) {
            v >>= 1;

            if (v & 0x80)
                v ^= (1 << (17+7)) + (1 << (12+7));
        }

        printf("%02x\n", v & 0xff);
    }
}
```

Appendix B
Physical Disk Format

B.1 Raw geometry

Single density

A single density disk contains 40 tracks of 18 sectors with 128 bytes per sector, giving a total of 90KB of storage. The tracks are spaced at 48 tracks per inch (tpi), so a more modern 80 track drive with 96tpi needs to double-step to read and write a single density disk. Only the first (bottom) side is used.

Enhanced (medium) density

An enhanced or medium density disk contains 40 tracks of 26 sectors with 128 bytes per sector, giving 130KB of storage. Track density is the same as for single density.

Double density

A double density disk contains 40 tracks of 18 sectors with 256 bytes per sector, giving 180KB of storage. Track density is the same as for single density.

For compatibility with the OS boot routines, the first three sectors of a double-density disk – track 0, sector 1 through 3 – are exposed to the computer as 128 byte sectors. However, they are still encoded on the physical disk as 256 byte sectors like the rest of the disk. The disk drive firmware extends the sector to 256 bytes on write and discards the extra bytes on read. The 128 bytes used are at the beginning of the sector and the other 128 bytes are usually $00.

Track/sector number conversion

All disk formats use the same method to convert between the sector numbers used by the SIO disk protocol and the track/sector numbers used on the physical disk: the sectors are numbered in sequential order starting at track 0 and going up to track 39. For a single density disk, SIOsector = track*18 + sector.

Index position

Neither the index mark in the track nor the index sensor is used. Tracks may start at any angular position on the disk, and in particular sectors may lie across the index position. During formatting, tracks are laid out such that there is skew between tracks and sector 1 lies at a different angular position on each track.

B.2 Bit encoding

Bit cell encoding

All data bits are encoded into a pair of bit cells before being written to disk. The first cell is the clock cell, used to maintain synchronization of the decoder's bit cell clock, and the second cell is the data cell. A '1' bit in either cell corresponds to a flux transition on disk, where the local magnetic field reverses, and detected on read as a pulse; a '0' bit is the absence of flux transition and corresponding read pulse.

Bits are stored on disk in MSB-to-LSB order. No framing (start/stop) bits are used, so bit- and byte-level synchronization is achieved through special synchronization bytes, and then maintained from then on for the field being read. There are no requirements for bit or byte timing to be synchronized between fields.

FM encoding (single density)

On single density disks, frequency modulation (FM) encoding is used. Each data bit is encoded as a pair of bit cells, the first being a clock bit that is always 1, and the second being the data bit. The clock bit ensures that the decoder sees enough clock transitions to be able to lock onto the bit cell timing used by the encoder. A bit cell is

4µs long at 288 RPM, giving approximately 52,000 bit cells per revolution, or ~3,250 raw data bytes per track.

For synchronization at the byte level, a special pattern of clock bits is used for data bytes that mark the beginning of address and data fields. For these, some of the clock bits encoded as 0 instead of 1 (missing flux transition), giving a pattern of $C7 instead of $FF.

Note that the decoder does not always validate the clock bits. They are checked in order to identify the special mark bytes, but otherwise do not need to be set as long as enough 1 bits are present in both clock and data bit cells to satisfy timing requirements. This is exploited by some copy protection schemes that omit clock bits within the data field to encode a $C7 clock byte, allowing the data field of one sector to overlap the address field of another to fit an otherwise impossible number of sectors on a track (36!).

MFM encoding (enhanced and double density)

The increased capacity of enhanced and double density formats is achieved with modified frequency modulation (MFM) encoding, which reduces the density of flux transitions (1 bits) and uses tighter timing to cram in more bit cells. Unlike FM, which always encodes a 1 bit in clock cells, MFM encodes a 0 in a clock cell if either of the two adjacent data cells has a 1. This allows the bit cell to be halved to 2µs at 288 RPM, encoding ~104,100 bit cells per revolution and ~6,500 raw data bytes per track.

Like FM, unusual clock bit patterns are used for synchronization, but the patterns are different. For MFM, a sequence of three $A1 bytes is used prior to each DAM/IDAM, with a clock pattern of $0A. In some literature, this is given as a combined shift pattern of $4489. The DAM/IDAM itself has a normal clock pattern.

B.3 Address field

The address field marks the beginning of a sector and includes the track, head, and sector numbers, as well as the size of the sector. It is written once during formatting and then only read afterward.

For MFM, the address field starts with three synchronization bytes ($0A clock / $A1 data), followed by the IDAM of $FE. For FM, it starts with the $FE IDAM with a $C7 clock. After that is the zero-based track number (0-39), the zero-based head number, the one-based sector number (1-18 or 1-26), and the sector size (0=128, 1=256, 2=512, 3=1024), and finally two CRC bytes.

In FM, at least one $00 data byte must precede the IDAM for it to be recognized.

When searching for a sector, the FDC checks the track and sector numbers of each address field. A mismatch in track number indicates that a seek error has occurred and causes the FDC to recalibrate and re-seek; a mismatch in sector number causes the FDC to continue searching until it either finds the desired sector or times out. The head number, however, is not checked by the FDC.

B.4 Data field

Following the address field is the data field for the sector, marked by a Data Address Mark (DAM). The DAM is normally $FB, but can also be values $F8-FA to indicate a deleted or custom type sector. This is interpreted as an error by Atari-compatible disk drives and used by some copy protection schemes since it cannot be written through the standard disk protocol.

As with the address field, the DAM is preceded by three $A1 synchronization bytes in MFM, and preceded by $00 and encoded with $C7 clock in FM. It is then immediately followed by the data bytes for the sector, and then two CRC bytes.

Data inversion

The original Atari 810 Disk Drive used a floppy drive controller that had an inverted data bus without a compensating inversion between the FDC and the CPU. As a result, all data is written to and read from disk inverted by this drive. This practice was maintained for compatibility reasons and continued with the MFM-

encoded enhanced- and double-density formats. However, synchronization, address, and CRC bytes are generated by the FDC itself and therefore not inverted.

B.5 CRC algorithm

Both index marks and sector data are protected by a 16-bit CRC to reasonably detect data corruption. The CRC has a polynomial of $x^{16}+x^{12}+x^5+1$. The initial value of the CRC register is all ones ($FFFF), with the checksum being shifted left as new bits are shifted in on the right, MSB first. The resulting CRC value is then stored or checked against the CRC stored after the checked region, stored MSB-first.

The following C code computes the disk CRC-16:

```
uint16_t ComputeCRC(const uint8_t *buf, uint32_t len) {
    uint16_t crc = 0xFFFF;

    for(uint32_t i=0; i<len; ++i) {
        uint8_t c = buf[i];

        crc ^= (uint16_t)c << 8;

        for(int j=0; j<8; ++j) {
            uint16_t feedback = (crc & 0x8000) ? 0x1021 : 0;

            crc += crc;
            crc ^= feedback;
        }
    }

    return crc;
}
```

FM format

For address fields in the FM format, the CRC includes the ID Address Mark (IDAM) byte of $FE, followed by the track, head, sector, and sector size bytes, for a total of five bytes. The address CRC of the first boot sector (track 0, sector 1) is $D2C3.

For data fields in the FM format, the CRC includes the Data Address Mark (DAM) byte of $F8-FB, followed by the 128 data bytes, for 129 bytes covered. These are the raw data bytes as seen by the FDC, so they are inverted from the data bytes seen by the computer. The CRC for a sector of all $00s as seen by the computer ($FF on disk) is $A580.

MFM format

The basic method of CRC calculation and checking is the same as for MFM, but the bytes covered differ slightly due to changes in synchronization. For both address and data fields, the three $A1 synchronization bytes before the IDAM or DAM are also included in the CRC, giving a total of eight bytes checksummed for addresses and 132 for data fields. The CRCs for track 0, sector 1 containing all $00s from the computer's perspective are $EA2D and $9A17.