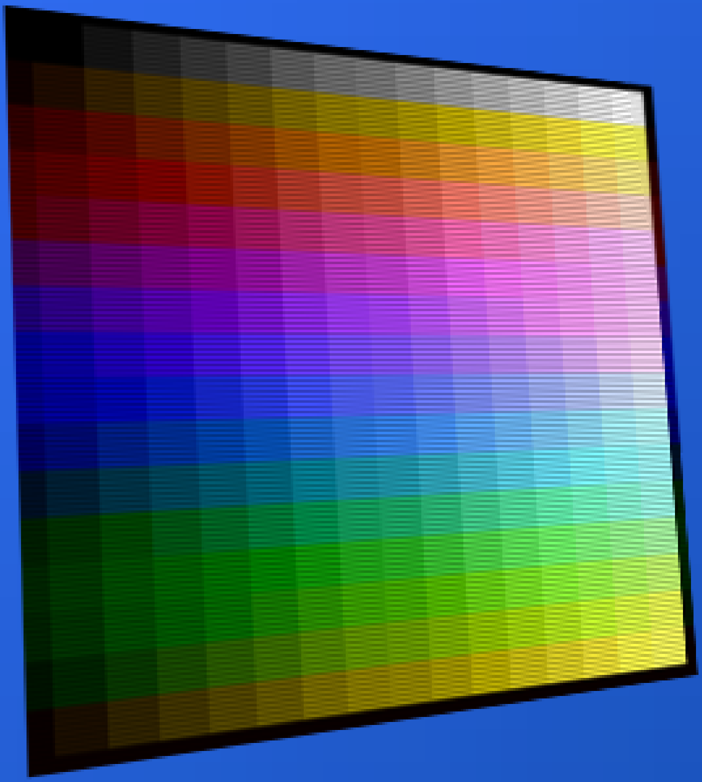


Altirra Hardware Reference Manual

2022-11-29 Edition

Avery Lee

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. You can also find the latest version of this manual there.

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

-- Avery Lee

1.2 What's new in this edition

This release

- Disk drives
  - Added info about XF551 firmware rev. 7.4 and 7.7 differences.
  - Added US Doubler firmware information.
- POKEY
  - More details on initialization mode.
  - Expanded on many audio topics.
  - Rotated around some of the noise generator patterns to canonically start from initialization state.
  - Cycle-precise timer behavior, including STIMER and AUDF1-4 write timing.

2022-07-07 release

- Fixed some typos referring to the PIA chip number as 6522 instead of 6520.
- CPU
  - Fixed typo regarding B flag, which is pushed *set* on the stack for a BRK and *cleared* for IRQ/NMI.
- ANTIC
  - Fixed typo in Display list > Suspended display list DMA: the last IR byte is repeated when display list DMA is *disabled*.
  - Fixed incorrect /RNMI timing – it only needs to be asserted across one leading edge of VBLANK, not two.
- Accessories
  - Expanded information on light pens.
  - Added ComputerEyes Video Acquisition System.
- Disk drives
  - Added info on 1050 firmware revision E.

2022-01-03 release

- ANTIC
  - Refresh row address counter is not cleared on reset.
- Disk drives
  - Clarified receive timing for Happy 1050 and added info about error handling when track buffering is enabled.
  - Added step timing for Speedy 1050.

- Parallel Bus Interface
  - Corrected reserved ranges for PBI ($D6xx and D7xx instead of $D5xx and D6xx).
- Appendices
  - Added appendix on physical tape format.

2021-10-02 release

- System architecture
  - Added brief overview of top-level system architecture.
- CPU
  - Expanded discussion of flags.
- ANTIC
  - Fixed erroneous starting cycles for playfield DMA in the text (the charts were correct).
- POKEY
  - Clarified some details about counter timing and distortion selection.
  - Added section on two-tone counter timing.
- Accessories
  - Note on XEP80 baud rate limits, row advance timing anomaly, and delete line behavior.
- Cartridges
  - New section on SIDE 3.
- Serial I/O bus
  - New section on 820 Printer.
- Disk drives
  - Removed Indus GT from list of disk drives that do not update PERCOM block on density detection.
  - Expanded information on PERCOM disk drive firmware revisions.
- Physical disk format
  - Added measured capture range for bit cell periods.
- Reference
  - Fixed swapped mode reference for the CHBASE register.
  - Fixed missing POTGO register in register listing.

2020-10-23 release

- Disk drives
  - Fixed swapped PB0 and PB7 definitions for 1050 drives.
  - Added behavior of various drives with no disk inserted.

- Clarified FDC handling of the head/side address field value.
- Added firmware revision information for 1050, Indus GT, and Percom RFD.
- Added Atari 815.
- Added Percom AT88-S1 and AT88-SPD.

- Internal devices
  - Added Bit-3 Full-View 80.
  - Added Atari 1090 80 Column Video Card.

2019-12-30 release

- System control
  - Fixed a typo in the PAL clock rate.
- ANTIC
  - Timing and behavior for the 400/800 System Reset NMI, even on XL/XE computers.
- GTIA
  - Clarified listed behavior for mode 9 combining with the fifth player (PF3) and for mode 11 regarding unusual COLBK values.
- POKEY
  - Fixed a typo in the keyboard layout for the Return key.
- Disk drives
  - Fixed 810 port B entries being listed in reversed order.
  - Added information on the 810 Turbo, Astra 1001 / The "One", and Amdek AMDC-I/II.
  - Added info on specific behaviors of the 6532 RIOT and 177X/277X floppy drive controllers.
  - Added info on Happy 1050 US Doubler emulation data corruption bug.
  - Added info on XF551 format behavior.
- Internal devices
  - Corrected VBXE blitter pattern width bitfield from 7 to 6 bits.
  - Added section on APE Warp+ 32-in-1.
- New appendices for analog video and audio models.

2018-08-12 release

- CPU: Clarified exact rules for when a branch crosses a page.
- CPU: Fixed some erroneous illegal instructions in the 6502 opcode chart.
- ANTIC: Fixed wrong modes being listed for 512 byte / 1K character set size.
- POKEY: Additional information about high-pass filter timing.

- GTIA:
  - New section on NTSC and PAL artifacting.
  - Clarified behavior of GR.9/GR.11 with fifth player or background having non-zero luma for GR.9 or hue for GR.11.
- Disk: Added information about task sequencing in disk drive controllers, I.S. Plate, sector interleaving order used by disk drive firmware, XF551 FDC error codes, long sector behavior, 810 revision B firmware, US Doubler hardware and commands, fixed incorrect sector ranges for XF551 back side encoding.

2017-05-17 release

- Additional light pen information.
- Rewritten and expanded section on POKEY’s serial port hardware, including precise timing diagrams.
- MyIDE-II CompactFlash reset behavior.
- SX212 power-on behavior.
- New chapter on disk drives, including information on the Happy 810, Happy 1050, ATR8000, Percom RFD-40S1, and the hardware for the 810, 1050, XF551, and Indus GT.

2016-03-25 release

- 65C816 opcode table.
- 800 floating I/O data bus.
- POKEY: Additional details on serial port behavior and keyboard and paddle scans.
- Additional XEP-80 details.
- Controllers: CX-20 Driving Controller, CX-21/23/50 Keyboard Controller.
- New device info: Indus GT disk drive, Corvus Disk Interface, Pocket Modem.
- 810 and 1050 updates: long sector behavior, FDC status.
- SIDE 2 corrections.
- Physical disk format: sector length behavior.

2015-07-05 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.

2014-04-27 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.

2013-05-14 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.

2012-09-15 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.

2010-11-23 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, 6502 conventions are used. Numbers without a prefix are given in base 10 (decimal), numbers prefixed by $ are given in base 16 (hexadecimal), and numbers prefixed by % are base 2 (binary).

In sections that describe Z80-based devices, Intel-style hex conventions are used instead with hex numbers ending in H, i.e. 50H.

Checksums

Where CRC32 values are given for firmware data, the CRC32 polynomial and algorithm is the same as that used by the Zip archive format, zlib, gzip, and the PNG image format. This is not to be confused with CRC32C, which uses a different polynomial and produces different values for the same data.

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 Concepts

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.

The opposite case is a big endian convention, where the higher order byte comes before the lower order byte. The 6809 is an example of a CPU that uses big endian byte ordering, and this endianness is also used in Percom block data because the convention originated in Percom's 6809-based disk drives.

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.

A *bit reversal* or *reverse bits* operation flips the order of the bits, exchanging bits 0 and 7, bits 1 and 6, etc. This has a few applications, including horizontally flipping 1-bit playfield or player/missile graphics, and compensating for different shift orders in serial protocols.

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. For instance, although the GTIA's HPOSP0 register is set by writing to $D000, it can't be read at that address; attempting to read $D000 gives M0PF instead.

There are a few notable exceptions where registers are read/write, such as CONSOL in GTIA and the data direction registers in the PIA. Even in those cases, it is often that the read address does not exactly read back the same register as the write address. For instance, reading CONSOL or PORTA doesn't actually read back the values written to the write register; it actually reads the input port connected to the same signal as the output port controlled by the write register, which means it can be different when the signal is being driven externally (e.g. the Start button being held down).

Partial registers

In some cases, an address maps to registers that have less than 8 bits. In the case of a write, the extra bits are ignored and lost. For a read, the extra unused bits are usually driven to a stable state by the chip, but this is not always the case. For example, the R-Time 8 only drives the low four data bits and leaves the higher ones floating. The PDVI register of the Parallel Bus Interface at $D1FF is a more extreme example, as it is actually a composite of single status bits from each device, leaving the bits for non-present devices floating.

Shadow registers

Because not being able to read back write-only registers makes saving and restoring registers difficult, the OS maintains a number of *shadow registers* in the kernel database to allow reading back the value of those registers. By updating the shadow register whenever the hardware register is updated, the hardware register can be "read back" by reading the shadow register instead. This is purely a software convention, however, and using shadow registers is not required. It also requires an additional write for every update to the hardware register.

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 (aliasing or mirroring)

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. This is also called *aliasing*, because two or more addresses serve as aliases for the same memory location.

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.

Overlapped decoding

As multiple circuits can independently decode addresses, it is possible for more than one device to decode and respond to the same address, when they are on the same bus. As the MMU handles most decoding in the base computer with non-overlapping address ranges, this is more common when the devices decode addresses independently of the MMU or off of the same MMU select signal.

In the case of a write, all devices accept and respond to the write in parallel, using the same data. This occurs in some stacked cartridge configurations, for instance, where the bottom cartridge does not exclude its register ranges from the forwarded signals passed to the pass-through cartridge port.

Overlapping reads result in a tug-of-war between the devices over the data bus, with conflicts causing one device to try to pull the data line down to 0 while the other to 1, and the result depending on which device wins. This is considered an undesirable electric condition due to the two devices effectively tying +V to ground, but is typically short-lived enough to avoid any ill-effects other than garbled returned data.

Power-on state

Individual hardware states may or may not be defined on power-on. The reset logic does ensure that circuits that take a reset signal are reset on power-up, and critical states such as NMIEN in ANTIC and the OS ROM enable are reset to ensure that the system can boot reliably. However, some states are not reset and are undefined on power-on.

In practice, non-reset states do have some bias as to their power-on state. The strongest bias is toward their last state, if the computer was powered off and then back on for a short period. The logic circuits will tend to keep their last state for up to several seconds, with increasing chance that the state is randomly lost with each second. If too much time passes and the system is powered up "cold", however, then the power-on state will tend to be influenced by the circuit design, which will bias the circuit strongly toward a particular polarity. This is the reason that dynamic RAM tends to power up with a characteristic stripe pattern that matches the memory bank configuration inside of the chip. In a "lukewarm" boot where the system has been powered off long enough for some but not all of the memory bits to have decayed, memory or hardware state bits will show a noisy mix of the last powered state and the natural cold power-on state of the circuits.

In some other cases where the circuit is well-balanced, the power-on state may essentially be random noise, such as with static RAMs.

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
System Architecture

2.1 Basic architecture

The Atari 800 series of computers uses an 8-bit architecture based on the MOS 6502 CPU, assisted by several custom chips and expandable by cartridge, serial, and parallel buses. Figure 1 shows the general architectural block diagram.

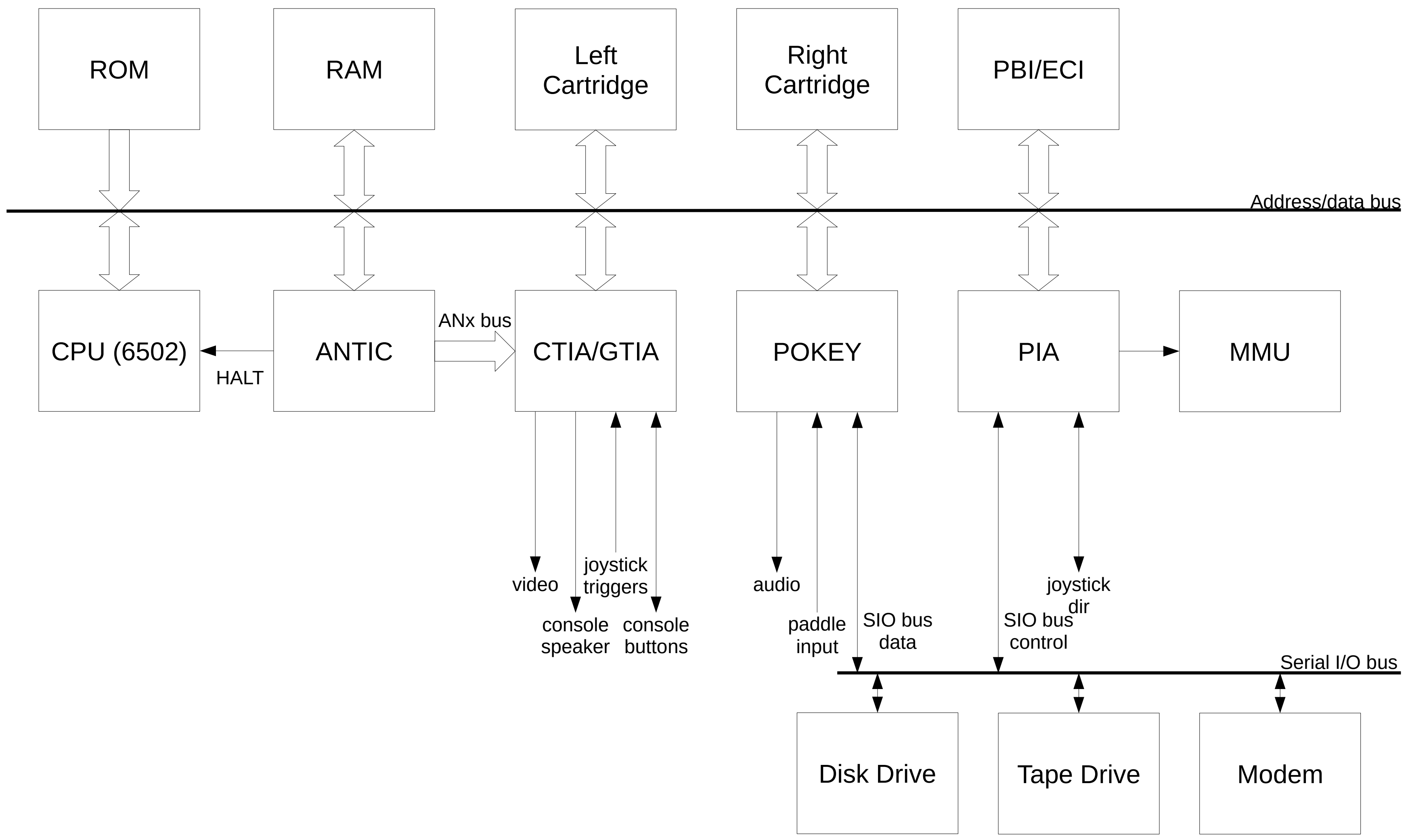


Figure 1: System Block Diagram

CPU

The CPU is a MOS 6502, with an 8-bit data bus and 16-bit address bus, capable of addressing 64K of memory. Some versions of the computer use a customized variant called a 6502C, which contains additional HALT logic built-in; it is not to be confused with the later and different 65C02, a revised CMOS version of the 6502.

ANTIC

The ANTIC custom chip handles DMA, memory refresh, display timing, and playfield graphics decoding for the system. It is the only other bus master in the system, halting the 6502 whenever it needs access to the memory bus for DMA or memory refresh.

CTIA/GTIA

The GTIA custom chip handles graphics generation, combining playfield data from ANTIC with player/missile graphics data and convert it to color video output.

The earliest computers may have a precedessor of the GTIA called the CTIA, which is missing some graphics functionality. The majority of computers have the GTIA.

GTIA also handles some auxiliary signals such as joystick triggers, console buttons, and the console speaker.

POKEY

The third custom chip, POKEY, handles sound generation, serial I/O bus data, and keyboard and paddle input.

PIA (6520)

A 6520 Peripheral Interface Adapter provides joystick directional I/O, serial I/O bus control signals, and on the XL/XE series, MMU control.

Some computers may use the 6820 or 6821/68B21 chips instead. They are pin- and software-compatible with the 6520.

MMU

The MMU maps memory address ranges to RAM, ROM, hardware, and expansion ports. On later models, it also supports banking in and out the ROMs and bank-switched expansion memory.

ROM

The operating system firmware is contained in a 10K ROM. Later models expand this to a 16K ROM containing a larger OS including a self-test, and also add an 8K internal BASIC ROM and an 8K game ROM.

RAM

8K to 128K of RAM is present, depending on the model. 16K, 64K, and 128K are the most common on stock, unmodified machines. Models containing more than 64K have the additional memory accessible as bank-switched expanded memory.

Cartridge port

The 800 contains two cartridge ports, a left cartridge port and a right cartridge port. Other models, including the 400, dropped the right cartridge port and only have the left port, which can support the same functions as the right port. The cartridge ports can support both reads and writes through up to a 16K aperture.

Joystick ports

Either two or four joystick ports are present, depending on the model, with the earlier 400/800 having four ports. The 9-pin joystick ports can support joysticks, paddles, driving controls, keypads, and even bidirectional parallel I/O.

Serial I/O bus

The serial I/O bus supports multiple peripherals on a daisy chain, including disk drives, cassette tape decks, printers, modems, and more exotic devices like MIDI ports. A standard communication protocol at 19200 baud allows the computer to address individual peripherals sharing the SIO bus.

Parallel Bus Interface (PBI) / Enhanced Cartridge Interface (ECI)

The Parallel Bus Interface on later models allows external logic to interface directly to the address and data buses of the computer for high-speed traffic, particularly for hard drives. The Enhanced Cartridge Interface (ECI) is a variant on some later models which combines the cartridge port with a smaller extra port beside it to provide an equivalent to the PBI.

2.2 Clocks

Machine clock (system clock)

The primary clock for the computer is approximately 1.77-1.79MHz, depending on whether the computer is made for the NTSC or PAL video standard. NTSC runs slightly faster at 1.79MHz. The CPU and memory bus run at this speed.

ANTIC preempts the CPU for DMA and memory refresh, stealing some cycles from the 6502 CPU. Thus, the CPU runs at ~60-90% of the maximum speed, depending on the DMA requirements of the current display model.

Color clock

The color clock runs at 3.58MHz, or double the machine clock, and refers to the rate at which color pixels are generated. It runs at the frequency of the NTSC color subcarrier and is also the rate at which the majority of the pixel processing logic in GTIA runs. This is highest rate at which player/missile graphics and color playfield graphics are produced.

On PAL computers, color pixels are still produced at twice machine clock rate, even though the PAL color subcarrier is substantially faster (4.43MHz).

Hi-res graphics produced in ANTIC modes 2, 3, and F -- or GRAPHICS 0 and 8 in the OS -- are generated even faster at double the color clock rate, using both phases of the color clock and thus an effective dot clock rate of 7.16MHz. Only the luminance portion of the playfield runs at this rate.

2.3 Memory system

Memory bus

The computer has a single memory bus, with 16 address bits addressing 64K of RAM, and an 8-bit wide data bus. Most chips are addressible directly through memory mapping, but only the 6502 and ANTIC can drive the bus as bus masters, with the rest of the chips only responding to memory cycles in their address ranges.

The 6502 drives the bus the majority of the time and ANTIC will steal cycles with higher priority as necessary to generate the display or refresh dynamic RAMs. Only the 6502 can issue write cycles, as ANTIC only issues read or refresh cycles.

Memory type

Typically the system memory is composed of 4K x 1, 16K x 1, 64K x 1, or 64K x 4 dynamic RAM chips. Third-party expansions, especially more modern ones, could use 256K chips or even SDRAM or SRAM.

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. $D500-D5FF with no cartridge, $4000-BFFF on the 400 with the standard 16K RAM configuration, and $C000-CFFF on the 400/800.

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.

Floating I/O data bus

On the 800, the system ROMs, PIA, POKEY, and cartridges are connected to a secondary I/O data bus that is split by a pair of data bus transceivers from the main data bus that the CPU, ANTIC, and CTIA/GTIA are connected to. The data buses are connected for some addresses that are not handled by any device, which means that floating data can be read from the I/O data bus separately from the main bus. This is only true on the 800; the 400, XL, XE, and XEGS have all devices on a single data bus.

The following address ranges are decoded for the secondary data bus: $C000-CFFF, $D100-D3FF, $D500-FFFF. The address ranges for CTIA/GTIA ($D000-D0FF) and ANTIC ($D400-D4FF) are excluded and occur over the main data bus only.

Reading an unhandled address on the I/O data bus reads the floating bus data on that bus, which is only affected by accesses to that bus. In particular, this means that reads and writes to main memory are not reflected. Writes to any address on the I/O bus will float data on that bus even if no device responds to the write, and this value can persist until the next read on the I/O bus even if other accesses occur to the main bus in between.

As an example, if PEEK(49152) is executed from Atari BASIC running on a cartridge, the value read will most often be 212 ($D4). This is because BASIC reads the supplied address with an LDA ($D4),Y instruction. The first two cycles of this instruction read the instruction bytes $B1 and $D4 from the I/O bus, the next two cycles read the address 49152 ($C000) from RAM at $D4 and $D5 on the main bus, and the last cycle reads the $D4 value from the floating data from the I/O bus at address $C000. (The value will vary in practice because ANTIC may halt the CPU temporarily and read character data from ROM at $E000-E3FF during the instruction.) On the other hand, if BASIC is loaded into RAM, the value will tend to reflect character data because the instruction fetches will no longer occur on the I/O bus.

Memory refresh

Because dynamic RAM requires periodic refresh to maintain contents, ANTIC does up to nine refresh cycles per scanline to refresh memory. The number of refresh cycles varies depending on playfield DMA requirements since playfield DMA has priority over memory refresh, while the full nine cycles are issued during vertical blank. On each refresh cycle, one row of memory within each memory chip is refreshed. An internal counter within ANTIC increments the refreshed row address with each refresh cycle. Depending on the model of ANTIC chip, this counter is either 7 or 8 bits wide, refreshing 128 or 256 rows (the latter being necessary for some later used memory chips).

While ANTIC is responsible for meeting DRAM refresh requirements solely by itself, any non-refresh access to memory, either by ANTIC or the CPU, will also refresh the accessed memory row. The lowest address bits determine the row. For the 130XE, both main and extended memory banks are refreshed together on any access, though only one may output to the data bus.

On a refresh cycle, the normal data output from the memory chips is suppressed either by disabling the /CAS signal (XL/XE) or by turning off buffers between the memory and the data bus (400/800). The memory decoding logic or MMU also suppresses any I/O or ROM mappings that would otherwise respond to the refresh address supplied by ANTIC. As a result, no device will drive the data bus and it will either float or be pulled up.[1]

2.4 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.

[1] It is possible to observe this by overlapping playfield and refresh DMA cycles. This is done by disabling playfield DMA via DMACTL mid-scanline and pulling the data bus contents during refresh cycles into the line buffer.

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.

2.5 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 registers DDRA/DDRB and input/output registers ORA/ORB share addresses. In order to read or write the data direction register, bit 2 of the port's control register must be set to 0, and to read or write the I/O register, 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.[2] 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 for a port resets both interrupt bits for that port. This must be the input register; reading the data direction register has no effect on interrupt status. This has implications for PIA interrupt handlers, which must either require that ORA/ORB be active when PIA IRQs are active and unmasked or temporarily switch from DDRA/DDRB to ORA/ORB to acknowledge the interrupt.

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

[2] [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.

of 100-500 ms. Delays vary between individual bits, between systems and can even vary widely on the same 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.

2.6 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.

ROM control

While the 400/800 use PIA port B to interface with joystick ports 3 and 4, the XL/XE computers only have two joystick ports. The otherwise unused port B is instead used to enable and disable the system ROMs.

Bit 0 controls the OS ROM at $C000-CFFF and $D800-FFFF. A '1' bit enables the OS ROM.

Bit 1 controls the BASIC ROM at $A000-BFFF (except on the 1200XL, which has no built-in BASIC). A '0' bit enables the BASIC ROM. Note that this is inverted from the OS ROM bit (bit 0).

Bit 7 controls the self-test ROM at $5000-57FF. A '0' bit enables the self-test ROM, if the OS ROM is also enabled. If the OS ROM is disabled, the self-test ROM is also disabled regardless of the state of bit 7.

Pull-ups ensure that port B bits 0 and 7, and also bit 1 on non-1200XL machines, are held high if those bits are switched to input mode on the PIA. Since the PIA switches all port bits to inputs on reset, this guarantees that the OS ROM is enabled and the BASIC and self-test ROMs are disabled on system reset.

Clearing bit 0 and setting bits 1 and 7 disables all system ROMs, enabling access to 62K of RAM. The 2K block of hardware registers at $D000-D7FF cannot be disabled.

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 range 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.

2.7 Extended memory

Starting with the XL models, a common way of expanding memory above the main 64K was to add additional expanded memory with bank switching. This involves mapping an extended memory window as an overlay over the $4000-7FFF region, allowing access to the extended memory 16K at a time. PIA port B is used to control the extended memory window.

Window control

In most expansions, bit 4 of PORTB enables or disables the extended memory window. Bit 4 is inverted, so a 1 disables the window while 0 enables it. When the window is enabled, all reads and writes to the $4000-7FFF region are directed to extended memory, and the main memory hidden underneath is untouched.

For a 128K system with 64K of extended memory, bits 2 and 3 select unique 16K extended memory banks. The bits are ignored if the extended memory window is disabled, though the value of those bits is still kept. Larger expansions use more bits in PORTB to select additional banks.

Note that the outputs from PIA port B control the extended memory window, so bits in PORTB are only effective if configured as outputs. The XL/XE OS configures port B as all outputs by default, so normally all bits function. If some bits are configured as inputs, however, they will not control extended memory functions. Typically there are pull-ups on all port B bits used for memory mapping, so any bits configured as inputs will function the same as a 1 bit in output mode.

Separate ANTIC access

Some expansions have the ability to enable the extended memory window independently for ANTIC and CPU access. For these, bit 4 enables the window for CPU reads and writes, while bit 5 enables it for ANTIC reads. This makes it possible to display from extended memory while reading and writing from main memory at $4000-7FFF and vice versa. There is still only one set of bank selection bits, however, so when both are enabled for extended memory access the same bank must be used for both.

For expansions that do not support separate ANTIC access, the window applies to both ANTIC and the CPU: either both access main memory or both access extended memory, controlled by the single enable bit.

Expansions labeled as COMPY typically support ANTIC access, while ones labeled as RAMBO do not, both names coming from model expansions with those behaviors. The 130XE, the only stock computer with extended memory support, also supports separate ANTIC access.

Self-test ROM conflict

The self-test ROM can conflict with the extended memory window since it occupies $5000-57FF. When both the self-test ROM and extended memory are enabled, the self-test ROM has priority in this region, with reads coming from the self-test ROM and writes being discarded without modifying memory. This situation is not possible with some expansions that disconnect the self-test ROM or reuse its bit while the extended memory window is enabled.

Describing extended memory schemes

Extended memory schemes are often described in terms of the typical PORTB bytes that can be used to access unique banks. Most expansions preserve the function of bits 0 and 1 and use bits 2 and 3 for bank selection, so the expansion can be described by the valid values for the high four bits. For instance, an expansion using bits 2-3 and 5-6 for bank selection would have unique banks for $8x, $Ax, $Cx, and $Ex, or 8ACE for short. Similarly, an expansion using bits 2-3 and 6-7 would use $2x, $6x, $Ax, and $Ex, making its banking pattern 26AE.

Bit reuse

Some particularly large expansions reuse PORTB bits already assigned to mapping functions in the XL/XE computers. Typically bit 0 (OS ROM) is kept, while bit 1 (BASIC) and bit 7 (self test) may be reused as banking bits. Depending on the expansion, these bits may either be reassigned entirely, losing their original function, or may only function as banking bits when extended memory is enabled. For instance, bit 7 may control the self test ROM when expanded memory is disabled, but control a bank selection bit when the window is enabled, with the self-test ROM forced off in that case.

Main memory aliasing

Some extended memory expansions may alias 64K of extended memory against main memory due to reusing the same memory addressing. The result is that four of the extended memory banks address main memory such that reading or writing the two address windows are equivalent. This is documented behavior for the ICD RAMBO XL product, which aliases banks 0-3 of its 256K extended memory space against main memory.[3]

Expansion list

Table 1 lists some extended memory configurations. This list is not exhaustive; there are many other extended memory configurations in use on actual hardware.

Type	Configuration	Banking bits	Bank blocks	Notes
130XE	64K + 64K	2, 3	E	Separate ANTIC access
192K (RAMBO)	64K + 128K	2, 3, 6	AE	
256K (RAMBO)	256K	2, 3, 5, 6	8ACE	$8x banks alias main memory
320K (RAMBO)	64K + 256K	2, 3, 5, 6	8ACE	
320K (COMPY)	64K + 256K	2, 3, 6, 7	26AE	Separate ANTIC access
576K (RAMBO)	64K + 512K	1, 2, 3, 5, 6	8ACE	
576K (COMPY)	64K + 512K	1, 2, 3, 6, 7	26AE	Separate ANTIC access
1088K (RAMBO)	64K + 1024K	1, 2, 3, 5, 6, 7	02468ACE	

Table 1: Some extended memory configurations

[3] [RamboXL] p. 14

2.8 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.[4]

2.9 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.

WarGames

This game has a unique check to verify that $C000-CFFF is not populated with either RAM or ROM on an 800 system. If the routines in this region do not match the checksum for the 1200XL or XL/XE ver. 2 OS, the game

[4] [ATAXL] p.15

writes a byte to a single location in this range and then reads a series of addresses, checking whether the data read from any of those addresses changes. Since the check routine is running from RAM, this relies on the write being floated on the I/O data bus without being disturbed by the instruction fetches. The check will pass if $C000-CFFF either contains RAM or returns floating I/O bus data, but will fail if it is ROM or main floating bus data.

Atari Operating System Rev. A/B

The RAM sizing test for OS-A/B tests for RAM by twice complementing the byte at the beginning of each 4K of memory starting at $1000 and checking that the value read back matches each time. This test normally stops at $C000 due to the floating I/O bus, relying on the instruction fetches from ROM to immediately overwrite the written value on the I/O data bus. This test will also stop at $C000-CFFF if the range returns a constant value due to either ROM or a pulled-up data bus. If the system is reconfigured so that the memory sizing code runs from the main data bus or that $C000-CFFF returns floating data from a different bus, the sizing code can incorrectly determine that range to be RAM.

2.10 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.

[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 3
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.

3.1 Flags

Flags are stored in the P register of the CPU and both capture and store persistent state across multiple instructions. Most instructions on the 6502 implcitly set at least one flag during execution of their main operation, commonly the N and Z flags. Some of the flags also modify global CPU behavior, such as the interrupt mask (I) flag.

Address calculations never use or modify the flags. Thus, the addition in the absolute indexed addressing mode abs,X neither uses the carry flag as input nor modifies it based on the resulting address, nor is it subject to modification from the decimal (D) flag.

Carry (C) flag (bit 0)

The carry flag extends many arithmetic and logical operations by one bit to allow processing of 16-bit or larger quantities in multiple 8-bit operations. For arithmetic operations, it supplies the carry or borrow input into bit 0 and receives the carry or borrow output out of bit 7, and similarly supplies inputs and captures outputs for shifts and rotates.

The 6502 notably differs from some other CPUs in the polarity of the C flag for subtract operations. On the 6502, C=1 indicates no borrow in/out for a subtract operation, and C=0 indicates a borrow. Thus, SBC is commonly preceded by SEC when a pure subtraction with no borrow in is desired. This interpretion is consistent with implementing subtraction by adding the one's complement (all bits inverted or XOR'd with $FF).

Zero (Z) flag (bit 1)

The zero flag is set when the result of an operation is zero or all bits cleared, and clear otherwise. It is purely a result bit.

Interrupt mask (I) flag (bit 2)

The I flag determines whether maskable interrupts are blocked or masked. If it is cleared, then IRQs are handled normally; if it is set, IRQs are "masked" and ignored by the CPU. NMIs are not affected by the I flag, as they are non-maskable.

The I flag is automatically set on reset to prevent the CPU from receiving stray IRQs until it has completed hardware and software initialization.

Decimal (D) flag (bit 3)

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, DEC, and indexed addressing 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.

Break (B) flag (bit 4)

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 set for a BRK and cleared for an IRQ/NMI. In rare circumstances, it is possible for an NMI to piggyback on a BRK and the NMI vector can also be invoked with bit 4 set on the flags on the stack.

On the 65C816, bit 4 is reused as the index size (X) bit in native mode.

Unused flag (bit 5)

The 6502 does not use bit 5 of the P register. It can't be cleared and always reads as a 1 when pushed to the stack with the PHP instruction or by interrupt entry.

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

Overflow (V) flag (bit 6)

The V flag is set during arithmetic operations to indicate if a *signed* overflow has occurred, where the result is outside of the -128 to +127 range of a byte and has been truncated. It is cleared otherwise. This is sometimes implemented as the XOR of the carries out of bit 6 and bit 7. It is uncommonly used, being changed only by add, subtract, compare, and flags-specific operations. It also captures bit 6 of the source for a BIT instruction.

There is another uncommon use of the V flag, to capture an event signaled on the Set Overflow (SO) input on the CPU. Asserting SO results in the V flag being set asynchronously to regular code execution. This is not used on the main computer, but some peripherals using 6502-family CPUs do use this facility.

Negative (N) flag (bit 7)

The N flag is usually set when the result of an operation is negative, i.e. the sign bit in bit 7 is set. More generally, it is usually copy of bit 7 of the result. There are a couple of exceptions, such as BIT setting the N flag to bit 7 of the source data rather than of the bitwise AND operation.

3.2 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.[5]

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.

[5] [IJO10]

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.

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.[6]

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.

3.3 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)

[6] [6502Dec]

- 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 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 or 65C816 in native mode).

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.

Note

A branch crosses a page boundary when the addition of the signed branch offset changes the high byte of the PC. This means that a page crossing occurs if the target is on a different page from the address of the *next* instruction, not from the address of the branch instruction. For instance, a BCC $80C0 instruction at $80FE crosses a page because it is branching from $8100 to $80C0, even though the branch instruction itself is entirely within the same page as its target. Similarly, a BEQ $8110 instruction at $80FE does *not* cross a page. This happens because the branch offset is added after the PC has already been incremented for both bytes of the branch instruction.

3.4 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. This effect occurs with CLI+SEI and CLI+PLP pairs; it does not occur with CLI+RTI, PLP+[SEI/RTI/PLP], or RTI+[SEI/RTI/PLP], for which no IRQ is dispatched even if one is pending.

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.[7]

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.

[7] 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, which usually doesn't happen.

105	106	107	108	109	110	111	112	113	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
IRQ							LSR abs						PHA			LDA #im		STA abs				NMI							PHA			
	IRQ							LSR abs						PHA			LDA #im		NMI							PHA			LDA #im			
		IRQ							LSR abs						PHA			LDA #im		NMI							PHA			LDA #im		
			IRQ							LSR abs						PHA			NMI							PHA			LDA #im			
				IRQ							LSR abs						PHA			NMI							PHA			LDA #im		
					IRQ							LSR abs						PHA			NMI							PHA			LDA #im	
						IRQ							LSR abs						NMI							PHA			LDA #im			
							IRQ							LSR abs						NMI							PHA			LDA #im		
								IRQ							LSR abs						NMI							PHA			LDA #im	
									IRQ							LSR abs						NMI							PHA			
										IRQ							LSR abs						NMI							PHA		
											IRQ							LSR abs						NMI								
												IRQ							LSR abs						NMI							
													IRQ							LSR abs						PHA			LDA #im			
														NMI							PHA			LDA #im		STA				LDA #im		
															NMI							PHA			LDA #im		STA				LDA #im	

Figure 2: 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.

3.5 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 2 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	SHA abs,Y
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,Y	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 2: 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.[8]

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.

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

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

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.[9]

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.

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.

3.6 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 3 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.

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

	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 3: 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.

3.7 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 4.

	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 4: 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.

3.8 65C816 new features

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. Changing the X flag back to 0 does not restore the previous contents of the high bytes, which will remain $00. 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, and TAY 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).

Switching to native mode

The only way to enter native mode is with the XCE instruction, which exchanges the carry and emulation state flags. Executing XCE with C=0 enters native mode and sets C=1 if the CPU was previously in emulation mode. Entering native mode switches the CPU to the alternate native mode set of interrupts vectors and unlocks the M/X bits in the P register.

Executing XCE with C=1 exits native mode and switches back to emulation mode. When this happens, the emulation interrupt vectors become active, M/X bits are set to 1, the upper bytes of the X and Y registers are lost and reset to $00, and the high byte of the stack pointer is set to $01. The values of the D, DBK (B), and PBK (K) registers are unaffected, however.

Many new features of the 65C816 do not require native mode to use. New instructions, new addressing modes, 24-bit addressing, and 16-bit operations that do not depend on clearing M/X bits can be used directly from emulation mode. However, the ability to execute code in banks other than bank 0 is of limited use as interrupts do not save the program bank on the stack in emulation mode, making it impossible to return to the interrupted routine.

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. This extra cycle occurs regardless of whether a page crossing occurs.

There are a couple of exceptions to direct page wrapping in emulation mode. Instructions that read words from direct page and are new to the 65C816 will cross pages regardless of the low byte of D. This includes PEI (dp) and instructions using the [dp] and [dp],Y addressing modes, which will cross over from $00FF to $0100 and $0101 with D=0. The (dp) and (dp),Y addressing modes will wrap in this case, with ($FF) and ($FF),Y reading the base address from $FF and $00.

The (dp,X) addressing mode has mixed behavior in emulation mode. With the low byte of D set to $00, (dp,X) has the 6502/65C02 compatible behavior of wrapping within the page. However, when the low byte of D is not $00, the address computation for the low byte will cross pages and then the high byte will wrap. For instance, ($FF,X) with X=$FF and D=1 will read the low byte from $01FF and the high byte from $0100.

In native mode, all direct page accesses cross page boundaries with any instruction regardless of the value of D. Indexing will cross pages freely, and 16-bit accesses starting at $xxFF will continue to $yy00 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 regardless of whether a page crossing occurs.

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.[10]

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.

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

3.9 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.

[10] See also [ObWrap].

3.10 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 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:

- NMIEN
- DMACTL
- Playfield DMA clock

The following items are **not** reset:

- Refresh row address counter
- Horizontal and vertical counters[11]
- WSYNC
- HSCROL/VSCROL
- PMBASE
- CHBASE
- PENH/PENV
- CHACTL
- DLISTL/H
- NMIST
- Memory scan counter
- Pending RNMI

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.

Typical power-up values

Any registers that are not internally cleared by ANTIC on reset have undefined contents on power-up. However, the internal architecture biases some registers toward specific values:

[11] There is evidence in the chip circuitry of a power-on detector that is supposed to reset the horizontal and vertical counters only on power-up, but this does not seem to work in practice as the machine powers up with varying values of VCOUNT.

Register	Typical power-up value
PENH	$00
PENV	$FF
NMIST	$7F

Table 5: Typical power-up values for ANTIC registers

These values are most likely to appear when the system is powered up cold. If it has been powered down recently – within a few seconds – these registers may instead show some partial bits from when the system was last turned off.

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	1.789773MHz (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 6: 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.[12] 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.

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.[13]

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.

[12] [TIVideoDec] p.2-7.

[13] 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.

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:

- **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 2-5, the character set is stored as 128 characters within a 1K block, aligned to a 1K boundary; for modes 6 and 7, 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

The DLISTL and DLISTH registers contain the instruction pointer used to fetch the display list. At the end of each mode line, ANTIC fetches a new instruction at the location pointed to by DLISTL/DLISTH into the instruction register (IR), and then increments the pointer. This continues until a jump instruction is reached, which then loads a new address into DLISTL/DLISTH. ANTIC does not store the start of the display list and has no registers to do so; the display list must either loop or be restarted by the CPU.

The display list can reside anywhere in the 64K address space, but it cannot cross a 1K boundary. This is because the DLISTL/DLISTH register is actually split into 6 bit and 10 bit portions, where the lower 10 bits increment and the upper 6 bits do not.[14] As a result, during normal execution the display list will wrap from the top of a 1K block to the bottom during fetching, e.g. $07FF to $0400. This will happen even in the middle of a three-byte LMS or jump instruction. Jump instructions are not limited and can cross 1K boundaries to any address.

DLISTL/DLISTH are live during display list execution and any write to either will immediately change the address 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:

[14] Hardware II.10

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.

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. As a result, a blank mode line is always considered to have the horizontal/vertical scroll and LMS bits cleared. However, it can trigger a DLI, and is also subject to height modification if at the end of a vertical scrolling region.

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.

DLIs can be triggered on jump commands.

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. If this occurs when a DLI is enabled on that mode line, the DLI is skipped since the last scan line never occurs.

Suspended display list DMA

DMACTL bit 5 controls display list DMA, but the display list itself is actually always enabled. When DMA is disabled, the display list instead repeats its previous instruction byte. Any Jump or Load Memory Scan (LMS) commands are disabled as the address fetch is also skipped, and the display list pointer does not increment. If the display list was stopped after a JVB instruction ($41), this produces blank lines and the display list is effectively stopped. However, any other instruction byte activates a mode line as usual, including multi-row blank lines, character and bitmap mode lines, and even activating DLIs as usual.

Turning off display list DMA has no effect after a jump and wait for vertical blank ($41) instruction executes, as no fetches occur anyway once JVB completes.

While bits 0-5 and bit 7 of the instruction register are preserved across vertical blank, bit 6 of the IR 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

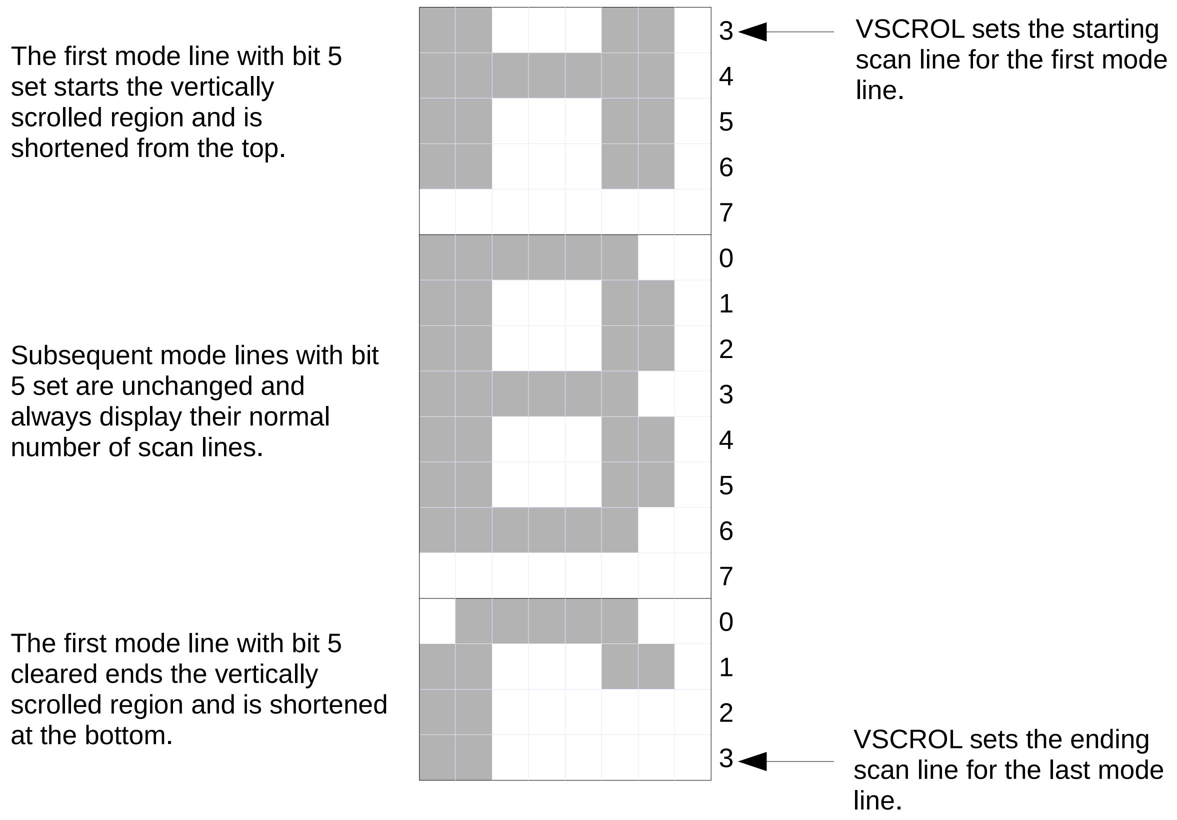

The first mode line with bit 5 set starts the vertically scrolled region and is shortened from the top.
Subsequent mode lines with bit 5 set are unchanged and always display their normal number of scan lines.
The first mode line with bit 5 cleared ends the vertically scrolled region and is shortened at the bottom.
VSCROL sets the starting scan line for the first mode line.
VSCROL sets the ending scan line for the last mode line.
3
4
5
6
7
0
1
2
3
4
5
6
7
0
1
2
3

Figure 3: Effect of vertical scrolling on mode lines

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.

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 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.

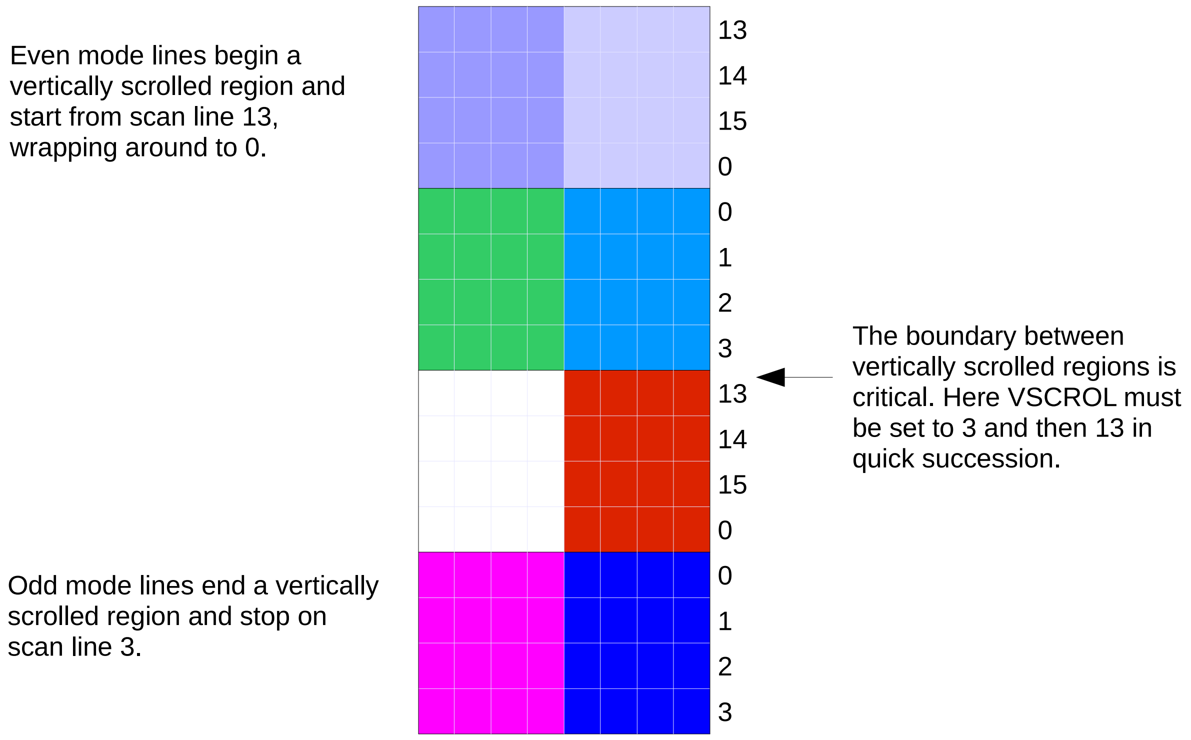

Even mode lines begin a vertically scrolled region and start from scan line 13, wrapping around to 0.
13
14
15
0
0
1
2
3
13
14
15
0
0
1
2
3
The boundary between vertically scrolled regions is critical. Here VSCROL must be set to 3 and then 13 in quick succession.
Odd mode lines end a vertically scrolled region and stop on scan line 3.

Figure 4: Abusing vertical scrolling in the "GTIA 9++" mode

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 or the starting row, the data fetch still always only occurs on the first scan line.

It is possible to extend mode lines beyond even 16 scan lines by changing VSCROL in the middle of the mode line. Since the delta counter (row counter) is only 4 bits wide, rows 0-15 are repeated until the mode line ends.

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 disabled interrupts.

The reset (RNMI) 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 happens regardless of whether either interrupt is masked in NMIEN. 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. It also means that it is possible to use DLIs passively by polling for them instead of using an interrupt handler.

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[15]:

- 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[16].
- 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,

[15] De Re Atari also has a good description of DLI timing and explains how to break a DLI routine into phases by timing requirements.
[16] [MOS76] 38

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 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.[17] 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.

System Reset NMI

ANTIC supports a third type of NMI, triggered by the System Reset button on 400/800 models. On these models, the System Reset button asserts the /RNMI input on ANTIC, and once this is held across the leading edge of VBLANK, the Reset NMI is triggered.

The Reset NMI is synchronized to the VBI, so it will always happen at the start of VBLANK and both the VBI and RNMI will trigger together. Both bits 5 and 6 of NMIST will also turn on, so the NMI handler must give the RNMI priority over VBI processing and trigger a warm start when bit 5 is set. All systems that use the RNMI also have a fixed OS and NMI vector, so ordinarily this is always handled by the OS before user code can see it.

As the Reset NMI cannot be masked, the OS cannot defend against it occurring before cold start initialization has completed. User manuals therefore politely ask the user to not hold System Reset on power-up to avoid premature warm start.

On the XL/XE series, the Reset NMI is unused and not connected; the System Reset button asserts the reset line across the system instead. However, it is still possible for the RNMI status bit to be set in NMIST as it is not cleared on power-up or reset. Once it has been cleared, it will no longer become set again, even if the Reset button is pressed.

[17] 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.

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 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.[18] The 6502 does not respond to RDY during a write cycle and therefore always performs this write on the next

[18] 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.

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 which affects this behavior. The 65C02 will stop on a write cycle, , 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 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 7 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 7: 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 bitwise AND of all fetch addresses 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 always be able to 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).[19]

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

[19] 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.

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.

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.[20]

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

[20] [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.

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 instruction 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 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 28, 20, or 12 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 starting at clocks 26, 18, and 10 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).

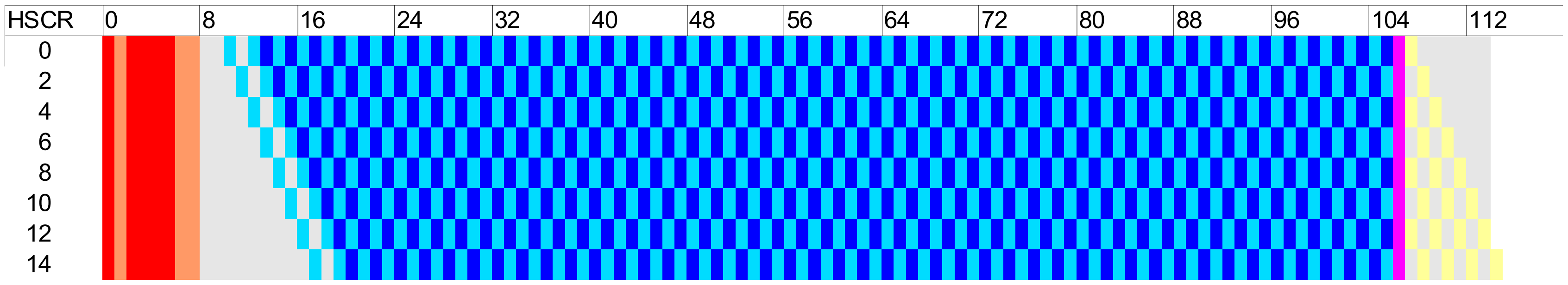
Virtual DMA cycles

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.

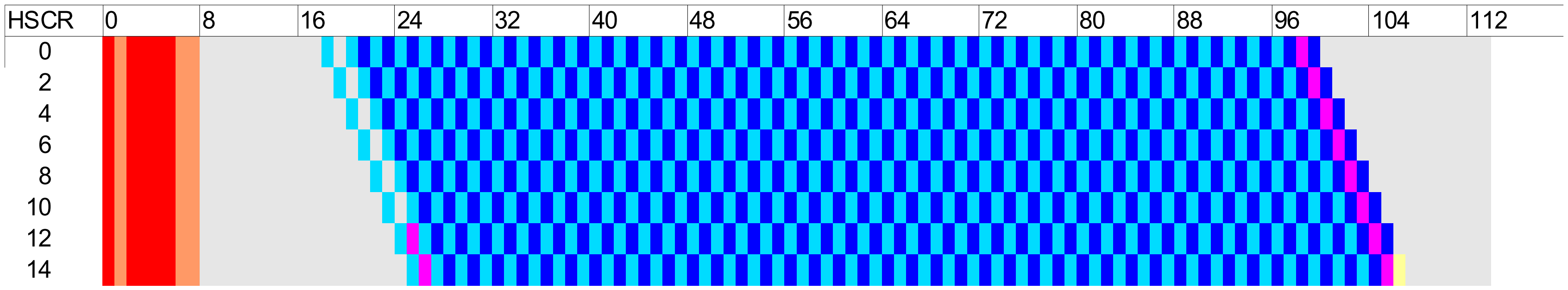
DMA timing charts

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

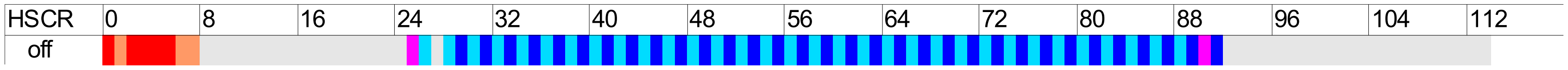


ANTIC modes 2-5, mode line, normal playfield



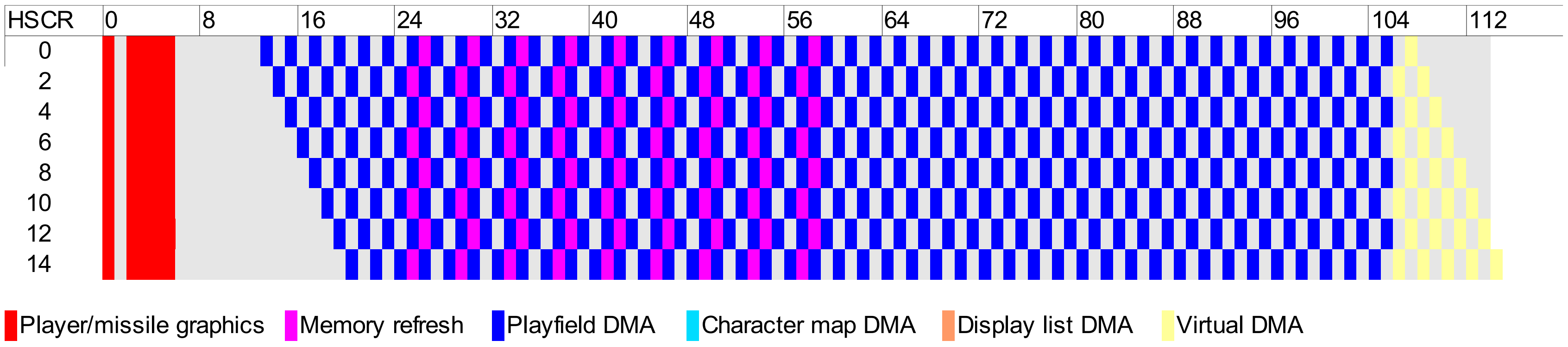
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC mode 2-5, mode line, narrow playfield

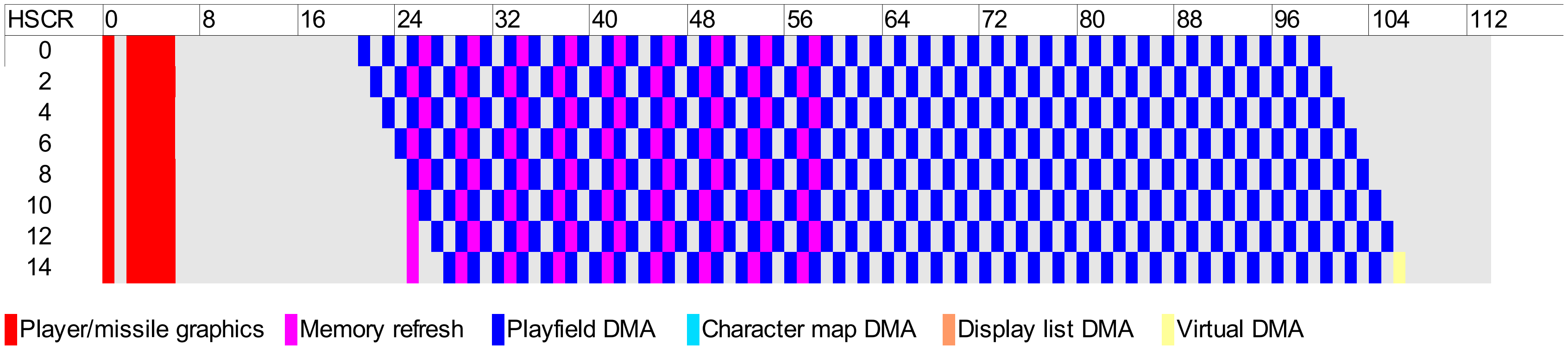


🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

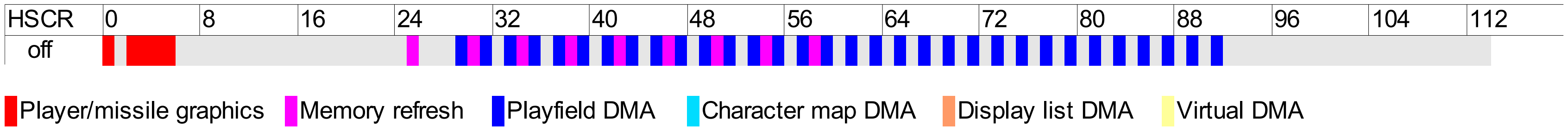
ANTIC modes 2-5, subsequent lines, wide playfield



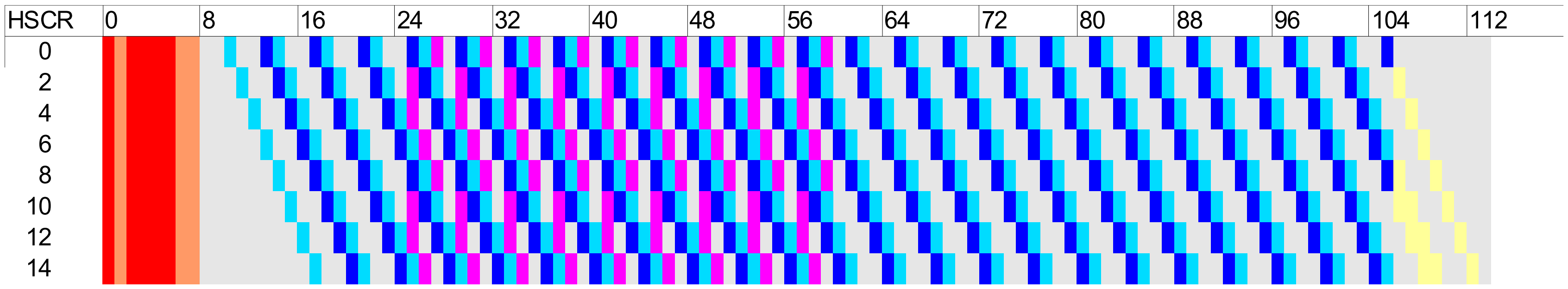
ANTIC modes 2-5, subsequent lines, normal playfield



ANTIC modes 2-5, subsequent lines, narrow playfield

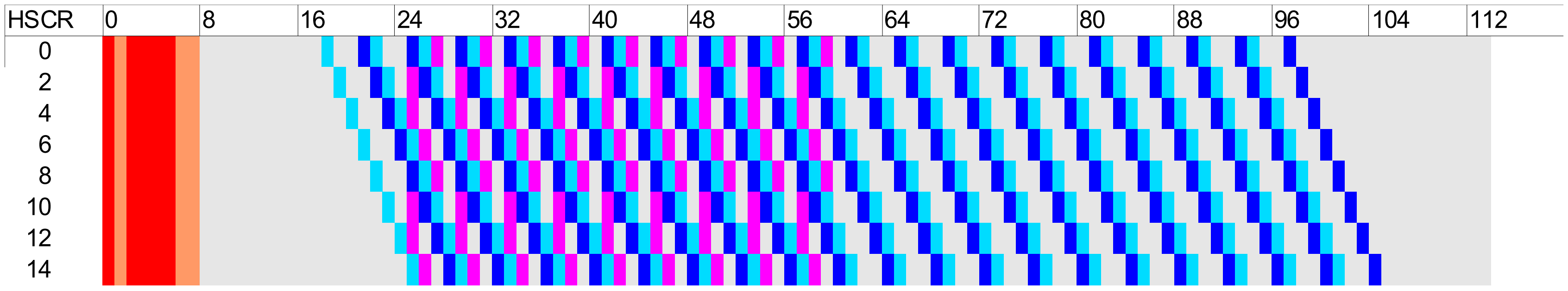


ANTIC modes 6 and 7, mode line, wide playfield



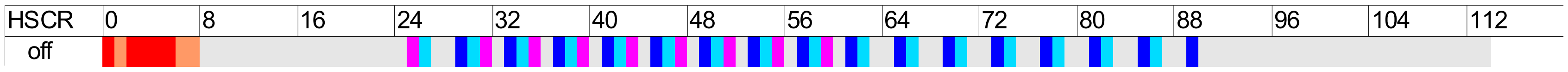
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 6 and 7, mode line, normal playfield



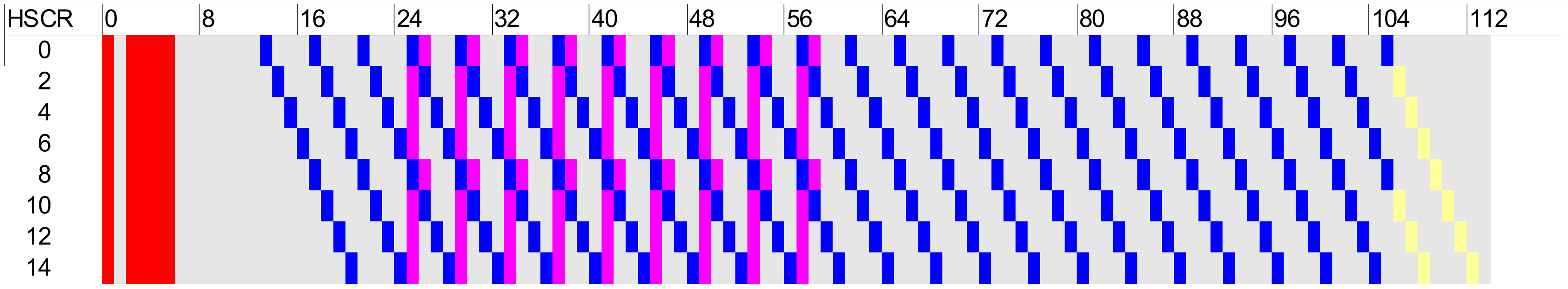
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 6 and 7, mode line, narrow playfield



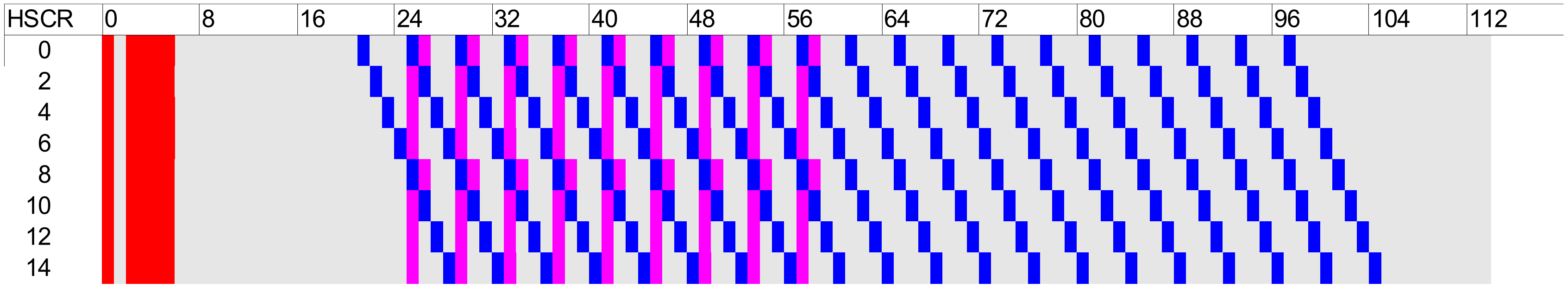
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 6 and 7, subsequent lines, wide playfield



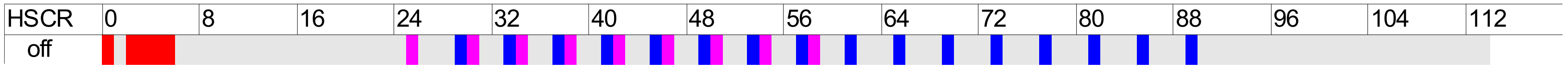
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 6 and 7, subsequent lines, normal playfield



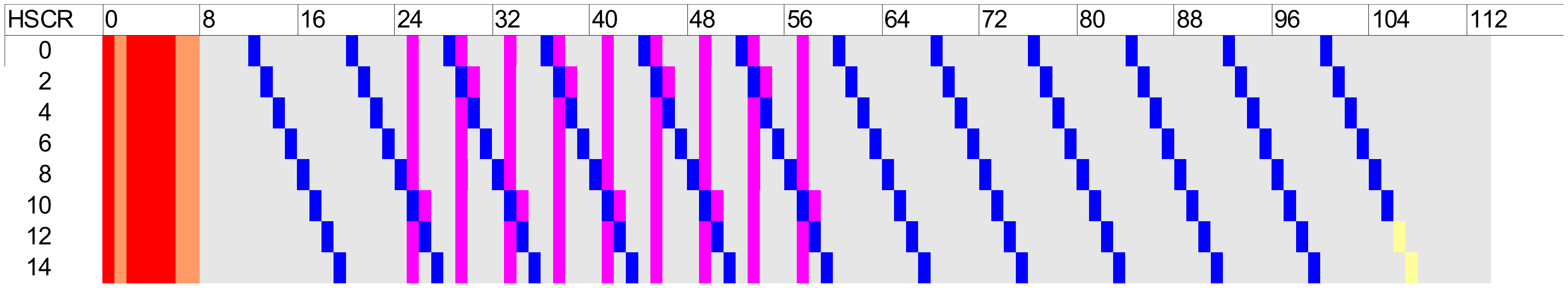
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 6 and 7, subsequent lines, narrow playfield



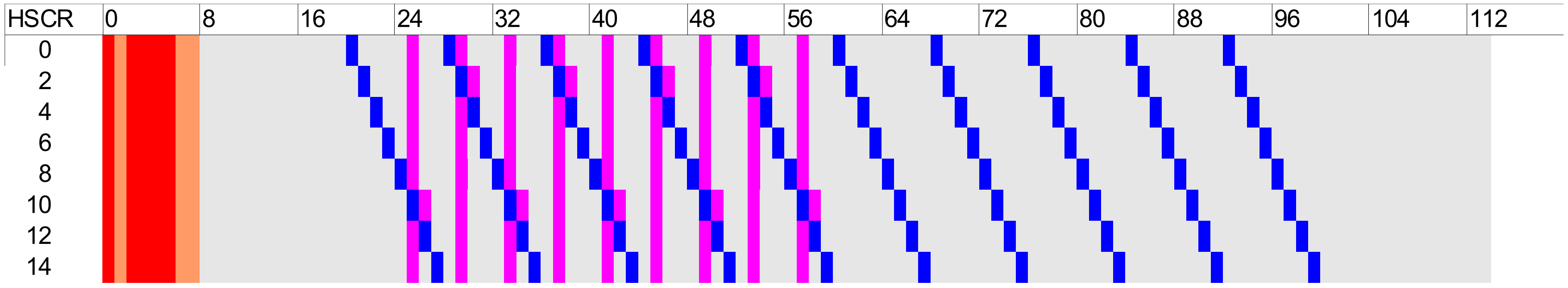
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 8 and 9, mode line, wide playfield



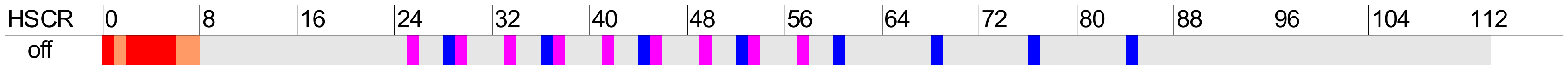
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 8 and 9, mode line, normal playfield



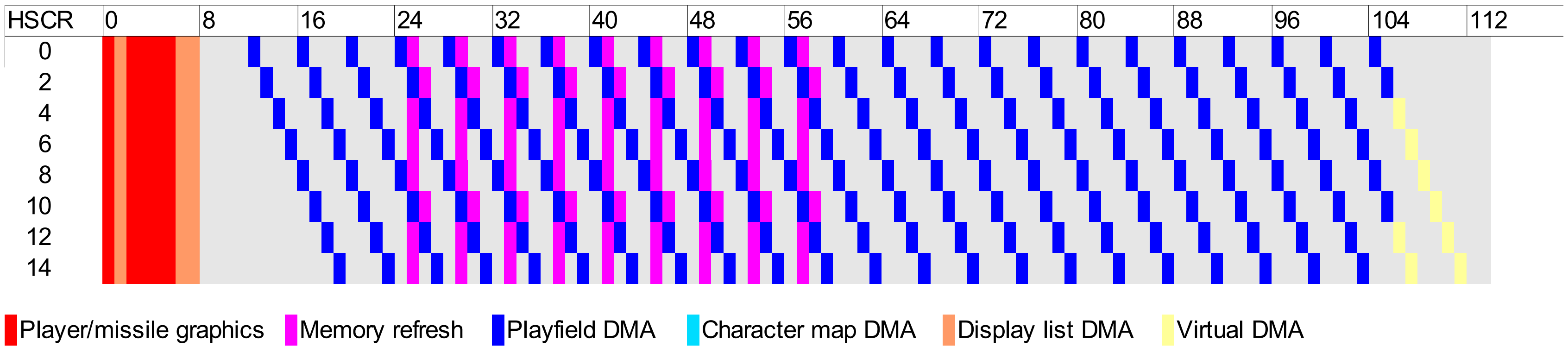
🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

ANTIC modes 8 and 9, mode line, narrow playfield

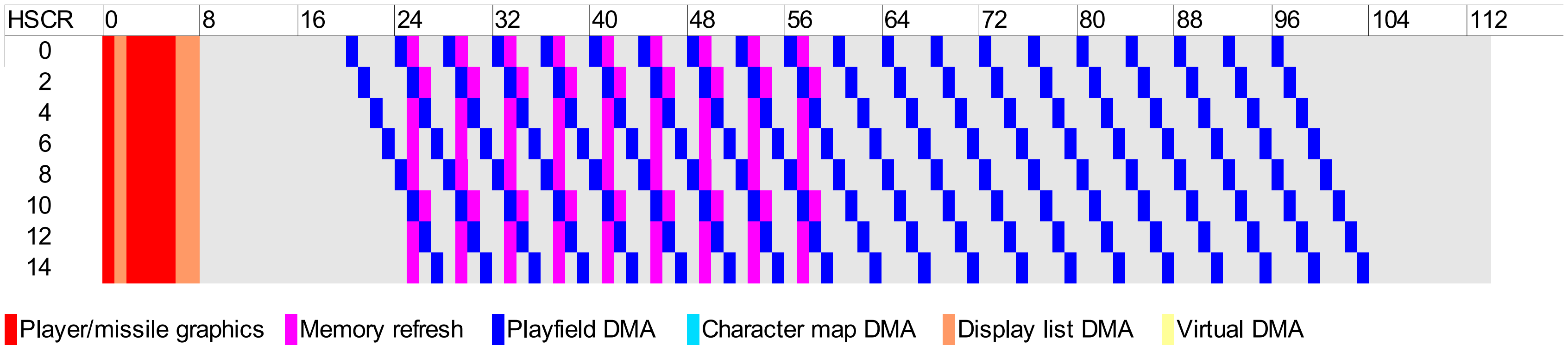


🟥 Player/missile graphics 🟪 Memory refresh 🟦 Playfield DMA 🟦 Character map DMA 🟧 Display list DMA 🟨 Virtual DMA

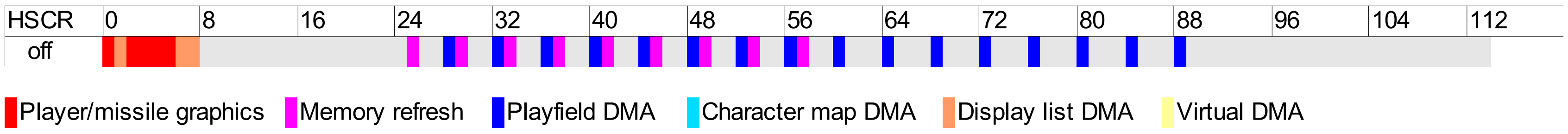
ANTIC modes A-C, mode line, wide playfield



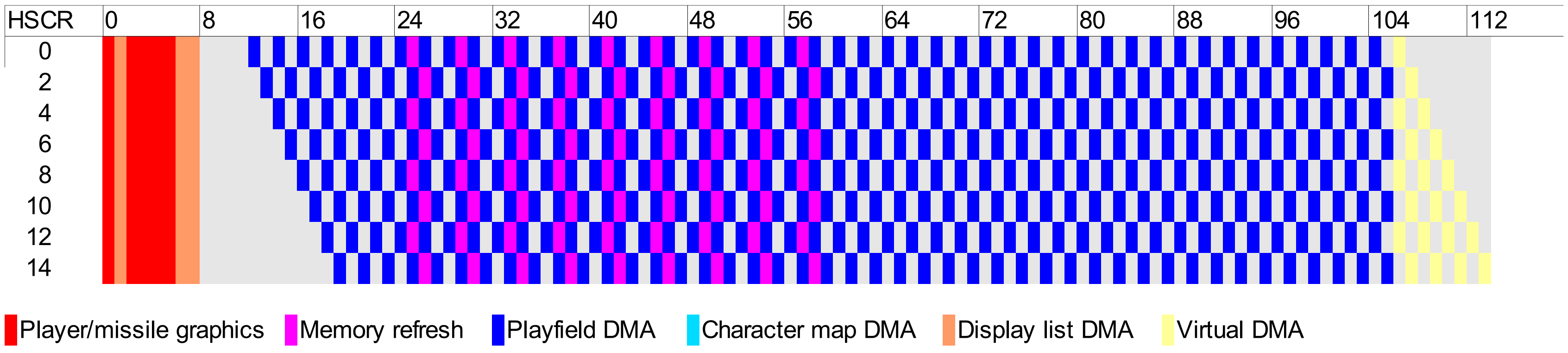
ANTIC modes A-C, mode line, normal playfield



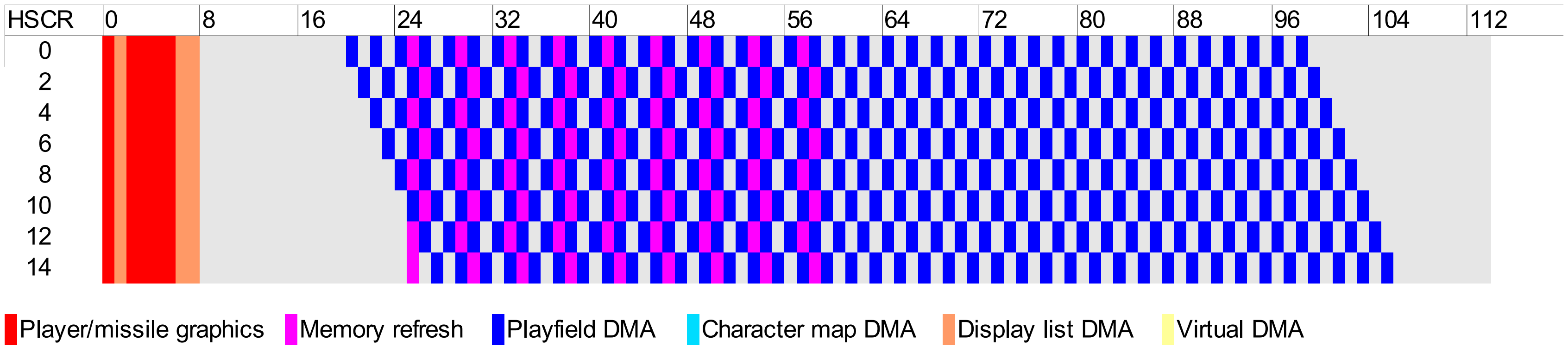
ANTIC modes A-C, mode line, narrow playfield



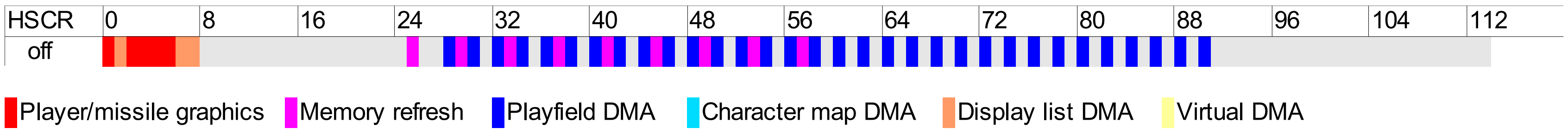
ANTIC modes D-F, mode line, wide playfield



ANTIC modes D-F, mode line, normal playfield



ANTIC modes D-F, mode line, narrow playfield



Event timing chart

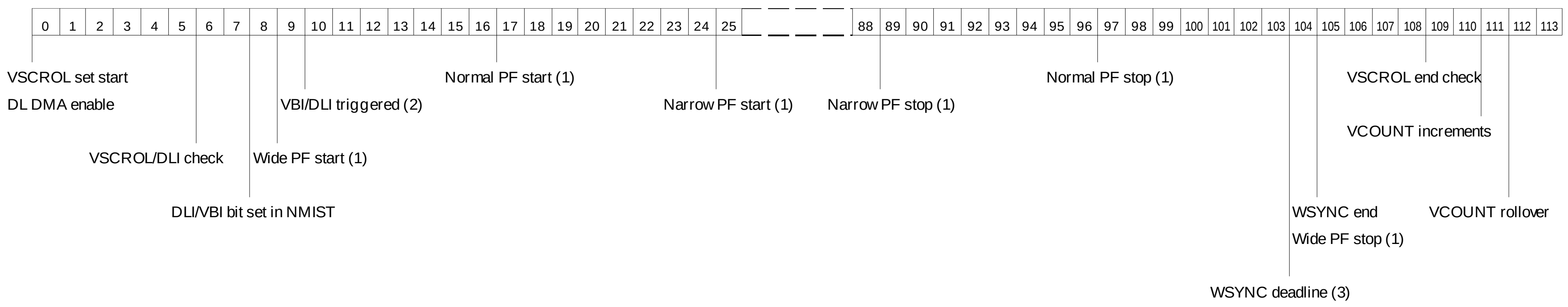


Figure 5: ANTIC event timing

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
```

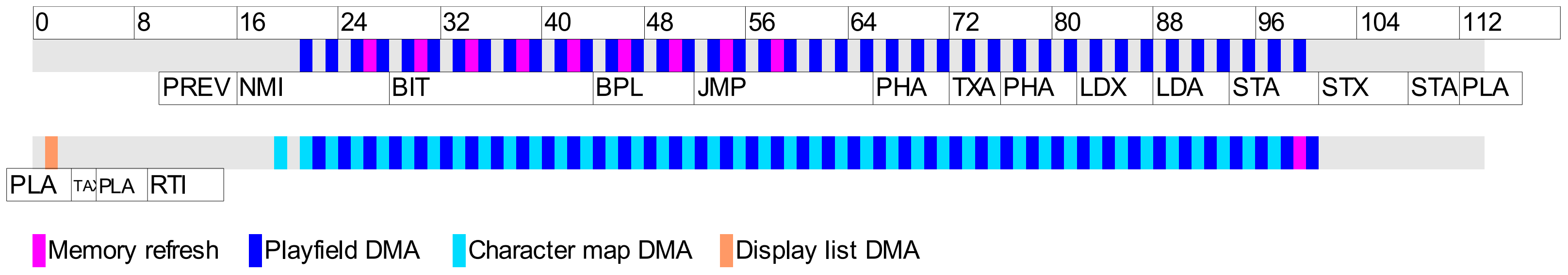


Figure 6: DMA and CPU timing for DLI handler.

Cycle counting breakdown

Figure 6 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.

4.16 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.

Atari OS C: handler

The cassette (C:) handler in the Atari OS has a bug where it can rarely compute bogus baud rates due to improper reading of VCOUNT. The OS tries to read both VCOUNT and a frame counter maintained by the VBI and assumes that scan line 248 or higher (VCOUNT ≥ 124) always occurs after the VBI, but this is not the case. ANTIC triggers the VBI about a dozen cycles after VCOUNT increments to 124, so it is possible for VCOUNT=124 to occur both before and after the VBI. The former causes an erroneous baud rate to be computed by the OS.

4.17 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.

Unassigned addresses

Reading locations assigned to POKEY but that don't correspond to a readable internal register return $FF, even on machines with a floating data bus.

Stereo modification

A popular unofficial 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

Power-up state

POKEY does not have a RESET line and therefore powers up in indeterminate state. Although various circuits within the chip are biased toward particular states from a cold start, the initial state is not guaranteed and may include interrupts being enabled in IRQEN and asserted in IRQST. Thus, IRQEN must be reset prior to clearing the I bit on the CPU to avoid stray IRQs.

Initialization mode

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 resets various clocks and state machines throughout the chip. The following logic circuits are reset:

- 15KHz clock
- 64KHz clock
- 4-bit and 5-bit polynomial noise generators
- 9/17-bit polynomial noise generator (which includes RANDOM)
- Serial port input state machine and shift register
- Serial port output state machine
- The SEROUT valid flip-flop
- Keyboard scan (due to clearing bit 1)

These circuits are held in reset state until initialization mode is exited. For instance, while SKCTL bits 0 and 1 are both cleared, any sound/timer channels using the 15KHz or 64KHz clocks will not count, as those clocks will be frozen, and RANDOM will lock at $FF.

Initialization mode does **not** reset:

- Interrupt enable (IRQEN) or status (IRQST)
- KBCODE
- SERIN
- AUDF1-4, AUDC1-4, or AUDCTL
- Timer counters
- Audio channel outputs
- Pot scan (although the pot scan will be suspended until init mode is ended, due to the frozen 15KHz clock)
- The output of the serial output shifter, including the bit sent out the SIO DATA OUT line and driving two-tone mode

Serial port reset

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.

Clock reset timing

The initialization function can be used to reset the 15KHz and 64KHz clocks to known offsets in the scan line. As long as init mode has been enabled long enough for both clocks to reset fully, the clock offsets will be determined by when init mode is exited. However, the design of the clocks causes the clocks to be reset to partway through their cycles instead of the beginning.

Both clocks are polynomial counters with truncated cycles. The 15KHz clock is a 7-bit XNOR counter with a polynomial of x^7+x^6+1. On a pattern of %1001001 (shifting left), a '1' bit is forced and a pulse emitted. This occurs 78 cycles after the reset state of %0000000. The 64KHz clock is a 5-bit XOR counter with polynomial x^5+x^3+1 and a forced '1' bit on %00010, occurring 19 cycles after the reset state of %11111. If IRQs are enabled for unlinked timers using these clocks with period 0, the IRQ is asserted in IRQST 83 and 24 cycles after the write to SKCTL that clears init mode.

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, even on different timers. In addition, timers 1 and 3 can be switched to the fast 1.79MHz clock through AUDCTL bits 6 and 5.

Timer period

For timers running at 1.8MHz with AUDFx = N, the period of the timer is N+4 cycles. +1 of this is because the counter is reloaded on underflow and thus must count below $00. The other +3 is because of three cycles of delay from the counter being split into multiple stages and for the underflow logic.

For timers using the 15KHz or 64KHz clock, the period is (N+1)*114 cycles for the 15KHz clock and (N+1)*28 cycles for the 64KHz clock. The three cycles of delay do not matter in this case because they are absorbed by the delay until the next audio clock pulse, which occurs 114 or 28 cycles from the last pulse regardless of how long the audio timer takes to reset. This also results in different timers using the same clock synchronizing to the nearest 15KHz or 64KHz tick.

Some references show an additional factor of two in the relation between timer AUDF1-4 value and the frequency. This is because in the common divide-by-two mode (distortion 10), each timer period causes the output to toggle. The frequency of the resulting square wave is thus half the rate at which the timer underflows. For instance, the closest NTSC AUDF value to play a 440Hz tone with the 64KHz clock is 72. This results in the timer generating pulses at a rate of 1.79MHz ÷ (73 × 28) = 875.6Hz. Two pulses are required to generate a cycle, so the resulting square wave is half that at 437.8Hz. A similar divide-by-two is seen with the serial port, which also requires two clock pulses per bit. However, the undivided rate is relevant for other uses of the timer, such as IRQ generation and the noise sampling distortion modes.

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, by making the following changes:

- The automatic reload on underflow is suppressed on the low timer.
- The normal clock input to the high timer is replaced by the low timer output, so the high timer only counts when the low timer underflows.
- When the high timer underflows, both the low and high timer counters are reloaded together.

Typically, linking is used with the 1.8MHz clock on the low timer to achieve a high-precision timer. However, it can also be used with the 15KHz and 64KHz clocks. The high timer – timer 2 or 4 – is the one that has the combined period and is the one that should be enabled for audio, IRQs, or serial port clocking.

For the 15KHz and 64KHz clocks, the period of the linked timer is (N+1)*114 or (N+1)*28, where N = AUDF1 + AUDF2*256 or AUDF3 + AUDF4*256. When the 1.79MHz clock is used, the period is N+7 cycles. This is three cycles greater than for an unlinked timer due to increased delay for the timers to reset, since low timer must underflow before the high timer can underflow, and only then can both reset. As with unlinked timers, this delay is effectively hidden when the 15/64KHz clocks are used.

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. Normally, the low audio channel is muted and only the high channel is used. However, it can also be reused for volume-only effects or even enabled for special effects without affecting the high channel.

Linked timer fire timing

While linked timers are intended to be used as a single high-precision timer, both channels are still active and in particular the low channel still sends clocking pulses to the output circuitry. Because linking disables the normal reload for the low channel, it first counts down and underflows from its initial period and then continues to count down and underflow every 256 ticks after that until the high timer also underflows and resets both timers. For instance, with a 16-bit period of $0140, the low timer will fire after counting down from $40, then again after another 256 ticks. The high timer in turn counts down from $01 to $00 after the first underflow and then to $FF after the second underflow, after which both timers are quickly reset. The reset of both counters does not cause the low timer to fire again. As a result, the low timer in a linked pair will fire AUDF2+1 or AUDF4+1 times for each time the high timer fires.

The high timer underflows and fires three cycles after the low timer does at the end of each high timer cycle, with a corresponding three cycle delay for changes to the audio output or interrupt status.

Distortion (waveform) selection

Bits 4-7 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 4 enables volume-only mode. When set, the waveform output is overridden and hardwired on at the output. None of the other distortion bits affect the audio output in this mode, though they still do affect hidden state in the audio circuitry, as the clocking and noise circuits still run but just don’t have an effect on the audio output.

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, omitting pulses that would cause the output to toggle or sample the 4/9/17-bit noise generators. This gives a rougher, uneven sound that is different than the other generators.

AUDCx[7:4]	Output
0000 (0)	Poly-5 clocked poly-9/17
0010 (2) 0110 (6)	Poly-5 clocked square wave
0100 (4)	Poly-5 clocked poly-4
1000 (8)	Poly-9/17
1010 (10) 1110 (14)	Square wave
1100 (12)	Poly-4
xxx1	Volume-only

Table 8: Distortion modes

The distortion mode setting has no effect on timer IRQs or serial port clocking, and the results of the distortion setting are not observable by the 6502.

Noise sampling artifacts

In POKEY, all channels share a common set of psuedo-random noise generators that all run at machine clock rate. Noise is generated for each channel by sampling the output of the generator at the channel’s rate, with a long period (lower pitch) causing channels to skip more bits in the noise output between each sample. This differs from the 2600’s TIA where each channel has a dedicated noise circuit generating bits at the channel rate. POKEY’s sampling behavior can result in undesirable patterns due to interactions between the periods of the noise generators and the channel timers. This happens when the two periods have a common factor.

For instance, the 4-bit generator has a period of N bits. An audio channel with a period of P cycles will sample the output of this generator every P cycles, or equivalently advance within the pattern every (P mod N) bits. If the period is divisible by N, the same bit in the pattern will be sampled each time and no sound will occur. With the 4-bit generator, N=15, and thus this occurs whenever the period is a multiple of 15 cycles.

For timer periods that aren't divisible by the noise period, artifacts will still occur if the two periods share a common factor. This occurs because advancing by (P mod N) bits will only visit a subset of the bits. Given an audio channel using the 64KHz clock and AUDFx=$CC, P=5740 and (P mod 15)=10, so it advances 10 bits at a time. As a result, only three bits of the pattern will ever be used.

The three bits that *are* used, however, will depend upon the timing offset of the channel relative to the noise pattern. This will tend to change whenever a sound is played, and the different sets of bits used will change the timbre of the output. The 4-bit generator's 15 bit pattern is 000111011001010, so sampling it every 10 bits gives the following possibilities: 001, 010, 001, 111, and 100. Four of these are similar, but one of them is a constant 1 bit -- giving a 1-in-5 chance that this configuration gives no sound.

Note that these interactions are based on the period in *cycles*, not ticks of the audio clock. When using the 15KHz or 64KHz audio clock, an AUDFx value of (F+1) divisible by N will produce silence, but other values can as well. As an sample, using AUDFx=$48 gives a period of 73 audio ticks, which is not divisible by the 9-bit generator's period of 511 bits. However, when combined with the 64KHz clock, the period in cycles is 73*28 = 2044 cycles. Since 2044 = 511*4, the result is silent.

This problem can be avoided by using a period in cycles that is relatively prime to the period of the noise generator, ensuring that all bits in the noise pattern are used and guaranteeing that the output is not constant. Always using period values that result in the same (P mod N) value will also guarantee that the sampled pattern is the same and avoid changing the timbre of the resulting sound. With the 1.8MHz clock, the period can also be detuned slightly to always advance the noise pattern by one bit each period, mimicking the TIA's behavior.

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.

The output from each channel is biased, producing either zero or a negative voltage depending on the channel's volume level. Thus, increasing the volume also increases the average DC bias in the output, and changing the volume can result in audible stepping noise. However, if the channel's digital output is 0, there will be no difference in the analog output regardless of volume level.

The 4-bit DAC for each channel also has somewhat mismatched outputs for each bit. In particular, the ratios between the drivers for the lower two volume bits don't quite match the ratios for the higher two bits, resulting the gaps between volume levels 3 and 4, 7 and 8, and 11 and 12 being a bit wider than expected.

In addition, the mixed output from all four channels starts to show non-linear saturation effects at higher total volume levels. The output is nearly linear within the range of a single channel, where the volume sum of channels with an active output is 15 or less. However, the remainder of the range 16-60 is only about double the amplitude, and two channels actively outputting at volume 15 only achieve about 50% higher amplitude than a single channel. This has the effect of compressing the output, amplifying quieter sounds and attenuating louder ones. Also, because of the previously mentioned biased output from each channel, a channel that is producing a constant 1 bit at non-zero volume can distort the output by shifting the audio output into the saturation range. A channel with constant output 0, however, contributes no distortion regardless of its volume level.

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.

Volume-only mode overrides the output of but does not disable the high-pass or noise flip-flops, which will continue to sample the noise patterns and high-pass sources.

High-pass filter

Channels 1 and 2 have a high-pass filter which is enabled by bits 2 and 1 of AUDCTL, respectively. This uses channel 3 or 4 to clock a high-pass flip-flop that captures the output of the lower channel and XORs against it, canceling the output back to a 0 whenever the higher channel ticks. This zeroes part of the lower channel's output, acting as a crude high-pass filter.

The high-pass signal path routes from the clock output of the high channel’s countdown timer to an XOR immediately before the volume-only override and DAC on the low channel. Thus, the high-pass effect will stack with any noise and volume settings on the low-channel, but is overridden by volume-only mode. None of the AUDC3/4 bits on the high channel affect high-pass operation. They will still affect the high channel’s audio output, which is usually set to volume 0 to mute it when using ch3/4 to drive the high-pass filter.

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 digital output 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.

The high-pass update path has about 1.5 cycles of delay from the high channel’s clock to the low channel’s XOR output. If channels 1 and 3 have their timers synchronized to a period of P cycles and channel 3 is running two cycles ahead of channel 1, a half cycle pulse will be produced per period. The half-cycle offset makes it impossible for the high-pass filter to completely cancel the lower channel's output.

Despite its name, the high-pass circuit does not behave like a conventional high-pass filter and has a complex effect on the output frequency spectrum. Similarly to adding sine waves, the high-pass output often has strong components at the sum and difference of the two channel frequencies. With square wave output on the low channel, this results in a varied duty cycle at double the frequency when the channel frequencies are the same, or a pulse width approximation to a sawtooth when they are slightly different.

For instance, with channels 1 and 3 at 1.79MHz (NTSC) with AUDF1=$3B and AUDF3=$3C, the frequencies of the timers are 1.78977MHz ÷ (59 + 4) = 28409.1Hz and 1.78977MHz ÷ (60 + 4) = 27965.2Hz. The result is a sawtooth at 28409.1Hz - 27965.2Hz = 443.9Hz and an inaudible carrier at 28409.1Hz + 27965.2Hz = 56.4KHz.

Resetting the timers

Writing to the STIMER register causes all of the timers to reload and sets the output flip-flops to 1. 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.

Resetting timers with STIMER does not fire the timers. No IRQs are asserted, and no clock pulses are sent to the audio circuits.

STIMER has no effect on the phase offset of the 15KHz and 64KHz clocks, which can only be reset through initialization mode. Regardless of when STIMER 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.

STIMER preemption

If STIMER is written soon enough before when a timer would fire, the timer counter is reloaded before it can underflow and the related actions are preempted. This can include sending a clock pulse to the audio circuit, updating the serial port, or triggering an IRQ. Due to pipeline delays, this must be done at least a couple of cycles ahead of the timer underflow.

The last cycle that STIMER can be written to preempt the next underflow action is four cycles before the timer IRQ would be asserted in IRQST. For 8-bit timer 1 at 1.8MHz with AUDF1=0 started with an STIMER write at T+0, STIMER must be written again at T+4 to block the timer 1 IRQ being asserted in IRQST at T+8.

The extra two cycle period adjustment from two-tone mode does not affect this timing, because that is from an extra counter reload that occurs after the initial underflow.

Reload timing

In general, the counter for a timer is only reloaded at the start of each period. Any changes to the timer's corresponding AUDF register only take effect at the next reload. For the first period after a write to STIMER, this reload occurs three cycles after a write to STIMER. This means that the last time the AUDF register can be updated to take effect for the first period is two cycles after the STIMER write. However, this is only possible with an accelerator as the 6502 can't write to both STIMER and AUDF that quickly.

Afterward, the next reload occurs a full period after the previous reload for an 8-bit timer. For instance, given an 8-bit timer at 1.8MHz with AUDF1=0, if the STIMER write is at cycle T+0, the first reload will be at T+3 and the second reload at T+7, making the deadlines to update T+2 and T+6. With AUDF1=5, the reloads are at T+3 and T+12 with deadlines T+2 and T+11. These reload times are also one cycle before the timer IRQ is signaled in IRQST, if enabled.

In linked (16-bit) mode, the timing for the low timer is changed due to the combined timer reload. The IRQ and STIMER preemption timing are similar to the 8-bit case, but the reload of the low timer is delayed by 3 cycles to match the high timer, and this also pushes out the deadline for writing AUDF1/3 by 3 cycles. For instance, with AUDF1=0 and AUDF2=0, the second reload for both timers is at T+9.

Two-tone mode also changes the reload timing deadlines because of the extra resync reload that occurs two cycles after the regular reload. Similarly to the linked case, the IRQ/preemption timing is the same, but the second reload overrides the first and pushes the reload times out by two cycles. This means that for timer 1 in 8-bit two tone mode with AUDF1=0, the reload times are T+3 and T+9 instead of T+3 and T+7. This can be delayed by one more cycle in the double-resync case where the timer 1 and timer 2 periods are one cycle part, causing both timers to resync each other and extending both the reload timing and effective period by 3 cycles.

	Cycle																										
Phase	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
Write to STIMER	W		F																								
8-bit loop 1 (AUDF1=0)					P		F		1		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨									
8-bit loop 2 (AUDF1=0)									P		F		1		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨					
8-bit loop 3 (AUDF1=0)													P		F		1		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨	
16-bit timer 1 loop 1 (AUDF1/2=0)									1	F	▨	🟦	🟦	🟦	🟦	🟦	🟦	▨									
16-bit timer 1 loop 2 (AUDF1/2=0)																1	F	▨	🟦	🟦	🟦	🟦	🟦	🟦	▨		
16-bit timer 1 loop 3 (AUDF1/2=0)																							1	F	▨	🟦	🟦
16-bit timer 2 loop 1 (AUDF1/2=0)										F		2		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨						
16-bit timer 2 loop 2 (AUDF1/2=0)																	F		2		▨	🟦	🟦	🟦	🟦	🟦	🟦
16-bit timer 2 loop 3 (AUDF1/2=0)																								F		2	
8-bit loop 1 (AUDF1=0 two tone)					P				1F		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨									
8-bit loop 2 (AUDF1=0 two tone)											P				1F		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨			
8-bit loop 3 (AUDF1=0 two tone)																	P				1F		▨	🟦	🟦	🟦	🟦
16-bit timer 1 loop 1 (AUDF1/2=0 two tone 0-bit)									1		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨									
16-bit timer 1 loop 2 (AUDF1/2=0 two tone 0-bit)																		1		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨
16-bit timer 2 loop 1 (AUDF1/2=0 two tone 0-bit)												2		▨	🟦	🟦	🟦	🟦	🟦	🟦	▨						
16-bit timer 2 loop 2 (AUDF1/2=0 two tone 0-bit)																					2		▨	🟦	🟦	🟦	🟦

W	Write to STIMER (last cycle of STA)	1	First cycle with IRQST bit 1 asserted (timer 1)
F	Last cycle to write AUDF to affect next period	2	First cycle with IRQST bit 2 asserted (timer 2)
P	Last cycle to write STIMER to preempt timer	🟦	Earliest timing for 6502 IRQ ACK (common)
1F	Last AUDF write and first IRQST bit 1 cycle	▨	Earliest timing for 6502 IRQ ACK (1c machine variation)

Figure 7: POKEY detailed timer timing

5.4 Clock generation

There are three clocks that can by used to drive the timer counters. The first is the machine clock, which runs at 1.79MHz for NTSC and 1.77MHz for PAL. However, this clock runs much too fast for audio. Thus, the machine clock is divided by 28 and 114 within POKEY to provide additional 64KHz and 15KHz clocks with more useful range for audio. Because these are based on the machine clock and the machine clock varies slightly between NTSC and PAL machines, this means that the same divisor values in the AUDF1-4 registers produce slightly different frequencies between NTSC/PAL machines as well.

		Ideal Clock Rate	
Clock	**Divisor**	**NTSC**	**PAL**
Machine	÷ 1	1.78977MHz	1.77345MHz
64KHz	÷ 28	63.920KHz	63.337KHz
15KHz	÷ 114	15.700KHz	15.557KHz

Table 9: POKEY clock frequencies

Channels 1 and 3 can use the 1.8MHz machine clock independently, depending on bits 6 and 5 of AUDCTL [D208]. Channels 2 and 4, however, can only use the 64KHz or 15KHz clocks unless linked to channels 1 or 3 in 16-bit mode. Additionally, all channels must exclusively use either the 64KHz or 15KHz clocks together, as there is only one control bit to select between those two clocks, AUDCTL bit 0.

Both the 64KHz and 15KHz clocks are generated by polynomial counters internal to POKEY and have no guaranteed phase relation to other clocks in the system. In particular, the 15.7KHz clock uses the same divisor as ANTIC's horizontal scan counter and thus runs at the same rate, but there is no connection to synchronize the two. Instead, its phase is determined by when initialization mode is ended. Thus, the 15KHz clock may tick at an arbitrary offset in the scan line, but will remain locked to the same horizontal offset until initialization mode is re-entered. This can be used in conjunction with timer IRQs for more flexible interrupt timing than DLIs.

While the 64KHz clock is used exclusively for audio, the 15KHz clock is also used for keyboard and paddle scanning.

5.5 Noise generators

POKEY contains three noise generators, all composed of XNOR-based maximal-length linear feedback shift registers (LFSRs, or polynomial counters) that run at 1.8MHz. These are used both for generating audio noise as well as random numbers for the CPU.

All noise generators are shared between all four audio channels, and once initialized, are constantly running at a rate of 1 bit per machine cycle. On each cycle, a bit is shifted out the end for use and a new bit shifted in from the other end, computed as the XNOR of the end bit and one of the middle bits.

As maximal-length generators, each N-bit generator has a period of $2^N - 1$, so the 4-bit generator repeats every 15 bits, and the 9 bit generator every 511 bits. This also means the generator patterns are slightly biased with one more 0 bit than 1 bit. When exiting initialization mode, all generators start zeroed.

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: 000011101100101.

The 5-bit generator has the pattern: 0000011100100010101111011010011.

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. RANDOM shifts right at the rate of one bit per machine cycle. Note that RANDOM reads bits inverted from the shift register itself and the bits seen by the audio circuits.

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.

For the 9/17-bit generator, the audio channels receive noise from the right end of RANDOM. This means that all noise bits are visible to the CPU before they reach the audio circuitry.

Linked channels sample the noise generators from whichever channels are generating audio. Normally only the higher channel has nonzero volume, so its timing determines the noise produced. If the lower channel is enabled to produce sound, its noise output will be determined by that channel's timing, one cycle away from the higher channel.

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 implementation of the counter. 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.

Shortly after entering initialization mode, the audio circuits see a constant 0 from the 4-bit and 9/17-bit generators, while the 5-bit generator is a constant 1.

When exiting initialization mode, the polynomial counters begin counting immediately. For instance, if 9-bit mode is selected, executing STA SKCTL + LDA RANDOM back-to-back will give A=$1F, which is four bits after the all ones state.

When initialization mode is re-entered, 1s are shifted into RANDOM from the left side. Beginning with nine cycles after SKCTL is written to first enter init mode, RANDOM will always read $FF. Another nine cycles are needed for the rest of the shift register to clear; it will work fine if restarted earlier, but not all of the bits will have been reset.

5.6 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. Two serial shift registers are used, one for input and one for output, and shared timers are used to clock the shift registers.

Input shift register

Serial reception involves a 10-bit shift register and an 8-bit SERIN data register. When the serial input line is pulled low for the start bit, the input shift register begins shifting until it has received a total of ten bits: the start bit, eight data bits in order from LSB to MSB, and the stop bit. The eight data bits are copied into the SERIN register and the serial input ready IRQ (IRQEN/IRQST bit 5) is asserted if enabled to indicate that a byte is available. The shift register can then begin immediately receiving a new byte while the CPU reads the last byte from SERIN.

The SERIN register always contains the last input byte received, whether or not it had errors. Reading the SERIN register has no side effects; it does not acknowledge reception of a byte and it can be read multiple times before the next byte arrives. Only one byte can be queued in SERIN, as the next byte will be copied into it as soon as it completes even if no further bytes arrive and the shift register goes idle.

SKSTAT bit 1 indicates when the serial input shift register is active. It switches to 0 when the start bit is sampled and back to 1 when the stop bit is sampled. This means that when receiving back-to-back bytes, this bit is low 90% of the time (9 bit cells out of 10); it does not stay low continuously.

Output shift register

Serial transmission involves another 10-bit shift register and 8-bit SEROUT data register. These are independent from the input shift register and both can act independently for full-duplex mode. Writing to the SEROUT register sets a bit indicating that a byte has been queued, after which it is copied into the shift register for serial transmission. After that happens, SEROUT can be loaded again with another byte. This allows one byte to queued so that bytes can be sent back-to-back even if small delays occur in loading SEROUT.

Only one byte can be loaded into SEROUT at a time. If a second byte is written before the first can be loaded into the shift register, the second byte replaces the first and only the second byte is sent.

The serial output ready IRQ (IRQEN/ST bit 4) is asserted when a load occurs from SEROUT to the ouput shift register and thus SEROUT is ready for another byte. This only occurs if a byte is pending in SEROUT, and thus the serial output must be primed by writing the first byte to SEROUT without waiting for the output ready interrupt. Continuous transmission is assured as long as SEROUT is reloaded before the previous byte finishes shifting out. The serial output ready IRQ is not asserted when shifting completes if no new byte is ready to load.

The serial output complete IRQ (IRQEN/ST bit 3) is asserted whenever the output shift register is idle. Unlike other POKEY IRQs, it is not latched – it will stay asserted even if it is disabled in IRQEN and will automatically deassert when shifting begins. This IRQ will stay inactive continuously while sending back-to-back bytes. One use for the output complete IRQ is to determine when shifting has completed so the serial hardware can be reconfigured. Another is to send with two stop bits instead of one stop bit, by using the output complete IRQ in place of the output ready IRQ.

Warning

The output shift register only attempts to load once every bit cell time on the rising edge of the serial clock and thus there is a delay from the first write to SEROUT until the serial output ready/complete IRQs update and SEROUT is ready for a second byte. Attempting to write SEROUT twice without a wait for serial output ready in between can fail when the second byte replaces the first before it can be loaded into the shift register.

This also means that it is necessary to wait for serial output ready before checking serial output complete to end a transmission. Otherwise, if there is a delay in queuing bytes and the serial output shift register temporarily idles, it is possible for the serial output complete IRQ to fire *after* SEROUT has been reloaded, in the short time until the output shift register restarts on the next bit cell.

Framing errors

SKSTAT bit 7 reports if a framing error occurs on the serial input port. A framing error occurs when shifting is started by detection of a 0 start bit, but the tenth bit that should be the stop bit is not a 1, indicating that the byte was not received correctly.

Framing errors do not affect the shift/load processes. The errant byte is still loaded into SERIN and a receive IRQ is still asserted, despite the error.

Overrun errors

SKSTAT bit 5 indicates whether an overrun has occurred.[21] 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 in SERIN and the previous byte is lost.

The overrun bit is set specifically when a new byte is received and the serial port logic attempts to assert the serial input interrupt when it is already active (IRQST bit 5 set to 0). This means that in order to acknowledge receipt of a byte from SERIN, the serial input interrupt (IRQST bit 5) must be reset. The interrupt should also be cleared before the start of a receive operation to clear any previously received stray data. Overruns are not detected if this interrupt is disabled.

Reading a data byte from SERIN by itself has no bearing on whether an overrun is detected, only the interrupt status.

The overrun bit in SKSTAT is sticky. Once an overrun occurs, it will remain set until the next write to SKRES, even if subsequent bytes are received without an overrun.

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, and because there is necessarily a delay between the IRQ being acknowledged and the byte being read from SERIN.

There is no overrun detection for output. Writing SEROUT again when a byte is already pending simply replaces the previously queued byte with the new byte. Only one byte can be queued behind the one that is actively being shifted out.

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

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 raw state of the serial input line. This bypasses all of the shifting and clocking logic and ignores all serial input settings, working even if all clocks are stopped. 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:[22]

Setting	Input clock	Output clock	External 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[23]	Channel 2	Output channel 4
111	Channel 3+4 (async)	Channel 2	Input

Table 10: Serial port timing modes

The modes for standard half-duplex SIO operation are 001 for reception and 010 for transmission. The external clock output 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.[24] Channels 3+4 should also be linked together and driven by the 1.79MHz clock for highest precision. For cassette

[22] [ATA82] II.27 has the official mode chart; see also unnumbered page with serial/audio diagram for exact switch and gate layout.
[23] [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.
[24] [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.

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.

The fastest that the serial port can run using internal clocking is 128 kilobaud, since the only way that the 1.79MHz clock can be connected to the serial port is through linked timers, which divides the clock by 7 to 256KHz, which is then divided by two in the serial port logic to 128 kilobits/sec.

Serial port 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.

This does not reset the serial input and output state machines, however. Initialization mode is required to do this, by setting bits 0-1 of SKCTL to 0. This interrupts any bytes currently being shifted in the input or output shift registers and flushes any byte that was queued for output in the SEROUT register. Finally, clearing bit 3 is needed to reset the output state of the toggle flip-flop used by two-tone mode. Thus, the serial port is fully reset by writing $00 to SKCTL.

One aspect of the serial point that is not reset by either means, however, is the bit currently being output on the SIO DATA OUT signal of the SIO bus. While initialization mode does reset the serial output state machine and stop the serial output shift register, it does not actually clear the shift register, leaving it outputting the last bit shifted out. Thus, interrupting a send can result in the SIO DATA OUT line being stuck in the low state, which can cause devices on the SIO bus to eternally attempt to shift in data bytes. This is not ordinarily a problem, as the command line is deasserted so that devices won’t attempt to read in command frames, but it can be an issue if a device is listening for non-command data or when attempting to select timer 1 in two-tone mode.

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.

Asynchronous receive mode

Setting bit 4 of SKCTL [$D20F] enables asynchronous receive mode. In this mode, timers 3 and 4 are held in reset state while POKEY is waiting for a start bit, allowing the timers to run only once a start bit is detected. This aligns timers 3+4 to the leading edge of the byte that the serial port input logic samples roughly in the center of each bit. Without this, the serial port input is not synchronized with the device on the other end and can sample between bits, producing errors. In most cases, and particularly with disk drives, serial input is not reliable without asynchronous mode.

Asynchronous receive mode also has the side effect of producing a characteristic audio tone when the sound output is enabled on timers 3 or 4. This occurs because the audio circuit receives an odd number of pulses for each byte (19), which in pure tone mode causes the audio channel to toggle once per byte. At 19200 baud, this produces approximately a 960Hz tone during the read of each disk sector. The exact pitch produced varies depending on the delay introduced by the device between bytes. This tone does not occur during transmission to the device as that is done in synchronous mode, where the timer(s) used for the output clock run continuously.

Since asynchronous mode holds timers 3 and 4 in reset while waiting for a start bit, those timers are stopped entirely when no data is being received. This means that leaving async mode enabled effectively disables channels 3+4 for all audio except volume-only mode. Therefore, bit 4 of SKCTL should be cleared before attempting to use those channels for audio.[25]

Shift timing

As stated earlier, the serial port logic shifts bits in or out at half the rate of the controlling timer, with the input and output shift registers use alternating phases of the clock if they share the same timer. When asynchronous receive mode is enabled, it presets the clock at the beginning of the start bit so that the input shift register shifts in the start bit one period (half bit) later, sampling the middle of the start bit. Setting SKCTL bits 4-6 to %000 resets the clocks to the opposite phase.

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), which indicates when the contents of SEROUT have been moved to the output shift register. This can take up to an entire bit cell time. If SEROUT is written again before this happens, the second byte overwrites the first and only the second byte is transmitted instead.

Second, if the output shift register is initially idle, the serial output complete IRQ (bit 3) will not deassert until the load occurs, as it only does so after the shift register loads from SEROUT and begins shifting. 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.

[25] [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.

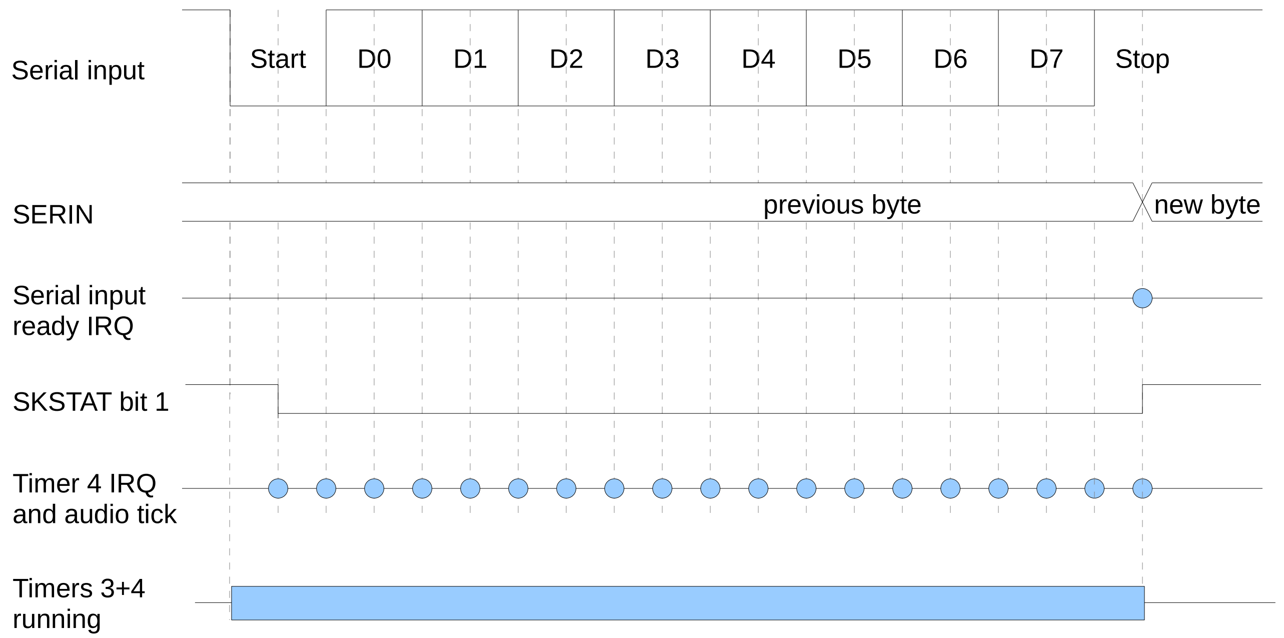


Figure 8: POKEY asynchronous serial receive timing

Figure 8 shows the timing relationships for asynchronous receive timing. Timers 3+4 are held frozen until POKEY detects the leading edge of the start bit, upon which the timers are started. Every timer 4 period is a half-bit, so sampling begins one period later and every two periods after that. 19 timer periods or 9½ bits later, the stop bit is sampled, SERIN is updated with the new data byte, the serial input ready IRQ is asserted, the overrun and framing error bits in SKSTAT are updated, and timers 3+4 are stopped. SKSTAT bit 1, which indicates whether the serial input shifter is busy, asserts halfway into the start bit and deasserts halfway into the stop bit.

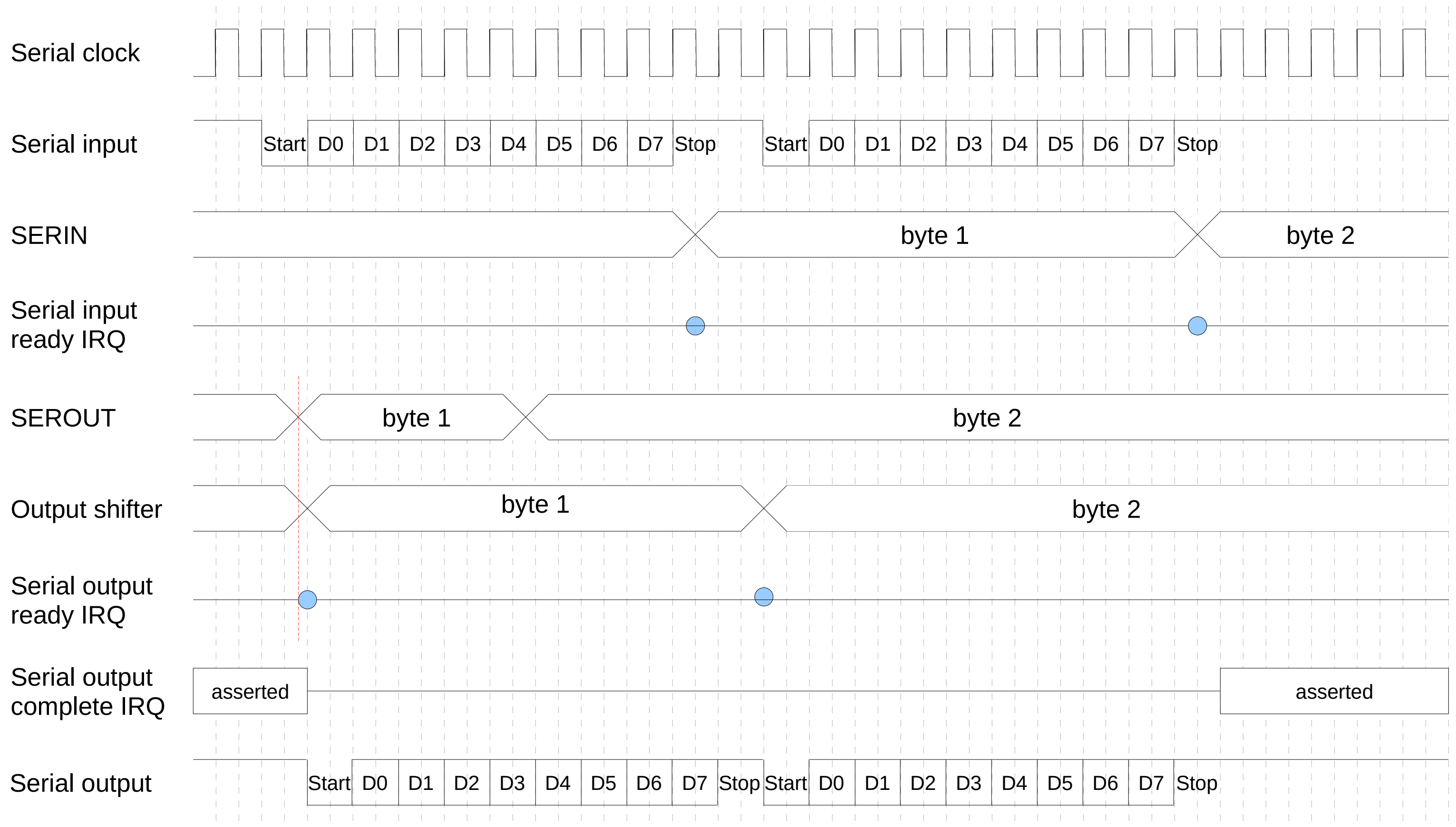

Serial clock
Serial input
Start D0 D1 D2 D3 D4 D5 D6 D7 Stop Start D0 D1 D2 D3 D4 D5 D6 D7 Stop
SERIN
byte 1
byte 2
Serial input ready IRQ
SEROUT
byte 1
byte 2
Output shifter
byte 1
byte 2
Serial output ready IRQ
Serial output complete IRQ
asserted
asserted
Serial output
Start D0 D1 D2 D3 D4 D5 D6 D7 Stop Start D0 D1 D2 D3 D4 D5 D6 D7 Stop

Figure 9: POKEY synchronous serial receive/transmit timing

Figure 9 shows the timing for synchronous reception and transmission. Reception is the same as for asynchronous mode, except that the shift timing is determined by falling edges of the clock alone without the clock being restarted by the leading edge of the start bit. For transmission, each write to SEROUT immediately loads the SEROUT register, but the shift register is only loaded on the next rising edge of the clock. Once this happens, the serial complete IRQ is deasserted and the serial output ready IRQ is asserted, if enabled. SEROUT can then be reloaded with the next byte at any time before the first byte finishes sending. The serial output register is automatically reloaded with the second byte from SEROUT, and after it completes and no more bytes are queued, the serial output complete IRQ is reasserted. The serial output ready IRQ is not asserted a third time.

Two-tone mode

Two-tone mode is used for frequency shift keying (FSK) encoding of serial output, replacing low and high signal levels with two different tones instead. This mode is enabled in POKEY by setting bit 3 of SKCTL. When enabled, the serial output port instead outputs square waves clocked by timers 1 and 2, where timer 1 is used for a 1 bit and timer 2 is used for a 0 bit. This is used most often for writing to cassette tape with the 64KHz clock, with AUDF1=$05 (5327Hz) for 1 bits and AUDF2=$07 (3995Hz) for 0 bits.

The switching between timers 1 and 2 is done in the serial logic and only affects the clock used there; it does not affect the audio circuitry, which will still react to timer 1 and 2 pulses without regard to the data bits being output by the serial port. Thus, the AUDC1 and AUDC2 registers do not affect the serial output and do not need to be set to pure tone mode; they are instead normally shut off at volume 0. The serial output port has its own divide-by-two to generate the tones that is also independent from the audio circutry. It is not reset by STIMER, though it *is* reset to 0 when two-tone mode is disabled.

Two-tone mode has no effect on serial input. Unlike tape write operations, where POKEY must be configured to encode to FSK on serial output, POKEY has no logic for FSK decoding on input. This is done by the cassette tape deck instead, which outputs a conventional data stream for POKEY's serial input. Thus, two-tone mode is not needed for and not typically enabled for tape reads.

Two-tone resync

Two-tone mode does still have some effect on audio output because it frequently resets timers 1 and 2 for continuous phase in the FSK output. Specifically, whenever the serial output toggles due to one of the timers, both timers are reset. The effect is that whenever the output data bit changes, the serial output seamlessly switches from one tone to the other without glitches in the output from partial pulse widths. However, resyncing the two timers this way can also alter the frequency or even mute the timers.

When a resync happens, both timers 1 and 2 have their counters reset to AUDF1 and AUDF2 values. As with STIMER, this does not send pulses to the audio circuits. This means that if the faster timer is driving the resync, it will continually interrupt the slower timer and prevent it from clocking its audio circuit. However, if the slower timer is driving the resync, then the faster timer will still be able to clock its audio circuit one or more times before being reset, although its frequency will be lowered to a multiple of the slower timer's frequency.

There is an asymmetry in the data bit switching logic which imposes a frequency requirement on the timers. Timer 1 pulses are only used by the serial output for a 1 bit, but timer 2 pulses are *always* used, causing a resync and toggling the output regardless of the current data bit.[26] This means that timer 2 must have a lower frequency for two-tone mode to function as intended.[27] Otherwise, timer 2 will preempt timer 1 before it can fire even when the current data bit is a 1, silencing the tone for 1 bits.

Two-tone resync timing

The timer 1+2 reset in two-tone mode occurs two cycles after the timer that triggered the resync reloads. This doesn't matter in normal cassette write operation with the 64KHz clock, but it becomes important with timer 1 clocked at 1.79MHz.

The first effect of the delay is that if timer 1 at 1.79MHz drives the resync, it will have a period of two cycles longer than usual, due to being re-reloaded two cycles after the normal reload. Note that this only affects the second and subsequent periods after an STIMER reset, as there is no timer 1+2 resync at the start. Thus, a normal timer 1 configuration that would fire 8, 12, and 16 cycles after STIMER would instead fire after 8, 14, and 20 cycles instead. This change in period doesn't occur when using the 15KHz/64KHz clock for timer 1+2, as the

[26] 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.
[27] [ATA82] II.26

delay will be absorbed into the period to the next clock tick, which will fire 28 or 114 cycles later regardless.

The second effect is that the non-resyncing timer can fire up to one cycle later than the resyncing timer without being preempted. For instance, if timer 2 is set to a period of 28*3 = 84 cycles, timer 1 can be set to up to 85 cycles and still fire. If timer 1 is set to 86 cycles, then it will be preempted and not assert an IRQ or fire an audio tick.

The third effect is that it is possible for the two timers to mutually reset each other due to the delay. If both timers fire on the same cycle, then there is only one resync that occurs. However, if one timer has a period one cycle later than the other and both are enabled for resync (mark output), then each timer will trigger a resync of both timers two cycles after their normal reload, with two resyncs occurring back-to-back. When timer 2 fires one cycle after timer 1, this effectively adds three cycles to the next timer 1 period.

Two-tone force break mode

In two-tone mode, the force break bit (SKCTL bit 7) forces the output to the *tone* for a 0 bit, instead of actually sending a constant 0 out the serial port. This is done by overriding the data bit input to the timer input switch so that pulses from timer 2, the timer for 0 data bits, is used for the output tone and timer resync regardless of the serial output data bit state.

This mode is sometimes useful when using two-tone mode for audio purposes instead of serial output, since it forces use of timer 2 regardless of the state of the serial output shift register.

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.

With the exception of bit 3, the status bit for a disabled interrupt is always locked to a 1. There is no interrupt queuing for a disabled interrupt – any interrupts that would have triggered while an interrupt is disabled are lost.

Disabling all interrupt sources in IRQEN does not block all IRQs; POKEY shares the IRQ control line with the PIA and the Parallel Bus Interface, which can also trigger IRQs on the CPU.

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.[28]

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.

It is also possible for the IRQ handler to be entered without an interrupt being signaled in IRQST by means of the serial output complete IRQ. This can happen because the serial output complete IRQ deasserts automatically once a new byte is loaded into the output shift register and there is a delay from when SEROUT is written to when this occurs.

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.

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. 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.

Whenever a key code is latched into KBCODE, bits 6 and 7 are also set to indicate the state of the Shift and Control keys, respectively. These bits are updated at the same time as bit 0-5 and do not change if Shift or Control changes state without another key press. However, SKSTAT bit 3 is updated whenever a change in the Shift key is detected even if no other key is pressed.

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

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. Tables 11 and 12 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	Ret	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	G	S	A

Table 11: Key codes (scan matrix layout)

Help **[11]**	Start	Select	Option	Reset

Esc **[1C]**	! 1 **[1F]**	" 2 **[1E]**	# 3 **[1A]**	$ 4 **[18]**	% 5 **[1D]**	& 6 **[1B]**	' 7 **[33]**	@ 8 **[35]**	(9 **[30]**	) 0 **[32]**	Clr < **[36]**	Ins > **[37]**	Bksp **[34]**	Break

Tab **[2C]**	Q **[2F]**	W **[2E]**	E **[2A]**	R **[28]**	T **[2D]**	Y **[2B]**	U **[0B]**	I **[0D]**	O **[08]**	P **[0A]**	_ - **[0E]**	| = **[0F]**	Return **[0C]**

Control	A **[3F]**	S **[3E]**	D **[3A]**	F **[38]**	G **[3D]**	H **[39]**	J **[01]**	K **[05]**	L **[00]**	: ; **[02]**	\ + **[06]**	^ * **[07]**	Caps **[3C]**

Shift	Z **[17]**	X **[16]**	C **[12]**	V **[10]**	B **[15]**	N **[23]**	M **[25]**	[, **[20]**	] . **[22]**	? / **[26]**	Shift	Inv **[27]**

Space **[21]**

Table 12: Key codes (130XE keyboard layout)

The Start, Select, Option, and Reset keys do not have key codes associated with them as they are detected differently; Shift and Control keys function as modifier keys.

Six key codes are not mapped to keys in the keyboard matrix and cannot be triggered in normal operation. The function keys F1-F4 are also absent on all models except the 1200XL. However, the absent key codes can be triggered on a stock keyboard if the keyboard debounce is disabled (SKCTL=$02). For instance, pressing O+V+Help quickly with debounce disabled can latch $09 into KBCODE. This is also theoretically possible with debounce enabled, but improbable due to the very tight timing requirements.

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 code replaces the old one in KBCODE. Reading KBCODE has no effect on the IRQ or overrun state.

A keyboard 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[29]:

- **State 0 (key up)**:
  - If a key is down, latch the counter in the compare register and go to state 1.
- **State 1 (debounce key down)**:
  - 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 (confirmed key down)**:
  - If the counter matches the compare register, and the current key is not down, go to state 2.
- **State 2 (debounce key up)**:
  - 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.

SKSTAT bit 2, which indicates whether a key is held down, is updated based on the current state of the keyboard state machine. It is only updated after a key up or down has been debounced, and asserts at the same time that KBCODE is updated and the IRQ is triggered.

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

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

Control/Shift/Break scan

The Control, Shift, and Break keys are detected in parallel with other keys during the scan. Control and Shift are detected concurrently with scan codes $00 and $10, while Break is detected with $30. Control and Shift states are captured into internal latches that are later fed into SKSTAT bit 3 and KBCODE bits 6 and 7; Break is used to trigger the Break key IRQ. These keys are not affected by debounce, are not reflected in SKSTAT bit 2, and do not trigger keyboard overruns.

Both Shift keys are wired identically and cannot be distinguished in software.

The Break key is monitored only for key down transitions. When a Break key press is detected, the Break key IRQ (bit 7) is asserted if enabled. The IRQ will not retrigger while the key is held down and the key press is lost if the IRQ is disabled when the key down occurs, as the IRQ will not retrigger when re-enabled even if Break is still down. There is no direct way to determine if it is still depressed or when it is released.[30]

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. This causes SKSTAT bit 2 to reset to 1. However, KBCODE and any previously signaled keyboard IRQs are unaffected, and SKSTAT bit 3 retains its current state regardless of subsequent movements of the Shift key.

The Break key cannot be detected while the keyboard scan is disabled.

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 or more keys are pressed simultaneously. This is because the keyboard scan logic does a second pass over the keyboard to verify a key press and rejects it if another key is depressed at the same time. However, once a key press has been registered, any other keys are ignored and the key will continue to be reported as depressed until it is released.

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 afterward will register it for only two additional scan lines before reporting a release, even if the key is still held down. This pattern will also repeat every time the keyboard is scanned, so holding down a pair of adjacent keys will cause the second key to be reported once every 64 scan lines (~240 times a second). These effects are typically undesirable and so debounce normally must be enabled for normal keyboard operation.

Turning debounce off while a key is recognized as held down will cause a key up transition, as the keyboard state machine will transition through key up as it scans unactivated keys and then fail to re-enter key down due to failing to find two consecutive keys down.

Note that the 5200 keyboard is the opposite: it requires debounce to be disabled to function. See chapter 13 for details.

Shift and Break key detection are not affected by the debounce function – they are reported via SKSTAT bit 3 and IRQST bit 7 in the same manner regardless.

[30] It is possible to detect the Break state indirectly by phantom keys if specific other keys are depressed at the same time and debounce is disabled, but this is a corner case useless in practice.

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 on a rectangle in the matrix can result in an additional phantom key appearing at the missing fourth corner of the rectangle. For instance, pressing L, J, and V will also result in the Help key being detected.

Ordinarily, this isn't a problem because the debounce logic prevents any keys from being registered when more than one non-modifier key is down, so adding a fourth key on top of three already pressed makes no difference. If debounce is disabled, however, the phantom keys can be detected when new pairs of keys with adjacent scan codes are activated. With L+J+V, for example, the L+J pair will result in KBCODE=$01, followed by V + phantom Help setting KBCODE=$11.

Where this particularly causes a problem is when two or more of the Control, Shift, or Break keys are pressed in conjunction with another key. These keys share a control line (KR2) and also share row lines with the regular keyboard, with Control sharing with $00-07, Shift with $10-17, and Break with $30-37. This effectively extends the 8x8 matrix to 8x9 and allows these keys to also participate in creating phantom keys. Pressing one of these keys at the same time as a regular key is OK because two keys are not enough to cause a problem, but pressing two of them in addition to a regular key will cause a phantom key 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, due to the keyboard scan seeing two keys down in the regular matrix. Also, Control+V+L will simulate Shift, and Control+L+9 will simulate Break.

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 cannot contribute to phantom keys.

Auto-repeat

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

5.9 Paddle scan

POKEY has inputs for eight potentiometer (“pot”) inputs, typically used to read paddles. Pulse timing is automatically handled in hardware, relieving the CPU of the need to poll the inputs and simply requiring values to be read after the polling has completed.

Polling mechanism

The pot inputs are designed to be connected to a capacitor charged up by a variable voltage, where the voltage is determined by a pot-based divider in the paddle controller. The voltage from the controller determines the rate at which the capacitor charges up, which in turn changes the time taken for the capacitor to charge up to threshold voltage. POKEY measures how long this takes and reports this as the controller position. Afterward, dumping transistors drain the capacitors and reset them for the next read.

The polling process is started by a write to the POTGO register. This resets the main counter, which begins counting up from 0 once per scan line. The value of this counter is continuously latched into the each pot position register (POT0-7) until the corresponding input reaches threshold. The counter stops at 228, which is latched into the registers for any pot inputs that have not yet reached threshold.

Normally, the polling counter is driven by the same clock that is used by the keyboard scan. Therefore, paddle scanning is inoperative when initialization mode is active and the keyboard clock is frozen. The exception is when fast pot scan mode is enabled, which selects the machine clock instead of the 15KHz clock.

Polling status

The ALLPOT register can be used to check the state of each input. Each bit in ALLPOT corresponds to one of the inputs and reads 1 while the input is being polled, then 0 when polling is complete. Therefore, the amount of time that each bit is 0 is proportional to the count eventually read. ALLPOT is forced to $00 once the scan has finished, regardless of the state of the inputs.

ALLPOT also indicates which POTn registers are being updated continuously from the master counter. If an input somehow dips back below input threshold during the scan, the corresponding bit in ALLPOT will revert to 1 and the POTn register will resume updating from the main counter.

The POT counters can be read while the scan is occurring. The main counter is continuously copied into the POT0-7 registers until each input reaches threshold, so all counters will count up in sync until then.

Fast pot scan mode

Setting bit 2 of SKCTL enables fast pot scan mode, which switches the counter from the 15.7KHz keyboard clock to the 1.79MHz machine clock. The scan time is shortened to two scan lines, and the values latched into the POT0-7 counters are proportional to clock cycles instead of scan lines. When used with regular paddles, this produces counts that are about 114 times higher, giving 0-229 at the extreme low end of the normal pot value range. The counter stops one value higher in this mode than in normal scan mode (229 vs. 228).

A side effect of enabling fast pot scan mode is that it also disables the capacitor dump transistors. This means that, by default, simply enabling fast pot scan mode will not work because the capacitors will never be drained below threshold voltage after they have changed up, no matter how many times POTGO is strobed. Inputs that have already reached input threshold at the start of the scan will not have their position registers updated at all. The dump transistors are disabled as soon as fast pot scan mode is enabled, so attempting to enable fast pot scan mode and then write POTGO can fail when the capacitors charge up past the threshold in between the two writes. The dump transistors are on whenever slow pot scan mode is enabled and a scan is not occurring.

Reading live counters

Attempting to read the POT0-7 registers during the active pot scan can produce non-monotonic results if those registers are still being updated. Specifically, the bits 0-4 of the count may be blended between cycles such that LSBs reset to 0 before the carry has propagated through to upper bits. The effect is that instead of counting $00-$10, the low bits count in the following sequence: $00, $10, $10, $12, $10, $14, $14, $16, $10, $18, $18, $1A, $18, $1C, $1C, $1E, $10. As a result, fast pot scan mode cannot be used to reliably count cycles.

Note that while the live values during the scan are always even, odd values can be recorded as the final value when the counters stop updating.

Truncated scans

A full paddle scan takes 228 scan lines or cycles, depending on the mode. The entire time is required regardless of when the individual pot scan counters latch. If POTGO is retriggered before the previous scan finishes, the dump transistors will not have had a chance to drain the capacitors before the next scan starts, resulting in truncated counts. For instance, restarting the scan after 64 scan lines will result in counts 64 lower for any inputs reaching threshold after the restart. Continuously restarting the scan can prevent the pot counters from updating at all, because the dump transistors will never get a chance to activate. It takes about 20-40 cycles before the capacitors are sufficiently drained to produce normal values.

Reset

Other than the stopping of the slow pot scan clock, the pot scan logic is not affected by initialization mode. The scan counter may need to count for up to a full 256 counts (not 228) to fully reset the pot scan logic.

Unconnected inputs

On the XL/XE computers and the XEGS, which only have two joystick ports, pot inputs 4-7 are hardwired to ground and will always read 228 (normal scan mode) or 229 (fast scan mode).

5.10 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.

5.11 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 System role

GTIA's primary role is to generate the composite video signal output, combining the playfield graphics produced by ANTIC with player/missile graphics and converting it to a color signal. It also has ancillary roles in reading joystick triggers, driving the console speaker, and sensing console buttons.

Addressing

GTIA's address range is $D000-D0FF, with the canonical range being $D000-D01F. Accesses to $D020-D0FF are mirrored to the same registers.

Address	Read	Write	Power-On Value* (Read)
$D000	M0PF	HPOSP0	$00
$D001	M1PF	HPOSP1	$00
$D002	M2PF	HPOSP2	$00
$D003	M3PF	HPOSP3	$00
$D004	P0PF	HPOSM0	$00
$D005	P1PF	HPOSM1	$00
$D006	P2PF	HPOSM2	$00
$D007	P3PF	HPOSM3	$00
$D008	M0PL	SIZEP0	$0F
$D009	M1PL	SIZEP1	$0F
$D00A	M2PL	SIZEP2	$0F
$D00B	M3PL	SIZEP3	$0F
$D00C	P0PL	SIZEM	$0E
$D00D	P1PL	GRAFP0	$0D
$D00E	P2PL	GRAFP1	$0B
$D00F	P3PL	GRAFP2	$07
$D010	TRIG0	GRAFP3	$01
$D011	TRIG1	GRAFM	$01
$D012	TRIG2	COLPM0	$01
$D013	TRIG3	COLPM1	$01
$D014	PAL	COLPM2	$0F / $01
$D015		COLPM3	$0F
$D016		COLPF0	$0F
$D017		COLPF1	$0F
$D018		COLPF2	$0F
$D019		COLPF3	$0F
$D01A		COLBK	$0F
$D01B		PRIOR	$0F
$D01C		VDELAY	$0F
$D01D		GRACTL	$0F
$D01E		HITCLR	$0F
$D01F	CONSOL	CONSOL	$00

Table 13: GTIA Register Map

Power-on values are for a cold start after the computer has been powered off for 30 seconds. Shaded values are observed but not architecturally guaranteed, and may also be different on a non-cold start since they are not affected by reset. Thus, GTIA registers must be cleared on startup to ensure consistent state.

All data bus bits are driven by GTIA in its address range and there are no floating data bits on read.

> **Note**
>
> VBXE checks for writes to $D080-D0FF to detect a GTIA reset.

Variants

There are three variants of GTIA: the original NTSC version, a version updated for PAL, and a rare third version called FGTIA for SECAM. Aside from the generated video output, the three versions function almost identically from a programming perspective, with the differences between NTSC and PAL being in the PAL detection register.

The FGTIA has slightly more differences in behavior due to some changes to accommodate reassignment of chip package pins: the trigger inputs are serially latched at the start of horizontal blank, and luminance bit 0 is missing in mode 9. However, not all SECAM systems use the FGTIA, as some use the PAL GTIA instead.

NTSC/PAL detection

The PAL register is not actually present in the original NTSC GTIA, with the value returned from unassigned addresses later retroactively defined as the NTSC/PAL detection register at $D014. Only bits 1-3 are defined in the register specification, but in practice all bits are stable. NTSC versions return $0F and PAL/SECAM versions return $01.

6.2 Display generation

Position coordinates

Horizontal and vertical positions of graphics generated by GTIA can be described in 8-bit coordinates. Numerical positions are used directly with player/missile graphics and can also be mapped to playfield graphics.

For horizontal positions, the full scanline consists of 228 color clocks. Player/missile graphics are positioned from the left edge of the player or missile, and a position of $80 aligns the left edge of the player/missile with the center of the screen. Accordingly, $7F is one color clock to the left, and $81 is one color clock to the right. These values are directly written into horizontal position registers in GTIA. P/M graphics are visible in the range of $22-DD (188 color clocks), with $DE positioning an object just barely off screen. A narrow-width playfield is visible at $40-BF, a normal-width playfield $30-CF, and a wide playfield $2C-DF ($20-DD clipped on the left side).

Vertical positions are by scanline, with the maximum range being 8-247 (240 visible scanlines), and the center at 128. By convention, the standard normal-width Gr.0 screen covers 32-223 (192 visible scanlines). GTIA does not use vertical positions, but ANTIC does so when fetching P/M graphics for GTIA.

6.3 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.

The TV's color circuitry is synchronized to GTIA's color output by means of a color burst emitted during horizontal blank. The phase of this signal corresponds to a light yellow-orange color, sometimes called “gold.” GTIA then produces colors by emitting a color subcarrier with various phase shifts from the color burst, with all of the phase shifts evenly spaced. Hue 1 has one unit of phase delay, and each successive hue has additional multiples of delay added. The delay is nominally around 24°. Ascending hues, and therefore increasing delays, produce colors of 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 very little except due to display differences, and low-numbered hues have less variation between systems.

NTSC brightness encoding

The four-bit brightness value specified in color registers or produced by blending is converted to a brightness (luminance) signal with roughly equal steps, with 0 = black and 15 = white. There are two quirks in this conversion in hardware to be aware of, however.

The first quirk is that brightness levels 7 and 8 can be spaced closer together than the rest of the steps, almost identical.

The second quirk is that brightness level 0 is “blacker than black.” This is because of a difference between blanking level and black level in most NTSC varieties (except for NTSC-J), where blanking is at 0 IRE, black is at 7.5 IRE, and white is at 100 IRE. The Atari encodes brightness level 0 at blanking level, putting it at about -8% on a scale of black to white.

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 14: 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 related to the I-Q color encoding space used by NTSC by a flip and a 33° rotation. The 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.

6.4 Artifacting

Composite video encodes both brightness (luma) and color (chroma) together into the same signal. The decoder separates these imperfectly, which leads to the ability for the luma signal to produce colors when specific patterns are used. This is known as *artifacting* and can be used to deliberately encode colors without using the color hardware in the computer, particularly in hires modes that do not have the ability to directly produce colors from the playfield. The resulting colors depend strongly on the specific model of computer hardware.

Artifacting only occurs with a composite video or RF connection. With separate Y/C (S-Video) outputs, the luma and chroma signals are never combined and there is no artifacting.

NTSC composite signal encoding

The chroma signal for NTSC is encoded using quadrature amplitude modulation (QAM) at the color subcarrier frequency, 3.579545MHz. This results in a sine wave where the phase of the sine wave determines the hue and the amplitude determines the saturation. Lack of this signal therefore produces no color. In order to decode this signal, the receiver must have a phase reference, which is provided by a color burst of defined phase and color in horizontal blank.

Besides hue/saturation, there is another interpretation of the color signal, which the sum of two separate I and Q signals. I is the in-phase signal, while Q is the quadrature signal. Both are encoded as sine and cosine waves 90 degrees apart in phase, and the sum of the two is the color signal. The two representations are equivalent; I-Q is the Cartesian (X-Y) representation, while hue/saturation is the polar (phase/magnitude) representation.

Luma/chroma crosstalk

In order to recover the luma and chroma from the composite signal, the decoder must separate the two signals. This is done imperfectly and the result is crosstalk between the two signals. When part of the original luma signal is decoded as chroma, artifacted color results.

Some of this results from imperfect separation circuitry in the decoder, but there are situations where it is theoretically impossible to separate the two. The primary issue is if the luma signal has a component at the color subcarrier frequency, since signal theory says that such a signal can be decomposed into the sum of a sine wave at that frequency and higher frequency harmonics. This makes it impossible to distinguish the fundamental sine wave in the luma signal from the chroma subcarrier. Avoiding this conflict requires the encoder to exclude chroma-like signals from the luma signal before mixing the two. The computer does not do this, which allows for false colors.

The literature for standard NTSC has references to this effect being avoided due to the luma and chroma sequences interleaving in frequency space as offset combs. This is due to the line and fields rates being an odd number of color cycles, which causes the color subcarrier to invert phase on adjacent lines and fields. In old TVs where the color subcarrier is visible, this shows as a fine checkerboard slowly moving up. A comb filter can then take advantage of this by combining adjacent lines or fields to cancel the color subcarrier. Unfortunately, the computer's output deviates from this and generates a signal that is a whole number of color cycles per line and frame. The effect is that the color subcarrier doesn't invert phase between adjacent lines or frames, and the frequency combs of the luma and chroma signals overlap instead of interleaving.

Artifacted color hues

In order to produce stable artifacted colors, the phase relationship between luma and chroma must be known. The NTSC GTIA produces pixels using a pixel clock (dot clock) that is a multiple of the chroma subcarrier, thus producing stable hues. The highest resolution lores modes use a pixel clock at the chroma subcarrier frequency f_{sc} = 3.58MHz, too low to produce artifacted colors. The hires modes, however, use 2 x f_{sc} = 7.16MHz, which means that alternating pixel patterns produce a signal at the chroma subcarrier rate. Two hues are thus available for even and odd patterns.

The colors produced by the even and odd pattern vary, however. The colors produced by the two patterns will always have the same saturation and opposite hues, by virtue of being signal inverses of each other, and the average of the signal means that the brightness will be halfway (approx. luma 7). The hues, however, depend on the relative phase offset between the luma signal and the color burst produced by GTIA in horizontal blank. This varies for different computer models due to differences in the video output circuitry resulting in different delays between luma and chroma. The most common combinations produced are blue/green on 800s, purple/green on XLs, and red/blue on XEs. The GTIA color adjustment does not affect artifacted hues, however, since it doesn't affect color 1, which doubles as the color burst.

Note that while the hues for even and odd patterns are opposite in the signal, they are not necessarily exactly opposite when displayed. A major reason for this is that artifacted colors are often so highly saturated that they fall outside of the displayable color range (gamut) of the display, causing the hues to shift when the color values are clamped to range. For instance, it is impossible for a display to produce negative color. This can particularly lead to noticeable differences in the displayed artifacted color between a computer monitor and a TV. The brighter display of the TV can provide more headroom in RGB space than the monitor for the same signal, causing the monitor to clamp more than the TV. The result is the TV showing blue/green artifacting, while the monitor clamps those colors to purple/green.

Brightness and saturation of artifacted colors

How saturated artifacted colors depends on the relative strength of the 3.58MHz waveform in the luma signal relative to the color burst. For the luma signal, this depends on the difference in brightness between the even and odd pixels. For instance, alternating luma 0 and 4 gives about the same saturation as 10 and 14.[31]

The other factor is the strength of the color burst, which varies between computer models. A strong color burst gives weaker artifacting, while a weak color burst gives stronger artifacting. Out of three sampled NTSC computers, an 800 had the weakest color signal with color burst peak-to-peak of about 4 luma units, an 800XL about 8, and a 130XE about 14. Correspondingly, the 800 showed the most saturated artifacting colors.

The brightness of artifacted colors is more straightforward, as it is simply the average of the luma pixels – the result after the f_{sc} component has been separated out.

[31] The difference in signal strength between 0/4 and 10/14 isn't exactly the same in practice, since the luma steps are slightly uneven.

Mixing artifacted and non-artifacted colors

When conventionally generated colors are mixed with artifacted colors, the result is the additive blend of the two, since the luma and chroma signals are added together, causing their color signal components to also add. This addition occurs in the I-Q space that the chroma is encoded in. Since YIQ is related to YUV and RGB by linear transforms, this is also equivalently an addition in those color spaces.

PAL artifacting

It is also possible to produce artifacted colors in the PAL system, though the technique and interpretation is more complex. One of the main differences is the color subcarrier is at a higher frequency, 4.43MHz instead of 3.58MHz. This is a factor of 5/4 faster than the NTSC subcarrier and results in the subcarrier no longer matching the dot clock. Instead, 5 color cycles align with 4 lores pixels, giving a larger repeating pattern than the simple even/odd patterns with NTSC.

PAL alternation

The phase reversal of PAL on alternating scanlines also affects the colors produced through artifacting, since it also requires the pattern to be reversed on successive scanlines to produce a consistent hue. This produces a herringbone or interrupted checkerboard pattern. This pattern can be placed at four different offsets for four different hues.

Two additional hues are possible by deliberately not reversing the phase direction of the pattern and instead using vertical stripes as with NTSC. This causes the pattern to have opposite V axis values on adjacent scanlines, which are then cancelled by the receiver. The result is blue and yellow, the two colors on the U axis.

6.5 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 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.

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

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 15 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 15. 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.

6.6 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 and the self-collision bit for a player 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.

6.7 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[32]:

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.

[32] 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 PF01+P01+P23	P01	black	black	black	black	black	black	PF01	black	black	black
PF01+P23	P23	P23	PF01	P23	P23	PF01	P23	PF01	P23	PF01	P23
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 P5+PF23+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 P5+PF23+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+P23	P01	black	P5	black	P01	P01	P01	black	black	black	black

Table 16: 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 layer of all four missiles to that of playfield 3 (PF3), thus allowing them to be used as a fifth player. No other change to the missiles occurs – the missiles retain independent positions and sizes, and in order to be used as a “fifth 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.

Enabling fifth player mode also switches the priority of the missiles to that of playfield 3, except that 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.

6.8 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.

6.9 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.

These modes only work with a GTIA chip. On rare older devices with a CTIA, the top two bits of PRIOR are ignored.

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. The luminance in the COLBK register is normally set to zero as otherwise it is OR’ed with the playfield luminance values, reducing the total number of shades available.

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 resulting color value is the bitwise OR of the PF3 register and 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 hues 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 luminance 0. The hue of COLBK is normally set to 0, or else it will be OR’ed with the color values from the playfield, reducing the total number of colors available.

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 and hue OR’d with the playfield's hue, except if the playfield is %0000 in which case the luminance is forced to 0 regardless of PF3’s luminance.

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.

Horizontally scrolling GTIA modes

GTIA modes can be horizontally scrolled like any other mode, with the extra provision that only even values of HSCROL will work properly. This is because GTIA is unaware of ANTIC's horizontal scroll offset when it groups bit pairs to form 4-bit pixels. Odd values of HSCROL will produce a valid display but mix the low two bits of one pixel with the high two of bits of the next pixel to form new pixels instead of shifting the pixels, the same as if the bitmap data were shifted by two bits. The fat pixels in GTIA modes cannot be horizontally shifted by finer than two color clock precision by normal means.[33]

Mixing GTIA modes with low resolution modes

As previously noted, the GTIA modes only work properly with the high resolution modes 2, 3, and F. The reason for this has to do with the encoding of the data on the AN0-2 bus between ANTIC and GTIA. While the highest bandwidth modes have a raw data rate of two bits per color clock, the ANx bus actually has a bandwidth of three bits per color clock, with ANTIC doing additional encoding of the data based on the display mode. The difference in encoding between low-resolution and high-resolution modes prevents the GTIA modes from fully working when ANTIC is set to a low-resolution mode.

The AN0-2 bus encodings are given in Table 17.[34]

[33] It has been reported that certain GTIA chips can shift mode 10 by a color clock when running warm, but this is neither reliable nor program controllable.
[34] [AHS99a] p. 7

Encoding	Lores	Hires
000	Background	
001	Vertical sync	
010	Horizontal blank and switch to lores	
011	Horizontal blank and switch to hires	
100	PF0	%00 pixel pair
101	PF1	%01 pixel pair
110	PF2	%10 pixel pair
111	PF3	%11 pixel pair

Table 17: ANx bus encodings

In low resolution modes, one pixel is sent per color clock, and in high resolution modes, two pixels are sent per color clock. ANTIC signals the lores/hires state of the next scan line during horizontal blank and this then determines GTIA's interpretation of the %100-%111 codes.

Normally, the GTIA modes are used with ANTIC high resolution modes, in which the raw bitmap data is sent on AN0-1 two bits at a time. GTIA groups two pairs of bits at a time to form 4-bit pixels, which are then interpreted according to the GTIA mode. The high order bit AN2 is ignored, so the background encoding is not distinguished from a PF0 or %00 bit pair encoding and the border works differently in GTIA modes.

In low resolution modes, the mapping from pixels to ANx encodings is not straightforward and causes problems with GTIA modes. In particular, most four-color modes map BAK and PF0-PF2 instead of PF0-PF3. This makes the available AN0-1 encodings %00, %00, %01, and %10, with %11 not represented. The result is that attempting to use mode E, for instance, results in only 9 of the 16 pixel values being available with %11xx and %xx11 patterns being unattainable. ANTIC modes 4-7 can send the PF3 encoding, but are relatively inflexible in doing so.

6.10 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.

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 18: Timing for mid-screen writes to GTIA registers

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.

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.

6.11 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.

6.12 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 400/800 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[35], R is 1MΩ for CX30 paddles[36], 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

[35] [AHS03] p. 22 (V_{T+} positive-going threshold voltage)
[36] 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.

2.25V, this gives a scan line range of 1-441. Since the scan line counter only counts up to 228, this means that 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.

Fast pot scan mode

POKEY can be configured to read the paddle inputs in two scan lines instead of 228 scan lines by means of SKCTL bit 2, but this does not work with paddles for a couple of reasons. First, the paddles charge too slowly for fast pot scan mode. The capacitance, charge rate, and threshold are still the same, so the POTn registers are simply loaded with a value 114 times as large and the 0-229 value range covers only a tiny rotational range on the right end.

The second issue is that the POTn registers will not consistently load. Normally, dumping transistors are turned on momentarily when POTGO is strobed to empty the capacitors so they begin charging up voltage from empty. These are disabled in fast pot scan mode, and thus the scan can start with capacitors already partially or fully charged. If the capacitor is partially charged, it will reach threshold sooner and the recorded count will be lower than expected. If the capacitor is already charged above threshold, the POTn register will not be updated at all.

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 ($\overline{\text{LP}}$) 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. This mechanism also means that the PENH/V update occurs just after the video signal for the sensed point is generated and drawn on the CRT by the beam, occurring earlier for higher positions and later for lower positions.

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.

Note that the horizontal positions reported in PENH do not correspond to GTIA horizontal positions, for a couple of reasons. One is that it is latched from ANTIC’s horizontal position counter instead of GTIA’s, and another is delays in the display-to-light-pen-input path. PENH values are approximately 30-60 counts higher than GTIA positions, depending on the display and light sensing device. This offset often causes PENH values to wrap from $E3 to $00 within the active display region.

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

Note that it is possible for the PENH and PENV registers to have any value in 0-255, even those corresponding to invalid beam positions. This occurs because the pen position registers are not cleared on a reset and therefore may have arbitrary contents on power-up. In particular, PENV may have the value $FF.

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 pulse from a VBI routine. Otherwise, the triggering pulse can be very narrow and difficult to catch, as quick as 16 cycles.

7.5 CX-75 Light Pen

The Atari CX-75 light pen is a black device about the shape and size of a thick pen, with a pushable tip and a thin cord to connect to the joystick port.

The measurements here were determined experimentally on a Commodore 1702 monitor attached to an NTSC 800XL.

Pickup range and offset

On a Commodore 1702 monitor driven from an NTSC 800XL, a single lo-res $0F pixel at the center of the screen – hardware horizontal pos and vertical pos $80, position (80,80) on a GR.14 screen – produced PENH values around $96 to $9C, but always PENV values of $40. Pickup range was narrow side-to-side, in a circle about 4mm wide, but the light pen could detect the beam as far as 10cm away.

Pickup range

The light pen has a relatively narrow opening about 4mm wide for the light sensor to detect the beam. For a single $0E luma lo-res point, this results in the light pen only picking up the point within around an area of that diameter when the point is close to the screen, and still only within very narrow angles when held away from the screen. For a 13” monitor like the 1702, this already limits pickup to about 1-2 color clocks, and for a larger monitor it is even more precise.

The CX-75 can detect the beam from surprisingly far away, as far as 10cm with a full white pixel, though this range is somewhat diminished with dimmer pixels. On the test system, the light pen could not pick up a $02 background, and at $04, its range was only about 1.5cm. At $06, this increased to around 4cm.

Reported positions

The light pen is generally quite accurate and stable in vertical positions, but much less so horizontally. In the test environment, a white lores pixel in the center of the screen – hardware horizontal and vertical position $80, position (80,80) on a GR.14 screen – produced PENH values around $96 to $9C, but always PENV values of $40.

The offset in horizontal positions reported in PENH stems from several factors, including an offset between the internal horizontal position counter within ANTIC and the horizontal positions used by GTIA, and the delays from when pixels are output in the video signal by GTIA, displayed by the TV, sensed by the light pen, and finally light pen input latching delays in ANTIC.

Horizontal position jitter

It is normal for horizontal positions to vary by ±1 due to noise. However, on XL/XE systems, the CX-75 commonly displays much larger errors of as much as ~6 color clocks. Furthermore, the noise is not random, but position dependent – it is spiky but somewhat deterministic when the light pen is held steady.

The additional positional noise is due to ringing on the trigger line causing glitches on the ANTIC $\overline{\text{LP}}$ input. Figure 10 shows an example. CH1 (yellow) shows the raw trigger input from the light pen, CH3 (purple) shows the GTIA trigger input after the filter, and CH2 (cyan) shows the final light pen input to ANTIC. In this example, the glitches are approximately 0.7µs and 1.4µs after the original falling edge. At 280ns period color clock, this corresponds to offsets of +2.5 and +5.0 counts. Under normal circumstances, the last edge would always take precedence. However, the glitches are short enough that ANTIC’s triggering is unreliable, causing the PENH/V registers to update inconsistently depending on the offset of the glitches relative to the system clock. These glitches don’t occur on an 800 system where the input circuitry is more tolerant.

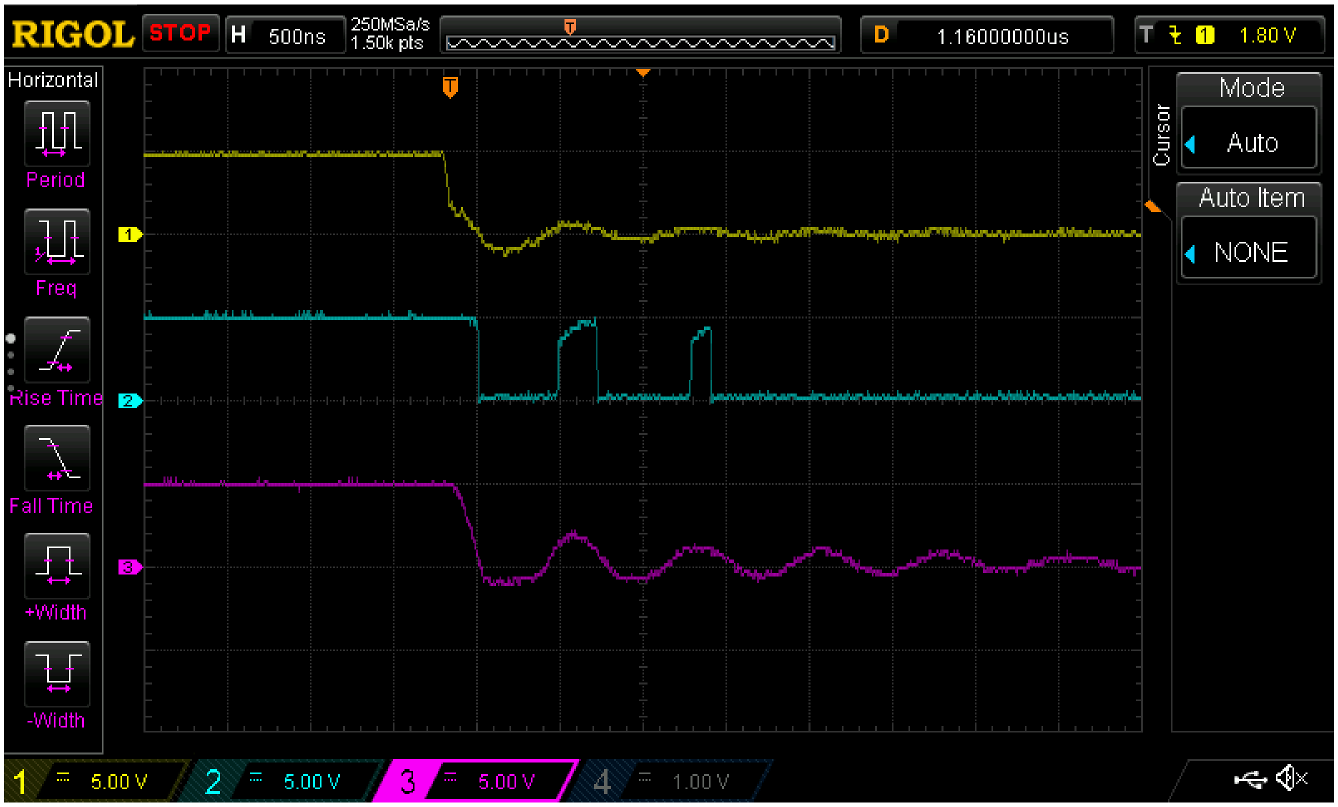


Figure 10: CX-75 triggering glitch on NTSC 800XL

Detected colors

For the light pen to detect the beam, the image must be bright enough for the beam to trip the light sensor. A full-bright screen is not necessary, but very dark areas of the screen can fail to trigger the sensor. On the test system, grayscale areas were picked up as dim as $04, but with low sensitivity/range, somewhat improved range at $06, and mostly full range at $08 and above. On the other hand, colors were picked up more readily, as low as luma 2. One theory for this is that colors cause one of the three color beams to be brighter than the average, which is then more easily picked up by the light pen.

Note that the exact thresholds are affected by the brightness and contrast settings of the display. Raising the brightness or contrast will make darker colors easier to pick up. If the brightness is raised enough that the black level rises, even black ($00) will work.

Trigger timing

As the beam is sensed by the light sensor, the light pen asserts the joystick trigger signal on the controller port for a short period of time. For the CX-75, the width of this pulse is mostly uniform, varying from about 9.1-9.8 µs depending on intensity, or about 32-35 color clocks. This is always enough to trigger ANTIC's light pen input and update the PENH and PENV registers, but is difficult to detect reliably with the 6502. It can be detected by a tight polling loop, but programs that need to detect when the light pen is triggering usually need to enable trigger latching on GTIA.

Measured delays related to light pen timing include:

- ~800 ns from the rising/falling edge of AN2 between GTIA/ANTIC to the corresponding change in the composite video output.

- Very approximately 9-13 µs from the start of the pixel in the video output to the leading edge of the joystick trigger pulse from the light pen.

Pen switch

The CX-75 also has a switch activated by pushing in the noise of the pen at the front. The switch is somewhat loose and easily pushed with its weak spring. Pressing in the tip asserts the joystick up signal, changing bit 0 or 4 of PORTA/B from 1 to 0 while the tip is depressed.

An oddify of the pen switch is that the a trigger pulse is also sent when the pen switch is depressed or released. This appears to be due to bounce noise from the switch and can result in multiple trigger pulses over hundreds of microseconds, especially on switch activation. This results in random positions being latched into the PENH/PENV registers, which must be ignored.

7.6 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 19: 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.

7.7 CX-20 Driving Controller

The CX-20 Driving Controller is similar to a paddle controller except there is only one connected per port, the rotary knob can be rotated indefinitely in either direction, and the resolution is much lower.

The trigger of the driving controller is connected the same way as a joystick trigger and is reflected in TRIG0-3. Rotations are transmitted over the up/down joystick lines as a two-bit Gray code which changes once for every sixteenth of a rotation and cycles every quarter rotation. For a left rotation, the bits 1/0 or 5/4 in PORTA/PORTB step 00-01-11-10-00, and for a right rotation, they step 00-10-11-01-00. This is similar to the X axis of an ST mouse, but at a much lower rate.

7.8 CX-21/23/50 Keyboard Controller

The CX-21, CX-23, and CX-50 keyboard controllers have different physical shapes but the same key layout and signal interface. They connect a four row, three column keypad to a joystick port using a combination of the joystick and paddle lines.

Signals

The four rows of the keyboard matrix are connected to the joystick lines and the three columns are connected to the paddle and trigger lines:

1	**2**	**3**	Up
4	**5**	**6**	Down
7	**8**	**9**	Left
*****	**0**	**#**	Right
Paddle B	Paddle A	Trigger	

Table 20: Keyboard Controller key matrix

To read the matrix, the four joystick direction signals are configured as outputs and brought low one at a time. The two paddle inputs and trigger are then read to sense the columns. The trigger input will be sensed as a 1 if its column is inactive and 0 if it is active; paddle inputs will tend to read ~1-2 for inactive and ~228 for active.

Fast pot scan mode

Unlike paddles, which only pull up the paddle lines, the keyboard controller also grounds them. This means that fast pot scan mode can be directly used with the Keyboard Controller without having to discharge the capacitors with slow scan. After enabling fast scan, the paddle inputs can be read directly by strobing POTGO and then reading ALLPOT within 228 cycles. This permits reading the Keyboard Controller much faster than once every 3-4 frames.

The POT0-7 registers are still not valid in this configuration by default since they do not update if the capacitors are already above charging threshold when the scan starts. They will read $E5 if a column is activated, but may retain their previous value otherwise. This limitation can be avoided by strobing POTGO in slow scan and then dynamically switching to fast scan, but this is inferior to using ALLPOT.

Multiple key presses

As is usually the case for a keyboard matrix without diodes, pressing multiple keys on a Keyboard Controller can cause phantom keys to appear in the matrix. Two keys can always be resolved independently without phantoms; three or more will cause phantoms if they occupy a shared row and column. For instance, pressing 1, 2, and 4 is indistinguishable from also having 5 pressed.

Pressing multiple keys also has the effect of reducing the paddle position values reported by POKEY in the POT0-7 registers for non-activated keys. The effect is hardly noticeable in normal pot scan mode where the counts drop from 2-3 to 1-2, but is more noticeable in fast pot mode where readings can decline from around $B0 to as low as $60. This occurs because depressed keys on inactive rows supply additional current from high row lines to the paddle inputs on active rows through the column lines. This will occur independently for each column unless the columns are connected by two keys depressed on both paddle columns of the same row.

7.9 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.[37] 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.

[37] [ATA87] p.11

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.

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.

End of line anomaly

Because of the use of EOLs in the frame buffer to track logical lines, the XEP80 can track spaces at the end of a logical line, unlike the standard E: driver. Compounding this issue is that the standard XEP80 E: handler will return these spaces and the XEP80 firmware will replace EOLs with spaces when moving the cursor for the move right ($1F), backspace ($7A), and tab ($7F) special characters. This can result in unwanted effects like splicing a large number of spaces at the end of a DATA statement in a BASIC program. EOLs cannot be reinserted into a logical line, so this can only be fixed by deleting spaces and shortening the line.

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.[38]

Video timing

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.

Another issue with the XEP80's video output is that it is significantly out of spec. The horizontal scan rate is 16.3KHz in 60Hz text mode and 16.1KHz in 60Hz graphics mode. The number of scan lines is also off, with the XEP80 producing 272 scan lines in 60Hz text, 269 scan lines in 60Hz graphics, 327 scan lines in 50Hz text, 323 scan lines in 50Hz graphics. The vertical scan rate is generally much better, but the poor horizontal scan rate and incorrect scan line count prevent some modern TVs from locking onto the XEP80's display and will show a vertically rolling picture. It is possible to reprogram the XEP80 for closer 60Hz timings at least by use of undocumented commands (50Hz timings are more awkward given the NS405's limitations).

[38] [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.

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.

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.[39] 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

[39] 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 character names and so the split between the latches is the more normal $00-7F / $80-FF.

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.

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 21: 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.

Pixel graphics

Command $99 switches the XEP80 from text mode to pixel graphics. In pixel graphics mode, all 8K of on-board memory is dedicated to a 320x200 monochrome display, similar to GRAPHICS 8. The pixel aspect ratio, however, is much narrower than a GR.8 display, as the pixels are output with a dot clock of 12MHz instead of 7.14MHz. This produces a narrower than square display on NTSC.

In graphics mode, the firmware writes normal bytes received directly to the cursor location, ignoring all normal special character processing logic. The cursor address is automatically incremented, but not wrapped within the display, so it cycles through all VRAM including non-displayed memory. Since the NS405 displays bits LSB-first instead of MSB-first as ANTIC does, the XEP80 firmware reverses the order of bits when writing bytes into display memory in graphics mode.

Internally, the graphics display is stored in on-board memory as contiguous rows at $0000-1F3F. However, the external character set ROMs must be bypassed, so the NS405 is set to address $4000-5F3F. If the display pointers are modified to within $0000-3FFF or $8000-BFFF, the display data will be translated through one of the ATASCII character generators, using a row height of 10 scan lines. However, due to the bit reversal done during writes, this requires sending the XEP80 bit reversed ATASCII so that the firmware un-reverses it to the correct values on write. Also, for some reason, the row counter is incremented after the fifth byte in each scan line instead of at the end.

The standard cursor positioning commands can be used in graphics mode, but since the text row pointers are not reinitialized by the command to enter graphics mode, vertical positioning is indeterminate unless a master reset is issued first. The standard cursor commands also only allow positioning up to $18FF. An alternative is the undocumented $E2 command, but that requires writing two bytes to the cursor address. One workaround is to modify BEGD/ENDD/HOME to warp the display around the status row location at $1800-18FF and then use the $98 command to move the cursor to the status row before issuing commands that require the last character or extra byte values.

Pixel graphics mode oddities

The cursor is still active in pixel graphics mode, although it is only a single scan line tall. Cursor blinking is disabled when pixel graphics mode is entered, but the cursor on/off state is preserved. There is a bug in the XEP80 that corrupts the display if the Cursor On or Cursor On, Blinking commands are issued in pixel graphics mode, due to resetting VCR back to text mode values.

The attribute latches are still also valid in pixel graphics. Bit 7 of display memory, or bit 0 in data written from the computer, selects the attribute latch (1 = latch B). There is no $9B anomaly. Reverse video, blink, double width, underline, and blank attributes are active, while half intensity, double height, and graphics attributes are ignored.[40] The Set Graphics command ($99) does not reset the attribute latches, so they must be manually reset back to $FF for proper pixel graphics display if they have been previously modified, unless attribute activation is actually desired.

Hardware scrolling

In text mode, the LSB of the address for each display row other than the status row is specified by command $DC. This scrolls the display horizontally without requiring copying of display data, and is thus fast and non-destructive. Similarly, the firmware uses the NS405's table lookup mode to quickly scroll vertically, since it allows doing so by swapping row address pointers.

Pixel graphics mode supports both horizontal and vertical scrolling, although doing so requires several undocumented firmware commands. The BEGD and ENDD registers can be modified to specify a wrapping region, and the HOME register to set a starting address within that region. This allows horizontal scrolling on a byte basis and vertical scrolling on a scan line basis. Although the XEP80 only has 8K of display memory, the address registers on the NS405 are full 16 bit; the display pointer will wrap around the 8K memory and cross the

[40] The NS405 manual says that attribute-based reverse video is disabled in pixel graphics mode, but this appears to be untrue for the XEP80's pixel graphics configuration – resetting bit 0 of either attribute latch will invert bytes mapped to that latch, same as in text mode.

boundaries for the different external character set generator modes specified by A13 and A14.

Unlike text mode, where the status row is a single row, in pixel graphics mode the status row is extended to a non-scrollable region of arbitrary row height. Timing chain register 8 specifies the row on which this region begins, with a starting address specified by SROW, after which rows are displayed sequentially.

> **Warning**
>
> There appears to be a bug in the NS405 with the non-scrollable region such that the display address is reset to SROW at the end of each scan line in the row specified by TC[8] instead of only at the end of the last scan line. The result is that with a character height of 10 scan lines, scan lines 1-9 of that row are repeated versions of the beginning scan line of the non-scrollable region pointed to by SROW.

Baud rate modification

It is possible to reprogram the XEP80 to run at a different baud rate than the default 15.6 Kbaud rate by means of the undocumented $FA-FC commands, which alter the UART parameters. The most useful combinations are asymmetrical 31.5Kbaud transmit / 15.6 Kbaud receive and symmetrical 31.5K transmit/receive.

The main limitation on the usable baud rate is the slew rate of the joystick control lines, affected greatly by RLC network protection circuits on the joystick ports. PIA port A uses single-ended drivers and this results in slow 0->1 transitions; the drive capability varies by chip and some variants of the 6521 are not able to transition to 1 fast enough with regular transmit timing for the XEP80 to see a 1 bit after a 0 bit. This can be fixed with precompensation to emit 1 bits earlier to give the output enough time to rise above threshold. However, the generally slow rise time makes faster speeds like 63.5KHz impossible.

Horizontal position and row advance anomaly

By default, the XEP80 firmware programs the NS405 to emit the leading edge of horizontal sync close to the start of horizontal blank, about 3.4 characters (2.0μs) after the last displayed character. This places the 80-character display about five characters right of ideal centered position, resulting in some columns on the right being cut off on some displays. The NS405 timing chain can be reprogrammed to adjust the horizontal sync position and recenter the display. However, hardware design issues in the NS405 and XEP80 cause an artifact that prevent this from working.

The NS405 requires an external row counter when an external character set is used, since it does not output its own internal row counter. A scan line reset signal is emitted to reset this counter, and it is incremented every horizontal sync to advanced to the next row within the character cell. Normally, this happens within a couple of character cells of horizontal blank, cleanly between rows.

When horizontal sync is moved later in HBLANK, however, an artifact appears where the first few characters are displayed one character row lower than intended. The reason for this is a four-byte fetch FIFO within the NS405 that begins prefetching a few character cells after horizontal blank begins. With the default HSBR setting of $52, this occurs after the row counter has incremented, but raising HSBR results in the FIFO being able to fetch some data through the character ROM before HSYNC starts and the row counter increments. This happens when the HSBR > HBR + 3. The last affected character flickers due to one-character variance in FIFO timing.

It is possible to work around this issue raising HBR to extend the displayed area and delay the start of horizontal blank until closer to horizontal sync. The additional displayed characters are usually blank unless horizonal scrolling has been used, which is uncommon.

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.

Delete line ($9C)

Deletes the logical line starting at the current vertical position. For instance, if the current logical line is three lines tall, three physical lines will be deleted.

The XEP80's behavior differs from the standard E: handler when the cursor is in the middle of a logical line. The standard E: handler will delete all physical lines comprising the logical line that the cursor is in, while the XEP80 will only delete physical lines starting from the current line. When the delete line command is sent when the cursor is in second physical line of a three line tall logical line, the XEP80 will only delete the second and third physical lines. This leaves the first line behind and also splices it with the next logical line.

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

All commands are sent with bit 8 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. The cursor is reset to the top-left byte in the display, cursor blinking is turned off, character blink mode is set to foreground only, and the display polarity is set to white-on-black.

This command sets HOME=$4000, BEGD=$4000, ENDD=$FFFF, and CURS=$4000.

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. This is the XEP80's equivalent to the E: handler's DSPFLG.

The list flag is ignored in pixel graphics mode.

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.

Warning

The Cursor On ($D9) and Cursor On, Blinking ($DA) commands use a shadow of the Video Control Register (VCR) that is not updated by the command to switch to pixel graphics. As a result, using either command will switch the NS405 out of pixel graphics mode back to text mode, corrupting the display. This can be fixed by rewriting the correct value into the VCR or turning the cursor on/off prior to enabling pixel graphics.

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. The left margin is ignored, so the cursor ends up at column 0 on screen regardless of the left margin setting.

This command has no effect in pixel graphics mode.

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).

Set Cursor Address ($E2) (undocumented)

Modifies the NS405 CURS register, which contains the cursor address, to have the high byte from the extra byte and the low byte from the extra byte. This moves the cursor on screen and also sets the next address used by cursor-driven operations.

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.

Set Display Home Address ($E7) (undocumented)

Sets the value of the NS405 HOME register. This register determines the starting address of the graphics screen at the top-left corner of the display. It is normally set to $4000 by the firmware.

In text mode, this register is regularly overwritten by interrupt at the end of each row, and so this command has no lasting effect.

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 Begin Display Address ($EF) (undocumented)

Sets the NS405 BEGD register, which determines the beginning address of the scrollable region in graphics mode. The display wraps back to this address after reaching the value of ENDD.

The BEGD register is not active in text mode, during which the firmware uses table lookup mode.

Set End Display Address ($F1) (undocumented)

Sets the NS405 ENDD register, which determines the ending address of the scrollable region in graphics mode. When ENDD is reached, the display wraps to the address specified in BEGD. Note that ENDD must be one past the last byte in the ending scan line before the wrap.

The display address must *exactly* match ENDD at the end of a scan line to wrap back to BEGD. It is possible to wrap around from ENDD to BEGD in the middle of a row, but not in the middle of a scan line. If the display address advances past ENDD during a scan line, the display will fail to wrap and continue displaying rows sequentially past ENDD.

The ENDD register is not active in text mode, during which the firmware uses table lookup mode.

Set Status Row Address ($F3) (undocumented)

Sets the NS405 SROW register, which determines the starting address of the status row. This is normally set to $1800, $3800, or $5800 by the firmware depending on the current character set. The vertical location of the status row is determined by timing chain register 8.

In text mode, the status row's address is determined by SROW instead of HOME, and therefore that single row is not affected by horizontal scrolling. In graphics mode, the status row defines the start of a non-scrollable vertical area that begins at the address specified by SROW and extends to the bottom of the display.

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.

Note that while the attribute latches are still active with pixel graphics, they are not reset by the command to set pixel graphics mode.

While the $F4/F5 commands are not listed in the XEP80.DOC file included on the XEP80 handler disk, they are officially documented in the ATRIBUTE.BAS demo program.

Set Timing Control Pointer ($F6) (undocumented)

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) (undocumented)

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. Table 22 gives the values applied to the timing chain by the firmware for various modes.

Register		60Hz Text	50Hz Text	60Hz Gfx	50Hz Gfx
0	Horizontal total	68		5C	
1	Horizontal display length	4F		27	
2	Horizontal sync start	52		3B	
3	Horizontal sync end	5F		46	
4	Char height + extra scans	91	B2	98	BA
5	Vertical total	1A		19	
6	Vertical display height	18		13	10
7	Vertical sync	02		79	
8	Status row	17			
9	Blink rate + duty cycle	F4			
10	Block graphics horiz.	30			
11	Block graphics vert.	36			
12	Underline	89			
13	Cursor	0F			

Table 22: XEP80 Timing Register Values

Warning

The NS405 documentation says to set HSBR and HSER to the character position after which horizontal sync is to begin and end, + 2. In reality, the beginning of horizontal sync occurs about 2.5 characters later than expected, while end of horizontal sync is close to expected. Thus, setting HSBR/HSER as documented results in horizontal sync starting ~2.5 characters late and 2.5 characters shorter than expected.

With character cell numbers starting at 0 for the leftmost displayed column, HSBR should be set to the character cell in which horizontal sync should start in the middle of, and HSER to the character cell at which horizontal sync should stop at the beginning of + 2. The default values of HBR=$4F, HSBR=$52, and HSER=$5F result in a horizontal sync pulse that starts 2.5 character cells after the end of the last displayed character and is 10.5 character cells long.

Set Vertical Interrupt Register ($F8) (undocumented)

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) (undocumented)

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) (undocumented)

Sets the NS405 UCR register to the value of the last character. The UCR is constantly rewritten by the serial I/O routines in the XEP80 firmware, so this command is ineffective.

Set UART Multiplex Register ($FC) (undocumented)

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) (undocumented)

Sets the NS405 XMTR register to the value of the last character, retransmitting that character back to the computer.

Strobe Printer ($FF) (undocumented)

Sends the printer a strobe indicating a new byte is available, without actually setting a new byte.

7.10 Corvus Disk System

The Corvus Disk System, made by Corvus Systems, was a multi-platform system for adding external hard disk storage to a computer. For the Atari 8-bit computers, the Corvus was connected to the computer through an Interface box that connected to joystick ports 3 and 4 on the 400/800.

Joystick port usage

The standard Corvus Interface is a bidirectional, byte-wide parallel port with additional control signals. Since combining two joystick ports only gives 8 bidirectional data lines total via the joystick direction lines connected to the PIA, the Interface provides a nibble-wide data interface to the computer and adapts it to the byte-wise interface of the Corvus.

Joystick port 3 (PORTB bits 0-3) is used as the bidirectional half-duplex data path, while joystick port 4 (PORTB bits 4-7) is configured as output-only for control signals. The control signals are used to drive the Interface as follows:

PORTB bits 4-6	Command
000	Write lower nibble
001	Write upper nibble and send byte to drive
010	Receive byte from drive and read lower nibble
011	Read upper nibble
100	Read interface status

Table 23: Corvus Interface modes

Bit 7 is used as a strobe, with commands taking effect on the falling edge of bit 7.

Reading interface status

Issuing the %100 command causes the Interface to place bus status onto the nibble bus, reflected as bits 0-1 of PORTB. Bit 0 indicates ready status, where 0 indicates that the device is busy and 1 indicates that it is ready to transfer data. Bit 1 indicates the direction of the bus for the current pending transfer, where 0 means computer-to-host and 1 means host-to-computer.

Command set

All communication with the disk system is initiated by the computer via commands. All commands follow the same form, with a command packet being sent from the computer, followed by a status code and return data from the device. The disk is exposed to the computer as a linear array of 512 byte blocks, which can be read or written as 128, 256, 512, or 1024 byte sectors via multiple types of read/write sector commands.

Details of the protocol and command set are too large to include here. Fortunately, Corvus Systems published complete documentation for both in *Corvus Mass Storage Systems General Technical Information* [CorvusGTI].

7.11 ComputerEyes Video Acquisition System

The ComputerEyes Video Acquisition System by Digital Vision, Inc. is a hardware device for capturing still video frames from a composite video signal. It connects to the computer through joystick ports 1 and 2, which are used to read the video signal.

Sampling mechanism

A video frame contains far too much data for the computer to record directly, so the frame is sampled over multiple seconds. Specifically, the ComputerEyes device samples one pixel per scanline and gradually scans across the screen at one column per frame. Thus, a 320x192 bi-level image takes 320 ÷ 59.94 = 5.3 seconds to capture. This relies on the frame being relatively stable across the capture process; if it is changing, the captured image will be a mix of all of the frames.

Only a bi-level image can be captured in a single pass. To capture a multi-level image, successive passes are required with different thresholds. The highest depth image that can be captured is a 16-level image, which takes 85 seconds to capture.

Communication protocol

Communication with the ComputerEyes device is through the joystick direction lines of control ports 1 and 2. Five output lines and two input lines are used. Table 24 shows the usage.

Port	Input	PORTA bit	Direction	Usage	Polarity
Port 1	Right	D7	Input	Sync sense	0 = sync
	Left	D6	Input	Video level sense	1 = above threshold
	Down	D5	Output	Sweep control	1 = run sweep
	Up	D4	N/A		
Port 2	Right	D3	Output	Threshold bit 3	1 = higher
	Left	D2	Output	Threshold bit 2	
	Down	D1	Output	Threshold bit 1	
	Up	D0	Output	Threshold bit 0	

Table 24: ComputerEyes controller port usage

Video timing synchronization

Video sync status is sensed through PORTA bit 7, which is pulled low by the ComputerEyes whenever sync level is detected. Horizontal and vertical sync status are combined and must be distinguished in software by the length of the sync pulses.

Because horizontal sync pulses are too short for the 6502 to detect reliably – 4.7µs, or approx. 8.4 cycles – the sync pulses are stretched by the device. The minimum sensed timing is unknown, but probably around 10-15µs. It needs to be at least enough to be reliably sensed by a BIT+Bcc loop in the presence of refresh cycles.

Capture sweep

PORTA bit 5 is set by the software to begin a capture sweep. This is normally done after detecting the vertical sync to start at a known scanline. When the sweep is enabled, the CaptureEyes device begins sampling the video signal once per scanline, comparing the brightness of the image at that point against the threshold. The result of this comparison is then held until the next sample and reported in PORTA bit 6.

Unlike the Apple II version of the ComputerEyes, which uses a synchronous 65 cycle/line loop to read the samples, the Atari version instead asynchronously samples by waiting for horizontal sync through PORTA bit 7 before reading each sample.

The ComputerEyes device automatically controls the delay timing from the start of horizontal sync to each sample, slowly increasing the delay over time. Thus, sweep timing/speed is not controllable by software, though the software can determine horizontal and vertical offset by varying when it starts recording samples. The software also determines the captured image size, which can be smaller or larger than standard 320x192.

The increase in sampling delay is continuously increased over time and not stepped on a frame basis. Thus, there is a slight skew in the captured image as lower pixels in a column are sampled a fraction of a pixel later than ones higher in the column.

When the sweep is stopped, both the sync and video bits in PORTA bits 7-6 are forced to 0.

Threshold control

The threshold used to convert video levels to bi-level samples is determined in two ways. One is a manual adjustment knob on the device. The other is PORTA bits 0-3, which select 16 different thresholds relative to the manual adjustment, with successively higher values thresholding at brighter levels. Thus, by sampling multiple passes with 1, 3, 7, or 15 levels, the software can capture 2, 4, 8, and 16 level images. These are full resolution images, which must be reduced in software by dithering or quantization to display through ANTIC+GTIA.

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. This reflects the physical state of the device interface, not bits 0 and 1 of $D50E. When no CF card is present, the interface is automatically powered down and bits 5-7 read 0.

The canonical address for this register is $D50E.

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 reset control / 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.

A rising edge on bit 1 is required to power up the device. This is significant if bit 1 is set when a CF card is removed and then reinserted. If bit 1 is already set when this happens, writing $02 again will neither power on the device nor light up the green power LED; it is necessary to clear and then set bit 1 again.

Similarly, a rising edge on bit 0 is necessary to bring the device out of reset once it has been powered, and if bit 0 is already set it must be reset first. The CF device cannot be both powered on and brought out of reset in the same write. This means that the normal sequence to initialize the device is $00 > $02 > $03. An attempt to write $00 and then $03 will result in the device being powered on but still held in reset state, and further writes of $03 will have no effect since a rising edge on bit 0 is required, i.e. $00 > $03 > $02 > $03 would be needed.

Once the device has been powered up and brought out of reset, toggling bit 0 switches between the main and alternate register sets. Bit 1 must be kept high to keep the device powered.

Power and reset states cannot be modified when no CompactFlash card is present, and the CF interface is always reported as powered down and in reset state in this case. However, bits 0 and 1 can still be written and their state is still tracked, which is why it is necessary to write $00 before attempting to power up the CF device.

$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 25 shows the register layout. Grayed out entries indicate locations not handled by SIDE and ignored for reads or writes.

	SIDE 1		**SIDE 2**	
Address	**Read**	**Write**	**Read**	**Write**
D5E0	▨	SDX bank	▨	▨
D5E1	▨	▨	SDX bank	SDX bank
D5E2	SPI sense	SPI control	SPI sense	SPI control
	▨	▨	▨	▨
D5E4	▨	Cart bank	Cart bank	Cart bank
	▨	▨	▨	▨
D5F0 D5F7	CF registers	CF registers	CF registers	CF registers
D5F8	▨	CF reset	Signature	CF reset
D5F9	▨	CF reset	Chg. sense	Chg. reset
D5FB	▨	CF reset	▨	▨
D5FC D5FF	Signature	▨	Signature	▨

Table 25: 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 CompactFlash hardware registers (read/write)

Parallel ATA register set, as exposed by the CompactFlash device.

$D5F8-D5FB CompactFlash 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. Also, bit 7 only controls $A000-BFFF, and it is possible for that window to be disabled while $8000-9FFF is still enabled through bit 6.

$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.

$D5F8 CompactFlash reset register (SIDE 2 only, write only)

Controls /RESET on CompactFlash card via bit 0, same as $D5F8 on SIDE 1. The difference is that SIDE 2 only responds to $D5F8 for this and not $D5F8-D5FB as on SIDE 1.

$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, if a CF card is inserted. If no CF card is inserted, the CF change latch cannot be reset.

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.

8.6 SIDE 3

SIDE 3 is an updated cartridge design adding both cartridge storage and mass storage to the Atari as in earlier SIDE cartridges, reworked to make use of Secure Digital (SD) storage instead of CompactFlash. It also adds temporary RAM storage, real-time clock, and DMA facilities.

Special thanks to Sebastian Bartkowicz (Candle'O'Sin) for providing a SIDE 3 cartridge for testing, and to him and Jonathan Halliday (flashjazzcat) for answering questions about the SIDE 3 hardware and firmware.

Secure Digital (SD) storage

A Secure Digital (SD) slot provides mass storage in SIDE 3. It is a full-size SD slot connected via Serial Peripheral Interface (SPI), which is an alternate operating mode for SD devices. The host computer is responsible for implementing the SD command protocol to communicate with the device, although the cartridge assists with a shift register, command CRC computation, and DMA transfers when reading from the device.

Flash storage

Persistent, rapid-access storage is provided by 8MB (64Mbit) of flash memory. Currently, the initial production runs are using a Macronix MX29LV640ET NOR flash chip, which primarily has a 64K block size with a boot/configuration area of 8 x 8K blocks at the top 64K of the addressing range.

Because the MX29LV640E is a word-oriented chip with a byte interface mode, its ID bytes in Read Mode are doubled up and the device ID is at chip address $000002 instead of $000001.

RAM

SIDE 3 also contains 2MB (16Mbit) of RAM for temporary storage and cartridge emulation. It can be mapped directly to the cartridge windows, and is also used for buffering DMA operations.

DMA engine

The DMA engine accelerates reads from SD to memory and memory-to-memory copies for up to 64K bytes. Variable address stepping and AND/XOR masks are also available, allowing memory operations other than block copies to be accelerated. However, it is not able to copy from flash or perform two-operand binary operations, which limits it from the functionality of a full blitter. DMA is fully interleaved with host accesses, allowing SD reads to run at 1 cycle/byte and memory copies at 2 cycles/byte.

There is no support for DMA write transfers to the SD card.

Activity LEDs

Two LEDs on the cartridge indicate activity to the user. One is a green LED that is fully under software control including enable/disable and variable brightness; the other one is a red LED that automatically activates whenever an SPI transfer is in progress. Both the red and green LEDs may be on simultaneously, though the green LED is considerably brighter.

The green LED is active whenever manually enabled ($D5FC bit 5) XOR SD power (primary set $D5F3 bit 5); it is off if both bits are set. The brightness is controlled by primary set $D5F0. There is no brightness control or direct toggle for the red LED.

Push button

A small push button at the top of the cartridge allows for direct user control without going through the computer. By default, it acts as a cartridge reset button, restoring it to the default power-up banking configuration. It can also be reconfigured as a pure signalling button, detectable by software but not modifying cartridge state.

The push button is also used to invoke recovery mode by holding the button during power-on. This inhibits the cartridge windows on startup, allowing reflashing code to be run when flashed image crashes on boot.

SpartaDOS X switch

A two-position switch on the front selects the default banking configuration of the cartridge, initializing $A000-BFFF to either flash bank $000 or $3C0. In the standard software configuration, this selects between the SIDE Loader at $000 or the SIDE 3 version of SpartaDOS X at $3C0. The switch also enables or disables the emulated cartridge enable at $D5FC bit 2, which is used to emulate the SpartaDOS X pass-through.

The switch is also readable in software, via $D5FC bit 6. Changes to the switch position are immediately reflected.

Banking mechanisms

SIDE 3 has both native and emulated cartridge modes. In the native mode, it has two independently controllable windows, window A for the right slot ($8000-9FFF) and window B for the left slot ($A000-BFFF). Either window can be disabled or mapped to either RAM or flash on an 8K boundary. RAM windows can also be made read-only, in which case they do not modify cartridge RAM but still block writes to main RAM.

The cartridge emulation engine is more flexible and can emulate many popular cartridge types through a set of configuration parameters that map CCTL writes to the banking configuration. This includes standard 8K/16K, XEGS, and MegaCart cartridge types.

Real-time clock

Time services and non-volatile configuration storage are provided by a Microchip MCP7951X Battery-Backed SPI Real-Time Clock/Calendar chip. It is accessed through the same SPI interface as the SD card, allowing for convenient byte access from the software side without the need to bit-bang the SPI protocol. 64 bytes of battery-backed SRAM and 272 bytes of EEPROM are available for arbitrary data storage.

Register interface

SIDE 3 is controlled by 16 addresses at the top of the CCTL region, at $D5F0-D5FF. The bottom twelve locations at $D5F0-D5FB are bank-switched between three different register sets while the top four locations at $D5FC-D5FF are common. The majority of register bits are read/write, avoiding the need for register shadowing.

Address	**Register set**			
	Primary (0)	**DMA (1)**	**Emulation (2)**	**Reserved (3)**
$D5F0	Green LED brightness	Source addr 20:16 + ctl	CCTL base	Read as $00 Writes ignored
$D5F1	Open bus	Source addr 15:8	CCTL mask	
$D5F2		Source addr 7:0	Address mask	
$D5F3	SD/SPI control	Dest addr 20:16	Data mask	
$D5F4	SPI data	Dest addr 15:8	Window A disable mask	
$D5F5	CRC-7	Dest addr 7:0	Window B disable mask	
$D5F6	Window A RAM bank	Length 7:0	Feature control	
$D5F7	Window B RAM bank	Length 15:8	Banking control	
$D5F8	Window A flash bank 7:0	Source step	Window A initial bank	
$D5F9	Window B flash bank 7:0	Dest step	Window B initial bank	
$D5FA	Window control	AND mask	Window disable invert	
$D5FB	Window write protect	XOR mask	Emulation mode	
$D5FC	Common control 1			
$D5FD	Signature (read); common control 2 (write)			
$D5FE	Signature (read)			
$D5FF	Signature (read)			

Table 26: SIDE 3 registers

The entire register file can be locked for writes by setting emulation $D5F7 bit 4, directing those writes to the cartridge emulation engine instead. This does not affect reads, as the register file can always be read regardless of the lock state.

Push-button reset

When the push button is used with reset enabled, the cartridge is reset to boot-up state. Table 27 lists which registers are changed and their initial values. The initial bank for window B depends on the state of the SpartaDOS X switch. In addition to disabling cartridge emulation, releasing the register write lock, and resetting the banking registers, the register bank is reset to the primary bank, DMA is stopped, and some of the DMA registers are reset. However, the flash bank for window A is not reset.

Address	Register set			
	Primary (0)	**DMA (1)**	**Emulation (2)**	**Reserved (3)**
$D5F0		(bit 7 cleared)		
$D5F1				
$D5F2				
$D5F3				
$D5F4				
$D5F5				
$D5F6	$00			
$D5F7	$00		(bits 4 and 7 cleared)	
$D5F8		$01		
$D5F9	$00 or $C0	$01		
$D5FA	$38 or $F8	$FF		
$D5FB	(bits 0 and 1 set)	$00		
$D5FC	(bits 0 and 1 cleared; bit 7 set)			
$D5FD				
$D5FE				
$D5FF				

Table 27: SIDE 3 registers changed by push button reset

If recovery mode has been enabled by holding down the push button when the computer is powered on, window B is instead disabled on power-up/reset.

Common register set

Common control 1 ($D5FC; read/write)

7						0
BTN	SDX	LED	CLD	BM	CAR	REG

D7 Button status

0	Button hasn't been pushed (default)
1	Button has been pushed

D6 SpartaDOS X switch status

0	Off (up position)
1	On (down position)

D5 Green LED mode

0	Light LED when SD power is on (default)
1	Light LED when SD power is off

D4 Cold start flag

0	Warm start
1	Cold start (default)

D3 Button mode

0	Button triggers cartridge reset (default)
1	Button is used for software control

D2 Emulated cartridge enable

0	Cartridge emulation window control disabled (SDX mode only) (default)
1	Cartridge emulation window control enabled

D1:D0 Register set select

00	Primary register set (default)
01	DMA register set
10	Cartridge emulation register set
11	Reserved set

Bit 7 indicates when the button on the cartridge has been pushed, regardless of whether it is being used for reset or software control. It stays set until cleared by a write to $D5FC with bit 7 clear, but cannot be set by write.

Bit 6 indicates the state of the SpartaDOS X switch on the cartridge. It is not latched and changes immediately when the switch is moved.

Bit 5 manually controls the green LED. It is XORed with the SD power enable (primary set $D5F3 bit 5), so it can be used to force the green LED to either on or off state.

Bit 4 indicates whether the boot is for a cold start or warm start. It is set by the hardware on power-up and can be reset by a write with bit 4 cleared. Once cleared it cannot be set again, even by the reset push-button.

Bit 3 determines whether the button on the cartridge triggers a cartridge reset or if it is only used for software control. When set, it suppresses the reset function and only sets bit 7.

Bit 2 allows window enable/disable control from the cartridge emulation to be forced off for emulation of the pass-through cartridge port on the SpartaDOS X cartridge. When set to 0 while cartridge emulation is enabled, the window disable logic from the cartridge emulation is disabled and the flash/ROM enable bits in primary set register $D5FA take effect as usual, but banking is still driven by the cartridge emulation engine. This only has an effect if cartridge emulation is enabled and the SpartaDOS X switch is turned on; if the switch is turned off or cartridge emulation is disabled, this bit has no effect, but can still be changed or read.

Bits 0 and 1 determine the register set that is visible at $D5F0-D5FB.

Common control 2 ($D5FD; write only)

7							0
	RAM						

D6 CCTL RAM aperture enabled

0	CCTL RAM aperture disabled (default)
1	CCTL RAM $1FF500-1FF57F mapped to $D500-D57F

Bit 5 enables a 128-byte window in the lower half of CCTL at $D500-D57F to a small area at the top of cartridge RAM at $1FF500. This can be used for temporary storage or as a staging area for DMA transfers. The window is read/write, unless write protected by primary $D5FB bit 4.

The CCTL RAM window operates in parallel to cartridge emulation banking. If both are enabled over the same region, accesses are handled by both the RAM window and the cartridge emulation. This includes a write protected window, in which case the write is ignored by the RAM window but can still change cartridge banks.

Unlike other registers, this register is write-only as it is underneath one of the signature bytes. The current state can be detected by the change in the first signature byte from 'S' ($53) to 'R' ($52).

Signature ($D5FD-D5FF; read only)

$D5FD-D5FF read the characters 'SSD' ($53 53 44) to identify the cartridge.

If the CCTL RAM window is enabled ($D5FD bit 5), $D5FD reads 'R' ($52) instead.

Primary register set

Green LED brightness ($D5F0; read/write)

Controls the brightness of the green LED, which is active when ($D5FC bit 5) xor (primary $D5F3 bit 5). A value of $00 turns off the LED, while a value of $FF gives maximum brightness. The initial power-on value is 80 ($50).

Due to PWM drive and LED behavior, the mapping from values written to $D5F0 to resulting intensity is very non-linear; the difference between $00 (off) and $01 (minimal) is much greater than between $01 and $02, and differences are minimal at the high end. This can (very) roughly be mapped to perceptual or sRGB values by a square or cube root.

Reserved ($D5F1-D5F2; unmapped)

These locations are unmapped in the primary register set. Depending on the bus configuration of the host computer, reads from these locations will either read $FF or open bus data, same as if no cartridge were present.

SD/SPI control ($D5F3; read/write)

7							0
CHG	LCK	PWR	SPD	FRE	RTC	BSY	SD

D7 SD card change detect (partial read/write)

0	No card inserted or insertion not acknowledged
1	Cart inserted and acknowledged

D6 SD lock state (read-only)

0	Card unlocked (slider up)
1	Card locked (slider down) or no card inserted

D5 SD power enable (read/write)

0	Unpowered (default)
1	Powered

D4 SPI transfer rate

0	Slow (32 clock cycles)
1	Fast (1 clock cycle)

D3 SPI free-run mode

0	Transfer only on write to $D5F4 (default)
1	Transfer on read/write to $D5F4 or during DMA transfer

D2 RTC select

0	Unselected
1	Selected (default)

D1 SPI transfer status (read-only)

0	Idle
1	Busy

D0 SD select

0	Unselected
1	Selected (default)

Bit 7 is used to detect SD card insertion. It is forced to 0 when no card is present, but may be set to 1 when a card is present. As the hardware does not set this bit unless a 1 is written into it, this allows detecting a card change without software needing to poll at a high enough rate to notice the removal. Once the bit is set to 1, it cannot be reset to a 0 by a write, only by card removal.

Bit 6 reads the state of the physical lock slider on the side of the SD card. It is 0 (unlocked) if a card is inserted with the slider down, and 1 (locked) if the slider is up or no card is inserted. The hardware only senses this slider, and it is up to software to enforce the write protection. This bit is read-only.

Bit 5 enables power to the SD card. Changing this bit also inverts the state of the green LED.

Bit 4 determines the SPI transfer rate. 0 selects a slow rate of 4 cycles/bit and is required for the real-time clock chip. 1 selects a fast rate of 1 cycle/byte and is used for SD transfers. SD transfers may also be done at low speed.

Bit 3 selects SPI free-run mode, where transfers are automatically triggered with $FF bytes on read. This allows for fast reads from the SD device without having to explicitly write $FF dummy bytes. This is intended (required) for DMA operation as it is too fast for the CPU at fast transfer rate, and the DMA engine can only read from the free-run buffer.

Bit 1 reports whether the SPI interface is still shifting. In slow mode, the input byte cannot be read and a new output byte cannot be written until this bit clears. In fast mode, this bit will always be 0 as the transfer will always finish within one cycle.

Bits 0 and 2 select the SD and RTC devices on the SPI bus, respectively. Only one bit should be set at a time.

SPI data ($D5F4; read/write)

$D5F4 is used for data transfers to and from the SPI bus. The SPI protocol uses a common clock for both reading and writing, so any transfer both writes a byte to the device and reads a byte back. Writing to $D5F4 sets the next byte to write and shifts a byte in both directions, after which the byte read can be read from $D5F4. This means that the read is one byte behind, as the byte that arrives is shifted in while the written byte is shifted out; after a command byte is written, one additional byte must be written for the first response byte to be received.

In normal transfer mode, reading from $D5F4 simply reads the last received byte and does not initiate a new transfer or shift in a new byte. In free-running mode, reading $D5F4 once a transfer is complete will start a new transfer with $FF being written out to selected devices.

For slow transfer mode, bit 4 of $D5F3 must be polled to check that the transfer has completed. This is particularly important for accelerated systems. The input shifter is not double-buffered as in POKEY and will be a mix of input and output bytes while the transfer is in progress. In fast mode, this is not necessary as the transfer occurs within one machine cycle and the cartridge interface bottlenecks any accelerator from outrunning the transfer.

Any SPI transfer, by writing $D5F4 or reading/DMAing in free-run mode, will cause the red LED to activate momentarily. The duration is approximately 36ms (2^{16} cycles). Repeated activity will reset this timeout and keep the LED on indefinitely. This occurs regardless of the SD/RTC select signals and even if neither device is enabled.

CRC-7 register ($D5F5; read-only)

This register reports the current value of the CRC-7 computation, which tracks all bytes written to through the SPI interface. The CRC-7 polynomial is $x^7 + x^3 + 1$ and reported in bits 1-7 with bit 0 always set, giving a value compatible as the end token for SD card commands.

$D5F5 is reset to $01 by any write to the $D5F3 register, even if the same value is re-written. This can be used to reset the CRC calculation without deselecting the SD card device.

The CRC-7 value is updated as bits are shifted out to the SPI bus, so it is only valid once the transfer has completed. This only matters in slow transfer speed where it can be read during the transfer.

In SPI free-run mode, the CRC-7 register is instead updated by the incoming data instead of the outgoing data, and is not generally useful.

For the SD GO_IDLE_STATE (CMD0) initialization command of $40 00 00 00 00 95, the CRC-7 register values after writing each of the first five bytes are: $C9 DB CD 93 95. Thus, after writing the fifth byte ($00), the $D5F5 register contains the sixth byte $95 to write.

Window A RAM bank ($D5F6; read/write)
Window B RAM bank ($D5F7; read/write)

Contains the bank used when window A or B is enabled for RAM access. RAM banks are aligned to an 8K boundary. 256 × 8K = 2MB total addressable RAM.

Window A flash bank 7:0 ($D5F8; read/write)
Window B flash bank 7:0 ($D5F9; read/write)

Contains the low 8 bits of the bank used when window A or B is enabled for flash access, with each bank being 8K in size on an 8K boundary. This maps to bits 13-20 of the byte offset in the flash memory.

Writes to these registers also latch the corresponding high flash bank bits from $D5FA into the active flash bank, for atomic bank switches. A write to $D5F8 latches bits 4-5 and a write to $D5F9 latches bits 6-7.

Window control ($D5FA; read/write)

7					0
A flash 9:8	B flash 9:8	A-F	B-F	A-R	B-R

D7:D6	Window A flash bank 9:8
D5:D4	Window B flash bank 9:8
D3	Window A flash enable
D2	Window B flash enable
D1	Window A RAM enable
D0	Window B RAM enable

0	Flash/RAM access disabled
1	Flash/RAM access enabled

Bits 4-7 provide the high two flash bank bits for windows A and B. Writes to these bits do not take effect immediately; they only do so after a write to the corresponding low bank register in $D5F8 or $D5F9.

Bits 0-3 enable windows A and B for either flash or RAM access. If both the flash and RAM bits for a window are 0, the window is disabled and main RAM is revealed. If both the flash and RAM bits are enabled for a window, flash has priority over RAM.

When cartridge emulation is active and not forced off by $D5FC bit 2, only the flash enable bits have effect; the RAM enable bits are ignored, and the high flash bank bits are latched but not used until cartridge emulation is disabled.

Window write protect ($D5FB; read/write)

7			0
0	CTL	A	B

D7:D3	Reserved (read as 0)	
D2	CCTL window write protect	
	0	Read/write (default)
	1	Read-only
D1	Window A RAM write protect	
D0	Window B RAM write protect	
	0	Read/write
	1	Read-only (default)

Allows write-protecting any of RAM windows for ROM emulation. An enabled, write-protected window blocks writes from affecting either cartridge RAM or the underlying main RAM. This has no effect on writes to flash-mapped windows, which can activate command mode regardless of the state of $D5FB.

Bit 2 write protects the CCTL window at $D500-D57F, if enabled by $D5FD bit 5.

DMA register set

Note that the DMA registers use big-endian ordering for multi-byte values, with the higher byte registers being first in address order.

Transfer control ($D5F0; read/write)

7		0
Ena.	Mode	Source address 20:16

D7	Transfer enable/status	
	0	Transfer stopped
	1	Transfer running
D6:D5	Transfer mode	
	00	SD-to-memory
	01	Memory-to-memory
	1X	Reserved
D4:D0	Source address 20:16	

A transfer is started by setting bit 7; this bit is then cleared by the hardware when the transfer completes. Bit 7 can also be cleared to stop a transfer in progress.

The source address, destination address, and transfer length are copied into internal registers when a DMA transfer is started. Changing these registers during a DMA transfer will not affect the transfer in progress, and the registers are not updated by the transfer. Thus, after a transfer completes, the same transfer can be repeated simply by setting the enable bit again. Other register states, such as the transfer mode, AND/XOR masks and stepping offsets, will take effect immediately if written during a transfer.

In SD-to-memory mode, the DMA engine copies bytes from the input shift register connected to the SPI interface and writes it to memory. This will work either with the RTC or the SD card and at either slow or fast speed, but SPI free-run mode must be enabled for the transfer to work properly. This transfer is designed for SD read commands, as after every 512 bytes transferred the DMA engine will discard two bytes for the CRC-16 and then wait for the Start Block token ($FE) before transferring up to another 512 bytes. This means that the transfer should be started and then free-run enabled after the Start Block token is read. An SD-to-memory mode transfer will not progress unless the SPI shift registers are active, and otherwise will stay in transfer-active status indefinitely.

Source address 15:8 ($D5F1; read/write)
Source address 7:0 ($D5F2; read/write)

Sets the lower 16 bits of the source address for DMA transfers in RAM.

Destination address 20:16 ($D5F3; read/write)
Destination address 15:8 ($D5F4; read/write)
Destination address 7:0 ($D5F5; read/write)

Sets the destination address for DMA transfers in RAM. Bits 5-7 of $D5F3 always read 0.

Transfer length 15:8 ($D5F6; read/write)
Transfer length 7:0 ($D5F7; read/write)

Sets the transfer length in bytes, minus one. $0000 specifies one byte to transfer, and $FFFF means to transfer 65,536 bytes.

Source address step ($D5F8; read/write)
Destination address step ($D5F9; read/write)

Value added to the internal source and destination addresses after each byte is transferred. Both values are 8-bit signed values, sign-extended to 21 bits during the update.

AND mask ($D5FA; read/write)
XOR mask ($D5FB; read/write)

Determines the masks used to modify byte values during a memory-to-memory transfer. For each transferred byte, the source byte is bitwise-ANDed with the AND mask and then bitwise-XORed with the XOR mask before being written to the destination. An AND mask of $00 results in a fill with the source byte being ignored and the destination being written with the XOR mask, while an AND mask of $FF and an XOR mask of $00 does a verbatim copy.

There are no changes in transfer timing for any specific values of AND or XOR mask.

The AND and XOR masks have no effect for an SD-to-memory transfer.

Cartridge emulation register set

CCTL range base ($D5F0; read/write)
CCTL range mask ($D5F1; read/write)

Controls the range of CCTL reads or writes used for emulation banking when the CCTL range is enabled ($D5F6 bit 1). When the mask is enabled, the banking emulation logic only responds to a CCTL access if the bits enabled by the mask match those in the base (((address AND mask) = base). Setting a bit in the base register that is not set in the mask register will cause all accesses to be rejected.

CCTL address bank mask ($D5F2; read/write)
CCTL data bank mask ($D5F3; read/write)

Controls which bits in the CCTL access address or data are passed through to the banking logic. A value of $03, for instance, passes through the low two bits for 4 banks (32K or 64K cartridge). Only one of these masks is active, depending on whether data or address based banking is being used ($D5F6 bit 7).

Window A disable mask ($D5F4; read/write)
Window B disable mask ($D5F5; read/write)

Determines which bits in the CCTL access disable window A or B of the emulated cartridge, when disable-by-value is enabled for the window (bits 4 and 3 of $D5F6). The window is disabled if the access value has any bits set in common with the mask ((value AND mask) ≠ 0). Typically only one bit will be set when this feature is used.

This function may be inverted by bits 0 and 1 of $D5FA, so that a 0 bit is required instead of a 1 to disable.

Feature control ($D5F6; read/write)

7							0
ADR			DISA	DISB	DBM	MSK	BB

D7 CCTL data/address option

0	Bank switch based on data value of CCTL write
1	Bank switch based on address of CCTL read or write

D5:D4 Reserved (read/write)

D4 Window A disable option

0	No disable option for window A
1	Disable window A when CCTL access value has any 1 bits in common with window A disable mask

D3 Window B disable option

0	No disable option for window B
1	Disable window B when CCTL access value matches disable mask, according to bit 2

D2 Window B disable by value option

0	Disable window B when CCTL access value has any 1 bits in common with window B disable mask
1	Disable window B when CCTL access value is equal to maximum value

D1 CCTL address range option

0	CCTL address base/mask ($D5F0/1) disabled
1	CCTL address base/mask ($D5F0/1) enabled

D0 Bounty Bob mode option

0	Normal mode

1	Bounty Bob Strikes Back mode -- window A split into independent 4K/4K banks with $xFF6-9 banking

Bit 7 is the master switch for whether CCTL bank switch requests are based on data writes or addresses. If cleared, only writes switch banks and the data determines the new bank; if set, either reads or writes will switch banks and the address determines the bank. This also determines whether $D5F2 or $D5F3 is used as the bank mask.

Bits 5 and 4 are currently unused, but read/write.

Bits 4 and 3 enable an option for disabling windows A or B when specific bits are set in banking value. If *any* bits are set in both the banking value and the disable mask, then the window will be disabled. For instance, a value of $0C causes the window to be disabled if either bits 2 or 3 are set. These bits do not have to be enabled in the address/data bank mask ($D5F2/3). Typically, only one bit will be set.

Bit 2 enables an alternate mode for window B disable when that is turned on (bit 3). Instead of a bitwise AND with the disable mask, the banking value is compared against its maximum value. For instance, if the banking mask is $07 (8 banks), window B will be disabled if bank 7 is selected.

Bit 1 enables the CCTL range registers in $D5F0/1 so that only a range of CCTL addresses cause bank switching.

Bit 0 enables special logic for emulating the Bounty Bob Strikes Back! cartridge, which has two independent 4K windows at $8000-8FFF and $9000-9FFF. In this mode, the banking register is split into nibbles, with the low nibble controlling the $8xxx window and the high nibble controlling the $9xxx window. Accesses to $8FF6-8FF9 set the low nibble and accesses to $9FF6-9FF9, with only the low four bits of the banking value and mask being used. These nibbles are then used to map $8xxx to RAM banks 0-15 and $9xxx to RAM banks 16-31.

Control ($D5F7; read/write)

7					0
ENA	IDA	IDB	LCK	0	Mode

D7 Enable/disable

0	Cartridge emulation disabled
1	Cartridge emulation enabled

D6 Invert disable A
D5 Invert disable B

0	Disable banking window when disable conditions satisfied
1	Enable banking window when disable conditions satisfied

D4 Register lock

0	Writes to $D5F0-D5FF handled by register set logic
1	Writes to $D5F0-D5FF handled by cartridge emulation banking logic

D2:D0 Emulation mode

000	Standard 8K/16K
001	Williams (8K)
010	XEGS (8K+8K) / BBSB (4K+4K+8K)
011	- Reserved -
100	MegaCart (16K)
101	- Reserved -
11x	- Reserved -

Bit 7 is the master enable/disable for cartridge emulation. A write to $D5F7 with bit 7 set also resets the current bank to the value of $D5F8 and the disable states from $D5FA.

Bit 6 and 5 invert the result of the window disable conditions set up by $D5F4-6, primary allowing a 0 bit to disable instead of a 1 bit. This only works in emulation modes other than standard 8K/16K (%000).

Bit 4 locks the SIDE 3 registers, preventing the SIDE 3 configuration from being changed and allowing $D5F0-D5FF to be used for emulated cartridge banking. Without it, writes to these addresses are handled by the register file and do not change emulated cartridge banks; with it, writes only affect the emulated cartridge. Reads are not affected and still return data from the register file. Bit 4 can only be set when bit 7 is already set or is being set at the same time; it cannot be set if cartridge emulation is disabled and not being enabled by the new write to $D5F7. Once bit 4 is set, it can only be reset by the push button if reset by button is enabled.

Bits 0-2 set the emulation mode that maps the initial and current banking values to the window banks:

- Standard mode (%000): $D5F8 and $D5F9 control the window banks and the current bank is not used.
- Williams mode (%001): Window A is always disabled, window B uses the current bank.
- XEGS mode (%010, $D5F6 bit 0 = 0): Window A uses the current bank, window B uses $D5F9 as its bank.
- BBSB! mode (%010, $D5F6 bit 0 = 1): Window A uses the two nibbles of the current bank for its 4K halves, and window B uses $D5F9 as its bank.
- MegaCart mode (%100): Windows A and B switch banks together using the lower 7 bits of the current bank, which chooses a 16K bank, with the lower 8K mapped to window A and the upper 8K mapped to window B.
- Reserved values: Window A is disabled; window B uses $D5F9 as its bank.

Windows may either source from flash or RAM depending on $D5FA in the primary register set; only the flash enable bits are used, with RAM being selected if neither the RAM or flash bits are set for a window. For flash mappings, the top two bank bits are always zero for cartridge emulation regardless of the high bits in $D5FA.

Initial bank A ($D5F8; read/write)

Determines the initial banking value and also provides the bank for window A in some modes. The current bank is reset to this value when cartridge emulation turned on or re-enabled (write to $D5F7 with bit 7 set).

Initial bank B ($D5F9; read/write)

Provides the bank for window B in some modes.

Initial disable states ($D5FA; read/write)

7		0
0	DISB	DISA

D1	Window B initial disable state
D0	Window A initial disable state

0	Disable state is negated
1	Disable state is asserted

Sets the initial states of the disable logic. These are copied into the disable states when cartridge emulation is turned on or re-enabled (write to $D5F7 with bit 7 set). Note that this is prior to the inversion that may be applied by $D5F7 bits 5 and 6.

Also, note that the window bits are reversed from the order in many other registers.

Emulation mode ($D5FB; read/write)

Specifies the emulation mode in use, with bit 7 indicating cartridge emulation active and the lower bits containing the banking mode as specified in the CAR file format.

The value written to this register has no effect in hardware and is simply a read/write register for software use. However, it is cleared to $00 on a push-button reset.

Reserved register set

Register set 3 is reserved and has no useful functions or status. All writes to $D5F0-D5FB in register set 3 are ignored and all reads return $00.

8.7 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.

8.8 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).

8.9 Veronica

The Veronica cartridge adds a 65C816 coprocessor with 128K of RAM to the computer.[41]

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

[41] The author would like to thank the Veronica team for permission and assistance in publishing technical information about the Veronica cartridge.

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.

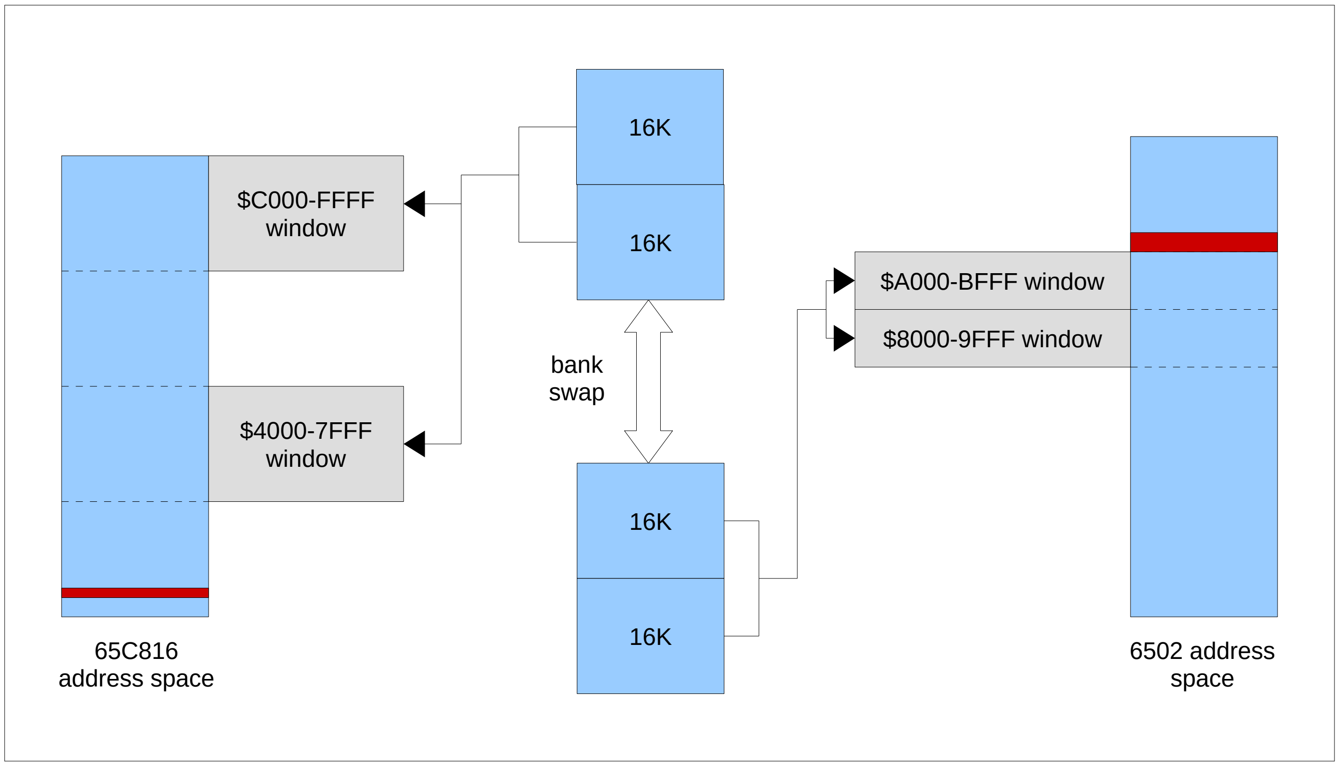


Figure 11: Veronica memory layout

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

Two data lines are used on the SIO bus, one for data sent from the computer (DATA OUT) and another for data received by the computer (DATA IN). The two are independent and can both can be used simultaneously for full duplex operation, even with different baud rates for the two directions. Multiple devices can listen simultaneously, but only one can send at any one time.

Each device on the SIO bus is configured as either a host or peripheral. The computer is wired as a host and sends on DATA OUT while receiving on DATA IN; peripherals are reversed and receive on DATA OUT while sending on DATA IN. As a result, peripherals cannot normally talk to other peripherals. One consequence of this is that popular SIO2PC devices, which are used to emulate a disk drive with a PC, cannot be used to control other disk drives because the SIO2PC is wired as a peripheral and a reverse-wired 10502PC device is required instead. A few devices are wired to operate in both modes, including auto-switching SIO2PC/10502PC adapters and the Indus GT disk drive.

On the computer, 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 from LSB to MSB, 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 28 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:)
$58	1030 serial device (T:)
$5F	Cassette tape (C:) (virtual OS device)
$6F	DOS2DOS (PCLink)
$70	SIO2USB
$71-74	SDrive
$72	SIO2SD

Table 28: 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.[42]

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.

> **Note**
>
> Although most OSes have no limitation on device ID range, the 65C816 XL OS is unable to access device IDs with the MSB set ($80-FF).

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[43]:

1. Command frame
    - The host lowers the command line to indicate the start of a command.

[42] 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.
[43] See [ATAXL] section 9 for the official SIO protocol description.

- 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 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.

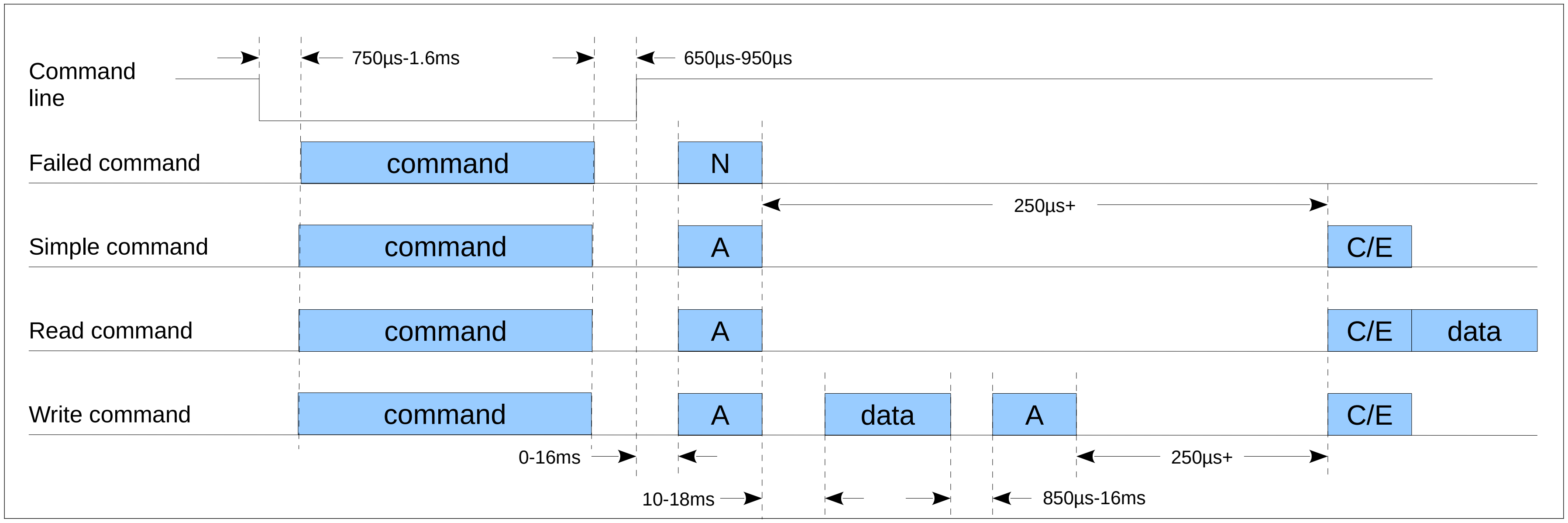


Figure 12: SIO command timing

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)

Warning

There is *no* specified delay between the Complete/Error byte and the beginning of the data frame during a read operation and several disk drives will send the data frame back-to-back with the C/E byte. The receive routine needs to keep the serial port hardware active in receive mode across these two sections of the protocol because the first data byte will likely start shifting in even before the CPU is able to read and acknowledge the C/E byte.

Interrupting commands

In general, the SIO bus does not support interrupting a command in progress. Once a device has recognized a valid command frame and begun acting upon it, it will likely ignore further commands on the bus until the operation has completed, as most devices do not have the ability to interrupt the controller when a command starts (the Indus GT being an exception). Other devices may still respond to further commands addressed to them, however; this can lead to a situation where two devices try to send data to the computer to the same time, producing garbled data.

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.[44] 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.[45] 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.[46] 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.

[44] [AHS05] p.8
[45] [AHS05] p.10
[46] The $0506 start address is, in fact, hardcoded by DOS 2.0's AUTORUN.SYS.

Type 2 Poll

Command $3F is also used for Type 2 Polls.[47] 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.[48] 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.[49]

The Poll Reset command resets all peripherals and restarts the polling process. It is issued with AUX1 and AUX2 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 28 slots available since that is the number of attempts from the OS (13 command retries with one device error retry gives (1 + 13) × 2 = 28 attempts).

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.[50] 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.[51] 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.[52]

[47] [AHS05] p.10
[48] [ATAXL] p.22
[49] [AHS05] p.9
[50] [AHS05] p.10
[51] [ATAXL] p.23
[52] [AHS05] p.11

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:

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 29: 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.

9.3 820 40 Column Printer

The Atari 820 40 Column Printer is a device with a similar case to the Atari 810 Disk Drive, but printing on roll paper coming out the top. It is natively supported by the P: device in the built-in OS.

SIO addressing

The 820 responds to SIO address $40. Only two commands are supported, Write ($57) and Status ($53).

Print Line command ($57)

The Print Line command is a write command that sends a line to the printer and waits for it to print. AUX1 is set to either $4E (Normal) or $53 (Sideways) for the character orientation; this also determines whether the line is 40 characters or 29 characters long, and thus the length of the data frame.[53] The sideways mode rotates characters to the right relative to normal orientation.

The data payload of the write command is the line, padded to the maximum line length. Although the OS P: handler inserts EOL characters into this buffer and other printers use these EOLs, the 820 ignores them. Instead, it simply treats any character outside of $20-7E in normal mode or $30-7F in sideways mode as a blank character, after masking off bit 7 (which is ignored). Thus, characters after the EOL must be filled with a whitespace-equivalent character for the 820 to print correctly.

In sideways mode, $60-7F are mapped down to $40-5F, converting the lowercase block to the uppercase block. Oddly, this means that $7F is a non-blank character in Sideways mode even though it is blanked in Normal.

9.4 820 hardware

Internally, the 820 has a 1MHz 6507, a 6532 RIOT with two data ports and 128 bytes of RAM, and 2K of ROM. It is thus similar to an 810, but with a faster CPU and without the floppy controller or extra memory. Like the 810, it drops A12 from the 6507, further reducing its address range to 4K.

[53] There is a error in [OSManual] p. 197, which says that the orientation is sent in AUX2.

Address range	Mapping
$0800-0FFF	ROM (2K)
$0780-07FF	RIOT registers
$0700-077F	Unmapped
$0680-06FF	RIOT registers
$0600-067F	Unmapped
$0580-05FF	RIOT memory
$0500-057F	Unmapped
$0480-04FF	RIOT memory
$0400-047F	Unmapped
$0380-03FF	RIOT registers
$0300-037F	Unmapped
$0280-02FF	RIOT registers
$0200-027F	Unmapped
$0180-01FF	RIOT memory (mirror)
$0100-017F	Unmapped
$0080-00FF	RIOT memory
$0000-007F	Unmapped

Table 30: 820 memory map

Port	Signal	Direction	Usage	Polarity
Port A	PA7	Output	Bottom-most print head pin	0 = activated
	PA6	Output	Print head pin	0 = activated
	PA5	Output	Print head pin	0 = activated
	PA4	Output	Print head pin	0 = activated
	PA3	Output	Print head pin	0 = activated
	PA2	Output	Print head pin	0 = activated
	PA1	Output	Top-most print head pin	0 = activated
	PA0	Output	Motor enable	0 = enabled
Port B	PB7	Input		
	PB6	Input		
	PB5	Input	SIO READY	0 = ready
	PB4	Output	SIO DATA IN (device to computer)	non-inverted
	PB3	Input	SIO COMMAND	1 = asserted
	PB2	Input	SIO DATA OUT (computer to device)	inverted
	PB1	Input	Paper Advance button sense	0 = depressed
	PB0	Input	Print ready sensor	0 = head at far left

Table 31: 820 RIOT port assignments

Print mechanism

The 820's print mechanism drives both paper advance and bidirectional print head motion with a single motor, requiring only minimal I/O from the controller. When the firmware enable the motor by lowering PA0 on the RIOT, the motor drives a cam that sweeps the print head first to the right to print and then back left to return. A microswitch raises PB0 to direct the firmware to output bitmaps for each character in sequence to PA1-PA7, driving the print head. A second cam advances the paper by one line after a line is printed.

A consequence of this design is that the 820 cannot precisely position the print head and must print a whole line at a time. The paper can only be advanced with a full head movement. In theory, this design does allow for bidirectional printing, though the positioning accuracy would be poor and the firmware does not attempt to do so.

9.5 850 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.[54] This program then loads, relocates, and

[54] As noted earlier, DOS II's AUTORUN.SYS hardcodes $0506 as the start address, so this can be relied upon.

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.

9.6 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. [55]

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.

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.

[55] 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.

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 32: 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.

9.7 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 33: 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.[56]

[56] 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.

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.

The SX212 powers up in high-speed mode.

9.8 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.

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.

9.9 410/1010 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.

Serial data is recorded onto the tape LSB-first, so every byte is written with a start bit, bits 0 to 7, and then a stop bit.

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.

9.10 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.

9.11 Pocket Modem

The Pocket Modem, by BOT Engineering, is a small SIO-based, 300 baud modem.

Data transfers

POKEY's standard serial port functionality is used for full-duplex communication with the modem. There is no setting for baud rate on the modem itself; the baud rate is determined solely by the rate at which POKEY sends and receives data. Supported baud rates are 110, 210, 300, and 500 baud.

Command protocol

The Pocket Modem does not use the standard SIO command protocol. Instead, it uses a custom command protocol based on both the command and motor control lines. The command line serves as the data line and the motor line serves as a clock.

Each command provides the setting for four mode bits on the device. The bits, in shifting order are:

- **Bit 0**: Enables data communication. When this is set, the modem transfers data between the SIO bus and the phone line.
- **Bit 1**: Off-hook state. Takes the phone off-hook (1) to answer or dial, or places it on-hook (0) to hang up.
- **Bit 2**: Sets the carrier frequencies to originate (0) or answer (1).
- **Bit 3**: Enables dialing if set to 1.

The command line is set to low for a 0 or high for a 1, and is shifted into the device on the falling edge (asserting edge) of the motor control line. There does not appear to be a direct indicator of when the last bit is shifted. The shift rate is *very* slow, about one bit every six video frames in the standard driver.

Commonly used modes include %0000 to hang up, %0011 for communication when originating a call, %0111 for comm when answering a call, %0010 or %0110 when temporarily suspending a call for disk access, and %1010 when dialing.

Pulse dialing

Tone dialing is not supported by the Pocket Modem, but it does support pulse dialing. Pulse dialing is enabled by switching to mode %1010, which takes the phone off hook and enables dialing mode. In dialing mode, the command line is used to control the on-hook state, such that asserting command (low) takes the phone off-hook, and negating command (high) places it back on-hook. Pulse dialing is then performed by pulsing the on-hook state with timing under software control.

Carrier detect

The SIO proceed line is used to communicate carrier detect state to the computer. It is asserted (low) when a remote carrier is absent and negated (high) when a carrier is present.

Ring detect

Auto-answer is also supported by means of a ring detection signal on the SIO interrupt line. The Pocket Modem sends pulses on the interrupt line whenever ringing is detected. It cannot answer the phone automatically, but software can count the ring pulses and switch the Pocket Modem to on-hook answer mode once enough pulses have arrived.

Chapter 10
Disk drives

10.1 Introduction

All floppy disk drives used with the 8-bit computer are built on top of the common 90K disk format and disk drive protocol established by the first disk drive, the 810 disk drive. The physical disk format is that used by the common Western Digital floppy drive controller (FDC) chips of the era and the disk protocol is a simple, high-level block protocol, leading to a high level of baseline compatibility between disk drives of differing internal architectures.

Geometry abstraction

All of the standard disk drive commands work in terms of logical sector numbering instead of physical track/sector numbers. For instance, a 90K single density disk is addressed as a linear array of sectors in the range 1-720. This frees the computer from having to take physical disk geometry into account. A hard disk or simulated disk may have very different physical storage, but as long as it exposes 720 sectors of 128 bytes each in the disk interface, it can substitute for a regular single density disk.

That having been said, the disk interface does not fully abstract away the geometry because it does not have a standard command to return the capacity of the disk in sectors or the size of each sector. This missing facility was later provided by the Percom block commands supported by many enhanced drives.

By convention, logical sector 0 is not used, although drives differ in whether they return an error for such a request. Since logical sector numbers are 16-bit unsigned values, the maximum disk size accessible through standard commands is 65535 sectors.

Disk formats

The most common Atari disk format is the original 90K single density format, composed of 720 sectors of 128 bytes each, arranged in 40 tracks of 18 sectors each on a 5.25” floppy disk. This is the only format supported by the 810 disk drive and the one used for most software distribution.

The 1050 drive added support for MFM encoding instead of FM, which allowed the capacity on each track to be increased from 18 sectors per track to 26 sectors per track, or 1040 sectors. This was called enhanced or medium density and increased storage per disk from 90K to 130K.

Some enhanced drives supported a double density format, encoding 18 sectors per track of 256 bytes each for 180K total. Beyond that, double-sided and 80 track encodings doubled or quadrupled the disk size further.

It is also possible for devices to support arbitrary disk sizes, such as 16384 sectors of 128 bytes each. When using such a disk geometry not related to a physical disk format, the Percom block will often indicate only a single track.

Boot sectors

To the boot loader used by the operating system, the boot sectors are a contiguous set of logical sectors starting at sector 1, where a header in sector 1 indicates the total number of boot sectors. The first stage of the boot loader can be anywhere from 1-255 sectors long. Standard disk operating systems typically use 1-3 sectors for their stage 1 loader, which can then scatter load the rest of DOS from the remainder of the disk.

Double density formats with 256 bytes per sector add an additional restriction to the boot loader, however. By convention, the first three sectors of a double-density disk are only 128 bytes long since the OS boot loader can only load sectors of that size. The stage 1 loader must therefore fit within 384 bytes and contain code to change the sector size for the remainder of the load. This precedent was set by DOS 2.0, which has support for bootstrapping from 3 x 128 boot sectors to 256 byte operation in its common 2.0S/2.0D boot code. Although the physical sectors on disk are 256 bytes long, the drive firmware only allows 128 bytes to be read or written from these sectors with the standard read/write commands.

There are a couple of exceptions to this behavior. Typically, devices using the disk drive protocol with a 512 byte sector do not special-case any boot sectors and transfer all sectors as 512 bytes. This includes hard drives and the TOMS Turbo Drive with 9 x 512 byte sectors per track. Second, the 1050 Duplicator is unique in formatting double density disks with only 128 byte boot sectors.

Reporting errors

Atari-compatible disk drives can report errors in one of two main ways. The first is to reply to the command with a NAK, and the second is to report an Error status instead of Complete. A NAK can only be returned within the very short command timeout of 1-2 frames and usually results from an invalid command, while an Error occurs later and generally results from a problem with accessing the disk. In either case, the Status ($53) command returns more detailed information about the error.

A third failure case that can occur is a checksum error when data is transferred from the computer to the drive. Command frames with bad checksums are always ignored without an error being returned, as they are either bogus or were intended for another device. Drives are inconsistent in how they handle data frames with bad checksums; some will NAK the data frame, while others will simply abort the command silently and rely on the computer timing out.

In the case of an actual error when reading or writing from the disk, drives will usually attempt to retry the operation internally at least once before returning an error to the computer. If the retry is successful, the only indication to the computer is a longer delay for the result. Depending on the specific error involved, the drive firmware may opt to seek the head away and back to the target track again if it suspects a head positioning problem. This is called a recalibrate or restore when the head is stepped all the way to track 0 and back, because it ensures that both the head position and the firmware’s tracking of it are consistent.

High-speed transfers

Standard disk drive commands always transfer data at 19,200 baud in both directions. Some enhanced drives, such as the XF551 and the US Doubler, support high speed transfers at 38,400 baud or higher speeds. About 60-70Kbaud is the highest speed that can reasonably be supported with the display enabled; beyond that it is difficult for the 6502 to keep up without dropping a byte due to display DMA and the vertical blank interrupt stealing cycles.

There are multiple methods of initiating a high-speed transfer, depending on the specific protocol. XF551-style protocols send the command at 19200 baud and switch to a higher speed for the data frame depending on bit 7 of the command. US Doubler type drives detect a command frame being sent at high speed and use that as the signal to run the entire protocol at that rate. Other drives may use high bits of the sector number or custom command IDs to signal. The US Doubler protocol is the most standardized as it allows the computer to query the drive for its high-speed transfer rate, permitting support of a variety of speeds with a single code path. Unfortunately, there is no direct and easy way to determine which high speed protocols that a disk drive supports.

High-speed disk transfer rates are sometimes referred to by “POKEY divisor,” which is the combined value written to POKEY's AUDF3 and AUDF4 registers to set the transfer rate. The US Doubler uses POKEY divisor 10, for instance, which gives a communications rate of 52,600 baud. The fastest rate possible with stock hardware is POKEY divisor 0, which gives a speed of 127 Kbaud. Note that the baud rate refers to the speed at which bits are sent; the delay between bytes can vary even at the same baud rate, and that does also affect the overall transfer rate.

NTSC/PAL incompatibilities

It is possible for compatibility issues to arise between a drive and NTSC/PAL computers due to the 1% difference in clock rate. This occurs when the transfer rate is marginal for either the computer or the drive for one of the types and the clock rate differences pushes the other type over the edge.

One example is the XF551, whose high-speed mode was incompatible with PAL computers with early firmware versions due to using too fast of a receive clock, just within 5% tolerance for the transmit rate of NTSC computers but over for PAL. Another is the Speedy 1050, whose high-speed receive loop on the drive is slightly too slow to keep up with the byte transfer rate of an NTSC computer.

Rotational latency

The timing of a read or write sector operation is determined not only by the time to transfer the sector, but also when the sector becomes available. When the floppy drive is active, the disk media is spinning at a constant speed of 288 or 300 RPM and the drive must wait until the desired sector arrives under the head before the sector can be read or written, causing additional delay for the operation. The maximum delay is determined by the rotational period of the disk, which at 288 RPM is 208ms.

Since the disk is constantly spinning when the drive is active, any delay for any reason results in a change in rotational position. This includes processing on the computer, receiving the command from the computer, and transferring sector data to/from the computer. One consequence of this is that these operations are hidden from a timing standpoint if the drive already has to wait for the sector to arrive. This is why high speed operation by itself is ineffective, because the 33ms that is saved with the higher transfer rate just results in 33ms more wait time for the next sector to arrive under the head.

On the other hand, if other operations take too long and the sector has already passed, the drive must wait for the media to spin around until the sector passes by again. This is known as “blowing a rev” and results in a major drop in read/write speed.

Interleaving

The single-density disk format places 18 sectors at a bit less than 20° apart on the track. However, a standard read or write sector command takes about 91ms end-to-end, or 156° at 288 RPM. This means that simply placing the sectors in order would perform poorly, because every time the drive attempted to read the next sector the drive head would be about 136° past it, requiring the disk to spin around more than another half revolution again. This would limit the transfer rate to less than one sector per full revolution of the disk.

A solution to this problem is interleaving, placing the sectors in a different order so each sector is as close as possible to where the drive would be ready to access it. Single density disks normally use a 9:1 interleave, which means that sequential logical sectors are placed nine physical sectors apart. This results in an ordering of 1, 10, 2, 11, 3, 12, … on the disk and doubles the transfer rate from one to two sectors per revolution. Enhanced density disks use a 13:1 interleave with 26 sectors per track, which also arranges about two sectors per revolution. Double density disks, on the other hand, require a longer delay due to having twice as long of a sector to transfer and use 15:1, putting each next sector almost all the way around the disk.[57]

Because the optimal interleave is dependent upon all delays including time to transfer the sector across the SIO bus, high-speed operation allows use of and often requires tighter interleave factors. XF551 high-speed mode, for instance, cuts 120° off the end-to-end sector time, allowing use of a 9:1 interleave instead of 15:1. However, this is only good at high speed and reads slowly when normal speed transfers are used.

The interleave pattern of a track is determined when the track is formatted and the sector address fields are

[57] The “about” part comes from the sectors not actually being all spaced nine apart: 9 is divisible into 18, so while sectors 1 and 2 are nine apart, sector 3 is 10 slots away from 2. This is why some poorly interleaved disks or slow loaders produce an alternating slow/fast beep pattern – the drive misses sectors on the short gaps but has enough time on the long ones.

written. After that, sector write operations reuse the existing address field and only rewrite the data field of each sector, preserving the existing sector order.

Interleaving does not change the sector IDs for each sector and thus a drive can always read tracks with any sector order, although slowly if sub-optiomal.

Track buffering

A more general way to solve the sector timing problem is track buffering. Instead of the reading each sector when requested and depending on sectors being interleaved optimally, the drive instead reads all sectors into a track buffer in memory first and transfers from the track buffer to the computer. This greatly reduces delays because the drive can read all sectors on the track in physical order instead of logical order, requiring only one revolution to do so, and then send buffered sectors to the computer without rotational delays. Track buffering can cut the time to read a full track by about 25% at normal speed, from 10 revolutions to 7½. Furthermore, the buffering eliminates the need to format the disk with an optimal interleave pattern, allowing even faster reads with high speed transfers even with a standard interleave.

There are a few downsides to track buffering. First, it requires a significant amount of additional memory. Base 810 and 1050 drives only have 256 bytes of RAM, but it takes a minimum of 2.25K to buffer a single density track. Second, some protected disks will not boot with track buffering because of the altered read timing. Third, track buffering can slow down random access because it requires two revolutions to read any sectors on a track even if only one sector is needed.

Drives vary in whether they buffer writes; some only buffer reads, while others have write buffering as an option.

The floppy drive controllers generally do not allow for reading all sectors on a track in a single revolution. Instead, one pass is required to read the sector order using Read Address commands to prepare for a second pass with Read Sector commands. To avoid this penalty on every read, buffering drives typically read the sector order once on track 0 and reuse it for other tracks, assuming that the sector order is the same on each track. A single Read Address command then suffices to find the next sector to arrive under the head before reading all sectors on the track. This reduces the latency for reloading the track buffer to about N + 1.5 sectors for N sectors/track.

Disk change detection

For both density detection and track buffering purposes, firmwares often have to detect when a disk has changed. There are a couple of ways to do this. On 1050-based drives, the drive lever state can be detected through the FDC Not Ready status bit, catching when the drive lever is opened and re-closed. On the 810, the write protect sensor will toggle when the disk is inserted or removed, due to the sensor being covered while the disk is being inserted or removed and uncovered when no disk is inserted. This allows the firmware to re-check the density of the disk and invalidate the track buffer when a new disk is detected.

In both cases, the firmware must continuously poll the state of the FDC in the drive idle loop to detect the changes. This means that disk change detection does not occur during a disk operation and can fail if the disk is changed during a running read or write sector command, and can lead to the new disk not being read or stale data from the track cache being returned to the computer. In the worst case, a drive with write buffering enabled can corrupt the new disk by writing pending data to the wrong disk.

Motor control and idle timeout

It is undesirable for the disk drive to constantly operate with the disk spinning since doing so puts additional wear on both the disk and the drive and consumes power. For these reasons, disk drives turn on the spindle and head stepping motors only when needed. The next operation requiring physical disk access has additional delay while waiting for the spindle motor to spin back up to speed and for the stepper motor to stabilize, usually around 50-

200ms.

Once the drive has gone idle for a predetermined amount of time, the drive firmware turns the motors back off. The idle timeout varies between drives but is typically about 2-3 seconds. Some drives also seek the head to a specific track when idling. A significant side effect of a drive idling is that the next access takes longer due to the need for the drive to spin back up and reposition the head.

Seeking

Although a disk has 40 or 80 tracks, the drive mechanism only has one head per side that can read one track at a time. The head must move between the different tracks as needed, a process known as seeking. This introduces additional delays into disk access when accessing sectors on different tracks, dependent on how many tracks the head has to move across.

The head is stepped between tracks by a *stepper motor*, a type of motor that moves in discrete steps. It is controlled by turning on and off different coils in the motor, called *phases*. Most stepper motors on Atari-compatible disk drives have four phases (810 and 1050), although three is also possible. For some stepper motors, one phase is turned on at a time in sequence to step the head (1, 2, 3, 4, 1...), and for others two adjacent phases have to be turned on at a time (1+2, 2+3, 3+4, 4+1, 1+2, …). The spindle motor must be active when the head is stepped.

80 track drives always step one phase per track, but some 40 track drives differ; some step by full tracks per phase (810) and others by a half track per phase (1050). For the latter, two steps are necessary to move a full track. Furthermore, because the head positioning may differ slightly for a half-step inward vs. outward, the firmware typically does two additional half-steps inward and outward when seeking inward to ensure consistent positioning regardless of direction.

Track 0 and recalibration

The stepper motor phases allow the motor to be relatively stepped between tracks, but provide no indication or control over absolute position. Therefore, the firmware has to first determining the position of the head before it can correctly seek. Stepping the head outward until it is known to be on track 0 establishes a known baseline from where the firmware can track the head's absolute position as it is stepped inward and outward. This is known as a *recalibrate* or *restore* operation.

One way to recalibrate the drive is to simply step it outward by enough steps to ensure that the head has stepped all the way to track 0, no matter where it may have been. This typically requires several more steps than the usual range of the head, such as 45 steps for a 40-track mechanism. Once the head reaches track 0, a stop prevents the head from moving any further, and so it will bang against the stop as it steps between track 0 and 1. On the 810, this produces the characteristic stuttery step sound when the drive is accessed with no disk. The stepper motor has a defined phase offset for track 0 that the recalibrate operation will always stop on, such as phase 1 or 4.

Some drive mechanisms have a track 0 sensor to detect when the head is positioned on track 0, avoiding the need to bang the head against the stop and minimizing recalibration time. A quirk of track 0 sensors is that they can also activate for an adjacent track or half-track; track 0 is reached when the sensor activates *and* the head is on the correct phase for track 0.

Besides startup, recalibration is also necessary when the firmware detects that the head may not be positioned over the correct track, such as when a sector is not found or sector address fields with the wrong track ID are found. When this occurs, the firmware recalibrates the head and re-seeks back to try to correct the head position. This produces the characteristic grr-grr sound when an error occurs.

Double-sided disks

Some drives support using both sides of the disk to double storage capacity. Side 0 is on the bottom and is the side used by both single-sided and double-sided disk formats. Side 1 is on the top and is the additional side used in a double-sided format. Inserting a double-sided disk into a single-sided drive will result in only side 0 being readable.

As with other formats, the translation between logical sector numbers and physical sides is done by the drive firmware and invisible to the computer, except when the drive geometry is queried or set through Percom block commands. The mapping from logical sectors to physical sectors on side 1, however, differs between disk drives. The ATR8000 maps side 1 in forward so that the logical sequence is from track 39, side 0, sector 17 to track 0, side 1, sector 0, while the XF551 maps it in reverse starting with track 39, side 1, sector 17 with reversed interleave. In both cases, all of side 0 is used before all of side 1 both for simplicity and optimal speed; this is different from some other platforms where both sides are used before stepping to the next track.

Task sequencing

For all contemporary physical disk drive types, the drive is a small computer in its own right with a standard CPU acting as its controller. The performance constraints on this controller CPU are typically strict enough that the drive firmware can only do one thing at a time, whether it be transferring data on the SIO bus, operating the floppy drive controller (FDC), or seeking the disk drive head.

The most direct consequence of this is the disk drive not being able to respond to changes in the hardware that it is not interacting with. For instance, during an actual sector read or write operation with the FDC, the controller is under very tight timing constraints, typically needing to transfer each byte within 32 cycles while also monitoring a watchdog timer. This leaves no CPU time for monitoring the SIO bus, which means that any attempts by the host computer to issue another command to the drive are ignored. Similarly, a track buffering drive will not notice a disk being changed while receiving or sending a data frame to the host computer since it is too busy managing the serial transfer to poll the ready or write protect state to detect the disk change.

Some disk drives do have some situational awareness during active command processing by means of interrupts. The Indus GT, for instance, can interrupt its Z80 processor when the SIO command line is asserted for the start of a new command. This gives it some ability to stop an active command in order to process a new incoming command, instead of always ignoring overlapping commands.

Drive characteristics

Base drive	Drive type	Controller	ROM	RAM	Formats	RPM	High speed	Code upload	Track buffering
Atari 810	Atari 810 (rev. B, C)	0.5MHz 6507	2K	0.25K	SD	288	-	-	-
	Atari 810 (rev. E)	0.5MHz 6507	2K	0.25K	SD		-	Yes	-
	Happy 810 (pre-rev.7)	0.5MHz 6507	3K	3.25K	SD		-	Yes	Read only
	Happy 810 (rev. 7)	0.5MHz 6507	6K	3.25K	SD		38K	Yes	Read/write
	Archiver / The Chip	0.5MHz 6507	4K	0.25K	SD		-	Yes	-
	810 Turbo	1MHz 6507	2K	4.25K	SD, DD		-	Yes	-
Atari 815		2MHz 6507	4K	0.375K	DD	288	-	Yes	-
Atari 1050	Atari 1050	1MHz 6507	4K	0.25K	SD, ED	288	-	-	-
	Happy 1050	1MHz 6502	8K	6.25K	SD, ED, DD		38K / 52K	Yes	Read/write
	Super Archiver	1MHz 6507	4K	2.25K	SD, ED, DD		-	Yes	-
	Speedy 1050	1MHz 65C02	16K	8.25K	SD, ED, DD		55K	Yes	Read/write
	US Doubler	1MHz 6502	4K	0.375K	SD, ED, DD		52K	-	-
	1050 Duplicator	1MHz 6502	4K	8.25K	SD, ED, DD		52K	Yes	Read only
	1050 Turbo	1MHz 6507	8K	0.25K	SD, ED, DD		71K	-	-
	TOMS 1050	1MHz 6502	8K	8.25K	SD, ED, DD			Yes	-
	Tygrys 1050	1MHz 6507	4K	2.25K	SD, ED, DD		38K / 55K	Yes	-
	I.S. Plate	1MHz 6502	8K	16.25K	SD, ED, DD		52K	Yes	Read/write
Atari XF551		8.3MHz 8040	4K	0.25K	SD, ED, DD, DSDD	300	38K	-	-
Indus GT		4MHz Z80	4K	2-64K	SD, ED, DD	288	38K / 68K	Yes	Read/write
ATR8000		4MHz Z80	4K	16-64K	SD, DD, DSDD+	300, 360	-	Yes	-
Percom RFD-40S1 / Astra 1001		1MHz 6809	2K	1K	SD, DD, DSDD+	300	-	-	-
Percom AT-88		1MHz 6809	2K	1K	SD, DD, DSDD+	300	-	-	-
Percom AT-88SPD/S1PD		1MHz 6809	4K	1K	SD, DD, DSDD+	300	-	-	-
Amdek AMDC-I/II		1MHz 6809	4K	1K	SD,DD,ED,DSDD+	300	?	Yes	-
Concorde C-221M		7.3MHz Z8681	2K	1K	SD		-	-	-

10.2 Basic protocol

All disk drives support a common set of commands, the commands supported by the original 810 disk drive. These commands can be used to access any disk of up to 65,535 sectors of 128 bytes each, for a maximum of 8MB.

Common command conventions

All disk commands are sent to the drive using a standard SIO command frame at 19,200 baud, and follow the usual conventions for ACK/NAK/Complete/Error.

Error conditions that can be detected from the command itself, like an attempt to read a sector with an invalid sector number, cause the command frame to be NAKed. On the other hand, error conditions only determined after an access to the disk are instead returned as an Error response, after the command frame is ACKed and possibly a data frame sent from the computer.

Drives differ on treatment of data frames that are received with invalid checksums. Some drives NAK the data frame, while others simply abort the command without sending a response.

The operation timeout for disk drive commands is typically $0F, or about 15 seconds. Most operations complete much more quickly, but this time accommodates the drive spinning up, seeking to the desired track, possibly performing multiple retries to read or write the sector, and then returning the result. The exception is the format command, which requires a much longer timeout specified in the Status command.

Status ($53)

The status ('S' = $53) command is used to query the status of the disk drive. AUX1 and AUX2 in the command frame are ignored. In response, the drive sends back a four byte status block.

Byte	Description
+0	Drive status • D7 = 1: Enhanced density (1040 sectors x 128 bytes) • D6 = 1: Double sided • D5 = 1: Double density (256 byte sectors) • D4 = 1: Motor running • D3 = 1: Failed due to write protected disk • D2 = 1: Unsuccessful PUT operation • D1 = 1: Last data frame received unsuccessfully • D0 = 1: Last command frame received unsuccessfully
+1	Inverted floppy drive controller (FDC) status • Bit 7 = 0: Drive not ready • Bit 6 = 0: Write protect error • Bit 5 = 0: Deleted sector • Bit 4 = 0: Record not found (missing sector) • Bit 3 = 0: CRC error • Bit 2 = 0: Lost data • Bit 1 = 0: Data request pending • Bit 0 = 0: Busy
+2	Format timeout, in units of 64 vblanks • $E0 (224) = 4 minutes (NTSC) or 4.8 minutes (PAL) • $FE (254) = 4.5 minutes (NTSC) or 5.4 minutes (PAL)
+3	Unused (usually $00)

Table 34: Disk drive status frame

Of the drive status bits, bits 7 and 5 are well supported among drives that support enhanced and double density operation, but bit 6 for double sided is less well supported. Percom drives do not set it even when configured for double-sided operation, while the XF551 will conservatively set this bit when detecting any double density disk, and the 815 always sets it despite not supporting double-sided operation. The TOMS Turbo Drive sets bit 6 when reading a 512 byte/sector disk.

The third byte suggests the timeout used for format commands. This is needed since a format disk operation takes much longer than other disk operations and needs a longer timeout, but using such a long timeout for regular disk commands would cause unacceptably long retry times for communication failures. The stock OS uses a value of $0F (16-19 seconds) for regular disk commands. Most disk drives use format timeout values of $E0 or above, which sets a timeout of at least four minutes. Some PC-based disk drive emulators, however, set this value to an abnormally low value of $03, which will cause a timeout error when DOS attempts to reuse this value with other real disk drives in the SIO chain.

Read ($52)

The read ('R' = $52) command reads a 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 the sector data, with the size determined by the format of the disk and whether the requested sector is a boot sector. This occurs even in the event of a read error.

On most drives, an attempt to read a sector with an invalid sector number – outside of 1-720 for a single density disk – will result in the command being immediately NAKed. There are drives that will accept sector 0, however, and the read sector command is also sometimes overloaded for memory access.

Put ($50)

The put ('P' = $50) command writes a 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 sector data is sent by the computer following acknowledgment of the command frame. The size of the sector data depends on the format of the disk and whether the requested sector is a boot sector.

On most drives, an attempt to write a sector with an invalid sector number – outside of 1-720 for a single density disk – will result in the command being immediately NAKed. There are drives that will accept sector 0, however, and the put sector command is also sometimes overloaded for memory access.

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. This requires another revolution on the disk (208ms), and thus significantly slows down the write operation compared to a Put command.

On most drives, an attempt to write a sector with an invalid sector number – outside of 1-720 for a single density disk – will result in the command being immediately NAKed. There are drives that will accept sector 0, however, and the write sector command is also sometimes overloaded for memory access.

Format ($21)

The format command ('!' = $21) command formats a disk, writing 40 tracks and then verifying all sectors. All sectors are fill with the data byte $00. On completion, the drive returns a sector-sized buffer containing a list of 16-bit bad sector numbers, terminated by $FFFF.

Because the format command takes much longer than other disk commands, it also requires a longer timeout. The status command returns the timeout required for a format operation, in multiples of 64 frames.

> **Note**
>
> Rev. 7.7 XF551s will continually format the disk in a loop when sent a format command with AUX1=$10 and AUX2=$50. It is recommended that both AUX bytes be set to $00 when invoking format commands.

10.3 Extended protocols

Many disk drives also support common extended protocols that allows flexible access to higher-capacity disk formats and high-speed operation.

Double density

Drives supporting true double density operation – 256 bytes per sector – also extend the read/write/put sector commands to use that size frame when accessing double density disks. Accesses to boot sectors (sectors 1-3) are an exception as they still use 128 byte data frames. The format command is also extended to return a 256 byte buffer when formatting a double-density disk.

High speed operation

The Get High Speed Index ($3F) command is used to query the high-speed index for a drive that follows the US Doubler protocol for high-speed operation. It is a read command that returns a data frame with a single byte, the POKEY divisor to use when communicating with the drive at high speed. This value is written into AUDF3, with

AUDF4 set to $00, to set the speed of serial transmission. A divisor of $28 is standard (18866-19040 baud), while high-speed divisors will usually be $0A or less (52640 baud or faster). Modern disk drive emulators can go all the way to POKEY divisor 0, for the fastest self-clocked rate of 127840 baud, although this requires a special ultra high speed SIO routine to transfer successfully on the computer side.

When a drive supports the US Doubler protocol, it accepts command frames at both normal 19200 baud and also the faster rate. The speed at which the command frame is sent determines the speed for the entire command, including ACK/NAK, complete/error, and data frames in both directions. No change to the command byte is necessary, nor are any high-speed flags used in the AUX bytes.

Command retry logic is essential when using the US Doubler protocol because many drives supporting it can only receive at one speed at a time and handle both speeds by toggling back and forth between them. This means that it is highly likely that the first command frame sent will be dropped. The TOMS Turbo drive is unusual in using the SIO clock lines to receive command frames at varying rates without alternating receive rates and relying on the computer to retry.

Not all disk drives use this method of high-speed communication. The Happy 810/1050, XF551, and Indus GT use differing methods that have no generic way of querying the drive, different high-speed command IDs, and use a low-speed command frame with a high-speed data frame. The 1050 Turbo uses bit 7 of AUX2 to enable high-speed data frame operation. Table 35 lists some known transfer rates used by disk drives.

		Baud rate (ideal)		
Type	**POKEY divisor**	**NTSC**	**SECAM**	**PAL**
Standard (19200 baud)	40	19040	18952	18866
XF551 Indus GT Synchromesh Happy 1050 (built-in) SIO2PC (38400 baud)	16	38908	38728	38553
US Doubler Tygrys 1050	10	52640	52397	52160
Speedy 1050	9	55930	55671	55420
SIO2PC (57600 baud)	8	59659	59383	59114
Indus GT SuperSynchromesh 1050 Turbo	6	68837	68519	68209
SIO2PC (115200 baud)	1	111860	111343	110840
Fastest possible (internal clock)	0	127840	127250	126674

Table 35: Disk drive transfer rates

Note that these transfer rates determine the timing of *bits*, not bytes; the byte transfer rate can be lower due to additional delays between bytes. Also, the rates given are ideal transfer rates when the computer is sending to the drive. When the drive is transmitting to the computer, it controls the transfer rate, and the actual ideal rate may vary from the above as long as it is within POKEY's tolerances for receive rate. The transmitter and receiver must agree in rate within 5% to avoid skewing by more than a half bit by the time each byte ends.

Percom block standard

High-capacity disk formats are almost universally managed by means of the Percom block standard, established by Percom Data Corporation with their line of disk drives. The current disk format setting on the drives is described in a 12 byte block called the Percom block, with a pair of query and set commands called Read Percom Block ($4E) and Write Percom Block ($4F).

Offset	Contents
+0	Number of tracks (typically 40 or 80)
+1	Step rate
+2-3	Sectors per track (typically 18 or 26; MSB first)
+4	Number of sides, minus 1 (0 or 1)
+5	Density 0 = 5.25" FM (4µs bit cell) 2 = 8" FM (2µs bit cell) 4 = 5.25" MFM (2µs bit cell) 6 = 8" MFM (1µs bit cell)
+6-7	Sector size in bytes (MSB first)
+8	Drive present (0 = absent, nonzero = present)
+9-11	Unused[58]

Table 36: PERCOM Block Contents

Note that all two-byte quantities in the Percom block are stored as big-endian (MSB first), because it was introduced on a 6809-based drive. This is opposite from the little-endian convention used by the 6502.

Most drives that support the Percom block only support the values 0 and 4 for the density byte. The Percom RFD-40S1 and the ATR8000 are exceptions as they also interpret bit 1 to enable clock-doubling for 8" disk operation. The Percom RFD directly ORs this value into the byte written into a hardware register, so all other bits must be 0 for normal operation.

The Percom block is used both to query the format detected for a disk and to set the format to use when formatting a new disk. There is typically only one setting on the drive and setting it differently from the currently mounted disk can render the disk unreadable by the drive. The drive may also overwrite the setting if the disk is changed, a read/write operation is triggered that causes the drive to auto-detect the format on the current disk, or the user selects a density via external controls on the drive.

Drive firmware varies widely in handling of the Percom block. The step rate setting is frequently ignored and firmware usually only validates and stores just enough information for the formats supported by the drives. The XF551, for instance, barely checks enough information to be able to distinguish the four formats that it supports, and on a read request re-formats the Percom block based on that information. Other drives may use more of the block – particularly the original Percom drives – and so it is a good idea to read the Percom block from the drive and modify it rather than simply writing a new block to the drive. This also means that re-reading the block after a modification attempt is necessary to verify that the changes have actually been accepted by the drive. Drives do not necessarily fail an attempt to write an unsupported configuration block but may do so. The 810 Turbo, for instance, will return Error for a double-sided format but silently ignores other fields, and the Percom AT-88 will return Error on an attempt to select double density on a controller only capable of single density.

For formatting, the current Percom block setting is used for the next format command, usually $21 (format), $66 (format skewed), or $A1 (format with high-speed skew).

Drives may or may not update the Percom block state when detecting the density of a new disk. The US Doubler and Super Archiver, in particular, do not. Table 37 lists the various status byte 0 and Percom block values detected by various drives.

[58] Byte 9 is documented in some places as a serial rate control byte, but is essentially unused. The only drive known to set it as such is the Percom RFD-40S1, which sets it to $41 to match its ACIA configuration, but the value in the Percom block is never used by the firmware even if it is changed.

Format	Drive	Status	Tracks	Step	Sectors	Sides-1	Density	Sector size	Present	Reserved
SD	1050 Duplicator	00	28	00	0012	00	00	0080	40	000000
	Happy 1050			00					FF	000000
	Speedy 1050			01					FF	000000
	TOMS 1050			00					28	000000
	1050 Turbo			00					01	000000
	Tygrys 1050			00					FF	000000
	XF551			00					01	410000
	ATR8000			00					40	000000
	Percom RFD-40S1			01					FF	Random
	810 Turbo			03					01	000000
	Amdek AMDC-I/II			00					FF	150000
ED	1050 Duplicator	80	28	00	001A	00	04	0080	40	000000
	Happy 1050			00					FF	000000
	Speedy 1050			01					FF	000000
	TOMS 1050			00					28	000000
	1050 Turbo			00					01	000000
	Tygrys 1050			00					FF	000000
	XF551			00					01	410000
	Amdek AMDC-I/II			00					FF	150000
DD	1050 Duplicator	20	28	00	0012	00	04	0100	40	000000
	Happy 1050	20		00					FF	000000
	Speedy 1050	20		01					FF	000000
	TOMS 1050	20		00					28	000000
	1050 Turbo	20		00					01	000000
	Tygrys 1050	20		00					FF	000000
	XF551	60[59]		00		01			01	410000
	ATR8000	20		00		00			40	000000
	Percom RFD-40S1	20		01					FF	Random
	810 Turbo	20		03					01	000000
	Amdek AMDC-I/II	20		00					FF	150000

Table 37: Detected Percom blocks for various disk drives and formats

Medium / enhanced density is a special case as it may not be accessible via the write Percom block command, particularly with the 1050 drive that does not support the Percom block. Instead, the enhanced density format must be detected via bit 7 of the drive status byte from the status ($53) command, and selected by formatting with the format medium command ($22). Double density is also such a case on the 815, which also does not support the Percom block commands but does indicate a double density disk via drive status bit 5.

[59] The XF551 is unable to detect whether a double density disk is double sided or not, so it simply assumes double sided.

10.4 Commands

Value	Command	Drive support
$20	Download	810 rev. E ($0080), 815 ($0000)
$21	Format	All drives
$22	Format medium	1050 and all other ED-capable drives
$23	Diagnostic (write/execute)	1050, Happy 1050 rev. 1, Tygrys 1050, 1050 Turbo
$24	Diagnostic (execute/read)	1050, Happy 1050 rev. 1, Tygrys 1050, 1050 Turbo
$24		TOMS 1050
$36	Clear sector broadcast flags	Happy 810 rev. 7
$37	Verify sector broadcast	Happy 810 rev. 7
$38	Broadcast sectors	Happy 810 rev. 7
$3F	Get high speed index	Most high-speed capable drives
$42		Archiver
$43	Check sectors	Archiver
$44		Archiver
$46	Format track	Archiver
$47	Scan track	Archiver
$48	Happy command	Happy 810 rev. 7, Happy 1050, Tygrys 1050
$4C	Load sector list	Archiver
$4D	Execute code	Archiver
$4E	Read Percom block	Most double density capable drives
$4E	Set idle timeout / idle drive	Archiver
$4F	Write Percom block	Most double density capable drives
$4F	Open Archiver	Archiver
$4F	Write sector with verify (alias)	810 rev. B, C
$50	Put sector (write without verify)	All drives
$51	Quiet (turn off motor)	Happy 810 rev. 7, Happy 1050, Tygrys 1050
$51	Spin test	815
$52	Read sector	All drives
$53	Status	All drives
$54	Read address	810 rev. E (128 bytes), 815 (256 bytes)
$54	Read trace buffer	Archiver
$55		TOMS 1050
$55	Motor delay	815
$56		TOMS 1050
$56	Verify sector	815
$57	Write sector with verify	All drives
$58	Execute code	Indus GT
$58	Extended info	815
$5A	Zero sector / trace buffer	Archiver
$66	Format skewed	US Doubler, Tygrys 1050

Value	Command	Drive support
$70	Put sector (high speed)	Happy 810 rev. 7, Happy 1050, Tygrys 1050
$72	Read sector (high speed)	Happy 810 rev. 7, Happy 1050, Tygrys 1050
$77	Write sector with verify (high speed)	Happy 810 rev. 7, Happy 1050, Tygrys 1050
$A1	Format with high-speed skew	Indus GT, XF551 (slightly different protocols)
$A2	Format medium with high-speed skew	Indus GT, XF551
$A3	Format boot tracks with high-speed skew	Indus GT
$CE	Read Percom block (high speed)	XF551, Indus GT
$CF	Write Percom block (high speed)	XF551, Indus GT
$D0	Put sector (high speed)	XF551, Indus GT
$D2	Read sector (high speed)	XF551, Indus GT
$D3	Get status (high speed)	XF551, Indus GT
$D7	Write sector (high speed)	XF551, Indus GT

10.5 Timing

Transmit rates

While transmit rates on the computer side are set by POKEY divisor, transmit rates on the drive side are determined by timing loops in the firmware. There is significant variance in the rates that are used. Table 38 gives both bit and byte timings for many drive types and transfer modes.

Type / mode	Method	Bits		Bytes	
		Cycles	Bits/ sec	Cycles	Bytes/ sec
Atari 810 rev. B, C, E	Bit-bang	26 cycles @ 0.5MHz	19230.8	265 cycles @ 0.5MHz	1886.8
Happy 810	Bit-bang	26 cycles @ 0.5MHz	19230.8	265 cycles @ 0.5MHz	1886.8
Happy 810 high speed	Bit-bang	13 cycles @ 0.5MHz	38461.5	151 cycles @ 0.5MHz	3311.3
810 Turbo V1.2	Bit-bang	53 cycles @ 1MHz	18867.9	531 cycles @ 1MHz	1883.2
Atari 815	Bit-bang	104 cycles @ 2MHz	19230.8	1043 cycles @ 2MHz	1917.5
Atari 1050	Bit-bang	51 cycles @ 1MHz	19607.8	549 cycles @ 1MHz	1821.5
1050 Duplicator	Bit-bang	51 cycles @ 1MHz	19607.8	573 cycles @ 1MHz	1745.2
1050 Duplicator high speed	Bit-bang	19 cycles @ 1MHz	52631.6	249 cycles @ 1MHz	4016.1
1050 Turbo	Bit-bang	52 cycles @ 1MHz	19230.8	520 cycles @ 1MHz	1923.1
1050 Turbo high speed	Bit-bang	14 cycles @ 1MHz	71428.6	151 cycles @ 1MHz	6622.5
Happy 1050[60]	Bit-bang	51 cycles @ 1MHz	19607.8	524-526 cycles @ 1MHz	1908.4
Happy 1050 high speed	Bit-bang	26 cycles @ 1MHz	38461.5	315 cycles @ 1MHz	3174.6
Happy 1050 USD emulation	Bit-bang	19 cycles @ 1MHz	52631.6	220 cycles @ 1MHz	4545.5
US Doubler	Bit-bang	53 cycles @ 1MHz	18867.9	534 cycles @ 1MHz	1872.7
US Doubler high speed	Bit-bang	19 cycles @ 1MHz	52631.6	220 cycles @ 1MHz	4545.5
Speedy 1050	Bit-bang	52 cycles @ 1MHz	19230.8	525 cycles @ 1MHz	1904.8
Speedy 1050 high speed	Bit-bang	18 cycles @ 1MHz	55555.5	214 cycles @ 1MHz	4672.9
Tygrys 1050[61]	Bit-bang	51 cycles @ 1MHz	19607.8	521 cycles @ 1MHz 523 cycles @ 1MHz	1919.4 1912.0
Tygrys 1050 high speed 1	Bit-bang	26 cycles @ 1MHz	38461.5	296 cycles @ 1MHz	3378.4
Tygrys 1050 high speed 2	Bit-bang	19 cycles @ 1MHz	52631.6	219 cycles @ 1MHz 221 cycles @ 1MHz	4566.2 4524.9
I.S. Plate	Bit-bang	53 cycles @ 1MHz	18867.9	532 cycles @ 1MHz	1879.7
I.S. Plate high speed	Bit-bang	19 cycles @ 1MHz	52631.6	221 cycles @ 1MHz	4524.9
Atari XF551	Bit-bang	29 cycles @ 0.55MHz	19157.1	290 cycles @ 0.55MHz	1915.7
Atari XF551 high speed	Bit-bang	14 cycles @ 0.55MHz	39682.5	140 cycles @ 0.55MHz	3968.3
Indus GT	Bit-bang	209 T-states @ 4MHz	19138.7	2237 T-states @ 4MHz	1788.1
Indus GT Synchromesh	Bit-bang	104 T-states @ 4MHz	38461.5	1163 T-states @ 4MHz	3439.4
Indus GT SuperSynchromesh	Bit-bang	58 T-states @ 4MHz	68695.5	598 T-states @ 4MHz	6689.0
ATR8000 v3.02	Bit-bang	208 cycles @ 4MHz	19230.8	2080 cycles @ 4MHz	1923.1
Percom RFD-40S1	UART	208 cycles @ 4MHz	19230.8	2080 cycles @ 4MHz	1923.1
Concorde C-221M v1.2	UART	384 cycles @ 7.3728MHz	19200.0	3840 cycles @ 7.3728MHz	1920.0

Table 38: Disk drive transmit timings by firmware

Most drives use the bit-bang method of SIO communication, where the controller – typically a 6507 – manually times and reads or writes one bit at a time. This places very tight timing constraints on the firmware, but allows high-speed operation to be added by a firmware change. The Percom drive is unusual in using a UART chip to handle serial conversion.

[60] Happy 1050 takes 526 cycles/byte for half of SD sectors when using track buffering due to page crossings.
[61] The Tygrys 1050 firmware indexes across pages when sending from internal non-sector buffers. This causes bytes sent from the Status and Read Percom Block commands to take two additional cycles.

The bit rate is critical to transfer reliability and must match as closely as possible between the computer and the drive. The reason for this is that the computer and drive only synchronize bit timing at the leading edge of the start bit and then shift bits in/out independently without a shared clock signal. In order to successfully transfer a byte, the transmit and receive rates must match to within 5% or else the timing error will exceed one-half bit by the stop bit, which will cause the receiver to mis-sample at least one bit. The bit rates are also affected by clock rate variations on both the drive and computer, as well as limited precision in the POKEY divisor and slightly different system clock speeds for NTSC and PAL computers. For these reasons, drive firmwares need to target a bit rate of range of 18866-19040 baud instead of 19200 baud, with considerably better margins than +/-5%.

The byte rate, however, is not as critical since the transfer is asynchronous and the receiver waits for a start bit each time, allowing for arbitrary delays between bytes. If the drive firmware takes longer than a bit period from the stop bit of one byte to the start bit of the next, there is no issue other than the transfer being slowed down slightly. As each byte is transferred as 8 bits (1 start + 8 data + 1 stop), the optimal byte rate is one-tenth the bit rate.

The transfer rates also affect the tone heard when read sector from the disk drive, due to the way that asynchronous receive mode in POKEY works. The per-bit tone is practically inaudible, so the tone is determined by the byte rate. The audio output is toggled 19 times per byte, giving a tone with a frequency at half the byte rate. For the 810, the main receive tone is 943.4Hz, and for the 1050, it is a noticeably lower 910.8Hz.

Step timings

The step timing determines the time taken for the stepper motor to move the head by one step between tracks. Some drives step by full tracks, where one step switches to a new data track, and other drives step by half-tracks, requiring at least two steps for every seek.

Type	Approximate Timing
Atari 810 (rev. B/C/E)	~2640 cycles @ 0.5MHz (5.3ms) per full track
Atari 815	~10260 cycles @ 2MHz (5.2ms) per full track
Atari 1050	~10200 cycles @ 1MHz (10.2ms) per half track
Speedy 1050	~4300 cycles @ 1MHz (4.3ms) per half track
Indus GT	~39650 T-states @ 4MHz (9.9ms) per full track

Table 39: Disk drive step rate timings

Additionally, at the end of each seek a *head settling* time is required to allow the head position to stabilize. This time is the same regardless of the distance seeked.

Format timing

Disk drives vary in their strategy for formatting a disk. The majority of Atari-compatible disk drives use a two-pass strategy where they first format all 40 tracks of a disk from track 0 inward and then verify each track in reverse outward. Some drives vary from this behavior and reverse the direction of the verify or both passes, and some do both formatting and verification together on each track. Table 40 lists the strategy for each drive.

Base drive	Drive type	Method	Behavior		Fill data[62]
			Single-sided	Double-sided	
Atari 810	Atari 810 rev. B/C/E	Overlap	Format in, verify out		$00
	Happy 810		Format in, verify out		$00
	810 Archiver		Format+verify in		$00
	810 Turbo		Format+verify in		$00 SD, $1A DD
Atari 815		Overlap	Format in, verify out		$B6
Atari 1050	Atari 1050 rev. L	Overlap	Format in, verify out		$00
	Happy 1050		Format in, verify out		$00
	Super Archiver		Format+verify in		$00
	Speedy 1050		Format+verify out		$00
	US Doubler		Format in, verify out		$00
	1050 Duplicator		Format in, verify out		$00 SD, $BF ED/DD
	1050 Turbo		Format+verify in		$00
	TOMS 1050		Format+verify in		
	Tygrys 1050		Format in, verify out		$00
Indus GT	Indus GT 1.2	Overlap	Format in, verify in		$00
	TOMS Turbo Drive		Format in, verify in		$00
Atari XF551		Index	Format in, verify in	Format in side 1, format out side 2, verify in side 1, verify out side 2	$00
ATR-8000 v3.20		Index	Format+verify in	Format+verify side 1 then 2 for each track inward	$00
Percom RFD-40S1 Percom AT-88 Percom AT-88SPD		Overlap	Format in, verify out	Format in side 1, format out side 2, verify in side 2, verify out side 1	$1A SD, $92 49 DD
Amdek AMDC-I/II		?	Format+verify in	Format+verify sides 1 then 2 for each track inward	$92 49

Table 40: Disk format strategies by drive type

The majority of drives ignore or cannot sense the index pulse of the drive mechanism and simply format each track starting at the current position. This must be done without a reference for the beginning or end of a track, so the firmware simply writes an overly long track with extra padding at the beginning to ensure that the entire track is formatted, some of which will be overwritten by the end of the track. Using this method also means that each track starts at a different rotational position. Drives that do use the index sensor format index-to-index instead, producing no track skew. However, this tends to take longer since the drive must wait a full revolution to begin formatting the next track.

Track layout

There is also a lot of variation in the layout of a formatted track between drives. All Atari-compatible disk drives use floppy drive controllers that are descendants of the Western Digital FD1771, which imposes some requirements on the layout of the track. However, the firmware is able to vary the content and length of some of the gaps between fields as well as the order of the sectors. A major purpose of the gaps between sectors and at

[62] Data written into the sector during the format process, as seen by the computer when read back with a Read Sector command. For most drives, this will be inverted from the raw data on the disk.

the end of the track is to accommodate variations in drive speed during formatting and when sectors are rewritten. The gap at the end of the track is also longer than the gap between sectors, causing slightly uneven delays between sectors; the tighter the sectors are packed together and the longer the gap at the end of the track, the more sector-to-sector time varies and the more critical of a deadline software has to meet to read sectors at top speed, but also the more tolerant the layout is of a fast drive.

Sectors are also written out of order on the track with an interleave pattern to allow extra time between successive logical sectors, since the drive firmware and computer can’t process each sector quickly enough in physical order. With the SIO disk drive protocol, the time to transfer the sector between the disk drive and the computer over the SIO bus dominates and thus the same interleave factors tend to be used for each format. However, the larger gap at the end of the track means that there are more subtleties to a sector interleave pattern than just an N:1 ordering. This is most apparent in the variations in the interleave used for single density formats, in which some drives manage to drop nearly a full revolution off the track read time with less than a 2% reduction in the minimum delay between sequential sectors.

Table 41 gives the sector layout and ideal timings for each drive type and format. All sector timings are for 2µs MFM / 4µs FM bit cells at 288 RPM for 5.25" formats and 1µs MFM / 2µs FM bit cells at 360 RPM for 8" formats.

Format	Drive type	Sector spacing	Min Delay	Max Delay	Interleave	Sector order	Revs to Read
SD	Atari 810 rev. B	11.072 ms	144.0 ms	153.0 ms	13:1	18,7,14,3,10,17,6,13,2,9,16,5,12,1,8,15,4,11	12.4
	Atari 810 rev. C	11.072 ms	99.6 ms	119.8 ms	9:1	18,1,3,5,7,9,11,13,15,17,2,4,6,8,10,12,14,16	9.0
	Atari 810 rev. E	11.008 ms	99.1 ms	120.3 ms			
	Speedy 1050	11.520 ms	103.7 ms	116.2 ms	9:1	1,3,5,7,9,11,13,15,17,2,4,6,8,10,12,14,16,18	9.0
	Happy 1050	11.200 ms	100.8 ms	118.7 ms			
	Atari 1050 rev. L Atari XF551 Indus GT 1.2	11.072 ms	99.6 ms	119.8 ms			
	ATR-8000 v3.20	10.880 ms	87.0 ms	110.4 ms	9:1	18,16,14,12,10,8,6,4,2,17,15,13,11,9,7,5,3,1	8.2
	Tygrys 1050	11.136 ms	97.0 ms	100.2 ms	9:1	17,15,13,11,9,7,5,3,1,18,16,14,12,10,8,6,4,2	8.1
	Happy 810 810 Archiver 810 Turbo 1050 Duplicator	11.072 ms	97.7 ms	99.6 ms			
	Percom RFD-40S1	10.944 ms	98.5 ms	98.9 ms			
	Super Archiver 1050 Turbo TOMS 1050	10.880 ms	97.9 ms	99.5 ms			
	US Doubler	10.816 ms	97.3 ms	100.1 ms			
	Amdek AMDC-I/II	10.752 ms	96.8 ms	100.8 ms			
SD H/S	Indus GT 1.2 (Sync 1.21)	11.072 ms	64.4 ms	66.4 ms	6:1	16,13,10,7,4,1,17,14,11,8,5,2,18,15,12,9,6,3	5.4
	Indus GT 1.2 (INDUS.SYS)	11.072 ms	53.3 ms	55.3 ms	5:1	4,8,12,16,1,5,9,13,17,2,6,10,14,18,3,7,11,15	4.5
	Indus GT 1.2 (SupSync 1.30)	11.072 ms	42.3 ms	44.3 ms	4:1	5,10,15,1,6,11,16,2,7,12,17,3,8,13,18,4,9,14	3.6
SD 8"	ATR-8000 v3.20	6.016 ms	84.2 ms	100.5 ms	14:1	1,14,3,16,5,18,7,20,9,22,11,24,13, 26,2,15,4,17,6,19,8,21,10,23,12,25	13.4
ED	Atari 1050 rev. L Happy 1050 Super Archiver	7.680 ms	99.8 ms	116.2 ms	13:1	1,3,5,7,9,11,13,15,17,19,21,23,25, 2,4,6,8,10,12,14,16,18,20,22,24,26	13.0
	Speedy 1050	7.424 ms	96.5 ms	119.2 ms			
	US Doubler	7.648 ms	99.4 ms	116.6 ms			
	Atari XF551 Indus GT 1.2	6.848 ms	89.0 ms	126.2 ms			
	1050 Duplicator	6.592 ms	85.7 ms	129.2 ms			
	Amdek AMDC-I/II	6.976 ms	90.7 ms	110.7 ms	13:1	25,23,21,19,17,15,13,11,9,7,5,3,1, 26,24,22,20,18,16,14,12,10,8,6,4,2	12.5
	TOMS 1050 Tygrys 1050	7.680 ms	92.2 ms	100.8 ms	12:1	9,18,7,16,25,5,14,23,3,12,21,1,10, 19,8,17,26,6,15,24,4,13,22,2,11,20	11.2
	1050 Turbo	7.648 ms	91.8 ms	101.3 ms			
DD	Indus GT 1.2	10.944 ms	306.8 ms	329.1 ms	27:1	1,3,5,7,9,11,13,15,17,2,4,6,8,10,12,14,16,18	26.0
	TOMS Turbo Drive	10.880 ms	306.3 ms	329.6 ms			
	Super Archiver	10.848 ms	197.5 ms	197.5 ms	17:1	18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1	16.2
	US Doubler	10.816 ms	197.5 ms	197.5 ms			
	Atari 815	10.752 ms	172.0 ms	197.6 ms	16:1	1,10,9,18,8,17,7,16,6,15,5,14,4,13,3,12,2,11	15.2
	810 Turbo	11.072 ms	175.1 ms	177.2 ms	15:1	6,12,18,5,11,17,4,10,16,3,9,15,2,8,14,1,7,13	14.3
	Speedy 1050	11.520 ms	173.8 ms	184.3 ms			
	Happy 1050	10.976 ms	175.4 ms	175.6 ms			
	Atari XF551	10.944 ms	164.2 ms	186.4 ms	15:1	1,7,13,6,12,18,5,11,17,4,10,16,3,9,15,2,8,14	14.3
	Percom RFD-40S1	10.880 ms	163.2 ms	186.6 ms			
	Amdek AMDC-I/II	10.752 ms	161.3 ms	186.8 ms			
	ATR-8000 v3.20	10.688 ms	160.3 ms	187.0 ms			
	Tygrys 1050	10.976 ms	164.4 ms	164.6 ms	14:1	14,9,4,18,13,8,3,17,12,7,2,16,11,6,1,15,10,5	13.5
	1050 Turbo TOMS 1050	10.880 ms	163.2 ms	164.8 ms			
DD H/S	Atari XF551	10.944 ms	98.5 ms	120.8 ms	9:1	1,3,5,7,9,11,13,15,17,2,4,6,8,10,12,14,16,18	9.0
	Indus GT 1.2 (Sync 1.21)	10.944 ms	109.4 ms	109.8 ms	9:1	2,4,6,8,10,12,14,16,18,1,3,5,7,9,11,13,15,17	9.0
	Indus GT 1.2 (INDUS.SYS)	10.944 ms	76.6 ms	87.9 ms	7:1	1,14,9,4,17,12,7,2,15,10,5,18,13,8,3,16,11,6	6.6
	Indus GT 1.2 (SupSync 1.30)	10.944 ms	65.6 ms	66.0 ms	6:1	16,13,10,7,4,1,17,14,11,8,5,2,18,15,12,9,6,3	5.4
DD 512	TOMS Turbo Drive	20.544 ms	437.2 ms	437.2 ms	19:1	1,2,3,4,5,6,7,8,9	8.9
DD 8"	ATR-8000 v3.20	5.936 ms	136.5 ms	148.9 ms	23:1	1,18,9,26,17,8,25,16,7,24,15,6,23,	22.1

Format	Drive type	Sector spacing	Min Delay	Max Delay	Interleave	Sector order	Revs to Read
						14,5,22,13,4,21,12,3,20,11,2,19,10	

Table 41: Track sector layouts by drive type and format

10.6 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 and its floppy drive controller (FDC) responding to abnormal sectors in specific ways.

Note that because of the inverted FDC data bus on the 810, all FDC status bits are inverted when returned to the computer in a Status command. This means that a ‘0’ bit indicates an error state, not a ‘1’ bit. Table 42 gives values of status byte 1 for various conditions encountered during a sector read command. A (C) at the end indicates that the operation succeeds with a Complete ($43) code instead of Error ($45).

Condition	Status			
	810	1050	XF551	815
Normal operation	$FF (C)		$FF (C)	$00 (C)
Write protected	$BF		$BF	Last Status
Record not found	$EF		$FF	$10 / $18
Data CRC error or weak sector	$F7		$F7	$08
Address CRC error	$E7		$FF	$10 / $18
Deleted sector	$9F	$DF	$DF	$00
Deleted sector with CRC error	$97	$D7	$D7	$00
Long sector	$FC / $FE	$F9	$F9	$10 / $18
Long sector with CRC error	$FC / $FE	$F1	$F1	$10 / $18
Deleted long sector	$9C / $9E	$D9	$D9	$10 / $18
Deleted long sector with CRC error	$9C / $9E	$D1	$D1	$10 / $18

Table 42: FDC status codes for various read sector conditions

For most cases, the drive will both return an (E)rror status and an FDC status code with at least one bit asserted (cleared). In a few specific cases and drives, an error will either be flagged only as Error without an FDC error set, or even go unreported entirely with the operation succeeding with Complete.

In general, the FDC does not retry operations on an error, the one exception being that it will make multiple passes on a track to try to find a valid sector address field. Once it has found a valid address field, it makes only one attempt to read or write the sector. Any retries, and re-steps or recalibrations of the head position, are done by the firmware. Generally only the error from the last attempt is returned.

Record not found (missing address field)

If a matching address field for a sector cannot be found on a track, the FDC 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 a Record Not Found (RNF) error.

The effort made to find the missing sector varies. The FDC is specified to wait up to five rotations, but many drives use a fake index pulse driven by a timer (RIOT) rather than the actual index pulse on the disk.

The XF551 is unusual in not reporting an FDC error for a missing sector. The read operation recalibrates and fails with an error, but the FDC error field in the status packet is $FF instead of $EF.

No disk in drive

A more general issue than a missing sector is a missing disk. The 810 lacks the ability to detect a missing disk or open drive door and will simply attempt requested disk operations. This typically results in the drive in the same behavior as for a missing sector, including bumping the head against the track 0 stop each time, and takes a couple of seconds for the operation to fail.

1050 derived drives are different in that the drive latch does have a hardware sensor and it is tied to the Not Ready input on the FDC. This allows the firmware to detect when the drive door is open and immediately return an error to the computer, without even spinning up or seeking the drive mechanism, returning an error almost immediately for each command. This behavior differs by firmware revision, as some spin up the disk first and others do not.

Disk Drive	Behavior
810 810 Turbo Happy 810 Archiver 815 XF551 Indus GT ATR8000 Percom RFD/AT/ATSPD Amdek AMDC-I/II	Turns on motor and attempts to read with retries before returning Error + RNF.
1050 (rev. H, J) 1050 Duplicator	Turns on motor and returns Error + Not Ready without seeking.
1050 (rev. K, L) Happy 1050 Super Archiver Tygrys 1050	Returns Error + Not Ready without turning on motor.
US Doubler I.S. Plate	Returns NAK without turning on motor.
Speedy 1050	Boots copy software from controller ROM if initial D1: boot, otherwise returns Error + Not Ready without turning on motor.
1050 Turbo 1050 Turbo II	Boots copy software from controller ROM.

Table 43: Disk drive behavior on Read Sector command with drive door/latch open and no disk inserted

Data CRC error

The data field 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 CRC Error.

Data CRC errors only occur for reads since for writes the CRC is generated and written instead of being read and checked. The CRC Error bit can still be asserted on an address CRC error, however.

Address CRC error

The address field of each sector is also protected by a CRC. If the FDC is attempting to find a sector and the only address fields it can find with that sector number all have CRC errors, both the CRC and Record Not Found (RNF) status bits will be set (bits 4 and 5). If duplicate sectors are present and one of the sectors has a valid address field, however, that sector will be used and the CRC Error bit will not be set.

The XF551 does not report either an RNF or CRC error for this condition. The read operation fails, but the FDC error reported is $FF.

Deleted/user 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, as well as $F9 and $FA for user type records. While the FDC considers this normal and reads the sector successfully, the firmware considers it an error and will retry the read. Upon failing out, it will send back the sector's data with an Error status.

FDC models, and therefore drives, differ in the status raised for a deleted or user type sector. 810 drives use a 1771 controller, which returns the two record type bits in bits 5 and 6. A deleted sector will therefore return FDC status to the computer with bits 5 and 6 cleared. 1050 drives, however, use a 279X controller that only returns the LSB of the record type in bit 5, with bit 6 set for read operations. This means that an 810 will return $9F for a deleted sector while a 1050 will return $DF. It also means that the two user data types appear the same as normal and deleted data marks and are indistinguishable to either the computer or the drive firmware.

Deleted/user sector errors can only occur during reads. When writing a sector, the FDC switches from reading to writing after the address field and writes a new data field along with a new DAM; any old data field and DAM is never seen by the FDC, if it even existed.

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, and more importantly, the sector size is recorded in the address field, but not in the command registers. The length of the transfer is determined solely by the size encoded in the address field. This provides an opportunity for the FDC and firmware to disagree on sector size.

Normally, the 810 firmware stops when the FDC asserts an interrupt, but due to a quirk in the firmware, it will also stop after reading 129 bytes and then immediately read the FDC status. Since the FDC is still reading the sector, the busy bit (bit 0) is guaranteed to be asserted, and the DRQ bit (bit 1) may also be asserted if another byte has been decoded in time. The lost data bit (bit 2) won't be asserted, though, because not enough time will have passed to lose a byte, and the CRC error bit (bit 3) won't be asserted because the FDC will have cleared it at the start of the command and not verified the CRC yet. Therefore, the FDC status will read either $FE or $FC. As usual, the first 128 bytes of the sector are returned along with the Error status.

The 1050 firmware differs here in that it waits for the FDC to complete its operation before reading status, while still not reading the pending data. This means that the Lost Data bit will be asserted, BUSY negated, and CRC Error state updated.

Not all drives report long sectors as errors. The firmware for some drives reads all bytes in the physical sector instead of using the logical sector size, meaning that they will read long sectors successfully without error. The Indus GT and Percom RFD-40S1 are two drives that do this.

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. As with other error types, the read data will be returned even with the CRC error, allowing the weak sector to be detected.

Phantom (duplicated) sectors

Multiple sectors within the same track can have duplicate sector IDs, in which case any of them may be found by the FDC. Since the FDC will return the first instance of the sector it finds, the phantom sector found depends on when the read command is issued and how long it takes for the firmware to issue the corresponding FDC command. 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 if the sectors also have errors.

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 C firmware at 500KHz, with a drive mechanism running exactly at 288 RPM, no seeks required, and motor already on are given in Table 44.

Event	Time	Rotation
Send command at 19200 baud	2.63 ms	4.54°
Deassert command line	?	?
Command line deasserted to ACK byte sent	0.59 ms	1.02°
ACK byte sent to FDC command issued	3.22 ms	5.56°
Rotational delay	0-208.3 ms	0-360°
Read physical sector	9.66 ms	16.69°
Compute checksum and return sector data	74.06 ms	127.98°

Table 44: Ideal 810 sector read timing

Not counting additional delays for host-side processing, it takes a minimum of 90.16 ms (155.8°) 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.

10.7 6532 RIOT

Interval

Writing to register addresses $4-7 or $C-F starts the interval timer, which counts down from a specified initial delay. It is useful both for measuring time intervals and waiting for a specified time delay and can be used either in interrupt or polled mode.

The value written to the register determines the initial count, while address bits A1-A0 of the register used to write to the timer selects the rate for the prescaler, which divides down the system clock rate to a lower rate for the timer counter. The prescaler can be set to divide by 1, 8, 64, or 1024. Note that the last divisor is not evenly spaced.

The prescaler is reset along with the counter, so it always outputs pulses at a predictable offset from the timer write. For a written count of N and a prescaler interval of P, the timer delay to underflow/interrupt is N*P + 1 cycles.

Timer interrupts

The interval timer can be configured to generate an interrupt every time its counter underflows. The timer IRQ is enabled by address bit 3 of accesses to read or write the timer. However, whether the timer IRQ is enabled does not affect the timer interrupt flag read by register $5, which will still be set or reset even if the IRQ is disabled.

The timer interrupt flag, and IRQ if enabled, are reset whenever the counter is read or written.

Timer underflow

When the timer counter underflows, it sets the timer interrupt flag and then wraps around to $FF and begins counting down every cycle at 1T rate instead of at the rate specified by the prescaler. This will continue indefinitely, with the counter repeatedly underflowing every 256 counts and setting the interrupt flag again if it has been reset in the meantime.

Reading the timer

Reading register $4 or $C will read the current counter value of the interval timer. This value counts down from the initial value until it underflows and then continues counting down indefinitely. When the timer underflows and the timer interrupt flag is set, the counter switches to counting down at 1T rate regardless of the prescaler setting. Thus, after underflow, the counter indicates the negated number of cycles passed since underflow, not intervals.

There are two side effects of reading the timer value. The first is that the timer interrupt flag is cleared, unless an underflow happens to occur on the same cycle as the read. This happens regardless of the timer IRQ enable bit (A3). The second effect is that clearing the interrupt flag also resets counter operation from cycles back to intervals. Thus, after reading an 8T counter that has underflowed and begun decrementing at 1T, the counter will resume decrementing once every 8T.

Timer counter timing

The prescaler is reset on writes to the timer so that it emits a pulse on the next cycle. Thus, the counter always decrements on the next cycle after the write to the timer.

Once the timer underflows and the timer flag is set, it immediately begins counting down at 1T rate, e.g.: 00 FF FE FD.

Prescaler timing is not affected by the switch to 1T or back. When a read occurs that resets the counter back to interval timing, it will next decrement at a multiple of 1T/8T/64T/1024T from when the counter first decremented, regardless of the offset at which the timer read occurred. This can cause an irregular delay from when the counter stops decrementing at 1T and next decrements at prescaler interval rate.

10.8 177X/179X/279X FDC

Almost all Atari-compatible disk drives use a derivative of the Western Digital FD1771 floppy drive controller chip, the original chip used in the 810. This chip is only capable of FM (single density) encoding, so any drive that supports MFM encoding for enhanced or double density has a newer 179X/279X or 1772 chip, which supports both FM and MFM modes.

1793/2793 and 1797/2797 compatibility

MFM capable drives with multiple firmware revisions sometimes have compatibility issues between the 1793/2793 and 1797/2797 chips. The reason is differing interpretation of command bits for Read Sector and

Write Sector commands depending on the capability of each chip. 1793/2793 chips use bits 1 and 3 of the read/write sector command to control side compare logic, while the 1797/2797 use it to control external side select signals.

Unfortunately, the two interpretations are not compatible, even for a single-sided drive that has no use for double-sided support. Early revisions of the 1050 firmware use a $82 command to read sectors, which on the 2793 enables side compare for side 0, but for the 2797 uses an alternate interpretation of the sector size field. Revision L of the firmware use $88 instead, which fixes the 2797 by disabling side compare on the 2793.

Record Not Found (RNF) timeout

Most FDC models return a Record Not Found (RNF) error when a sector header is not found after five revolutions of the disk. The 1771, however, only requires two revolutions.

Step rates

The FDC has four different options for step rate when executing seek commands, specified by the two low bits of the seek command.

Model	00	01	10	11	Notes
1770	6 ms	12 ms	20 ms	30 ms	At 8MHz; scale by 96% for 8.3MHz (XF551)
1771	3 ms	6 ms	10 ms	15 ms	At 2MHz; double for 1MHz (810)
1772	2 ms	3 ms	5 ms	6 ms	At 8MHz; scale by 96% for 8.3MHz (XF551)
279X	3 ms	6 ms	10 ms	15 ms	At 2MHz; double for 1MHz (1050)

Table 45: Step rates for various FDC models

It is important to note that these are not the step rates used by most disk drives, which expose stepper phases directly to the CPU and control step rate by wait loops in the firmware. This includes the 810 and the 1050, which actually step at 5.2 ms and 13 ms/track instead. The XF551, however, does use the FDC stepping circuitry and step rates.

Head load delays

Read and write commands have an option to add a fixed delay to allow the head to load onto the disk surface. This option is commonly enabled by drive firmware. However, the specific delay varies between different chip models. It also varies by the clock rate, which is important since many of the FDC chips are clocked slower than normal for 5.25" drives.

Chip	Specification	Actual delay in drive
1770	30 ms (8MHz)	28.8 ms at 8.3MHz (XF551)
1771	10 ms (2MHz)	20 ms at 1MHz (810)
1772	15 ms (8MHz)[63]	14.4 ms at 8.3MHz (XF551)
279X	15 ms (2MHz)	30 ms at 1MHz (1050)

Table 46: Head load delays for various FDC models

[63] Older 1770/1772 datasheets specify either 30ms for both chips or 15ms for the 1772. The newer WD177X-00 datasheet which also includes the 1773 clarifies the timing being different between the two.

Write Track

Waiting for index pulse

Write Track commands always write a complete track from index pulse to index pulse. The FDC pauses after waiting for the head to load and for the CPU to load the first byte before waiting for the leading edge of the index pulse. This is important since most Atari-compatible drives use a fake index pulse under CPU control. The FDC must see the index pulse signal deasserted and then asserted while it is waiting for the index pulse. If the IP signal is already deasserted or asserted and stays that way, the FDC will wait indefinitely. This is used by some drive firmware to hold off the FDC until the firmware is ready to begin writing bytes for the formatted track.

Writing CRC bytes

The FDC will automatically compute CRC-16 values for ID and data fields while writing tracks. The FDC initializes its CRC logic when writing an ID address mark ($FE token) or data address mark ($F8-FB token) in FM, or an $A1 sync mark in MFM ($F5 token). That byte and all following bytes are included in the CRC computation until an $F7 token is encountered, upon which the CRC value is written out as two consecutive bytes. Because two bytes are written, the delay to the next DRQ is twice as long.

Except when reset by $F8-FB/FE in FM or $F5 in MFM, the CRC logic is always running and updating with every byte written out to the disk. This includes the $F7 CRC token, for which the two CRC values are themselves included in the running CRC and can be reflected in a later CRC write. However, the FDC cannot write two CRC values back-to-back. When two consecutive $F7 tokens are written back-to-back, the FDC writes the second one as a regular $F7 data byte with normal one-byte timing. The sequence FB F7 F7 F7 thus writes to disk the data bytes FB BF 84 F7 9F F8 with clock bytes C7 FF FF FF FF FF, where $BF84 is the CRC-16 of [FB] and $9FF8 is the CRC-16 of [FB BF 84 F7].

Read Track

Initialization

Head load and index pulse synchronization for Read Track are the same as for Write Track.

Sync mark behavior

When executing Read Track, the FDC returns only data bits, discarding clock bits. This prevents the CPU from accurately distinguishing sync bytes from non-sync bytes, as $FE may be both a valid data byte or an IDAM byte in FM mode.

The FDC does maintain byte synchronization from the last DAM or IDAM. Whenever a special sync mark is encountered, the FDC immediately returns it regardless of how many bits have been shifted into the shift register since the last byte returned. This means that the previous returned byte is likely to have some overlapping bits with the address mark and the DRQ for the address mark can arrive sooner than usual. For instance, a flux pattern with clock bits $C7FFFC7 and data bits $FEA00FE is returned as $FE A0 0F FE, because the second $FE IDAM is recognized and returned four bits after the previous $0F byte.

10.9 810 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
  - Bit 0 = 0: Busy
- 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.

Due to a bug in the firmware, the 810 accepts $4F as an alias for $57.

Format ($21)

The format command ('!' = $21) command formats a disk, writing 40 tracks and then verifying all sectors. All sectors are fill with the data byte $00. On completion, the drive returns a 128 byte buffer containing a list of 16-bit bad sector numbers, terminated by $FFFF.

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.

Firmware revisions

There are three known revisions of the 810 firmware, revisions B, C, and E. The most common version is revision C. All firmware ROMs are 2K in size.

Revision	CRC32
Revision B	19227D33
Revision C	0896F03D
Revision E	AAD220F4

Table 47: 810 drive firmware revisions

Revision B

Revision B is a rarer, earlier version of the 810 firmware, differing from revision C only in the sector order used when formatting a disk. Revision B used a slower sector order with a minimum spacing of 13 sectors, whereas C reduces this to the typical 9 sector distance.

Revision E changes

The revision E ROM adds a few new commands to the 810 over rev. C, specifically ones that allow execution of custom code on the drive.

$20 (execute code)

Command $20 takes a 128 byte data frame sent from the computer and executes it on the 810 at address $80. The firmware sends the command and data frame ACKs before calling the custom routine, and expects that routine to return with RTS. Afterward, the firmware sends the Complete signal back to the computer.

$80-FF and $180-1CF are available for use by the custom routine, the former for temporary storage during the routine's execution, and the latter for permanent storage (not used by the firmware at all).

$4F

The bug in revision C that allowed $4F as an alias for $57 (write sector) has been fixed; rev. E rejects $4F with a NAK.

$54 (transfer memory)

Reads 128 bytes of memory starting at the address specified by AUX1 and AUX2, where AUX1 is the LSB of the address, and sends that buffer back to the computer.

$FE and $FF are trashed by this command and cannot be read back.

10.10 810 hardware

While a standard 810 does not allow custom code execution, ones with revision E or other modified firmware do, and in those cases it becomes necessary to know how to access the 810 hardware.

CPU

The 810 uses a 6507 CPU running at 0.5MHz (500KHz). The 6507 only has 13 address bits and can only access 8K of memory; this is the same memory model as a 6502 with A13-A15 ignored and thus all memory ranges are mirrored eight times in the 64K address space. The 6507's memory map is as follows:

Address range	Mapping
$0800-0FFF	ROM (2K)
$0700-07FF	RIOT registers (mirror)
$0680-06FF	6810 memory (mirror)
$0600-067F	1771 FDC (mirror)
$0500-05FF	RIOT memory (mirror)
$0480-04FF	6810 memory (mirror)
$0400-047F	1771 FDC (mirror)

Address range	Mapping
$0380-03FF	RIOT registers
$0300-037F	RIOT registers (mirror)
$0280-02FF	6810 memory (mirror)
$0200-027F	1771 FDC (mirror)
$0180-01FF	RIOT memory
$0100-017F	RIOT memory (mirror)
$0080-00FF	6810 memory
$0000-007F	1771 FDC

Table 48: 810 memory map

A12 is not used for any decoding, and thus in a stock 810 all address ranges are mirrored 16 times at every 4K. There is no DMA or memory refresh in the 810, so all memory cycles are available for the CPU and the 6507 runs at a constant speed.

The 6507 also has no interrupts, either IRQ or NMI. This means that the NMI vector is used and the IRQ vector is only used for BRK instructions, which can be executed without fear of interference with interrupts.

1771 Floppy Drive Controller

The floppy drive controller used in the 810 is a Western Digital FD1771 or equivalent, which only handles single density. For speed it is mapped in the lower half of page zero, with the canonical range being $00-03. The 1771 is run at 1MHz timings and is missing several of the usual connections to the drive hardware.

First, none of the head stepping and loading signals are hooked up. The stepper motor phases are controlled through RIOT instead of the FDC, the head is always loaded, and there is no track 0 sensor. The seek commands on the FDC are not used for seeking the head and the FDC is simply told which track it should use for comparisons against the track field in sector address fields.

Second, no index pulse sensor is hooked up either. Instead, the index pulse input to the FDC is driven by the IRQ signal from the RIOT, traditionally asserted via the RIOT's timer. This places write track and sector search timeouts under control of the firmware and allows the 810 to read and write sectors across the index position. The 810 in fact has no way of even determining the position of the index on the disk.

An additional quirk is that the 1771 FDC has an inverted data bus. This requires all register read and writes to the FDC to be inverted by the CPU, and is also responsible for the standard of inverted data on Atari floppy disks.

6810 RAM

The 6810 provides 128 bytes of memory at $80-FF, used mainly by the 6507 as the transfer buffer for FDC reads/writes and data frame transmission with the computer. This memory is also mirrored at $180-1FF so that its upper portion can hold the stack.

ROM

2K of ROM provides the firmware for the drive. Its canonical address range is $0800-0FFF, although $F800-FFFF is also required in order for interrupt vectors to be read.

6532 RAM I/O Timer (RIOT)

The RIOT chip provides a majority of the support functions in the 810, providing memory for the stack and working variables, timing facilities, and control of all the miscellaneous signals. The canonical ranges are $0180-

01FF for the memory and $0380-039F for the control registers.

The timer facility is mainly used to drive the index pulse facility on the FDC, but does not need to match the actual rotational timing of a floppy disk and is typically not set to do so; instead, it is used to force the FDC to time out if a sector is not found on a track or the FDC would otherwise continue indefinitely with a command. This can also be done with the PA7 edge detection feature since both are connected to a common IRQ output, but this usage is less common (some versions of the Happy diagnostic do so).

The 810 firmware relies on an undocumented quirk of the RIOT timer in that once the initial delay has expired and the timer switches to 1T timing, it will continuously underflow and reassert the timer IRQ every 256 cycles. This is used by the firmware to force a series of quick "index pulses" after a multi-rotational delay of about half a second has expired, quickly making the FDC think that five revolutions have passed. For this to work, the firmware needs to repeatedly read the timer in order to deassert the timer IRQ after each revolution.

The RIOT's I/O ports are connected to a wide variety of signals:

Port	Signal	Direction	Usage	Polarity
Port A	PA7	Input	FDC DRQ	1 = DRQ asserted
	PA6	Input	FDC INTRQ	1 = INTRQ asserted
	PA5	Input	Jumper	
	PA4	Input	Write protect sensor	1 = write protected
	PA3	Output	ROM bank switch (Archiver only)	
	PA2	Input	Drive code A sense[64]	1 = D1/D2, 0 = D3/D4
	PA1	Output	Spindle motor control	1 = enabled
	PA0	Input	Drive code B sense	1 = D1/D4, 0 = D2/D3
Port B	PB7	Input	SIO data output (computer to device)[65]	inverted
	PB6	Input	SIO command	1 = asserted
	PB5	Output	Stepper phase 4	1 = activated
	PB4	Output	Stepper phase 3	1 = activated
	PB3	Output	Stepper phase 2	1 = activated
	PB2	Output	Stepper phase 1	1 = activated
	PB1	Input	SIO ready	1 = ready
	PB0	Output	SIO data input (device to computer)	non-inverted

Table 49: 810 RIOT I/O port assignments

Track stepping

As previously noted, the 810 does not use the FDC to step the head between tracks, and instead drives the stepper phases using PB2-PB5 on the RIOT. One reason for this is that the 810's drive mechanism uses a 4-phase stepper motor instead of 3 phases as expected by the FDC. Two adjacent phases are activated at the same time whenever the head is being stepped or held in position during a read or write operation; all phases are turned off when the drive goes idle. A shift of one phase in the stepper motor steps the head by one track.

Because no track 0 sensor is hooked up, the 810 cannot determine when the head is over track 0 and instead simply steps the head outward a bit more than 40 tracks to recalibrate. This causes the head to bump against the track 0 stop slightly. Phases 1 and 4 are active when the head is at track 0, i.e. %1001 for PB2-PB5;

[64] Note that the drive codes are encoded using a Gray code, not binary.
[65] The SIO data lines are named from the computer side, so the SIO data input line is an output for the 810.

stepping inward to higher numbered tracks requires turning on successive phases in a pattern of %0011 for track 1, %0110 for track 2, etc.

10.11 Happy 810

The Happy 810 is an 810 disk drive modified with a larger, custom firmware ROM and an additional 3K of RAM. Custom code can be uploaded to the drive, allowing new functions to be added to the firmware.

Memory map

The Happy 810 maps 3K of RAM at $0800-13FF and 3K of ROM to $1400-1FFF. The firmware is stored in a 4K ROM, but the bottom 1K of it is inaccessible to the drive and can only be read in an external ROM reader. The top 2K of the firmware at $1800-1FFF is a modified version of the 810 rev. C firmware, relocated from $0800-0FFF and patched. Many routines, such as the transfer routines, are in analogous positions.

Address range	Mapping
$1400-1FFF	ROM (3K)
$0800-13FF	RAM (3K)
$0700-07FF	RIOT registers (mirror)
$0680-06FF	6810 memory (mirror)
$0600-067F	1771 FDC (mirror)
$0500-05FF	RIOT memory (mirror)
$0480-04FF	6810 memory (mirror)
$0400-047F	1771 FDC (mirror)
$0380-03FF	RIOT registers
$0300-037F	RIOT registers (mirror)
$0280-02FF	6810 memory (mirror)
$0200-027F	1771 FDC (mirror)
$0180-01FF	RIOT memory
$0100-017F	RIOT memory (mirror)
$0080-00FF	6810 memory
$0000-007F	1771 FDC

Table 50: Happy 810 memory map

In the revision 7 version of the firmware, the firmware was extended to 6K by bank switching. Banks are selected by reading or writing $1FF8 and $1FF9. The bottom 1K of each bank is inaccessible. The revision 7 firmware adds write buffering and a number of Happy 1050 compatible commands to the drive.

Memory read/write commands

No new commands are added in the Happy 810. The only added functionality are memory read and write sub-commands, which can then be used to patch into the command handler.

To access RAM, a sector read or write command is issued with a “sector” in the range of $0800-1380. 128 bytes of RAM at that address are then read or written to the drive according to the standard command protocol. An attempt to read or write any other address fails with a command NAK. For writes, command $57 must be used, not command $50.

Track buffering

The Happy 810 buffers tracks by default when reading sectors.

Command handler patch

It is possible to upload a patch at $0800 to inject custom commands into the command handler. This is composed of three sections:

- $0800-0808: Custom command signature. These bytes must be exactly: $26 11 34 14 15 57 37 85 86. They have no meaning other than as a signature.
- $0809-0820: Dispatch addresses for commands $50-57, as a table of JMP instructions.
- $0821+: Memory available for custom command implementation.

Because the drive does not preinitialize these tables, successfully uploading a custom command table to the drive typically requires knowledge of addresses within the firmware. A relocated and slightly patched version of the standard 810 rev. C firmware resides at $1800-1FFF in the bank visible at command dispatch. All standard command entry points, as well as utility functions for sending and receiving data, are in the same locations with a $1000 offset.

Fast/slow switch (rev. 7 only)

The rev. 7 firmware supports a fast/slow switch by means of bits 3 and 5 on port A of the RIOT.[66] If PA3 and PA5 are connected together, allowing PA3 to drive PA5, then the firmware interprets this as a signal to disable track buffering and run in slow mode.

High speed commands (rev. 7 only)

The built-in firmware supports high-speed versions of the read, write, and put sector commands as $72, $77, and $70, respectively. The command frame and command ACK/NAK are sent at low speed, while the completion/error and data frame are sent at 38,400 baud ($10 divisor).

Quiet command (rev. 7 only)

Command $51 (‘Q’) turns off the drive motors immediately. AUX1/2 are ignored.

Happy command (rev. 7 only)

Command $48 (‘H’) is a command with multiple functions, multiplexed by AUX1.

Set idle timeout (AUX1=$01)

Sub-command $01 sets the inactivity timeout for the drive. AUX2 sets the timeout in units of about 50ms.

Set alternate device ID (AUX1=$02)

Sub-command $02 changes the alternate device ID for the drive. Bits 0-6 of AUX2 set the alternate device ID, while bit 7 determines whether the drive only responds to the alternate ID. For instance, for a drive normally configured for D1: by hardware switches, $34 would map it to both D1: and D4:, and $B4 would map the drive to D4: only. In the latter case, the drive would no longer respond to D1:.

The change to the device ID does not take place immediately, but is instead queued for the next disk change.

[66] The 810 mod is documented in [Happy] p. 31.

Reinitialize drive (AUX1=$03)

Sub-command $03 reinitializes the drive by performing a software reset. The Complete byte is actually sent before the reset instead of after, as the drive forgets all state when the reset occurs; this means that the drive will be uncommunicative for a short time afterward.

Configure drive (AUX1=$04-FF)

Sub-commands $04-FF reconfigure the Happy-specific functionality of the drive. The bits in AUX1 determine the mode bits to change, while the bits in AUX2 are the new states for those mode bits. Table 51 lists the functions of the individual mode bits.

Bit	Function (if set)
Bit 6 ($40)	Disable track read/write buffering
Bit 5 ($20)	Disable track write buffering
Bit 2 ($04)	Enables alternate device ID

Table 51: Happy 810 rev.7 drive mode bits

Thus, AUX1=$58, AUX2=$40 would disable track buffering and restore normal write protect sensing.

Changes to all but the track buffering modes – bit 2 – take effect immediately. Track buffering and write buffering can be disabled immediately, but read buffering can only be enabled if it has already been initialized, and write buffering cannot be enabled immediately; otherwise, these changes take effect on the next disk change.

Unlike the Happy 1050, the Happy 810 does not have the ability to go into “unhappy mode” by command. Instead, the Happy Warp menu uploads custom dispatch code to the drive to disable extended commands.

Multi-drive broadcast ($36-38) (rev. 7 only)

Commands $36-38 are used to broadcast data from the computer to multiple Happy 810 drives simultaneously. The 3K of Happy RAM is divided into 24 sectors of 128 bytes each, which are also tracked by 24 byte flags. The commands allow the computer to send the same sector data to all drives in one transmission, verify that the drives received the data, and then trigger all drives to write the data to disk. The firmware only provides the broadcast commands; custom code must be uploaded to the drive to actually use the data. All commands use high speed for the data frame and result portions of the protocol (38400 baud).

Command $38 is used to broadcast to multiple drives that are set to use the same alternate device ID. It is unique in not sending any response back from the drive, avoiding conflicts between the multiple drives receiving the same command. The command receives a 128 byte data frame, which is uploaded to drive RAM at the address specified in AUX1/2, within $0800-13FF. The corresponding flag is also set to indicate that the sector was received successfully.

Commands $36 and $37 are used to verify reception. $36 clears all sector flags and then returns Complete; $38 reads back the sector flags as a 24-byte frame. Each byte is $FF for a successfully received sector, or $00 if the sector was not received or had a checksum error.

10.12 810 Turbo

The 810 Turbo Double Density Conversion Board by Neanderthal Computer Things (NCT) is an expansion that uniquely adds double-density support to the 810, providing true double-density operation.

Hardware

To support MFM operation, the 810 Turbo replaces more hardware than usual. The 6507 CPU is retained but raised in clock rate from 500KHz to 1MHz, the FD1771 floppy drive controller is replaced by a 2791, and 4K of static RAM is added. To support this, A12 is now also decoded. Table 52 shows the memory map with the new mapping in green.

Address range	Mapping
\$1800-1FFF	ROM (2K)
\$0800-17FF	RAM (4K)
\$0700-07FF	RIOT registers (mirror)
\$0680-06FF	6810 memory (mirror)
\$0600-067F	279X FDC (mirror)
\$0500-05FF	RIOT memory (mirror)
\$0480-04FF	6810 memory (mirror)
\$0400-047F	279X FDC (mirror)
\$0380-03FF	RIOT registers
\$0300-037F	RIOT registers (mirror)
\$0280-02FF	6810 memory (mirror)
\$0200-027F	279X FDC (mirror)
\$0180-01FF	RIOT memory
\$0100-017F	RIOT memory (mirror)
\$0080-00FF	6810 memory
\$0000-007F	279X FDC

Table 52: 810 Turbo memory map

The 2791 floppy drive controller has an inverted data bus, like the 1771 in the 810 and unlike the 2793/2797 in the 1050.

The official V1.2 firmware is 2K in size with a CRC32 of 0126A511.

Supported feature set

In addition to the standard set of 810 commands, the 810 Turbo supports the Read/Write PERCOM Block commands and a special memory access command.

While the hardware should support it, the firmware does not support medium (enhanced) density. It can only read and write single or double density disks.

Density switching

The 810 Turbo only detects density on the first read/write sector command to a disk and retains that density mode thereafter. It will not switch to another density unless either the computer is power-cycled, which re-enables density detection, or the density setting is overridden by a Write PERCOM Block command. The density check relies on the standard timeout for a record not found (RNF) error, which leads to a long delay on the initial access. The firmware continues toggling density modes until retries are exhausted or the operation succeeds.

Status command

Double-density disks are reported with bit 5 set in the first status byte, like other drives.

The format timeout for the 810 Turbo is \$FF.

PERCOM block support

For Read PERCOM Block, the firmware returns a step rate of 3, present flag of $01, and zero for unused bytes. The remaining bytes are set as expected for standard SD and DD formats.

The Write PERCOM Block command is very minimally implemented: it fails with Error on an attempt to set a double-sided format and then checks if the density byte is non-zero for SD/DD. Everything else in the PERCOM block is ignored and unvalidated.

Memory access command ($20)

Access to drive memory and code execution is allowed through a single custom command, $20. Bits 0-5 of AUX2 provide A8-A13 of the address while AUX1 provides A0-A7. Bit 6 of AUX2 specifies the transfer direction, 0 for writing to drive memory and 1 for reading from it. 256 bytes are always transferred.

For write operations, setting bit 7 also causes the firmware to execute the code at the given load address. RTS returns to the service loop. In V1.2 firmware, $0800-08FF is used by the firmware for buffering command and data frames, but otherwise the whole 4K RAM is uncommitted and available for use. Addresses $009A-00BD contain the command table as triplets of command ID, handler low byte, handler high byte.

10.13 815 disk drive

The Atari 815 is a dual-mechanism double density disk drive, with support for both a top drive (drive 1) and bottom drive (drive 2). Actual working units exist but in very low numbers, as reportedly as low as 50 hand-assembled units were shipped before the drive was canceled.

Format support

The 815 *only* supports double density disks, with 256 bytes per sector. The hardware and firmware are unable to use FM encoding, and thus cannot support single density. Theoretically the hardware could support enhanced density as it simply uses a 128 byte sector size in MFM instead of 256 bytes, but the firmware has no support for it. The drive therefore could not be used with DOS 2.0S and required DOS 2.0D instead, which was the same as 2.0S except for being reconfigured to use 256 byte sectors.

There is also some provision in the hardware and firmware for double-sided and 80 track operation, but both are incomplete: the firmware lacks track/sector translation support for side 2 or 80 tracks, and reportedly 815 units were shipped without parts on the circuit board for the top side.

Support for the double-sided format is standard: 40 tracks with 18 sectors per track and 256 bytes per sector, using IBM format. The first three boot sectors on track 0 are encoded on disk as 256 bytes but read or written by the computer as 128 bytes, and the status command returns the first byte with bit 5 set to indicate double density. However, 815 disks are not interoperable with other double-density disk drives, because the 815 does not store sector data inverted as other drives do. Other drives can read disks written by an 815, but all data will be read inverted.

Commands

In addition to the standard status, read sector, and write/put sector commands, the 815 also supports some unique commands that were enumerated in the Atari Home Computer Operating System User's Manual, but never supported by any other disk drive. Although the Operating System Manual does not describe the details of these commands, they are specified in detail in the 815 Dual Disk Drive Operator's Manual [815].

Download ($20)

This is a write command that takes a 128 byte data frame and stores it in $0000-007F, which is then called at $0000. The routine is expected to RTS back to the firmware with four status bytes, the first three of which are returned in A, X, and Y. The fourth byte is stored on the stack immediately above the return address at 3,S and can be modified by the called routine.

> **Note**
>
> The 815 Dual Disk Drive Operator's Manual specifies that this command accepts a 256 byte data frame, but the actual firmware only accepts a 128 byte frame.

Spin test ($51)

Seeks to the track containing the sector number in AUX1/2 and then exits back to the main loop with the spindle motor still running, until the next command is issued.

Status ($53)

This command reads the same four byte status frame on the 815 as on other disk drives, with a couple of important differences.

First, bit 6 is always set on the 815. This signifies double-sided operation on some other drives like the XF551, but the 815 only supports single-sided operation. However, this bit is not generally reliable anyway as disk drives usually do not try to detect whether a disk is double-sided.

A more serious issue is that the 815 sets the second byte to *non-inverted* FDC status, versus other drives reporting inverted status. For example, a successful sector read will set the second byte to $00, and a missing sector will set it to $10.

Read address ($54)

Intended to read memory from the address at AUX1/2. Unfortunately, the firmware implementation is broken and due to a register conflict it always returns 256 bytes of firmware ROM starting at between $1100-11FF.

Motor delay ($55)

Extends the default motor idle timeout from 3 seconds to 30 seconds.

Verify sector ($56)

Same as write sector ($57), except that the actual disk write is skipped so that the data frame is only verified against the sector on disk.

Extended info ($58)

Returns a 128 byte fixed descriptor of unknown format from the firmware ROM.

High-speed operation

Despite some references to 31.5Kbaud operation in publicly released source code listings of the firmware, there is no support in the firmware for high-speed operation. The listings imply that this was intended to be added in the 2K ROM expansion that was never shipped; the base 4K firmware only contains support for high-speed dispatch hooks that are never activated and cannot be used (they jump to existing ROM locations used for other purposes).

Motor timeout

Unusually, not only does the 815 support two disk drives, it also supports independent motor idle timeouts for each of them: after accessing D1: and D2: in quick succession, both motors will be on, and then D1: will turn off shortly before D2: turns off. However, since the motor idle logic can only run in the main loop, both motors will keep running as long as a disk operation is in progress.

Error behavior

The 815 lacks either a disk ready latch or a track 0 sensor. It will still attempt to read even if no disk is in the drive or the head is unlatched, and when recalibrating bump the head against the track 0 stop.

The firmware has severe bugs in the handling of the FDC status in the second status byte. The most serious bug is that it uses the wrong polarity and doesn't emulate the inverted status of the 810, which means that the 815 always reports inverted status from all other drives, including a no-error value of $00 instead of $FF. It also fails to consistently clear old status and will leak old CRC errors along with new Record Not Found (RNF) errors, sometimes producing $18 codes instead of $10.

Several error conditions also are reported differently by the 815 due its lack of FDC and the way it handles address and data fields. It precomputes and searches for the entire address field rather than just a matching sector number, so address field CRC errors and long sectors are never matched and are either reported as RNF or another sector with the same ID is used instead. Sectors with non-zero side fields are also reported as RNF. Sectors with user-defined or deleted data address marks of $F8-FA are reported as errors without reading the sector or setting any FDC error bits, so data is not returned as some protection code expects and the record type bits are never set. Finally, a write protect error results in an Error result but leaves the existing FDC status unchanged.

Read/write latency

Despite its fast 2MHz CPU, the 815 has higher latency for reading and writing double density sectors than other double density capable drives. The primary culprit is the lack of FDC requiring the 6507 to compute the CRC-16 in software, which is particularly slow for the data field. This is done through shift-and-add in the firmware due to lack of space for table lookup, and takes 16.6ms for 256 bytes. On other drives, this is computed by the FDC during the read or write operation with no additional delay.

10.14 815 hardware

The 815's hardware is unusually powerful for an Atari 8-bit disk drive: 2MHz 6507, three 6532 RIOTs with 384 bytes of memory, an MC6852 Synchronous Serial Data Adapter, and 4K of ROM. However, one important thing it is missing is a floppy disk controller -- the job that would be normally done by a 177X/179X/279X is instead done by custom logic and firmware.

Memory map

Address range	Mapping
$1800-1FFF	High ROM mirror
$1000-17FF	High ROM (2K)

Address range	Mapping
$0800-0FFF	Low ROM (2K)
$0400-07FF	Not mapped
$0380-03FF	RIOT 1 registers
$0300-037F	Not mapped
$0280-02FF	RIOT 2 registers
$0200-027F	RIOT 3 registers
$0180-01FF	RIOT 1 memory
$0100-017F	SSDA
$0080-00FF	RIOT 2 memory
$0000-007F	RIOT 3 memory

Table 53: 815 memory map

The only currently known and verified version of the 815 firmware has CRC32s of 32BD3CFD for the A107 low ROM at $0800-0FFF and C13657E0 for the high ROM at $1000-17FF/$1800-1FFF.

Both the $1000-17FF and $1800-1FFF mirrors of the high ROM are necessary for operation. The firmware executes with $1000-17FF as the canonical address space, but the second mirror is needed for the 6507 to fetch the reset vector to start at $0800 in the low ROM.

RIOT

The 815 relies heavily on 6532 RIOT chips for the majority of its I/O. Table 54 lists the assignments. The $\overline{\text{IRQ}}$ output is not used on any RIOT. The RIOT timers are used via polling, but only for timeouts; without an FDC there is no need for index pulse simulation in the 815.

Port	Signal	Direction	Usage	Polarity
RIOT 1 Port A	PA7	Input	Sync status	1 = partially synced
	PA6	Output	Clock reset	0 = reset
	PA5	Output	SSDA /CTS reset output	
	PA4	Input	Drive 1 write protect	1 = protect
	PA3	Output	SSDA master reset	
	PA2	Output	Not used	
	PA1	Output	Drive 1 motor/LED enable	1 = on
	PA0	Input	Drive 2 write protect	1 = protected
RIOT 1 Port B	PB7	Input	SIO data output (computer to device)	inverted
	PB6	Input	SIO command	1 = command
	PB5	Output	Drive 1 stepper phase 1	0 = activated
	PB4	Output	Drive 1 stepper phase 2	0 = activated
	PB3	Output	Drive 1 stepper phase 3	0 = activated
	PB2	Output	Drive 1 stepper phase 4	0 = activated
	PB1	Input	SIO ready	0 = ready
	PB0	Output	SIO data input (device to computer)	non-inverted
RIOT 2 Port A	PA7	Output	Drive 2 write LED	1 = on
	PA6	Output	Drive 2 read LED	1 = on
	PA5	Output	Not used	
	PA4	Output	Drive 2 bottom head read enable	0 = enabled
	PA3	Output	Drive 2 top head read enable	0 = enabled

Port	Signal	Direction	Usage	Polarity
	PA2	Output	Not used	
	PA1	Output	Drive 1 bottom head read enable	0 = enabled
	PA0	Output	Drive 1 top head read enable	0 = enabled
RIOT 2 Port B	PB7	Output	Drive 2 bottom erase head enable	0 = enabled
	PB6	Output	Drive 2 bottom write head enable	0 = enabled
	PB5	Output	Drive 2 top erase head enable	0 = enabled
	PB4	Output	Drive 2 top write head enable	0 = enabled
	PB3	Output	Drive 1 bottom erase head enable	0 = enabled
	PB2	Output	Drive 1 bottom write head enable	0 = enabled
	PB1	Output	Drive 1 top erase head enable	0 = enabled
	PB0	Output	Drive 1 top write head enable	0 = enabled
RIOT 3 Port A	PA7	Input	Starting drive ID select high	1 = D1:/D3:, 0 = D5:/D7:
	PA6	Input	Not used	
	PA5	Input	Not used	
	PA4	Output	Drive 1 write LED	1 = on
	PA3	Output	Drive 1 read LED	1 = on
	PA2	Input	40/80 track jumper (not used)	
	PA1	Output	Drive 2 motor/LED enable	1 = on
	PA0	Input	Starting drive ID select low	1 = D1:/D5:, 0 = D3:/D7:
RIOT 3 Port B	PB7	Input	Drive mechanism type jumper (not used)	
	PB6	Input	Double-sided jumper (not used)	
	PB5	Output	Drive 2 stepper phase 1	0 = activated
	PB4	Output	Drive 2 stepper phase 2	0 = activated
	PB3	Output	Drive 2 stepper phase 3	0 = activated
	PB2	Output	Drive 2 stepper phase 4	0 = activated
	PB1	Input	Extended ROM space jumper (not used)	
	PB0	Output	Write precompensation	0 = enabled

Table 54: 815 RIOT I/O port assignments

Head stepping

There is no track 0 sensor on the 815, so the firmware must bump the head against the stop to confirm track 0.

Head positioning is driven by a four phase stepper motor connected to PB2-PB5 on RIOT 1 (drive 1) or RIOT 3 (drive 2). It is driven in the same way as the 810, with two phases being activated at once and phases 1 and 4 active at track 0.

Head control

RIOT 2 drives the enable signals for the read, write, and erase heads on each drive. The write and erase heads are used to write to the disk; the read head is used for both read and write operations to search for a matching address field before reading or writing the data field.

Although the firmware can sense the write protect state and must do so to return a proper error to the computer, there is additionally a hardware interlock to lock out the write and erase heads if the disk is write protected. This prevents the firmware from writing to a write protected disk even if it fails to detect the WP state.

SSDA and data separation

An MC6852 Synchronous Serial Data Adapter (SSDA) is used for data serialization and synchronization, allowing the 6507 to work on a per-byte basis rather than per-bit. When reading, the SSDA detects MFM sync bytes for byte synchronization and then buffers data with its three-byte receive FIFO; when writing, the SSDA serializes encoded MFM data buffered with its three-byte transmit FIFO.

The data separator circuit has two modes, clock+data and data-only. When starting a sector operation, the data separator is reset to run in clock+data mode and drives the SSDA receive clock at one bit per 2µs, returning both MFM clock and data bits in one interleaved stream. At the same time, the SSDA is set to scan the incoming bitstream for the initial $44 byte of the $4489 MFM sync word in two-byte sync mode. When $44 is detected, the firmware reconfigures the SSDA to check for the second $89 byte. In both cases, the SSDA is also set to issue pulses to the data separator whenever a sync match occurs. If both tests pass and two pulses are received, the data separator automatically switches to half speed to only return data bits and the firmware turns off the sync output. Otherwise, the firmware resets both the SSDA and the data separator to search for $44 again. PA6 and PA7 on RIOT 1 are used to control the data separator clock rate.

The SSDA's two-byte sync mechanism results in some limitations when scanning for MFM sync. In one-byte mode, the SSDA simply scans the incoming bitstream until the entire sync pattern is found. In two-byte mode, however, it must match one byte, wait for the firmware to switch the sync value, and then match the second byte. This means that when a false match occurs on the first sync byte where the second sync byte will fail, the SSDA is unable to properly detect a correct sync word starting within 1-16 bits later as it will not have been reconfigured in time with the first sync byte. In the 815's case, the drive is unable to sync to a pattern like $44489 since it is too late to begin syncing with the second $44 once the firmware realizes that the first $44 is bogus. This situation is avoided in the 815 by formatting with a sequence of $00 data bytes prior to the sync marks, as they encode to $AAAA and cannot be read as a $44 prefix.

During transfers, the SSDA is configured in two byte transfer mode so that it only sets the receive or transmit FIFO ready bits when two bytes can be transferred. This allows the firmware to read or write two bytes at a time without checking the FIFO again in between, important as the 6507 only has 32 cycles for each byte.

For transmission, the SSDA always transmits at the MFM bit cell rate of 500KHz. There is no hardware support for MFM encoding and so this must be done in the firmware.

Because the SSDA normally shifts bits from LSB-to-MSB order while disks use MSB-to-LSB order, the 815 attaches the SSDA to the data bus with reversed bit order. Correspondingly, all status register reads and control register writes use bit-reversed byte values.

10.15 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 true double density with 256 byte MFM sectors.

Disk sense

Unlike the 810, the 1050 is able to sense when a disk is inserted since the drive latch is connected to the floppy drive controller's ready input. FDC status bit 7 reflects this status, and the firmware also uses this to determine when a disk change has occurred. When a disk is inserted and the latch is closed, the drive seeks to track 1 and issues the READ ADDRESS command to detect whether the disk is using FM (single density) or MFM (double density) encoding.

Different revisions of the 1050 firmware react differently to attempts to read or write sectors without a disk or with the mechanism unlatched. Revisions H and J of the firmware turn on the motor before returning an error, while revisions K and L return an error without turning on the motor.

Status command

The status command in the 1050 returns one additional bit of information: drive status 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).

The write protect bits (bit 4 of drive status and bit 6 of FDC status) are updated every time the status command is issued, even if no disk drive mechanism activity occurs. Even if the latch is open and the drive motor is off, the status command will still reflect the change in the write protect sense as disks are inserted and removed.

When no disk is inserted, bit 7 of the FDC status byte will be cleared to indicate a not ready condition.

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.

Testing commands

The 1050 has an extensive number of undocumented testing modes used by the 1050 diagnostics software to test the drive. They are triggered by command $23, which is a write command that takes a 128 byte buffer with a subcommand in the first byte and sometimes a parameter in the second byte. For sub-commands $00 and $01, a second command $24 is needed, a read command that executes the test and reads back the result as another 128 byte buffer.

The $23 and $24 testing commands are not very well supported by other drives, even 1050 drives with modified firmware. No other Atari drive supports them, and in modified 1050 firmware, these commands are often either not implemented or reduced to save on ROM space.

Command $24 has another side effect -- when invoked with any sub-command other than $01, it also returns ROM signature bytes in the last three bytes of the returned buffer. The intended three bytes are revision (ATASCII), and the model number as binary big endian (1050). However, due to an addressing bug in rev. H and rev. J ROMs, the returned values are off by one byte in those versions. This is fixed in rev. K and L.

	Byte 125	**Byte 126**	**Byte 127**
Rev. H	$AA	$48	$04
Rev. J	$AA	$4A	$04
Rev. K	$4B	$04	$1A
Rev. L	$4C	$04	$1A

Table 55: 1050 returned ROM signatures

Subcommand $00 (RPM test)

Measures the speed of the spindle motor by measuring the time between successive reads of sector 1 on the current track. Command $24 is needed afterward to begin the test and read back the results. If successful, the period between the two reads is stored in units of 100us in the first word of the returned sector buffer.

Subcommand $01 (Motor start test)

Seeks to track 20, turns off the spindle motor for 5 seconds and lets the disk come to a complete stop, then spins up the motor and measures the time until the FDC can read a valid address field. Command $24 is required afterward to actually execute the test and read back the results. The first word of the returned buffer contains approximately the number of milliseconds needed for the spin-up, and the last six bytes of the buffer have the address field that was successfully read, if any.

Subcommand $02 (Head step/settle test)

Steps through all 40 tracks on disk, reading sector 1 of each track and then reading the address field of the next sector in physical order, verifying that it reads successfully and that the sector number is equal to or lower than the expected max. Sector 0 is allowed despite not being valid. The highest sector number expected is given in the second data frame byte.

Subcommand $03 (Step half track)

Steps the head by a *half*-track in either direction. The second byte of the data frame selects the direction -- $00 for outward to track 0, $01 for inward to track 39.[67]

Subcommand $04 (Double-density burn-in test)

Runs a lengthy read/write test with MFM enabled.

Subcommand $05 (Seek to track)

Seeks the head to the track number given by the second data frame byte. Error is returned if the track number is invalid (above 39).

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 inward to higher-numbered tracks than when seeking outward. When seeking inward, the 1050 does an additional half-track step inward, followed by a half-track step outward. This ensures that the 1050 always finishes a seek with an outward step, for better consistency in head positioning.

Formatted track layout

Medium density disks are formatted by 1050 with a 13:1 interleave, separating even and odd sectors within each track. There is about a two sector skew between tracks such that track 2, sector 1 has approximately the same angular position as track 1, sector 5.

[67] The stock 1050 firmware checks bit 7 for this, but the 1050 Turbo checks for zero/non-zero.

Long sector handling

Unlike the 810, which reads FDC status immediately after transferring 128 data bytes from the FDC during read sector commands, the 1050 always waits for the command to complete before reading drive status. This leads to different FDC status values when reading long sectors of 256 bytes or more. In particular, the DRQ and lost data bits will be asserted due to the additional data being ignored by the firmware, the BUSY flag will be negated, and the CRC error bit will accurately reflect where the CRC was correct on the entire sector, regardless of the data bytes that were dropped. This generally produces FDC status values of either $F9 for sectors with correct CRCs and $F1 for sectors with bad CRCs.

User record types

Due to differences in the FDC chips, the 1050 also differs in behavior from the 810 in handling of two of the four record types that can be encoded into a sector data field. Specifically, the 1771 in the 810 reports both of the record type bits in bits 5-6 of the FDC status register, while the 279X in the 1050 only reports the LSB in bit 5. This means that the 1050 is unable to distinguish the $F9 and $FA user record types from the normal $FB and deleted $F8 record types, and more importantly, returns an FDC status of $DF instead of $9F for $F8 deleted records.

Firmware revisions

Revision E (CRC32 F9A7AFB2)

Earliest known publicly released firmware.

Revision H (CRC32 6D9D589B)

Next known publicly released firmware after rev. E, with a few compatibility fixes and many reliability fixes.

There is now a timeout in the SIO receive code when waiting for a start bit, so the drive doesn't hang indefinitely.

Rev. E checked the AUX1/2 bytes for a valid sector number even for commands that don't take a sector number. In Rev. H, these bytes are now ignored for those commands.

Read/write sector commands for sector 0 are now trapped and rejected. In rev. E, attempting to read or write sector 0 results in an attempt to seek to track 56.

The Format command fills sectors with $00 instead of $E5, as seen by the computer ($FF or $1A on disk).

Fixes the Status command to return cached FDC status from the last read/write sector operation and only update from the FDC on a subsequent Status command. In rev. E, the Status command always returns the current type I status.

The sector buffer is no longer cleared after every command in rev. H. This reduces command latency and fixes a compatibility issue compared to the 810 when a Read command completes without having read a data frame from the disk.

In Rev. E, the double half-step in and then out to force a final outward step when seeking was suppressed for track 39. In rev. H, this is also done for track 39 for consistency.

Rev. H now enables side compare for read/write sector operations, unlike rev. E which has side compare disabled. (This was later re-disabled in rev. L.)

Revision J (CRC32 91BA303D)

Next publicly released firmware after Revision H. Besides bumping the revision byte, it has a minor adjustment to the SIO receive routine to improve timing margin for PAL.

The OS approximates 19200 baud as 94 cycles/byte when sending, which at the 1050's 1MHz translates to 52.5 cycles for NTSC and 53.0 cycles for PAL. In Revision H, the receive loop takes 51 cycles/bit; Revision J adjusts this to 52 cycles/bit to improve timing margins. This particularly helps with PAL, for which the timing error in Revision H is almost 40% of a bit cell by the stop bit.

Revision K (CRC32 3ABE7EF4)

Fixes a bug in the ROM checksum routine that prevented the computed checksum from actually being verified.

Improves the receive timing for PAL further, by adding another cycle to the receive routine (53 cycles/bit). This is a wash for NTSC, which is about 10% off at the end either way.

Adds an early check for disk ready before initiating read/write sector commands. Revision J spins up the spindle motor before returning an error, whereas revision K returns an error without turning on the mechanism.

The diagnostic double-density write test (command $23 subcommand $04) has been modified to let the motors idle for ~36 seconds between each phase of the test.

Command $24 was fixed to properly return the intended three signature bytes (revision, model ID lo/hi). In rev J, the location was off by one byte and a garbage byte was returned ($AA 4A 04). In rev K, the addressing was corrected and $4B 04 1A is returned.

Revision L (CRC32 FB4B8757)

Revision L changes the FDC command bytes used by the firmware for Read Sector and Write Sector commands from $A2/$82 to $A8/$88 for compatibility with the 2797 FDC, which interprets the pertinent bits for alternate sector length instead of side compare. Previous firmware revisions only work with the 2793, whereas Revision L can use either the 2793 or 2797.

10.16 1050 hardware

Like the 810, the 1050 also has no command to upload code to the drive. However, a large number of enhancements were made for the 1050 which do, and writing custom code for the drive requires knowing how to interface with the hardware.

Controller

The controller in the 1050 is a 6507 running at 1MHz. Its memory map is as follows (along with canonical ranges):

Address range	Mapping
$F000-FFFF	ROM (4K)
$0C00-0FFF	279X FDC (mirror)
$0B80-0BFF	RIOT registers (mirror)
$0B00-0B7F	Not mapped
$0A80-0AFF	RIOT registers (mirror)
$0A00-0A7F	Not mapped
$0980-09FF	RIOT RAM (mirror)
$0900-097F	6810 RAM (mirror)

Address range	Mapping
$0880-08FF	RIOT RAM (mirror)
$0800-087F	6810 RAM (mirror)
$0400-07FF	279X FDC
$0380-03FF	RIOT registers (mirror)
$0300-037F	Not mapped
$0280-02FF	RIOT registers
$0200-027F	Not mapped
$0180-01FF	RIOT RAM
$0100-017F	6810 RAM (mirror)
$0080-00FF	RIOT RAM (mirror)
$0000-007F	6810 RAM

Table 56: 1050 memory map

Due to the 6507's limited address bus, all memory mappings are repeated every 8K in the address space. The set of memory mapped devices is similar to the 810, with the notable exception that the FDC is no longer mapped in page zero. Instead, a single full page of RAM is mirrored to both page zero and the stack page.

FDC

A WD2793/2797 acts as the floppy drive controller for the drive. It differs mainly from the FD1771 used in the 810 by supporting MFM encoding. Like the 810, the FDC's head stepping facilities are not used and the index pulse is supplied from the RIOT timer instead of the drive. However, notable differences are that the drive latch is connected to the ready input on the FDC, so both the FDC and the firmware can detect when the drive door is open, and a track 0 sensor is also hooked up.

The 2793/2797 FDC also has a non-inverted data bus instead of the 1771's inverted bus. The firmware must therefore invert data on read or write for compatibility with the 810.

RIOT

Port	Signal	Direction	Usage	Polarity
Port A	PA7	Input	FDC DRQ	1 = DRQ asserted
	PA6	Input[68]	RIOT IRQ / index pulse sense	
	PA5	Output	FDC density control	1 = FM, 0 = MFM
	PA4	Output	FDC write precompensation enable	1 = precomp on
	PA3	Output	Spindle motor control	0 = enabled
	PA2	Output	ROM bank switch (Super Archiver only)	
	PA1	Input	Drive code A sense[69]	1 = D1/D2, 0 = D3/D4
	PA0	Input	Drive code B sense	1 = D1/D4, 0 = D2/D3
Port B	PB7	Input	SIO command	1 = asserted
	PB6	Input	SIO data output (computer to device)	inverted
	PB5	Output	Stepper phase 1	0 = activated
	PB4	Output	Stepper phase 2	0 = activated
	PB3	Output	Stepper phase 3	0 = activated

[68] Normally input, but some Happy 1050 software drives PA6 in output mode to directly inject index pulses.
[69] As with the 810, the 1050 encodes the drive ID in Gray code instead of binary.

Port	Signal	Direction	Usage	Polarity
	PB2	Output	Stepper phase 4	0 = activated
	PB1	Input	SIO ready	0 = ready
	PB0	Output	SIO data input (device to computer)[70]	non-inverted

Table 57: 1050 RIOT I/O port assignments

Track stepping

The read/write head in the 1050 is stepped between tracks by port B bits 2-5 like in the 810, but with some important differences. The outputs are pulled low instead of high to activate phases, only one phase is active at a time instead of two, and each change in phase steps by a half-track instead of a full track. In terms of values written to port B, a right shift steps inward to higher number tracks, and a left shift steps outward to lower numbered tracks.

Track 0 sensor

Oddly, the track 0 sensor is hooked up inverted so that the FDC senses a track 0 condition when the head is not on track 0 and vice versa. This is no issue for the FDC since its seek facilities are not used, but it does mean that the firmware has to interpret the Track 00 bit in FDC status oppositely from its intended interpretation.

The track 0 sensor in the 1050 also activates for track 0.5 as well as track 0.

10.17 US Doubler

The US Doubler is a hardware modification to the 1050 drive that adds high-speed and double density support. It consists of a replacement 4K firmware and an additional 128 bytes of memory.

Hardware changes

The only hardware change made by the US Doubler is to add another 128 bytes of memory next to the RIOT registers, canonically addressed at $0200-027F and mirrored at $0300-037F. This increases the total memory to 384 bytes and allows use of 256 byte sectors without convoluted programming tricks.

Known official firmware is CRC32 605B7153, marked as © 1985 ICD, Inc. US Doubler rev. L on the chip.[71]

High-speed operation

Communication with the US Doubler can be run at high-speed to greatly decrease loading times. The first step is to query the drive's high-speed index with command $3F, which returns a single-byte data frame with the POKEY divisor, which is always $0A for the US Doubler. Sending a command at this high speed then causes the drive to use this speed for all communications for that command.

The firmware is only capable of receiving commands at one speed at a time, so it handles speed switching by toggling high-speed for the command frame receive whenever a command frame receive error occurs. This means that the drive ignores the first command sent after a speed switch and relies on the computer to retry the command without a response.

Since the US Doubler lacks a track buffer, effective high-speed operation depends on the disk being formatted with a suitable high-speed skew. Using high-speed communication with a disk formatted with standard sector

[70] Note that the SIO data lines are named from the computer side, so the SIO data input line is an output for the 810.
[71] https://atariage.com/forums/topic/156462-1050-roms/?do=findComment&comment=4116994

order doesn't give an improvement. Similarly, using standard speed with a high-speed skew formatted disk gives bad read performance.

Format skewed command ($66)

By default, the US Doubler uses a standard speed skew for the $21 and $22 format commands. For high-speed operation, it is necessary to format disks with a high-speed sector skew instead. This is done with command $66. This command is a write command that takes a 128 byte data frame containing the PERCOM block parameters in the first 12 bytes, followed by the sector list. The sector list contains 1-based sector numbers.

The format skewed command is flexible with its support of PERCOM block parameters. In particular, custom track and sector per track counts are supported, allowing partial disk formatting and missing/duplicated sectors.

Write PERCOM block command ($4E)

PERCOM blocks set using command $4E are partially parsed by the US Doubler. Bytes 0 (track count), 3 (sectors per track), 5 (density), and 7 (bytes per sector low) are used; all other bytes are ignored.

10.18 Happy 1050

The Happy 1050 adds a 6502 CPU, 6K of RAM, and an additional 4K of ROM to the 1050. Additional features include track buffering, custom code upload, and high-speed operation. It is, however, not compatible with the Happy 810 as the two use different command sets and require different code uploads.

Memory map

The Happy 1050 replaces the 6507 with a 6502, expanding the address space from 8K to 64K. This is used to map 6K of RAM at $8000-97FF and a 4K ROM window at $F000-FFFF. The ROM is bank-switched, exposing 8K of ROM as two 4K banks switched by an access to $FFF8 or $FFF9.

Two official revisions of the firmware are known to exist, both 8K in size. Revision 1 has a CRC32 of 19B6BFE5, while revision 2's CRC32 is F76EAE16.

Memory read/write commands

As with the Happy 810, the read and write sector commands have been extended to allow memory access. A sector address with bit 15 set reads or writes to that address on the drive. Unlike the Happy 810, both put/write commands work and the start address can be anywhere in $8000-FFFF, not just in RAM. Also, the transfer size depends on the current density setting, so 256 bytes are transferred if the current drive setting is double density.

High speed commands

The built-in firmware supports high-speed versions of the read, write, and put sector commands as $72, $77, and $70, respectively. The command frame and command ACK/NAK are sent at low speed, while the completion/error and data frame are sent at 38,400 baud ($10 divisor).

Revision 2 firmware is also capable of using US Doubler compatible high speed protocols. Like the US Doubler, it uses a POKEY divisor of $0A. Unlike other drives using this protocol, support for this is not enabled until the Get High Speed Index ($3F) command is received, upon which the firmware enables the USD-compatible high speed code.

USD high-speed corruption bug

The revision 2 support for US Doubler style high speed has a critical bug: in the default power-up configuration, all sector writes will have byte 2 (the third byte) corrupted. This is due to the USD high speed single-byte transmit routine stomping part of the sector staging buffer at memory location $0002 and its mirror $0102 prior to writing the sector. A typical value is $82, left over from shifting out $41 (ACK). Enabling write buffering bypasses the problem. However, this is disabled by default, making the drive dangerous to use with a high-speed OS that does not have a workaround.

Track read buffering

By default, the Happy 1050 has track read buffering enabled to speed up read operations. Instead of reading each sector as requested by the computer, the drive reads all sectors on a track in the order stored on disk, and then uses the buffer to satisfy read sector commands without issuing commands to the floppy disk controller. This greatly reduces rotational delays and speeds up sequential reads on a single density disk by about 25%. It is even possible for the drive to idle the disk mechanism and stop the motors while still responding to read sector commands, as long as the requested sectors are all in the buffer.

Whenever a sector is requested on a track not in the buffer, the firmware seeks to and reads the new track. It first issues a Read Address command to read the sector ID of the next sector on the disk, and then begins reading sectors into the buffer after that. Buffering a track thus takes slightly more than a revolution.

In order to do this, the firmware needs to know the sector order beforehand, which it does so by scanning and recording the sector order from track 0 via Read Address commands when a new disk is inserted, which is then used for all tracks. The firmware detects a disk change by polling for changes in the state of the Not Ready status bit on the FDC, which is connected to the 1050’s drive lever. It can only do so when the drive is idle, however, so it is possible for a disk change to be missed if the change occurs very quickly while a command is being processed.

Track read buffering is automatically disabled if the Controller is present and the Fast/Slow switch is set to Slow. It can be re-enabled on the fly by changing the switch back to Fast, upon which the firmware will re-scan the sector order and resume buffered reads.

Track read buffering error handling

Track read buffering supports most types of sector errors, caching the FDC error code along with each sector read. The main exception is duplicate/phantom sectors, which cannot be supported in the track buffer; only one of the sectors is cached.

As errors may also occur transiently, the track buffering logic attempts to re-read sectors that were not read successfully. This is done in entire passes over the track. For instance, if all sectors but 2, 3, and 7 read correctly, the firmware will then attempt to read only those three sectors on the next pass. The exception is deleted sectors, which are considered successful reads and are not re-read. This process continues over and over as long as unsuccessful sectors remain, with a quick double half step head reposition after two passes, until an ultimate timeout of about six seconds.

The firmware tracks the last sector read and will pass back buffered errors whenever a new sector is requested. If a sector is re-read, however, it will activate the disk mechanism for another pass at re-reading all remaining errored sectors on the currently read buffered track. Unfortunately, for a permanent sector error, this re-read pass continually runs until the track buffering read timeout expires. The result is that any sector with a permanent error takes a full six seconds to read when track buffering is enabled. This is substantially longer than a stock 1050 drive, which only takes about half a second to complete all retries for most types of read errors.

Sector 1 buffering

When track buffering is enabled, the firmware also maintains a special side cache for track 0, sector 1, independently of the normal track buffering cache. This allows the drive to service requests for sector 1 without invalidating the track buffer and re-reading all of track 0. In particular, this speeds up accesses by SpartaDOS, which frequently re-reads sector 1 to check if the disk volume has been changed.

Track write buffering

The Happy 1050 has the ability to buffer writes as well as reads. Up to an entire track can be buffered, which is then written out in a single rotation to disk after enough time has passed or the buffer needs to be reused for another track. Both write and put commands are cached by the write buffer.

When a write sector command is issued for a different track, new track is not immediately read into the track buffer; only the written sector is stored. This avoids unnecessary sector reads for sectors that will be overwritten. The track buffer remains sparse until either the track is flushed to disk or a read occurs. If a read occurs, a full track read occurs and the missing slots in the track buffer are filled in. This occurs even if the sector that is read is already resident in the track buffer as pending for write.

The write buffer is also automatically flushed on any recognized command that is not a read, write, or put sector command or the Happy command ($48). All modified sectors in the write buffer and written back to the disk before the command is executed.

When writing modified sectors back to disk, the drive first does a Read Address command to determine the current position of the head in sector order, and then begins writing modified sectors starting with the next sector. As long as the disk has a consistent sector order between all tracks, this allows the drive to write all sectors in a single revolution regardless of sector interleave. A verify pass is always done afterward by re-reading the sectors, regardless of whether the sectors were written with a write or put sector command. Only sectors that were written are re-read.

Track 0, sector 1 is only partially write buffered and cannot handle writes by itself. If the write buffer is not currently holding track 0, writing to sector 1 will cause the write buffer to be flushed and reassigned to track 0. However, once track 0 is write buffered, any writes to sector 1 will also be written to the sector 1 buffer, where it can be re-read without reassigning the track read buffer.

Track read/write buffering

In single density with 6K of RAM, the Happy 1050 is capable of buffering different two tracks for read and write, and can alternate between reading from one and writing to another without having to flush or seek in between. It cannot buffer two read or two write tracks, though.

Read-after-write behavior on the same track with dual track buffers is still similar to with a single track buffer. Reading a sector that is pending write will still cause the read track buffer to be invalidated and reloaded if it doesn't already contain that track, even though the data is available from and will be read from the write track buffer anyway.

For enhanced and double density disks, the firmware drops to single track buffering.

1050 diagnostic commands (rev. 1 only)

Revision 1 firmware supports the $23 and $24 1050 diagnostic commands. These commands were removed and will return a NAK on rev. 2 firmware.

Quiet command

Command $51 (‘Q’) turns off the drive motors immediately. AUX1/2 are ignored. As with other non-sector commands, the Quiet command flushes buffered write data back to the disk before turning off the motors.

Happy command

Command $48 (‘H’) is a command with multiple functions, multiplexed by AUX1.

All sub-commands flush track buffers before performing their function.

Warning

Command $48 flushes the write buffer without writing modified sectors back to disk, so any modified sectors still left in the track write buffer are lost! This can be avoided by flushing the write buffer with a status command ($53) first.

Set idle timeout (AUX1=$01)

Sub-command $01 sets the inactivity timeout for the drive. AUX2 sets the timeout in units of about 50ms.

Set alternate device ID (AUX1=$02)

Sub-command $02 changes the alternate device ID for the drive. Bits 0-6 of AUX2 set the alternate device ID, while bit 7 determines whether the drive only responds to the alternate ID. For instance, for a drive normally configured for D1: by hardware switches, $34 would map it to both D1: and D4:, and $B4 would map the drive to D4: only. In the latter case, the drive would no longer respond to D1:.

The change to the device ID does not take place immediately, but is instead queued for the next disk change.

Reinitialize drive (AUX1=$03)

Sub-command $03 reinitializes the drive by performing a software reset. The Complete byte is actually sent before the reset instead of after, as the drive forgets all state when the reset occurs; this means that the drive will be uncommunicative for a short time afterward.

Configure drive (AUX1=$04-FF)

Sub-commands $04-FF reconfigure the Happy-specific functionality of the drive. The bits in AUX1 determine the mode bits to change, while the bits in AUX2 are the new states for those mode bits. Table 58 lists the functions of the individual mode bits.

Bit	Function (if set)
Bit 7 ($80)	Unhappy mode (disable all extended functions)
Bit 6 ($40)	Disable track read/write buffering
Bit 5 ($20)	Disable track write buffering
Bit 4 ($10)	Force write protected
Bit 3 ($08)	Force write enabled
Bit 2 ($04)	Enables alternate device ID

Table 58: Happy 1050 drive mode bits

Thus, AUX1=$80, AUX2=$80 would set Unhappy Mode, while AUX1=$58, AUX2=$40 would disable track buffering and restore normal write protect sensing.

If mode bits 3 and 4 are set at the same time, the write protect state is sensed normally, the same as if both functions were disabled.

Changes to all but the track buffering modes – bits 2-4 and 7 – take effect immediately. Track buffering and write buffering can be disabled immediately, but read buffering can only be enabled if it has already been initialized, and write buffering cannot be enabled immediately; otherwise, these changes take effect on the next disk change.

Fast/slow switch

Two hardware switches are added to drives that have the Controller Option. The fast/slow switch is used to toggle the track buffering feature of the drive, which speeds up disk access but is incompatible with some disks. When this switch is set to the Slow setting, accesses to $9800-9FFF or $B800-BFFF toggle a flip-flop that is connected to the Set Overflow line on the controller 6502. This means that every other access sets the V flag on the CPU. If the switch is set to Fast or the Controller is not present, these accesses are ignored. There is no preset state for this flip-flop and thus the firmware must always issue at least two accesses.

Write protect switch

The second switch on the Controller is a three-way switch to control the write protect setting on the disk. The three settings are Normal, Protect, and Write.

In the Normal setting, the write protect sensor of the drive mechanism is modified by an inversion flip-flop before being passed to the FDC. An access to $4000-7FFF toggles the flip-flop, switching between inverting and not inverting the state of the sensor. An access to the fast/slow switch ranges ($9800-9FFF and $B800-BFFF) clears this flip-flop, restoring normal non-inverted sensing. There is no way to directly set the flip-flop, but the firmware can achieve any required configuration by a combination of clear/toggle accesses and checking the FDC's write protect state.

In the Protect and Write settings, the Write Protect state seen by the FDC is overridden to either always on or off, regardless of the disk and flip-flop state. The firmware can still clear or toggle the flip-flop, but cannot affect the WP signal. It is therefore not possible for the firmware to write to the disk when the switch is set to Protect.

10.19 I.S. Plate

The I.S. Plate – also referred to in the official documentation as the I.S 1050 Plate Enhancement, Innovated Software's Plate, ISP Plate, ISP PLATE, and 1050 ISP – is a single-board modification to 1050 drives to add revised firmware and additional RAM. It replaces the 6507 CPU with a 6502, the 4K firmware with 8K firmware, and adds 16K of static RAM.

Memory map

A12 in the 6507's 13-bit address space is replaced by the 6502's A15, which causes $0000-7FFF to reflect $0000-0FFF in the original memory map eight times, containing all of the hardware and excluding the firmware ROM. The new 8K of firmware ROM is at $E000-FFFF, though only 4K at $F000-FFFF is used. The 16K of RAM is split into two 8K halves, one at $8000-9FFF and the other at $C000-DFFF.

High-speed operation

US-Doubler style high speed operation is invoked by sending command frames at 52Kbaud (divisor $0A).

10.20 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	Description
Bit 7	0 = single or double density 1 = enhanced density
Bit 6	1 = double sided
Bit 5	0 = single or enhanced density 1 = double density
Bit 4	Always 0
Bit 3	1 = last put operation failed due to write protect
Bit 2	1 = last put operation failed
Bit 1	1 = receive error on last data frame
Bit 0	1 = receive error on last command frame

Table 59: XF551 drive status flags

The XF551 does not indicate motor status via bit 4.

The timeout field in the returned status is set to $FE on the XF551 instead of $E0.

Format command

The standard format command ($21) is extended on the XF551 to also format disks in double density and DSDD formats, depending on the current mode set by either automatic density detection or the Write PERCOM Block command. However, it will not format disks in enhanced/medium density, to preserve compatibility with the 1050; when the current density is enhanced density, command $21 will change the current mode to single density before formatting the disk.

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.

Format self-test

All format commands have a special mode in XF551s with rev 7.7 firmware: invoking a format command with AUX1=$10 and AUX2=$50 causes the drive to format the disk over and over in a loop. This continues forever until an error occurs, at which time the drive exits to waiting for a new command without sending a reply to the computer. Drives with rev 7.4 firmware do not have this feature.

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:

		Format			
Data	**Index**	**SD**	**ED**	**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 60: XF551 PERCOM configuration block values

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).

Command line timing

Most disk drives are lenient with or even ignore timing for releasing /COMMAND after the command frame is sent. The XF551 is unusually strict in this regard and will reject any command frame for which /COMMAND is not still asserted at the end of the command frame. Therefore, it is essential to wait for the last byte to finish transmitting completely before deasserting /COMMAND.

No disk behavior

The XF551 has no latch sensor and uses a WD1772 FDC that lacks a Ready signal input, so it will repeatedly restep and retry on accesses with no disk present.

Density switching problem

Unfortunately, the XF551 firmware has problems with reliably detecting double density disks and can sometimes fail to read such disks after a disk change. The firmware will automatically switch from FM to MFM, but defaults to enhanced (medium) density first, and relies on a long sector error to indicate when it should switch to double density. If the logical sector number of the next request maps in enhanced density to a physical sector with number greater than 18, the FDC will not be able to find a sector to read and the firmware is unable to trigger a density change on the resulting Record Not Found error.

Compounding this problem is an additional quirk in the firmware, which is that boot sectors 1-3 are exempted from the long sector check. This prevents the XF551 from switching densities properly when a double density MyDOS disk is used with SpartaDOS X, since SDX first reads sector 1 to detect the filesystem type and check for a volume change before reading the directory at sector 361; sector 1 causes a change only to ED and then attempting to read 361 causes a RNF on track 13, sector 23. Some XF551-oriented DOSes deliberately issue a dummy read to sector 4 to force a density change to double density.

Firmware versions

There are two confirmed stock versions of the XF551 firmware, revision 7.4 [CRC32 FC02766B] and revision 7.7 [38B97AE3]. The main difference between rev 7.4 and rev 7.7 are tweaks to the timing of the receive loops from 28 machine cycles/bit to 29 machine cycles/bit. This changes the sampled baud rate from 19,841 baud to 19,157 baud, increasing reliability for NTSC and especially for PAL, which is why rev 7.7 is sometimes known as the PAL fix version.

Rev 7.7 also has another change, which is the addition of the format self-test loop when format commands are executed with AUX1=$10 and AUX2=$50.

10.21 XF551 hardware

Drive mechanism

The XF551's drive mechanism is a double-sided, 40 track drive spinning at 300 RPM.

Controller

The XF551's controller is an 8040 running at 8.33MHz. The system clock speed is slightly higher to compensate for the 300 RPM rotational speed; the controller and FDC are clocked faster by the same amount that the drive spins faster, by 4.2%. This ensures that the FDC reads and writes at the same density on disk that other drives do at 288 RPM.

Although an 8.33MHz master clock is faster than most drives, the 8040 internally runs 15 clock cycles per machine cycle, so the CPU core runs machine cycles at a much lower 0.55MHz speed. The 8040 still maintains reasonable throughput at this rate because its instructions only take 1-2 machine cycles.

The only RAM in the system is 256 bytes of internal RAM within the 8040; 4K of firmware ROM is addressed externally. There is no memory map for the XF551 because the only peripheral connected to the data bus is the FDC and the address put on the bus is ignored, with register selection done through port 2.

The two I/O ports on the 8040 serve similar purposes to the RIOT I/O ports on the 810 and 1050.

Port	Signal	Direction	Usage	Polarity
Port 1	P1.7	Output	SIO data input (device to computer)	inverted
	P1.6	Output	Floppy drive side select	0 = bottom, 1 = top
	P1.5	Output	FDC data bus read/write control	1 = read, 0 = write
	P1.4	Output	FDC reset	0 = asserted (reset)
	P1.3	Output	FDC density control	1 = FM, 0 = MFM
	P1.2	Input	FDC INTRQ	1 = asserted
	P1.1	Output	FDC address bit 1	non-inverted
	P1.0	Output	FDC address bit 0	non-inverted
Port 2	P2.7	Input	Drive select	1 = D3: or D4:
	P2.6	Input	Drive select	1 = D2: or D4:
	P2.5	-	Not connected	
	P2.4	-	Not connected	
	P2.3	-	Not connected	
	P2.2	-	Not connected	
	P2.1	-	Not connected	
	P2.0	-	Not connected	

Table 61: XF551 8040 I/O port connections

The large number of non-connected lines on port 2 is not an accident or underutilization of the hardware. The output latches for those bits are instead used by the firmware as internal storage. This is required to support double-density operation with only 256 bytes of RAM; the entire RAM is used as the transfer buffer, requiring all other state to be stuffed into unconventional locations such as port 2, the timer (T) register, and even the stack pointer.

Two additional input pins on the 8048, T0 and T1, are connected to the SIO DATA OUT and FDC DRQ signals, respectively. T0 receives the inverted data sent from the computer; T1 is raised high when the FDC requires data transfer.

FDC

The floppy drive controller chip in the XF551 is a WD1770/1772. Unlike earlier Atari drives, the XF551 connects the 1770/1772 even more conventionally, with spindle motor, head stepping phases and the index sensor hooked up as well as ready and track 0 control lines. Thus, seeking in the XF551 is done by regular FDC seek commands rather than manually rotating control lines with the CPU, and motor idle is also under FDC control.

The difference between the 1770 and 1772 is the step rate used for seeking. The 1770 results in a seek rate of 28.9ms/track, whereas the 1772 gives 5.8ms/track. The seek times are slightly faster than standard due to the FDC being clocked at 8.3MHz instead of 8MHz.

10.22 Indus GT disk drive

The Indus GT disk drive from Indus Systems, Inc. provides up to 180K of single-sided, double-density disk storage over an extended version of the SIO disk protocol. It contains a Z80A running at 4MHz with 2K of RAM.

Firmware revisions

Two stock revisions of the Indus GT firmware are known and have been dumped. The revision numbers are stored in the firmware image and also reported by the Syncromesh and Super Synchromesh loaders.

Version	CRC32	Notes
1.10	D125CAAD	Unable to read sectors >720 in enhanced density
1.20	D8504B4A	Fixes ED and adds support for booting CP/M

Table 62: Indus GT stock firmware revisions

The version 1.10 ROM is mislabeled as v1.4 or INDUS14 in some places, which is unfortunate as it is buggy and cannot read enhanced density disks properly due to a broken sector number check. There is also a "1.2X" or "1.3X" firmware that are modified versions of 1.20.

The known original versions as noted in the official Indus GT Field Service Manual are 1.00, 1.10, and 1.20. No copy of 1.00 has been found yet.

Command set

Nine commands are supported by the built-in firmware: read sector ($52), status ($53), write sector ($57), put sector ($50), read PERCOM block ($4E), write PERCOM block ($4F), format ($21), format enhanced ($22), and execute ($58). This can be extended through uploaded code.

Execute command ($58)

The Indus GT specific 'X' ($58) command allows arbitrary code to be uploaded and executed on the drive. This is useful both for diagnostic purposes and to extend the functionality of the drive. In particular, allows the Syncromesh and SuperSynchromesh extensions to add (working) high speed operation.

Bit 0 of the AUX2 byte determines whether to upload code (1) or execute it (0). When uploading code, AUX1 specifies the length of code to be transferred in bytes. The code is located in the drive's memory at 7F00h and is invoked with AUX2 bit 0 = 0. The code can perform arbitrary transfers in response to being invoked, including both receiving data (write) and sending data (read).

Commonly uploaded code sequences

While arbitrary code can be executed on the Indus GT, there are a few standard code sequences used by software that interfaces with the Indus. Table 63 gives lengths, SIO checksums, and descriptions of the commonly uploaded code sequences.

Length (checksum)	Description
$19 ($52)	Checks firmware version on drive. Returns a two-byte payload containing the major and minor version in BCD, i.e. $01 20 for 1.20.
$8A ($EB)	Synchromesh loader from Indus GTSYNC.COM utility.
$67 ($49) $100 ($22) $100 ($A0) $100 ($CF)	Synchromesh firmware (SV1.20) from Indus GTSYNC.COM utility.
$E5 ($45)	SuperSynchromesh loader from Indus GTSYNC.COM utility.
$69 ($DA) $100 ($BA) $100 ($62) $100 ($F5)	SuperSynchromesh firmware (SV1.30) from Indus GTSYNC.COM utility.
$F5 ($A4) $100 ($EE) $100 ($96)	RamCharger firmware from Indus GTSYNC.COM utility.
$E5 ($F0)	SuperSynchromesh loader from SpartaDOS X INDUS.SYS.
$69 ($6E) $100 ($BA) $100 ($62) $100 ($F5)	SuperSynchromesh firmware (SV1.30 patched) from SpartaDOS X INDUS.SYS.

Table 63: Commonly used Indus GT uploadable code fragments.

The loaders are invoked repeatedly using the execute mode of the $58 command, supplying blocks of code each time. The short block is given first, but the blocks are arranged in ascending address order. No length is specified in AUX1 for each block and the loader and uploader must be matched. At the end of a firmware upload, an additional execute call is made with no length to finalize the upload.

High-speed operation

Synchromesh is the name of the Indus GT's high speed operation at 38,400 baud. It is very similar to but not exactly the same as the XF551's high speed operation. Bit 7 is set on a command to indicate that the data phase should be performed at high speed. As with the XF551, the command frame itself and the command ACK occur at low speed (19,200 baud), but the data frame ACK/Complete/Error bytes, data payload, and checksum are transferred at high speed.

The original version of Synchromesh on the drive uses the clock in/out signals from the computer to control send and receive timing on the drive. For these reasons, receive operations must always *synchronous* mode instead of asynchronous mode with the original version of Synchromesh. This doesn't apply to standard speed communication as the firmware synchronizes to the leading edge of the start bit in that case. The updated version of Synchromesh uploaded to the drive by GTSYNC.COM also uses conventional timing methods for high-speed operation and thus requires the usual asynchronous receive mode on POKEY.

SuperSynchromesh increases the transfer speed to 69,000 baud, equivalent to POKEY's divisor 6.

High-speed format commands

The stock firmware on the drive supports high-speed versions of the format commands, but they simply transfer the final sector list in high-speed – no change is made to the sector skew. All disks are formatted with even and odd sectors partitioned, giving a 9:1 interleave in SD/DD and 13:1 for ED.

The uploaded versions of Synchromesh revise the $A1 command to use more appropriate interleaves for high-speed operation. Synchromesh uses 6:1 interleave for SD and 9:1 for DD, while SuperSynchromesh uses 5:1 for SD and 7:1 for DD.

Also added with the firmware update is $A3, which reformats only the boot tracks of a disk with standard, non-high-speed appropriate interleaves. The first three tracks are formatted on SD, while the first two tracks are formatted on DD. There is no corresponding low-speed $23 command.

$A2 is not supported by the updated firmware, and neither the $A1 nor the $A3 commands can format ED disks.

Caution

The Indus GT and XF551 are incompatible with respect to the $A1 and $A2 format commands. The $A1 format command sends a low-speed data frame on the XF551, while on the Indus GT it sends a high-speed data frame. The $A2 command has the same behavior on the stock firmware and is not supported at all by the uploaded Synchromesh or SuperSynchromesh firmware.

TOMS Turbo Drive

The TOMS Turbo Drive is a clone of the Indus GT that is compatible hardware-wise, but uses different firmware. It is unique for using the SIO CLOCK OUT signal for bit timing when receiving data frames from the computer, which most drives either can't or don't do, and won't work if this signal is not sent from the computer.

The firmware for the TOMS Turbo Drive is unusually able to format and read/write PC 180K format disks, with 9 sectors per track of 512 bytes each. It does so with a 1:1 interleave, which leads to suboptimal read/write speeds. A 512 byte per sector disk is indicated by both the Percom block and bit 6 being set in drive status.

10.23 Indus GT hardware

Controller

The Indus GT is controlled by a Z80 microprocessor running at 4MHz, with the following memory map:

Address range	Mapping
$8000-FFFF	Upper 32K of RAMCharger RAM (if present)
$7800-7FFF	RAM (2K)
$7000-77FF	RAM (2K) (mirror)
$6000-6FFF	1770/2797 FDC
$4000-5FFF	LED/misc control
$3000-3FFF	Step control (write only)
$2000-2FFF	Status 2 (read only)
$1000-1FFF	Status 1 (read only)
$0000-0FFF	ROM (4K)

Table 64: Indus GT memory map

The lower 32K of address space can also be replaced with the lower 32K of RAMCharger memory.

Status 1 port

Bit	Description
Bit 7	1 = Write protect sensor latch (stays set when write protect is sensed)
Bit 6	1: Error button is depressed
Bit 5	1: Drive type button is depressed
Bit 4	1: Track button is depressed
Bit 3	1: Not connected (always 1)
Bit 2	0: S1-3 is closed (up) – selects double density as default
Bit 1	0: S1-2 is closed (up) – assigns D3: or D4: to drive[72]
Bit 0	0: S1-1 is closed (up) – assigns D2: or D4: to drive

Table 65: Indus GT status 1 port signals

In addition, accessing the status 1 port with an even address clears bits 4-7 of the status port. Typically, the firmware accesses an even address first and then re-reads an odd address to query the port.

Status 2 port

Bit	Description
Bit 7	1 = FDC INTRQ asserted
Bit 6	1 = FDC DRQ asserted
Bit 5	0 = SIO COMMAND line asserted
Bit 4	1 = SIO READY line asserted
Bit 3	SIO DATA IN (non-inverted)
Bit 2	SIO DATA OUT (non-inverted)
Bit 1	SIO CLOCK IN (non-inverted)
Bit 0	SIO CLOCK OUT (non-inverted)

Table 66: Indus GT status 2 port signals

Step control port

Writing to 3000-3FFFH controls the stepper motor. Bits 0-3 control the four phases, where a 1 bit activates a phase and only one phase is active at a time. Phases are activated in ascending order (0-1-2-3) for stepping inward to higher numbered tracks and descending order (3-2-1-0) for stepping outward to lower numbered tracks.

LED/misc control ports

Bits 0-6 of $4xxx and $5xxx control the LED readout display, composed of two 7-segment digits. $4000-4FFF controls the ones (right) digit, while $5000-5FFF controls the tens (left) digit. There is no decimal point. Bits 0-6 are hooked up to segments A-G, which are, in order: top, top right, bottom right, bottom, bottom left, top left, and center. The bits are active low so that a 0 bit lights up a segment.

Bit 7 controls miscellaneous signals on both ports. Writing D7=0 to $4000-4FFF enables the busy light; writing D7=1 to $5000-5FFF enables write precompensation.

Since both ports are write-only, the LEDs must be rewritten to toggle the busy light or write precomp and vice

[72] Unlike the 810 and 1050, the Indus GT uses plain binary for drive ID.

versa.

I/O ports

A number of signals are also tied to I/O ports in the range 00-0FH. Either an input or output operation will work as only the address matters:

- 00/01H: SIO AUDIO IN control
- 02/03H: Enable/disable drive index pulse (02H enables, 03H disables).
- 04/05H: SIO DATA IN / transmit to computer (04H transmits a 1).
- 06/07H: SIO DATA OUT / transmit as computer (06H transmits a 1).
- 08/09H: Set FDC density (08H = MFM, 09H = FM).
- 0A/0BH: Motor control (0AH = on).
- 0C/0DH: Direct index pulse control (0CH asserts).
- 0E/0FH: RAMCharger lower bank enable (0FH enables).

SIO connections

The Indus GT is unusual in being wired such that it can both transmit and receive on both the SIO DATA IN and SIO DATA OUT lines, allowing it to act as a host as well as a peripheral. It can also send pulse waves on SIO AUDIO IN and read both the SIO CLOCK IN and SIO CLOCK OUT lines, also uncommon connections. However, it cannot drive the SIO clock lines, nor can it drive or sense the proceed, interrupt, or motor control lines.

RAM Charger

The optional RAM Charger adds 64K of RAM to the Indus GT, allowing it to buffer tracks and to run CP/M. The upper 32K of RAM is always mapped at $8000-FFFF, while the lower 32K of RAM can be bank-switched in by accessing ports 0EH and 0FH, where 0FH enables the RAM and 0EH disables it.

10.24 ATR8000 hardware

The ATR8000 connects up to four standard disk drive mechanisms, an RS-232 serial port, and a parallel printer port to the computer.

Controller

A Z80 microprocessor running at 4MHz controls the drive. The memory map is simple: either 16K of RAM with $C000-FFFF as the canonical range, or 64K of RAM. The drive’s firmware is contained on either a 2K or 4K ROM that is mirrored throughout the lower 32K of address space and can be switched off for access to the RAM. Writes to RAM are always enabled and so writing through the ROM is possible.

I/O ports are used to communicate with the peripherals, which include the floppy disk controller (FDC), a Z8430 Counter/Timer Circuit (CTC) chip, and discrete control hardware.

Ports	Description
00-1FH	Not connected

Ports	Description
20-2FH	Printer status
30-3FH	Not connected
40-4FH	Floppy drive controller
50-5FH	RS-232 status
60-6FH	Not connected
70-7FH	SIO status
80-BFH	Counter/Timer Circuit
C0-FFH	Not connected

Table 67: ATR8000 input port assignments

Ports	Description
00-1FH	Not connected
20-2FH	Printer data latch
30-3FH	Drive control
40-4FH	Floppy drive controller
50-5FH	Miscellaneous control (bit 0 only) • 50/58H: SIO DATA IN • 51/59H: RS-232 transmit data • 52/5AH: ROM enable (0 = enabled) • 53/5BH: Printer strobe (1 = asserted) • 54/5CH: Index control flip/flop clear (0 = asserted) • 55/5DH: RS-232 DTR/RTS • 56/5EH: Index control flip/flop preset (0 = asserted) • 57/5FH: CTC trigger 0 select (0 = SIO DATA OUT, 1 = SIO COMMAND)
60-7FH	Not connected
80-BFH	Counter/Timer Circuit
C0-FFH	Not connected

Floppy drives

Up to four 5.25" or 8" disk drives can be connected to the ATR8000. The firmware detects which types are connected by checking the rate of index pulses from each drive, checking for 300 RPM for a 5.25" mechanism or 360 RPM for 8". Only one drive can be selected at a time.

The hardware has the ability to turn off the index sensor, allowing the FDC to read and write sectors across the index. This is done by controlling an index pulse enable flip/flop via a pair of ports; writing a 0 to bit 0 of 54/5CH asserts the clear line to disable the index pulse, and writing a 0 to bit 0 of 56/5EH asserts the preset line to enable the index pulse, with only one of these normally asserted at a time. However, it cannot manually inject index pulses under CPU control as most other 810-compatible drives can.

Floppy drive controller (FDC)

The FDC in the ATR8000 is a FD1797, which supports double density and side selection but not motor control. The motor is still controlled by the FDC, but using the head load output. Head stepping is controlled by the FDC. Due to support for 8" disk drives, the ATR8000 has the ability to clock the FDC higher than typical at double speed, permitting 2µs/cell FM and 1µs/cell MFM.

The DRQ and INTRQ lines from the FDC are wired up unusually to the Z80: they assert an NMI, but only when the Z80 is executing a HALT instruction. This allows the NMI to be used in a controlled manner. The two different interrupt sources can only be distinguished by means of the FDC status register.

Counter/Timer Circuit (CTC)

A Z8430 CTC chip provides both timer and signal triggering services. Of the four channels, channel 0 can be used to monitor either incoming SIO data or command state, while channel 2 can sense incoming data on the serial port. Channel 3 is chained to channel 2’s output in the ATR8000 so that both of them can be linked together for a extended range watchdog timer.

Any of the four channels can issue an interrupt to the Z80 on counter or timer expiration. The interrupt is delivered in vectored mode (IM 2), allowing the firmware to use distinct routines for each channel without needing to manually dispatch.

Drive control

Output to ports 30-3FH controls various drive and floppy controller signals.

Bit	Description
Bit 7	Density select (1 = FM, 0 = MFM)
Bit 6	Clock select (0 = 1µs/2µs, 1 = 2µs/4µs)
Bit 5	Side select
Bit 4	FDC reset
Bit 3	Drive select 4 (1 = selected)
Bit 2	Drive select 3 (1 = selected)
Bit 1	Drive select 2 (1 = selected)
Bit 0	Drive select 1 (1 = selected)

Table 68: ATR8000 drive control port layout (OUT 30-3FH)

Bits 0-3 are used to select drives, with a one bit selecting a drive and all zero bits deselecting all drives. It is possible to select more than one drive at a time, and in fact the firmware does this at startup to seek all four drives to track 0 at the same time.

Bit 6 selects the data rate for the floppy drive controller. A 1 bit selects a 4µs bit cell for FM or 2µs for MFM for 5.25” drives; a 0 bit doubles the data rate to 2µs FM or 1µs MFM for 8” drives.

Bit 7 selects FM for single density (1) or MFM for double density (0).

SIO bus connection

Input operations from ports 70-7FH read inputs from the SIO bus, as shown in Table 69. Output operations to port 50/58H bit 0 send data on the SIO DATA IN line with normal polarity. There is no hardware support for driving PROCEED/INTERRUPT or the clock lines.

Bit	Description
Bit 7	SIO DATA OUT (normal polarity)
Bit 3	SIO READY (1 = computer powered)
Bit 1	SIO COMMAND (0 = asserted)

Table 69: ATR8000 SIO status port layout (IN 70-7FH)

Parallel printer port

Output operations to ports 20-2FH write to the parallel printer port data latch. This data is then sent to the printer on the next data strobe, which is accomplished by raising and then lowering bit 0 on port 53H/5BH. Ports 20-2FH can also be read to query printer status, particularly the BUSY line. Table 70 gives the layout of this port.

Bit	Description
Bit 7	BUSY (1 = printer busy)
Bit 6	ERROR
Bit 5	/BUSY
Bit 4	/ACK

Table 70: ATR8000 printer status port layout (IN 20-2FH)

RS-232 serial port

The ATR8000 also supports an RS-232 compatible serial port, with support for control lines. However, it has no UART to drive it and no connection to the CTC, so all serial port I/O is bit-banged by the Z80. Output to port 51/59H bit 0 controls the transmit line, while input from port 50-5FH bit 7 senses the receive line. Both bit use normal data polarity, i.e. 0 and 1 bits give 0/1 data, start bits are 0, and stop bits are 1.

Output to port 55H/5DH bit 0 controls the DTR or RTS line, depending on jumper configuration. Input from ports 50-5FH read control lines; bit 2 senses Ring Indicator (RI), while bit 1 can sense pin 11, DCD, DSR, or CTS lines. Signal selection is controlled by jumpers.

10.25 Percom RFD

The Percom RFD is a 6809-based disk controller with one or two 5.25" drive mechanisms built-in and an expansion port to control up to two more external drives. Various sub-models include the RFD40-S1, RFD44-S1, RFD40-S2, and RFD44-S2.

Firmware revisions

There are four known main revisions of the Percom RFD firmware and one close relative for Astra drives, which have compatible hardware and use nearly identical firmware.

Revision	Dates	CRC32	Notes
V1.00		5A396459	
V1.10	1982-08-28 1982-11-08	E2D4A05C	Fixes deleted sectors not being returned as errors; reassembled with shorter branches and 8-bit displacements.
V1.10 (Astra 1001/1620)		2AB65122	Changes default step rate from 1 to 2; differs by only one byte

			from V1.10
V1.20		C6C73D23	Reverses sector interleave on side 2; fixes boot sector size override triggering on side 2
V2.10		AC141045	Formatted sectors are filled with $00; delay added before ACK and after Complete; density detection no longer limited to D1:; recalibration skipped for CRC errors; long sectors reported as RNF

Table 71: Percom RFD firmware revisions

The revision numbers and dates are not in the firmware but come from stickers on the EPROMs.

Firmware quirks

V1.00 firmware fails to report deleted sector marks as an error when reading sectors, result in compatibility issues with some copy protected disks. This is fixed in V1.10.

V1.00-1.10 incorrectly check for a boot sector after mirroring logical sector numbers for side 2, resulting in some sectors on side 2 being forced to 128 bytes in double density. V1.20 fixes this and also reverses the sector interleave when formatting side 2 to account for the backwards sector mapping.

V1.00-1.20 will only do density detection for D1:, and format disks with non-zero data. These bugs are fixed in V2.10.

All known firmware revisions have an off-by-one bug when mapping logical sectors to physical sectors on side 2, using one physical sector lower than they should. The last logical sector is incorrectly mapped to track 0, sector 0 as a result.

Controller

The Percom's controller is a 6809 running at 1MHz.

Address range	Mapping
$F800-FFFF	ROM (2K)
$F000-F7FF	ROM (2K) (mirror)
$DC00-DFFF	Static RAM (1K)
$D030-D033	6850 ACIA
$D018-D01B	FDC control (write only)
$D014-D017	Switch sense and drive control
$D010-D013	FDC

Table 72: Percom RFD-40S1 memory map

Interrupts

The fast IRQ (FIRQ) input of the 6809 is connected to the data request (DRQ) line of the floppy drive controller. For fast response time, the firmware does not actually use interrupts for data transfers; instead, it issues the SYNC instruction to block execution until DRQ is asserted.

The non-maskable interrupt (NMI) input on the 6809 is driven by a combination of the FDC's interrupt signal and a one-shot timer. The NMI signal is asserted at all times except when the timer is running and the FDC is not requesting an interrupt. Since the NMI is edge-activated, this effectively triggers an NMI whenever either the

FDC finishes a command or the timer expires, whichever is first. The timer is activated by writing a 1 to bit 0 of $D018-D01B and has a period of at least a few hundred milliseconds, sufficient to cover a few rotations of the disk; it can be prematurely ended by clearing bit 0.

ACIA

Communication over the serial I/O bus is handled by a 6850 ACIA. The SIO command signal is connected to the ACIA's Clear to Send (CTS) input.

The transmit and receive clocks are hardwired to 19200 baud (4MHz ÷ 13 as the 16x clock, to be exact), so the RFD-40S1 is one of the few drives that cannot do high-speed operation even with modified firmware.

Switch sense

Addresses $D014-D017 read the state of the configuration switches and jumpers. Bits 0-3 read the switches on the back, while bit 4 reads an internal jumper. Bits 0-1 select the starting drive number, from D1: to D4: in binary code; bit 4, if set, shifts to the D5-D8: range.

Bit 7 reads the inverted index pulse (/IP) signal from the currently selected drive.

Drive control

$D014-D017 controls miscellaneous drive functions. Bit 3, if set, enables drive select and turns on the motor for the selected drive; the drive is selected by bits 1-2. Bit 0 controls side (head) selection.

FDC control

$D018-D01B drives logic to control the floppy drive controller.

Bit 3 gates the index pulse from the drive, disabling it if set.

Bit 2 selects single density (0) or double density (1). This bit is responsible for the strange 0/4 value of the density flag in the Percom block, as the firmware simply copies that bit to the hardware.

Bit 1 selects 8" or 5.25" operation. It is normally cleared for 5.25" operation, which selects a 4µs bit cell for FM and 2µs for MFM. When set, the data clock rate is doubled for 8" operation with 2µs for FM and 1µs for MFM.

Bit 0 controls the command timeout circuitry. When enabled, any write to the FDC resets the command timer, which will assert the NMI signal to the 6809 after the timer expires. The NMI signal will also be asserted if the FDC itself requests an interrupt. Resetting bit 0 disables the timer, causing it to hold NMI in the *asserted* state. This design means that NMI is asserted constantly when idle, but this does not cause a problem as the NMI signal is edge triggered and only causes an interrupt when it transitions to an active state. This also prevents the FDC from triggering a second interrupt shortly after the timer expires.

10.26 Percom AT88

The Percom AT88 or AT-88 is a similar but incompatible design from the RFD, supporting only single density in the base model but having an option to add double density support through the AT88-DDA Doubler. Despite being less functional, the AT88 is a later design than the RFD. Like the RFD, the AT88 comes in both single drive (-S1) and dual drive (-S2) versions.

Note that the AT88-S1 is a different hardware design than the AT-88S1PD, which adds a printer port.

Firmware versions

AT88 firmwares appear to be related to RFD firmware. Table 73 lists known firmware revisions. All revisions but the second are named after version labels on the ROM chips.

Revision	CRC32	Based on	Notes
V1.10	942606D1	RFD 1.10	Earliest confirmed version, dated 12-8-82 (1982-12-08)
<V1.2 Unknown	404EFBE1	RFD 1.10	Hotfix for crash when executing P or W commands with invalid sector number, possibly V1.11
V1.2	2372FAE6	RFD 1.20	Fixes reversed interleave and bogus boot sector check on side 2.
V1.30	92FB8C9E	RFD 1.20	Double density sector format data changed to $FF on disk, $00 to computer (single sector formatted data unchanged).

Table 73: Percom AT88 firmware revisions

Controller

The Percom's controller is a 6809 running at 1MHz.

Address range	Mapping
$F800-FFFF	ROM (2K)
$5000-53FF	Static RAM (1K)
$000C-000F	6850 ACIA
$0008-000B	6821 PIA
$0004-0007	1771 or 1791/1795 FDC

Table 74: Percom AT88-S1 memory map

Interrupts

As in the RFD-40S1, the DRQ signal from the FDC is connected to the FIRQ input of the 6809 for fast synchronization using the SYNC instruction. The NMI timeout logic is omitted and instead FDC INTRQ is connected to 6809 IRQ indirectly through the PIA's CA2 input, using the PIA to invert the interrupt signal.

ACIA

Serial I/O bus communication is the same in the AT88-S1 as on the RFD-40S1: a 6850 ACIA is used with a fixed 19,200 baud clock.

PIA

The PIA replaces miscellaneous logic in the RFD.

Port	Signal	Direction	Usage	Polarity
Port A	PA7	Input	Not connected	
	PA6	Input	Write protect sensor	0 = write protected
	PA5	Input	Not connected	
	PA4	Input	Not connected	

Port	Signal	Direction	Usage	Polarity
	PA3	Input	Not connected	
	PA2	Input	Not connected	
	PA1	Input	Not connected	
	PA0	Input	Not connected	
	CA2	Input	FDC interrupt request (INTRQ)	1 = FDC interrupt
Port B	PB7	Output	FDC index pulse (/IP)	0 = index hole
	PB6	Output	FDC select (DDA only)	1 = 1771, 0 = 1795
	PB5	Output	Side select	1 = side 2
	PB4	Output	Motor enable	1 = motor on
	PB3	Output	Drive 4 select	1 = selected
	PB2	Output	Drive 3 select	1 = selected
	PB1	Output	Drive 2 select	1 = selected
	PB0	Output	Drive 1 select	1 = selected

Table 75: Percom AT88-S1 PIA connections

The CA2 input is configured by the firmware to trigger IRQA2 on a low-to-high transition of the FDC's INTRQ input, translating the FDC's active high INTRQ signal to the 6809's active low /IRQ signal. Reading the Port A data register is required to clear the PIA's IRQ (reading DDRA does *not* acknowledge the interrupt).

The 6821 PIA is functionally equivalent to the 6520 PIA used in the computer. However, the AT-88S1's PIA is wired with its address lines in conventional order instead of the computer's reversed order, so the order of the registers is: ORA/DDRA, CRA, ORB/DDRB, CRB.

Floppy drive controller

The floppy drive controller in the base AT88-S1 model is the 1771, the same as in the Atari 810. It supports single-density operation only and has an inverted data bus.

Double density operation is supplied by the optional Percom AT88-DDA Doubler, which adds a 1791 or 1795 controller. Although the 179X is capable of both single and double density operation, the DDA unusually connects both at the same time, allowing the firmware to switch between them via the PIA's PB6 output. Only one FDC is active at a time, but the firmware checks for the presence of the 179X by checking whether it can store different values in the track register of the two FDCs.

The index pulse signal from the drive mechanism is not hooked up to the FDC. Instead, software timeout loops are used for read/write sector commands and a manually drive index pulse through PIA PB7 is used for format operations.

10.27 Percom AT88-SPD/S1PD

The AT88-SPD is the more recent line of Percom disk drives with support for a built-in parallel port. It is an evolved design from the RFD and AT88.

As with earlier disk drives, the model number indicates the number of included disk drives, and so SPD and S1PD are often used interchangably.

Firmware revisions

Although the SPD firmware is very similar to the AT88 firmware, there are minor incompatibilities in the hardware design. The SPD firmware is twice as large (4K instead of 2K) and incompatble with the RFD and AT88 lines.

There are four known firmware revisions for the AT88-SPD/S1PD. All known versions contain fixes through RFD 1.20. The last, post-V1.21 revision contains similar fixes to RFD 2.10. V1.11 and V1.21 have been versioned and dated based on stickers on the EPROM chips.

Revision	CRC32	Date	Notes
V1.01	518900E6		Earliest known version
V1.11	87A0D7E7	1984-01-09	Default step rate changed from 0 to 1; space+CR printed for blank lines; printer delay logic fixed and improved
V1.21	FC6675F2	1984-05-04	Printer timeout and delays increased
>V1.21 Unknown	575C9833		Sector fill changed to $00; delays added after ACK; read sector does density detection for all drives instead of only D1:

Table 76: Percom AT88-SPD/S1PD firmware revisions

Controller

As with the earlier models, the AT88-SPD uses a 6809 as its controller. While earlier models used a 2K ROM firmware, the SPD firmware occupies the full 4K to accommodate the extra code for printer support. Table 77 gives the memory map.

Address range	Mapping
$F000-FFFF	ROM (4K)
$D000-DFFF	Static RAM (1K) (mirror)
$B000-BFFF	ROM (4K) (mirror)
$9000-9FFF	Static RAM (1K) (mirror)
$7000-7FFF	ROM (4K) (mirror)
$5400-5FFF	Static RAM (1K) (mirror)
$5000-53FF	Static RAM (1K)
$4010-4FFF	FDC/PIA/ACIA (mirror)
$400C-400F	6850 ACIA
$4008-400B	6821 PIA
$4004-4007	1791/1795 FDC
$3000-3FFF	ROM (4K) (mirror)
$1000-1FFF	Static RAM (1K) (mirror)
$0000-0FFF	FDC/PIA/ACIA (mirror)

Table 77: Percom AT88-SPD/S1PD memory map

Floppy drive controller

The floppy drive controller in the AT88-SPD is ether a 1791 or a 1795, with the firmware able to use either. Unlike the AT88-S1, the AT88-SPD lacks a 1771 and relies on the 179X for single density operation.

PIA

Although the AT88-SPD has the same 6821 PIA as the AT88-S1, the connections are different, making the firmwares incompatible.

Port	Signal	Direction	Usage	Polarity
Port A	PA7	Output	Printer /D7	Inverted
	PA6	Output	Printer /D6	Inverted
	PA5	Output	Printer /D5	Inverted
	PA4	Output	Printer /D4	Inverted
	PA3	Output	Printer /D3	Inverted
	PA2	Output	Printer /D2	Inverted
	PA1	Output	Printer /D1	Inverted
	PA0	Output	Printer /D0	Inverted
	CA2	Output	SIO external clock enable	1 = enable ext.
Port B	PB7	Input	Printer /BUSY	0 = busy
	PB6	Input	Printer FAULT	1 = fault
	PB5	Output	FDC index pulse (/IP)	0 = index hole
	PB4	Output	Density select	1 = FM, 0 = MFM
	PB3	Output	Side select	1 = side 2
	PB2	Output	Motor enable	1 = motor on
	PB1	Output	Drive select 0	0 = D1/D3:, 1=D2/D4:
	PB0	Output	Drive select 1	0 = D1/D2:, 1=D3/D4:
	CB2	Output	Printer /STROBE	0 = asserted

Table 78: Percom AT88-SPD/S1PD PIA connections

ACIA

A 6850 ACIA is used to communicate over the SIO bus, with SIO command connected to /CTS and SIO ready to /DCD, both through inverters to correct polarity.

In normal operation, a 4MHz ÷ 13 = 308KHz clock is used as the transmit/receive clocks for the ACIA, with it configured to apply an additional ÷16 down to 19.2KHz. New to the SPD/S1PD is the ability to also switch to using the SIO CLOCK OUT line as the transmit/receive clock through PIA CA2, allowing for high-speed operation. Unfortunately, this is incompatible with the US Doubler high speed protocol since it typically uses SKCTL=$13 to receive, which does not output a clock; this method can only work if channel 1+2 is programmed to the same rate as channel 3+4 and mode SKCTL=$73 is used instead.

Interrupts

As with earlier drives, the FDC data request (DRQ) signal is connected to the 6809's fast interrupt request (FIRQ) input for use with the SYNC instruction. FDC INTRQ is still connected to 6809 IRQ, but through an inverter instead of the PIA.

The ACIA's IRQ signal is not connected, and the ACIA is always used in polled mode.

10.28 Amdek AMDC-I/II

The Amdek AMDC-I/II is a unique Atari-compatible disk drive using a non-standard 3" disk form factor.

Enhanced density support

The AMDC has support for 1050-style enhanced density, but with significant quirks. It does not have the ability to automatically detect both enhanced density and double density disks, only able to detect one of them as selected by the 1050 compatibility switch.

Also, when selecting enhanced density, the firmware sets the total sector count to 1024 instead of 1040. This is presumably because of DOS 2.5, which only uses 1024 sectors. The format command does format the disk for the full 1040 sectors, and the drive can access the additional 16 sectors if the total sector count is overridden with the Write PERCOM Block command.

Memory access commands ($58-5A)

Commands $58 and $59 allow arbitrary memory read/write access to the address given by AUX1 and AUX2, with AUX2 being the high-order byte. There is no restriction on transfer address. The transfer length is always 128 bytes.

Command $5A executes a subroutine call to the address given by AUX1/2. An RTS ends the command and returns control to the firmware. All usual command replies are suppressed, so not even the command ACK is sent by default.

These commands are only enabled if the internal jumper is connected and SW1-4 are set to either all off or all on. Otherwise, they are omitted from the command table check and NAKed by the drive.

High-speed commands ($72/73/77)

The Amdek supports high-speed transfers, but not through a standard protocol. The $70, $72, and $77 commands are the high-speed versions of the put/read/write sector commands. The command frame, ACK/NAK, and Complete/Error bytes are sent at 19200 baud, with only the data frame and data frame checksum using high speed.

Currently, the high speed rate is currently unknown, as it is controlled by hardware and not determined by firmware. The reliability of high speed reads is also questionable as the firmware does not have a delay to wait for the checksum byte to complete transmission before reverting the transmission clock.

Set density commands ($74/75/76)

The $74-76 commands set the current density to medium, single, and double density, respectively. They are equivalent to setting the same parameters through the Write PERCOM Block command.

Reset command ($78)

Re-runs the power-on initialization code, clearing RAM, recalibrating drives, and resetting all internal settings based on drive detection and switch settings. Unlike similar commands in some other drives, control flow is preserved and the command completes normally with a Complete byte after re-initialization.

Set interleave command ($66)

Sends a 26-byte data frame to the drive containing the sector interleave order to use for the next format command ($21). The format command must immediately follow as any other command will clear the custom interleave table.

This command is notable for conflicting with the US Doubler's Format Skewed ($66) command, which has a similar purpose but is incompatible -- the data frame size is different and the Amdek requires a separate command to actually initiate the format operation.

Get/set drive IDs commands ($63/64)

Gets or sets the drive IDs for all four drives as a four-byte data frame. There is no restriction on the drive IDs used.

Get drive count ($69)

Returns the number of drive mechanisms detected by the firmware. This must be sent to the lowest drive ID as configured by SW5-6 and that drive must not have had its ID changed; sending this command to any other drive ID or to the lowest drive with the ID changed will result in a NAK.

Check disk command ($43)

Turns on the disk drive and checks track 1 to determine density; returns a single byte data frame with $00 for no disk, $01 for single density (FM), and $02 for enhanced or double density (MFM). The density setting for the drive is also updated.

Get drive speed command ($71)

Turns on the disk drive and measures index pulses to determine the spindle motor speed. On success, a two-byte data frame is returned containing a value that is 300 when the drive is correctly calibrated to 300 RPM, and lower or higher when the spindle is running slower or faster. The value is not quite a correct RPM value as it is calculated as (600 - period*1500) instead of 60/period. An error is returned if two index pulses are not detected within a timeout period.

Read configuration ($73)

Returns the state of the configuration switches and jumpers on the drive as a one-byte data frame. Bits 0-6 contain SW1-SW7, while bit 7 contains the internal jumper state. SW8 is not returned. All bits are 1 for switch off and 0 for switch on.

Get version commands ($56/$6D)

Command $56 returns a 25 byte data frame containing the version string for the drive. For version 1.1 firmware, this string is: "` AMDC VER 1.1    1/30/84`".

Command $6D returns another 25 byte data frame with alternate firmware information. For v1.1, it is: "` RBM 10/27/83  HI MOM!!!!`"

Controller

The controller in the Amdek AMDC is a 1MHz 6809. Table 79 gives the memory map.

Address range	Mapping
$F000-FFFF	ROM (4K)
$E000-EFFF	ROM (4K) (mirror)
$C000-DFFF	Watchdog timer (read/write strobe only)
$8000-BFFF	Switch sense (read only)
$6000-7FFF	Control port (write only)
$4000-5FFF	1797 FDC
$2000-3FFF	6850 ACIA
$1800-1FFF	Printer data port (write only)
$1000-17FF	Printer/jumper status port (read only)
$0800-0FFF	(Unknown write strobe)
$0400-07FF	4K static RAM (mirror)
$0000-03FF	4K static RAM

Table 79: Amdek AMDC-I/II memory map

Switches

$8000 and $8001, and all mirrors above that within $8000-BFFF, read the state of eight externally settable DIP switches on the drive. All switches are read as a 0 for the 'on' state and 1 for 'off'. SW1-4 set the default density for each of the four disk drives and are returned in bits 0-3 of $8000, where 0 means single density and 1 means double density.

$8001 returns mixed settings from the state of SW5-8 in bits 0-3. SW5-6 in bits 0-1 determine the starting drive ID to use, with %00 assigning D1: to the first drive and %10 assigning D3: to the first drive. SW7 in bit 2 selects whether the internal or external drives are assigned first -- if 0, the internal drives are assigned drive IDs first, and 1 gives the external drives the lowest IDs. SW8 in bit 7 controls 1050 compatibility mode, where a 0 forces enhanced density only for MFM mode, while 1 enables double-density mode.

Control port

Writes to $6000-7FFF activate a control port driving miscellaneous functions.

Bit	Description
Bit 7	FDC density select (1 = FM, 0 = MFM)
Bit 6	(Unknown disk-related function)
Bit 5	Printer strobe (1 = asserted)
Bit 4	Spindle motor enable (1 = on)
Bit 3	High speed enable (0 = 19200 baud, 1 = unknown speed)
Bits 0-1	Drive select (non-inverted)

Printer port

Writes to $1800-1FFF set the printer data latch, which is then processed by the printer when the printer strobe is triggered through bit 5 of the control port.

Reads from $1000-17FF return printer status, including BUSY in bit 7 (1 = busy), SELECT in bit 6 (0 = printer present). Bits 4 and 5 are repurposed for other drive status, with bit 5 returning a 1 if there is a second internal drive (AMDC-II configuration).

Bit 4 reads the state of an internal jumper, returning 0 if the jumper is connected. The internal jumper enables test functions selected by SW1-4, including loops for testing memory, the printer port, and the SIO port, and the memory read/write/execute commands.

ACIA

Serial bus communication occurs through a 6850 ACIA. SIO COMMAND is connected to CTS, and SIO READY to DCD. For standard speed communication, the ACIA receives a 19.2KHz × 16 clock. High speed uses a ×1 clock, but the clock rate is currently unknown.

Chapter 11
Parallel Bus Interface

11.1 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.

11.2 Common memory map

Address regions

The address pages $D1xx, D6xx, and D7xx 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.

11.3 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 80: 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	SCSI BSY asserted (+BSY)
	1	SCSI BSY negated (-BSY)
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.

11.4 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	6821 PIA
D17F D150	6522 VIA
D13F D120	6551 ACIA
D11F D100	Unused

Table 81: 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.

6821 PIA connections

The BlackBox uses a 68B21 PIA, which is pin and software-compatible with the computer's 6520 PIA.

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.

11.5 Atari 1090 80 Column Video Card

The Atari 1090 80 Column Video Card (CVC) was an unreleased card for the 1090XL Expansion System, providing an 80 column display through the Parallel Bus Interface. Prototypes and development information for the card have been found, however, and the following information is derived from schematics and firmware released by the Atari Historical Society [AHS-80CVC].

Hardware

The 80CVC uses a 68B45 CRTC to drive video timing and video addressing. Video memory is provided by a 2K static RAM, which provides character names for lookup in a 2K character ROM. Firmware is additionally provided by another 2K static ROM.

The character ROM contains 8x8 character cell graphics, stored in the same format as an ANTIC mode 2 character set. Two character sets are included, each containing 128 characters of 8 bytes each; the character set is selected by a hardware jumper. Only the low 7 bits of each character name are used for lookup, with bit 7 applying inversion to the character cell.

The existing known firmware ROM has a CRC-32 of [D2777837], and the known character set ROM a CRC-32 of [829F5706]. The character set ROM contains verbatim copies of the standard and international character sets found in standard Atari OS ROMs, with the standard character set in the low half. Thus, the character names in video memory are stored as INTERNAL character codes.

Address	Read	Write
$D1FA		CRTC register address
$D1FB	CRTC register data	
$D1FC		VRAM aperture base
$D1FD		

Table 82: 80CVC hardware registers

Video signal

The CRTC parameters established by the firmware create an 80x25 text area of 8-line tall character cells. The total scanned area is 114x32 with 6 extra scanlines. Character cells are 8 pixels wide in the hardware and shifted out at a dot clock of 14.318MHz. All together, this produces a 262 scan line display with a horizontal scan rate of 15.7KHz and vertical scan rate of 59.92Hz. The frame timing is thus the same as an NTSC ANTIC, though the timing is not necessarily synced with ANTIC's scan.

No color burst is emitted, so the display is monochrome and no artifacting is possible.

The video logic inverts the signal when the cursor is present. If the character cell is also inverted by bit 7 of the character name, this is XORed with the cursor, so no inversion results if both are on. The hardware cursor is not used by the firmware.

CRTC access

The 6845's register address and data ports are mapped to $D1FA and $D1FB, respectively. $D1FA is write-only and selects a register via bits 0-4. $D1FB is read/write depending on the specific CRTC register selected.

Video memory access

Only the low 11 of the total 14 address bits of the CRTC are used to address video RAM. It would be possible to hardware scroll the display by wrapping around the 2K VRAM address space, but the firmware does not use this capability.

An unusual write-through arrangement is used for CPU access to video memory. Bits 3-7 of $D1FC or $D1FD set the base address of the 2K video memory aperture, which can be placed anywhere in RAM on a 2K boundary. Writes to this window take place in both main memory and video memory, but reads are serviced from main memory. Thus, the contents of video memory and reads from the aperture will be out of sync until the CPU has cleared the region, but the CPU always reads the content of main memory. This is fortunate since the aperture base address is random on startup.

Because the aperture works by snooping the address and data buses and does not depend on signals from the MMU, it can be placed at any address, including over I/O and ROM regions not normally usable for external mapping over the Parallel Bus Interface.

PBI support

The 80CVC is connected to the Parallel Bus Interface as device 0 ($01), and must be selected through $D1FF before its registers and firmware are visible. Per PBI standards, the 2K firmware is overlaid over the math pack at $D800-DFFF when the device is selected. However, the VRAM write-through aperture is always active regardless of the PBI selection state.

IRQs are not supported by the 80CVC, but the hardware pulls up D0 on a read to $D1FF to signal a lack of interrupt.

OS compatibility

The firmware contains a number of hardcoded addresses of internal tables and routines within the Display Handler of the internal OS ROM, and as a result will only with the XL/XE OS.

Chapter 12
Internal devices

12.1 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.

12.2 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.[73]

12.3 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:[74]

- 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.

[73] 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.
[74] The author would like to thank Candle for providing Ultimate1MB technical information and hardware for testing.

For official programming information: [U1MB].

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.

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.

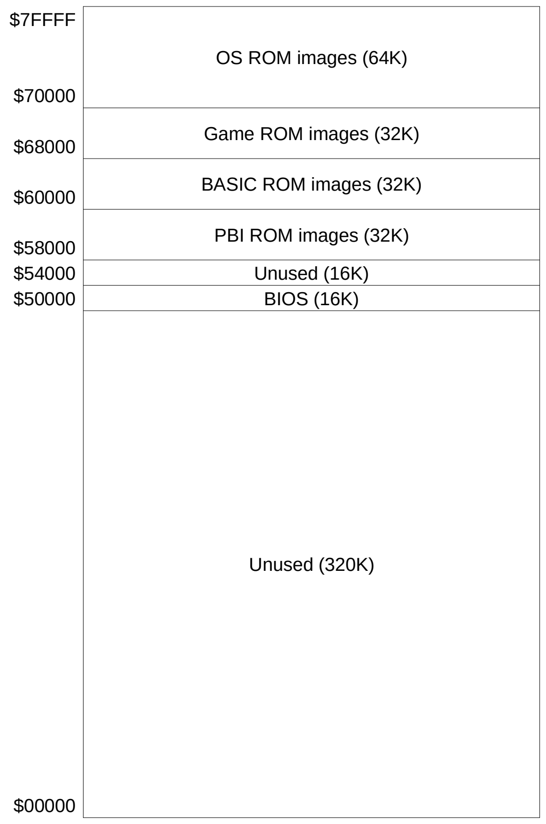


Figure 13: Ultimate1MB flash memory map

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.

The M0 output signal is a generic output with no specific behavior in the U1MB, but is commonly connected to a switch to toggle a stereo POKEY expansion. When connected, setting bit 0 = 1 enables address decoding and analog output for the second POKEY.

$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.

12.4 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 83 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 83: 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. Each bit in the priority mask controls whether the overlay has priority over the corresponding GTIA layer, including playfields, player/missile graphics, and the background. A ‘1’ bit gives the overlay priority over the GTIA layer.

Bit	FX1.24	FX1.26
7	PF3/P5	Background color
6	PF2	PF2/PF3/P5
5	PF1	
4	PF0	
3	P3	
2	P2	
1	P1	
0	P0	

Table 84: VBXE overlay priority bits

Warning

The definition of priority bits 6 and 7 is different between FX1.24 and FX1.26. Bit 7 is particularly troublesome as VBXE software commonly turns off most or all of the ANTIC+GTIA graphics, and a priority of $00 in that case will display the overlay normally in FX1.24 but cause it to vanish on FX1.26.

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 85: VBXE attribute map block layout

The first three bytes override the PF0-PF2 playfield colors used for rendering the ANTIC playfield. These colors are subject to the *xcolor* setting – if extended color mode is disabled, the LSB of these color values are forced to 0, the same as for the GTIA color registers.

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 86: 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, 1-64 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 87 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 87: 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 88 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 88: 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

	Read	**Write**
$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 89: 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.

12.5 APE Warp+ OS 32-in-1

The APE Warp+ OS 32-in-1 is a small internal device from AtariMax that replaces the OS ROM in an XL/XE computer with bank-switched flash ROM, allowing access to 32 different OS ROM images switchable under software control.

Hardware connections

The Warp+ OS has access to A0-A13 and the data bus through the OS ROM socket, but not R/W or A14-A15. Thus, it can map flash ROM within the OS and Self-Test ROM address spaces, but not anywhere else. It also has three external connections to the Select button, PIA port B bit 7, and the reset signal. The PB7 and reset signal connections are bidirectional, allowing the Warp+ OS to drive the signals as well as sense them.

Flash ROM

4Mbit (512KB) of flash ROM provides the OS slots for the Warp+ OS, divided into 32 slots of 16KB each. Each slot uses the same mapping as the standard XL/XE OS ROM, with the first 4K mapped to $C000-CFFF, the next 2K to the self-test window at $5000-57FF, and the last 10K to $D800-FFFF.

The last slot ($1F) is used for the menu. Aside from being directly reachable with Select+Reset, it is simply mapped as an OS ROM image like the rest of the slots. In particular, during normal operation it is not run as a pre-OS like the Ultimate1MB BIOS, but skipped entirely with the computer booting up directly on the selected OS slot without software intervention.

Because the hardware lacks sufficient bus connections to fully decode the OS region or to detect writes, it is not possible to switch the Warp+ OS's flash ROM to autoselect mode or to program/erase it from the computer itself.

PB7 communications protocol

The self-test enable signal on PIA port B bit 7 is used for bidirectional communications with the Warp+ OS. A low-speed asynchronous serial stream is sent over this signal in either direction, with start+stop bits and 8 data bits LSB-first. The baud rate is low, about 1100 baud.

Both commands and responses are sent as three-byte packets, with the first byte being a signature byte and the second and third bytes the same. This provides some protection against false commands when toggling the self-test ROM or communication errors. Commands are of the form $55 XX XX and replies $AA YY YY.

Reading the current slot selection

A $55 20 20 command queries the current ROM slot, returned as $AA YY YY where YY is the current slot in the range $00-1F. This reads the slot setting in NVRAM. It is not necessarily the currently active slot, as this command is also used by the menu OS in slot $1F.

Selecting a slot

Writing a $55 XX XX command with XX in $00-1F selects that as the new slot, writing it to NVRAM and remapping the OS ROM address space to the corresponding flash memory. The computer is also reset to start up with the new OS ROM image.

Forced menu entry

A Select+Reset key combination will restart the computer in the menu stored in slot $1F. This does not change the slot setting in NVRAM. The Select key is sensed in hardware, so it cannot be blocked or repurposed in software.

There is no corresponding hardware support for the Option key, which is sensed in software within the menu code for the Option+Select+Reset shortcut to enter APE remote control mode.

12.6 Bit-3 Full-View 80

The Bit-3 Full-View 80 is an add-on card that replaces the RAM card in slot 3 to provide 80-column display capability in an Atari 800. The use of RAM card slot 3 allows the Full View 80 to map memory ranges that are not available from the cartridge port, at the cost of requiring a 32K card in another slot to maintain 48K RAM.

CRTC

The 80 column display is driven by a Synertek 6545-1 CRTC, which provides the video timing and fetches characters from a 2K static RAM. The CRTC is mapped to $D580-D581, with writes to $D580 selecting a CRTC register to read or write through $D581, and reads from $D580 reading the CRTC status register.

Video memory

CPU access to VRAM is exclusively through the CRTC. As the CRTC requires full access to VRAM during active display, the CPU can only read or write VRAM during blanking periods. This is mediated by a transparent memory addressing mechanism exclusive to the Synertek version of the 6545, where the SY6545-1 automatically generates update cycles to VRAM during blanking times based on an update address register pair.[75]

[75] [SY6545-1-AN3]

CRTC registers $12 and $13 contain the high and low address bytes of the update address. Selecting the dummy register $1F -- without needing an explicit read or write -- clears the update ready bit 7 in the status register and schedules an update at the next available blanking time. The byte read is automatically latched into $D583. Writing to $D585 causes the CRTC to repeat the update cycle to trigger the write to VRAM at the next available interval. After a read or write, the update address is auto-incremented.

VRAM is deliberately mirrored into $0800-0FFF so that the display address can wrap around the RAM for scrolling purposes.

Character generator

The character map is stored in 2K of static RAM and translated to character bitmaps through a character generator stored in a 4K EPROM. Each character is stored with pixels in LSB-to-MSB bit order, reversed from ANTIC, and with 16 bytes for a 8x16 character matrix. The default CRTC settings used by the firmware only use the first 10 rows for an 8x10 matrix.

Bit 7 of each character cell selects inverse video. However, the character generator EPROM still stores 256 characters, and the second 128 characters corresponding to inverse video are uninverted in the EPROM, with inversion occurring externally.

Video switch

The Full-View 80 is designed to pass the standard ANTIC+GTIA video output through its own output when not enabled. The video output can be switched between the 40-column and 80-column display under software control via bit 3 of $D508.

Video timing

The default video timing set by the firmware is for 113x26 character cells of 8x10 with 4 extra scan lines, at a dot clock of $2 \times F_{sc}$ (14.31MHz). This gives a horizontal rate of 15.839KHz, vertical 59.995Hz, and a pixel aspect ratio (PAR) of 6 ÷ 14 = 0.429.

Firmware

The firmware is stored in a second 4K EPROM as 16 banks of 256 bytes each, mapped into $D600-D6FF.

Control register

Write-only register $D508 controls firmware bank switching and video output.

Bit	Description
Bit 5	Firmware bank bit 3
Bit 4	Video switch (1 = 80-column, 0 = GTIA output)
Bit 3	80-column video blank (1 = blanked, 0 = unblanked)
Bit 2	Firmware bank bit 2
Bit 1	Firmware bank bit 1
Bit 0	Firmware bank bit 0

Table 90: Bit-3 Full-View 80 $D508 register bits

$D508 is reset to $00 on power-up.

Chapter 13
5200 SuperSystem

13.1 Introduction

The 5200 SuperSystem is a console with hardware similar to the 400/800 computer line. The 5200 cannot run software made for the 400/800 or vice versa, but they share the same CPU and many of the same custom chips, and therefore software can be ported between the two with only minor difficulty.

13.2 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.

13.3 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.[76]

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.[77]

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.

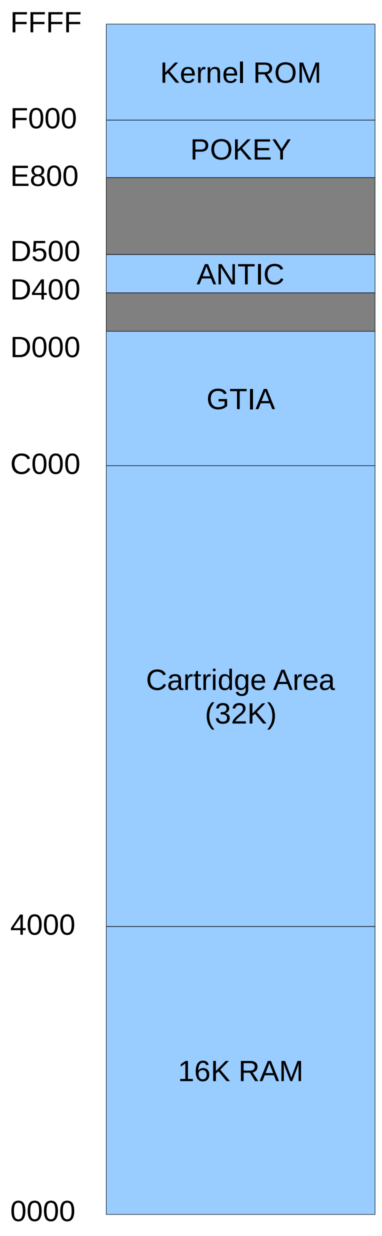
[76] [AHS03a] p. 25.
[77] [AHS03a] p. 21.

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.

Triggers

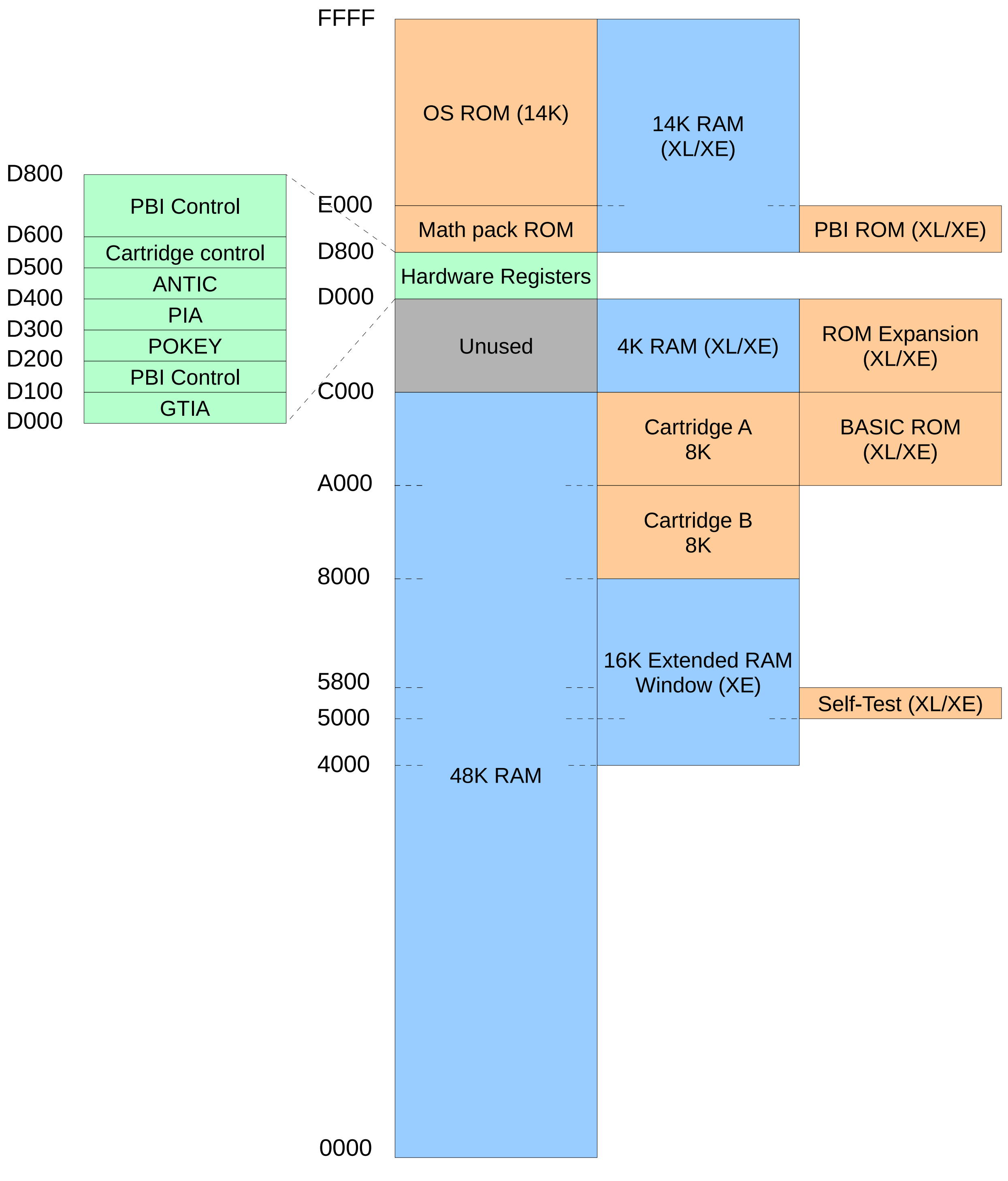
There are two trigger buttons on the 5200 controller, an upper trigger and a lower trigger. The bottom button of each controller is wired to TRIG0-TRIG3, depending on the controller, and functions the same as a joystick button on the 8-bit computer line. The top button is instead wired to the $\overline{\text{KR2}}$ 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.

13.4 5200 Memory map



Chapter 14
Reference

14.1 Memory map



14.2 Register list

HPOSP0-3 [D000-D003, W]
M0PF-M3PF [D000-D003, R]
HPOSM0-3 [D004-D007, W]
P0PF-P3PF [D004-D007, R]
SIZEP0-SIZEP3 [D008-D00B, W]
M0PL-M3PL [D008-D00B, R]
SIZEM [D00C, W]
P0PL-P3PL [D00C-D00F, R]
GRAFP0-3 [D00D-D010, W]
TRIG0-3 [D010-D013, R]
GRAFM [D011, W]
COLPM0-3 [D012-D015, W]
PAL [D014, R]
COLPF0-3 [D016-D019, W]
COLBK [D01A, W]
PRIOR [D01B, W]
VDELAY [D01C, W]
GRACTL [D01D, W]
HITCLR [D01E, W]
CONSOL [D01F, R/W]
AUDF1-4 [D200/2/4/6,W]
POT0-7 [D200-D207,R]
AUDC1-4 [D201/3/5/7, W]
AUDCTL [D208, W]
ALLPOT [D208, R]
STIMER [D209, W]
KBCODE [D209, R]
SKRES [D20A, W]
RANDOM [D20A, R]
POTGO [D20B, W]
SEROUT [D20D, W]
SERIN [D20D, R]
IRQEN [D20E, W]
IRQST [D20E, R]
SKCTL [D20F, W]
SKSTAT [D20F, R]
PORTA [D300, R/W]
PORTB [D301, R/W]
PACTL [D302, R/W]
PBCTL [D303, R/W]
DMACTL [D400, W]
CHACTL [D401, W]
DLISTL/DLISTH [D402-3, W]
HSCROL [D404, W]
VSCROL [D405, W]
PMBASE [D407, W]
CHBASE [D409, W]
WSYNC [D40A, W]
VCOUNT [D40B, R]
NMIEN [D40E, W]
NMIST [D40F, R]

NMIRES [D40F, W]

Unit	Address	Description
GTIA	HPOSP0, HPOSP1, HPOSP2, HPOSP3 $D000-$D003	Player 0-3 horizontal position (Write Only)

Register layout

7 … 0
Horizontal position

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

7							0
0	0	0	0	PF3	PF2	PF1	PF0

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

7 … 0
Horizontal position

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

7							0
0	0	0	0	PF3	PF2	PF1	PF0

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

7	0
Ignored	Size

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

7							0
0	0	0	0	P3	P2	P1	P0

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

7							0
0	0	0	0	P3	P2	P1	P0

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

7 … 0
Player graphics data

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

7							0
0	0	0	0	0	0	0	T

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

7			0
Missile 3	Missile 2	Missile 1	Missile 0

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

7		0
Hue	Luminance	Ign.

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

7				0
0	0	0	0	PAL

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.[78]

[78] [ATA82] III.1

Unit	Address	Description
GTIA	COLPF0, COLPF1, COLPF2, COLPF3 $D016-$D019	Playfield 0-3 color register (Write Only)

Register layout

7		0
Hue	Luminance	Ign.

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

7		0
Hue	Luminance	Ign.

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

7							0
P3	P2	P1	P0	M3	M2	M1	M0

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

7			0
Ignored	LT	P	M

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

7 … 0
Ignored

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.[79] 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.

[79] [ATA82] III.15

Unit	Address	Description
POKEY	AUDF1, AUDF2, AUDF3, AUDF4 $D200, $D202, $D204, $D206	Audio channel 0-3 frequency (Write Only)

Register layout

7 0
Frequency

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

7 … 0
Potentiometer read counter value

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 cleared 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

7				0
CLK	NM	NC	D	Volume level

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

7							0
PLY	CH1	CH3	L12	L34	HP1	HP3	15K

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

7 0
Potentiometer read status

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

7 … 0
Ignored

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

7		0
CRL	SHF	Keyboard scan code

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

7 … 0
Ignored

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

7 0
Random data

Description

Reads the state of the top eight bits of the 17-bit polynomial noise generator. This generator shifts right one bit every machine cycle (1.79MHz).

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

7 … 0
Ignored

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

7 … 0
Serial data

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

7 … 0
Serial input data

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

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
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

7					0
FB	Serial clk mode	2T	FP	KS	KD

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 5.6, 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	PORTA $D300	Port A data/direction register (Read/Write)

Register layout

7		0	
Direction bits			Direction (PACTL bit 2 = 0)
Jack 2	Jack 1		Input/output (PACTL bit 2 = 1)

Description

PORTA accesses the joystick direction signals for controller ports 1 and 2. Normally port A is configured as all-inputs to read the joystick states, but can be configured for output for other controller types.

PORTA is a multiplexed location that accesses either the data direction register or the input/output register, depending on the state of PACTL bit 2. When PACTL bit 2 is cleared, the data direction register is accessed and is read/write; when PACTL bit 2 is set, read accesses read the input register, and write accesses write the output register. The output register cannot be read, though the input register will read the same value if no external device is pulling down the signal lines.

Both the data direction and output registers are cleared to $00 when the PIA is reset.

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

7				0
I1	I2	CA2	DIR	CA1

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

7				0
I1	I2	CB2	DIR	CB1

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

7					0
Ignored	D5	D4	D3	D2	D1:D0

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

7			0
Ignored	D2	D1	D0

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

15 … 0
Display list address

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.[80]

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.[81]

[80] [ATA82] III.6
[81] [ATA82] III.5

Unit	Address	Description
ANTIC	HSCROL $D404	Horizontal scroll offset (Write Only)

Register layout

7	0
Ignored	Horizontal scroll

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

7	0
Ignored	Vertical scroll

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

7	0
P/M base address	Ignored

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 are used and therefore the P/M data must be aligned to a 2K boundary. For two-line resolution, the top six bits are used and 1K alignment is required. Bit 2 can be changed even in one-line resolution mode, however, and its value will take effect when the mode is switched to two-line resolution.

Unit	Address	Description
ANTIC	CHBASE $D409	Character data base address (Write Only)

Register layout

7	0
Character data base address	Ign.

D7:D2	Bits 10-15 of character data base address (ANTIC modes 2, 3, 4 and 5)
D7:D1	Bits 9-15 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

7 … 0
Ignored

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

7 … 0
Bits 1-8 of vertical position counter

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

7		0
DLI	VBI	Ignored

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.[82]

The reset interrupt cannot be masked through NMIEN.[83]

[82] Hardware II.28
[83] 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. However, the corresponding status bit is still commonly set on power-up.

Unit	Address	Description
ANTIC	NMIRES $D40F	Non-maskable interrupt reset (Write Only)

Register layout

7 … 0
Ignored

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.

14.7 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							
	D20B (W)	POTGO	Pot scan start strobe	Ignored							
	D20D (R)	SERIN	Serial input data	Received serial data							

Unit	Address	Name	Desc	Bits							
	D20D (W)	SEROUT	Serial output data	Serial data to transmit							
	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 15
Bibliography

[6502Dec] Clark, Bruce, Decimal Mode, 2009. Retrieved on June 28, 2009 from http://www.6502.org/tutorials/decimal_mode.html.

[815] Atari, 815 Dual Disk Drive Operator's Manual, 1981. Retrieved on September 9, 2020 from https://archive.org/details/atari815manual.

[AHS-80CVC] , Atari 1090 80 Column Video Card (Atari 80CVC), 2020. Retrieved on May 25, 2020 from http://www.atarimuseum.com/computers/8bits/xl/xlperipherals/1090/80%20column%20board/index.htm.

[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.

[CorvusGTI] Corvus Systems, Corvus Mass Storage Systems General Technical Information, 1984.

[CRA82] Crawford, Chris, De Re Atari, 1982.

[EYE86] Eyes, David and Lichty, Ron, Programming the 65816, 1986.

[Happy] HAPPY COMPUTERS, Inc., Happy 810 & 1050 Enhancement Warp Speed Software Rev. 7 User's Manual, 1986. Retrieved on January 30, 2017 from http://www.atarimax.com/members/happy/users.pdf.

[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.

[Lindbloom] Lindbloom, Charles, RGB/XYZ Matrices, 2017. Retrieved on November 10, 2019 from http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html.

[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.

[OSManual] Atari, Atari 400/800 Operating System User's Manual, 1982.

[RamboXL] ICD, RAMBO XL Installation & Operations Manual, 1986. Retrieved on December 20, 2015 from http://www.atarimania.com/documents/rambo_manual.pdf.

[RFC1071] Braden, R., Borman, D., and C. Partridge, Computing the Internet checksum, 1988. Retrieved on July 18, 2015 from http://www.rfc-editor.org/info/rfc1071.

[SlidingDFT] Jacobsen, Eric and Lyons, Richard, The Sliding DFT, IEEE Signal Processing Magazine, 2003.

[SMPTE 170M-2004] Society of Motion Picture and Television Engineers, Television - Composite Analog Video Signal - NTSC for Studio Applications, 2004.

[SY6545-1-AN3] Synertek, SY6545 CRT Controller Applications Information AN3, 1980. Retrieved on April 19, 2020 from http://archive.6502.org/appnotes/synertek_an3_6545_crtc.pdf.

[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.

Feedback polynomials

A polynomial counter is composed of a shift register and a feedback network generating new bits to shift into that register. The feedback network combines multiple bits at various *taps* on the shift register using an XOR or XNOR function, usually two or four. This then causes the shift register to “count” through various states. The sequence produced, however, is not in numerical order as it is for a binary counter.

The feedback network can be represented as a characteristic feedback polynomial composed of powers x^i, where each power represents a specific tap off the shift register. For instance, a polynomial for a 4-bit counter 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. Thus, the following sequence is generated: 000100110101111.

As noted earlier, XNOR may be used instead of XOR, where XNOR is the inverted result of XOR. XNOR produces an equivalent polynomial counter, except that the feedback and shift register states are inverted.

Lock-up state

A polynomial counter always has one state that is not generated by the counter sequence. This state is all zeroes for an XOR-based counter and all ones for an XNOR-based counter. The feedback network produces the same bit from this state. This called the *lock-up state*, because the counter will never advance from this state.

The presence of the lock-up state means that polynomial counters must have a reset mechanism to force at least one bit of the non-lock-up polarity into the shift register. Only one bit is necessary since there is only one lock-up state and reaching any of the other 2^N-1 states is sufficient to re-enter the full cycle.

Generated sequence

Typically, the polynomial is chosen to produce a *maximal-length* sequence of 2^N-1 bits for an N bit counter, cycling through all possible states except one. The missing value is due to one state not being generated in the sequence, the all zero state for XOR feedback. 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.

A polynomial counter does not have to use a maximal-length sequence. Instead, it may use a polynomial which causes the sequence to repeat with a shorter cycle using a subset of states, or additional logic to reset the counter midway through the cycle. POKEY and GTIA contain counters with non-maximal sequence lengths of 28 and 114 from 5-bit and 7-bit counters, respectively. In this case, all of the non-lockup states that are not part of the looping sequence will eventually lead into the loop.

Polynomial counter simulation

The polynomial for POKEY's 9-bit generator 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 mod (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 $x^{2\wedge i}$ mod M.
- Given a position P, multiply together the appropriate values of $x^{2\wedge i}$ mod 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.

For the XF551's DSDD format, SIO sectors 1-720 are on side 0 (bottom) and SIO sectors 721-1440 are on side 1 (top). The sectors are in reverse order on side 1, so SIO sectors 721, 722, 723... are on track 39 sector 18, track 39 sector 17, track 39 sector 16, etc.

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.

Capture range

Testing on a 1050 drive shows that the FDC is able to synchronize to bitstreams within approximately 83-127% of nominal bit period for FM and 86-122% for MFM. This margin is needed to accommodate spindle speed variations for both the reading and writing drives. If the writing drive is 5% slow and the reading drive is 5% fast, the FDC effectively sees a bit cell period 90% of nominal.

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 length (0=128, 1=256, 2=512, 3=1024), and finally two CRC bytes.

Only the two low bits of the sector length field are used by the FDC. Bits 2-7 are ignored.

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.

Different FDC models differ in whether the head/side field is checked. The 1771, 1795/1797, and 2795/2797 do not; the 1791/1793 and 2791/2793 can if requested in the command byte.

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.

Only the 1771 FDC supports all four types of address marks. The 179X/279X and 1772 only report the LSB and thus treat $F8 and $FA the same, and $F9 and $FB the same.

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 Python 3 code computes the disk CRC-16:

```
def crc16(data:bytes, initial_value:int = 0xFFFF) -> int:
    crc = initial_value

    # for each data byte
    for c in data:
        # add in next byte
        crc ^= c << 8

        # compute (crc << 16) mod P by multiplying by 2^8
        for i in range(0,8):
            # multiply CRC by 2
            crc <<= 1

            # modulo by polynomial
            if crc & 0x10000:
                crc ^= 0x11021

    return crc
```

As with other CRC algorithms, this can be computed more quickly by precomputing a table of 16-bit modulo values (v × x^{16} mod P) for all 256 byte values P:

```
crc16_table = [crc16(bytes([x]), 0) for x in range(0,256)]

def crc16_fast(data:bytes, initial_value:int = 0xFFFF) -> int:
    crc = initial_value

    for c in data:
        crc = ((crc << 8) & 0xFFFF) ^ crc16_table[c ^ (crc >> 8)]

    return crc
```

This CRC algorithm is also available as the crc_hqx() function in the binascii module:

```
>>> import binascii
>>> sd_sector1 = b'\xFE\x00\x00\x01\x00'
>>> ed_sector1 = b'\xA1\xA1\xA1\xFE\x00\x00\x01\x00'
>>> '{:02X}'.format(binascii.crc_hqx(sd_sector1, 0xFFFF))
'D2C3'
>>> '{:02X}'.format(binascii.crc_hqx(ed_sector1, 0xFFFF))
'EA2D'
```

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 on an enhanced density disk containing all $00s from the computer's perspective are $EA2D and $9A17.

Appendix C
Physical Tape Format

C.1 Signal encoding

The baseline encoding for a tape is binary frequency shift keying (FSK), using 3995Hz and 5327Hz as the FSK tones for 0 and 1. These are the frequencies generated by POKEY using the 64KHz clock with divisors of 8 and 6, respectively. The FSK encoding is vaguely phase coherent, as the two timers switch off toggling a common output flip-flop.

The standard bit rate for a tape is 600 baud, but somewhat higher or lower baud rates can be used with lower error tolerance, such as 900 baud. The FSK tones, however, cannot be changed as they must match decoding filters hardwired into the cassette tape unit.

This binary FSK encoding is used to encode the asynchronous serial output stream from POKEY. Thus, it uses the same byte framing of one start bit (0), 8 data bits in LSB to MSB order, and one stop bit (1). As it is an asynchronous stream, the start bit does not have to immediately follow the last stop bit and can be delayed by additional mark tone (‘1’ bit encoding). Tapes can also hold arbitrary binary data, however, as it can be written using the force break bit in the SKCTL register and read using the direct input bit in SKSTAT.

As a result of the FSK encoding and tones used, the serial stream can be visualized in an audio editor’s spectrogram mode. With a sufficiently narrow window and appropriate gain and frequency range parameters, the digital serial stream will directly appear in the spectrogram.

C.2 Framing

By convention, tapes written by the standard OS C: handler use a data frame format consisting of a leader tone or inter-record gap, sync bytes, control byte, data frame, and checksum. However, none of this framing is required by the hardware. The hardware only requires a baud rate in range with the correct FSK tones and supports an arbitrary bit stream.

C.3 FSK demodulation

In order to extract data from tape audio, the hardware or an equivalent software decoder must decode the FSK encoded audio back to digital form. There are many standard ways to decode FSK, but the general idea is to use a pair of detectors to estimate the presence of the FSK tones and to choose the dominant one with a discriminator, producing a 0/1 bit decision.

C.4 Zero crossing detection

The simplest method of FSK decoding is to keep track of every time the waveform crosses zero level. For each complete cycle of a square wave or sine wave, the waveform will cross twice, with opposite slopes each time. The distance between crossings of the same slope will then give the frequency, which can then be converted to a bit by checking if it is closer to the FSK tone frequencies (3.9KHz and 5.3KHz).

Zero crossing detectors are simple, but have a number of drawbacks. They are sensitive to both DC bias and variations in amplitude, which can distort the individual timings of each cycle, though that variation will average out over many cycles. More seriously, zero crossing detectors can fail when significant noise or distortion are present, which can cause the wave to cross zero level more than once per cycle. Pre-filtering the waveform can help somewhat, but in general zero crossing detectors are not very reliable for decoding actual tape recordings.

C.5 Peak detection

Another decoding method is to detect peaks in the waveform by checking for when the slope reverses. For a sine wave or other relatively smooth wave, there will be two opposite peaks per cycle, which can be used to measure cycles similarly to a zero crossing detector. This method is used by some early WAV2CAS utilities. In practice, some tolerance is required to avoid extra false triggering at peaks due to noise or high-frequency harmonics.

Peak detection has the advantage of being less sensitive to DC and low frequency noise, though it is still subject to issues with high frequency noise and distortion. It has another advantage, which is that it is somewhat better able to detect the original SIO waveform than a zero crossing detector. This is because analog signal effects tend to transform the original square wave into more of a triangle wave, with the peaks roughly corresponding to changes in the SIO data level. This is of not much benefit for FSK decoding, but is more of an advantage for turbo decoding where lower frequencies are used and the precise timing of each transition is more critical.

C.6 DFT detection

A more sophisticated method in practice is to use a pair of filters to detect and track the amplitude of the mark and space frequency signals, using a comparator to decide between a 0 and 1 bit based on the stronger signal. This provides much better rejection of noise due to the filters only being sensitive to the signal frequencies around the FSK tones. It is also more similar to the circuitry in the 410/1010 tape drives, which use analog filters and comparators in a similar manner.

There are many kinds of filters that can be used to detect the FSK tones. Two considerations for choosing the filters are the bandwidth and response time of the filters. To successfully decode a 600 baud stream, the filters need to be sensitive to at least ±300Hz and have a response time below a single bit (1.67ms).

Frequency binning

The discrete Fourier transform (DFT) can be used to separate the signal into frequency with two of the bins centered on the FSK tones. This is possible because the tones are generated by POKEY by dividing the clock by 28×16 and 28×12, with the result that with an appropriate sampling rate and DFT size, the tones exactly match two of the frequency bins. Using a sampling rate of 31.96KHz (1.79MHz ÷ 56) and a DFT size of 24 points results in a frequency bin width of 1.33KHz, which centers bins 3 and 4 on the FSK frequencies of 4KHz and 5.3KHz. The magnitude of the responses in these two frequency bins can then be subtracted, with the resulting sign determining the output bit.

DC bias and low frequencies are naturally rejected by this technique as they are separated out into lower bins that are ignored or simply not computed. However, the reliance on a fixed sampling rate means that signals at other rates, such as the more common 44.1KHz and 48KHz audio sampling rates, must be resampled first.

Computing a sliding DFT

To provide sufficient resolution for pulse edges, the DFT needs to be computed on a per-sample basis. Doing this is expensive with a direct DFT. It can be sped up with an FFT, but doing so is complicated by non-power-of-two transform sizes.

A more efficient way to do so is via a sliding DFT, which takes advantage of the redundancy between transforms one sample part to do a cheaper update from one sample position to the next. Given the direct formula for computing a given frequency bin, both on one sample and the next:

$$X_k(i)=\sum_{n=0}^{N-1} x_{i+n} e^{-j2\pi\frac{kn}{N}}$$

$$X_k(i+1)=\sum_{n=0}^{N-1} x_{i+n+1} e^{-j2\pi\frac{kn}{N}}$$

It is possible to rewrite the expression for the result of the next sample in terms of the previous:

$$\begin{aligned}
X_k(i+1) &= e^{j2\pi\frac{k}{N}} \sum_{n=0}^{N-1} x_{i+n+1}\, e^{-j2\pi\frac{k(n+1)}{N}} \\
&= e^{j2\pi\frac{k}{N}} \sum_{n=1}^{N} x_{i+n}\, e^{-j2\pi\frac{kn}{N}} \\
&= e^{j2\pi\frac{k}{N}} \left[\sum_{n=0}^{N-1} x_{i+n} e^{-j2\pi\frac{kn}{N}} + (x_{i+N} - x_i) \right] \\
&= e^{j2\pi\frac{k}{N}} \left[X_k(i) + (x_{i+N} - x_i) \right]
\end{aligned}$$

This is a recurrence relation that updates a frequency bin by adding the new sample, removing the sample N points back, and then rotating the bin value by one sample (360/N degrees). As usual for the DFT, the value X_k for the frequency bin is a complex number, which is then converted to a magnitude by computing the vector length. Tracking and comparing the magnitudes of the X_3 and X_4 bins at each sample then determines whether the output should be a 0 or a 1.

Advantages of the sliding DFT include easy use of non power of two transform sizes and O(1) cost for each update independent of the transform size. However, the above equation has an issue with error accumulation due to the cumulative rotations. As the comparator only needs the magnitude of each frequency bin and is independent of phase, this can be addressed by counter-rotating the frequency bins at each step to cancel the overall rotation, which instead results in only needing to rotate the addend:

$$X_k(i+1) = X_k(i) + (x_{i+N} - x_i) e^{-j2\pi\frac{ik}{N}}$$

There is an issue with error accumulation when this is implemented in floating point, since floating point arithmetic is not associative and so a sample’s contribution is not guaranteed to cancel exactly. However, if fixed point is used, the terms for each sample will exactly cancel when added and then later removed from the sliding window. The final update per bin requires three additions and two multiplications, along with an N-sample delay line and a precomputed table of N rotation values. This is much cheaper than a 24-point FFT for the two bins required for FSK decoding.

An alternative to the sliding DFT is the sliding Goertzel DFT, which combines a sliding window with a Goertzel filter.[84] It is cheaper than the sliding DFT when computing bins with the correct phase, but more expensive when only magnitude is needed. Stability and controlling error accumulation are also more difficult due to the IIR filter structure.

Windowing

The above effectively calculates the DFT with a rectangular window, where the transform abruptly starts and stops at the edges of the window. Although the ideal FSK frequencies align exactly with frequency bins as long as the transform size and sampling rate are chosen appropriately, in practice there is a little bit of spectral leakage which causes a small amount of response to be leaked to or from frequencies far away from the FSK tones.

Windowing is normally used to combat this problem, ramping the signal down to zero or near-zero at the edges of the transform. The Hann window, for instance, multiplies the signal by a raised cosine before the DFT. This is difficult to do with a sliding DFT, however, because the window value for each sample changes as the transform advances over it. This can be worked around by instead applying the window in the frequency domain, since multiplication in time domain is equivalent to convolution in frequency domain. For the Hann window, this is a

[84] [SlidingDFT]

simple three-tap filter:

$$X_k' = \frac{1}{2} X_k - \frac{1}{4}(X_{k-1} + X_{k+1})$$

This requires additional frequency bins to be tracked. It is also possible to implement Hamming and Blackman windows this way. Note that the complex values of the bins must be combined, not the magnitudes.

More troublesome is that this technique doesn't directly work with the rotation trick above, since the X_k frequency bin values are out of phase relative to each other compared to a standard DFT and can't be directly combined this way, requiring additional rotation corrections. There is some optimization that can be applied for adjacent bins computed together as they will share unwindowed bins and phase corrections in their calculations. The cost for updating Hann-windowed X_3 and X_4 values is 24 additions and 16 multiplications.

There is another side effect of windowing, which is reduced resolution as the frequency bins cover wider ranges of frequencies below the low FSK tone and above the high FSK tone. On the other hand, the responses also taper smoothly to zero instead of ringing as with the rectangular window.

In practice, windowing doesn't seem to have a significant effect on FSK decoder effectiveness, and so it's easier and cheaper just to omit it.

C.7 Quadrature demodulation

Another FSK decoding method is quadrature demodulation:

- Multiply the signal by a complex sine wave half way in frequency between the FSK tones (4.66KHz) to shift the spectrum down such that the FSK tones are symmetrical around 0Hz at ±0.67KHz.
- Apply a low pass filter to limit the signal to the desired bandwidth around each tone (or equivalently, apply a band pass filter before the previous step). This excludes frequencies far away from the FSK tones, as well as aliasing from the first step.
- Multiply the resulting signal by the complex conjugate of itself, offset by one sample. This effectively computes the phase delta between each sample, which in turn indicates whether the fundamental frequency is above or below the 4.66KHz midpoint.
- Compute the phase of the resulting delta vectors, outputting a 0 bit for negative phase deltas and 1 for positive phase deltas. This can be done without polar conversion by taking the sign of the imaginary component.

Note that all steps above must be done in complex arithmetic, including the frequency shift and the low pass filter.

Quadrature demodulation has some advantages over the sliding DFT technique. Arbitrary sampling rates are easier to accommodate as there is no requirement for a transform window fitting an integer number of cycles, and the bandwidth around the FSK tones can be varied by adjusting the cutoff frequency of the low pass or band pass filter(s). It is somewhat more expensive than the sliding DFT, with the majority of the cost being in the complex valued LPF/BPF – which is around at least 15 taps at 32KHz.

In practice, quadrature demodulation seems to be less effective than the sliding DFT, especially with degraded tape signals. Portions of the signal with dropouts cause problems as the computed phase from the demodulator goes haywire as the signal drops to zero or near-zero, which is handled more gracefully by the DFT. It is less sensitive to amplitude differences between the mark and space tones, however, which can otherwise require manual tuning or an AGC.

C.8 Asynchronous serial decoding

Once the audio has been demodulated from FSK to digital form, standard asynchronous serial decoding can be applied to extract a byte stream. A common way to do so is as follows, which is similar to what POKEY does:

1. Wait for a 1 → 0 edge that indicates the possible leading edge of a start bit.
2. After one-half bit period, sample the center of the potential start bit. If it is not a 0, reject the start bit and go back to waiting for the next leading edge.
3. Sample 8 data bits at successive bit periods from the center of the start bit.
4. Sample the stop bit after one more bit period. If it is not a 1, a framing error has occurred.

This assumes that the digital stream is reasonably clean and noise-free. If this is not the case, then prefiltering must be applied beforehand to de-glitch the stream. One way to do this is a filter that only changes its output once the input has a minimum continuous period of the same bit, or alternatively above a threshold of same bits within a window (e.g. at least 75% within 1ms).

Note that this technique assumes that the baud rate is already known within a narrow threshold (±5%), which must be agreed upon beforehand or determined by other means. In the standard OS C: format, this is done by counting the width of pulses in the two $55 sync bytes at the beginning of each record.

Appendix D
Analog Video Model

D.1 Introduction

The NTSC system originally started as a black-and-white only system and later had color support added to it in a compatible manner, so that existing receivers could still display the black-and-white portion of color broadcasts. This was done through the addition of a *chroma subcarrier*, which was added to the existing luminance signal to supply color information.

The manner in which GTIA generates video signals is tailored to this encoding. Unlike more modern systems that internally use RGB colors and then encode to composite video, GTIA generates colors directly into luminance and chrominance signals. The GTIA palette therefore has no RGB basis, and one can only be determined by decoding it from the generated composite video signals. Later, GTIA was adapted to PAL and SECAM, with changes to adapt the color generation for those systems.

There has been a lot of confusion about GTIA palettes, and particularly misapplication of PAL color palettes to software and emulators of NTSC systems. This appendix is a description of an model of the end-to-end video pipeline and an attempt to accurately model the complete conversion from GTIA color register values to final displayed RGB colors on a modern display.

D.2 NTSC color encoding

Luminance and chrominance

In the NTSC system, color values are split into luminance and chrominance. The luminance value is the brightness or grayscale value, and is simply encoded as a signal with luminance mapped to voltage. Chrominance is the color information ignoring luminance. Typically the luminance value is named Y in equations while chrominance has two values which can be I and Q or U and V depending on the color space used. Together, the three values form a 3D coordinate in YIQ or YUV space, which is different from but can be mapped to and from RGB space.

Color subcarrier

The two chrominance values are encoded into a single color signal through *quadrature amplitude modulation (QAM)*, encoding both values into a single sine wave at 3.58MHz. This signal is called the color subcarrier. The phase of the sine wave determines the hue of the color, while the amplitude determines the saturation. For areas of the picture that have no color, the amplitude is 0 and the subcarrier is absent.

The rate at which the NTSC GTIA produces pixels is directly related to the color subcarrier: the width of one lo-res pixel is one cycle of the color subcarrier or color clock, and a hi-res pixel is half a cycle. Thus, there are exactly 228 cycles of the color subcarrier per scanline.

YIQ color space

The color subcarrier can be represented as the sum of two component sine waves, one 90° out of phase with the other. The amplitudes of these two components are called in-phase (I) and quadrature (Q), and can represent the color subcarrier at any arbitrary amplitude *r* and phase ϕ:

$$chroma = r\cos(2\pi f_{sc} t + \phi) = I\cos(2\pi f_{sc} t) + Q\sin(2\pi f_{sc} t)$$

The two representations are equivalent. On a 2D chrominance chart, the I/Q coordinates are the Cartesian coordinates of the point with (r, ϕ) as the polar coordinates, which also correspond to saturation and hue. Adding in Y gives a coordinate in the YIQ color space.

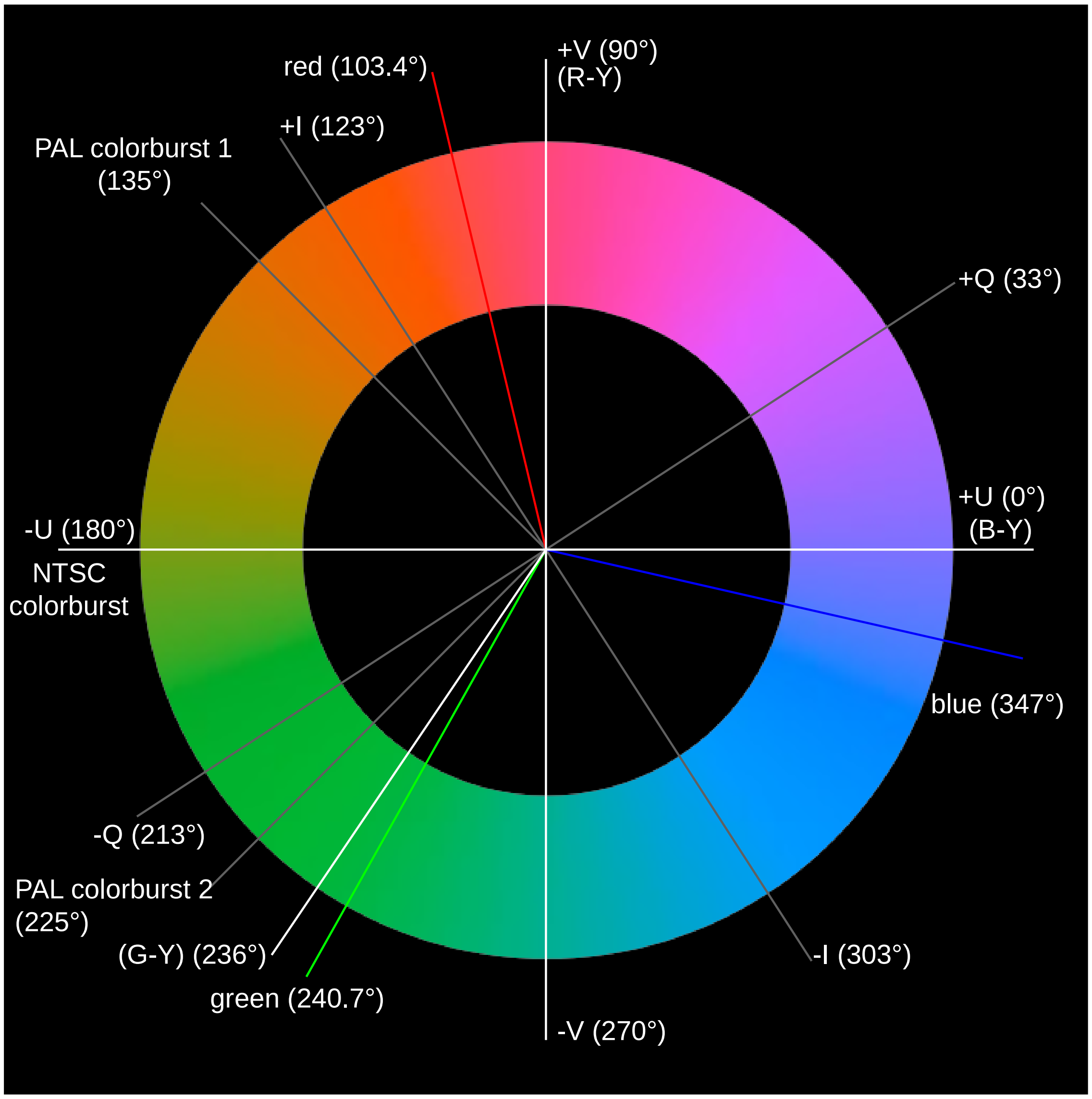


Figure 14: YIQ/YUV color space relations

The YIQ color space is defined in terms of the RGB color space, with an affine transform converting converting between them. To three decimal places, the decoding and encoding conversions are as follows:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.107 & 1.704 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

For R/G/B values in [0, 1], the Y range is [0, 1], I range is [-0.596,0.596], and Q range is [-0.523, 0.523].

The exact conversion matrices vary due to specifications giving the constants to different precision and also use of non-standard matrices in the field. An extended discussion of the derivation of these matrices and variance in specification is in [SMPTE 170M-2004].

YUV equivalence

An alternate color space that can be used is YUV, defined as follows:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.140 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

YUV has a simpler definition of the chroma axes, with U being directly proportional to the blue color difference (B-Y) and V to the red color difference (R-Y), as indicated by the zeroes in the decoding matrix. This simplifies decoding, with only the green difference (G-Y) requiring full matrix treatment.

However, as YUV is related to RGB by an affine transform, it is also similarly related to YIQ, and conversion between them simply requires a 33° rotation:

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} -\sin 33° & \cos 33° \\ \cos 33° & \sin 33° \end{bmatrix} \begin{bmatrix} I \\ Q \end{bmatrix}$$

This means that generating colors in YIQ space and YUV space are equivalent, as long as the 33° angle difference is taken into account. Some TVs also use this to simplify decoding hardware by directly decoding NTSC composite video to YUV, directly extracting blue and red difference signals from demodulation and combining those to get the green difference signal.

Note that the definition of YUV here should not be confused with the “YUV” sometimes used to refer to component or digital video, also known as YCbCr. These equations are often found for converting from YUV to RGB, and should not be confused with the ones used for analog video, as they are based on similar concepts but use additional scaling and bias factors for the components:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164 & 0 & 1.596 \\ 1.164 & -0.392 & -0.813 \\ 1.164 & 2.017 & 0 \end{bmatrix} \begin{bmatrix} Y-16 \\ Cb-128 \\ Cr-128 \end{bmatrix} \quad \text{(Rec. 601 limited range (SD))}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.344 & -0.714 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ Cb-128 \\ Cr-128 \end{bmatrix} \quad \text{(Rec. 601 full range (JPEG JFIF))}$$

Similarly, a matrix using [0.2126 0.7152 0.0722] as the luminance (Y) axis in RGB space is for the Rec. 709 color space for HD, and not applicable to NTSC decoding. It is possible to map YIQ or YUV values to one of these YCbCr spaces, but doing so takes a color matrix comparable to conversion to or from RGB.

Luminance encoding

The luminance (Y) portion of the signal is generated by GTIA off of the brightness portion of the color register or color value providing a pixel's color. This conversion is done outside of GTIA through a simple resistor DAC, which linearly converts the brightness value to equal voltage steps. A simple and serviceable approximation is therefore just:

$$Y = \frac{brightness}{15}$$

A more accurate conversion takes into account a quirk in the DAC which results in brightness values 7 and 8 being noticeably closer than expected. This is due to slightly mismatched resistor values in the DAC. LUM0 and LUM1 use resistor values 36KΩ and 18KΩ, which are matched by a factor of two as expected, and LUM2 uses 9.1KΩ, which is only a small bit off. LUM3 uses 4.7KΩ, however, which is off by a larger amount and results in brightness values 8-15 being visibly darker. Table 91 gives measured signal levels on three different NTSC machines.

	Signal levels (+/-0.008V)			
Element	**800XL**	**130XE**	**800**	**400**[85]
Sync	0.580V	0.044V	0.080V	3.604V
Luma 0	1.112V	0.468V	0.640V	3.736V
Luma 1	1.184V	0.532V	0.696V	3.752V
Luma 2	1.264V	0.596V	0.744V	3.768V
Luma 3	1.334V	0.660V	0.800V	3.780V
Luma 4	1.408V	0.716V	0.856V	3.792V
Luma 5	1.488V	0.780V	0.904V	3.808V
Luma 6	1.568V	0.852V	0.960V	3.820V
Luma 7	1.648V	0.908V	1.016V	3.836V
Luma 8	1.688V	0.940V	1.056V	3.848V
Luma 9	1.760V	1.004V	1.112V	3.864V
Luma 10	1.840V	1.068V	1.160V	3.876V
Luma 11	1.928V	1.132V	1.224V	3.892V
Luma 12	1.992V	1.196V	1.272V	3.904V
Luma 13	2.072V	1.260V	1.328V	3.920V
Luma 14	2.144V	1.324V	1.384V	3.936V
Luma 15	2.224V	1.380V	1.440V	3.952V

Table 91: Measured signal levels for sync and all luminances

An additional complication arises from the presence of a *setup* or *pedestal* in NTSC. The NTSC video standard specifies that the signal level for black during the active period of a scanline is higher than the 0V used during blanking periods. This is the *pedestal*. In the conventional IRE unit used to measure video signal levels, the blanking level is 0 IRE, the pedestal or black level is 7.5 IRE, and white level is 100 IRE.

GTIA does not use the pedestal when encoding its signal, which has implications for the encoded color. Instead, it maps luminance level 0 directly to blanking level at 0 IRE, which causes it to be “blacker-than-black.” The result is that GTIA colors are slightly darker than expected at the low end, and additionally causes some noticeable hue shifts in the darkest color due to negative RGB outputs clamping to zero. Mapping brightness 0-15 to the 0-100 IRE range gives a brightness 0 luma value of -8%. However, actual results may vary on TVs, as some use non-standard or dynamic black levels. The Japanese variant standard NTSC-J also lacks setup, for instance.

[85] The 400 does not have a composite output, which makes this measurement tricky. These measurements were taken with a probe on R203 before the RF modulator and low pass filtering enabled on the oscilloscope to filter out the audio subcarrier.

NTSC GTIA color encoding

The NTSC GTIA chip generates colors by mapping the hue portion to the phase of the chroma signal, with no chroma signal being generated for hue 0. All other hues generate chroma with the same amplitude, so saturation is the same for all hues.

The phase of the chroma signal is determined by a delay chain within GTIA, which produces 15 different phases by successive delays from a reference clock, with each higher numbered hue producing an additional phase delay. The delay is the same for each stage and thus all hues are evenly spaced in angle, but the spacing is determined by a color adjustment potentiometer on the motherboard. Thus, the 15 hues do not necessarily cover the full 360° evenly, as the cumulative phase delays to hue 15 may result in that color being short or even wrapping around past hue 1. The color delay can be adjusted all the way down to 0°, but is typically somewhere within 23.5° to 27.5° per hue.

Hue 1 is also special because it doubles as the phase for the *color burst*, a small portion of color signal emitted in horizontal blank to synchronize the display to the correct phase for the color signal. The color burst is defined to be at 180° (-U) in color space, or -57° behind I in I-Q space. This means that the color of hue 1 is deterimined solely by the display and not by the color delay adjustment.

Hue values can therefore be mapped to I-Q space by a simple polar mapping:

$$\begin{bmatrix} I \\ Q \end{bmatrix} = saturation \times \begin{bmatrix} \cos(-57° + delay \times (hue-1)) \\ \sin(-57° + delay \times (hue-1)) \end{bmatrix}$$

Or, equivalently, in U-V:

$$\begin{bmatrix} U \\ V \end{bmatrix} = saturation \times \begin{bmatrix} \cos(180° - delay \times (hue-1)) \\ \sin(180° - delay \times (hue-1)) \end{bmatrix}$$

The tint control on the display normally rotates the hue space, shifting the colorburst color and all other colors by the same amount. The -57° specification used by calibrated and modern TV sets gives a greenish yellow color to hue 1. This may differ on TV sets that have had their tint controls adjusted or were tuned differently from the NTSC standard. For instance, the Commodore C1702 monitor has its tint control centered at a location that produces a hue 1 color closer to around -33°. The notion of a "correct" contemporary display look is very subjective, but research into color selections on various games by members on AtariAge has suggested that many NTSC games were developed on displays at the time that had more reddish hue 1 colors than produced by modern displays.

Saturation

The saturation of the color produced is determined by amplitude of the chroma signal. This amplitude varies with different computer models, but this variation is cancelled by the chroma automatic gain control (AGC) in the display as the chroma amplitude is interpreted relative to the color burst, which also has the same amplitude. The color burst is specified in NTSC as having a peak-to-peak amplitude of 40 IRE, so the magnitude of the color vector in I-Q space is 20 IRE.

However, interpreting this value is complicated by the various models having different luma ranges, none of which place white at 100 IRE. One method of AGC corrects the luma using the sync pulse as a reference, which is supposed to have a signal level of -40 IRE. Using the signal levels in Table 91, the saturation can be derived as a ratio from the chroma vector in I-Q space to the full 0-15 brightness value range:

	800XL	130XE	800	400
Sync to luma 0 voltage (IRE -40 reference)	-0.532V	-0.424V	-0.560V	-0.132V
Luma 0 to 15 voltage	1.112V	0.912V	0.800V	0.216V
AGC corrected luma 15 level	83.6 IRE	86.0 IRE	57.1 IRE	65.4 IRE
Saturation	23.9%	23.3%	35.0%	30.6%

Table 92: NTSC saturation ratios

That is, on the 800XL, the chroma and luma AGCs produce a chroma signal that is 23.9% of the total range of luma from brightness 0 to brightness 15. Note that this ignores the pedestal, which biases Y during conversion to RGB but does not affect saturation when computing I/Q or U/V.

The 800's luma signal is noticeably weaker than that of the XL/XE models, which causes it to have much more saturated colors.

Gamma correction

NTSC color encoding is designed to work with the transfer characteritics of CRT cameras and displays, and therefore use non-linear RGB encoding. This means that conversions between YIQ/YUV and RGB should be performed directly in gamma space; they should not be performed in linear color and then converted to gamma space.

The reference gamma for NTSC is specified either as either an exponent of 2.2 or a short linear section near zero combined with a 2.22 curve. This is similar to but not quite the same as the sRGB color space now ubiquitous on modern computers, which uses a short linear section of different length followed by a 2.4 curve. Converting from NTSC gamma space to linear and back to sRGB produces the most correct results, although the difference is not significant.

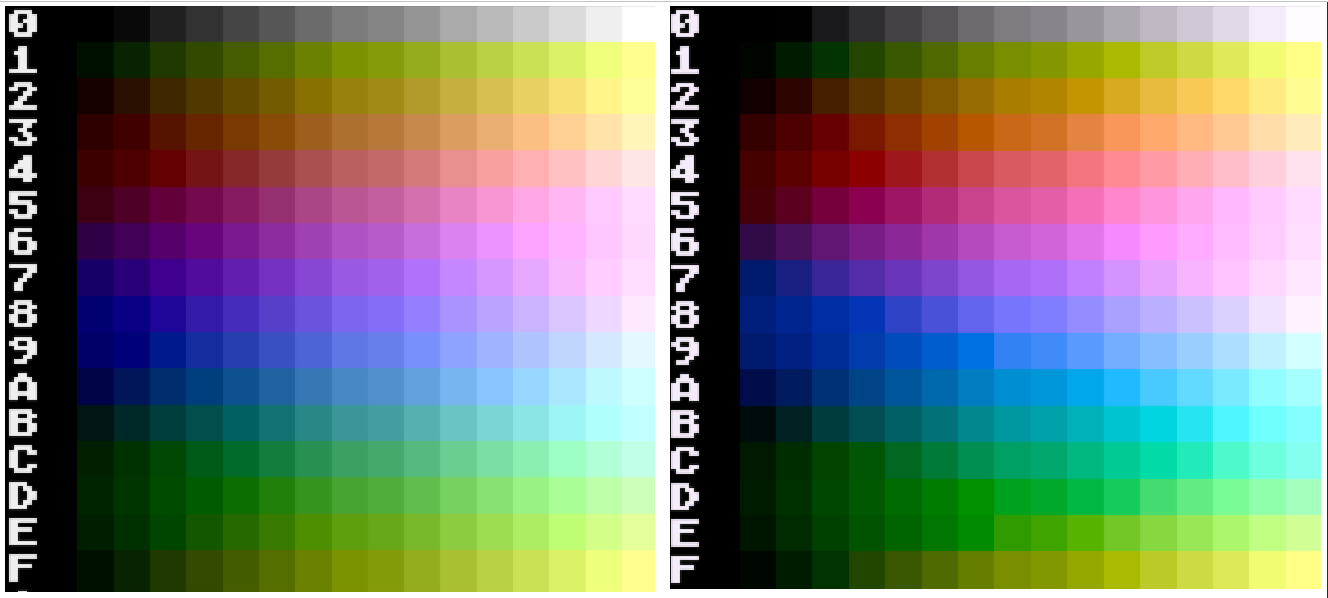


Figure 15: Effect of color correction on computed palette

The left image is a palette computed with the result of the YIQ to RGB conversion directly interpreted as sRGB, the right image is with correction from NTSC primaries to sRGB. The main effect is to boost the contribution from blue, which reduces makes the $94 GR.0 background color less purple, and to reduce contribution from green. Saturation at the low end is also boosted somewhat.

Color correction

A subtle source of error in NTSC color reproduction is the difference in color primaries between NTSC and modern sRGB displays. Failing to correct for this produces small color shifts in some critical portions of GTIA's palette, most notably affecting the GR.0 blue color.

One method of correcting for this difference involves converting from NTSC RGB to sRGB through a common color space, such as XYZ. This requires converting from NTSC gamma to linear, multiplying by two affine matrices to convert from NTSC linear RGB to XYZ and then to linearized sRGB, then to regular sRGB. A set of conversion matrices to accomplish this is given by [Lindbloom].

Figure 15 shows the effect of applying this type of correction to the final color palette. This helps improve the accuracy of the palette as it would be displayed on a modern TV set. The necessary conversion may differ when simulating a contemporary monitor, however, as older monitors may use different primaries than the original NTSC specification.

Non-standard decoding matrices

Another method used in practice for compensating for the differing color primaries is with a non-standard decoding matrix. The standard decoding matrix places the (R-Y) and (B-Y) color differences at right angles to each other in chrominance space, with U and V orthogonal. This is not universal practice, and many NTSC decoding chips have been specifically designed to use non-standard matrices that produce different decoding angles.

> **Note**
>
> The angles for the (B-Y) and (R-Y) color differences aren't the same as the angles for pure red and blue. Red and blue have non-zero luma and thus both non-zero (R-Y) and (B-Y).

The hue angles and gain values for the RGB difference signals can be computed by converting the U-V basis vectors in the YUV-to-RGB conversion matrix to polar form:

$$\theta_{R\text{-}Y} = atan2(1.140, 0) = 90°$$
$$\theta_{G\text{-}Y} = atan2(-0.581, -0.395) = 235.8° - 360°$$
$$\theta_{B\text{-}Y} = atan2(0, 2.032) = 0°$$

These are sometimes specified in hardware datasheets for decoders as relative to B-Y, placing R-Y at 116.3° and G-Y at 253.6°. Two of these can be customized; adjusting the third is equivalent to modifying tint or the reference color angle.

The gain values determine the amplitude of the decoded color differences, and are determined by the magnitudes of the basis vectors:

$$gain_{R\text{-}Y} = |(1.140, 0)| = 1.140$$
$$gain_{G\text{-}Y} = |(-0.581, -0.395)| = 0.703$$
$$gain_{B\text{-}Y} = |(0, 2.032)| = 2.032$$

As with angles, two of these gain values can be customized as adjusting the third is redundant with overall saturation. Because of this, the gain values are sometimes specified as ratios against either the B-Y or R-Y gain:

$$gain_{G\text{-}Y} / gain_{R\text{-}Y} = 0.703/1.140 = 0.617$$
$$gain_{B\text{-}Y} / gain_{R\text{-}Y} = 2.032/1.140 = 1.782$$
$$gain_{R\text{-}Y} / gain_{B\text{-}Y} = 1.140/2.032 = 0.561$$
$$gain_{G\text{-}Y} / gain_{B\text{-}Y} = 0.703/2.032 = 0.346$$

Color clamping

The standard color matrices will produce colors that are outside of the RGB color cube, and in particular negative color values. These should generally be hard clamped to zero, without luminance compensation or soft clamping. A contemporary CRT display neither had delay memory for carrying such negative colors across scan lines nor could produce negative light through its CRT tube, so the negative values were simply clamped to black. This necessarily produces hue shifts for colors with very low luminance values, which is representative of such displays. In a 256 color chart, this manifests as sine-wave shaped artifacts on the left and the right where the colors intersect the RGB cube and the hues are shifted by the clamping.

Some modern displays may show effects where some of these negative colors are partially preserved. This has the most impact with alternating GTIA mode 9/11 line displays in APAC-like modes, which ordinarily look dull on NTSC due to negative colors being lost on the dark mode 11 lines, but appear with full colors on some devices. This is caused by comb filters or downsampling filters within the device working in U-V, I-Q, or RGB-Y space and allowing negative colors to be blended before clamping. Video capture devices will also cause this effect when configured to capture with a format that has vertical chroma subsampling, such as 4:2:0 YCbCr (YV12) or 4:1:0 YCbCr (I420). This can sometimes be avoided by capturing in a format with full chroma resolution, such as RGB or 4:2:2 YCbCr (YUY2/UYVY).

Over-bright colors are more troublesome and hard clamping typically produces objectionable color shifts, particularly ugly yellows. Tone mapping techniques like soft clamping typically produce bland results. The only generally good way to deal with excessively bright colors is to reduce the overall contrast to bring them within the

RGB gamut without distorting the color palette. Raising the overall brightness of the display is therefore beneficial to faithful reproduction of NTSC colors. However, this is often a compromise since the overall color levels need to be brought down as low as white=70% to avoid clipping, which is generally too dim unless the monitor brightness is increased.

The purely correct way to reduce color intensity is in linear color, which requires converting from sRGB or NTSC to linear RGB, scaling there, and then converting to sRGB. Practically, colors can be scaled imperceptably directly in sRGB space. This convenient cheat works because sRGB's transfer function is close to gamma 2.2 and the scaling constant can be factored out to allow the conversions between gamma and linear color to cancel:

$$(k \times rgb^{gamma})^{(1/gamma)} = k^{(1/gamma)} \times (rgb^{gamma})^{(1/gamma)} = k^{(1/gamma)} \times rgb$$

For the reduced color intensity to have benefit, it must be done on extended color values. Scaling down color values that have already been clamped to 1 has no benefit as the extra saturation has already been lost.

Monochrome displays

Color signals can also be displayed on a monochrome display. This may be tinted bluish-white, amber, or green, but in all cases the display is produced by displaying the signal as luminance only without color decoding. This results in the 3.58MHz chroma subcarrier being visible to some extent, within the bandwidth and resolution limits of the display.

That the display neither reproduces nor decodes the chroma signal, however, does not mean that it is not visible. It is still visible as a haze or blurriness on the displayed picture, since the 3.58MHz chroma subcarrier is somewhat reproduced as an alternating pixel pattern on the display. The effect depends on the bandwidth and resolution of the display. High resolutions displays will simply reproduce the subcarrier as a striped pattern, while lower resolution displays will filter it somewhat. The subcarrier has no luma component if averaged at the signal level, but it does have a luma component if averaged in the display due to the non-linear encoding of the signal and averaging in light producing a different result than averaging in the signal.

The luminance change from the color signal can be simulated by converting the peak signal values to linear space, averaging the linear values, and converting back to gamma space. For instance, given a chroma signal with 40% amplitude relative to luma 15, displayed on top of luma 8 and using a gamma of 2.2, gives the following peak values:

$$v_{lo} = max\left(0, \frac{8}{15} - 0.40\right) = 0.133$$

$$v_{hi} = max\left(0, \frac{8}{15} + 0.40\right) = 0.933$$

$$v_{avg} = \left(\frac{(v_{lo}^{2.2} + v_{hi}^{2.2})}{2}\right)^{\frac{1}{2.2}} = 0.685$$

This is somewhat brighter than the pure luma 8 value of 8/15 = 0.533. However, this only applies for an infinite bandwidth decoder and averaging purely in light on the display; the actual value would likely be lower in practice due to some signal attenuation, which would be done on the gamma-encoded signal. The average effective luma is also higher for darker colors because of clamping in the display, which cannot produce negative light for the signal peaks below black.

D.3 NTSC artifacting

Hires pixels of alternating intensity produce a signal with a component at the color subcarrier frequency. This component is indistinguishable to the display from a regular color signal and produces color in the same manner.

The color produced is determined by the relative offset from the luminance pixels and the color subcarrier, which varies based on the computer model. Neither the luminance pixels nor the color signal produced by GTIA are clean sine waves, but only the phase of the sinusoidal component at 3.58MHz matches.

Model	Estimated offset (+/- 5 ns)	Hue delta (+/- 5°)
800	0.132 us	170°
800XL	0.251 us	323°
130XE	0.185 us	238°

Table 93: Measured artifacting delays

Luma/chroma separation

Because artifacting is the result of luma/chroma crosstalk, luma/chroma separation must be simulated in order to simulate artifacting. There are several strategies for separating luma and chroma from the composite signal.

One way to separate the signals is by frequency, using a bandpass filter centered at 3.58MHz to isolate chroma and the inverse, a notch reject filter, to isolate luma. A difficulty with this approach is selecting a good bandpass filter that doesn't result in excessive blurring of luma, especially when the filter has too shallow of a rolloff. Simply selecting a filter with sharp frequency response is not enough, as such a filter will also be very long in the time domain and have huge horizontal ringing artifacts.

There are a couple of ways to improve the quality of the separated luma. First, the luma and chroma filters should be matched so that there is no signal loss between them. This is most easily accomplished by subtracting the chroma from the composite signal to produce luma, equivalent to ensuring that the luma and chroma filters are perfect inverses.[86] Second, a post-separation peaking filter can be used that has increased gain around 3.58MHz, to combat attenuation of frequency components around the chroma subcarrier. This is a common option in NTSC decoding chips.

Modern decoders often use a comb filter to improve Y/C separation. A 1D comb filter works only horizontally or vertically, a 2D comb filter both horizontally and vertically within a frame, and a 3D comb filter also incorporates filters from adjacent frames (temporal axis). Comb filters can either use fixed or adaptive coefficients, the latter better able to deal with motion. The signal produced by ANTIC and GTIA invalidates traditional comb filters since the horizontal and frame periods are integral multiples of the color subcarrier, preventing the subcarrier from alternating phase and producing the checkerboard patterns targeted by the comb filter.

Chroma demodulation

The chroma signal must be further demodulated into components for input into the color decoding matrix. Either I-Q or U-V axes can be decoded, with appropriate adjustments to the decoding matrix. This is typically done by multiplying the signal by sine and cosine waves and applying a low-pass filter to the result.

Choosing the right sampling rate and phase can make demodulation easier. In particular, using a sampling rate of $4 \times f_{sc} = 14.3$ MHz and aligned to the color burst reference aligns the samples to the peaks of the sine and cosine waves in the repeating pattern -U, +V, +U, -V. Similarly, delaying by 57° gives +I, +Q, -I, -Q. This makes demodulation easier and also low-pass filtering, as it is easier to avoid beat patterns from imperfect filtering.

It is also possible to demodulate the RGB color differences by sampling at the peaks for R-Y, G-Y, and B-Y instead of I/Q or U/V. Evaluating the color differences with the modulated forms of I and Q to produce the encoding equations gives the phase angles:

[86] This is also referred to as a pair of *complementary filters*, where the transfer functions of the two filters sum to one, and the components removed by one filter exactly match those kept by the other.

$$\begin{aligned}
R-Y &= 0.956\,I+0.621\,Q \\
&= 0.956\cos(2\pi f_{sc}t)+0.621\sin(2\pi f_{sc}t) \\
&= 1.140\cos(2\pi f_{sc}t+33.0^{o}) \\
G-Y &= -0.272\,I-0.647\,Q \\
&= -0.272\cos(2\pi f_{sc}t)-0.647\sin(2\pi f_{sc}t) \\
&= 0.493\cos(2\pi f_{sc}t-123.5^{o}) \\
B-Y &= -1.107\,I-1.704\,Q \\
&= -1.107\cos(2\pi f_{sc}t)-1.704\sin(2\pi f_{sc}t) \\
&= 2.032\cos(2\pi f_{sc}t+123.0^{o})
\end{aligned}$$

This is essentially U-V decoding for the (B-Y) and (R-Y) components, differing only in the recovery of (G-Y) by direct demodulation. When working with discrete samples, this is less convenient than matrix decoding of (G-Y) from (B-Y) and (R-Y).

Finally, it is possible to demodulate RGB directly, by converting the color matrix into demodulation phase offsets:

$$\begin{aligned}
signal &= (0.299\,R+0.587\,G+0.114\,B) \\
&\quad +(0.596\,R-0.275\,G-0.321\,B)\cos(2\pi f_{sc}t) \\
&\quad +(0.212\,R-0.523\,G+0.311\,B)\sin(2\pi f_{sc}t) \\
&= R(0.299+0.596\cos(2\pi f_{sc}t)+0.212\sin(2\pi f_{sc}t)) \\
&\quad +G(0.587-0.275\cos(2\pi f_{sc}t)-0.523\sin(2\pi f_{sc}t)) \\
&\quad +B(0.321-0.321\cos(2\pi f_{sc}t)+0.311\sin(2\pi f_{sc}t)) \\
&= R\cdot(0.299+0.633\cos(2\pi f_{sc}t-19.5^{o})) \\
&\quad +G\cdot(0.587+0.591\cos(2\pi f_{sc}t-117.7^{o})) \\
&\quad +B\cdot(0.114+0.447\cos(2\pi f_{sc}t+135.9^{o}))
\end{aligned}$$

This is not recommended, though, since this mixes luma and chroma together, and the low pass filtering needed results in unacceptably poor quality luma.

Luma/chroma signal synthesis

The luma signal is produced by a filtered version of a square pulse, with a width of 1/3.58MHz = 279ns for a lores pixel and half that at 140ns for a hires pixel. The sharpness of the pulse determines not only the crispness of the pixels but also the strength of chroma artifacts at edges. The pulse is not a perfect rectangular pulse in practice and this attenuates chroma edge artifacts.

The chroma signal is formed similarly, except that the pixel pulses are also multiplied by the appropriate phase-shifted chroma subcarrier. The most straightforward way is to generate the sine wave directly with the required phase offset, rather than computing and modulating I and Q, which ends up being equivalent. Technically GTIA does not actually generate a sinewave, but rather a phase shifted square wave which is then (very) crudely filtered. This does not seem to be significant in practice due to the low bandwidth of the chroma signal.

Afterward, the luma and chroma signals can be combined to produce the composite signal, which is then demodulated to regenerate luma and chroma, but with the appropriate crosstalk to produce artifacting effects.

Filtering implementation

All filtering operations must take place in NTSC gamma space and not linear color, since this is how NTSC encoding is specified.

Because all of the operations listed so far are linear operations, some significant shortcuts can be taken in impementation. It is not actually needed to generate a composite signal, as the luma and chroma signals can be demodulated and filtered separately, producing luma-to-luma, luma-to-chroma, chroma-to-luma, and chroma-to-chroma signals. This is convenient given the hue-brightness separation inherent in GTIA and produces equivalent results. Also, this allows for some of the crosstalk effects to be attenuated if desired, with the chroma-to-luma crosstalk being the least desirable. Combining this with the brightness-to-luma and hue-to-chroma conversions at the front and the color matrix at the back produces streamlined brightness-to-RGB and hue-to-RGB transforms, dramatically reducing the number of filtering steps.

The types of filters used has impact on the visual attractiveness of the output. In particular, linear phase filters give a more modern look as they are only truly possible with digital signal processing, producing ringing artifacts that are centered around the originating pixels. Older CRTs used analog filters that had phase shifts, with ringing artifacts extending mainly forward in time, to the right in the picture. For simulating this, either IIR filters or truncated IIR filters converted to FIR filters are preferable. These do not have to be very long -- at a 14MHz dot clock (double hires), filters in the 12-16 point range are adequate, with longer filters either being lost in roundoff or producing unnaturally long ringing artifacts.

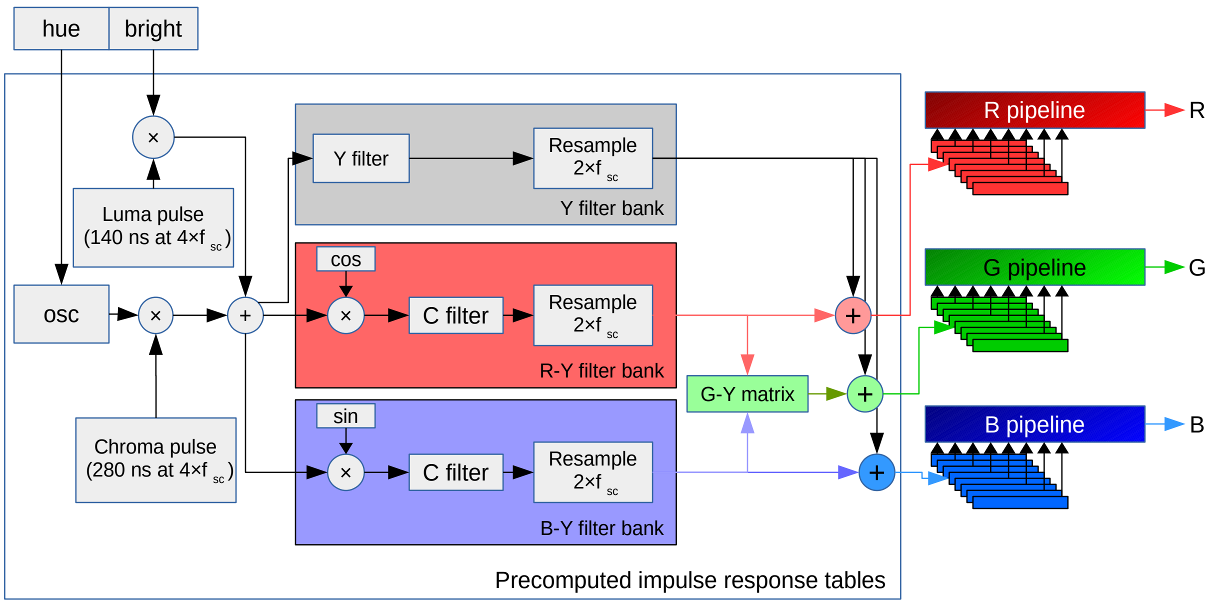


Figure 16: Altirra NTSC high artifacting pipeline

The 8-bit GTIA color is first split into hue and brightness fields. The brightness field is multiplied by the luma pulse shape with one hires pixel width, while the hue drives the phase of an oscillator running at 3.58MHz, the output of which is multiplied by a chroma pulse shape with one lores pixel width. The luma and chroma pulses are added and then fed into the luma and chroma stages. In the luma stage, a notch filter isolates the luma signal. In the chroma stages, the signal is multiplied by sine and cosine waves and then low-pass filtered to extract U and V. The Y, U, and V signals are then resampled from 4×f_{sc} to 2×f_{sc}, or hires pixel rate, processed through a decoding matrix to produce R-Y, G-Y, and B-Y color differences, and added to Y to produce R, G, and B signals. These are stored in a precomputed table of impulse responses for all 256 palette colors × 2 horizontal phases, for each of red/green/blue outputs.

To produce the output line, the the incoming palette indices and H/V phases are used to select impulses from the precomputed tables, which are then added into shift registers for R, G, and B. The outputs from the shift registers are interleaved to produce the final RGB pixels.

D.4 PAL color encoding

Phase alternation

One of the main distinctions between NTSC and PAL is the use of alternating phase direction to cancel phase errors between lines, suppressing hue shift artifacts. This is done by inverting the V axis on each scanline, so that even scanlines use opposite phases for the same colors as even lines. Doing so also causes phase errors to reverse direction, which can then be canceled between adjacent lines.

There are two main ways that contemporary displays handled this cancellation. One is to use a delay line to carry the chroma from one scan line to the next, averaging the previous scan line's chroma with that of the current scan line. This results in the well-known artifact of color being carried over between alternating GTIA mode 9 and mode 11 lines, along with a general blurring and half scan line shift down of chroma. Note that only chroma is averaged and not luma. Averaging the separated chroma signal, the demodulated U and V signals, or the decoded RGB values from a UV-to-RGB linear decoder matrix are equivalent for this.

The other common method is simply to ignore the issue and decode chroma for each scan line separately, as with NTSC. This preserves full chroma resolution but shows alternating color stripes when the chroma isn't matched between even and odd scan lines.

The swinging color burst

As with the NTSC system, the PAL GTIA uses hue 1 to also generate the PAL color burst, making this color invariant to the computer color adjustment on PAL systems as well. The color burst is more complicated in the PAL system, however, which causes additional complication and sources of error in the PAL GTIA.

In NTSC, the color burst is consistently generated along the -U axis at 180° in U-V space. In order to identify scan lines that have V inverted, however, PAL uses a *swinging color burst* that changes reference angles on even and odd scan lines between 135° and 225°. This means that in addition to changing the phase direction for successive hues, the PAL GTIA must also encode an additional 90° phase shift.

The 45° difference between the NTSC and PAL subcarriers in U-V space means that hue 1 is encoded as a different color by the NTSC and PAL GTIAs. In NTSC, the 180° angle produces a yellow-green color with a standard reference decoder, while the 135° or reflected 225° angle in PAL produces an orange color. This results in one of the notorious differences in NTSC and PAL color encoding when viewed with a modern display.

Phase encoding

The NTSC GTIA generates different phases for hues by successively delaying the reference subcarrier signal through a delay line. The PAL GTIA's phase encoding is more complex due to needing to support V-axis inversion, the swinging color subcarrier, and approximating the original NTSC phase angles with the higher frequency PAL color subcarrier. This is still done with only a single reference color subcarrier signal input, so the internal delays must be changed on alternating scan lines.

An important clue to the internal functioning of the PAL GTIA is given by the behavior of the color adjustment pot. On an NTSC system, turning the pot to one extreme causes the phase delay to drop to zero, causing all hues to shift to match hue 1. On a PAL system, this produces two bands of opposite orange and blue colors, revealing the existence of an inverter in the color signal path that produces a fixed 180° shift. The inverter is active for hues 3-9 and 4-10 depending on the V polarity. Hues 3 and 10 use different inverter settings for even and odd scan lines, which results in them losing color with the color delay reduced to zero due to averaging of opposing chroma vectors.

There are two other visual anomalies from the color adjustment providing additional insight. Hues 1 and 15 are also always the same regardless of the color adjustment, which reduces the number of unique hues to 14, and there is uneven spacing between colors that results in some noticeable divisions between hues. This is a result of conflicting requirements, the need to produce seven hues out of 180° from the delay line and also provide a 90° delay between two of the delay line taps to accommodate the swinging color burst.

The solution used in the PAL GTIA is to prioritize the accuracy of the swinging color burst, using four delay taps to provide the 90° spacing, and sacrifice hue accuracy by using uneven phase steps for successive hues. This guarantees two discontinuities in the color wheel, one for each 180° half. The wide jumps are between hues 6/7 and hues 10/11. The required delay is 22.5° per delay tap, which also allows the differing even/odd encodings for hues 3 and 10 to align to the same color angle.

Imperfect color adjustments

The previous discussion assumes that the color adjustment is set to produce an ideal delay. In practice, the actual delay differs from this ideal, not only due to imperfect adjustment but also intentionally for aesthetic reasons: the ideal adjustment produces a sea-green Graphics 0 screen. Screenshots of the color palette from actual PAL systems commonly show a more conventional bluish Gr.0 screen while still showing standard orange for hue 1, indicating a narrower phase delay more in the 18-19° range. The altered adjustment adds additional discontinuity bands in the hue spectrum at the inverter switch points at hues 3/4 and hues 9/10.

The resulting colors depend upon the response of the display to the non-90° angle between the swinging color subcarriers. In one interpretation, this deviation is ignored and the even/odd color subcarriers are mapped to 135° in U-V color space. This produces consistent color for hue 1 and between even/odd lines, with the exception of hues 3 and 10. Other decoders handle this differently and show striped colors for solid hue bars.

D.5 PAL artifacting

Chroma blending

The adjacent scan line chroma blending performed by many PAL displays is conceptually done in U-V space, but other color spaces can be used. Since the blending operation is just a simple average that is a linear operation, it can be done in any other color space that is related to U-V by a linear transform. This includes I-Q, but the RGB-Y color difference space is particularly convenient:

$$\begin{aligned}
& M_{yuv2rgb}\begin{bmatrix} Y_2 \\ (U_1+U_2)\div 2 \\ (V_1+V_2)\div 2 \end{bmatrix} \\
= & \left(M_{yuv2rgb}\begin{bmatrix} Y_2 \\ U_1 \\ V_1 \end{bmatrix} + M_{yuv2rgb}\begin{bmatrix} Y_2 \\ U_2 \\ V_2 \end{bmatrix} \right) \div 2 \\
= & M_{yuv2rgb}\begin{bmatrix} Y_2 \\ 0 \\ 0 \end{bmatrix} + \left(M_{yuv2rgb}\begin{bmatrix} 0 \\ U_1 \\ V_1 \end{bmatrix} + M_{yuv2rgb}\begin{bmatrix} 0 \\ U_2 \\ V_2 \end{bmatrix} \right) \div 2 \\
= & Y_2 + \frac{(RGB_1 - Y_1)}{2} + \frac{(RGB_2 - Y_2)}{2}
\end{aligned}$$

Thus, the color difference values can be averaged with the previous scan line and the luminance for the current scan line added afterward, allowing for a fast implementation without multiplies or dot products.

For accurate results, the blend must be done with *unclamped* colors -- it should not be done on colors that have already been converted to RGB in [0, 1] range. An efficient workaround is to store extended range RGBY with the range [-0.5, +1.5] mapped to 0-255, from which the following can be computed:

$$\begin{aligned}
& Y_2 + \frac{RGB_1 - Y_1}{2} + \frac{RGB_2 - Y_2}{2} \\
= & Y_2 - \frac{Y_1 + Y_2}{2} + \frac{RGB_1 + RGB_2}{2}
\end{aligned}$$

This permits leveraging fast pixel averaging operations typically available for video motion prediction operations (PAVGB in SSE2, VRHADD in NEON).

These RGB blending operations must be done in gamma space and not linear color, for equivalence with U-V blending and also because the source colors can be impossible -- negative color values in non-linear RGB encoding have no equivalent in linear RGB.

V inversion does *not* need to be taken into account when performing this blending, as it has already been removed by the chroma demodulation stage before this blending occurs.

Note that chroma blending in this manner results in more than 256 total colors and colors with variable levels of saturation. This means that an 8-bit indexed color pixmap cannot store the blended result. Algorithms that only target APAC modes may work around this by copying GTIA hue and luminance values between mode 9 and 11 lines; this produces an oversaturated picture and does not generalize to other modes.

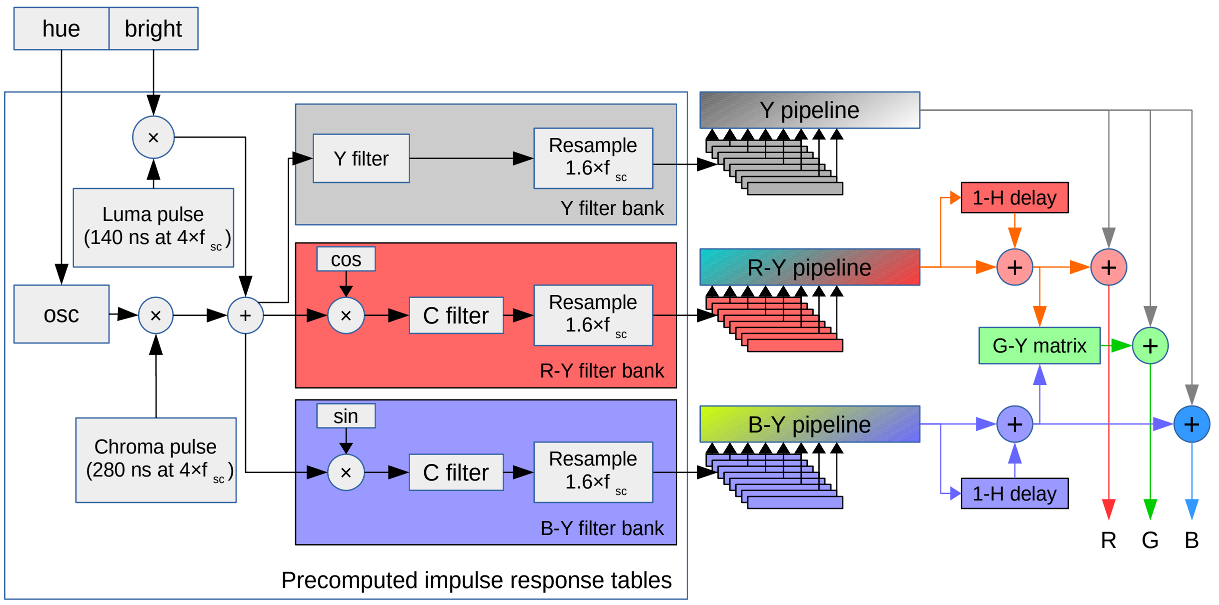


Figure 17: Altirra PAL high artifacting pipeline

The 8-bit GTIA color is first split into hue and brightness fields. The brightness field is multiplied by the luma pulse shape with one hires pixel width, while the hue drives the phase of an oscillator running at 4.47MHz, the output of which is multiplied by a chroma pulse shape with one lores pixel width. The luma and chroma pulses are added and then fed into the luma and chroma stages. In the luma stage, a notch filter isolates the luma signal. In the chroma stages, the signal is multiplied by sine and cosine waves and then low-pass filtered to extract U and V. The Y, U, and V signals are then resampled with a 4-tap cubic filter from 4×f_{sc} to 1.6×f_{sc}, or hires pixel rate. These are stored in a precomputed table of impulse responses for all 256 palette colors × 8 horizontal phases × 2 vertical phases.

To produce the output line, the the incoming palette indices and H/V phases are used to select impulses from the precomputed tables, which are then added into shift registers for Y, R-Y, and B-Y. The R-Y and B-Y differences are averaged with the previous line and then combined in a matrix to produce G-Y. Adding the color differences to Y gives the final RGB output.

Hires artifacting

Hires pixel patterns produce false colors with PAL as with NTSC, though the effect is harder to use due to the PAL chroma subcarrier frequency. Unlike NTSC, where alternating pixels are sufficient, the PAL subcarrier advances by 5/8^{ths} of a cycle per hires pixel and thus requires an 8-pixel pattern horizontally to produce a consistent color.

Simulating PAL hires artifacting is conceptually similar to NTSC, but with a bit more complexity. With NTSC, only two filter banks are required, for even and odd pixels. The PAL chroma subcarrier advances by 5/8^{ths} of a cycle per hires pixel instead of half a cycle, which requires filter banks for eight horizontal phases instead of two. This may again be doubled to 16 banks for V inversion, depending on the implementation.

Appendix E
Analog Audio Model

E.1 Introduction

The audio signal produced by an Atari computer results from a combination of digital and analog circuitry. The digital logic within POKEY is relatively well documented and understood, but the analog path less so. Some complex sound effects can only be realized through simulation of the non-linear analog path. In this appendix, a model is described for simulating the analog path as based on measurements taken from the actual hardware. Figure 18 shows a block overview of the analog model.

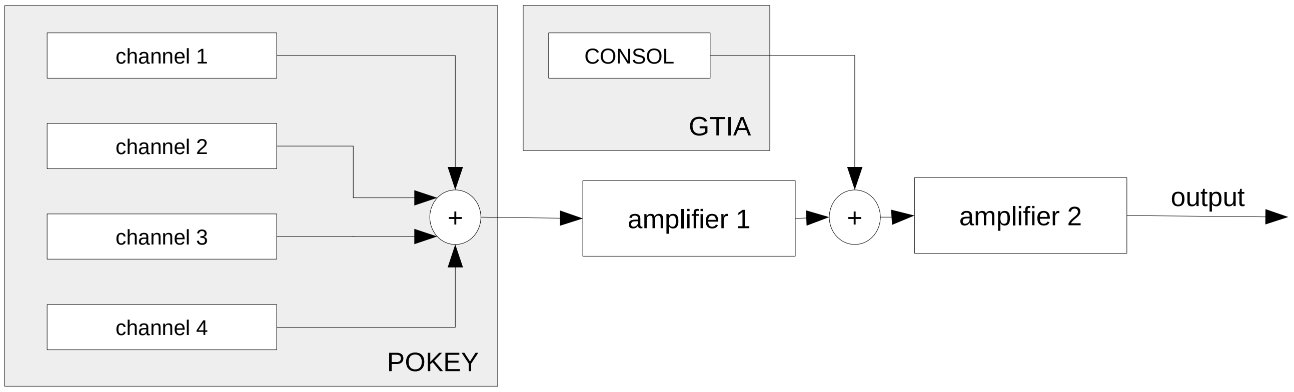


Figure 18: Analog audio block model

E.2 POKEY output

The raw audio signal from POKEY is a sum of the outputs of all four audio channels. The audio output logic for each channel combines the clock signal from the channel timer, noise generators, toggle flip/flop, and high-pass XOR to produce an on/off output signal for the channel. This is then multiplied by the volume to produce an analog output, after which all four outputs from each channel are summed.

Because the channel output is an on/off signal, the output from each channel is unbalanced. For a square wave (AUDCn = $A0-AF), the average volume level is half the channel volume. This also means that POKEY’s mixed output is unbalanced. When all channels are off – either an output or volume of 0 – the output is +5V, and increasing output levels lowers the output voltage. At this stage, the signal holds steady as long as the output volume is held in the audio circuitry.

Channel DAC

Each individual channel has a 4-bit DAC to modulate the channel's digital output by the volume level. The DAC is structured as four gates contributing voltage in power of two ratios, one for each volume bit. In practice, the volume bit outputs are not quite matched. Measurements of the steady-state levels from a single channel show voltage drops of approximately 0.12V, 0.26V, 0.56V, and 1.12V, within about 0.01V. The gap between volume bit 1 and volume bit 2 is the most pronounced and visible in volume ramp signals.

Non-linear output

The output from the four channel DACs is added, but the result is subject to a non-linear saturation effect as the total volume level rises. The output is roughly linear up to a total volume level of 12 before compression starts to become visible in the signal, and distortion becomes audible at around total levels of 30 or higher.

The saturation curve can be modeled roughly as a piecewise linear section followed by an exponential curve, based on the summed output from the channel DACs. A hand-fitted approximation to the curve is as follows, with both the raw summed input and non-linear output normalized to 0-1:

$$y = \begin{cases} 2.171\,x & x \le 0.14 \\ 2.171\left(0.14 + \dfrac{1.0 - e^{-2.85*(x-0.14)}}{2.85}\right) & x \ge 0.14 \end{cases}$$

Note that the effect applies only to the sum of all channels, not to the output of individual channels. The effect when multiple channels are playing is therefore dynamic based on the instantaneous sum of all channels and cannot be applied to individual channel volumes. For the voltage ratios given earlier, the voltage drops should be divided by (0.12 + 0.26 + 0.56 + 1.12) × 4 = 8.24 to obtain the normalized outputs.

An implication of this is that channels that are inaudible by themselves due to either producing a constant level or an ultrasonic sound can still cause distortion of other channels. In particular, a channel that has a non-zero constant output due to either volume-only mode or having a stalled timer will bias the output farther into the non-linear region of the curve. As the saturation effect takes place before to AC coupling in the amplifier stages, the DC bias causing this is not cancelled out over time.

E.3 First amplifier stage

The first amplifier stage after POKEY’s output has the greatest effect on the shape of the audio signal due to its input being AC coupled through a capacitor. This has the effect of applying a high-pass filter with an approximate time constant of 2.6 ms. A high-pass filter blocks DC signals entirely (constant levels) and attenuates (reduces) low frequency components more than high frequency components. In a signal waveform, it converts square pulses to exponential decay curves. The first amplifier stage has the faster decay curve of the two stages and is more responsible for limiting the bass output from POKEY.

The exponential decay curve is the result of the capacitor charging or discharging based on the difference in voltage across the capacitor, which increases when POKEY’s output changes and decreases as the capacitor gradually charges or discharges. As the voltage difference decreases, the rate of change in voltage decreases, which causes the curve to gradually level off according to the formula:

$$N(t) = N_0 e^{-t/\tau}$$

Tau (τ) is the time constant, which determines the rate of decay of the curve. It is the amount of time for the curve to reach 1/e = 36.8% of its original value, or 63.2% toward its target. After 2τ periods, the curve is at 1/e^2 = 13.5% or 86.4% to the target. Given a constant non-zero input, the output takes *infinite* time to converge, but it quickly reaches a point where the difference is negligible. After 12τ, the remaining difference is 6×10^{-6}, which is below half a step for 16-bit sampled audio. This means that any constant level output (DC bias) from POKEY is quickly eliminated.

This type of high-pass filter is *linear*. One consequence of this is that the filter is invariant to both time shifts and changes in amplitude – shifting or amplifying the input results in the same time shift or gain on the output. It also means that the high-pass of the sum of two signals is the same as the sum of the two signals individually high-pass filtered.

A quick transformation converts the time constant to other useful periods. Multiplying by ln(2) gives the half life, the time for the output or output difference to halve. Multiplying by a sampling rate gives the decay factor at that rate, which can then be used for discrete simulation:

$$y_{n+1} = y_n + (x_n - y_n) \times \left(1 - e^{-1/(\tau \cdot f_s)}\right)$$

The decay term can be precomputed, leaving two additions and one multiplication per sample.

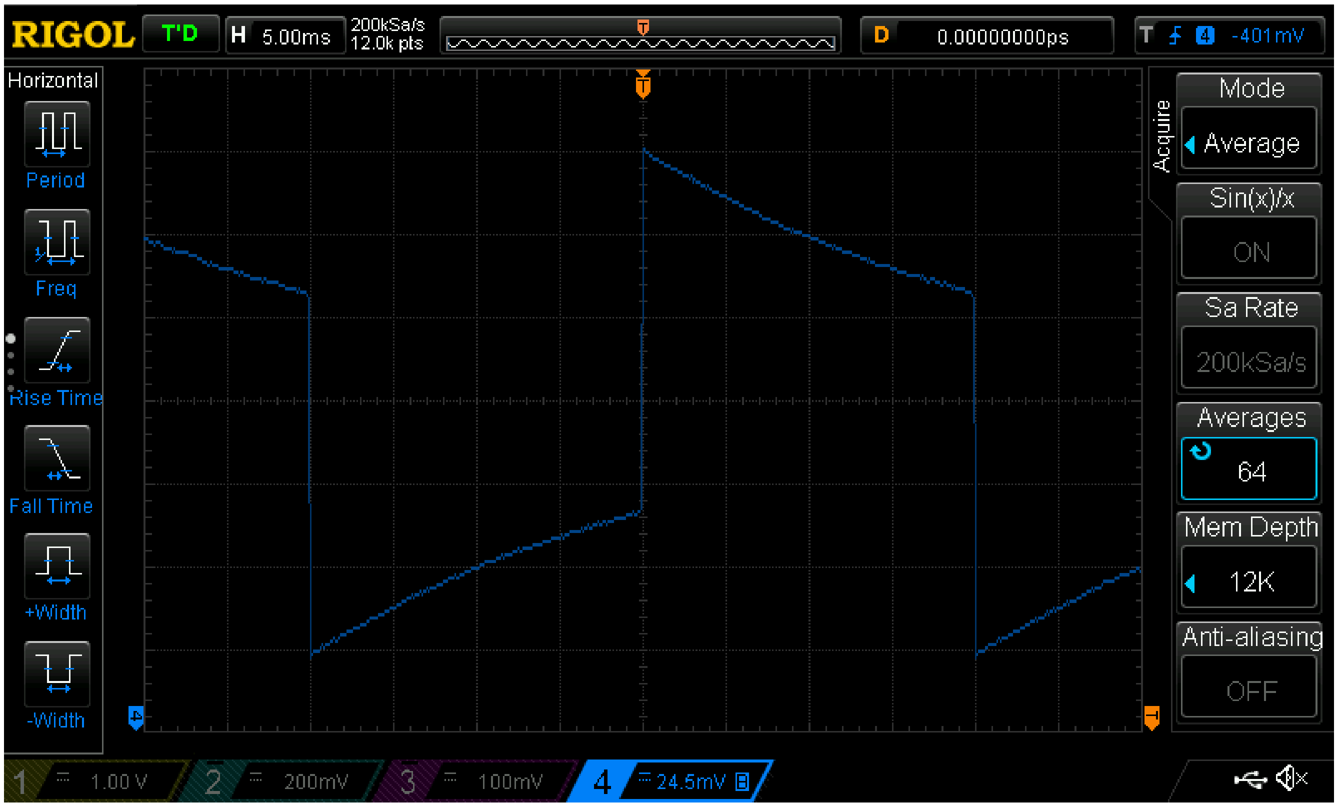
For instance, given a 48KHz sampling rate, the 2.6 ms delay of the first stage can be simulated with a decay factor of e^(-1/(0.0026 × 48000)) = 0.992. This means that after each sample, the difference between the output and input is reduced by 0.8% and 99.2% of the difference remains.

E.4 External signal sum point

Between the first and second amplifier stages, external audio sources such as GTIA's CONSOL bit 3 on XL/XE machines, SIO audio, and PBI audio are included. This means that those sources are not affected by the input coupling of the first amplifier stage. In particular, GTIA audio shows much less decay over time than POKEY audio.

E.5 Second amplifier stage

The second amplifier stage is downstream of all audio inputs and handles the full audio signal. All of its inputs are also AC coupled and show similar decay effects, which removes any DC bias over time and ensures that the output eventually drops to zero with quiet input. The decay is much slower than the first stage with a time constant of approximately 24.7 ms. This effect can be most clearly seen when playing a square wave through GTIA's CONSOL bit 3 on an XL/XE machine, which only shows the decay effect from the second stage amplifier.



Mode: Average
Sin(x)/x: ON
Sa Rate: 200kSa/s
Averages: 64
Mem Depth: 12K
Anti-aliasing: OFF

Figure 19: Decay from second stage amplifier

A square wave played on CONSOL bit 3 on an NTSC 800XL, sampled from the monitor jack audio output. The square wave period is 40 ms per cycle (20 ms per pulse).

The combination of the different decay curves from the two amplifier stages can result a mild overshoot effect, where the faster decay of the first stage causes the output signal to cross the zero level slightly before the slower decay of the second stage cancels the overshoot.

A clamping effect can also occur in the second amplifier when the input signal has high amplitude, clipping peaks of the input signal. This effect is shown in Figure 20.

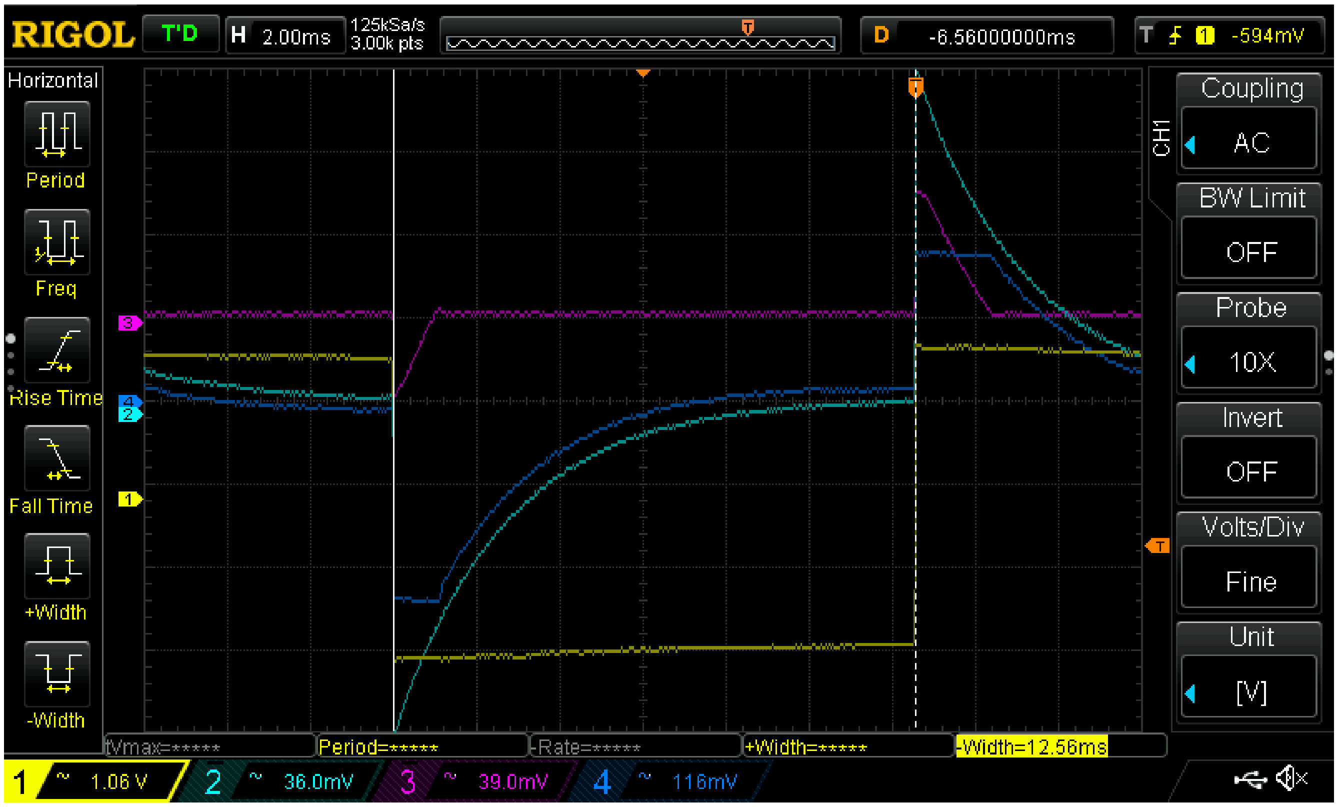


Figure 20: Dynamic clamping effect

Signal capture from an NTSC 800XL playing a square wave on all four channels at maximum volume. CH1 (yellow) is POKEY output, CH2 (cyan) is at the output of the first stage amplifier at R2, CH3 (purple) is at the input of the second amplifier at R7, and CH4 (blue) is the final signal from the monitor audio output. CH2 (cyan) shows the change in waveform shape from the first amplifier stage; CH4 (blue) shows the saturation effect in the second stage as the amplifier gradually eliminates the error shown in CH3 (purple).

The clamping effect is asymmetrical, with the amplifier able to output a more negative signal than positive (more headroom toward high POKEY total volume / 0V). For a normalized max POKEY volume of 1, the approximate clamping range is [-0.50, +0.65], with more excursion toward higher volume levels. However, it is important to note that this clamp is based on the output of the first amplifier, so it clips dynamic peaks rather than simply high volume levels. The limit at +0.65 is easily hit by a high total volume level alone, but the limit at -0.50 is only seen dynamically when the total output swings back to from high to low levels.

For example, a single channel playing audio at volume levels 0-15 will generally not show clipping. If audio is momentarily played at a high volume level of 40-60, though, it will shift the reference voltage such that upon returning to the 0-15 range, the second amplifier will see a spike in the opposite direction, then momentarily clamping some of the low volume levels and causing some distortion until the reference swings back. This is of most consequence when playing audio on multiple channels since POKEY's volume control is unbalanced and higher volume levels introduce DC bias into POKEY's output.

E.6 Final output

The output from the second amplifier stage is sent directly out through the audio output of the monitor jack. There is no low-pass aliasing filter on this output, such as the 7KHz filter in the Amiga. Frequencies in the 15KHz range easily appear in the output at full amplitude, and frequencies as high as 60-120KHz will still appear, though attenuated. Suppression of these high frequencies is mainly due to either the inability of the monitor to reproduce them or the user to hear them. A user that has retained high-frequency hearing and has high quality equipment will therefore hear more of these high frequency components and may therefore perceive the sound output differently.

Appendix F
Quick Reference

F.1 CPU opcode table

Table 94 lists all official opcodes supported by the 6502, 65C02, and 65C816 CPUs. Opcodes highlighted in green require a 65C02, while ones in blue require a 65C802/65C816.

The flags column shows flags affected or used by the instruction. 0/1 indicates that the flag is set or cleared; X indicates that the flag is modified. S indicates that the flag is not modified, but the instruction changes behavior depending on its value. For the M/X flags, this indicates a change in operand and operation size.

Name	-	#i	dp				abs			long		(dp)				[dp]		[a]	(abs)		rel	r16	*	Flags							
			dp	,X	,Y	,S		,X	,Y		,X		,X	,Y	,SY		,Y			,X				N	V	M	X	D	I	Z	C
ADC		69	65	75		63	6D	7D	79	6F	7F	72	61	71	73	67	77							X	X	S		S		X	X
AND		29	25	35		23	2D	3D	39	2F	3F	32	21	31	33	27	37							X		S				X	
ASL	0A		06	16			0E	1E																X		S				X	X
BCC																					90										S
BCS																					B0										S
BEQ																					F0									S	
BIT[87]		89	24	34			2C	3C																X	X					X	
BMI																					30			S							
BNE																					D0									S	
BPL																					10			S							
BRA																					80	82									
BRK	00																														
BVC																					50				S						
BVS																					70				S						
CLC	18																														0
CLD	D8																											0			
CLI	58																												0		
CLV	B8																								0						
CMP		C9	C5	D5		C3	CD	DD	D9	CF	DF	D2	C1	D1	D3	C7	D7							X	X	S				X	X
COP																							02								
CPX		E0	E4				EC																	X	X		S			X	X
CPY		C0	C4				CC																	X	X		S			X	X
DEC	3A		C6	D6			CE	DE																X		S				X	
DEX	CA																							X			S			X	
DEY	88																							X			S			X	
EOR		49	45	55		43	4D	5D	59	4F	5F	52	41	51	53	47	57							X		S				X	
INC	1A		E6	F6			EE	FE																X		S				X	
INX	E8																							X			S			X	
INY	C8																							X			S			X	
JMP							4C			5C								DC	6C	7C											
JSR							20			22										FC											
LDA		A9	A5	B5		A3	AD	BD	B9	AF	BF	B2	A1	B1	B3	A7	B7							X		S				X	
LDX		A2	A6		B6		AE		BE															X			S			X	
LDY		A0	A4	B4			AC	BC																X			S			X	
LSR	4A		46	56			4E	5E																0		S				X	X
MVN																							54				S				
MVP																							44				S				
NOP	EA																														
ORA		09	05	15		03	0D	1D	19	0F	1F	12	01	11	13	07	17							X		S				X	
PEA							F4																								
PEI												D4																			
PER																						62									
PHA	48																									S					
PHB	8B																														
PHD	0B																														
PHK	4B																														
PHP	08																														
PHX	DA																										S				

[87] The BIT #imm instruction of the 65C02 differs from the other BIT opcodes: it does not set the N and V flags.

Name	-	#i	dp				abs			long		(dp)				[dp]		[a]	(abs)		rel	r16	*	Flags							
			dp	,X	,Y	,S		,X	,Y		,X		,X	,Y	,SY		,Y			,X				N	V	M	X	D	I	Z	C
PHY	5A																										S				
PLA	68																							X		S				X	
PLB	AB																														
PLD	2B																														
PLP	28																							X	X	X	X	X	X	X	X
PLX	FA																							X			S			X	
PLY	7A																							X			S			X	
REP		C2																						X	X	X	X	X	X	X	X
ROL	2A		26	36			2E	3E																X		S				X	X
ROR	4A		46	56			4E	5E																X		S				X	X
RTI	40																							X	X	X	X	X	X	X	X
RTL	6B																														
RTS	60																														
SBC		E9	E5	F5		E3	ED	FD	F9	EF	FF	F2	E1	F1	F3	E7	F7							X	X	S		S		X	X
SEC	38																														1
SED	F8																											1			
SEI	78																												1		
SEP	E2																							X	X	X	X	X	X	X	X
STA			85	95		83	8D	9D	99	8F	9F	92	81	91	93	87	97									S					
STP	DB																														
STX			86		96		8E																				S				
STY			84	94			8C																				S				
STZ			64	74			9C	9E																		S					
TAX	AA																							X			S			X	
TAY	A8																							X			S			X	
TCD	5B																							X						X	
TCS	1B																														
TDC	7B																							X						X	
TRB			14	1C																						S				X	
TSB			04	0C																						S				X	
TSC	3B																														
TSX	BA																										S				
TXA	8A																							X		S				X	
TXS	9A																														
TYA	98																							X		S				X	
TYX	BB																							X			S			X	
WAI	CB																														
WDM	42																														
XBA	EB																							X						X	
XCE	FB																									X	X				X

Table 94: CPU opcode table