Handy Specification

Original June-87	Dave Needle
20-Jan-89	Rev M
6-Mar-89	Rev M-2
27-Mar-89	Rev M-3
13-Apr-89	Rev N
22-May-89	Rev N-1
30-May-89	Rev N-2
16-June-89	Rev N-3
9-Aug-89	Rev P (DevCon)

Private, confidential, secret, and don't look at it under P. of V.P.

Table Of Contents

Appendices

1 General Overview

Handy is a portable hand held programmable video game. It is totally self contained with its own power source, display screen, fluorescent backlight speaker, and data input reader.

1.1 Marketing Feature Set

The unit is portable, allowing its use in many areas previously untouched by arcade games. The unit is programmable thru the use of small data cartridges. The unit can be privately used with earphones. The marketing advantages are obvious. They can be detailed in a marketing document. Certain advantages, however, are worth mentioning here.

In addition to using the game in the usual 'play' areas, the player can easily carry the game and several of the compact data cartridges to nearly any 'play' location. For example, long car rides, airplane flights, sales meetings, and visits to relatives. With the use of an earphone, the game can be played without audible disturbance of the other car passengers, relatives, or classmates.

By adding the expansion module (an infra-red communications pod), many players may play simultaneously and interactively in a common game (car race, space battle, snow ball fight, etc.) Additionally, a matching receiver unit could plug into the joystick connector of a C64 or other game machine and Handy could be used as a wireless joystick.

Intergame communications can also be done with a wire. Many players can be connected in parallel on the same wire.

1.2 External Specification

1. The display is a 3.4 inch diagonal color LCD with a resolution of 160 horizontal triads and 102 vertical lines. The LCD can display 16 levels (each) of red, green, and blue, for a total of 4096 colors.
2. There is a fixed intensity backlight for the display.
3. There is a contrast control (viewing angle) for the LCD.
4. The display rate is programmable and includes 50 and 60 Hz.
5. The display can be used in 3 orientations, allowing for both left handed and right handed landscape as well as portrait operation.
6. The audio system is mono with 4 voices and a frequency response from 100Hz to 4KHz.
7. There is a 2 inch speaker, a volume control and an earphone jack for private listening.
8. The required batteries are 6 'AA' cells.
9. There is a power jack for use with an external power supply.
10. The game input controls are a thumb controlled joy stick, two sets of two independent fire buttons, pause, and two flablode buttons.
11. The data input is a ROM cartridge with 1 Mbyte of address space and can contain writable elements.
12. The expansion connector interfaces to a bi-directional serial communications port. The only expansion devices considered at this time are an infra-red communications pod (or wire) to allow for multiple player games, a steering wheel adaptor, and a wireless joystick adaptor.

2 Hardware Overview
(see the Handy Block Diagram, appx 1)

The system hardware consists of:

'Mikey' and 'Suzy', custom digital ICs.
A 16 MHz crystal.
Two 64k by 4 DRAMs.
A 2 inch speaker with an earphone jack and volume control.
An LCD and its related drivers, backlight, and contrast control.
A data input system. (Tape or ROM)
Batteries, power supply, and external power jack.
An expansion port.
A joystick, 2 fire buttons, and other switches.

The division of circuitry between the 2 digital ICs is that Mikey contains all of the non-sprite hardware and Suzy is only a sprite generation engine. Some non-sprite functions (the switch readers and the ROM reader) are in Suzy due to pin limitations. In addition the math functions are part of Suzys sprite engine.

The crystal is the only source of timing information in the system. The basic timing 'tick' of the system is 62.5 ns. Let us now define the term 'tick' to be 62.5 ns.

The system RAM is 64K bytes. This RAM houses the video buffer(s) and collision buffer (total maximum of 24K bytes) in addition to the game software (worst case minimum of 40K bytes). The RAMs have a 120ns RAS access time and 60ns page mode CAS access time. This allows us to have a 125ns (8MHz) page mode memory access rate and a 250ns (4MHz) normal memory access rate.

The speaker is a 2 inch diameter 8 ohm speaker. The volume control range includes zero. The earphone jack is the standard stereo 'Walkman' style (only mono sound, however).

The LCD has a resolution of 480 horizontal pixels by 102 vertical pixels. Three pixels, one of each color, form a square triad with a resultant screen resolution of 160 triads by 102 lines. The column drivers can generate 16 levels of intensity for each pixel, resulting in a palette of 4096 colors. The LCD circuitry includes the power generation for the LCD driver ICs, the decoding of the Mikey strobes for the LCD driver ICs, and the power generation for the backlight.

The data input system is either a ROM cartridge or a magnetic tape reader. The system hardware will support both, but units will be made with either one or the other. The data input systems are explained elsewhere.

The power system provides raw power to the regulator, and if used, the motor. The regulator has a soft on/off function so that the system can power itself off when not in use. This is required so as to avoid the customer frustration of expensive battery replacements which could then cause loss of software revenue. The motor is separately powered so that its load is not part of the regulators problem. The soft off function also disconnects it from the power source.

The expansion port has a bi-directional serial port operating asynchronously at a programmable speed with a maximum of 62500 baud. This is approximately 104 bytes per 60 Hz frame. By allowing all of the games in a multiple player game to operate one frame behind their control functions, up to 104 bytes of data can be communicated. Using an example maximum of 8 players in a group and for some overhead, 12 bytes per player are available.

The human input controls consist of a 4 switch (8 position) joy stick, two sets of 2 independent fire buttons, game pause button, 2 flablode buttons, power on, and power off. The two sets of fire buttons are wired together, there are only 2 'fire' signals. Two sets are available to allow for left and right handed operation. Flablode is a Jovian word meaning a device or function that we know is required or desired but for which we don't have an actual definition (noun: flabloden, verb: to flablode).

3 Software Related Hardware Perniciousness

(or why some software people hate some hardware people)

There are certain things that the software ought not to do to the hardware. While these things are not physically destructive on their own, they will cause the hardware to act unpredictably strange, which may cause the user to be physically destructive.

While we certainly could have protected the system from many of the following problems, I doubt that we would have found them all. In addition, the act of software doing one of these things means that there is a problem in the software anyway and the intended function would probably not happen. Additionally, this is only a toy. If this unit were a bio-medical device I would have found a safer solution.

The things that software MUST NOT do and the precautions software MUST take are neither intuitive nor obvious, so I will detail them here.

3.1 Don't Do These Things.

If you do any of the things that I tell you not to do, the hardware will act unpredictably and will not be expected to recover without a complete system initialization. So don't do them.

3.1.1 Suzy SCB Register Accesses

All of the registers in the Suzy SCB are unsafe to touch while the sprite engine is in operation. This is true for BOTH read and write accesses. Prior to accessing any of those registers (they are identified in the hardware address appendix as 'unsafe') you must first check to see if the sprite engine is running (either for sprites or math). Those status bits are in SPRSYS.

3.1.2 Sequential Memory Accesses

CPU instructions that result in a Suzy access followed immediately by another Suzy access will probably break Suzy. Please do not do them.

3.1.3 Compiler Stuff

Some compilers do stupid things when they convert complex statements. For example, "A = B = C = 0" may get converted to:

store 0 to C,
read C and write it to B,
read B and write it to A.

This will break if you try it on any of my hardware, so unless you are real sure that your compiler will never do it, don't write your code that way.

3.1.4 Writes to the Game Cartridge

Writes to Suzy are blind, but will complete before any other CPU function can disturb them EXCEPT in the case of writes to the game cart. For 12 ticks after a write to the game cart is started, the CPU MUST NOT access Suzy (read or write). If it does, the write to the cart will be trashed.

3.1.5 Palettes and Page Breaks

This one is an actual hardware bug of mine. The hardware signal that ends the loading process for any SCB element comes 2 bytes prior to the actual end of the element. If a page break comes at the same time as the 'end' signal, then that end signal needs to be delayed by one cycle. If not, the loading process will end and not start up again at the end of the page break. Well, I screwed up the logic for the pen index palette loader. Therefore if a pen index palette starts at address xxFA, a page break will occur at the same time as the 'end' signal and this page break will be ignored by the palette loader. The last 2 bytes of this 8 byte palette will not be loaded from RAM, and the values loaded by the previous SCB will still be in the hardware. Pen index numbers C,D,E, and F will be incorrectly unchanged. Sometimes I am such a jerk.

3.2 Please Don't Do These Things.

There are also things that software people do that merely annoy the hardware people rather than annoy the end user. This includes seemingly harmless activities like sub-decoding bit values from byte descriptions (sprite types, for instance), or assuming address continuity between areas of hardware (like SPRCTL0 follows the Sprite Control Block).
Please don't do them.

It would be difficult to list all of the possibilities, so I will list the general considerations and ask you to be sensible about them. In addition, please feel free to submit a request for an exemption on a specific case. If it is granted, we will change the specification so as to make the special case forever legal. The price will be small.

The list:
Do not decode any bits within any byte definitions.
Do not assume any address continuity between hardware sections.
Do not assume that any hardware addresses that are multiply mapped will remain so.
Do not use un-defined bits within a byte for anything.

If you notice an unspecified but clever interaction between two or more sections of hardware, please come and tell me about it. It is more likely that it is a mistake rather than an omission from the spec.

I will try to list the approved exemptions in the spec.

Thank You

3.3 Do It This Way

Some of the hardware functions, as designed by us mindless brutes, require special handling. As we discover these requirements, I will list them here.

3.3.1 Timer Reload Disable

If a particular timer reload bit is set to disable, the timer ought to count down to zero, stop counting, and set its 'timer done' bit. However, if the 'timer done' bit was already set, the timer will not even do its first count. Therefore, when the setting of 'timer reload' is 'disabled', you must clear the 'timer done' bit in order for the timer to count at all.

This 'clear' can be accomplished by setting the 'reset timer done' bit, but that can cause a new and exciting problem. The 'clear' can also be accomplished by writing directly to the byte that contains the 'timer done' bit.

The answer of choice is to write a 0 to bit 3 of the control byte (TIMnCTLB). Since the other bits are adjusted by hardware, I recommend writing a full byte of 0 to that register when you need to do the 'clear' function.

3.3.2 Reset Timer Done

The 'reset timer done' bit in the hardware registers was mistakenly designed as a 'level' signal rather than a 'pulse' signal. Therefore, to use it correctly, you must 'pulse' it in software. This means first setting it high, and then setting it low. The problem with just leaving it high is that the dumb hardware may (depending on other conditions in that particular timer) send a continuous interrupt to the interrupt logic. In addition, since a lucky interrupt might actually slip in during this 'software pulse', the recommended process is to clear the interrupt enable bit at the same time that the reset timer done bit is set. Happily, both bits are in the same register. Please remember to restore the 'enable' bit as you release the 'reset' bit.

4 CPU/ROM

The CPU is a 65C02 cell imbedded in the Mikey IC. The specifics of its instruction set and register operations are found in the VTI specification.

4.1 CPU Cycle Timing

While the number of CPU cycles for each instruction are defined in the VTI spec, the number of system 'ticks' used by each CPU cycle are defined here. Some cycles will require a fixed number of ticks, some will require a variable number of ticks based on the instruction that preceded it, and some will require a variable (and sometimes non-deterministic) number of ticks based on the functioning of other pieces of hardware in the system. In addition, the CPU can be paused by the video and refresh accesses, and it can be interrupted by the timers and the serial port. The point of listing all of these variances to the CPU timing numbers is to advise the programmer that CPU cycles can not be used as the timing elements in small software timers. Under certain specific circumstances, the environment will be sufficiently controlled to allow for small software timers (initial mag tape reading), but in general, that practice should be avoided.

4.1.1 RAM Page Mode

The RAMs used in the system have a page mode operation in which one of the control signals (RAS) is not repeated for each memory access. This allows the cycle to be shorter than a normal access cycle. The requirement for using a page mode cycle is that the current access is in the same 256 address page of memory as the previous access. While comparing current and previous addresses is certainly a valid method of determining if a page mode cycle can be used, it usually takes the same or more time than just repeating the control signal. As a result, page mode is often not used by many CPU designers.

We use the method of decoding the current op-code to see if the next cycle could be a page mode cycle and then observing the other pertinent states of the system to see if some reason exists to NOT allow a page mode cycle. While this method is not 100% efficient in allowing all possible page mode opportunities, its silicon requirements are small and when properly designed it does not permit false page mode cycles.

The CPU makes use of the page mode circuitry in its op-code reads. All writes and all data reads are done in normal memory cycles. A page mode op-code read takes 4 ticks, a normal read or write to RAM takes 5 ticks.

4.1.2 Hardware Accesses, DTACK

CPU accesses to hardware require an acknowledge signal (DTACK) from that hardware in order for the CPU cycle to proceed. These CPU accesses fall into 2 classes. One is for hardware that is always available and the other is for hardware that becomes available at a time not related to the CPU.

For always available hardware , the DTACK can be generated from the address decode and not cause the cycle to be longer than a RAM read or write cycle (5 ticks). Writes to Suzy are handled as 'always available' whether or not Suzy is actually available.

For eventually available hardware, (some of Suzy and some of Mikey), the DTACK is generated as a combination of the address decode, sometimes the column strobe, and the particular hardware being accessed. This cycle has a minimum requirement of 5 ticks and a maximum requirement of 128 ticks (probable actual maximum of 40). The maximum is not related to the CPU, it is required to prevent video and refresh underflows. All writes to Suzy are 'Blind' in that the write cycle is always 5 ticks long and does not get a DTACK. Suzy will accept the data immediately and then place it internally as required. The maximum time required for this internal placement is 6 ticks (except for writes to the game cart). This is less than the fastest turnaround time for 2 sequential writes to Suzy, so no collisions will occur. Poof for unsafe addresses.

Some of the hardware in Mikey is constructed of Dual Ported RAMs addressed in a cyclical manner. These DPRAMs will have a maximum latency equal to their cycle time (1.25 us). Some of the hardware in Suzy is also constructed of Dual Ported RAMs. These RAMs are not cycling, but their latency is still slightly variable (+1 tick, -0) due to clock synchronization.

The CPU accessible addresses in both Mikey and Suzy are not all readable and writeable. See the hardware address appendix (appx 2) for the specifics.

4.1.3 CPU Cycle Tick Summary

Cycle	Min	Max
Page Mode RAM(read)	4	4
Normal RAM(r/w)	5	5
Page Mode ROM	4	4
Normal ROM	5	5
Available Hardware(r/w)	5	5
Mikey audio DPRAM(r/w)	5	20
Mikey color palette DPRAM(r/w)	5	5
Suzy Hardware(write)	5	5
Suzy Hardware(read)	9	15

4.1.4 CPU NMI Latency

The NMI signal path from the pin of Mikey to the pin of the internal CPU contains clocked delays. This will make the usability of the pin questionable in the debug environment.

Fortunately for us, Howard and Craig arranged the diagnostic hardware to still use it effectively. See the Howard board spec for details.

4.1.5 Suzy Bus Request-Bus Grant Latency

The maximum allowable latency at Suzy is constrained by the needs of the video DMA circuit. As a compromise between bigger FIFOs in the Mikey video DMA and reduced performance in Suzy, we are setting the maximum latency from Bus Request to Bus Grant at 2.5 us.

The time between Mikey requesting the bus and Suzy releasing it is dependant on the state of the currently running process inside of Suzy. The longest process is 30 ticks. Adding the overhead of accepting the bus request and releasing the bus grant brings the total to 40 ticks.

4.2 ROM

The system ROM is imbedded in Mikey. Its size is 512 bytes. Its main and perhaps only function is to read in the initial data from the data input system and then execute that data. In the case of the magnetic tape system, some error correction will be performed.

4.3 CPU Sleep

BIG NOTE: Sleep is broken in Mikey. The CPU will NOT remain asleep unless Suzy is using the bus. There is no point in putting the CPU to sleep unless you expect Suzy to take the bus. We will figure out how to save power some other way.

I will still keep the following paragraphs pertaining to sleep in this spec.

The CPU can put itself to sleep by writing to the CPU disable address. The CPU wants to sleep for two reasons. The first is that it is done with all of its work and we would like to conserve battery power. In this case, the CPU would pre-arrange for an interrupt to wake it up at the next time that work needs to be done (probably vertical restart). When awoken by an interrupt, normal interrupt vectoring takes place. Note that if no interrupt occurs, the CPU sleeps forever. An interrupt will wake you up even if they are disabled inside the CPU. The second reason to sleep is that Suzy has been requested (by the CPU) to do work. In that case, it is up to Suzy to awaken the CPU when it is done. When awoken by Suzy, processing continues from where it left off.

The CPU can not fall asleep unless it puts on its pajamas. This is accomplished by disabling all interrupts (register in Mikey) that you do not want to wake you up, then clearing them (in case they came in while you were disabling them), and then writing to the Suzy Wakeup Acknowledge address IF you were woken up by Suzy (or if this is the first time you are going to sleep). The purpose of the Suzy Wakeup Acknowledge address is to prevent missing the wakeup request from Suzy should it occur exactly at the same time as an interrupt.

5 Display

5.1 Frame Rate

We have a programmable frame rate to accommodate either 50 or 60 Hz lighting systems. We may also want to vary the frame rate for 'game' reasons.

While no vertical blanking is actually required by the LCD, the system likes to have some non-display time in which to cleanly change the display characteristics. In addition, multiple player games need some dead time between frames to re-synchronize to the master. Since reloading the color palette could take 150 us, at least 1 and perhaps 2 scan lines of time should be allocated as vertical blank time. The current method of driving the LCD requires 3 scan lines of vertical blank.

During vertical blank time, none of the display lines are allowed to be on. If they were, that line would be noticeably 'brighter' than the others.

Additionally, the magic 'P' counter has to be set to match the LCD scan rate. The formula is: INT((((line time - .5us) / 15) * 4) -1)

Don't forget to set the display control bits. Normal 4 bit color is:
'DISPCTL' (FD92) = x'0D'

Some frame rate choices are:

60Hz:

159 us x 105 lines =16.695 ms [59.90 Hz], 3 lines of Vertical Blank

For normal 60Hz video operation, set the relevant timers as follows:

Timer 0: clock=1us, backup=158.
FD00=x'9E', FD01=x'18'
Timer 2: clock=linking, backup=104.
FD08=x'68', FD09=x'1F'
Pcount: 'PBKUP' = 41.
FD93= x'29'

50Hz:

190 us x 105 lines =19.950 ms [50.13 Hz], 3 lines of Vertical Blank

For 50Hz video operation, set the relevant timers as follows:

Timer 0: clock=1us, backup=189.
FD00=x'BD', FD01=x'18'
Timer 2: clock=linking, backup=104.
FD08=x'68', FD09=x'1F'
Pcount: 'PBKUP' = 49.
FD93= x'31'

75Hz:

127 us x 105 lines =13.335 ms [74.99 Hz], 3 lines of Vertical Blank

For 75Hz video operation, set the relevant timers as follows:

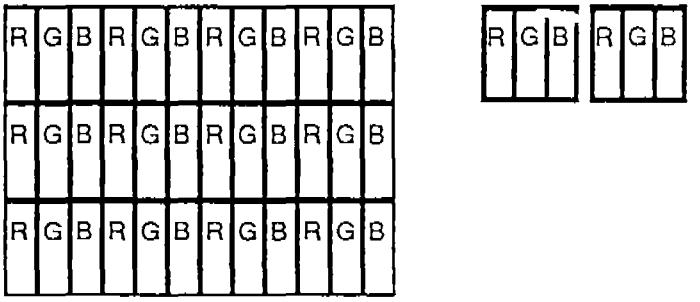
Timer 0: clock=1us, backup=126.
FD00=x'7E', FD01=x'18'
Timer 2: clock=linking, backup=104.
FD08=x'68', FD09=x'1F'
Pcount: 'PBKUP' = 32.
FD93= x'20'

The maximum frame rate occurs when Hcount is 121 (backup = 120) which results in a vertical frequency of 78.7 Hz. 75 Hz (a useful rate) is achieved by setting the H backup value to 126 (x'7E'). The vertical values do not change.

We notice that 50Hz operation causes a massive flicker that is probably due to the speed of the LCD itself. 50Hz may not be usable.

5.2 Pen Numbers/Color Palette

The display system has 4 bits of pen number per pixel in each display buffer. The color value of a particular pixel is converted from its pen number by the color palette in Mikey. There is only one color palette in the system. The color palette can be written to at any time (it is 'always available hardware'). The color display has rectangular shaped pixels (resulting in square triads) arranged in the following grid.



5.2.1 Flipped Color Palette

When the screen is flipped, the data is read from RAM starting at the bottom of the screen. This means that any screen related color palette changing must take 'flip' into account and reverse its order accordingly.

5.2.2 Monochrome Display

A long time ago, in a galaxy far away, monochrome display was considered. The chips still have some monochrome logic in them, and certain bits still need to be set in order to insure color operation. I have removed all non-pertinent monochrome information from the specifications. Please let me know if I missed any.

5.3 Display World/Suzy Painting Buffer/Mikey Display Buffer

The display world is an imaginary 64K pixel by 64K line space. Sprites may be positioned anywhere within the world by use of their 16 bit horizontal and 16 bit vertical positions. The origin of this world (0H, 0V) is the upper left corner.

The 160 pixel by 102 line Suzy painting buffer can be positioned anywhere in that world by setting a horizontal and vertical offset. The numbers sent to the Suzy hardware will be the distance from the upper left corner of the display world to the upper left corner of the Suzy painting buffer. Portions of sprites outside the Suzy painting buffer will be 'clipped' by the hardware and therefore not drawn into any RAM. Since the offset numbers are used in the placement of a sprite in real RAM, changing the numbers will affect previously drawn sprites differently from sprites yet to be drawn. Any sprites drawn prior to a change in the offset numbers will remain unmoved in the Suzy painting buffer. Any sprites drawn after the change in the offset numbers will be positioned in accordance with those new numbers.

The Mikey Display buffer is the 8160 byte (102 lines x 80 bytes) area of RAM that is displayed on the LCD.

5.3.1 Display Buffer Address Register

The software can elect to have any number of display buffers. Two will allow for normal double buffering of the screen.

The value in the register is the start (upper left corner) of the display buffer in normal mode and the end (lower right corner) of the display buffer in FLIP mode. The address of the upper left corner of any display buffer must have '00' in the bottom 2 bits (only the upper 14 bits of the address are used). The hardware registers in Suzy are loaded with the start (upper left corner) of the painting or collision buffer, regardless of FLIP mode.

5.3.2 Display Buffer Address Register

The hardware address in Mikey (DISPADDR) is the backup value for the actual address counter. The backup value is transferred to the address counter at the very start of the third line of vertical blanking. In addition, the actual address register can be changed in real time by writing a 1 into the bit 'VBLANKEF' of Mikey register 'MTEST2'.

5.4 External Power Plug Detector

The presence of a plug in the 'external power' jack can be detected by the hardware. Whether or not any power is provided by the inserted plug is not actually discernable, since inserting the plug will disconnect the batteries. Therefore, the detection of an inserted plug ought to mean that the unit is operating on externally supplied power. Please note that this external power might be a large battery pack in addition to being an Ac adaptor. The main purpose of providing this information to the software is so that it can select the appropriate time-out values for the auto power down modes.

Since the I/O pin we are using used to be an output (backlight control), it is set to 'output' by the Mikey ROM. While this will not cause any damage to the parts, the pin will not function correctly until the software has set it to 'input'.

6 Sprite/Collision

This system has a 'sprite' engine that paints pixel information into a bit mapped screen. While the hardware does not actually manipulate individual sprites, the software 'sees' the hardware as a sprite system. The software may also directly read or write the bit mapped screen. While sprites are being painted, each pixel can be checked for collision with a previously painted sprite.

6.1 General Sprite Features

In addition to the normal features that accompany sprites, two major features are present in this system. One is that all sprites are sized in real time by the hardware prior to being painted to the screen. The other is that the data used to represent a sprite image is stored in RAM in a compacted format.

Feature List:

1. Sprites have unlimited vertical size.
2. A sprites horizontal size is limited by a maximum of 254 bytes of source data. This is approximately 508 pixels unscaled.
3. Sprites have 1, 2, 3, or 4 bits per pixel of 'pen index number'. Pen index is converted to pen number by the contents of a 16 nybble (8 byte) pen index palette specific to each sprite.
4. The horizontal and vertical reference point of a sprite can be defined to be any pixel within that sprite at the time the programmer performs the compaction of the original image. The starting quadrant must also be specified at that time. The reference point is not dynamically changeable at run time, but the starting quadrant is. Note that there will be a positional alteration if the starting quadrant is changed. The sprite image is then painted in each of the 4 quadrants from that reference point. This allows sized sprites to be referenced to their related objects (trees to the ground, doors to walls, etc.).
5. The processing of an actual sprite can be 'skipped' on a sprite by sprite basis.
6. Sprites or portions of sprites that are positioned off-screen will be clipped by the hardware during the painting process.
7. Sprites can be horizontally and/or vertically flipped. The pivot point is the sprite's reference point.
8. Sprites are sized as they are painted. They can be reduced or enlarged. Horizontal and vertical sizing rates are independent.
9. Sprite source data is unpacked as the sprite is painted. For the purpose of allowing for simple modification of sprite data (text , scores, etc.) the packing/unpacking algorithm allows for literal images. The literal definition does not affect sizing.
10. As a sprite is painted, the collision buffer (topographically identical to the display buffer) is checked for the existence of a previously painted sprite. If one exists, a collision has occurred. This collision information is saved by the hardware for later interrogation by the software. Several specific functions are performed by the hardware on the collision data so as to make it usable by the software. They are described elsewhere. The software may elect to not have a collision buffer and thus have an additional 8K bytes of RAM for the game. Sprites may be individually defined as non-colliding and thus reduce painting time.
11. The horizontal size of a sprite can be modified every time a scan line is processed. This allows for 'stretching' a sprite and in conjunction with 'tilt' can be useful in creating arbitrary polygons.

12. The horizontal position of a sprite can be modified every time a scan line is processed. This allows for 'tilting' a sprite and in conjunction with 'stretch' can be useful in creating arbitrary polygons.
13. The vertical size of a sprite can be modified every time a scan line is processed. This allows for 'stretching' a sprite vertically. The vertical stretch factor is the same as the horizontal stretch factor. Vertical stretching can be enabled on a sprite by sprite basis.

6.2 Specific Sprite Functionality

6.2.1 Pen Number Functions

The pen numbers range from '0' to 'F'. Pen numbers '1' thru 'D' are always collideable and opaque. They are the 13 generally usable pens for ordinary imagery. There are 8 types of sprites, each has different characteristics relating to some or all of their pen numbers.

6.2.1.1 Normal Sprites

A sprite may be set to 'normal'. This means that pen number '0' will be transparent and non-collideable. All other pens will be opaque and collideable. This is the sprite that is used for most 'action' images, weapons, enemies, obstacles, etc.

6.2.1.2 Boundary Sprites

A sprite may be set to 'boundary'. This is a 'normal' sprite with the exception that pen number 'F' is transparent (and still collideable). This allows for a 'boundary' at which a collision can occur without actual penetration (as in a ball bouncing off of a wall).

6.2.1.3 Shadow Sprites

A sprite may be set to 'shadow'. This is a 'normal' sprite with the exception that pen number 'E' is non-collideable (but still opaque). This allows for a non-colliding shadow on a sprite that is collideable (as in the shadow of a tree).

6.2.1.4 Boundary-Shadow Sprites

This sprite is a 'normal' sprite with the characteristics of both 'boundary' and 'shadow'. That is, pen number 'F' is transparent (and still collideable) and pen number 'E' is non-collideable (but still opaque).

6.2.1.5 Background-Shadow Sprites

A sprite may be set to 'background'. This sprite will overwrite the contents of the video and collision buffers. Pens '0' and 'F' are no longer transparent. This sprite is used to initialize the buffers at the start of a 'painting'. Since the buffers are not read before being written to, the painting process for this sprite will take approximately 1/4 less time. Additionally, no collision detection is done, and no write to the collision depository occurs.

There is no automatic clear of the video or collision buffers. If initialization is required, then it should be done using 'background' sprites. Note that partially used bytes (only one pixel) at either end of the line of pixels will do a read-modify-write cycle for that byte. The un-used pixel will remain unchanged.

BIG NOTE!!!

The 'E' error will cause the pen number 'E' to be non-collideable and therefore not clear the collision buffer. If you use a background sprite to clear the collision buffer, be aware that 'E' will not clear its respective pixel.

6.2.1.6 Background Non-Colliding Sprites

This is a 'background' sprite with the exception that no activity occurs in the collision buffer. This sprite will require 1/3 less time to paint than a normal background sprite.

6.2.1.7 Non-Collideable Sprites

A sprite may be set to 'non-collideable'. This means that it will have no affect on the contents of the collision buffer and all other collision activities are overridden (pen 'F' is not collideable). Since the collision buffer is not touched, the painting process for this sprite will take approximately 1/4 less time. This sprite can be used for 'non-action' images such as clouds, cockpits, scoreboards, etc.

6.2.1.8 Exclusive-Or Sprites

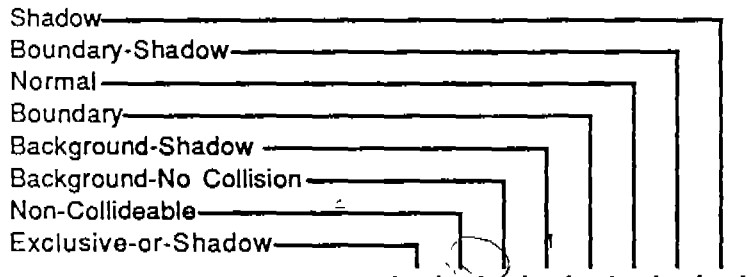
This is a 'normal' sprite with the exception that the data from the video buffer is exclusive-ored with the sprite data and written back out to the video buffer. Collision activity is 'normal'. Since a read-modify-write cycle is required for each video byte, this sprite could take up to 1/4 longer to paint than a 'normal' sprite.

BIG NOTE!!!

The 'E' error will cause the pen number 'E' to be non-collideable and therefore not react with the collision buffer. If you use an exclusive or sprite, be aware that 'E' will not collide with its respective pixel.

6.2.1.9 Hardware details of Sprite Types

The sprite types relate to specific hardware functions according to the following table:



1	1	1	1	0	1	0	1	F is opaque
0	0	0	0	1	1	0	0	E is collideable
0	0	1	1	0	0	0	0	0 is opaque and collideable
1	0	0	0	1	1	1	1	allow collision detect
1	0	0	1	1	1	1	1	allow collision buffer access
1	0	0	0	0	0	0	0	exclusive-or the data

6.2.1.9.E Shadow Error

The hardware is missing an inverter in the 'shadow' generator. This causes sprite types that did not invoke shadow to now invoke it and vice versa. The only actual functionality loss is that 'exclusive or' sprites and 'background' sprites will have shadow enabled. The above table shows the NEW truth.

6.2.2 Byte Boundary Conditions

Bytes will contain more than one pixel, some of those pixels may be transparent, some sprites may be non-aligned background types, and some pen numbers may be transparent. Rather than explain how to handle all of the variations of those conditions, I decided to make the hardware handle it automatically. Therefore, when using the sprite engine to paint to the screen, there are no special conditions relating to screen byte boundaries.

6.2.3 Horizontal Stretch

The horizontal size of a sprite can be modified every time a destination scan line of that sprite is processed. This includes scan line processing while the actual sprite is off screen. The modification consists of adding the 16 bit stretch value ('STRETCH') to the 16 bit size value ('HSIZE') at the end of the processing of a scan line. If the stretch value was 0, no effective modification occurs. This modification only takes place if enabled by the reload bits in SPRCTL1. Since the STRETCH 16 bit value consists of 8 bits of size and 8 bits of fraction, the stretch increment can be as small as 1/256 of a unit size and as large as 128 unit sizes. A STRETCH value of >128 unit sizes will cause a wrap in the ALU and result in a size reduction.

6.2.4 Vertical Stretch

The vertical size of a sprite can be modified every time a destination scan line of that sprite is processed. The specifics are the same as those for horizontal stretching. The value used is also the same as that used for horizontal stretching. Vertical stretch can be independently enabled or disabled. Even though the stretch factor is added at each destination line processing, the new vertical size only takes effect at the next source line fetch.

6.2.5 Tilt

The horizontal position of a sprite will be modified every time a scan line of that sprite is processed. This includes scan line processing while the actual sprite is off screen. The modification consists of 2 steps. The first is to get the 8 bit integer tilt value by adding the 16 bit tilt value ('TILT') to the tilt accumulator ('TILTACUM') at the end of the processing of a scan line, and shifting the answer to put the upper 8 bits in the lower position. The second is to add that integer to the horizontal position of the sprite ('HPOSSTRT'). This modification only takes place if enabled by the reload bits in SPRCTL1. The increments and negativeness are the same as for the stretch bit. Positive values of tilt will cause a tilt to the right. If the sprite is painting to the left (either due to flip or quadrant), then the software will have to set a negative tilt value if left tilt is desired.

6.2.6 Skip Sprite

The processing of a particular sprite can be mostly skipped. The only activity that will occur is the reading of the first 5 bytes of the SCB. It is required that those 5 bytes be legitimate, they will be loaded into the hardware. Note that the last 2 bytes are the address of the next SCB.

6.2.7 Horizontal and Vertical Size Offset

In order to balance the visual 'bump' that occurs at the reference point of a multi-quadrant sized sprite, we implemented a 'cheat' value of initial offset for both horizontal and vertical size. They are programmed independently at the time of sprite engine initialization. For horizontal, the left direction offset is forced to zero, and the right direction offset is programmed to 007F. For vertical, the up direction offset is forced to zero, and the down direction offset is programmed to 007F.

When using specific sizes to achieve specific effects, these offsets must be taken into account. You may also program them to any 16 bit value you desire. They are common to all sprites, so remember to set them back if you want the usual offset correction.

6.3 Collision Description

Sprites have a 4 bit collision number associated with them. The range of this collision value is 0->15. This number is the lower 4 bits of the 3rd byte in the SCB, 'SPRCOLL'. One of the upper 4 bits is used by the hardware to disable collision activity for this sprite. The other 3 are currently ignored by the hardware, but ought to be set to '0' for future compatibility. The software must assign this collision number for each use of each sprite. The rules for assignment are discussed in a software document.

There is a collision buffer that is topographically identical to the video buffer. The number in each 'cell' of the collision buffer is the collision number of the sprite whose video pixel is in the corresponding 'cell' of the video buffer. A value of '0' means that there is no collideable object in this pixel (although there may be a visible image). Any other number is the collision number of the last object that was painted into that pixel. This buffer is written to by the sprite painting process. All hardware collision detection is done with the data in the collision buffer, not the data in the video buffer. This has obvious advantages and will be explained at seminars and in newsletters forever.

At the completion of the processing of each collideable sprite, the hardware writes a byte into the 'collision depository' of that sprites SCB. The contents of that byte are the result of the collision process described below. After all of the sprites are painted, each of them will have a relevant number in their collision depository. These numbers can later be read by the CPU for sprite collision detection.

In addition, the software can either do its own collision detection, or use the contents of the collision buffer for some weird collision detection algorithm. In either event, I will be mortally offended.

6.3.1 Collision Process

During sprite painting to the video buffer, a collision process also takes place. Note that the collision process does NOT occur when the CPU directly accesses the video buffer. The collision process is also disabled by the appropriate selection of 'sprite type' or the setting of any of the other collision disable bits. If the sprite is 'collideable', the hardware will write to the collision depository each time the sprite is painted, relieving the software from having to clear the byte after collision detection. If the sprite is not 'collideable', the hardware will not write to the collision depository, and the software may wish to initialize the byte to a 'safe' value. 'Everon' also causes writing to the collision depository.

At the start of painting a particular sprite, a hardware register called fred is cleared to 0. In the course of painting this particular sprite, as each pixel is painted, the corresponding 'cell' in the collision buffer is read (actually done in bursts of 8 pixels). If the number read from the collision buffer cell is larger than the number currently in fred, then this larger number will be stored in fred. At the end of the processing of this particular sprite, the number in fred will be written out to the collision depository. If more than one collideable object was hit, the number in fred will be the HIGHEST of all of the collision numbers detected.

If a particular sprites collision depository value is zero, then this particular sprite did not collide with any other collideable object while it was being painted to the video buffer. This does not mean that this sprite will not be in collision with some other object that is painted later, or that it is not in visual collision with any object, it only means that the collideable pixels of this sprite did not overlay any collideable pixels ALREADY in the collision buffer. If, later on, a sprite is painted that collides with this sprite, then that later sprite will register the collision.

If a particular sprites collision depository value is non-zero, then this particular sprite collided with a collideable object that ALREADY existed in the collision buffer. The number found in the collision depository is the collision number of the object that was hit. It is hoped that the software will assign collision numbers in a meaningful fashion. My guess at a meaningful assignment is that the numbers will be assigned in visual depth order so that hardware detected collisions will be representative of visually noticed collisions.

6.3.2 Anti-Collision Devices

In response to popular demand, hardware collision is disableable in many ways.

1. The original method of declaring a sprite to be of type 'non-collideable'.
2. Set the 'don't collide' bit in the sprite collision number 'SPRCOLL' of the SCB.
3. Set the 'don't collide' bit in the system control byte 'SPRSYS'.

These bits override all other collision settings, and will disable all collision activity for either the current sprite (when set in 'SPRCOLL') or for all sprites (when set in 'SPRSYS').

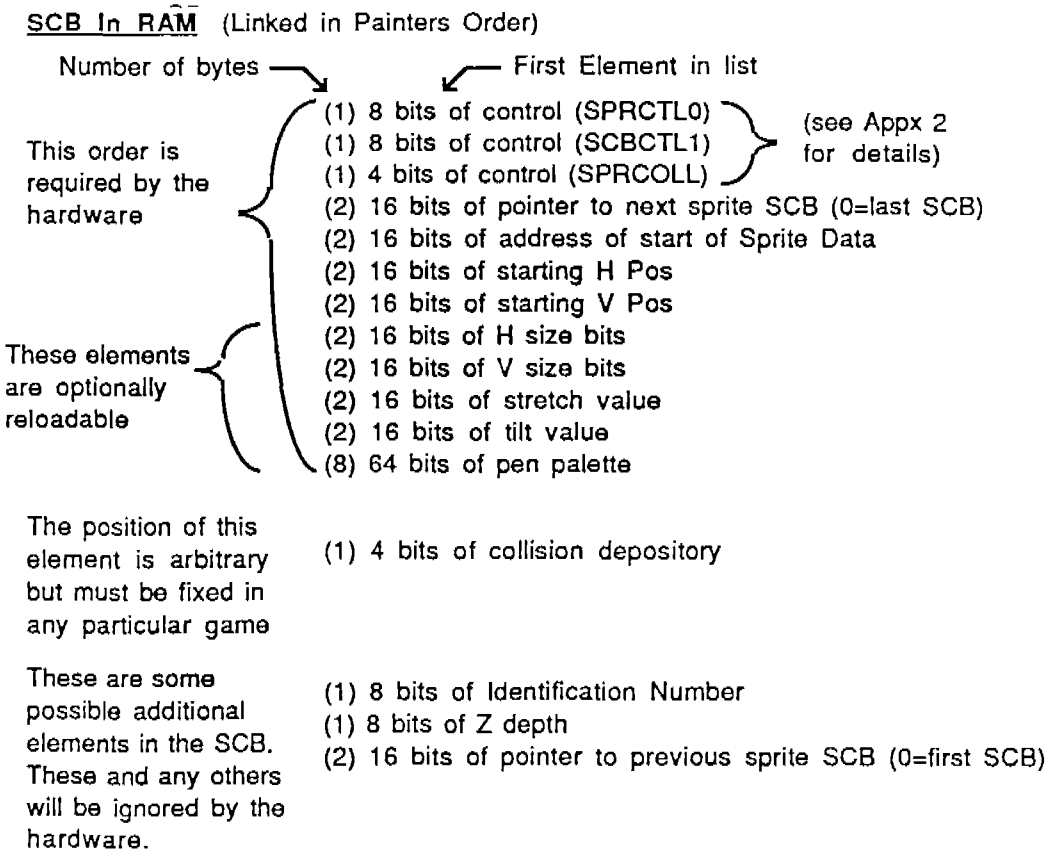
6.4 Sprite Data Structure

Sprite data consists of a sprite control block (SCB) and a sprite data block. The SCBs are linked by pointers in 'Painters Order'. Each SCB also points to the sprite data block containing the image of interest. Many SCBs may point to one sprite data block. Each occurrence of a sprite on the screen requires 1 SCB. Since all of the SCBs and sprite data blocks are accessed by pointers, they may be located anywhere in RAM space. Neither SCBs nor sprite data may be located in Mikey ROM.

6.4.1 Sprite Control Block

Each SCB contains certain elements in a certain order as required by the hardware. In addition, elements may be added by the software as desired. The hardware will not be aware of, nor be affected by, these additional elements.

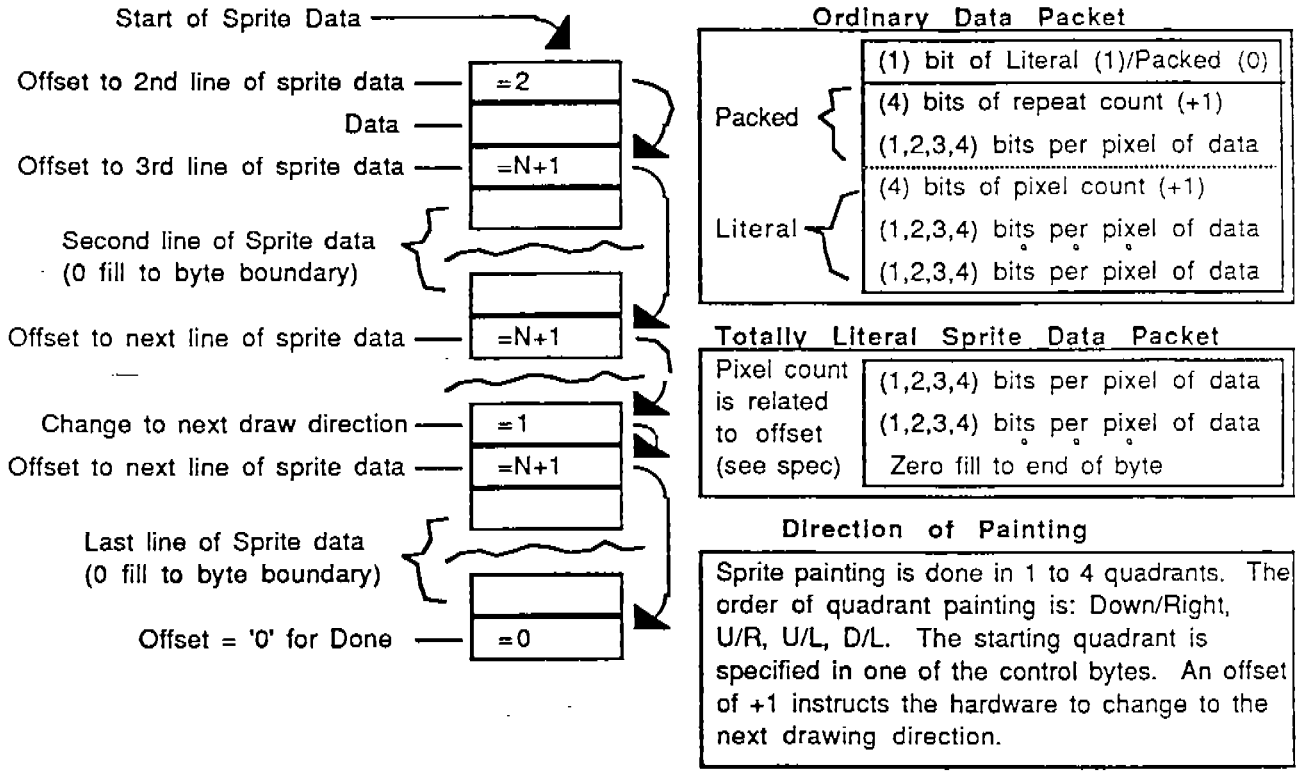
This list shows the identification and order of the required elements and some possible additional elements.



The 8 bytes of pen palette are treated by the hardware as a separate block of data from the previous group of bytes in the SCB. This means that the reloadability of some of the previous bytes does not affect the reusability of the pen palette. In addition, this means that when some of the bytes are not reloaded, the length of the SCB will be smaller by the number of bytes not used. If I have said this in a confusing manner, then I have.

6.4.2 Data Packing Format

All sprite data is formatted. The format consists of offsets and actual image data. The image data can be totally packed, a combination of packed and literal, and totally literal. The format is shown below:



Well, I finally found the bug that required a pad byte of 0 at the end of each scan line of data. But, It is actually 2 bugs. I have fixed one of them, but the other requires an extensive change. Too bad, I am out of time. Therefore:

There is a bug in the hardware that requires that the last meaningful bit of the data packet at the end of a scan line does not occur in the last bit of a byte (bit 0). This means that the data packet creation process must check for this case, and if found, must pad this data packet with a byte of all 0s. Don't forget to adjust the offset to include this pad byte. Since this will only happen in 1/8 of the scan lines, it is not enough overhead to force me to try to fix the bug. Sorry.

A data Packet header of '00000' is used as an additional detector of the end of the line of sprite data. This is normally used as the filler in the last byte of a line of data, but it can be located anywhere in the data line, not just the last byte. Note that this would ordinarily decode to a packed packet of 1 pixel. Please use a literal packet of 1 pixel when the imagery calls for such an event. In addition, this special header is used when a scan line with no pixels is desired. You must use an offset of 2 with a data byte of all 0, since an offset of 1 will cause a change of painting direction.

oops, may have a bug, more later on the truth of '00000'

In a totally literal sprite, there is no pixel count. Source data is converted to pixels until it runs out of bytes in that line of sprite data. The odd bits that may be left over at the end of the last byte will be painted. The special data packet of '00000' is not recognized in a totally literal sprite.

6.5 Bits in Bytes

In the display buffer, there will be 2 pixels in a byte of data. When painting the data to the right on an un-flipped screen, the pixel in the most significant bits will be painted first, and therefore on the left side of the line. The completed bytes will be painted in ascending order of byte address.
The picture is:

LEFT RIGHT

	address 0		address 1		address 2	
	pix0	pix1	pix2	pix3	pix4	pix5
Bits	7654	3210	7654	3210	7654	3210

This order also applies to the pen index palette in the SCB. The pen number referenced by PIN 0 (Pen Index Number) is in the UPPER nybble of the '01' byte.

6.6 Sprite Engine Description

The sprite engine consists of several 8 bit control registers, a 16 bit wide sprite control block register set and ALU, an address manipulator, an 8 byte deep source data FIFO, a 12 bit shift register for unpacking the data, a 16 nybble pen index palette, a pixel byte builder, an 8 word deep pixel data FIFO, a data merger, and assorted control logic.
The basic flow is:

1. The address manipulator gets the address of the source data, gets chunks of 4 bytes for the data FIFO, and appropriately increments the address.
2. The unpacker logic pulls data from the FIFO, serializes it in the shift register, and forms it into individual pen index numbers.
3. The pen index number is fed to the pen index palette which produces a 4 bit pen number,
4. The 4 bit pen numbers go to the pixel byte builder where their transparency and collideability are noted, they are combined into bytes, and placed in the pixel data FIFO.
5. The pixel data is then merged with existing RAM data in the video buffer (using read-modify-write cycles in the RAM) based on transparency and other control bits.
6. The collideability data is then used to perform the collision process with the collision buffer.

All of the operations are pipe-lined and operate in an interleaved fashion that reduces bottlenecks and results in almost the highest system performance possible.

6.6.1 Sprite Engine Use

In order to correctly use the sprite engine, 3 things must be done. First, the CPU must initialize certain registers within the engine. Second, appropriately formatted sprite control blocks (in RAM) must be created, and they must point to correctly formatted sprite data blocks. Third, the sprite engine must be started by the CPU. The start bit is described in the hardware address appendix. The formats of sprite control blocks and sprite data blocks are described in this document. The required initialization is described here. A condensed 'cheat-sheet' can be created later.

6.6.2 Sprite Engine Initialization

Certain registers must be initialized in the sprite engine after power up. It was not done in hardware for two reasons. First is that the engine is useful for other tasks besides painting sprites; hardware divide, hardware multiply, etc. Second is that it would have been too expensive in silicon to perform the initializations on the style of register array used.

The initialization for the purpose of painting sprites is listed below. The order that the registers are initialized is not important as long as ALL of them are initialized to a legitimate value before the sprite engine is turned on.

All of these values can be changed during the course of operation as long as all of them remain legitimate.

Initialization steps for sprite painting:

1. Load the appropriate value into the system control register (SPRSYS).
2. Write the correct value into (SPRINIT).
3. Write the horizontal pixel offset from zero to the left edge of the screen into HOFF.
4. Write the vertical pixel offset from zero to the top edge of the screen into VOFF.
5. Write the base address of the collision buffer into COLLBASE.
6. Write the value of the offset between the collision depository location in a RAM sprite control block and the location of the first byte of a RAM sprite control block into COLLOFF.
7. Write the magic numbers into (HSIZOFF) and (VSIZOFF).
8. Write a '01' to SUZYBUSEN, giving Suzy permission to access the bus.

When you are ready to start the sprite engine, the following steps are required:

1. Write the base address of the video build buffer into VIDBASE.
2. Write the address of the start of the first RAM sprite control block into SCBNEXT.
3. Write a '01' (or '05') to SPRGO, turning on the sprite engine.
4. Write a '00' to SDONEACK, allowing Mikey to respond to sleep commands.
5. Put the CPU to sleep, allowing Suzy to actually get the bus.

Writing a 01 or 05 to SPRGO sets the 'SPRITESEN' flip flop which causes the sprite engine to start its operation. When the engine finishes processing the sprite list, or if it has been requested to stop at the end of the current sprite, or if it has been forced off by writing a 00 to SPRGO, the SPRITESEN flip flop will be reset. This engine starts only from its beginning. It can not be stopped and

restarted from its stopped point. Stopping the engine does not affect the contents of any of the other registers.

SUZYBUSEN contains the bit that allows Suzys access to the bus. It does not affect the internal operation of the sprite engine until such time that the sprite engine needs the RAM bus. Then the engine will wait until it gets the bus before continuing its processing. It is suspected that in ordinary sprite engine use, this bit will be set once and then left on. This bit can be used to pause the sprite engine. The 'unsafe' condition of the internal registers is not directly affected by this bit.

I need to think about how to use it.

7 Audio

There are four identical audio channels. The 4 signals are digitally mixed and provided to an analog filter. The output of the filter is amplified and drives a speaker. There is a manual volume control whose range includes ZERO output. There is a stereo earphone jack that outputs the mono sound on both channels.

7.1 Audio Channel Specification

Each audio channel consists of a programmable base frequency, a programmable polynomial sequence generator, a waveshape selector, and a programmable volume control. The specific bits and addresses are listed in appendix 2. Functionally, they are:

3 bits of counter pre-selector giving a clock choice of 1us, 2us, 4us, 8us, 16us, 32us, 64us, and previous. Previous is the reload state of the previously processed counter. This is used to cascade audio channels or timers.

8 bits of down counter. The counter is clocked by the output of the pre-selector (source clock). When the counter reaches 0, it is reloaded with the contents of its 8 bit backup register (at the next edge of its source clock). The period of this reload is the product of the number in the backup register + 1 and the source clock. This period constitutes the base frequency and clocks the polynomial sequencer. The value of 0 in the down counter is valid for 1 full cycle of the output of the pre-selector. A down count of 3 results in 4 cycles of the source clock.

12 bits of shift register and 9 bits of tap selector. 9 outputs of the 12 bit shift register are individually selectable as inputs to a large exclusive or gate. The inversion of the output of the gate is used as the data input to the shift register. This is known as a polynomial sequence generator and can generate waveforms that range from square (for musical notes) to pseudo random (for explosions). This same inverted output is taken from the exclusive or gate and sent to the waveshape selector.

1 bit of waveshape selector. The rectangular waveform from the polynomial generator can be unmodified or integrated. The purpose of the integration is to create an approximately triangular waveshape. This sounds 'better' in many cases than a rectangular waveshape.

7 bits of volume control and 1 bit of sign (2s complement number notation). Each channel has its own 128 settings of volume, with an 'invert' bit. The range includes ZERO output.

7.2 Audio filter

The 4 audio channels are mixed digitally and a pulse width modulated waveform is output from Mikey to the audio filter. This filter is a 1 pole low pass filter with a frequency cutoff at 4 KHz. The output of the filter is amplified to drive an 8 ohm speaker.

7.3 Stereo

The 4 audio channels can be individually enabled into either of the two output drivers. This is not now in the hardware. It may never get into the hardware. After all, I hate music. However, since I acknowledge that I am wrong about some kinds of music (in the right circumstances, with me not present) I have graciously allocated one entire byte for the possible implementation of another useless and annoying sound feature.

7.4 Bug in the Lower nybble

The lower nybble of the audio out byte is processed incorrectly in the digital to pulse width converter. The upper bit of this lower nybble should have been inverted prior to conversion. This is how we achieve a 50% duty cycle for 0 volume. (we did it right for the upper nybble) This error results in a single glitch in the sound when the value transitions from 8 to 9.

The effect is at most, 1/16 of the max volume for one tick of the D/A clock, and is generally not noticed.

8 Magnetic Tape

The circuit for reading magnetic tape is in Mikey. However, the Mikey ROM does not currently activate the tape mode. When the decision to use tape is made, the ROM may have to be changed. The alternative would be to use an external ROM that would activate the tape circuits.

9 ROM Cart

9.1 ROM Cart Address Space

In a ROM Cart unit, the addresses for the ROM Cart are provided by an 8 bit shift register and a 11 bit counter. A particular ROM Cart will be wired to the address generator such that the upper 8 bits of its address will come from the 8 bit shift register and the remaining lower bits of its address will come from the lower bits of the counter. A 64k byte ROM Cart will have 8 bits of counted address and 8 bits of shifted address. A 128k byte ROM Cart will have 9 bits of counted address and 8 bits of shifted address. The maximum address size is (8+11) 19 bits which equates to 1/2 megabyte of ROM Cart address space. Since there are 2 strobes available to the cart, there is a total of 1 megabyte of address space without additional hardware support.

The 8 bit shifter is controlled by 2 signals from Mikey, 'CartAddressData' and 'CartAddressStrobe'. 'CartAddressData' is the data input to the shift register and 'CartAddressStrobe' is the clock to the shift register. The shift register accepts data from the 'CartAddressData' line on rising (0 to 1) transitions of the 'CartAddressStrobe' line. Data is shifted into the register most significant bit first. The 'CartAddressStrobe' line is also the reset signal for the 11 bit counter. The counter is reset whenever the line is high (1). The software must remember to set the 'CartAddressStrobe line low after shifting in the address so as to release the reset of the counter.

9.2 ROM Cart Data Read

In order to prevent a reduction of battery life when using the ROM Cart, we only enable the ROM itself when we wish to read a data byte. We do it in such a manner as to guarantee that no RAM access can occur simultaneously with a ROM access. This is achieved by driving one of the 'chip enable' lines from SUZY. The trailing edge of this chip enable signal also advances the address counter by 1. Only one address is available to the Cart and the activation of either chip enable signal will increment the address.

The 8 tri-state data bits of the ROM are tied to 8 of the switch reading lines on SUZY. The switches are isolated from the ROM by resistors. When a read request comes from MIKEY, Suzy will hold off MIKEYs DTACK, enable the ROM (which overdrives any pressed switches), wait 437.5 ns (for ROM access time), and disable the ROM. Towards the end of the wait, SUZY passed the data from the ROM to Mikey and released the DTACK. The actual access time available at the pins of the Cart is 392 ns. The CPU cycle that performed the actual read uses 15 ticks of the clock.

9.3 ROM Cart Data Write

The ROM Cart can also be written to. The addressing scheme is the same as for reads. The strobe is also self timed. The length of the strobe is 562.5 ns, the data is stable for 125 ns prior to the strobe and for 62.5 ns after the strobe. This is a 'blind' write from the CPU and must not be interrupted by another access to Suzy until it is finished. The CPU must not access Suzy for 12 ticks after the completion of the 'blind' write cycle.

9.3 ROM Cart Power-Up

Since some types of ROM do not have a useful power-down mode, we provide a switched power pin to the cartridge. This pin is controlled by the state of the 'CartAddressData' signal from Mikey. Yes, this is the same pin that we use as a data source while clocking the address shift register and therefor, we will be switching ROM power on and off while loading that register. Unless the software is poorly arranged, that interval of power switching will be short. The switched power pin is powered up by setting the 'CartAddressData' signal low. It is suggested that the pin be powered up for the read of any ROM cart since carts that do not need it will not be wired to that pin. Additionally, information in that ROM cart can tell the software if it needs to further manipulate the pin.

10 Timers/Interrupts

10.1 Timers

There are 8 independent timers. Each has the same construction as an audio channel, a 3 bit source period selector and an 8 bit down counter with a backup register. This gives a timer range of 1 us to 16384 us. Timers can be set to stop when they reach a count of 0 or to reload from their backup register. In addition, they can be linked, with the reload of one timer clocking the next timer.

The linking order is as follows:

Group A:
Timer 0 -> Timer 2 -> Timer 4.

Group B:
Timer 1 -> Timer 3 -> Timer 5 -> Timer 7 -> Audio 0 -> Audio 1->
Audio 2 -> Audio 3 -> Timer 1.

As with the audio channels, a count of 0 is valid for 1 full cycle of the selected clock. A backup value of 5 will result in 6 units of time. Actually, due to hardware limitations, the first utilization of that timer after it has been set will result in a time value between 5 and 6 units. Subsequent utilizations (reload is 'on') will result in a value of 6 units.

10.2 Timer Utilization

Two of the timers will be used for the video frame rate generator. One (timer 0) is set to the length of a display line and the second (timer 2) is set to the number of lines.

One of the timers (timer 4) will be used as the baud rate generator for the serial expansion port (UART).

Note that the hardware actually uses the bits in these timers and it would be dangerous to arbitrarily fiddle with them in software.

The other 5 timers are for general software use. POOF. See the mag tape description.

10.3 Interrupts

7 of the 8 timers can interrupt the CPU when it underflows. Each interrupt can be masked. The value of the interrupt bit can be polled independent of its mask condition. The interrupt bit for timer 4 (UART baud rate) is driven by receiver or transmitter ready bit of the UART.

If an interrupt occurs while the CPU is asleep, it will wake up the CPU.

Since one of the timers is the vertical line counter, the useful 'end of frame' interrupt can be generated there.

The interrupt signal comes from the timer when the timer value is zero AND the timer is attempting to perform a 'borrow'. Based on the control bits, the borrow may not actually occur, but the interrupt signal will. This signal then requests the bus control circuit to give the bus to the CPU. If the CPU already has the bus, then this function causes no delays. If Suzy has the bus, then the maximum Suzy latency time could be incurred. Then, the interrupt signal waits for the end of the current CPU cycle before actually interrupting the CPU.

11 UART

The Mikey UART is a pretty standard serial kind of thing. However, since most of the communicating world has managed to bolix up the definitions and use of the so-called standards of serial communications, I will attempt to explain all of our UARTs functions. Yes, I will leave something out so as to continue to trick earthlings into thinking that I am human.

11.1 Connector Signals

The UART connects to the REDEYE connector which has pins. There is 1 data pin. This signal is bi-directional serial data. Its idle state is open-collector with a pull-up resistor to +5. There is one ground pin, and one VCC pin.

(a design error causes the power up state of the output to be TTL high, ALL code must set the TXOPEN bit in order to fix this. Its not my fault.)

There is also a signal called NOEXP which indicates the presence of a plug in the REDEYE socket.

Any devices other than RedEye will need to have an intelligent protocol to distinguish themselves from a RedEye type device. See the RedEye specification for protocol details.

11.2 Baud Rate

The baud rate is generated by TIMER4 according to the equation

CLOCK4 / (TIMER4 + 1) / 8

The minimum number that can be used in TIMER4 is 1, the maximum number is 255. The fastest CLOCK4 is 1MHz, the slowest is 15625 Hz. This gives a baud rate maximum of 62.5 Kbaud and a minimum of 7.63 baud. The settings for the common baud rates are:

Baud Rate	CLOCK4	TIMER4	Actual
62500	1us	1	62500
9600	1us	12	9615
2400	1us	51	2404
1200	1us	103	1202
300	2us	207	300.5

11.3 Data Format

The serial data format is the standard 11 bits. We do not offer a choice.

The standard bits are:

1 start bit (binary 0);
8 data bits, LSB first;
1 parity bit (or 9th bit as defined by the control byte);
1 stop bit (binary 1).

The parity (or 9th) bit operates as follows:

Receive:

The state of the 9th bit is always available for read in the control byte. In addition, the parity of the received character is calculated and if it does not match the setting of the parity select bit in the control byte, the parity error bit will be set. Receive parity error can not be disabled. If you don't want it, don't read it.

Transmit:

The 9th bit is always sent. It is either the result of a parity calculation on the transmit data byte or it is the value set in the parity select bit in the control register. The choice is made by the parity enable bit in the control byte. For example:

If PAREN is '1' and PAREVEN is '0', then the 9th bit will be the result of an 'odd' parity calculation on the transmit data byte.

If PAREN is '0', then the 9th bit will be whatever the state of PAREVEN is.

We have just discovered that the calculation for parity includes the parity bit itself. Most of us don't like that, but it is too late to change it.

11.4 Break

A break of any length can be transmitted by setting the transmit break bit in the control register. The break will continue as long as the bit is set.

A 'break' is defined as a start bit, a data value of 0 (same as a permanent start bit), and the absence of a stop bit at the expected time.

The receiver requires that a break lasts 24 bit times before it is recognized. There are many 'standard' break lengths, this is the one we chose.

11.5 Transmitter Status Bits

There are 2 status bits for the transmitter, TXRDY (transmit buffer ready) and TXEMPTY (transmitter totally done).

If TXRDY is a '1', then the contents of the transmit holding register have been loaded into the transmit shift register and the holding register is now available to be loaded with the next byte to be transmitted. This bit is also set to '1' after a reset.

If TXEMPTY is a '1', then BOTH the transmit holding register and the transmit shift register have been emptied and there are no more bits going out the serial data line.

11.6 Errors

There are 3 receive errors, parity error (already explained), framing error, and overrun error. Once received, these error bits remain set until they are cleared by writing to the control byte with the reset error bit set. Writing to the control byte with the reset error bit cleared has no effect on the errors. Note that the reset error bit is NOT an error enable bit. Receive errors are always enabled.

Framing error indicates that a non-zero character has been received without the appropriate stop bit.

Overrun error indicates that a character (of any kind or error) has been received and the previously received character has not yet been read from the receive buffer.

11.7 Unusual Interrupt Condition

Well, we did screw something up after all. Both the transmit and receive interrupts are 'level' sensitive, rather than 'edge' sensitive. This means that an interrupt will be continuously generated as long as it is enabled and its UART buffer is ready. As a result, the software must disable the interrupt prior to clearing it. Sorry.

11.8 left out

I left something out. I know what it is but by the next time I revise this spec, I may have forgotten.

I have forgotten.

12 Other Hardware Features

12.1 Hardware Multiply, Accumulate and Divide

We have a 16 by 16 to 32 unsigned and signed multiply with accumulate and a 32 / 16 into 32 unsigned divide. The results of a multiply are a 32 bit product, a 32 bit accumulate and an accumulator overflow bit. The results of a divide are a 32 bit dividend, a 16 bit remainder, and a flag bit indicating a divide by zero. The number in the dividend as a result of a divide by zero is 'FFFFFFFF' (BigNum). The accumulator is 32 bits and accumulates the result of multiply operations.

The basic method of performing one of these math operations is to write the starting values into Suzy registers (all of the addresses are different) and then polling for completion prior to reading the results. The act of writing to the last register starts the math process.

The functions of 'sign/unsign' and 'accumulate' are optional. The value for SPRINIT is currently unchanged for the multiply and divide operations. The following picture shows the operations available.

Each letter represents a different byte address. These addresses are identified in the hardware address Appx 2. Each grouping represents the kind of math operation available.

```
   AB                    EFGH
* CD                  / NP
EFGH                    ABCD
Accumulate in JKLM    Remainder in (JK)LM
```

Some Rules:

Writing to B,D,F,H,K, or M will force a '0' to be written to A,C,E,G,J, or L, respectfully. Therefore, if you only have 8 bits in a particular number, there is no need to write the upper byte to '0'. (except for signed multiplies)

Writing to A will start a 16 bit multiply.

Writing to E will start a 16 bit divide.

The actual steps required to perform some of the functions are:

16 x 16 multiply:

Write LSB to D, MSB to C
Write LSB to B, MSB to A
Poll MULTSTAT until done (or just wait for 54 ticks)
Read answer (LSB->MSB) from H,G,F,E

Accumulate:

To initialize the accumulator, write a '0' to K and M (This will put 0 in J and L). The write to 'M' will clear the accumulator overflow bit. Note that you can actually initialize the accumulator to any value by writing to all 4 bytes (J,K,L,M).

32 x 16 divide:

Write LSB to P, MSB to N,
Write LSB -> MSB to H,G,F,E
Poll DIVSTAT until done (or just wait 400 ticks max)

Read answer (LSB->MSB) from D,C,B,A
Read remainder (LSB->MSB) from M,L

As a courtesy, the hardware will set J,K to zero so that the software can treat the remainder as a 32 bit number.

12.1.1 Signed Multiplies

When signed multiply is enabled, the hardware will convert the number provided by the CPU into a positive number and save the sign of the original number. The resultant positive number is placed in the same Suzy location as the original number and therefore the original number is lost. At the end of a multiply, the signs of the original numbers are examined and if required, the multiply result is converted to a negative number.

The conversion that is performed on the CPU provided starting numbers is done when the upper byte is sent by the CPU. Therefore, while writing to the lower byte will set the upper byte to zero, it WILL NOT change its sign to positive. Therefore, when using signed multiply, you MUST write both bytes of a number.

12.1.2 Repetitive Multiplies and Divides

Since none of the input values are disturbed, repetitive multiplies and divides are easily done by only changing the required bytes and the 'starting' byte.

There are several restrictions in the utilization of hardware multiply and divide:

1. Suzy is not processing sprites.
2. The steps to perform the function must be performed exactly as specified. Failure to do so will result in botched data.
3. The hardware is not re-entrant. This means that once started, a multiply or divide can not be messed with.

There is an interrupt consideration. Although the hardware is not re-entrant, some clever interrupt routine may wish to wait for an operation to complete, save the answer, perform its own operation, get its own answer, and then restore the previous values. While this works for the simple singular operation, it will disturb the input values which may still be required by the original software that is performing repetitive operations. If the interrupt software wants to save and restore the original input values, it will start a new math operation when it writes them back to the hardware. The various implications of this act have not yet been explored, so be cautious.

12.1.3 Math Timing

Multiplies with out sign or accumulate take 44 ticks to complete.
Multiplies with sign and accumulate take 54 ticks to complete.

Divides take 176 + 14*N ticks where N is the number of most significant zeros in the divisor.

12.1.4 Bugs in MathLand

BIG NOTE: There are several bugs in the multiply and divide hardware. Lets hope that we do not get a chance to fix them.

1. In signed multiply, the hardware thinks that 8000 is a positive number.
2. In divide, the remainder will have 2 possible errors, depending on its actual value. No point in explaining the errors here, just don't use it. Thank You VTI.
3. In signed multiply, the hardware thinks that 0 is a negative number. This is not an immediate problem for a multiply by zero, since the answer will be re-negated to the correct polarity of zero. However, since it will set the sign flag, you can not depend on the sign flag to be correct if you just load the lower byte after a multiply by zero.
4. The overflow bit is not permanently saved in the hardware. You must read the overflow bit before doing anything to Suzy that might disturb it. Basically, read the overflow bit before you access the SCB registers.

12.2 Upward Compatibility

Hardware changes can be desired for many reasons. We will probably not implement desired hardware changes for many other reasons. However, we may introduce product upgrades for which software compatibility is important.

To support the upgrades, a read only hardware identifier register and a write only software identifier register in both Mikey and Suzy (4 registers total) will be provided. These registers will have 'bit' significance in that the individual bits will have individual revision or capability significance.

In the first Mikey and Suzy, the hardware identifier value will be '01'. The software identifier register will not actually exist, only its address is allocated.

12.3 Parallel Port

Some of the prototypes will have a parallel port for debug purposes. The port uses 2 addresses, one for status and one for data. Both are read/write. See the address appendix for the details. Basically, you write to the status address to set the port for either input or output, and to wiggle the 'Paper Out' pin. You read the status port to see if the port has data available, is ready for data, or read a the 'Busy' pin. The hardware handshake timing on the parallel port itself is totally handled in hardware.

Of special note is that changing the direction of the data port will clear the data available bit.

Additionally, the 'SEL' line in the parallel cable is connected to the NMI/ line of the CPU. Pulling it low will cause an NMI to happen in the handy CPU. Don't forget to set it high again before the handy CPU finishes the NMI software routine. If you are wiggling it from an Amiga, you can pull it high as soon as you want to. The Amiga hardware will not let its duration be too short.

12.4 Qbert Root

As a compromise between the square root and qube root desires of the software people, and the schedule desires of the management, we have decided to incorporate the function of QbertRoot. The required steps are:

1. Start a 16 by 16 multiply.

2. Immediately write to 'E' which will try to start a divide.
3. Read the result from "D,C,B,A'.

12.5 Everon

At a particular point in the life of a sprite, it may move in such a manner that it no longer appears on the screen. It may also have started its life off-screen. It appears to be useful to the software to know that a sprite is completely off-screen. The function used to discover this truth is called 'Everon' and is enabled by the 'Everonoff' bit. The function is enabled when the sprite engine is started with an 05 instead of an 01 at 'SPRGO'. This function will cause the 'Everon' bit to reflect the off-screen situation of a particular sprite. This bit is returned to each SCB in bit 7 of the collision depository. The bit will be a '0' if Everon is disabled or if the sprite is ever on the screen. The bit is a '1' only when Everon is enabled (Everonoff is on) and the sprite is neveron the screen.

12.6 Unsafe Access

As expressed earlier in this document, there are times when certain addresses are unsafe to access. If, by some unknown means, the software does an unsafe access, the unsafe access bit in SPRSYS will be set. This bit will remain set until it is cleared as explained in the hardware address appendix.

BIG NOTE: Unsafe access is broken for math operations. Please reset it after every math operation or it will not be useful for sprite operations.

12.7 Howie Notification

For purposes of diagnostics, there is a Suzy address that is used to notify the Howard board of a desired communication. See the hardware address spec.

12.8 General I/O Port

In the beginning, there was a general purpose 8 bit I/O port. As pins on Mikey became unavailable, the number of bits was reduced. Now all we have are the 5 bits of IODAT and they are not even pure read/write.

The direction of the pins (in or out) still needs to be set even though all but one are forced in the PCB to be either an in or an out but not both. The function of the bits are not apparent from the description in the address appendix, so I will explain them here.

1. External Power.

   This bit detects the presence of a powered plug. The ROM sets it to an output, so the system code must set it to an input.

2. Cart Address Data.

   This bit must be set to an output. It has 2 functions. One is that it is the data pin for the shifter that holds the cartridge address. The other is that it controls power to the cartridge. Power is on when the bit is low, power is off when the bit is high.

3. Noexp.

   This bit must be set to an input. It detects the presence of a plug in the expansion connector.

4. Rest.

   This bit must be set to an output. In addition, the data value of this bit must be set to 1. This bit controls the rest period of the LCD display. The actual Rest signal is a high except during the last 2 scan lines of vertical blank and the first scan line after vertical blank. It is generated by the vertical timing chain in Mikey but its output on this pin can be disabled by the incorrect setting of this bit. In addition, when reading this bit, you will get the actual state of the Rest signal 'anded' with the value of the bit set in IODAT. Yes, we know that the polarity of the name is wrong, but that is the way it came to us.

5. Audin.

   This bit can be an input or an output. In its current use, it is the write enable line for writeable elements in the cartridge. It can also be used as an input from the cartridge such as 'audio in' for 'talking-listening' games. Whether it is set to input or output, the value read on this pin will depend on the electronics in the cartridge that is driving it.

6. The other bits.

   The other 3 bits in the byte are not connected to anything specific. Don't depend on them being any particular value.

13 Expansion Connector

The expansion connector is a 3 wire stereo earphone jack. The pin assignments are:

Shield: Ground
Tip: VCC
Center: Data

There are certain considerations for data transfer, more later.

Note that when using a wire, any unit that is powered off and connected to the link will disable communications for all of the units in the link.

14 System Reset/Power Up

The state of all of the control bits during and immediately after system reset is given in appendix 2. During reset, the main clocks are still running and are used to propagate the reset condition thru the logic. Reset must be long enough to insure a stable and consistent startup.

The process that takes place upon release from system reset and the steps required to initialize the hardware are described below.

14.1 Suzy Reset Recovery.

At release from reset, all of Suzy is disabled and remains so until enabled by the CPU. During reset, the bus handshake logic has stabilized with bus grant asserted. Any incidental internal activity will be listed when the logic design is completed. Suzy is not expected to require any software initialization until a specific function of Suzy is desired.

14.2 Mikey Reset Recovery.

At release from system reset, the video bus request is disabled and will remain so until enabled by the CPU. The CPU is requesting the bus. During reset, the bus handshake logic has stabilized and is ready to give the bus to the CPU. The ROM overlay is enabled and the CPU will fetch its reset vector from ROM at the normal 6502 address of FFFC and FFFD.

Since there are some pieces of hardware that are running at rates slower than the main system clock, there is the possibility that there will not be a deterministic relationship between the phases of these slower clocks (eg. CPU and audio/timer). To prevent possible peculiarities of operation (due to hardware bugs) and to assist in truthful software emulation, it is suggested that these individual hardware chunks get synchronized at system start. It may be expensive to do it all in silicon, so some of them may need to be done by software. At this time, Glenn says that the audio section is done in silicon. There are other hardware registers that must be initialized correctly in order to let the system come up consistently. They are stated in appendix 2 (hardware addresses).

The current process that must be followed at system initialization is:

1. Disable interrupts
2. Clear decimal mode
3. Read SUZYHREV
4. If = 0, jump to test code
5. Read 8 locations in RAM (each must cause a new RAS to occur)

(more ??)

15 Definitions and Terminology

The polarity of signals and their effect on the circuitry is not always apparent from the name of the signals. For this document and its related documents and appendices, the following conventions will be used:

Set = On = Asserted = Active = Occurs

These mean that the signal is in the state that causes or allows its named function to occur. For example, if bus grant is set, we are granting a bus. If an interrupt mask bit is set, that interrupt is masked.

Reset = Off = Cleared = De-asserted = Inactive = Dropped

These mean that the signal is in the state that does not allow its named function to occur. For example, when bus request is dropped, we are no longer requesting the bus.

The terms 'hi' and 'low' are well applied to actual schematic referenced signals, but should be avoided in generally descriptive text.

Acquired = Taken

These relate to tri-state busses and indicate that the bus is now being driven by the 'acquirer'.

Released = Let Go

These also relate to tri-state busses and indicate that the 'releaser' is no longer driving the bus.

SimWait = Boil

The act of waiting for garbage collection on the VTI tools

16 Known Variances From Anticipated Optimums

This is a list of the known bugs in the hardware.

16.1 Mikey

1. Sleep does not work if Suzy does not have the bus.
2. The UART interrupt is not edge sensitive.
3. The UART TXD signal powers up in TTL HIGH, instead of open collector.
4. The lower nybble of the audio out byte is processed incorrectly.
5. The Reset Timer Done bit is not a pulse.
6. The Timer Done bit requires clearing in order to count.
7. The Mikey ROM sets the External Power Detect pin to output.
8. The REST pin on Mikey is initialized as an input and its data is set to 0. Both are wrong. You must set it to an output with a data value of 1.
9. The IODAT register is not really a R/W register.

16.2 Suzy

1. Remainder does not correctly handle the upper bit in two ways.
2. Signed multiply thinks 8000 is a positive number.

3. Auto clear of the upper byte of an SCB word does not also auto clear the sign flag.
4. The page break signal does not delay the end of the pen index palette loading.
5. The polarity of the 'shadow' attribute is inverted.
6. Signed multiply thinks that 0 is a negative number. (delayed effect)
7. The circuit that detects a '0' in the SCB_NEXT field of an SCB only looks at the upper byte. Therefore, we can't have scabs in page 0. I'm sorry.
8. A data packet header of '00000' can be used to indicate end of data. There appears to be a bug with it. I don't understand it yet. Beat me. Kick me.

17 Approved Exemptions

1. Only the upper byte of the 'NEXT' word in the SCB needs to be set to zero in order to indicate that this is the last SCB in the list. The lower byte of that word can then be used for any other function since the hardware will ignore it if the upper byte is zero.

2. Some of the sprite values (H size, V size, etc) are re-usable. The normal way to reuse them is to have the first sprite in the local list actually initialize the values, and then have the remaining sprites in the local list re-use them. One of the difficulties in doing it this way is that it is not always reasonable to arrange the list and the SCABs appropriately. One of the ways to simplify the problem is to use an initializing sprite with the desired numbers in the reusable registers.

I have been asked to provide an exemption that would allow the software to write directly to the hardware registers in the Suzy SCB and thus avoid all of the overhead of arranging the lists or creating null sprites.

Since this section of hardware is firmly tied to the sprite software process, I believe that it will be OK to write directly to the hardware registers. I could be wrong, so I am going to approve the following conditional exemption.

It will be OK to write to the twentyfour 16 bit registers of the Suzy SCB PROVIDING that you only do it via the MACRO PROVIDED TO YOU BY RJ MICAL. In addition, you must understand that the contents of this macro may change if future revisions of the hardware so require. In addition, you must understand that future hardware may make the process not work and therefore the macro will be changed to be a warning that you can't use it anymore.

Don't cheat.

18 Common Errors

There are errors that many first time programmers and some experienced programmers make with this hardware. Some of these errors may be difficult to identify due to the complexity of the ICs. I will list some of them here to aid in the debugging of 'Mystery Bugs'.

1. The presence of an interrupt in Mikey, regardless of the state of the CPU enable interrupt bit, will prevent the CPU from going to sleep, and thus prevent Suzy from functioning. So if sprites stop working, unintentional interrupt bits can be the hidden cause.

2. The Suzy Done Acknowledge address must be written to prior to running the sprite engine. This is required even prior to the first time the sprite engine is activated. If it is not written to in the appropriate sequences, the CPU will not go to

sleep when so requested. In addition, if some software accidentally allows a Suzy operation to complete without then following that completion with a write to SDONEACK, the CPU will not sleep. So if sprites stop working, something may have gone wrong with your SDONEACK software.

3. Writes to the cartridge are blind. If you accidentally access Suzy before the required delay period, you will modify some internal bus in Suzy. The results are not definable.

Handy Appendix 1, Block Diagram

8-Aug-89