Foenix F256jr Reference Manual

Peter Weingartner

November 7, 2022

Contents

List of Tables

1 Introduction

This manual is meant to be as complete as possible an introduction to the various hardware features of the F256jr. In it, I will attempt to explain each of the major subsystems of the F256jr and provide simple but practical examples of their use.

One thing this manual will not provide is a tutorial in programming the 65C02 processor at the heart of the F256jr. There are plenty of excellent books and videos explaining how the processor works and how to do assembly programming. While examples will generally be written in assembly, I will try to annotate them fully so that what is happening is very clear even to the novice assembly language coder.

2 F256jr Basics

2.1 Features

2.2 Ports

The connectors of the back of the F256jr from left to right are (see figure: 2.1):

Audio Line Out the stereo audio output. These are standard RCA style line level outputs.

SD Card Slot for standard SD cards for storage of files and programs.

DVI Monitor Port for output to your monitor. This can be connected to the DVI input of a monitor or run through a simple DVI-VGA connector to use with an older VGA input.

IEC Serial Port supports the Commodore serial bus. A Commodore disk drive (1541, 1571, 1581, *etc.*), a Commodore compatible serial printer, or other device supporting the Commodore serial bus can be connected here.

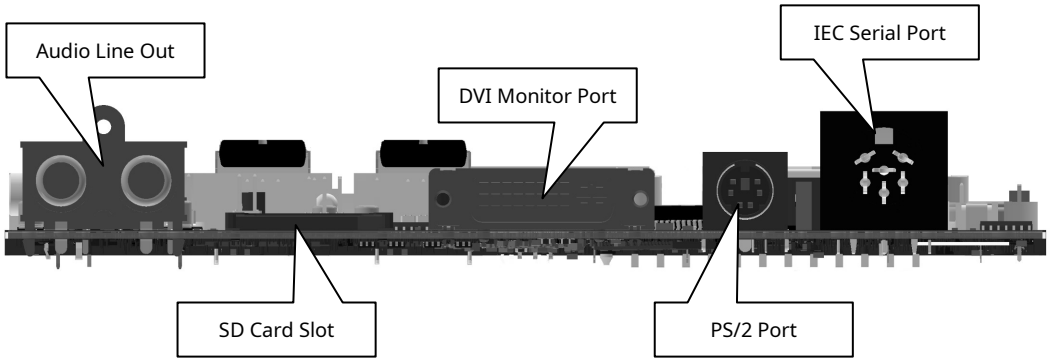

Figure 2.1: F256jr Rear Connectors

The top of the board has several connectors and other features that should be explained (see figure: 2.2):

Power In this is a standard ITX/ATX style power connector. Pretty much any PC power supply should work here, and a Pico-ATX style power adapter is more than sufficient.

Debug USB Port this provides access to the debug interface of the F256jr for a desktop computer. You can use it to upload data to the F256jr's memory or examine the memory. There is a Mini USB B connector on the board, but there is also a header that can be used to connect the USB jack on some cases to the board.

Case Buttons and LEDs this collection of headers is used to connect the power and reset button from the case as well as the power LED and SD access LED.

Joystick Ports these connectors can be used with a standard IDC to DB-9 adapter cable (such as was used by some PCs to provide RS-232 serial ports) to provide Atari style joystick connectors.

DIP Switches these switches allow you to manage certain aspects of the F256jr. In particular, you can control gamma correction and some boot options, depending on the kernel installed.

Stereo SIDs out of the box, these will be bare sockets, but they are where you would install your SID chips or SID emulators. The sockets support the original 6581, the lower voltage 8581, and the different replacements like the SwinSID, ARMSID, and BackSID.

Wi-Fi Module this optional module works with the built-in serial port to allow for Wi-Fi access, if a program or operating system supports it.

RS-232 Port this IDC header works with a standard IDC to DB-9 adapter cable to provide an RS-232 serial port. The same serial port is used for this port as is used by the Wi-Fi module, so only one of the two can be used at a time.

GPIO this header provides access to the I/O pins of the WDC65C22 VIA. The pin assignments are compatible with the Commodore C64 keyboard connector.

Expansion Port for future expansion. This is a PCI-E style connector with a custom pinout. In the future, it might be used for memory expansion or other devices.

Clock Battery this CR2032 cell holder provides power for the real time clock chip.

FPGA JTAG Port this connector is used to apply any future updates to the FPGA. A special adapter would need to be used to connect to this port.

Gamepad Ports this header provides access for an NES or SNES style gameport interface.

Case Audio Port this header provides access to the headphone and microphone signals to connect to a PC case.

Headphone Out this is a standard headphone adapter port that can be used if the case does not provide headphone output.

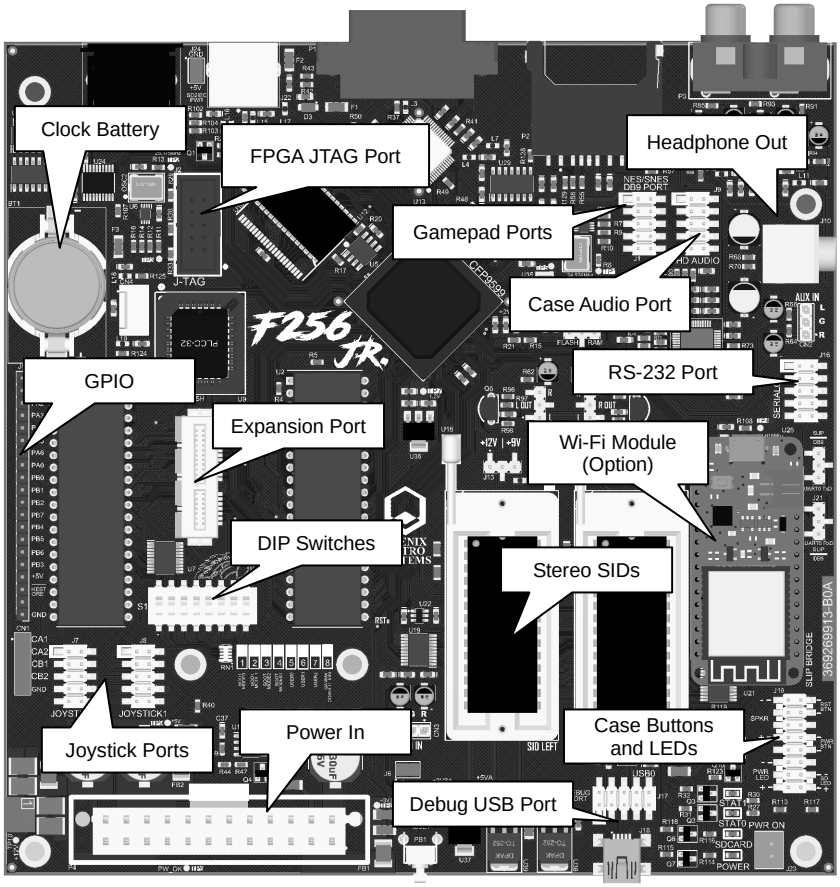

Figure 2.2: F256jr Top View

2.3 Set Up

2.4 DIP Switches

3 Memory Management

The F256jr has 256 KB of system RAM which can be used for programs, data, and graphics. It also has 512 KB of read-only flash memory that can be used by whatever operating system is installed. Now, the 65C02 CPU at the heart of the F256jr has an address space of only 64 KB, so how can it access all this memory, not to mention the I/O devices on the system? The answer is paging. The F256jr has a special memory management unit (MMU) that can swap banks of memory or I/O registers into and out of the memory space of the CPU.

To understand how it all works, we first need to look at how RAM and flash memory are handled by the F256jr. Because there are 768 KB of total storage on the system, the system has a 20-bit address bus to manage the memory. RAM and flash have address on that 20-bit bus as shown in table 3.1.

Start	End	Memory Type
0x00000	0x2FFFF	System RAM (256 KB)
0x30000	0x7FFFF	Reserved for future use (256 KB)
0x80000	0xFFFFF	Flash Memory (512 KB)

Table 3.1: C256jr memory layout

This memory is divided up into “banks” of 8 KB each. The 16-bit address space of the CPU is also divided up into 8 KB banks. The MMU allows the program to assign any bank of system memory to any bank of the CPU’s memory. It does this through the use of memory look-up tables (LUT), which provide the upper bits needed to select the bank out of system memory for any given bank in CPU memory. It takes 13 bits to specify an address within 8 KB, which means for a 16-bit address from the CPU, the upper 3 bits are the bank number. Since the system bus is 20 bits, a bank number there is 7 bits. So a LUT must provide a 7-bit system bank number for each 3-bit bank number provided by the CPU.

The F256jr’s MMU supports up to four LUTs, only one of which is active at any given moment. This allows programs to define four different memory layouts and switch between them quickly, without having to alter a LUT on the fly.

Bank	A[15..13]	Start	End
0	000	0x0000	0x1FFF
1	001	0x2000	0x3FFF
2	010	0x4000	0x5FFF
3	011	0x6000	0x7FFF
4	100	0x8000	0x9FFF
5	101	0xA000	0xBFFF
6	110	0xC000	0xDFFF
7	111	0xE000	0xFFFF

Table 3.2: CPU Memory Banks

Of the eight CPU memory banks, one is special. Bank 6 can be mapped to memory as the rest can, or it can be mapped to I/O registers, which are not memory mapped in the same way as RAM and flash. All I/O devices on the F256jr therefore live within 0xC000 through 0xDFFF on the CPU, but only if the MMU is set to map I/O to bank 6. There is quite a lot of I/O to access on the F256jr, so there are four different banks of I/O registers and memory that can be mapped to bank 6 (see table 3.3).

The MMU is controlled through two main registers, which are always at locations 0x0000 and 0x0001 in the CPU’s address space (see table 3.4). These registers allow programs to select an active LUT, edit a LUT, and control bank 6:

ACT_LUT these two bits specify which LUT (0 - 3) is used to translate CPU bus address to system bus addresses.

I/O Bank	Purpose
0	Low level I/O registers
1	Text display font memory and graphics color LUTs
2	Text display character matrix
3	Text display color matrix

Table 3.3: I/O Banks

Address	R/W	Name	7	6	5	4	3	2	1	0
0x0000	RW	MMU_MEM_CTRL	EDIT_EN	—	EDIT_LUT		—	—	ACT_LUT	
0x0001	RW	MMU_IO_CTRL	—					IO_DISABLE	IO_PAGE	

Table 3.4: MMU Registers

EDIT_EN if set (1), this bit allows a LUT to be edited by the program, and memory addresses 0x0010 - 0x0017 will be used by the LUT being edited. If clear (0), those memory locations will be standard memory locations and will be mapped like the rest of bank 0.

EDIT_LUT if EDIT_EN is set, these two bits will specify which LUT (0 - 3) is being editted and will appear in memory addresses 0x0010 - 0x0017.

IO_DISABLE if set (1), bank 6 is mapped like any other memory bank. If clear (0), bank 6 is mapped to I/O memory.

IO_PAGE if IO_DISABLE is clear, these two bits specify which bank of I/O memory (0 - 3) is mapped to bank 6.

3.0.1 Example: Setting up a LUT

In this example, we will set up LUT 1 so that the first six banks of CPU memory map to the first banks of RAM, bank 7 of CPU memory maps to the first bank of flash memory, and bank 6 maps to the first I/O bank.

```
    lda #$90      ; Active LUT = 0, Edit LUT#1
    sta $0000

    ldx #0        ; Start at bank 0
l1: txa           ; First 6 banks will just be the first banks of RAM
    sta $0010,x   ; Set the LUT mapping for this bank
    inx           ; Move to the next bank
    cpx #6        ; Until we get to bank 6
    bne l1

    lda #$40      ; Bank 7 maps to $80000, first bank of flash
    sta $0017

    stz $0001     ; Bank 6 should be I/O bank 0

    lda #$01      ; Turn off LUT editting, and switch to LUT#1
    sta $0000
```

4 The Text Screen

The display on the F256jr is managed by TinyVicky, which is the smaller member of the Vicky family of display controllers in the other Foenix machines. TinyVicky provides several display engines to let your programs control the screen:

- Text: an old school style text screen where the characters to display are stored in a text matrix, and the shape of those characters comes from font memory. Text mode characters are 8 pixels wide by 8 pixels high.
- Bitmap: a simple pixel graphics mode that can be either 300x240 or 300x200.
- Sprite: an engine to display small, movable sprites on the screen.
- Tile: an engine to display images on the screen made up of tiles from a tile set.

The bitmap, sprite, and tile engines are considered graphics modes. TinyVicky will let you display either text by itself, a mix of the graphics modes by themselves, or text overlayed on top of the graphics modes.

4.1 Text Matrix

The memory for the characters to display on the screen is the text matrix, which is stored in I/O page 2. When this I/O page is swapped into the CPU address space, it appears at 0xC000. Each byte of memory corresponds to a single character on the screen in left to right, top to bottom order. The byte at 0xC000 is the upper left corner of the screen, the byte at 0xC001 is the next character to the right, and so on. The number of bytes per line is set by the base resolution of the screen, but is generally 80. When a border is displayed, while that limits the number of characters displayed, the layout in memory remains the same.

The text screen has two core resolutions, tied to the refresh rate of the screen: 80 by 60 at 60 Hz, and 80 by 50 at 70 Hz. Beyond that, the character display my be made double width or double height, or both. This gives the following possible character displays: 80×60, 40×60, 80×30, 40×30, 80×50, 40×50, 80×25, and 40×25.

4.1.1 Example: Print an A to the Screen

```
lda $0001       ; Save the current MMU setting
pha

lda #$02        ; Swap I/O Page 2 into bank 6
sta $0001

lda #'A'        ; Write 'A' to the upper left corner
sta $C000

pla             ; Restore the old MMU setting
sta $0001
```

Note: this example does not set the font or the color, so depending on how your F256jr is initialized, you may not see an actual "A" on the screen.

4.2 Text Color LUTs

Characters in TinyVicky text mode have two colors: the foreground and the background. The foreground and background colors are picked for each character out of two different palettes of 16 colors each. The colors in the palettes are picked from

the full range of colors F256jr can produce, which is more than 16 million colors. This is all managed through two color lookup tables (LUTs) provided by TinyVicky: a text foreground color LUT, and a text background color LUT.

The text LUTs are stored in I/O page 0. The foreground LUT starts at 0xD800, and the background LUT starts at 0xD840.

Each LUT is a list of 16 entries. Each entry is a set of four bytes: blue, green, red, and alpha. Each byte indicates how much of that primary color is present as a component of the actual color. The values range from 0 (none) to 255 (as much as possible). Currently, the alpha channel is not used and is there for future expansion.

Index	R/W	Foreground	Background	0	1	2	3
0	W	0xD800	0xD840	BLUE_0	GREEN_0	RED_0	X
1	W	0xD804	0xD844	BLUE_1	GREEN_1	RED_1	X
2	W	0xD808	0xD848	BLUE_2	GREEN_2	RED_2	X
3	W	0xD80C	0xD84C	BLUE_3	GREEN_3	RED_3	X
4	W	0xD810	0xD850	BLUE_4	GREEN_4	RED_4	X
5	W	0xD814	0xD854	BLUE_5	GREEN_5	RED_5	X
6	W	0xD818	0xD858	BLUE_6	GREEN_6	RED_6	X
7	W	0xD81C	0xD85C	BLUE_7	GREEN_7	RED_7	X
8	W	0xD820	0xD860	BLUE_8	GREEN_8	RED_8	X
9	W	0xD824	0xD864	BLUE_9	GREEN_9	RED_9	X
10	W	0xD828	0xD868	BLUE_10	GREEN_10	RED_10	X
11	W	0xD82C	0xD86C	BLUE_11	GREEN_11	RED_11	X
12	W	0xD830	0xD870	BLUE_12	GREEN_12	RED_12	X
13	W	0xD834	0xD874	BLUE_13	GREEN_13	RED_13	X
14	W	0xD838	0xD878	BLUE_14	GREEN_14	RED_14	X
15	W	0xD83C	0xD87C	BLUE_15	GREEN_15	RED_15	X

Table 4.1: Text Color Lookup Tables

4.3 Color Matrix

The way that text color is selected for each character is through the color matrix. This section of memory is in I/O page 3 and starts at 0xC000 when page 3 is swapped into the CPU's address space. The layout is precisely the same as the text matrix (e.g. the character at 0xC123 in the text matrix has its color information at 0xC123 in the color matrix).

Each byte in the color matrix specifies two colors by providing an index into each of the two text LUTs. The most significant four bits is the number of the foreground color to use. The number of the least significant four bits is the number of the background color to use.

Let's say the color value at 0xC123 is 0x45. This means that the foreground color of the character is color 4 from the text foreground LUT, which starts at 0xD810 (0xD800 + 4 * 4), and the background color of the character is 5 from the text background LUT, which starts at 0xD854 (0xD840 + 4 * 5). If the bytes at 0xD810 are 0x00, 0x80, 0x80, that means the foreground will be a medium yellow. If the bytes at 0xD854 are 0xFF, 0x00, 0x00, that means the background will be blue.

4.3.1 Example: Make That "A" Yellow on Blue

```
lda $0001       ; Save the MMU state
pha

stz $0001       ; Switch in I/O Page #0

stz $D810       ; Set foreground #4 to medium yellow
lda #$80
sta $D811
sta $D812

lda #$FF        ; Set background #5 to blue
sta $D854
stz $D855
stz $D856

lda #$03        ; Switch to I/O page #3 (color matrix)
sta $0001

lda #$45        ; Color will be foreground=4, background=5
```

```
    sta $C000

    pla             ; Restore the MMU state
    sta $0001
```

4.4 Entering Text Mode

Whether or not text mode is being displayed (and in what resolution) is controlled by the VICKY Master Control Registers (see table 4.2). For now, we're going to ignore most of the bits, which are used by other display modes. For text mode, we really only care about the TEXT bit, which needs to be set to turn on the text display. The resolution is controlled by DBL_Y, DBL_X, and CLK_70. If we set 0xD000 to 0x01 and 0xD001 to 0x00, that will put us into text mode at 80×60.

Address	R/W	7	6	5	4	3	2	1	0
0xD000	R/W	X	GAMMA	SPRITE	TILE	BITMAP	GRAPH	OVRLY	TEXT
0xD001	R/W	X					DBL_Y	DBL_X	CLK_70

Table 4.2: VICKY Master Control Registers

TEXT if set (1), text mode display is enabled

OVRLY if set, text will be overlayed on graphics

GRAPH if set, one or more of the graphics modes may be used

BITMAP if set (and GRAPHICS is set), bitmap graphics may be displayed

TILE if set (and GRAPHICS is set), tile graphics may be displayed

SPRITE if set (and GRAPHICS is set), sprite graphics may be displayed

GAMMA if set, gamma correction is enabled

CLK_70 if set, the video refresh will be set to 70 Hz mode (640x400 text resolution, 320x200 graphics). If clear, the video refresh will be set to 60 Hz (640x480 text resolution, 320x240 graphics).

DBL_X if set, text mode characters will be twice as wide (320 pixels)

DBL_Y if set, text mode characters will be twice as high (240 or 200 pixels, depending on CLK_70)

4.5 Text Fonts

Character shapes (or "glyphs," if you prefer) are defined in font memory, which is in I/O page 1 and starts at 0xC000. The F256jr treats each character as a square of pixels, 8 pixels on a side. A pixel may be either in the foreground color for the character or in the background color for the character. The way this is managed is that each character has a sequence of eight bytes in the font memory. Each byte represents a row in the character, and each bit represents a pixel in the row (■ for foreground, □ for background).

As an example, let's say we wanted to have a fancy "F" for character 0:

□	□	□	■	■	■	■	■	0x1F
□	□	■	■	□	□	□	□	0x30
□	□	■	■	□	□	□	□	0x30
□	■	■	■	■	■	□	□	0x7C
□	■	■	□	□	□	□	□	0x60
■	■	□	□	□	□	□	□	0xC0
■	■	□	□	□	□	□	□	0xC0

Table 4.3: A sample character

The glyph to display would be defined by the eight byte sequence 0x1F, 0x30, 0x30, 0x7C, 0x60, 0xC0, 0xC0. We would store that sequence in I/O page 0, starting at 0xC000 (0x1F), through 0xC007 (0xC0). After that was set, any time the byte 0x00 is written to screen memory, the glyph "F" would be displayed in that position.

5 Graphics

The F256jr provides three separate graphics engines, providing programs with a choice in how they display information to the user. Those different engines do share certain features, however, and this chapter will cover the common elements. The three graphics engines are bitmaps, tile maps, and sprites. What is common between all these elements is how they determine what colors to display and how to determine, when two or more objects are in the same place, which object is displayed.

- Bitmaps are simple raster images. They are the size of the screen (320 × 200 or 320 × 240) and cannot be moved. The TinyVicky chip used by the F256jr allows for three separate bitmaps to be displayed at the same time.
- Tile maps are images made up of tiles. The tiles come in a tile set, which is a raster image like a bitmap but provides 256 tiles. The tile map itself creates its image by indicating which tile is displayed at every position in the tile map. This mapping can be changed on the fly, allowing tile maps to be altered, and tile maps can also be scrolled horizontally and vertically to a limited degree. This allows for possibility for smooth scrolling of a tile map scene. TinyVicky allows for three separate tile maps to be displayed simultaneously.
- Sprites are small, square graphics elements that may be moved to any position on the screen. Sprites are typically used to represent game characters or very mobile UI elements. TinyVicky sprites may be 8, 16, 24, or 32 pixels on a side. There may be as many as 64 sprites active on the screen at once (without using special techniques).

5.1 Graphics Colors

The graphics modes use a color lookup system similar to text mode to determine colors. The pixel data for a tile, bitmap, or sprite is composed of bytes, where each byte specifies the color of that pixel. The byte serves as an index into a color lookup table where the red, green, and blue components of the desired color are stored (see figure: 5.1). As with text, the color components are bytes and specify an intensity from 0 (none of that primary color) to 255 (as much of that primary color as possible). Also, as with text, there is a fourth byte that is reserved for future use, meaning that each color takes up four bytes in the CLUT. In short, the byte order of a graphics CLUT entry is exactly the same as for a text CLUT.

However, there is a key difference from text mode. In text mode, there are two colors (foreground and background), and each color is one out of sixteen possibilities. With graphics modes, there are 256 possibilities. So a CLUT with only 16 entries will not work. There are therefore separate CLUTs for graphics. TinyVicky provides for four separate graphics CLUTs with 256 entries. Each graphic object on the screen specifies which graphics CLUT it will use for its colors. These CLUTs may be found in I/O page 0 (see table: 5.1).

Address	R/W	Purpose
0xD000	R/W	Graphics CLUT 0
0xD400	R/W	Graphics CLUT 1
0xD800	R/W	Graphics CLUT 2
0xDC00	R/W	Graphics CLUT 3

Table 5.1: Graphics Color Lookup Tables

5.1.1 Example: A Simple Gradient

Let's set up a CLUT so that we have the colors for a gradient fill between red and blue. In this example, `pointer` is a two byte variable down in zero page, which will be used to point to the first byte of the CLUT entry the code is updating. The `Y` register is being used to point to the individual components of the entry.

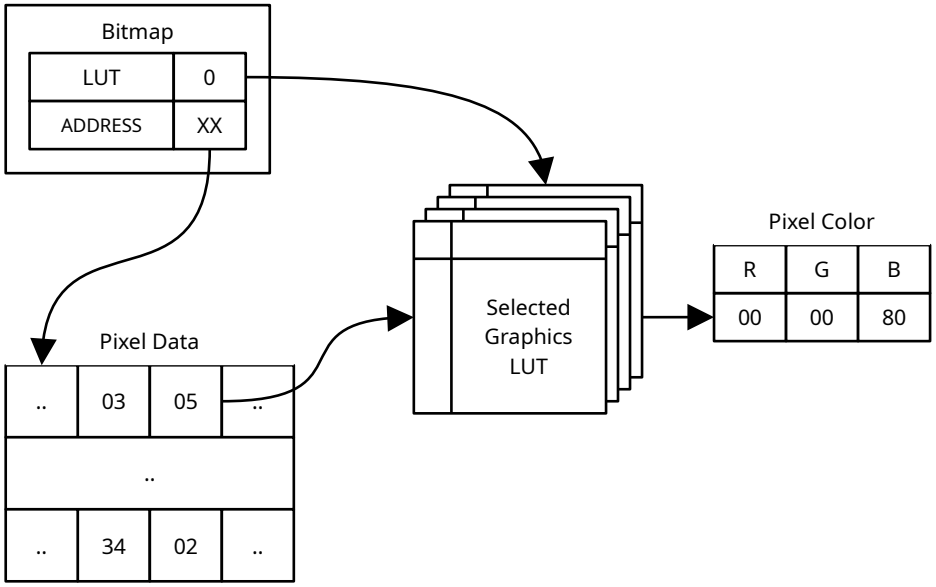

Figure 5.1: Bitmap Data to Pixels

```
MMU_IO_CTRL = $0001             ; MMU I/O Control Register
VKY_GR_CLUT_0 = $D000           ; Graphics LUT #0

;
; Initialize the LUT to greyscale from (255, 0, 0) to (0, 0, 255)
;

            lda #$01            ; Set the I/O page to #1
            sta MMU_IO_CTRL

            lda #<VKY_GR_CLUT_0 ; pointer will be used to point to a particular LUT entry
            sta pointer
            lda #>VKY_GR_CLUT_0
            sta pointer+1

            ldx #0              ; Start with blue = 0

lut_loop:   ldy #0              ; And start at the offset for blue
            txa                 ; Take the current blue color level
            sta (pointer),y     ; Set the blue component
            iny

            lda #0
            sta (pointer),y     ; Set the green component to 0
            iny

            txa                 ; Get the blue component again
            eor #$ff            ; And compute the 2's complement of it
            inc a
            sta (pointer),y     ; Set the red component
            iny

            inx                 ; Go to the next color
            beq lut_done        ; If we are back to black, we're done with the LUT

            clc                 ; Move pointer to the next LUT entry (+ 4)
            lda pointer
            adc #4
            sta pointer
            lda pointer+1
            adc #0
```

```
            sta pointer+1

            bra lut_loop

lut_done:
```

5.2 Pixel Data

All three graphics engines arrange their pixel data in the same manner. They all use rectangular raster images as a base, although the width and height of the rectangle can vary. The pixels are placed in memory in sequential order in left-to-right and top-to-bottom order. That is, the first pixel in the sequence is the upper-left pixel in the image. The next pixel is the pixel to the immediate right and so on. If the image is $w \times h$, the position of a pixel at (x, y) in the list is $y \times w + x$.

5.3 Graphics Layers

Now, what happens if two sprites take up the same position or if a program displays a tile map and a bitmap together? How does TinyVicky determine what color to display at a given position? TinyVicky provides a flexible layering system with several layers. Elements in "near" layers (lower numbers) get displayed on top of elements in "far" layers (higher numbers). If a sprite in layer 0 says a pixel should be blue while a tile in layer 1 says it should be red, the pixel will be blue. Color 0, however, is special. It is always the transparent "color". A pixel that is 0 in an element will be the color of whatever is behind it (or the global background color, if there is nothing behind it).

TinyVicky provides for seven layers, but they are split up a bit. Three of the layers are for bitmaps and tile maps. Only one bitmap or tile map can be placed in any of those three layers. The other four layers are for sprites only. Any sprite can be assigned to any of the sprite layers, and there can be multiple sprites in a layer. The sprite layers are interleaved with the bitmap and tile map layers (see figure: 5.2).

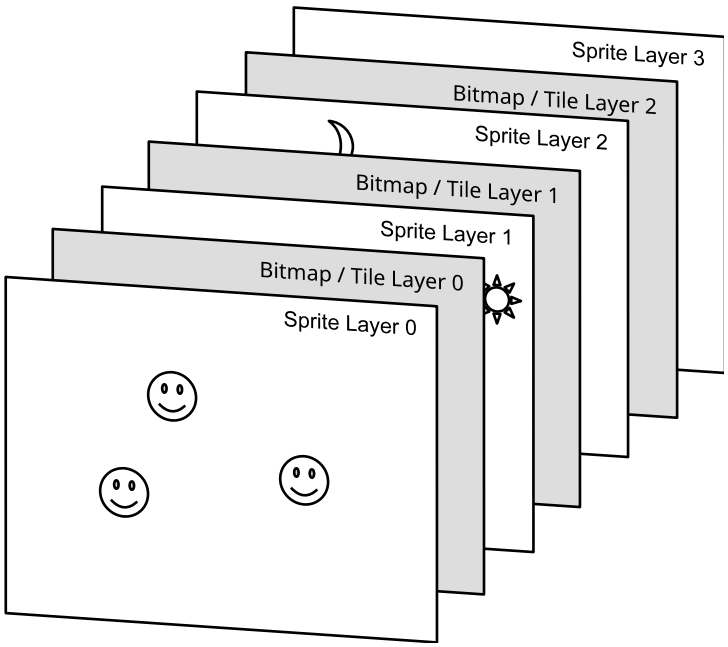

Figure 5.2: TinyVicky Graphic Layers

Bitmaps and tile maps are assigned to their layers using the layer control registers (see table: 5.2). The three fields LAYER0, LAYER1, and LAYER2 in the layer registers are three bit values, which indicate which graphical element to assign to that layer (see table: 5.3).

Address	R/W	7	6	5	4	3	2	1	0
0xD002	R/W	—	LAYER1			—	LAYER0		
0xD003	R/W	—					LAYER2		

Table 5.2: Bitmap and Tile Map Layer Registers

Code	Layer
0	Bitmap Layer 0
1	Bitmap Layer 1
2	Bitmap Layer 2
4	Tile Map Layer 0
5	Tile Map Layer 1
6	Tile Map Layer 2

Table 5.3: Bitmap and Tile Map Layer Codes

5.3.1 Example: Put Bitmap 0 on Layer 0

As an example of how to use layers, we can set things up for future examples by putting bitmap 0 in the front layer (0), tile map 0 in the next layer (1), and bitmap 1 in the back layer (2).

```
lda #$20            ; Layer 0 = BM 0, Layer 1 = TM 0
sta VKY_LAYER_CTRL_0
lda #$01            ; Layer 2 = BM 1
sta VKY_LAYER_CTRL_1
```

5.4 Bitmaps

TinyVicky allows for three full screen bitmaps to be displayed at once. These bitmaps are either 320 × 200 or 320 × 240, depending on the value of the CLK_70 bit of the master control register. A bitmap's pixel data contains either 64,000 bytes, or 76,800 bytes of data. In both cases, the pixel data is arranged from left to right and top to bottom. The first 320 bytes are the pixels of the first line (with the first pixel being the left-most). The second 320 bytes are the second line, and so on. Additionally, the bitmaps can independently use any of the four graphics CLUTs to specify the colors for those indexes. TinyVicky provides registers for each bitmap set the CLUT and the address of the bitmap:

Address	R/W	Bitmap	7	6	5	4	3	2	1	0
0xD100	R/W	0	—					CLUT		ENABLE
0xD101	R/W		AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD102	R/W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD103	R/W		—						AD17	AD16
0xD108	R/W	1	—					CLUT		ENABLE
0xD109	R/W		AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD10A	R/W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD10B	R/W		—						AD17	AD16
0xD110	R/W	2	—					CLUT		ENABLE
0xD111	R/W		AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD112	R/W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD113	R/W		—						AD17	AD16

Table 5.4: Bitmap Registers

ENABLE if set and both graphics and bitmaps are enabled in the Vicky Master Control Register (see table 4.2), then this bitmap will be displayed.

CLUT sets the graphics color lookup table to be used for this bitmap

AD give the address of the first byte of the pixel data within the 256 KB system RAM. Note that this address is relative to the system bus of 20 bits and is not based on the CPU's addressing.

To set up and display a bitmap, the following things need to be done. The order is not terribly important, although updates to the bitmap's pixel data after the bitmap is displaying will be visible. That could be desirable, depending on what the program is doing.

1. Enable bitmap graphics in the TinyVicky Master Control Register (see table: 4.2). This means you need to set both the GRAPH and BITMAP bits and either clear TEXT or set the OVRLY to display text and bitmap together.
2. Set up the pixel data for the bitmap somewhere in the first 256 KB of RAM.
3. Set the address of the bitmap's pixel data in the AD field.

4. Assign the bitmap to a layer using the layer control registers (see table: 5.2).
5. Set the bitmap's CLUT and ENABLE bit in its control register.

5.4.1 Example: Display a Bitmap

This example will build on the previous examples of setting up the CLUT and display a gradient on the screen. First, it needs to turn on the bitmap graphics:

```
MMU_MEM_CTRL = $0000            ; MMU Memory Control Register
MMU_IO_CTRL = $0001             ; MMU I/O Control Register
VKY_MSTR_CTRL_0 = $D000         ; Vicky Master Control Register 0
VKY_MSTR_CTRL_1 = $D001         ; Vicky Master Control Register 1
VKY_BM0_CTRL = $D100            ; Bitmap #0 Control Register
VKY_BM0_ADDR_L = $D101          ; Bitmap #0 Address bits 7..0
VKY_BM0_ADDR_M = $D102          ; Bitmap #0 Address bits 15..8
VKY_BM0_ADDR_H = $D103          ; Bitmap #0 Address bits 17..16

bitmap_base = $10000            ; The base address of our bitmap

stz MMU_IO_CTRL     ; Go back to I/O page #0

lda #$0C            ; enable GRAPHICS and BITMAP. Disable TEXT
sta VKY_MSTR_CTRL_0 ; Save that to VICKY master control register 0
stz VKY_MSTR_CTRL_1 ; Make sure we're just in 320x240 mode (VICKY master control register 1)
```

Next, it needs to set up the bitmap: setting the address, CLUT, and enabling the bitmap:

```
;
; Turn on bitmap #0
;

stz VKY_BM1_CTRL    ; Make sure bitmap 1 is turned off

lda #$01            ; Use graphics LUT #0, and enable bitmap
sta VKY_BM0_CTRL

lda #<bitmap_base   ; Set the low byte of the bitmap's address
sta VKY_BM0_ADDR_L
lda #>bitmap_base   ; Set the middle byte of the bitmap's address
sta VKY_BM0_ADDR_M
lda #`bitmap_base   ; Set the upper two bits of the bitmap's address
and #$03
sta VKY_BM0_ADDR_H
```

Now, the code needs to create the pixel data for the gradient in memory. This is a bit tricky on the F256jr, because the program is using the larger 320 × 240 screen, which requires more than 64 KB of memory. In order to write to the entire bitmap, the program will have to work with the MMU to switch memory banks to access the whole bitmap. The program will use bank 1 (0x2000 – 0x3FFF) as its window into the bitmap, which will start at 0x10000. It will walk through the memory byte-by-byte, setting each pixel's color based on what line it is on (tracked in a `line` variable). Once it has written a bank's worth of pixels (8 KB), it will increment the bank number and update the MMU register. Once it has written 240 lines, it will finish.

NOTE: in the following code, `bm_bank` and `line` are byte variables, and `pointer` and `column` are two-byte variables in zero page (although really only `pointer` has to be there).

```
            ; Set the line number to 0
            stz line

            ; Calculate the bank number for the bitmap
            lda #(bitmap_base >> 13)
            sta bm_bank

bank_loop:  stz pointer         ; Set the pointer to start of the current bank
            lda #$20
            sta pointer+1
```

```
            ; Set the column to 0
            stz column
            stz column+1

            ; Alter the LUT entries for $2000 -> $bfff

            lda #$80            ; Turn on editing of MMU LUT #0, and work off #0
            sta MMU_MEM_CTRL

            lda bm_bank
            sta MMU_MEM_BANK_1  ; Set the bank we will map to $2000 - $3fff

            stz MMU_MEM_CTRL    ; Turn off editing of MMU LUT #0

            ; Fill the line with the color..

loop2:      lda line            ; The line number is the color of the line
            sta (pointer)

inc_column: inc column          ; Increment the column number
            bne chk_col
            inc column+1

chk_col:    lda column          ; Check to see if we have finished the row
            cmp #<320
            bne inc_point
            lda column+1
            cmp #>320
            bne inc_point

            lda line            ; If so, increment the line number
            inc a
            sta line
            cmp #240            ; If line = 240, we're done
            beq done

            stz column          ; Set the column to 0
            stz column+1

inc_point:  inc pointer         ; Increment pointer
            bne loop2           ; If < $4000, keep looping
            inc pointer+1
            lda pointer+1
            cmp #$40
            bne loop2

            inc bm_bank         ; Move to the next bank
            bra bank_loop       ; And start filling it

done:       nop                 ; Lock up here
            bra done
```

6 Sprites

In addition to bitmaps and tiles, the F256jr provides support for sprites, which are mobile graphical objects that can appear anywhere on the screen. F256jr sprites are similar to the sprites on the Commodore 64 or player-missile graphics on the 8-bit Atari computers, but they are more flexible than either of those. A sprite is essentially a little bitmap that can be positioned anywhere on the screen. Each one can come in one of four sizes: 8×8, 16×16, 24×24, or 32×32. Each one can display up to 256 colors, picked from one of the four graphics color lookup tables.

A program for the F256jr can use up to 64 sprites, each one of which is controlled by a block of sprite control registers. The sprite control registers are in I/O page 0, and start at 0xD900. Each sprite takes up 8 bytes, so sprite 0 starts at 0xD900, sprite 1 starts at 0xD908, sprite 2 at 0xD910, and so on. The registers for each sprite are arranged within that block of 8 bytes as shown in table 6.1.

Offset	R/W	Name	7	6	5	4	3	2	1	0
0	W	Sprite Control	—	SIZE		LAYER		LUT		ENABLE
1	W	Sprite Address	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
2	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
3	W		—						AD17	AD16
4	W	Sprite X	X7	X6	X5	X4	X3	X2	X1	X0
5	W		X15	X14	X13	X12	X11	X10	X9	X8
6	W	Sprite Y	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
7	W		Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8

Table 6.1: Sprite Registers for a Single Sprite

These registers manage seven fields:

ENABLE if set, this particular sprite will be displayed (assuming the graphics and sprite engines are enabled in the Vicky Master Control Register).

LUT selects the graphics color lookup table to use in assigning colors to pixels

LAYER selects which sprite layer the sprite will be displayed on

SIZE selects the size of the sprite (see table 6.2)

AD the address of the bitmap (must be within the first 256 KB of RAM). The address is based on the 24-bit system bus, not the CPU's address space.

X the X coordinate where the sprite will be displayed (corresponds to the sprite's upper-left corner)

Y the Y coordinate where the sprite will be displayed (corresponds to the sprite's upper-left corner)

SIZE		Meaning
0	0	32×32
0	1	24×24
1	0	16×16
1	1	8×8

Table 6.2: Sprite Sizes

6.1 Sprite, Layers, and Display Priority

While a sprite can be assigned to any of four layers, this layer is only used for determining how the sprite interacts with bitmap or tile map graphics and not how sprites layer with each other. When sprites "collide," a built-in sprite priority order is used to determine which sprite determines a pixel's color. When two sprites are both trying to set the color of a pixel on the screen, the sprite with the lowest number is the one that determines the color. For example, if sprite 0 and sprite 5 are in the same location, it is sprite 0 that will display in the foreground. The sprite layers *cannot* be used to change this.

The best practice for assigning sprites is to place the images that need to be on top in the first sprites and those that need to be in the back in the higher numbered sprites. Use the LAYER field for the sprites to control how the sprites layer with the tile maps and bitmaps.

6.1.1 Example: Displaying a Sprite

In this example, we'll just put a ball on the screen. First, the program needs to set up TinyVicky to be in sprite mode with no border and a light purple background:

```
MMU_IO_CTRL = $0001             ; MMU I/O Control Register

VKY_MSTR_CTRL_0 = $D000         ; Vicky Master Control Register 0
VKY_MSTR_CTRL_1 = $D001         ; Vicky Master Control Register 1
VKY_BRDR_CTRL = $D004           ; Vicky Border Control Register
VKY_BKG_COL_B = $D00D           ; Vicky Graphics Background Color Blue Component
VKY_BKG_COL_G = $D00E           ; Vicky Graphics Background Color Green Component
VKY_BKG_COL_R = $D00F           ; Vicky Graphics Background Color Red Component

VKY_SP0_CTRL = $D900            ; Sprite #0's control register
VKY_SP0_AD_L = $D901            ; Sprite #0's pixel data address register
VKY_SP0_AD_M = $D902
VKY_SP0_AD_H = $D903
VKY_SP0_POS_X_L = $D904         ; Sprite #0's X position register
VKY_SP0_POS_X_H = $D905
VKY_SP0_POS_Y_L = $D906         ; Sprite #0's Y position register
VKY_SP0_POS_Y_H = $D907
```

```
            ;
            ; Set up TinyVicky to display sprites
            ;
            lda #$24                    ; Graphics and Sprite engines enabled
            sta VKY_MSTR_CTRL_0
            stz VKY_MSTR_CTRL_1         ; 320x240 @ 60Hz

            stz VKY_BRDR_CTRL           ; No border

            lda #$96                    ; Background: lavender
            sta VKY_BKG_COL_R
            lda #$7B
            sta VKY_BKG_COL_G
            lda #$B6
            sta VKY_BKG_COL_B
```

Next, the program loads the sprite's colors into the CLUT (`ptr_src` and `ptr_dst` are 16-bit storage locations in zero page and are used as pointers):

```
            ;
            ; Load the sprite LUT into memory
            ;

            lda #$01                    ; Switch to I/O Page #1
            sta MMU_IO_CTRL

            lda #<balls_clut_start      ; Set the source pointer to the palette data
            sta ptr_src
            lda #>balls_clut_start
            sta ptr_src+1
```

```
            lda #<VKY_GR_CLUT_0         ; Set the destination pointer to Graphics CLUT 1
            sta ptr_dst
            lda #>VKY_GR_CLUT_0
            sta ptr_dst+1

            ldx #0                      ; X is a counter for the number of colors copied
color_loop: ldy #0                      ; Y is a pointer to the component within a CLUT color
comp_loop:  lda (ptr_src),y             ; Read a byte from the code
            sta (ptr_dst),y             ; And write it to the CLUT
            iny                         ; Move to the next byte
            cpy #4
            bne comp_loop               ; Continue until we have copied 4 bytes

            inx                         ; Move to the next color
            cmp #16
            beq done_lut                ; Until we have copied all 16

            clc                         ; Advance ptr_src to the next source color entry
            lda ptr_src
            adc #4
            sta ptr_src
            lda ptr_src+1
            adc #0
            sta ptr_src+1

            clc                         ; Advance ptr_dst to the next destination color entry
            lda ptr_dst
            adc #4
            sta ptr_dst
            lda ptr_dst+1
            adc #0
            sta ptr_dst+1

            bra color_loop              ; And start copying that new color

done_lut:   stz MMU_IO_CTRL             ; Go back to I/O Page 0
```

Finally, we point sprite 0 to the pixel data (which is included in the assembly code below), set its location on the screen (which will be the upper left corner of the screen), and then we turn on the sprite setting its LUT and LAYER in the process:

```
            ;
            ; Set up sprite #0
            ;
init_sp0:   lda #<balls_img_start       ; Address = balls_img_start
            sta VKY_SP0_AD_L
            lda #>balls_img_start
            sta VKY_SP0_AD_M
            stz VKY_SP0_AD_H

            lda #32
            sta VKY_SP0_POS_X_L         ; (x, y) = (32, 32)... should be upper-left corner of the screen
            stz VKY_SP0_POS_X_H

            lda #32
            sta VKY_SP0_POS_Y_L
            stz VKY_SP0_POS_Y_H

            lda #$41                    ; Size = 16x16, Layer = 0, LUT = 0, Enabled
            sta VKY_SP0_CTRL
```

Here is the pixel data for the sprite:

```
balls_img_start:
    .byte $00, $00, $00, $00, $00, $00, $03, $02, $02, $01, $00, $00, $00, $00, $00, $00
    .byte $00, $00, $00, $00, $05, $05, $04, $03, $03, $03, $03, $02, $00, $00, $00, $00
```

```
    .byte $00, $00, $00, $07, $07, $07, $06, $05, $04, $04, $03, $03, $01, $00, $00, $00
    .byte $00, $00, $07, $09, $0A, $0B, $0A, $08, $06, $05, $04, $03, $02, $01, $00, $00
    .byte $00, $05, $07, $0A, $0D, $0E, $0D, $0A, $07, $05, $05, $04, $03, $01, $01, $00
    .byte $00, $05, $07, $0B, $0E, $0E, $0E, $0C, $07, $05, $05, $04, $03, $01, $01, $00
    .byte $03, $04, $06, $0A, $0D, $0E, $0D, $0A, $07, $05, $05, $04, $03, $02, $01, $01
    .byte $02, $03, $05, $08, $0A, $0C, $0A, $08, $06, $05, $05, $04, $03, $02, $01, $01
    .byte $02, $03, $04, $06, $07, $07, $07, $06, $05, $05, $05, $04, $03, $01, $01, $01
    .byte $01, $03, $04, $05, $05, $05, $05, $05, $05, $05, $05, $03, $03, $01, $01, $01
    .byte $00, $03, $03, $04, $05, $05, $05, $05, $05, $05, $04, $03, $02, $01, $01, $00
    .byte $00, $02, $03, $03, $04, $04, $04, $04, $04, $03, $03, $02, $01, $01, $01, $00
    .byte $00, $00, $01, $02, $03, $03, $03, $03, $03, $03, $02, $01, $01, $01, $00, $00
    .byte $00, $00, $00, $01, $01, $01, $02, $02, $01, $01, $01, $01, $01, $00, $00, $00
    .byte $00, $00, $00, $00, $01, $01, $01, $01, $01, $01, $01, $01, $00, $00, $00, $00
    .byte $00, $00, $00, $00, $00, $00, $01, $01, $01, $01, $00, $00, $00, $00, $00, $00
```

Here are the colors for the sprite (note that this example is using only 15 colors, to make the example more understandable in print):

```
balls_clut_start:
    .byte $00, $00, $00, $00
    .byte $88, $00, $00, $00
    .byte $7C, $18, $00, $00
    .byte $9C, $20, $1C, $00
    .byte $90, $38, $1C, $00
    .byte $B0, $40, $38, $00
    .byte $A8, $54, $38, $00
    .byte $C0, $5C, $50, $00
    .byte $BC, $70, $50, $00
    .byte $D0, $74, $68, $00
    .byte $CC, $88, $68, $00
    .byte $E0, $8C, $7C, $00
    .byte $DC, $9C, $7C, $00
    .byte $EC, $A4, $90, $00
    .byte $EC, $B4, $90, $00
```

7 Tiles

The third graphics engine TinyVicky provides is the tile map system. The tile map system might seem a bit confusing at first, but really it is very similar to text mode, just made more flexible. In text mode, we have characters (256 of them). The shapes of the characters are defined in the font. What character is shown in a particular spot on the screen is set in the text matrix, which is a rectangular array of bytes in memory. In the same way, with the tile system we have tiles (256 of those, too). What those tiles look like are defined in a "tile set." What tile is shown in a particular spot on the screen is set in the "tile map." So there is an analogy:

character	$\approx$	tile
font	$\approx$	tile set
text matrix	$\approx$	tile map

There are several differences with tile maps, however:

- A tile map may use tiles that are either 8×8 pixels or 16×16 pixels.
- A tile map can be scrolled smoothly horizontally or vertically.
- A tile may use 256 colors in its pixels as opposed to text mode's two-color characters. In particular, this means that a tile set uses one byte per pixel, with that byte's value being an index into a CLUT (as with bitmaps and sprites), where text mode fonts are one *bit* per pixel choosing between a foreground and background color.
- The tile map system allows for up to eight different tile sets to be used at the same time, where text mode has a single font.
- Up to three different tile maps can be displayed at one time, where text mode can only display one text matrix.
- A tile map can be placed on any one of three display layers, where text mode is always on top.

7.1 Tile Maps

There are three tile maps supported by TinyVicky, each of which has 13 bytes worth of registers (see table: 7.1). Tile map 0 starts at 0xD200. Tile map 1 starts at 0xD20C. Tile map 2 starts at 0xD218.

TILE_SIZE if 0, tiles are 8 pixels wide by 8 tall. If 1, tiles are 16 pixels wide by 16 pixels tall

ENABLE if set, the tile map will be displayed (if GRAPH and TILES are set in TinyVicky's Master Control Register)

AD the address of the tile map data in RAM

MAP_SIZE_X the width of the tile map in tiles (*i.e.* the number of columns)

MAP_SIZE_Y the height of the tile map in tiles (*i.e.* the number of rows)

X horizontal scroll in tile widths

SSX horizontal scroll in pixels. How these bits are used varies with the size. If tiles are 16 pixels wide, then flags SSX[3..0] are used. If tiles are only 8 pixels wide, then only SSX[3..1] are used.

DIR_X the direction of the horizontal scroll.

Offset	R/W	7	6	5	4	3	2	1	0
0	W	—			TILE_SIZE	—			ENABLE
1	W	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
2	W	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
3	W	—						AD17	AD16
4	W	MAP_SIZE_X							
5	W	RESERVED							
6	W	MAP_SIZE_Y							
7	W	RESERVED							
8	W	X3	X2	X1	X0	SSX3	SSX2	SSX1	SSX0
9	W	DIR_X	—	X9	X8	X7	X6	X5	X4
10	W	Y3	Y2	Y1	Y0	SSY3	SSY2	SSY1	SSY0
12	W	DIR_Y	—			Y7	Y6	Y5	Y4

Table 7.1: Tile Map Registers

Y vertical scroll in tile heights

SSY vertical scroll in pixels. How these bits are used varies with the size. If tiles are 16 pixels wide, then flags SSY[3..0] are used. If tiles are only 8 pixels wide, then only SSY[3..1] are used.

DIR_Y the direction of the vertical scroll.

One way tile maps get their flexibility is that, where text mode uses 8-bit bytes for the text matrix, a tile map is actually a rectangular collection of 16-bit integers in memory. A tile map entry is divided up into two pieces: the first byte is the number of the tile to display in that position (much like the character code in text mode), but the upper byte contains attribute bits (see table: 7.2), which have two fields:

SET is the number of the tile set to use for this tile's appearance

CLUT is the number of the graphics CLUT to use in setting the colors

This attribute system makes tiles very powerful. Effectively, a single tile map can display 1,024 completely unique shapes at one time by using all eight tile sets. Also, since the CLUT is set for each tile in the attributes, the number of tiles needed can be reduced by clever use of recoloring.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—			CLUT		SET			TILE NUMBER							

Table 7.2: A Tile Map Entry

7.1.1 Scrolling

Tile maps can scroll across the screen both horizontally and vertically. The position of the tile map on the screen is controlled through the registers at offsets 8, 9, and 10. The horizontal position is controlled by DIR_X, X, and SSX. The vertical position is controlled by DIR_Y, Y, and SSY. The bits X and Y set the position in units of tiles. That is, the number in X[9..0] specifies how many complete tile columns the tile map is moved left or right. Likewise, Y[9..0] specifies how many tile rows the tile map is moved up or down. The SSX and SSY bits are used to specify how many rows of pixels within a tile the tile map is to move. SSX and SSY are therefore "smooth scroll" registers. They have a small trick to their use, however:

If the tile map uses tiles 16 pixels on a side, SSX[3..0] is used to specify the number of pixels to shift the tile map left or right: from 0 to 15. If, on the other hand, the tile map uses tiles 8 pixels on a side, only SSX[3..1] are used to specify the number of pixels to move: from 0 to 7. Note that SSX[0] is not used at all in this case. The SSY bits work in exactly the same way for smooth scrolling in the vertical direction.

Finally, DIR_X and DIR_Y are used to control the direction of the scrolling. If DIR_X is 0, the tile map will move to the left. If DIR_X is 1, the tile map will move to the right. If DIR_Y is 0, the tile map will move to up. If DIR_Y is 1, the tile map will move down. Note that the representation of the amount of the scrolling is separate from the direction (set X to 3 to scroll by 3 tiles, whether that is 3 to the left or 3 to the right). One way to look at the scroll registers is that they are one's complement numbers: a magnitude and a separate sign bit.

To make sure that scrolling will work properly, the tile map needs to be at least as big as the full screen (even if it is largely "empty"), and there should be blank columns to the left and the right and blank rows above and below. That is, it is best to leave an empty margin all the way around your working tile map.

7.2 Tile Sets

Essentially, a tile set is just a bitmap, but of a smaller size and arranged in a particular way. If the smaller (8×8) tiles are used, the tile set image is 128 pixels wide by 128 pixels tall. If the larger (16×16) tiles are used, the tile set image is 256 pixels wide by 256 pixels tall. The image is then divided up into 256 equal sizes squares; that is 16 squares by 16 squares. Each square area of the tile set image is the image data for a particular tile. The tiles are arranged left-to-right and top-to-bottom. Tile 0 is in the upper left of the image and tile 255 is in the lower right. Table 7.3 shows graphically how they are arranged.

As with bitmaps and sprites, the pixels of the tiles are each an individual byte. The contents of the byte (0 – 255) serving as an index into a color lookup table. The pixels are also laid out in left-to-right and top-to-bottom order, just as with bitmaps and individual sprites.

	128 or 256 pixels															
128 or 256 pixels	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	27	28	29	30	31
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	46
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	64	65	66	67	68	69	60	71	72	73	74	75	76	77	78	79
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Table 7.3: Arrangement of Tiles in a Tile Set Image

TinyVicky supports eight separate tile sets. Each one has a single three byte address register, which provides the address to the tile set pixel data, and a configuration register (see table: 7.4). To use them, a program simply stores the address of the pixel data to use into the appropriate address register. The configuration register contains a single SQUARE flag, which indicates the layout of the tile set image. If SQUARE is set (1), the tile set image is square (128×128 pixels for 8×8 tiles or 256×256 pixels for 16×16 tiles). If SQUARE is clear (0), the tile set image is vertical ($8 \times 2,048$ pixels for 8×8 tiles, or $16 \times 4,096$ pixels for 16×16 tiles).

7.2.1 Example: A Simple Tile Map

```
;
; Set up TinyVicky to display tiles
;
lda #$14                    ; Graphics and Tile engines enabled
sta VKY_MSTR_CTRL_0
stz VKY_MSTR_CTRL_1         ; 320x240 @ 60Hz

lda #$40                    ; Layer 0 = Bitmap 0, Layer 1 = Tile map 0
sta VKY_LAYER_CTRL_0
lda #$15                    ; Layer 2 = Tile Map 1
sta VKY_LAYER_CTRL_1

stz VKY_BRDR_CTRL           ; No border

lda #$19                    ; Background: midnight blue
sta VKY_BKG_COL_R
lda #$19
sta VKY_BKG_COL_G
lda #$70
sta VKY_BKG_COL_B
```

To define the tile set, all we really need to do is to set the address register for the tile set to point to the actual pixel data. In this particular case, the code is just going to use tile set 0.

Address	R/W	Tile Set	7	6	5	4	3	2	1	0
0xD280	W	0	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD281	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD282	W		—						AD17	AD16
0xD283	W		—				SQUARE	—		
0xD284	W	1	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD285	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD286	W		—						AD17	AD16
0xD287	W		—				SQUARE	—		
0xD288	W	2	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD289	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD28A	W		—						AD17	AD16
0xD28B	W		—				SQUARE	—		
0xD28C	W	3	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD28D	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD28E	W		—						AD17	AD16
0xD28F	W		—				SQUARE	—		
0xD290	W	4	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD291	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD292	W		—						AD17	AD16
0xD293	W		—				SQUARE	—		
0xD294	W	5	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD295	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD296	W		—						AD17	AD16
0xD297	W		—				SQUARE	—		
0xD298	W	6	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD299	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD29A	W		—						AD17	AD16
0xD29B	W		—				SQUARE	—		
0xD29C	W	7	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD29D	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD29E	W		—						AD17	AD16
0xD29F	W		—				SQUARE	—		

Table 7.4: Tile Set Registers

```
;
; Set tile set #0 to our image
;

lda #<tiles_img_start
sta VKY_TS0_ADDR_L
lda #>tiles_img_start
sta VKY_TS0_ADDR_M
lda #‘tiles_img_start
sta VKY_TS0_ADDR_H
```

Finally, the code sets up the tile map itself, setting the size of the tiles, the size of the tile map, setting the position of the screen in the tile map, and pointing to the tile map data.

```
;
; Set tile map #0
;

lda #$01                    ; 16x16 tiles, enable
sta VKY_TM0_CTRL

lda #22                     ; Our tile map is 20x15
sta VKY_TM0_SIZE_X
lda #16
sta VKY_TM0_SIZE_Y
```

```
            lda #<tile_map              ; Point to the tile map
            sta VKY_TM0_ADDR_L
            lda #>tile_map
            sta VKY_TM0_ADDR_M
            lda #‘tile_map
            sta VKY_TM0_ADDR_H

            lda #$0F                    ; Set scrolling (15, 0)
            sta VKY_TM0_POS_X_L
            lda #$00
            sta VKY_TM0_POS_X_H

            stz VKY_TM0_POS_Y_L
            stz VKY_TM0_POS_Y_H
```

The tile map itself. In this case, we just define it in-line. The data is formatted to match the dimensions of the tile map for ease of reading. Note that the left-most and right-most columns are essentially blank, providing some buffer space to allow for scrolling. Similarly, there is a spare row on the bottom. This data is formatted as single hexadecimal digits, to make it easier to format this data on the page, but the data is actually stored as 16-bit values. This is taking advantage of the fact that the code is using CLUT 0 and LAYER 0 for the tiles and that there are no more than 16 tiles in the tile set.

```
tile_map:
.word $4,$1,$0,$1,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$4,$0,$4,$0
.word $0,$0,$1,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$4,$0,$0
.word $0,$1,$0,$1,$0,$0,$6,$7,$7,$7,$7,$7,$7,$7,$7,$8,$0,$0,$4,$0,$4,$0
.word $0,$0,$0,$0,$0,$0,$9,$1,$2,$3,$4,$5,$0,$0,$0,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$2,$1,$2,$3,$4,$5,$0,$0,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$3,$2,$1,$2,$3,$4,$5,$0,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$4,$3,$2,$1,$2,$3,$4,$5,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$5,$4,$3,$2,$1,$2,$3,$4,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$0,$5,$4,$3,$2,$1,$2,$3,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$0,$0,$5,$4,$3,$2,$1,$2,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$9,$0,$0,$0,$5,$4,$3,$2,$1,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$B,$C,$C,$C,$C,$C,$C,$C,$C,$D,$0,$0,$0,$0,$0,$0
.word $0,$3,$0,$3,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$2,$0,$2,$0
.word $0,$0,$3,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$2,$0,$0
.word $0,$3,$0,$3,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$2,$0,$2,$0
.word $0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$2,$0,$0,$4
```

8 Sound

The F256jr supports two different sound chips, although only one is built-in. The built-in sound chip is the SN76489 (called the "PGS" here), which was used by many vintage machines including the TI99/4A, the BBC Micro, the IBM PCjr, and the Tandy 1000. On the F256jr, the PSG is implemented as two stereo PSGs in the FPGA. The other chip supported is the Commodore SID chip (6581 or 8580). The SID is not installed by default, but the board comes with two sockets for SID chips. The F256jr supports the original 6581, the lower voltage 8580, and the modern SID replacement projects.

8.1 CODEC

The F256jr (and indeed all the Foenix computers up to this point) makes use of a WM8776 CODEC chip. You can think of the CODEC as the central switchboard for audio on the F256jr. The CODEC chip has inputs for several different audio channels (both analog and digital), and each audio device on the F256jr is routed to an input on the CODEC. The CODEC then has outputs for audio line level and headphones. The CODEC will convert analog inputs to digital, mix all the audio inputs according to its settings, and then convert the resulting digital audio to analog and drive the outputs. With the CODEC, you can turn on and off the various input channels, control the volume, and mute or enable the different outputs.

The CODEC is a rather complex chip with many features, and the full details are really beyond the scope of this document. Most programs for the F256jr will not need to use it or will only use it in very specific ways. Therefore, this document will really just show how to access it and initialize it and then leave a reference to the data sheet for the chip that has the complete data on the chip.

Raw access to the CODEC chip is fairly complex. Fortunately, the FPGA on the F256jr provides three registers to simply access for programs. The FPGA takes care of the actual timing of transmitting data to the CODEC, serializing the data correctly, and so on. All the program needs to know about are the correct format for the 16-bit command words that are sent to the CODEC, and then a status register to monitor.

The CODEC commands are based around a number of registers. Each command is really just writing values to those registers. The command words are 16-bits wide, with the 7 most significant bits being the number of the register to write, and the 9 least significant bits being the data to write. For instance, there is a register to enable and disable the headphone output. Bit 0 of the register controls whether or not the headphone output is enabled (0 = enabled, 1 = disabled). The register number is 13. So, to disable the output on the headphones, we would need to write `000000001` to register 13. The register number in binary is `0001101`, So the command word we would need to send is `0001101000000001\` or `0x1A01`.

The registers for the CODEC on the F256jr are:

Address	R/W	7	6	5	4	3	2	1	0	Purpose
0xD620	W	D7	D6	D5	D4	D3	D2	D1	D0	Command Low
0xD621	W	R6	R5	R4	R3	R2	R1	R0	D8	Command High
0xD622	R	X							BUSY	Status
0xD622	W	X							START	Control

Table 8.1: CODEC Control Registers

Bit 0 of the status/control register both triggers sending the command (on a write) and indicates if the CODEC is busy receiving a command (writing a 1 triggers the sending of the command, reading a 1 indicates that the CODEC is busy).

So to mute the headphones, we would issue the following:

```
wait:  lda $D622    ; Wait for the CODEC to be ready
       and #$01
   cmp #$01
       beq wait  ; Bit 0 = 1, CODEC is still busy... keep waiting
```

```
lda #$01 ; Set command to %0001101000000001, or R13 <- 000000001
sta $D620
lda #$1A
sta $D621

lda #$01 ; Trigger the transmission of the command to the CODEC
sta $D622
```

8.2 Using the PSGs

The F256jr has support for dual SN76489 (PSG) sound chips, emulated in the FPGA. The SN76489 was used in several vintage machines, including the TI-99/4A, BBC Micro, IBM PCjr, and Tandy 1000. The chip provides three independent square-wave tone generators and a single noise generator. Each tone generator can produce tones of several frequencies in 16 different volume levels. The noise generator can produce two different types of noise in three different tones at 16 different volume levels.

Access to each PSG is through a single memory address, but that single address allows the CPU to write a value to eight different internal registers. For each tone generator, there is a ten bit frequency (which takes two bytes to set), and a four bit “attenuation” or volume level. For the noise generator, there is a noise control register and a noise attenuation register.

R2	R1	R0	Channel	Purpose
0	0	0	Tone 1	Frequency
0	0	1	Tone 1	Attenuation
0	1	0	Tone 2	Frequency
0	1	1	Tone 2	Attenuation
1	0	0	Tone 3	Frequency
1	0	1	Tone 3	Attenuation
1	1	0	Noise	Control
1	1	1	Noise	Attenuation

Table 8.2: SN76489 Channel Registers

There are four basic formats of bytes that can be written to the port:

D7	D6	D5	D4	D3	D2	D1	D0	Purpose
1	R2	R1	R0	F3	F2	F1	F0	Set the low four bits of the frequency
0	X	F9	F8	F7	F6	F5	F4	Set the high seven bits of the frequency
1	1	1	0	X	FB	F1	F0	Set the type and frequency of the noise generator
1	R2	R1	R0	A3	A2	A1	A0	Set the attenuation (four bits)

Table 8.3: SN76489 Command Formats

Note: there is a PSG sound device for the left stereo channel and one for the right. The left channel PSG can be accessed at 0xD600, and the right channel at 0xD610. Both are in I/O page 0.

8.2.1 Attenuation

All the channels support attenuation or volume control. The PSG expresses the loudness of the sound with how much it is attenuated or dampened. Therefore, an attenuation of 0 will be the loudest sound, while an attenuation of 15 will make the channel silent.

8.2.2 Tones

Each of the three sound channels generates simple square waves. The frequency generated depends upon the system clock driving the chip and the number provided in the frequency register. The relationship is:

$$f = \frac{C}{32n}$$

where f is the frequency produced, C is the system clock, and n is the number provided in the register. Expressed a different way, the value we need to produce a given frequency can be computed as:

$$n = \frac{C}{32f}$$

For the F256jr the system clock is 3.57 MHz, which means:

$$n = \frac{111,563}{f}$$

So, let us say we want channel 1 to produce a concert A, which is 440Hz at maximum volume. The value we need to set for the frequency code is $111,320/440 = 253$ or 0xFE. We can do that with this code:

```
lda #$90        ; %10010000 = Channel 1 attenuation = 0
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG

lda #$8E        ; %10001100 = Set the low 4 bits of the frequency code
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG

lda #$0F        ; %00001111 = Set the high 7 bits of the frequency
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG
```

To turn it off later, we just need to write:

```
lda #$9F        ; %10011111 = Channel 1 attenuation = 15 (silence)
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG
```

8.2.3 Noise

Noise works differently from tones, since it is random. The noise generator on the PSG can produce two styles of noise determined by the FB bit: white noise (FB = 1), and periodic (FB = 0). The noise has a sort of frequency, based on either the system clock or the current output of tone 3. This frequency is set using the F1 and F0 bits:

F1	F0	Frequency
0	0	$C/512$
0	1	$C/1024$
1	0	$C/2048$
1	1	Tone 3 output

Table 8.4: SN76489 Noise Frequencies

As an example, to set white noise of the highest frequency ($C/512$ or around 6 kHz), we could use the code:

```
lda #$F0        ; %10010000 = Channel 3 attenuation = 0
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG

lda #$E4        ; %11100100 = white noise, f = C/512
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG
```

To turn it off later, we just need to write:

```
lda #$FF        ; %1ff11111 = Channel 3 attenuation = 15 (silence)
sta $D600       ; Send it to left PSG
sta $D610       ; Send it to right PSG
```

8.3 Using the SIDs

The SID is a full-featured analog sound synthesizer, and a full explanation of how to use it is really beyond the scope of this document. In this document, I will provide just an introduction to the chip and list the register addresses for the SID chips that can be installed on the F256jr (see table 8.5).

The SID chip provides three independent voices (so it can play three notes at once). The three voices are almost identical in their features, with voice 3 being the only one different. Each voice can produce one of four basic sound wave forms: randomized noise, square waves, saw tooth waves, and triangle waves. These waves can be generated over a range of frequencies, and for the square waves, the width of the pulse (*i.e. duty cycle*) may be adjusted.

The type of wave form produced by a voice is controlled by the NOISE, PULSE, SAW, and TRI bits. If NOISE is set to 1, the output is random noise. If PULSE is set, a square wave is produced. If SAW is set, a saw tooth wave is produced. If TRI is set, the voice produces a triangle wave. If PULSE is set, the duty cycle of the square wave (or pulse width, if you prefer) is set by the PW bits according to the formula PW/40.95 (expressed as a percent).

Voice	Offset	R/W	7	6	5	4	3	2	1	0
V1	0	W	F7	F6	F5	F4	F3	F2	F1	F0
	1	W	F15	F14	F13	F12	F11	F10	F9	F8
	2	W	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
	3	W	X				PW11	PW10	PW9	PW8
	4	W	NOISE	PULSE	SAW	TRI	TEST	RING	SYNC	GATE
	5	W	ATK3	ATK2	ATK1	ATK0	DLY3	DLY2	DLY1	DLY0
	6	W	STN3	STN2	STN1	STN0	RLS3	RLS2	RLS1	RLS0
V2	7	W	F7	F6	F5	F4	F3	F2	F1	F0
	8	W	F15	F14	F13	F12	F11	F10	F9	F8
	9	W	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
	10	W	X				PW11	PW10	PW9	PW8
	11	W	NOISE	PULSE	SAW	TRI	TEST	RING	SYNC	GATE
	12	W	ATK3	ATK2	ATK1	ATK0	DLY3	DLY2	DLY1	DLY0
	13	W	STN3	STN2	STN1	STN0	RLS3	RLS2	RLS1	RLS0
V3	14	W	F7	F6	F5	F4	F3	F2	F1	F0
	15	W	F15	F14	F13	F12	F11	F10	F9	F8
	16	W	PW7	PW6	PW5	PW4	PW3	PW2	PW1	PW0
	17	W	X				PW11	PW10	PW9	PW8
	18	W	NOISE	PULSE	SAW	TRI	TEST	RING	SYNC	GATE
	19	W	ATK3	ATK2	ATK1	ATK0	DLY3	DLY2	DLY1	DLY0
	20	W	STN3	STN2	STN1	STN0	RLS3	RLS2	RLS1	RLS0
	21	W	X					FC2	FC1	FC0
	22	W	FC10	FC9	FC8	FC7	FC6	FC5	FC4	FC3
	23	W	RES3	RES2	RES1	RES0	EXT	FILTV3	FILTV2	FILTV1
	24	W	MUTEV3	HIGH	BAND	LOW	VOL3	VOL2	VOL1	VOL0

Table 8.5: SID Registers

The frequency of the waveform is set by the bits `F[15..0]`. This number sets the actual frequency according the the formula:

$$f_{\text{out}} = \frac{FC}{16777216}$$

where: f_{out} is the output frequency, F is the number set in the registers, and C is the system clock driving the SIDs. For the F256jr, C is 1.022714 MHz, so the formula for the F256jr is:

$$f_{\text{out}} = \frac{F}{16.405}$$

or:

$$F = 16.404 f_{\text{out}}$$

For example: concert A, which is 440 Hz, would be: $F = 16.405 \times 440 \approx 7218$. So, to play a concert A, you would set the frequency to 7218, or 0x1C32.

Each of the three voices has a sound “envelope” which changes the volume of the sound during the duration of the note. There are four phases to the sound envelope: attack, decay, sustain, and release (ADSR). When the note first starts playing (that is, the GATE bit for the voice is set to 1), it starts at the attack phase when the volume starts at zero and goes up to the current maximum volume. How fast this happens is determined by the attack rate (ATK3-0 in the registers). Once the volume reaches the maximum, the volume goes down again to the sustain volume. This phase is called decay, and the speed at which the volume drops is determined by the DCY3-0 register values. Next, the envelope enters the sustain phase, where the volume is held steady at the sustain level (STN3-0). It stays here until the note is to stop playing (GATE is set to 0). At this point, the envelope enters the release stage, where the volume drops back to zero at the release rate (RLS3-0).

The ADSR envelope allows the SID chip to mimic the qualities of various musical instruments or shape various sound effects. For instance, a pipe organ’s notes are typically either on or off, so the attack, decay, and release rates would be set to be instantaneous, and the sustain level would be set to full. A piano, on the other hand tends to have a sharp, somewhat percussive sound at the beginning with the note holding a long time on release if not dampened.

While the different voices are independent, they can be set to alter one another through two different effects: synchronization, and ring modulation. With these features, the voices can interact with each other in the following pairs:

- Voice 1 → Voice 2
- Voice 2 → Voice 3
- Voice 3 → Voice 1

8.3.1 Ring Modulation

If a voice's RING bit is set and the voice is set to use the triangle wave form (TRI is set), then the triangle wave will be replaced by the combination of the two voice's frequencies. So if the RING bit of voice 1 is set, the result will be the ring modulation of voice 1 and voice 3. Ring modulation tends to produce harmonics and overtones and can be used for bell like sounds.

8.3.2 Synchronization

If a voice's SYNC bit is set, the frequency it produces will be synchronized to the controlling voice. So if voice 1's SYNC bit is set, its frequency will be synchronized to voice 3.

9 Interrupt Controller

The 65C02 has two interrupts: non-maskable interrupts (NMI) for high priority events, and the regular interrupt request line (IRQ) for normal priority events. Currently, the C256 series of computers do not use NMI for any purpose, so the only interrupt is the IRQ line. There are many devices on the F256jr which can trigger interrupts, so to save the interrupt handler the chore of querying each device in turn, the F256jr provides an interrupt controller module. The individual devices route their interrupt request signals to the interrupt controller. When an interrupt comes in, the controller knows which device it is and decides whether or not to forward the interrupt to the CPU. The interrupt handler can then query the interrupt handler to see which device or devices have interrupts pending and can then acknowledge them once they have been properly handled.

The interrupt controller provides two sets of registers to provide its functionality: interrupt masks, and interrupt pending. The mask registers provide a mask flag for every possible interrupt. If the flag is true (1), the interrupt from that device is masked, and an interrupt coming in from it will not trigger an IRQ on the processor. The pending register provides a pending flag for every possible interrupt. If the flag is true (1) on a read, the device has requested an interrupt, if false (0) there is no interrupt pending from the device. Setting a flag in a pending register to 1 will acknowledge the interrupt and cause the controller to drop the pending flag. Writing a 0 to a flag will have no effect.

NOTE: Some of the devices on the F256jr have their own interrupt enable flags (separate from the mask flags). For example, the 65C22 VIA has an interrupt enable bit in one of its registers and will not send an interrupt to the F256jr's interrupt controller if that bit is not enabled. For such devices, the interrupt enable flag on the device must be set and the corresponding mask bit in the interrupt controller must be clear in order for interrupts to be sent to the CPU. Other devices, like VICKY, do not have a separate enable flag. In their case, only their corresponding mask bits must be cleared to enable their interrupts.

Mask	Pending	Bit	Name	Purpose
0xD666	0xD660	0x01	INT_VKY_SOF	TinyVicky Start Of Frame interrupt [1]
		0x02	INT_VKY_SOL	TinyVicky Start Of Line interrupt [2]
		0x04	INT_PS2_KBD	PS/2 keyboard event
		0x08	INT_PS2_MOUSE	PS/2 mouse event
		0x10	INT_TIMER_0	TIMER0 has reached its target value
		0x20	INT_TIMER_1	TIMER1 has reached its target value
		0x40	INT_DMA	DMA has completed
		0x80	RESERVED	
0xD667	0xD661	0x01	INT_UART	The UART is ready to receive or send data
		0x02	RESERVED	
		0x04	RESERVED	
		0x08	RESERVED	
		0x10	INT_RTC	Event from the real time clock chip
		0x20	INT_VIA	Event from the 65C22 VIA chip
		0x40	INT_IEC	Event from the IEC serial bus
		0x80	INT_SDC_INS	User has inserted an SD card

As an example of working with the interrupt controller, let's try using the SOF interrupt to alter the character in the upper left corner.

To start, we will need to install our interrupt handler to respond to IRQs. For this example, we're going to completely take over interrupt processing, so we'll do some things we wouldn't ordinarily do. Also, since an interrupt could come in while we're setting things, up, we need to be careful about how we do things.

1. First, we want to disable IRQs at the CPU level.

2. Then we set the interrupt vector.
3. Next, we want to mask off all but the SOF interrupt, since that is the only one we will process (in real programs, we will either need to handle several interrupts or play nicely with the operating system).
4. Now, there might be interrupts that came in earlier, so will go ahead and just clear all the pending interrupt flags so we start cleanly.
5. Finally, we enable CPU interrupt handling again and loop forever... processing the SOF interrupt when it comes in.

```
VIRQ = $FFFE

INT_PEND_0 = $D660  ; Pending register for interrupts 0 - 7
INT_PEND_1 = $D661  ; Pending register for interrupts 8 - 15
INT_MASK_0 = $D666  ; Mask register for interrupts 0 - 7
INT_MASK_1 = $D667  ; Mask register for interrupts 8 - 15

start:      ; Disable IRQ handling
            sei

            ; Load my IRQ handler into the IRQ vector
            ; NOTE: this code just takes over IRQs completely. It could save
            ;       the pointer to the old handler and chain to it when it had
            ;       handled its interrupt. But what is proper really depends on
            ;       what the program is trying to do.
            lda #<my_handler
            sta VIRQ
            lda #>my_handler
            sta VIRQ+1

            ; Mask off all but the SOF interrupt
            lda #$ff
            sta INT_MASK_1
            and #~INT00_VKY_SOF
            sta INT_MASK_0

            ; Clear all pending interrupts
            lda #$ff
            sta INT_PEND_0
            sta INT_PEND_1

            ; Put a character in the upper right of the screen
            lda #SYS_CTRL_TEXT_PG
            sta SYS_CTRL_1

            lda #'@'
            sta $c000

            ; Set the color of the character
            lda #SYS_CTRL_COLOR_PG
            sta SYS_CTRL_1

            lda #$F0
            sta $c000

            ; Go back to I/O page 0
            stz SYS_CTRL_1

            ; Make sure we're in text mode
            lda #$01            ; enable TEXT
            sta $D000           ; Save that to VICKY master control register 0
            stz $D001

            ; Re-enable IRQ handling
            cli
```

To actually process the interrupt, we need to read and then increment the character at the start of the screen, clear the pending flag for the SOF interrupt, and then return. However, the screen and the interrupt control registers are in different I/O banks, so we'll need to change the I/O bank a couple of times during interrupt processing. So, the first thing we will do is to save the value of the system control register at 0x0001, so we can restore it before we return from the interrupt.

```
SYS_CTRL_1 = $0001
SYS_CTRL_TEXT_PG = $02

my_handler: pha

            ; Save the system control register
            lda SYS_CTRL_1
            pha

            ; Switch to I/O page 0
            stz SYS_CTRL_1

            ; Check for SOF flag
            lda #INT00_VKY_SOF
            bit INT_PEND_0
            beq return              ; If it's zero, exit the handler

            ; Yes: clear the flag for SOF
            sta INT_PEND_0

            ; Move to the text screen page
            lda #SYS_CTRL_TEXT_PG
            sta SYS_CTRL_1

            ; Increment the character at position 0
            inc $c000

            ; Restore the system control register
return:     pla
            sta SYS_CTRL_1

            ; Return to the original code
            pla
            rti
```

10 Tracking Time

10.1 Interval Timers

10.2 Real Time Clock

For programs needing to keep track of time, F256jr provides a real time clock chip (RTC), the bq4802. This chip, keeps track of the year (including century), month, day, hour (in 12 or 24 hour mode), minute, and second. The coin cell battery on the F256jr motherboard is to provide power to the RTC so it can continue tracking time even when the F256jr is turned off or unplugged. Additionally, the RTC can send interrupts to the CPU, either periodically or at a specific time.

The RTC is relatively straightforward to use, but one potentially tricky thing to keep in mind is that there is a specific procedure to follow when reading or writing the date-time. As well as the registers the CPU can access, the RTC has internal registers which are constantly updating as time progresses. Normally, the internal registers update their external counterparts, but this should not be allowed to happen while the CPU is getting or setting the externally facing registers. So, to access the external registers, the program must first disable the automatic updates to the external registers. Then it can read or write the external registers. Then it can re-enable the automatic updates. If the program has changed the registers, when updates are re-enabled the data in the external registers will be sent to the internal registers in one action. This keeps the time information consistent.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD690	R/W	Seconds	0	second 10s digit			second 1s digit			
0xD691	R/W	Seconds Alarm	0	second 10s digit			second 1s digit			
0xD692	R/W	Minutes	0	minute 10s digit			minute 1s digit			
0xD693	R/W	Minutes Alarm	0	minute 10s digit			minute 1s digit			
0xD694	R/W	Hours	AM/PM	0	hour 10s digit		hour 1s digit			
0xD695	R/W	Hours Alarm	AM/PM	0	hour 10s digit		hour 1s digit			
0xD696	R/W	Days	0	0	day 10s digit		day 1s digit			
0xD697	R/W	Days Alarm	0	0	day 10s digit		day 1s digit			
0xD698	R/W	Day of Week	0	0	0	0	0	day of week digit		
0xD699	R/W	Month	0	0	0	month 10s digit	month 1s digit			
0xD69A	R/W	Year	year 10s digit				year 1s digit			
0xD69B	R/W	Rates	0	WD			RS			
0xD69C	R/W	Enables	0	0	0	0	AIE	PIE	PWRIE	ABE
0xD69D	R/W	Flags	0	0	0	0	AF	PF	PWRF	BVF
0xD69E	R/W	Control	0	0	0	0	UTI	STOP	12/24	DSE
0xD69F	R/W	Century	century 10s digit				century 1s digit			

Table 10.1: MMU Registers

There are 16 registers for the RTC (see table 10.1). There is a register each for century, year, month, day of the week (*i.e.* Sunday - Saturday), day, hour, minute, and second. Each one is expressed in binary-coded-decimal, meaning the lower four bits are the ones digit (0 - 9), and the upper bits are the 10s digit. In most cases, the upper digit is limited (*e.g.* seconds and minutes can only have 0 - 6 as the tens digit). For seconds, minutes, hours, and day there is a separate alarm register, which will be described later. Finally, there are the four registers for rates, enabled, flags, and control:

The Enables register has four separate enable bits:

AIE if set (1), the alarm interrupt will be enabled. The RTC will raise an interrupt when the current time matches the time specified in the alarm registers.

PIE if set (1), the RTC will raise an interrupt periodically, where the period is specified by the RS field.

PWRIE if set (1), the RTC will raise an interrupt on a power failure (not relevant to the F256jr).

ABE if set (1), the RTC will allow alarm interrupts when on battery backup (not relevant to the F256jr).

The Flags register has four separate flags, which generally reflect why an interrupt was raised:

AF if set (1), the alarm was triggered

PF if set (1), the periodic interrupt was triggered

PWRF if set (1), the power failure interrupt was triggered

BVF if set (1), the battery voltage is within safe range. If clear (0), the battery voltage is low, and the time may be invalid.

The Control register has four bits which change how the RTC operates:

UTI if set (1), the update of the externally facing registers by the internal timers is inhibited. In order to read or write those registers, the program must first set UTI and then clear it when done.

STOP this bit allows for a battery saving feature. If it is clear (0) before the system is powered down, it will avoid draining the battery and may stop tracking the time. If it is set (1), it will keep using the battery as long as possible.

12/24 sets whether the RTC is using 12 or 24 hour accounting.

DSE if set (1), daylight savings is in effect.

The Rates register controls the watchdog timer and the periodic interrupt. The watchdog timer is not really relevant to the F256jr, but it monitors for activity and raises an interrupt if activity has not been seen within a certain amount of time (specified by the WD field). The periodic interrupt will be raised repeatedly, the period of which is set by the RS field (see table 10.2).

RS3	RS2	RS1	RS0	Period
0	0	0	0	None
0	0	0	1	30.5175 μs
0	0	1	0	61.035 μs
0	0	1	1	122.070 μs
0	1	0	0	244.141 μs
0	1	0	1	488.281 μs
0	1	1	0	976.5625 μs
0	1	1	1	1.95315 ms
1	0	0	0	3.90625 ms
1	0	0	1	7.8125 ms
1	0	1	0	15.625 ms
1	0	1	1	31.25 ms
1	1	0	0	62.5 ms
1	1	0	1	125 ms
1	1	1	0	250 ms
1	1	1	1	500 ms

Table 10.2: RTC Periodic Interrupt Rates

10.2.1 Example: Display the Time

In this example, we will read the time from the real time clock chip and print it out to the screen in *hh:mm:ss* format. To make things a little different, the code will also use the OpenKernal calls to initialize the text screen and print text. OpenKernel is a clean-room implementation of the Commodore Kernel calls. The basic procedure is fairly simple: first the code disables the update of the transfer registers, then the code reads the hours and prints them, then the code reads the minutes and prints them, then the code fetches the seconds and prints them. Finally, the code re-enables the update of the transfer registers by dropping the UTI flag.

NOTE: This code resets the MMU I/O page to 0 before it tries to read from the clock chip. This is just to allow for the possibility of the kernel routines changing the I/O page without restoring it to 0.

```
ok_cint = $FF81 ; OpenKernal call to initialize the screen
ok_cout = $FFD2 ; OpenKernal call to print the character code in A

RTC_SECS = $D690 ; RTC Seconds register
RTC_MINS = $D692 ; RTC Minutes register
RTC_HOURS = $D694 ; RTC Hours register

RTC_CTRL = $D96E ; RTC Control register
RTC_24HR = $02 ; 12/24 hour flag (1 = 24 Hr, 0 = 12 Hr)
RTC_STOP = $04 ; 0 = STOP when power off, 1 = run from battery when power off
RTC_UTI = $08 ; Update Transfer Inhibit

start:      jsr ok_cint                 ; Initialize the text screen

            stz MMU_IO_CTRL             ; Make sure we're on I/O page 0

            lda RTC_CTRL                ; Stop the update of the RTC registers
            ora #RTC_UTI | RTC_24HR
            sta RTC_CTRL

            stz MMU_IO_CTRL ; Make sure we're on I/O page 0

            lda RTC_HOURS               ; Print the hours
            jsr putbcd

            lda #':'
            jsr ok_cout

            stz MMU_IO_CTRL ; Make sure we're on I/O page 0

            lda RTC_MINS                ; Print the minutes
            jsr putbcd

            lda #':'
            jsr ok_cout

            stz MMU_IO_CTRL ; Make sure we're on I/O page 0

            lda RTC_SECS                ; Print the seconds
            jsr putbcd

            stz MMU_IO_CTRL ; Make sure we're on I/O page 0

            lda RTC_CTRL                ; Reenable the update of the registers
            and #~RTC_UTI
            sta RTC_CTRL
```

Since the time registers of the clock chip are encoded in binary-coded-decimal, printing is relatively straightforward, and is handled by a simple `putbcd` subroutine:

```
;
; Print a BCD number to the screen
;
putbcd:     pha                         ; Save the number
            and #$F0                    ; Isolate the upper digit
            lsr a
            lsr a
            lsr a
            lsr a

            clc                         ; Convert to ASCII
            adc #'0'
            jsr ok_cout                 ; And print

            pla                         ; Get the full number back
```

```
        and #$0F                    ; Isolate the lower digit

        clc                         ; Convert to ASCII
        adc #'0'
        jsr ok_cout                 ; And print

        rts
```

11 Versatile Interface Adapter

The F256jr includes a Western Design Center WDC65C22 versatile interface adapter or VIA. The VIA provides several useful features for I/O and timing:

- Two independent I/O ports of eight parallel bits (PA, and PB).
- Four handshake control lines (CA1, CA2, CB1, and CB2)
- Programmable serial register for serial I/O operations
- Two independent timer counters

On the F256jr, the VIA is connected to header which is compatible with the keyboard header on the Commodore VIC-20 and C-64. This means that a Commodore compatible keyboard could be connected to the F256jr and used for keyboard input with appropriate programming. The VIA also provides access to the two Atari-style joystick ports. The pins could also be used for general purpose I/O, although the voltage levels are for 3 volt logic instead of the 5 volt logic used in older 8-bit machines.

A complete description of the VIA would be rather long, so this guide will merely list out the register addresses and provide a quick break-down on the register functions. For a complete description, please see the data sheet from Western Design Center.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xDC00	R/W	IORB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
0xDC01	R/W	IORA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
0xDC02	R/W	DDRB	DDRB7	DDRB6	DDRB5	DDRB4	DDRB3	DDRB2	DDRB1	DDRB0
0xDC03	R/W	DDRA	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0
0xDC04	R/W	T1C_L	T1C7	T1C6	T1C5	T1C4	T1C3	T1C2	T1C1	T1C0
0xDC05	R/W	T1C_H	T1C15	T1C14	T1C13	T1C12	T1C11	T1C10	T1C9	T1C8
0xDC06	R/W	T1L_L	T1L7	T1L6	T1L5	T1L4	T1L3	T1L2	T1L1	T1L0
0xDC07	R/W	T1L_H	T1L15	T1L14	T1L13	T1L12	T1L11	T1L10	T1L9	T1L8
0xDC08	R/W	T2C_L	T2C7	T2C6	T2C5	T2C4	T2C3	T2C2	T2C1	T2C0
0xDC09	R/W	T2C_H	T2C15	T2C14	T2C13	T2C12	T2C11	T2C10	T2C9	T2C8
0xDC0A	R/W	SR	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0
0xDC0B	R/W	ACR	T1_CTRL		T2_CTRL	SR_CTRL			PBL_EN	PAL_EN
0xDC0C	R/W	PCR	CB2_CTRL			CB1_CTRL	CA2_CTRL			CA1_CTRL
0xDC0D	R/W	IFR	IRQF	T1F	T2F	CB1F	CB2F	SRF	CA1F	CA2F
0xDC0E	R/W	IER	SET	T1E	T2E	CB1E	CB2E	SRE	CA1E	CA2E
0xDC0F	R/W	IOPA2	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0

Table 11.1: VIA Registers

IORA Input/Output Register for Port A. The eight bits correspond to the eight pins on port A.

DDRA Data Direction Register for Port A. Each bit configures the corresponding pin to be input (0) or output (1).

IORB Input/Output Register for Port B. The eight bits correspond to the eight pins on port B.

DDRB Data Direction Register for Port B. Each bit configures the corresponding pin to be input (0) or output (1).

T1C Timer 1 counter value

T1L Timer 1 latch

T2C Timer 2 counter value

SR is the shift register. Serial input may be read here, or data may be written here to be shifted out.

ACR Auxillary Control Register. Contains fields to control the function of timer 1, timer 2, the shift register, and how Port A and Port B latch data.

PCR Peripheral Control Register. Contains fields to control how the CA1, CA2, CB1, and CB2 handshake pins are used.

IFR Interrupt Flag Register. Contains flags indicating which condition triggered an interrupt request. Possible conditions are timer 1, timer 2, CB1, CB2, CA1, CA2, and shift register complete.

IER Interrupt Enable Register. Contains flags to enable or disable interrupts based on the different possible conditions.

IOPA2 Same as IOPA except that the built-in handshaking capability is not used.

11.1 Joystick Support

The F256jr has two IDC headers compatible with standard IDC to DB-9 cables and which may be used to connect Atari style joysticks. Joystick header 0 is wired to the pins of Port A, and joystick header 1 is connected to Port B. The various joystick switches are connected to the ports in same manner as on the C-64, with the exception that more buttons are supported (see table: 11.2).

In order to use the joysticks, the DDR bits for the ports must be set to 0 for input. Then the input/output register for the port may be read. If a button or switch is closed on the joystick, the corresponding bit in the I/O register will be clear (0). If the button is not pressed, the bit will be set (1).

As a reminder: be aware that the WDC65C22 on the F256jr is being used with a 3 volt supply. This means that any device plugged into the joystick ports should be 3 volt tolerant and should not raise any pin above 3 volts. Otherwise damage could occur.

7	6	5	4	3	2	1	0
—		BUTTON1	BUTTON0	RIGHT	LEFT	DOWN	UP

Table 11.2: Joystick Flags

12 SD Card Interface

The F256jr includes a controller for SD cards. This controller provides for a simplified interface to SD cards, allowing programs to relatively easily transfer blocks of data to and from an SD card. This simplified interface has its limitations, and it only works for older, standard SD cards. More advanced SD cards, like SDHC and SDXC cards, will not work with this simpler interface. The interface does provide a direct access method, however, which allows programs to send commands and data directly to an SD card. With adequate programming, any SD card should be usable with this method.

The simplified interface works off the idea of a transfer. The program wanting to use the SD card sets up a transfer, providing the information the controller needs to perform the transfer, and then starts the transfer. Once the transfer is completed, the program either receives an interrupt from the SD controller or monitors the BUSY status to see if the transfer is complete. There are four types of transfers:

INIT for initializing the SD controller. A program will very rarely need to call this.

DIRECT for sending data directly over the SPI interface to the SD card. This is only needed if the program needs to access functionality on the card the SD controller does not support.

READ to read 512 bytes of data from the SD card

WRITE to write 512 bytes of data to the SD card

12.1 Reading from the SD Card

To read from an SD card, a program needs to:

1. Set the tranfer type to READ
2. Set the SD Address Register with the address of the desired memory block
3. Set the START flag to begin the transfer
4. Wait for the transfer to complete (either poll BUSY or wait for an interrupt)
5. Read bytes from RXR until RCR is 0

12.2 Writing to the SD Card

To write to an SD card, a program needs to:

1. Set the tranfer type to WRITE
2. Set the SD Address Register with the address of the desired memory block
3. Write the 512 bytes to store in the block to TXR
4. Set the START flag to begin the transfer
5. Wait for the transfer to complete (either poll BUSY or wait for an interrupt)

VER contains the version of the SD controller, broken up into a major version number and a minor version number

SCR is the SD control register. There is just one flag here: RESET. If set, this will cause the controller to reset itself.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xDD00	R	VER	VER_MAJ				VER_MIN			
0xDD01	R/W	SCR	—							RESET
0xDD02	R/W	XTR	—						XFER_TYPE	
0xDD03	R/W	XCR	—							START
0xDD04	R	XSR	—							BUSY
0xDD05	R	XSR	—		WRITE_ERR		READ_ERR		INIT_ERR	
0xDD06	R/W	DAR	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0
0xDD07	R/W	SAR	SA7	SA6	SA5	SA4	SA3	SA2	SA1	SA0
0xDD08	R/W		SA15	SA14	SA13	SA12	SA11	SA10	SA9	SA8
0xDD09	R/W		SA23	SA22	SA21	SA20	SA19	SA18	SA17	SA16
0xDD0A	R/W		SA31	SA30	SA29	SA28	SA27	SA26	SA25	SA24
0xDD0B	R/W	SCLK	SPI_CLK							
0xDD10	R	RXR	RX7	RX6	RX5	RX4	RX3	RX2	RX1	RX0
0xDD12	R	RCR	RC15	RC14	RC13	RC12	RC11	RC10	RC9	RC8
0xDD13	R		RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0
0xDD14	R	RFCR	—							RX_CLR
0xDD20	W	TXR	TX7	TX6	TX5	TX4	TX3	TX2	TX1	TX0
0xDD24	R	TFCR	—							TX_CLR

Table 12.1: VIA Registers

XTR is the transfer type register. The SD controller supports four types of transfers: direct access (00), initialization (01), read from the SD card (10), and write to the SD card (11).

XCR is the transfer control register. There is just one flag: START. When set, this begins the transfer. It will clear itself when the transfer is done.

XSR is the transfer status register. There is one flag: BUSY. When set, the controller is busy with a transfer.

XER is the transfer error register. This contains three fields that describes any error condition for the transfer type that applies.

DAR is the register used to send and recieve data for a direct access transfer. This direct access feature is for low-level access to the SPI interface on the SD card and allows for access to more general SD cards than the higher level transfers can use.

SAR is the SD address register. This is a 32-bit address for the block of data on the SD card to read or write.

SCLK is the SPI clock register. It sets the speed of the SPI bus after SD intialization has completed.

RXR is the received data register. All data read from the SD card will be read into a 512 byte FIFO and can be read by the CPU through this register.

RCR is the 16-bit count of the number of bytes available to read from the receive FIFO. Note that the count is in big-endian format.

RFCR is the receive FIFO control register. There is one flag: RX_CLR. When set, it will force the receive FIFO to clear. The flag will clear itself.

TXR is the transmission data register. All data written to the SD card will be written into a 512 byte FIFO through this register.

TFCR is the receive FIFO control register. There is one flag: TX_CLR. When set, it will force the transmit FIFO to clear. The flag will clear itself.

13 PS/2 Keyboard and Mouse

The F256jr provides a single PS/2 port for use with either a keyboard or a mouse. A simple PS/2 keyboard controller is included in the F256jr. The controller emulates the classic Intel 8042 keyboard controller.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD640	R	RXD	RX_DATA							
0xD640	W	TXR	TX_DATA							
0xD644	R	STAT	PE	TE	—		CMD_DAT	SYSFLAG	RXF	TXF
0xD644	W	CMD	COMMAND							

Table 13.1: UART Registers

RXD Data returned by either the keyboard or the controller

TXD Data to be sent to the keyboard or the controller

STAT Status register of the controller:

- **PE** Parity Error
- **TE** Timeout Error
- **CMD_DAT** 0 = data written to the data register is for the PS/2 device. 1 = data is for the controller
- **SYSFLAG** System flag
- **RXF** If set, receive FIFO contains data
- **TXF** If set, transmit FIFO is full

CMD Commands to the controller (not the keyboard) are written here

14 Serial and Wi-Fi Port

The F256jr has a simple UART for serial communications. This UART can be used to provide an RS-232 serial connection (via an IDC header on the board compatible with IDC to DB-9 cables) or a Wi-Fi serial connection using an ESP Feather adapter board. The UART is compatible with the standard 16750.

Address	R/W	Name	7	6	5	4	3	2	1	0
DLAB = 0										
0xD630	R	RXD	RX_DATA							
0xD630	W	TXR	TX_DATA							
0xD631	R/W	IER	—				STATUS	ERROR	TX_EMPTY	RX_AVAIL
0xD632	R	ISR	—				STATUS	ERROR	TX_EMPTY	RX_AVAIL
0xD632	W	FCR	RXT		FIFO64	—	—	TX_RST	RX_RST	FIFO_EN
0xD633	R/W	LCR	DLAB	—	PARITY			STOP	DATA	
0xD634	R/W	MCR	—			LOOP	OUT2	OUT1	RTS	DTR
0xD635	R	LSR	ERROR	TEMT	THRE	BI	FE	PE	OE	DR
0xD636	R/W	MSR	DCD	RI	DSR	CTS	DDCD	TERI	DDSR	DCTS
0xD637	R	SPR	scratch data							
DLAB = 1										
0xD630	R/W	DLL	DIV7	DIV6	DIV5	DIV4	DIV3	DIV2	DIV1	DIV0
0xD631	R/W	DLH	DIV15	DIV14	DIV13	DIV12	DIV11	DIV10	DIV9	DIV8
0xD632	W	PSD	prescaler division							

Table 14.1: UART Registers

LCR1	LCR0	Length
0	0	5
0	1	6
1	0	7
1	1	8

Table 14.2: UART Data Length

LCR2	Stop Bits
0	1
1	1.5 or 2

Table 14.3: UART Stop Bits

LCR5	LCR4	LCR3	Parity
—	—	0	NONE
0	0	1	ODD
0	1	1	EVEN
1	0	1	MARK
1	1	1	SPACE

Table 14.4: UART Parity

FCR7	FCR6	Trigger Level (bytes)
0	0	1
0	1	4
1	0	8
1	1	14

Table 14.5: UART RX FIFO Trigger

15 Direct Memory Access

The DMA engine can either write a specific byte to RAM or copy a set of bytes from one location in RAM to another. The DMA engine can also treat memory as being arranged either linearly (that is, as a certain number of consecutive locations) or rectangularly (the data is a rectangular area of an image).

15.1 Linear Data

Linear data (or "1D", if you prefer) is just a single block of sequential memory locations. When filling or copying data linearly, you need a destination address (and a source address if copying), and a count of bytes to copy. That is really all there is to it.

15.2 Rectangular Data

Rectangular data (or "2D") is a bit more complicated and is meant to be working with image data. With a bitmap, the pixel bytes are arranged in memory left to right and top to bottom. If the image starts at address a and is w pixels wide, then the pixel at (x, y) can be found at location $a + y \times w + x$. Rectangular fills and copies are meant to work on data that is arranged in this fashion. In this case, you can use DMA to fill or copy a rectangular area within that image. As with linear fills and copies, you will need a destination address (and source address if doing a copy), but instead of a count of bytes you need the width and height of the rectangular areas affected. But you need one other thing, too. You need to tell the DMA the geometry of the over-all image... you need to tell it the width of the image containing the rectangular areas. This is called the "stride" and effectively tells the DMA how many pixels to skip between lines when it finishes one line of the rectangle before getting to the next line.

START set to trigger the DMA

INT_EN enables triggering an interrupt when DMA is complete

FILL when set, DMA will write a specific byte to memory. When clear, DMA will copy data from a source address to the destination address

2D when set, DMA copies or fills a rectangular region of memory. When clear, DMA copies or fills a certain number of sequential bytes

ENABLE set to enable DMA

FD the byte to be written to memory when FILL is set

BUSY status bit set when DMA is busy copying data

SA the 18 bit source address (must be a location in the first 256KB of RAM). Only relevant when FILL is clear.

DA the 18 bit destination address (must be a location in the first 256KB of RAM)

CNT the number of bytes to copy (only available when 2D is clear)

W the width of the rectangle of data to copy (only available when 2D is set)

H the height of the rectangle of data to copy (only available when 2D is set)

SX the width of the "stride" (only available when 2D is set)

SY the height of the "stride" (only available when 2D is set)

Address	R/W	7	6	5	4	3	2	1	0
0xDF00	R/W	START	—			INT_EN	FILL	2D	ENABLE
0xDF01	W	FD7	FD6	FD5	FD4	FD3	FD2	FD1	FD0
0xDF01	R	BUSY	—						
0xDF04	R/W	SA7	SA6	SA5	SA4	SA3	SA2	SA1	SA0
0xDF05	R/W	SA15	SA14	SA13	SA12	SA11	SA10	SA9	SA8
0xDF06	R/W	—						SA17	SA16
0xDF08	R/W	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0
0xDF09	R/W	DA15	DA14	DA13	DA12	DA11	DA10	DA9	DA8
0xDF0A	R/W	—						DA17	DA16
0xDF0C	R/W	CNT7	CNT6	CNT5	CNT4	CNT3	CNT2	CNT1	CNT0
0xDF0D	R/W	CNT15	CNT14	CNT13	CNT12	CNT11	CNT10	CNT9	CNT8
0xDF0E	R/W	—						CNT17	CNT16
0xDF0C	R/W	R/W7	R/W6	R/W5	R/W4	R/W3	R/W2	R/W1	R/W0
0xDF0D	R/W	R/W15	R/W14	R/W13	R/W12	R/W11	R/W10	R/W9	R/W8
0xDF0E	R/W	H7	H6	H5	H4	H3	H2	H1	H0
0xDF0F	R/W	H15	H14	H13	H12	H11	H10	H9	H8
0xDF10	R/W	SX7	SX6	SX5	SX4	SX3	SX2	SX1	SX0
0xDF11	R/W	SX15	SX14	SX13	SX12	SX11	SX10	SX9	SX8
0xDF12	R/W	SY7	SY6	SY5	SY4	SY3	SY2	SY1	SY0
0xDF13	R/W	SY15	SY14	SY13	SY12	SY11	SY10	SY9	SY8

Table 15.1: DMA Registers

16 System Control Registers

16.1 The Buzzer and Status LEDs

The F256jr has several software controllable LEDs. There are the SD card access LED and the power LED, but there are also two status LEDs on the board which may be controlled either manually or set to flash automatically. All the LEDs under "manual" control can be controlled by setting or clearing their relevant flags in the SYS0 register (0xD6A0) (see table: 16.1). The power LED is controlled by PWR_LED. The SD card LED is controlled by SD_LED.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD6A0	R/W	SYS0	RESET	SD_WP	SD_CD	BUZZ	L1	L0	SD_LED	PWR_LED
0xD6A1	R/W	SYS1	L1_RATE		L0_RATE		—		L1_MAN	L0_MAN

Table 16.1: System Control Registers

The two status LEDs on the board are a little more complex. They may be in manual or automatic mode. The two flags L0_MAN and L1_MAN in SYS1 control which mode they are in. If an LED's flag is clear (0), then the LED is under manual control and its equivalent flag in SYS0 controls whether the LED is on or off. If the flag is set, then the LED is set to flash automatically, and the LED's flashing rate will be set by pair of bits L0_RATE or L1_RATE according to table 16.2.

For the PC speaker, there is the BUZZ flag. By toggling BUZZ, a program can tweak the speaker and make a noise.

RATE1	RATE0	Rate
0	0	1s
0	1	0.5s
1	0	0.4s
1	1	0.2s

Table 16.2: LED Flash Rates

16.2 Software Reset

A program can trigger a system reset. This can be done by writing the value 0xDE to 0xD6A2 and the value AD to 0xD6A3 to validate that a reset is really intended (see table: 16.3), and then setting the most significant bit (RESET) of 0xD6A0 to actually trigger the reset.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD6A2	R/W	RST0	Set to 0xDE to enable software reset							
0xD6A3	R/W	RST1	Set to 0xAD to enable software reset							

Table 16.3: System Reset

16.3 Random Numbers

The F256jr has a built-in pseudo-random number generator that produces 16-bit random numbers (see table: 16.4). To use the random number generator, a program just sets the enable flag and then reads the random numbers from RNDL and RNDH (0xD6A4 and 0xD6A5). The program can set the seed value to better randomize the numbers by storing a seed value in those same locations and then toggling SEED_LD (set to load the seed value then reclear).

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD6A4	W	SEEDL	SEED7	SEED6	SEED5	SEED4	SEED3	SEED2	SEED1	SEED0
0xD6A4	R	RNDL	RND7	RND6	RND5	RND4	RND3	RND2	RND1	RND0
0xD6A5	W	SEEDH	SEED15	SEED14	SEED13	SEED12	SEED11	SEED10	SEED9	SEED8
0xD6A5	R	RNDH	RND15	RND14	RND13	RND12	RND11	RND10	RND9	RND8
0xD6A6	W	RND_CTRL	—						SEED_LD	ENABLE
0xD6A6	R	RND_STAT	DONE	—						

Table 16.4: Random Number Generator

ENABLE set to turn on the random number generator

SEED_LD set to load a value stored in SEEDL and SEEDH as the seed value for the random number generator

RNDL and RNDH read 16-bit random numbers from these registers when the random number generator is enabled

16.4 Machine ID and Version Information

Nine registers are set aside to identify the machine, the version of the printed circuit board, and the version of the FPGA. See table 16.5 for the various registers. All of the registers are read-only, and only the chip information will change over the course of the machine's life span. The machine ID contains a four-bit code that is common between all the Foenix machines (see table:).

For the F256jr, the machine ID will be 2.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD6A7	R	MID	—				ID			
0xD6A8	R	PCBID0	ASCII character 0: "A"							
0xD6A9	R	PCBID1	ASCII character 1: "0"							
0xD6AA	R	CHSV0	Chip subversion in BCD (low)							
0xD6AB	R	CHSV1	Chip subversion in BCD (high)							
0xD6AC	R	CHV0	Chip version in BCD (low)							
0xD6AD	R	CHV1	Chip version in BCD (high)							
0xD6AC	R	CHN0	Chip number in BCD (low)							
0xD6AD	R	CHN1	Chip number in BCD (high)							

Table 16.5: Machine ID and Versions

MID3	MID2	MID1	MID0	Machine
0	0	0	0	C256 FMX
0	0	0	1	C256 U
0	0	1	0	F256jr
0	0	1	1	A2560 Dev
0	1	0	0	Gen X
0	1	0	1	C256 U+
0	1	1	0	Reserved
0	1	1	1	Reserved
1	0	0	0	A2560 X
1	0	0	1	A2560 U
1	0	1	0	A2560 M
1	0	1	1	A2560 K

Table 16.6: Machine IDs

17 Memory Maps

Address	Purpose
0x00000 0x3FFFF	System RAM for programs, data, and graphics (256 KB)
0x40000 0x7FFFF	Reserved (256 KB)
0x80000 0xFFFFF	Flash memory (512 KB)

Table 17.1: System Memory Map for the F256jr

Bank	Purpose	
0	0x0000	MMU Memory Control Register
	0x0001	MMU I/O Control Register
	0x0002	RAM or Flash
	0x000F	
	0x0010	RAM, Flash, or MMU LUT Registers
	0x001F	
	0x0020	65C02 Page Zero
	0x00FF	
	0x0100	65C02 Stack
	0x01FF	
	0x0200	RAM or Flash
	0x1FFF	
1	0x2000	RAM or Flash
	0x3FFF	
2	0x4000	RAM or Flash
	0x5FFF	
3	0x6000	RAM or Flash
	0x7FFF	
4	0x8000	RAM or Flash
	0x9FFF	
5	0xA000	RAM or Flash
	0xBFFF	
6	0xC000	RAM, Flash, I/O, Text mode character, or color data
	0xDFFF	
7	0xE000	RAM or Flash
	0xFFFA	
	0xFFFA	65C02 NMI Vector
	0xFFFB	
	0xFFFC	65C02 Reset Vector
	0xFFFD	
	0xFFFE	65C02 IRQ Vector
	0xFFFF	

Table 17.2: CPU Memory Map for the F256jr

Start	End	Purpose
0xC000	0xC3FF	Gamma Table Blue
0xC400	0xC7FF	Gamma Table Green
0xC800	0xCBFF	Gamma Table Red
0xCC00	0xCFFF	Reserved
0xD000	0xD0FF	VICKY Master Control Registers
0xD100	0xD1FF	VICKY Bitmap Control Registers
0xD200	0xD2FF	VICKY Tile Control Registers
0xD300	0xD3FF	Reserved
0xD400	0xD4FF	SID Left
0xD500	0xD4FF	SID Right
0xD600	0xD60F	PSG Left
0xD610	0xD61F	PSG Right
0xD620	0xD62F	CODEC
0xD630	0xD63F	UART
0xD640	0xD64F	PS/2 Interface
0xD650	0xD65F	Timers
0xD660	0xD66F	Interrupt Controller
0xD670	0xD67F	DIP Switch
0xD680	0xD68F	IEC Controller
0xD690	0xD69F	Real Time Clock
0xD6A0	0xD6AF	System Control Registers
0xD6B0	0xD7FF	Reserved
0xD800	0xD83F	Text Foreground Color LUT
0xD840	0xD87F	Text Background Color LUT
0xD880	0xD8FF	Reserved
0xD900	0xDAFF	VICKY Sprite Control Registers
0xDB00	0xDBFF	Reserved
0xDC00	0xDCFF	65C22 VIA Control Registers
0xDD00	0xDDFF	SD Card Controller

Table 17.3: I/O Page 0 Addresses

Start	End	Purpose
0xC000	0xC7FF	Text Mode Font Memory
0xC800	0xCFFF	Reserved
0xD000	0xD3FF	Graphics Color LUT 0
0xD400	0xD7FF	Graphics Color LUT 1
0xD800	0xDBFF	Graphics Color LUT 2
0xDC00	0xDFFF	Graphics Color LUT 3

Table 17.4: Memory Map for I/O Page 1

18 Using the Debug Port

One of the ways to get software and data onto the F256jr is through the USB debug port. The debug port uses a USB serial protocol to allow a host computer to issue commands to the F256jr. These commands allow the host computer to stop and start the CPU, write to memory, read from memory, erase the flash memory, and reprogram the flash memory. With this port, it is possible to load a program and its data directly into the F256jr's memory and start it running. It is also possible to examine the F256jr's memory to see what state a program has left it in.

Three are three main tools available to provide user access to the debug port:

Foenix IDE A full featured emulator and development tool for the Foenix line of computers. Among the many tools provided by the IDE is a built-in GUI tool to upload and download data to the F256jr and program the flash. The main limitation of the IDE is that it was written in .NET and uses features that are available under under the Windows API.

Foenix Uploader Tool A stand-alone version of just the uploader tool from the Foenix IDE. This tool is more limited (it may only support binary files) and is tailored to specific machines.

FoenixMgr A script written in Python 3 which provides command line access on the host computer to the debug port. It supports files in Intel HEX, Motorola SREC, raw binary, PGX, and PGZ files. It should run on any computer or operating system that can run Python 3 and provide sufficient access to USB serial interfaces. It runs under Windows and Linux definitely and may be able to run under Mac OS X eventually.

18.1 Debug Protocol

The USB debug port is accessed over the USB Serial protocol. Data is sent from the host computer to the F256jr using data packets, each one of which is a command. The general process is:

1. Host PC sends the command to enter debug mode
2. The F256jr replies
3. Host PC sends one or more command packets
4. The F256jr replies
5. Host PC send the command to exit debug mode
6. The F256jr replies and sends a reset signal to the CPU

The commands sent from the host PC are in the form of command packets show in table 18.1. The command codes them selves are listed in table 18.3. The F256jr will respond to each command packet with a response packet as shown in table 18.2. The size of a packet can vary depending on the command. Some commands and responses include no actual data payload bytes. Others will transfer actual data and will include however many bytes of payload are needed.

Each command and response packet includes an LRC check byte, which is simply the exclusive-or of all the bytes in the packet, with the exception of the LRC value itself. This provides only rudimentary error checking, but the connection itself is generally pretty reliable, so more sophisticated error checking is really not needed.

Command sync byte This is always `0x55` and signals the start of a command packet

Command byte This byte specifies what command is being sent (see table: 18.3)

Address This is a three byte, big-endian integer that provides the address relevant to the command. For a write command, it is the address of the first block of memory to receive data. For a read command, it is the address of the first byte of memory to read. For the program flash command, it is the address of the first byte of data to write to flash.

Offset	Size	Name
0	1	Command sync byte
1	1	Command byte
2	3	Address
5	2	Length
7	n	Payload
$7+n$	1	LRC check byte

Table 18.1: USB Debug Port Command Packet

Length This is the number of bytes to transfer. For a write command, it is the number of bytes to be sent to the F256jr and will be control the size of the payload section of the write command packet. For the read command, it is the number of bytes to read from the F256jr and will control the size of the payload section of the response packet (the payload section of the read command packet is empty).

Payload This is an option section of the packet that contains the actual data to transfer between the host PC and the F256jr.

LRC check byte This byte provides for simple error checking on the packet transmission.

Offset	Size	Purpose
0	1	Response sync byte (`0xAA`)
1	2	Status bytes
3	m	Payload
$3+m$	1	LRC check byte

Table 18.2: USB Debug Port Command Packet

Response sync byte This is always `0xAA` and signals the start of a response packet

Status bytes These two bytes contain the status codes for the success or failure of the command

Payload This is an option section of the packet that contains the actual data to transfer between the host PC and the F256jr.

LRC check byte This byte provides for simple error checking on the packet transmission.

Command	Purpose
`0x80`	Enter debug mode
`0x81`	Exit debug mode (resets CPU)
`0x00`	Read a block of data from the F256jr to the host PC
`0x01`	Write a block of data to RAM on the F256jr
`0x10`	Program flash memory from data in F256jr's RAM
`0x11`	Erase flash memory
`0x12`	Erase flash sector
`0x13`	Program flash sector
`0xFE`	Fetch the revision number of the debug interface

Table 18.3: USB Debug Port Commands

18.2 Flash Sectors

Individual blocks or sectors of flash may be erased or programmed without affecting the rest of flash memory. This can be done through the commands `0x12` to erase flash sectors and `0x13` to program them from RAM. The packets for sectors are a little different from the others. The main difference is that third byte of the packet (ordinarily the high byte of the address) is the number of the sector to program, and addresses are limited to 16-bits. Each sector is a 4KB block, with 0 being the first 4KB of flash, 1 being the second 4KB of flash, and so on.

The flash of the F256jr has a limitation that the smallest block of flash that can be erased is 8KB, so when erasing sectors, two sectors must be erased, not just one. And the sector pairs must be aligned to 8KB. So sector 0 and sector 1 would be erased together, but not sector 1 and sector 2 (although sectors 0 – 3 would be fine).

Programming flash sectors has no such limitation (it is fine to flash just a 4KB block). However, for simplicity's sake, it would probably be best for any program directly accessing the debug port to limit erasing and programming to 8KB blocks. Programming the flash sectors does have a limitation: since the address is limited to 16-bits, the data can only be stored in the first 64KB of the 256KB system RAM.