Foenix F256jr Reference Manual

Peter Weingartner

17-02-2023

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.

Several of chapters in this manual include example assembly code to show how the various features

of the F256jr work. While the code included in the text should be executable, the complete examples can be found on the Github repository that hosts the manual itself. Most of the examples are able to run on their own, but a few of them expect there to be some sort of operating system providing text display routines compatible with the old Commodore kernel. The examples were all written on a machine using OpenKERNAL, which was written for the F256jr, but really anything that provides the CINT and CHROUT calls should work fine. Of course, the examples could be tweaked without too much trouble to run on essentially any operating system.

A Note on Notation

Important side notes are called out with a black bar in the margin. These notes generally call out key differences between the different versions of the F256jr and may have some important considerations for programs that need to run on all versions.

Example code in this manual is presented as assembly language. While the code is fairly generic 65C02 assembly code, the code was tested using the 64TASS assembler, and there points of 64TASS syntax that are worth explaining.

- Numeric literals are in decimal unless prefixed by the dollar sign ($)
- There are several math and logical operators that can be used to calculate a numeric literal value at assembly time: + adds numbers, – subtracts, ∗ multiplies, | calculates the bitwise OR of two values, and ~ computes the bitwise NOT or negation of a value.
- The `.byte` directive stores a set of 8-bit values into the assembled code. The directive takes a list of values puts them into the assembled code as bytes.

- The `.word` directive stores a set of 16-bit values into the assembled code. The directive takes a list of values and treats them as 16-bit (even if they are less than 256).
- There are special operators for selecting specific bit ranges from a number to allow the assembly code to write the number byte-by-byte. The less than character (<) selects bits 0–7. The greater than character (>) selects bits 8–15. The back tick character (‘) selects bits 16–23. For example:

```
SAMPLE = $123456

lda #<SAMPLE
sta $0800           ; Store $56 to $0800
lda #>SAMPLE
sta $0801           ; Store $34 to $0801
lda #‘SAMPLE
sta $0802           ; Store $12 to $0802
```

About the Machine

Ports

The connectors of the back of the F256jr from left to right are (see figure: 1.1):

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.

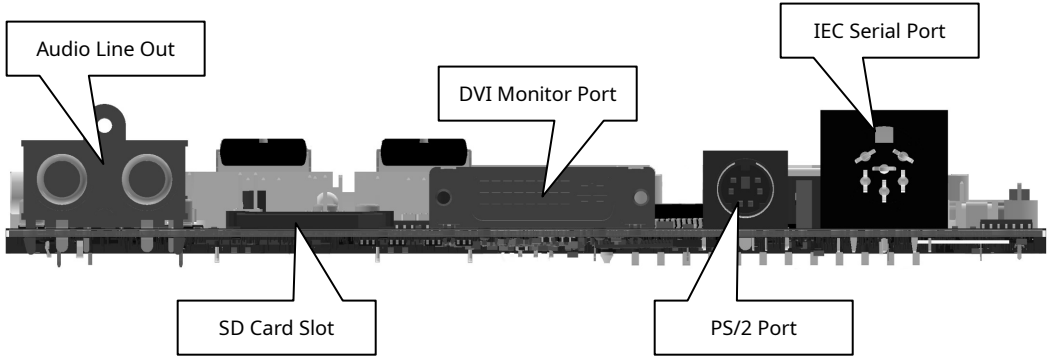

Figure 1.1: F256jr Rear Connectors

The top of the board has several connectors and other features that should be explained (see figure: 1.2):

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 allow you to plug in Atari style joysticks

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.

System Architecture

For being so small, the F256jr has a lot of components to it, so it is worth mapping out the over all structure of the computer. One of the main things to note is that most of what makes the F256jr the F256jr is the FPGA TinyVicky. TinyVicky provides the MMU, the various text and graphics engines, most of the I/O devices, controllers for the sound chips, and the controller for the first 256KB of SRAM. The CPU,

VIA, RTC, flash memory, and expansion RAM are separate from TinyVicky, although TinyVicky is still responsible for translating CPU addresses to the appropriate chip selection logic and bank selection. One of the most important aspects of this architecture is that, while the first 256KB of SRAM is accessible to both the CPU and TinyVicky, TinyVicky cannot access the data in the flash or in any expansion RAM.

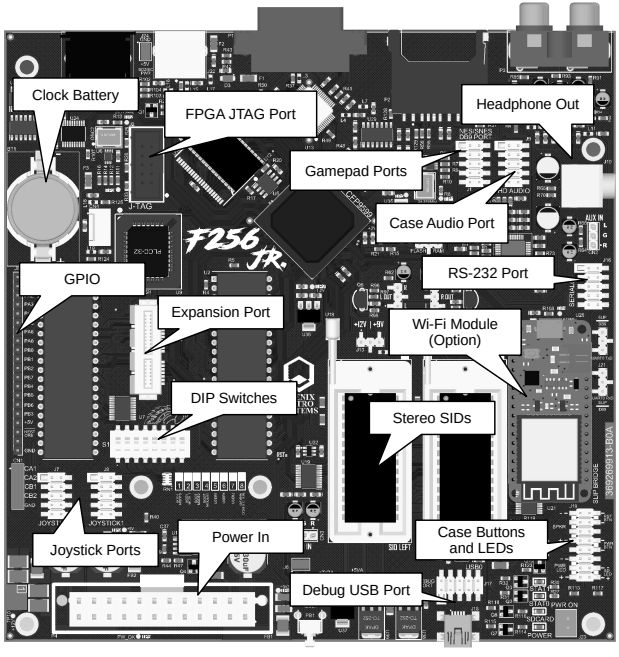

Figure 1.2: F256jr Top View

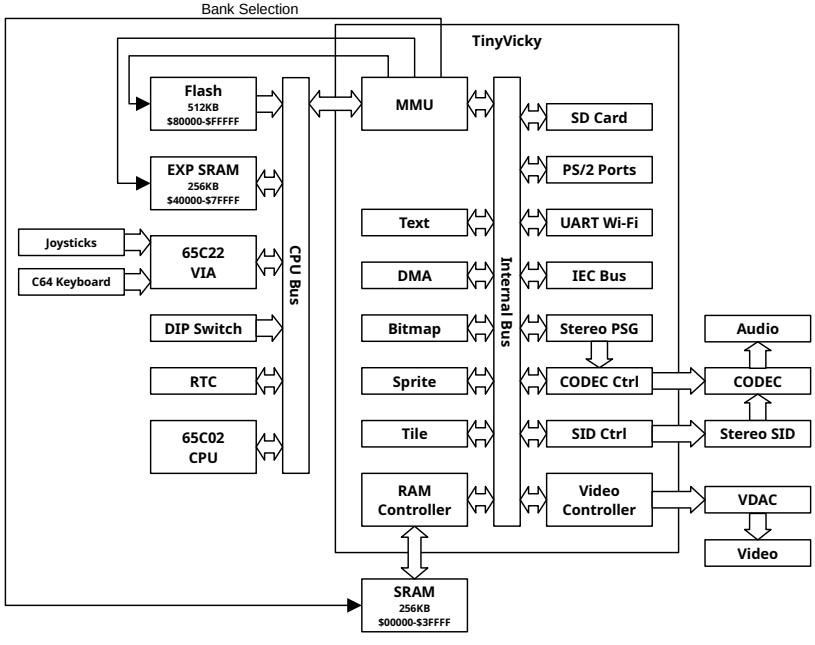

Figure 1.3: F256jr Internal Architecture

2 Memory Management

The F256jr has 512 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. Finally, the F256jr comes with an expansion port and allows for 256KB of expansion RAM to be added. 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 1,280 KB of total storage on the system, the system has a 21-bit address

bus to manage the memory. RAM and flash have address on that 21-bit bus as shown in table 2.1.

Start	End	Memory Type
0x000000	0x07FFFF	System RAM (512 KB)
0x080000	0x0FFFFF	Flash Memory (512 KB)
0x100000	0x13FFFF	Expansion RAM (256 KB)

Table 2.1: F256jr 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 21 bits, the upper 8 bits are used to address the correct 8 KB bank in the full system memory. So a MLUT must provide a 8-bit system bank number for each 3-bit bank number provided by the CPU. Figure 2.1 shows the translation of a CPU address to a full system address through the currently selected memory MLUT.

NOTE: The original RevA developer’s board has a different memory map from the RevB. The RevA does not include an expansion slot and has only 256KB of system RAM. See table 22.2 for the overall memory map of the RevA board. Additionally, the RevA bus is only 20-bits wide, and the memory MLUTs are only 7 bits wide.

The F256jr's MMU supports up to four MLUTs, 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 MLUT 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 2.2: 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 2.3).

The MMU is controlled through two main registers, which are always at locations 0x0000 and 0x0001 in the CPU's address space (see table 2.4). These registers allow programs to select an active

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

Table 2.3: I/O Banks

MLUT, edit a MLUT, and control bank 6:

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 2.4: MMU Registers

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

EDIT_EN if set (1), this bit allows a MLUT to be edited by the program, and memory addresses 0x0008–0x0010 will be used by the MLUT 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 MLUT (0 - 3) is being edited and will appear in memory addresses 0x0008–0x0010.

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.

Example: Setting up a MLUT

In this example, we will set up MLUT 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 MLUT = 0, Edit MLUT#1
    sta $0000

    ldx #0        ; Start at bank 0
l1: txa           ; First 6 banks will just be the first banks of RAM
    sta $0008,x   ; Set the MLUT 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 $000f

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

lda #$01      ; Turn off MLUT editing, and switch to MLUT#1
sta $0000
```

MMU Boot Configuration

While the MMU registers allow the MMU to select one of four memory MLUTs to be used for address translation or to be edited, in fact the F256jr's MMU actually has eight MLUTs in two sets of four. At any given time, only one of those sets of four MLUTs is active. One set of MLUTs is the "boot from RAM" set, and the other is the "boot from flash" set. As the names imply, one set is meant to allow you to boot the F256jr to run code you have loaded into RAM (useful for development and debugging), while the other is meant to be used to boot up an operating system you have loaded into flash memory (useful for just running programs and playing games).

When the F256jr powers on, it initializes the MLUTs in two different ways. The "boot from RAM" MLUTs are initialized so the 64KB of CPU address space is simply mapped to the first 64KB of system RAM. The "boot from flash" MLUTs are initialized to be the same, except that the last bank of CPU address space (0xE000 – 0xFFFF) is mapped to the last bank of flash memory (0x07E000 – 0x07FFFF). After the MLUTs are initialized, the F256jr checks to see which of the two sets of MLUTs should be used and enables them. The memory MLUTs that are not selected are completely ignored. See figure 2.2 to

see how the MLUTs are related and how they are initialized on power up.

How the F256jr decides which set of MLUTs to use depends upon the board. The older, RevA boards have a command available over the USB debug port that switches the active MLUTs on the fly. The newer, RevB boards have a jumper to choose between the MLUTs.

> NOTE: the memory MLUTs are really just tables stored in RAM in the TinyVicky chip, and apart from the power-up initialization, TinyVicky does not change the MLUTs except when directed by a program. Pressing the RESET button does not re-initialize the MLUTs. This means that a program should not assume the MLUTs are set to any particular value on reset, unless the operating system is initializing the MLUTs. A program running as an operating system or even just taking complete control over the board should always initialize the MLUTs to the values it needs as one of its first tasks. Of course, a complete power cycle of the board will reset the MLUTs, but a program will not always be starting from a complete power cycle.

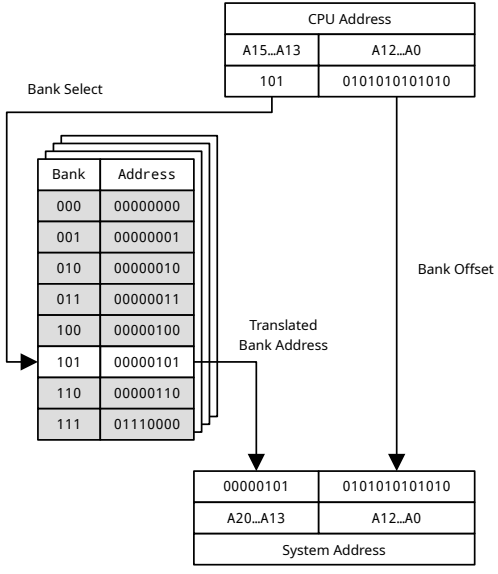

Figure 2.1: MMU Address Translation

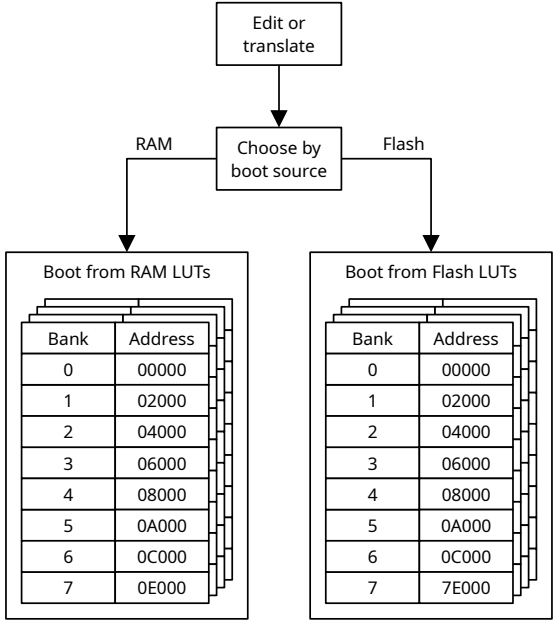

Figure 2.2: MMU Boot Configuration

3 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 overlaid on top of the graphics modes.

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 may 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.

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.

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.

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.

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
```

Entering Text Mode

Whether text mode is being displayed (and in what resolution) is controlled by the VICKY Master Control Registers (see table 3.2). 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.

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

OVRLY if set, text will be overlaid 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)

MON_SLP if set, the monitor SYNC signal will be turned off, putting the monitor to sleep

FON_OVLY if clear (0), only the text foreground color will be displayed when text overlays graphics (all background colors will be completely transparent). If set (1), both foreground and background colors will be displayed, except that background color 0 will be transparent.

FON_SET if set (1), the text font displayed will be font set 1. If clear (0), the text font displayed will be font set 0.

NOTE: The F256jr RevA board does not support the MON_SLP or FON_OVLY register bits or functions.

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. TinyVicky provides for two font sets, and which one is used for text mode is controlled by the FON_SET bit in the Vicky Master Control Register. Only one will be in use at any given time in normal operation. Font set 0 is from 0xC000 through 0xC7FF. Font set 1 is from 0xC800 through 0xCFFF.

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:

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.

Text Cursor

F256jr has a text mode cursor. The text mode cursor is implemented as a character which is displayed in a (x, y) position on the screen, visually replacing the character ordinarily at that position. It may be displayed continuously, or it may flash at one of four rates. When flashing, that position in the text screen will alternate between the text cursor and the character at that position in the text matrix. The color for the text cursor comes from the color for the position on the screen as specified in the color matrix. In other words, the text cursor does not have its own color.

ENABLE if this flag is set (1), the cursor is enabled

FLASH_DIS if this flag is set (1), the cursor will not flash. If clear (0), it will flash.

RATE these two bits set the rate at which the cursor flashes (see table 3.5)

CCH the character code for the cursor character to display

CURX the column number (16-bit) for the cursor

CURY the row number (16-bit) for the cursor

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 3.1: Text Color Lookup Tables

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	—		FON_SET	FON_OVLY	MON_SLP	DBL_Y	DBL_X	CLK_70

Table 3.2: VICKY Master Control Registers

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

Table 3.3: A sample character

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD010	R/W	CCR	—				FLASH_DIS	RATE		ENABLE
0xD012	R/W	CCH	Cursor character							
0xD014	R/W	CURX	X7	X6	X5	X4	X3	X2	X1	X0
0xD015	R/W		X15	X14	X13	X12	X11	X10	X9	X8
0xD016	R/W	CURY	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0xD017	R/W		Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8

Table 3.4: Text Cursor Registers

RATE1	RATE0	Rate
0	0	1s
0	1	1/2s
1	0	1/4s
1	1	1/5s

Table 3.5: Text Cursor Flash Rates

4 Graphics

The F256jr provides three separate graphics engines, giving programs 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 graphic 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).

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: 4.1). 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 1 (see table: 4.1).

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 4.1: Graphics Color Lookup Tables

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.

```
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:
```

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 size is $w \times h$, the position of a pixel at (x, y) in the list is $y \times w + x$.

> NOTE: Pixel data for all graphics modes can be stored anywhere in the initial system RAM. For the F256jr RevB board and the F256K, this means the pixel data can be stored anywhere in the first 512KB of system address space. For the F256jr RevA boards, however, pixel data may only be stored in the first 256KB of system address space, since the RevA boards were built with only 256KB of system RAM.

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, see table 7.3).

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: 4.2).

Bitmaps and tile maps are assigned to their layers using the layer control registers (see table: 4.2). 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: 4.3).

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

Table 4.2: Bitmap and Tile Map Layer Registers

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 #$40            ; Layer 0 = BM 0, Layer 1 = TM 0
sta VKY_LAYER_CTRL_0
```

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 4.3: Bitmap and Tile Map Layer Codes

```
lda #$01             ; Layer 2 = BM 1
sta VKY_LAYER_CTRL_1
```

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		—					AD18	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		—					AD18	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		—					AD18	AD17	AD16

Table 4.4: Bitmap Registers

ENABLE if set and both graphics and bitmaps are enabled in the Vicky Master Control Register (see table 3.2), 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: 3.2). 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: 4.2).
5. Set the bitmap's CLUT and ENABLE bit in its control register.

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 regi
```

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
```

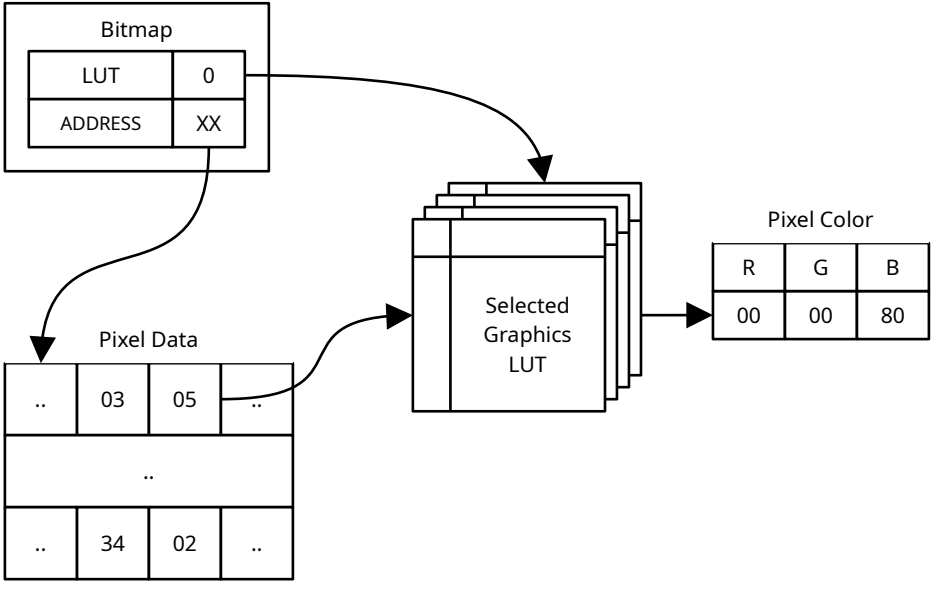

Figure 4.1: Bitmap Data to Pixels

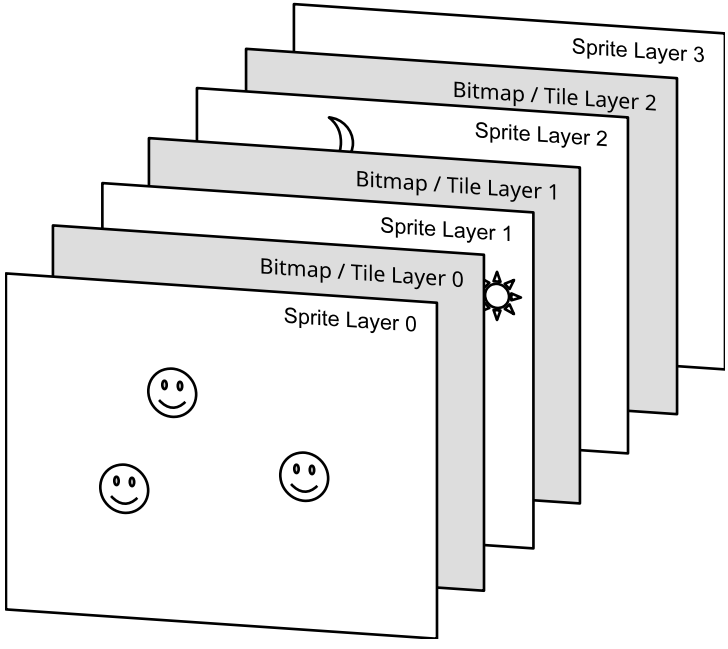

Figure 4.2: TinyVicky Graphic Layers

5 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 5.1.

Offset	R/W	7	6	5	4	3	2	1	0
0	W	—	SIZE		LAYER		LUT		ENABLE
1	W	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
2	W	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
3	W	—					AD18	AD17	AD16
4	W	X7	X6	X5	X4	X3	X2	X1	X0
5	W	X15	X14	X13	X12	X11	X10	X9	X8
6	W	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
7	W	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8

Table 5.1: 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 5.2)

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 5.2: Sprite Sizes

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.

Sprite Positioning

The coordinate system for sprites is similar to that for bitmap graphics, but it is offset by 32 pixels in both the horizontal and vertical directions. There is a sort of margin area around the entire displayed screen that a sprite can be in and be either partially or completely hidden from view. The horizontal coordinate for a sprite ranges from 0 to 352. The vertical coordinate can range from 0 to 232 or 272, depending on the vertical resolution. For a sprite to have its top-left corner in the top-left of the screen, its position would need to be $(32, 32)$. This coordinate system is the same for all sprites, regardless of their size. Figure: 5.1 shows how the coordinate system is arranged.

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
            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 $0, $0, $0, $0, $0, $0, $3, $2, $2, $1, $0, $0, $0, $0, $0, $0
.byte $0, $0, $0, $0, $5, $5, $4, $3, $3, $3, $3, $2, $0, $0, $0, $0
.byte $0, $0, $0, $7, $7, $7, $6, $5, $4, $4, $3, $3, $1, $0, $0, $0
.byte $0, $0, $7, $9, $A, $B, $A, $8, $6, $5, $4, $3, $2, $1, $0, $0
.byte $0, $5, $7, $A, $D, $E, $D, $A, $7, $5, $5, $4, $3, $1, $1, $0
.byte $0, $5, $7, $B, $E, $E, $E, $C, $7, $5, $5, $4, $3, $1, $1, $0
.byte $3, $4, $6, $A, $D, $E, $D, $A, $7, $5, $5, $4, $3, $2, $1, $1
.byte $2, $3, $5, $8, $A, $C, $A, $8, $6, $5, $5, $4, $3, $2, $1, $1
.byte $2, $3, $4, $6, $7, $7, $7, $6, $5, $5, $5, $4, $3, $1, $1, $1
.byte $1, $3, $4, $5, $5, $5, $5, $5, $5, $5, $5, $3, $3, $1, $1, $1
.byte $0, $3, $3, $4, $5, $5, $5, $5, $5, $5, $4, $3, $2, $1, $1, $0
```

```
.byte $0, $2, $3, $3, $4, $4, $4, $4, $4, $3, $3, $2, $1, $1, $1, $0
.byte $0, $0, $1, $2, $3, $3, $3, $3, $3, $3, $2, $1, $1, $1, $0, $0
.byte $0, $0, $0, $1, $1, $1, $2, $2, $1, $1, $1, $1, $1, $0, $0, $0
.byte $0, $0, $0, $0, $1, $1, $1, $1, $1, $1, $1, $1, $0, $0, $0, $0
.byte $0, $0, $0, $0, $0, $0, $1, $1, $1, $1, $0, $0, $0, $0, $0, $0
```

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
```

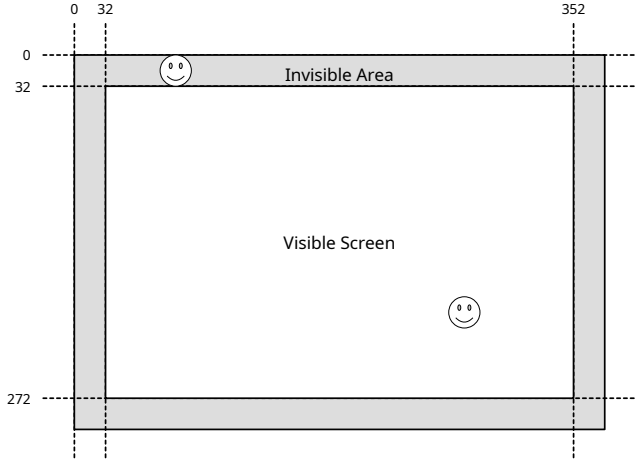

Figure 5.1: Sprite Positions

6 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:

$$\begin{aligned} \text{character} &\approx \text{tile} \\ \text{font} &\approx \text{tile set} \\ \text{text matrix} &\approx \text{tile map} \end{aligned}$$

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. 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.

Tile Maps

There are three tile maps supported by TinyVicky, each of which has 12 bytes worth of registers (see table: 6.1). Tile map 0 starts at 0xD200. Tile map 1 starts at 0xD20C. Tile map 2 starts at 0xD218.

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	—					AD18	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
11	W	DIR_Y	—			Y7	Y6	Y5	Y4

Table 6.1: Tile Map Registers

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

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: 6.2), 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 6.2: A Tile Map Entry

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.

Tile Sets

Essentially, a tile set is just a bitmap, but of a smaller size and arranged in a specific pattern. A tile set can be either a linear arrangement of tiles or a square arrangement of tiles. In the linear arrangement, the image is one tile wide by 256 tiles high. So for 8×8 tiles, the tile set is 8 pixels wide by 2,048 pixels high. For 16×16 tiles, the tile set is 16 pixels wide by 4,096 pixels high. The tiles are arranged vertically, so the first 8 or 16 (depending on tile size) rows are tile 0, the second set of rows are tile 1, and so on.

For the square arrangement, the tile set is either 128 pixels wide by 128 pixels high (for 8×8 tiles), or it is 256 pixels wide by 256 pixels high (for 16×16 tiles). In both cases, the tiles are laid out left to right and top to bottom in a grid that is 16 tiles wide by 16 tiles high (see table 6.3).

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.

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: 6.4). 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).

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.

```
;
; 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,$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,$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,$0,$9,$1,$2,$3,$4,$5,$0,$0,$0,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$0,$9,$2,$1,$2,$3,$4,$5,$0,$0,$0,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$0,$9,$3,$2,$1,$2,$3,$4,$5,$0,$0,$A,$0,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$0,$9,$4,$3,$2,$1,$2,$3,$4,$5,$0,$0,$A,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$0,$9,$5,$4,$3,$2,$1,$2,$3,$4,$0,$0,$A,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$0,$9,$0,$5,$4,$3,$2,$1,$2,$3,$0,$0,$A,$0,$0,$0,$0,$0
.word $0,$0,$0,$0,$0,$0,$0,$9,$0,$0,$5,$4,$3,$2,$1,$2,$0,$0,$A,$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,$C,$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,$0,$2,$0,$2,$0
.word $0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$0,$2,$0,$0,$4
```

	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 6.3: Arrangement of Tiles in a Tile Set Image

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		—					AD18	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		—					AD18	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		—					AD18	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		—					AD18	AD17	AD16
0xD28F	W		—				SQUARE	—		

Table 6.4: Tile Set 0–3 Registers

Address	R/W	Tile Set	7	6	5	4	3	2	1	0
0xD290	W	4	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD291	W		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD292	W		—					AD18	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		—					AD18	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		—					AD18	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		—					AD18	AD17	AD16
0xD29F	W		—				SQUARE	—		

Table 6.5: Tile Set Registers 4–7

7 Miscellaneous Features of TinyVicky

DIP Switches

F256jr has eight DIP switches on the board, which can be used to configure various options. The DIP switches are present on a single register (see table: 7.1). The DIP switches ground their signals when placed in their “on” positions. So a true or asserted value is 0, while the false or de-asserted value is 1.

There are five fields of switches:

GAMMA this is a dedicated switch to indicate if gamma correction should be turned on (0) or not (1)

Address	R/W	7	6	5	4	3	2	1	0
0xD670	R	GAMMA	USER2	USER1	USER0	BOOT			

Table 7.1: DIP Switch Register

USER0, USER1, USER2 these three switches are reserved for use by the operating system or programs. On is 0, off is 1.

BOOT these four switches provide information to the operating system for boot options.

The Border

The F256jr's display can have a border, which overlays all the other display elements. The border can have any color which TinyVicky can display, and can have a width from 0 to 31 pixels. The border can also be turned off, leaving the full display for graphics or text.

When using graphics modes, the border simply hides the graphics elements underneath it. For text mode, things are a little different. The text display will be shifted so that the character at $(0, 0)$ is still the upper-left character. The layout of the text and color matrixes do not change, however. Cells that are under the right side or bottom of the border will still be in the matrixes but will not be displayed. Another way to put it is that, if the text resolution is 80 characters wide, it will remain 80 characters per line even if the border is on and only 76 characters are displayed.

ENABLE when set (1), the border will be displayed

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD004	R/W	BRDR_CTRL	—	SCROLL_X			—			ENABLE
0xD005	R/W	BRDR_BLUE	Blue component of border color							
0xD006	R/W	BRDR_GREEN	Green component of border color							
0xD007	R/W	BRDR_RED	Red component of border color							
0xD008	R/W	BRDR_WIDTH	—			SIZE_X				
0xD009	R/W	BRDR_HEIGHT	—			SIZE_Y				

Table 7.2: Border Registers

SCROLL_X the number of pixels the border should be shifted in the horizontal direction

BRDR_BLUE the amount of blue in the border (0 = none, 255 = maximum amount)

BRDR_GREEN the amount of green in the border (0 = none, 255 = maximum amount)

BRDR_RED the amount of red in the border (0 = none, 255 = maximum amount)

SIZE_X the width of the left and right sides of the border in pixels (from 0 to 31)

SIZE_Y the height of top and bottom of the border in pixels (from 0 to 31)

Background Color

In text mode, the background color is determined by the color matrix and the text color LUTs. For the graphics modes, however, a background color is specified separately. There are three registers to specify the background color's red, green, and blue components (see table: 7.3). This is the color that will be displayed in graphics modes, if all the layers specify that a given pixel has the color 0 (which is always the transparent pixel color).

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD00D	R/W	BGND_BLUE	Blue component of background color							
0xD00E	R/W	BGND_GREEN	Green component of background color							
0xD00F	R/W	BGND_RED	Red component of background color							

Table 7.3: Background Color Registers

Line Interrupt and Beam Position

TinyVicky can trigger a SOL interrupt (see table: 9.2) when the display has reached a given raster line. This can be useful for split-screen style effects or other programming tricks that work off of partitioning the screen into separate areas. To use this feature, a program would enable the line interrupt and set a register to the number of the line on the screen when the interrupt should be triggered. In addition to setting a line interrupt, there are two 12-bit registers that allow the program to see what line

and column is TinyVicky is currently drawing. The addresses for all these registers overlap. The line interrupt registers are write-only, and the current beam position registers are read only (see table: 7.4)

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD018	W	LINT_CTRL	—							ENABLE
0xD019	W	LINT_L	L7	L6	L5	L4	L3	L2	L1	L0
0xD01A	W		—				L11	L10	L9	L8
0xD01B	W	—	Reserved							
0xD018	R	RAST_COL	X7	X6	X5	X4	X3	X2	X1	X0
0xD019	R		—				X11	X10	X9	X8
0xD01A	R	RAST_ROW	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0xD01B	R		—				Y11	Y10	Y9	Y8

Table 7.4: Line Interrupt and Beam Position Registers

ENABLE if set (1), TinyVicky will trigger line interrupts (write only)

LINT_L the line number (12 bits) on which to trigger the next line interrupt (write only). The top of the display is line 0, and the bottom of the screen is 400 for 320×200 mode, and 480 for 320×240 mode.

RAST_COL the number (12 bits) of the current pixel being drawn (read only)

RAST_ROW the number (12 bits) of the current line being drawn (read only)

Example: Changing Border with the Line

In this example, we will play with a split-screen style effect changing the color of the border so that the top and bottom borders are blue while the left and right borders are red. To do this, we will use the line interrupt twice for each frame: once when we are on the line just below the last line of the top border, and once when we are on the first line of the bottom border.

To make this work, the example has a single `state` variable, which will track which color border is being rendered. It will enable the line interrupt, and then set the line number to wait for. When that interrupt comes in, it will check the state variable, setting the border color and new line number based on `state`. It will also flip `state` to the other value (0 or 1).

```
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
INT01_VKY_SOL = $02

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_BRDR_COL_B = $D005          ; Vicky Border Color -- Blue
VKY_BRDR_COL_G = $D006          ; Vicky Border Color -- Green
VKY_BRDR_COL_R = $D007          ; Vicky Border Color -- Red
```

```
VKY_BRDR_VERT = $D008           ; Vicky Border vertical thickness in pixels
VKY_BRDR_HORI = $D009           ; Vicky Border Horizontal Thickness in pixels

VIRQ = $FFFE

LINE0 = 16                      ; Start at line 16 (first line on the text display)
LINE1 = 480 - 16                ; End on line 464 (last line of text display)

;
; Variables
;
* = $0080

state       .byte ?             ; Variable to track which color we should use

;
; Code
;
* = $e000

start:      ; Disable IRQ handling
            sei

            ; Go back to I/O page 0
```

```
stz MMU_IO_CTRL

; 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 SOL interrupt
lda #$ff
sta INT_MASK_1
and #~INT01_VKY_SOL
sta INT_MASK_0

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

; Make sure we're in text mode
```

```
lda #$01                ; enable TEXT
sta VKY_MSTR_CTRL_0     ; Save that to VICKY master control register 0
stz VKY_MSTR_CTRL_1

; Set the border
lda #$01                ; Enable the border
sta VKY_BRDR_CTRL

lda #16                 ; Make it 16 pixels wide
sta VKY_BRDR_VERT
sta VKY_BRDR_HORI

lda #$80                ; Make it cyan to start with
sta VKY_BRDR_COL_B
sta VKY_BRDR_COL_G
stz VKY_BRDR_COL_R

lda #$01                ; Turn on the line interrupt
sta VKY_LINE_CTRL

lda #<LINE0             ; set the line to interrupt on
sta VKY_LINE_NBR_L
lda #>LINE0
sta VKY_LINE_NBR_H
```

```
            stz state               ; Start in state 0

            ; Re-enable IRQ handling
            cli

loop:       ; Just loop forever... a real program will do stuff here
            nop
            bra loop

;
; A simple interrupt handler
;
my_handler: .proc
            pha

            ; Save the system control register
            lda MMU_IO_CTRL
            pha

            ; Switch to I/O page 0
            stz MMU_IO_CTRL

            ; Check for SOL flag
```

```
            lda #INT01_VKY_SOL
            bit INT_PEND_0
            beq return              ; If it's zero, exit the handler

            ; Yes: clear the flag for SOL
            sta INT_PEND_0

            lda state               ; Check the state
            beq is_zero

            stz state               ; If state 1: Set the state to 0

            lda #<LINE0             ; Set the line to interrupt on
            sta VKY_LINE_NBR_L
            lda #>LINE0
            sta VKY_LINE_NBR_H

            lda #$80                ; Make the border blue
            sta VKY_BRDR_COL_B
            stz VKY_BRDR_COL_G
            stz VKY_BRDR_COL_R
            bra return

is_zero:    lda #$01                ; Set the state to 1
```

```
            sta state

            lda #<LINE1             ; set the line to interrupt on
            sta VKY_LINE_NBR_L
            lda #>LINE1
            sta VKY_LINE_NBR_H

            lda #$80                ; Make the border red
            sta VKY_BRDR_COL_R
            stz VKY_BRDR_COL_G
            stz VKY_BRDR_COL_B

            ; Restore the system control register
return:     pla
            sta MMU_IO_CTRL

            ; Return to the original code
            pla
            rti
            .pend
```

Gamma Correction

TinyVicky has the ability to apply gamma correction to the video signal. This allows users to adjust their images to match their monitors. Activating gamma correction is done by setting the GAMMA flag in the Vicky master control register (see table: 3.2). When enabled, colors will be adjusted through the gamma look up tables. There are three tables: blue is at 0xC000, green is at 0xC400, and red is at 0xC800.

The way that the gamma look up tables work is very straight forward. When drawing a pixel, the separate color components are used as indexes into their respective gamma LUTs, and the value in the LUT is used as the new component value. For instance, if a pixel's color is (r, g, b), then the new color is:

```
r_corrected = gamma_red[r]
g_corrected = gamma_green[g]
b_corrected = gamma_blue[b]
```

On power up, TinyVicky sets up a default gamma correction of 1.8, although software (either the user's program or the operating system) has to turn on gamma correction to use it.

Bitmap Coordinate Math Block

The calculation of the address for a particular pixel can be somewhat involved, since it involves not just calculating the absolute address but also what MMU bank the pixel is in. To help with this, the F256jr includes a special purpose math block for calculating the address of a pixel, its MMU bank, and

the offset within the MMU bank, given the starting address of a bitmap and the (x, y) coordinates for the pixel. The block is made up of six registers—three inputs and three outputs (see table 7.5).

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD300	W		Reserved							
0xD301	W	XY_BASE	BA7	BA6	BA5	BA4	BA3	BA2	BA1	BA0
0xD302	W		BA15	BA14	BA13	BA12	BA11	BA10	BA9	BA8
0xD303	W		—						BA17	BA16
0xD304	R/W	XY_POSX	X7	X6	X5	X4	X3	X2	X1	X0
0xD305	R/W		X15	X14	X13	X12	X11	X10	X9	X8
0xD306	R/W	XY_POSY	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0xD307	R/W		Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8
0xD308	R	XY_OFFS	O7	O6	O5	O4	O3	O2	O1	O0
0xD309	R		O15	O14	O13	O12	O11	O10	O9	O8
0xD30A	R	XY_BANK	B7	B6	B5	B4	B3	B2	B1	B0
0xD30B	R	XY_AD	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
0xD30C	R		AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8
0xD30D	R		—						AD17	AD16

Table 7.5: Bitmap Coordinate Calculator

The calculation performed is simply:

$$\text{XY\_AD} = (\text{XY\_BASE}) + (\text{XY\_POSY} \times 320) + (\text{XY\_POSX})$$

where XY_BASE is the address of the pixel at $(0, 0)$, XY_POSY is the y coordinate of the pixel, XY_POSX is the x, coordinate of the pixel, and XY_AD is the system address of the pixel. The block then breaks that address down into the MMU bank number for that address (XY_BANK) and the offset to the pixel in that bank (XY_OFFS).

An example of

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 “PSG” 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.

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 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 shown in table 8.

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
```

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.

There are four basic formats of bytes that can be written to the port, as shown in table 8.

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. There is also a sound “device” for managing the left and right PSGs together, which starts at 0xD608. The combined registers work in the same way as the left and right PSGs. Writing to the combined registers is equivalent to writing to the left and right channel registers simultaneously.

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

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.

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 rela-

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 six 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

tionship 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 6 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
```

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
```

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).

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 (which is controlled by VOL3-0). 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

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.

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.

NOTE: Voice 3 can be muted by setting MUTEV3. This is useful to have the wave forms generated by voice 3 be used for ring modulation and synchronization without having voice 3's wave forms being actually audible.

Filtering

The SID chip can apply a filter to the audio before sending it out for amplification. The filter works at an adjustable frequency and may be used as either a high-pass filter (if HIGH is set), a low-pass filter (if LOW is set), or as a band-pass filter (if BAND is set). The filter frequency is set by the bits FC0-10.

The filter may be applied or not to each voice independently. Bits FILTV1, FILTV2, and FILTV3 control whether the filter is applied to voices 1, 2, and 3 respectively. Finally, a resonance effect may be tuned on the filter using the RES0-3 bits: 0 indicates no resonance, and 15 indicates maximum resonance.

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

Table 8.5: SID V1 and V2 Registers

Voice	Offset	R/W	7	6	5	4	3	2	1	0
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 8.6: SID V3 and Miscellaneous Registers

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 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.

Each interrupt that the interrupt controller manages belongs to one of three separate groups. Each group manages at most eight interrupts, and each interrupt within that group has its own bit within the group. That bit is used in the four registers that control the interrupts for that group (see table 9.1). The four different registers for each group are:

PENDING In this register, there are eight flags, one for each interrupt in the group. When reading the register, if the flag is set, the interrupt controller has received that interrupt. When writing to the register, setting a flag will clear the pending status of the interrupt.

POLARITY This register, together with EDGE, controls how the interrupt controller interprets the inputs to recognize an interrupt condition (see table 9.5).

EDGE This register, together with POLARITY, controls how the interrupt controller interprets the inputs to recognize an interrupt condition (see table 9.5).

MASK This register controls whether interrupts asserted by the devices will trigger an IRQ. If an interrupt's flag is set in the MASK register, then the interrupt will be ignored. If the flag is clear, the interrupt being asserted by the device will trigger an IRQ on the processor.

The interrupt controller registers are divided on the F256jr into three groups: 0, 1, and 2. Group 0 represents seven of the interrupts: two video interrupts, two PS/2 controller interrupts, two timer interrupts, and the DMA interrupt. Group 1 represents the other interrupts: UART, real time clock, VIA, and the SD card controller. Group 2 represents interrupts used by the IEC serial port. See tables 9.2, 9.3, and 9.4 to see how device interrupts are assigned to their groups.

Group	Address	Name
0	0xD660	INT_PENDING_0
	0xD664	INT_POLARITY_0
	0xD668	INT_EDGE_0
	0xD66C	INT_MASK_0
1	0xD661	INT_PENDING_1
	0xD665	INT_POLARITY_1
	0xD669	INT_EDGE_1
	0xD66D	INT_MASK_1
2	0xD662	INT_PENDING_2
	0xD666	INT_POLARITY_2
	0xD66A	INT_EDGE_2
	0xD66E	INT_MASK_2

Table 9.1: Interrupt Registers

NOTE: Some 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.

Bit	Name	Purpose
0x01	INT_VKY_SOF	TinyVicky Start Of Frame interrupt.
0x02	INT_VKY_SOL	TinyVicky Start Of Line interrupt
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	RESERVED	
0x80	Cartridge	Interrupt asserted by the cartidge

Table 9.2: Interrupt Group 0 Bit Assignments

The Start Of Frame (SOF) and Start of Line (SOL) interrupts could use some further explanation. The SOF interrupt is raised at the beginning of the vertical blanking period, when the raster has reached the bottom of the screen and starts to return to the top. This interrupt is raised either 60 times a second or 70 times a second, depending on the value of CLK_70 (see table 3.2), which sets the base resolution of the screen. The SOF interrupt is good for timing updates to graphics (like placement of sprites) to avoid screen tearing. It can also be used for rough timing of events, provided the code takes into account the fact that the timing changes with screen resolution. The Start of Line interrupt is raised when the raster line has reached a target line (see table 7.4). When the interrupt is raised, the raster is in the process of drawing the screen and has reached the desired target line.

As an example of working with the interrupt controller, let's try using the SOF interrupt to alter the

Bit	Name	Purpose
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_VIA0	Event from the 65C22 VIA chip
0x40	INT_VIA1	F256K Only: Local keyboard
0x80	INT_SDC_INS	User has inserted an SD card

Table 9.3: Interrupt Group 1 Bit Assignments

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

Bit	Name	Purpose
0x01	IEC_DATA_i	IEC Data In
0x02	IEC_CLK_i	IEC Clock In
0x04	IEC_ATN_i	IEC ATN In
0x08	IEC_SREQ_i	IEC SREQ In
0x10	RESERVED	
0x20	RESERVED	
0x40	RESERVED	
0x80	RESERVED	

Table 9.4: Interrupt Group 2 Bit Assignment

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 we will just clear all the pending interrupt flags to ensure the program starts 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 = $D66C  ; Mask register for interrupts 0 - 7
INT_MASK_1 = $D66D  ; 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
```

Polarity and Edge Controls

The POLARITY and EDGE registers work together to control how the interrupt controller recognizes an interrupt condition from the input signal. The EDGE register controls if the interrupt is triggered by the transition of the signal between high and low, and POLARITY controls which direction or logic level is the triggering condition. Table 9.5 lists how the two work together to choose the specific condition.

For groups 0 and 1, these registers are really not needed, and they should be left in their default settings. For group 2, these registers will be more useful for recognizing changes to the IEC input lines.

EDGE	POLARITY	Function
0	0	Interrupt is triggered if input line is LOW
0	1	Interrupt is triggered if input line is HIGH
1	0	Interrupt is triggered when the input transitions from HIGH to LOW
1	1	Interrupt is triggered when the input transitions from LOW to HIGH

Table 9.5: Interrupt Polarity and Edge Function

10

Tracking Time

Interval Timers

The F256jr provides two 24-bit timers. The two timers work on different clocks: timer 0 works off the CPU clock (6.29 MHz), while timer 1 works off the start-of-frame timing (either 60 Hz or 70 Hz, depending on the resolution). The timers have a few features in how they time things:

- they can count up from 0 or down from a starting value
- they can be set to trigger an interrupt on a specific value

- they can either reload a start value or reset the value to 0 on reaching the target value

There are five registers for each timer:

CTR the master control register for the timer. There are five flags:

INT_EN if set, the timer will trigger an interrupt on reaching the target value

UP if set, the timer will count up. If clear, it will count down.

CLR if set, the timer will reset to 0

LD if set, the timer will be set to the last value written to VAL

EN if set, the timer will count clock ticks

STAT this register (read on the same address as CTR) has just one flag EQ, which indicates if the timer has reached the target value

VAL when read, gives the current value of the timer. When written, sets the value to use when loading the timer.

CMP_CTR this register contains two flags to control what happens when the target value is reached. When RECLR is set, the timer will return to 0 on reaching the target value. When RELD is set, the timer will be set to the last value written to VAL.

CMP this register contains the target value for comparison

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.

There are 16 registers for the RTC (see table 10.2). 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 0-5 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 10.3).

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. 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
```

Address	R/W	Name	7	6	5	4	3	2	1	0
D650	W	T0_CTR	INT_EN		—		UP	CLR	LD	EN
D650	R	T0_STAT				—				EQ
D651	R/W	T0_VAL	V7	V6	V5	V4	V3	V2	V1	V0
D652	R/W		V15	V14	V13	V12	V11	V10	V9	V8
D653	R/W		V23	V22	V21	V20	V19	V18	V7	V6
D654	R/W	T0_CMP_CTR				—			RELD	RECLR
D655	R/W	T0_CMP	C7	C6	C5	C4	C3	C2	C1	C0
D656	R/W		C15	C14	C13	C12	C11	C10	C9	C8
D657	R/W		C23	C22	C21	C20	C19	C18	C17	C16
D658	W	T1_CTR	INT_EN		—		UP	CLR	LD	EN
D658	R	T1_STAT				—				EQ
D659	R/W	T1_VAL	V7	V6	V5	V4	V3	V2	V1	V0
D65A	R/W		V15	V14	V13	V12	V11	V10	V9	V8
D65B	R/W		V23	V22	V21	V20	V19	V18	V7	V6
D65C	R/W	T1_CMP_CTR				—			RELD	RECLR
D65D	R/W	T1_CMP	C7	C6	C5	C4	C3	C2	C1	C0
D65E	R/W		C15	C14	C13	C12	C11	C10	C9	C8
D65F	R/W		C23	C22	C21	C20	C19	C18	C17	C16

Table 10.1: Timer Registers

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 10.2: Real Time Clock Registers

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 10.3: RTC Periodic Interrupt Rates

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 3.3 volt logic instead of the 5 volt logic used in older 8-bit machines.

> NOTE: While the F256jr has a single VIA, the F256K has two VIA chips. The second VIA chip is located at 0xDB00. The purpose of this second VIA is to manage the built-in keyboard of the F256K. The keyboard/GPIO header pins on the F256jr are moved to VIA1 and include some additional I/O pins. See page 159 for a more complete description of the keyboard.

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. See table 11.1 for a listing of all the 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_L, T1C_H Timer 1 counter value

T1L_L, T1L_H Timer 1 latch

T2C_L, T2C_H Timer 2 counter value

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

ACR Auxiliary 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. See table 11.2 for details.

PCR Peripheral Control Register. Contains fields to control how the CA1, CA2, CB1, and CB2 handshake pins are used. See table 11.2 for details.

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. See table 11.2 for details.

IER Interrupt Enable Register. Contains flags to enable or disable interrupts based on the different possible conditions. See table 11.2 for details.

IORA2 Same as IOPA except that the built-in handshaking capability is not used.

Joystick Support

The F256jr has two IDC headers that can be connected to a DB-9 socket to allow Atari style joysticks to be used (see figure: 11.1 for the pinouts). Joystick header 0 is wired to the pins of Port B, and joystick header

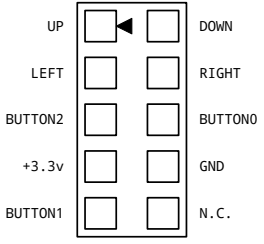

Figure 11.1: Joystick Port Pinouts

1 is connected to Port A. 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: 11.3).

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 3.3 volt supply. This means that any device plugged into the joystick ports should be 3.3 volt tolerant and should not raise any pin above 3.3 volts. Otherwise damage could occur.

Example: Displaying Joystick 1

In this example, we will poll joystick 1 and print out the state of all the buttons by printing the byte we read from the joystick port as a simple binary number. The example will try to be a little bit smart by only printing the value when the value has changed. NOTE: this example expects OpenKernal to be installed, and will call two of its routines for initializing the screen and printing a character.

First, we initialize the screen, the variable we use to track the old value of the joystick port, and the VIA (setting port A to be an input port):

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

; Variables

* = $0080

value:      .byte ?                     ; Variable to store the previous value of the joystick
prv:        .byte ?                     ; Copy of value for printing

* = $e000

start:      jsr ok_cint                 ; Set up the screen

            lda #$FF                    ; Set the previous value to $FF
            sta value
```

```
            stz MMU_IO_CTRL             ; Switch to I/O Page 0

            lda #$00                    ; Set VIA Port A to input
            sta VIA_DDRA
```

Next, we print the OpenKernal code to clear the screen, and we print out the byte in `value` as a binary number.

```
loop1:      lda #147                    ; Print the CBM clear screen code
            jsr ok_cout

            lda value                   ; Copy the value to prv
            sta prv

            ldx #8                      ; Loop for all eight bits
loop2:      asl prv                     ; Shift MSB into the carry
            bcc is0                     ; If it's 0, print '0'

            lda #'1'                    ; Otherwise, print '1'
            jsr ok_cout
            bra repeat                  ; And go to the next bit

is0:        lda #'0'                    ; Print '0'
            jsr ok_cout
```

```
repeat:     dex                         ; Count down
            bne loop2                   ; Repeat until we've done all 8 bits
```

Next, we read the value of port A. If it is different from `value`, we save it to `value` and go back to print the byte we read. Otherwise, we keep waiting and polling the joystick port.

```
            stz MMU_IO_CTRL             ; Switch to I/O Page 0

wait:       lda VIA_IORA                ; Get the status of port A
            cmp value                   ; Is it different from before?
            beq wait                    ; Yes: keep waiting

            sta value                   ; Save this value as the previous one
            bra loop1                   ; And go to print it
```

Address	R/W	Name	Purpose
0xDC00	R/W	IORB	Port B data
0xDC01	R/W	IORA	Port A data
0xDC02	R/W	DDRB	Port B Data Direction Register
0xDC03	R/W	DDRA	Port A Data Direction Register
0xDC04	R/W	T1C_L	Timer 1 Counter Low
0xDC05	R/W	T1C_H	Timer 1 Counter High
0xDC06	R/W	T1L_L	Timer 1 Latch Low
0xDC07	R/W	T1L_H	Timer 1 Latch High
0xDC08	R/W	T2C_L	Timer 2 Counter Low
0xDC09	R/W	T2C_H	Timer 2 Counter High
0xDC0A	R/W	SDR	Serial Data Register
0xDC0B	R/W	ACR	Auxiliary Control Register
0xDC0C	R/W	PCR	Peripheral Control Register
0xDC0D	R/W	IFR	Interrupt Flag Register
0xDC0E	R/W	IER	Interrupt Enable Register
0xDC0F	R/W	IORA2	Port A data (no handshake)

Table 11.1: VIA Registers

Name	7	6	5	4	3	2	1	0
ACR	T1_CTRL		T2_CTRL	SR_CTRL			PBL_EN	PAL_EN
PCR	CB2_CTRL			CB1_CTRL	CA2_CTRL			CA1_CTRL
IFR	IRQF	T1F	T2F	CB1F	CB2F	SRF	CA1F	CA2F
IER	SET	T1E	T2E	CB1E	CB2E	SRE	CA1E	CA2E

Table 11.2: VIA Control Registers

7	6	5	4	3	2	1	0
—	BUTTON2	BUTTON1	BUTTON0	RIGHT	LEFT	DOWN	UP

Table 11.3: Joystick Flags

12

SD Card Interface

TinyVicky includes an interface to the SD card port on F256jr. This interface provides access to the SPI bus interface SD cards support. This interface will allow a program to exchange bytes of data with an SD card using one of two clock speeds for the transfer rate (400 kHz or 12.5 MHz). Use of these registers requires an understanding of the SD card protocols and conventions, which are really outside the scope of this manual. So only the basic information about the control registers are provided here.

CS_EN This bit controls the chip select input on the SD card. If clear (0), the SD card is disabled. If set (1), the SD card is enabled.

SPI_CLK This bit controls the clock speed for the SPI interface to the SD card. If set (1), the clock speed

Address	R/W	7	6	5	4	3	2	1	0
0xDD00	RW	SPI_BUSY	—					SPI_CLK	CS_EN
0xDD01	RW	SPI_DATA							

Table 12.1: SD Card Interface Registers

is 400 kHz. If clear (0), the clock speed is 12.5 MHz.

SPI_BUSY This read only bit indicates if the SPI bus is busy exchanging bits with the SD card. The SPI_DATA register will not be ready for access while SPI_BUSY is set (1).

SPI_DATA this register is for the data to exchange with the SD card. A byte written to this register will be send to the SD card. The data read from this register are the bits received from the SD card. If SPI_BUSY is set, the program must way until SPI_BUSY is clear before reading or writing data to this register

NOTE: The system control registers have two bits relevant to the SD card interface: SD_WP, which indicates the write-protect status of the card, and SD_CD which indicates if a card is detected in the slot. See table 16.1 for details.

13

Keyboard and Mouse

The F256jr provides a single PS/2 port for use with either a keyboard or a mouse. This port is accessed through five registers, which provide very simple access to a PS/2 device. The F256jr does not have a full PS/2 controller, but instead provides mostly raw access to the data stream. It does make some attempt to translate set 2 scan codes to ASCII character code, although raw scan codes may be read instead. See table 13.1 for details.

K_WR set to 1 then 0 to send a byte written on PS2_OUT to the keyboard

M_WR set to 1 then 0 to send a byte written on PS2_OUT to the mouse

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD640	W	PS2_CTRL	—		MCLR	KCLR	M_WR	—	K_WR	—
0xD641	W	PS2_OUT	Data to send to keyboard							
0xD642	R	KBD_IN	Data from the keyboard input FIFO							
0xD643	R	MS_IN	Data from the mouse input FIFO							
0xD644	R	PS2_STAT	K_AK	K_NK	M_AK	M_NK	—		MEMP	KEMP

Table 13.1: PS/2 Port Registers

KCLR set to 1 then 0 to clear the keyboard input FIFO queue.

MCLR set to 1 then 0 to clear the mouse input FIFO queue.

K_AK when 1, the code sent to the keyboard has been acknowledged

K_NK when 1, the code sent to the keyboard has resulted in an error

M_AK when 1, the code sent to the keyboard has been acknowledged

M_NK when 1, the code sent to the keyboard has resulted in an error

KEMP when 1, the keyboard input FIFO is empty

MEMP when 1, the mouse input FIFO is empty

Mouse Support

The F256jr provides special support for a PS/2 mouse, including support for a hardware mouse pointer.

This is currently done with magic, the details of which will be made apparent when the stars are in proper alignment.

Mouse Pointer

The F256jr provides for a grayscale hardware mouse pointer. The pointer is a 16×16 grayscale image of 256 levels. Each pixel of the image is a single byte. The bitmap data is stored in the address range 0xCC00–0xCFFF.

The position of the mouse pointer is controlled in one of two ways. In the default approach (MODE = 0), the system software will monitor mouse movements, determine the mouse position programmatically, and set the TinyVicky mouse position registers directly. In the legacy approach (MODE = 1), the system software will receive the three byte PS/2 mouse data packet and set the TinyVicky mouse PS2_BYTE registers. In this legacy mode, TinyVicky will interpret the mouse packets and track the mouse position for the system. This approach is less work for the system software, but is less flexible.

EN if set (1), the mouse pointer is displayed. If clear (0), the mouse pointer is not displayed

MODE if clear (0), the mouse position is specified by setting the X and Y registers. If set (1), the program must pass along the 3 byte PS/2 mouse packet to the packet registers (this is a legacy mode).

X this is the X coordinate of the mouse and both readable and writable if MODE is clear (0)

Address	R/W	7	6	5	4	3	2	1	0
0xD6E0	W	—						MODE	EN
0xD6E2	RW	X7	X6	X5	X4	X3	X2	X1	X0
0xD6E3	RW	X15	X14	X13	X12	X11	X10	X9	X8
0xD6E4	RW	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0xD6E5	RW	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8
0xD6E6	W	PS2_BYTE_0							
0xD6E7	W	PS2_BYTE_1							
0xD6E8	W	PS2_BYTE_2							

Table 13.2: Mouse Pointer Registers

Y this is the Y coordinate of the mouse and both readable and writable if MODE is clear (0)

PS2_BYTE_0 the first byte of the PS/2 mouse message packet. Only used if MODE is set (1).

PS2_BYTE_1 the second byte of the PS/2 mouse message packet. Only used if MODE is set (1).

PS2_BYTE_2 the third byte of the PS/2 mouse message packet. Only used if MODE is set (1).

F256K Keyboard

NOTE: this section pertains to the F256K only and is not relevant to the F256jr.

The F256K includes a second WDC65C22 VIA chip at 0xDB00 on I/O page 0 (see page 143 for details on the VIAs) which manages the keyboard input from the F256K's built-in keyboard. The F256K's keyboard is a matrix keyboard. Except for the RESTORE key, all keys on the keyboard are arranged in a matrix with the columns assigned to VIA1's P xxx pins and the rows assigned to VIA1's P xxx pins. VIA0's PB7 pin is also used for xxx.

TODO: flesh this out properly and include a schematic of the matrix.

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.

RXD (read only) register contains data from the receive FIFO

TXR (write only) writing a byte stores it in the transmission FIFO to be sent over the serial connection

IER this is the interrupt enable register. There are flags for each of the four conditions that the UART can use to trigger an interrupt

ISR this is the interrupt STAT register. There are flags for each of the four conditions that can trigger an interrupt

FCR FIFO control register. This register controls the FIFOs for transmission and receiving:

RXT sets the number of characters in the receive FIFO to trigger an interrupt. See table: 14.5.

FIFO64

TXR if set, clear the transmition FIFO

RXR if set, clear the receive FIFO

FIFOE if set, the FIFOs are enabled. Otherwise, only a single character can be waiting to send or pending a read

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	—				STAT	ERR	TXE	RXA
0xD632	R	ISR	—				STAT	ERR	TXE	RXA
0xD632	W	FCR	RXT		FIFO64	—	—	TXR	RXR	FIFOE
0xD633	R/W	LCR	DLAB	—	PARITY			STOP	DATA	
0xD634	R/W	MCR	—			LOOP	OUT2	OUT1	RTS	DTR
0xD635	R	LSR	ERR	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

BPS	Divisor
300	5,244
600	2,622
1,200	1,311
1,800	874
2,000	786
2,400	655
3,600	437
4,800	327
9,600	163
19,200	81
38,400	40
57,600	27
115,200	13

Table 14.6: UART Divisors

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 as a rectangle (the data is a rectangular area of an image).

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.

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

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

Table 15.1: DMA Registers (Part 1)

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.

Address	R/W	7	6	5	4	3	2	1	0
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	W7	W6	W5	W4	W3	W2	W1	W0
0xDF0D	R/W	W15	W14	W13	W12	W11	W10	W9	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	SS7	SS6	SS5	SS4	SS3	SS2	SS1	SS0
0xDF11	R/W	SS15	SS14	SS13	SS12	SS11	SS10	SS9	SS8
0xDF12	R/W	SD7	SD6	SD5	SD4	SD3	SD2	SD1	SD0
0xDF13	R/W	SD15	SD14	SD13	SD12	SD11	SD10	SD9	SD8

Table 15.2: DMA Registers (Part 2)

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)

SS the width of the "stride" for the source bitmap (only available for 2D copy operations)

SD the width of the "stride" for the destination bitmap (only available when 2D is set)

The term "stride" might be a little confusing. It is a concept only relevant for 2D DMA operations. For 2D operations, the DMA engine is always copying or filling a rectangular area of a given width and height. The width of the rectangle need not be the full size of the overall bitmap image. If a program performs a DMA operation on a 32×32 area of the screen, the DMA engine will need to skip many pixels on each affected line. Thus, the program needs to inform the DMA engine how many pixels wide the bitmap is. For example, if a program is filling 32×32 rectangle in a 320×240 bitmap, it needs to tell the DMA engine that the width of the bitmap is 320. This number is the "stride" for the operation. The DMA engine will operate on 32 pixels, then skip $320 - 32$ pixels to get to the next set of 32 pixels to copy or fill. Figure 15.1 illustrates how the various addresses, sizes, and strides work in the case of a 2D copy operation.

For 2D fill operations, only the destination stride (SD) is needed. The destination stride specifies how wide the bitmap being altered is. For 2D copy operations, both the destination stride and the source stride (SS) are needed. The source stride specifies how wide the bitmap is that provides the source data. Why would you ever have a source and destination stride be different? As a simple example, let's say a program needs to copy graphical font data to a bitmap for the screen in 320×240 mode. The characters are in cells that are 8 pixels wide by 16 pixels deep, and that the characters are arranged vertically. So the over all image for the characters is 8 pixels wide by 4,096 pixels high. In that case, the source stride is 8, but the destination stride is 320 (since it is the bitmap for the full screen).

Example: Using DMA to Fill a Bitmap

In this example, we will use the 1D fill operation to set the bitmap shown on the screen to a set color. Note that the full example code includes the various setup operations for the bitmap mode and the graphics CLUTs, which are exactly the same as were used in the bitmap example (see page 4).

```
DMA_CTRL = $DF00                ; DMA Control Register
DMA_CTRL_START = $80            ; Start the DMA operation
DMA_CTRL_FILL = $04             ; DMA is a fill operation
DMA_CTRL_ENABLE = $01           ; DMA engine is enabled

DMA_STATUS = $DF01              ; DMA status register (Read Only)
DMA_STAT_BUSY = $80             ; DMA engine is busy with an operation

DMA_FILL_VAL = $DF01            ; Byte value to use for fill operations

DMA_DST_ADDR = $DF08            ; Destination address (system bus)
DMA_COUNT = $DF0C               ; Number of bytes to fill or copy

bitmap_base = $10000            ; The base address of our bitmap

bitmap_width = 320              ; The size of our bitmap
bitmap_height = 240
bitmap_size = bitmap_width*bitmap_height
```

First, we need to enable the DMA engine and set it up for a fill operation:

```
lda #DMA_CTRL_FILL | DMA_CTRL_ENABLE
sta DMA_CTRL
```

Next, we provide the value to fill:

```
lda #$ff
sta DMA_FILL_VAL    ; We will fill the screen with $FF
```

Then we need to provide the destination address:

```
lda #<bitmap_base   ; Our bitmap will be the destination
sta DMA_DST_ADDR
lda #>bitmap_base
sta DMA_DST_ADDR+1
lda #‘bitmap_base
and #$03
sta DMA_DST_ADDR+2
```

Next, we provide the number of bytes to write:

```
lda #<bitmap_size   ; We will write 320*240 bytes
sta DMA_COUNT
lda #>bitmap_size
```

```
            sta DMA_COUNT+1
            lda #‘bitmap_size
            sta DMA_COUNT+2
```

Finally, we flip the START flag to trigger the DMA operation and wait for it to complete:

```
            lda DMA_CTRL
            ora #DMA_CTRL_START
            sta DMA_CTRL

wait_dma:   lda DMA_STATUS      ; Wait until DMA is not busy
            and #DMA_STAT_BUSY
            cmp #DMA_STAT_BUSY
            beq wait_dma

            stz DMA_CTRL        ; Turn off the DMA engine
```

Example: Calculating an Address for DMA

We can extend our 1D DMA example to draw a 2D rectangle at a given coordinate on the screen. Using the coordinate math block, we can quickly calculate the starting address for the destination address of the rectangle to fill.

```
DMA_CTRL_2D = $02               ; DMA is 2D operation (otherwise it is 1D)
```

```
DMA_WIDTH = $DF0C               ; Width of rectangle to fill or copy
DMA_HEIGHT = $DF0E              ; Height of rectangle to fill or copy

DMA_STRIDE_DST = $DF12          ; Width of the destination bitmap image

XY_BASE = $D301         ; Starting address of the bitmap
XY_POS_X = $D304        ; X-coordinate desired
XY_POS_Y = $D306        ; Y-coordinate desired
XY_OFFSET = $D308       ; Offset within an MMU bank of the pixel for (X, Y)
XY_BANK = $D30A         ; MMU bank containing the pixel for (X, Y)
XY_ADDRESS = $D30B      ; System address of the pixel for (X, Y)
```

We start off very similar to the 1D fill and just turn on the flag for 2D DMA:

```
lda #DMA_CTRL_FILL | DMA_CTRL_2D | DMA_CTRL_ENABLE
sta DMA_CTRL
```

Now, we can take the base address of our bitmap and use it at the base address of the coordinate calculation:

```
lda #<bitmap_base   ; Give the coordinate math unit the base address
sta XY_BASE
lda #>bitmap_base
sta XY_BASE+1
lda #‘bitmap_base
```

```
and #$03
sta XY_BASE+2
```

Next, we give it our x and y coordinates for the upper left corner of the rectangle we want to draw. In this case, $(100, 40)$:

```
lda #100              ; Set (x,y) of the rectangle to (100, 40)
sta XY_POS_X
stz XY_POS_X+1
lda #40
sta XY_POS_Y
stz XY_POS_Y+1
```

Then, we can just read off the address from the coordinate calculator and use it as the destination address for our DMA operation:

```
lda XY_ADDRESS      ; Get the address of the upper left corner of the rect
sta DMA_DST_ADDR    ; And use it as the DMA address
lda XY_ADDRESS+1
sta DMA_DST_ADDR+1
lda XY_ADDRESS+2
sta DMA_DST_ADDR+2

lda #$30
sta DMA_FILL_VAL    ; We will fill the screen with $30
```

A difference from 1D is that the size of the fill is a width and a height, so we provide the width and height of the rectangle we want to draw. In this case, we are setting it to (100, 30):

```
lda #100            ; Size of rectangle is (100,30)
sta DMA_WIDTH
stz DMA_WIDTH+1
lda #30
sta DMA_HEIGHT
stz DMA_HEIGHT+1
```

For 2D DMA operations, we need to provide the "stride". This is the width of the overall bitmap into which the DMA operation is writing. In this case, we are updating a bitmap that is the full screen, so we set the destination stride to 320:

```
lda #<bitmap_width  ; Set the width of the destination bitmap for 2D DMA
sta DMA_STRIDE_DST
lda #>bitmap_width
sta DMA_STRIDE_DST+1
```

And then we can start the DMA operation and wait for it to complete:

```
lda DMA_CTRL
ora #DMA_CTRL_START
sta DMA_CTRL
```

```
wait_dma2d: lda DMA_STATUS      ; Wait until DMA is not busy
    and #DMA_STAT_BUSY
    cmp #DMA_STAT_BUSY
    beq wait_dma2d
```

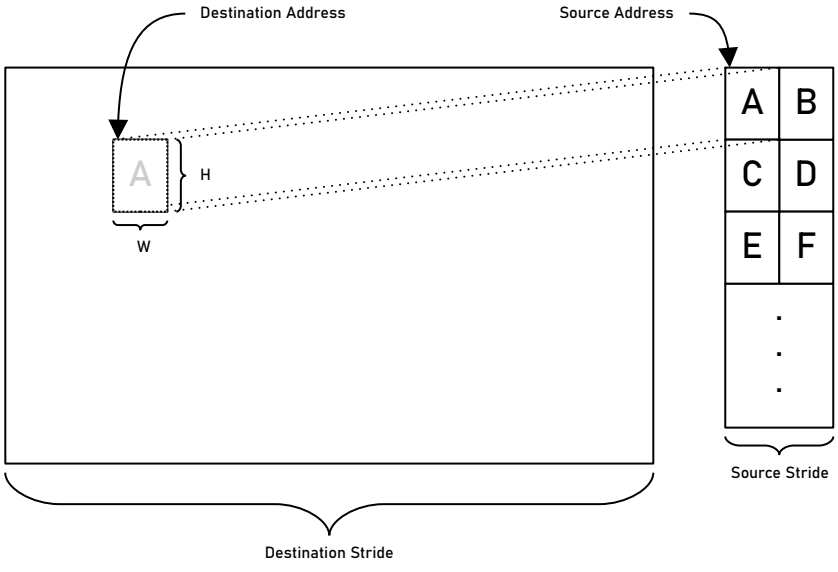

Figure 15.1: A 2D Copy DMA Operation

16

System Control Registers

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.

The two status LEDs on the board are a little more complex. They may be in manual or automatic

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

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

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

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).

ENABLE set to turn on the random number generator

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD6A4	W	SEEDL	SEED[7…0]							
0xD6A4	R	RNDL	RND[7…0]							
0xD6A5	W	SEEDH	SEED[15…0]							
0xD6A5	R	RNDH	RND[15…0]							
0xD6A6	W	RND_CTRL	—						SEED_LD	ENABLE
0xD6A6	R	RND_STAT	DONE	—						

Table 16.4: 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

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 16.6).

For the F256jr, the machine ID will be 2.

NOTE: The F256jr RevA board does not support the PCB major and minor revision number registers or the PCB date code registers.

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”							
0xD6EB	R	PCBMA	PCB Major Rev (ASCII)							
0xD6EB	R	PCBMI	PCB Minor Rev (ASCII)							
0xD6EB	R	PCBD	PCB Day (BCD)							
0xD6EB	R	PCBM	PCB Month (BCD)							
0xD6EB	R	PCBY	PCB Year (BCD)							
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)							
0xD6AE	R	CHN0	Chip number in BCD (low)							
0xD6AF	R	CHN1	Chip number in BCD (high)							

Table 16.5: Machine ID and Versions

MID4	MID3	MID2	MID1	MID0	Machine
0	0	0	0	0	C256 FMX
0	0	0	0	1	C256 U
0	0	0	1	0	F256jr
1	0	0	1	0	F256K
0	0	0	1	1	A2560 Dev
0	0	1	0	0	Gen X
0	0	1	0	1	C256 U+
0	0	1	1	0	Reserved
0	0	1	1	1	Reserved
0	1	0	0	0	A2560 X
0	1	0	0	1	A2560 U
0	1	0	1	0	A2560 M
0	1	0	1	1	A2560 K

Table 16.6: Machine IDs

17

IEC Serial Port

The F256jr had an IEC serial port included (this is the Commodore serial port variation of the IEEE-488 interface). There are two registers supporting the IEC port. There is a read-only register that shows the current state of the individual lines on the serial bus, and there is a read/write register that can be used to control the various lines as well as how IEC interrupts are handled.

DAT_i Reflects the current state of the DATA line on the IEC bus.

CLK_i Reflects the current state of the CLK line on the IEC bus.

ATN_i Reflects the current state of the ATN line on the IEC bus.

Address	R/W	Name	7	6	5	4	3	2	1	0
0xD680	R	IEC_I	SRQ_i	—				ATN_i	CLK_i	DAT_i
0xDC01	R/W	IEC_O	SRQ_o	RST_o	NMI_EN	ATN_o	—	—	CLK_o	DAT_o

Table 17.1: IEC Registers

SRQ_i Reflects the current state of the SREQ line on the IEC bus.

DAT_o Sets the DATA line on the IEC bus.

CLK_o Sets the CLK line on the IEC bus.

ATN_o Sets the ATN line on the IEC bus.

SRQ_o Sets the SREQ line on the IEC bus.

RST_o Resets the IEC bus and the SIDs. This works on the RevB and F256K.

NMI_EN If set (1), the IEC interrupts will trigger an NMI interrupt. If clear (0), the IEC interrupts will trigger an IRQ interrupt.

18

Integer Math Coprocessor

The F256jr includes a built-in math coprocessor for integer math. This coprocessor provides fast 16-bit multiplication and division in both signed and unsigned forms. The use of this coprocessor is straightforward: both operands are written to the appropriate registers and then the result is read for the corresponding answer register. The math units are completely separate blocks using separate registers, so they function independently of each other.

NOTE: The input registers can be read or written by the CPU, but the addresses for the registers are different depending upon whether the CPU is reading or writing. The read address and write address for each register are listed separately in the tables below.

Address W	Address R	Name	Data
0xDE00	0xDE00	MULU_A_L	Unsigned A (bits 7–0)
0xDE01	0xDE01	MULU_A_H	Unsigned A (bits 15–8)
0xDE02	0xDE02	MULU_B_L	Unsigned B (bits 7–0)
0xDE03	0xDE03	MULU_B_H	Unsigned B (bits 15–8)
	0xDE04	MULU_LL	Unsigned $A \times B$ (bits 7–0)
	0xDE05	MULU_LH	Unsigned $A \times B$ (bits 15–8)
	0xDE06	MULU_HL	Unsigned $A \times B$ (bits 23–16)
	0xDE07	MULU_HH	Unsigned $A \times B$ (bits 31–24)
0xDE04	0xDE08	MULS_A_L	Signed A (bits 7–0)
0xDE05	0xDE09	MULS_A_H	Signed A (bits 15–8)
0xDE06	0xDE0A	MULS_B_L	Signed B (bits 7–0)
0xDE07	0xDE0B	MULS_B_H	Signed B (bits 15–8)
	0xDE0C	MULS_LL	Signed $A \times B$ (bits 7–0)
	0xDE0D	MULS_LH	Signed $A \times B$ (bits 15–8)
	0xDE0E	MULS_HL	Signed $A \times B$ (bits 23–16)
	0xDE0F	MULS_HH	Signed $A \times B$ (bits 31–24)

Table 18.1: Math Multiplication Registers

Address W	Address R	Name	Data
0xDE08	0xDE10	DIVU_DEN_L	Unsigned Denominator (bits 7–0)
0xDE09	0xDE11	DIVU_DEN_H	Unsigned Denominator (bits 15–8)
0xDE0A	0xDE12	DIVU_NUM_L	Unsigned Numerator (bits 7–0)
0xDE0B	0xDE13	DIVU_NUM_H	Unsigned Numerator (bits 15–8)
	0xDE14	DIVU_QUO_L	Unsigned Quotient NUM/DEN (bits 7–0)
	0xDE15	DIVU_QUO_H	Quotient NUM/DEN (bits 15–8)
	0xDE16	DIVU_REM_L	Unsigned Remainder NUM/DEN (bits 7–0)
	0xDE17	DIVU_REM_H	Unsigned Remainder NUM/DEN (bits 15-8)
0xDE0C	0xDE18	DIVS_DEN_L	Signed Denominator (bits 7–0)
0xDE0D	0xDE19	DIVS_DEN_H	Signed Denominator (bits 15–8)
0xDE0E	0xDE1A	DIVS_NUM_L	Signed Numerator (bits 7–0)
0xDE0F	0xDE1B	DIVS_NUM_H	Signed Numerator (bits 15–8)
	0xDE1C	DIVS_QUO_L	Signed Quotient NUM/DEN (bits 7–0)
	0xDE1D	DIVS_QUO_H	Signed Quotient NUM/DEN (bits 15–8)
	0xDE1E	DIVS_REM_L	Signed Remainder NUM/DEN (bits 7–0)
	0xDE1F	DIVS_REM_H	Signed Remainder NUM/DEN (bits 15-8)

Table 18.2: Math Division Registers

19

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 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.

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 a command packet
4. The F256jr replies

5. The host repeats from step 3 until finished
6. Host PC send the command to exit debug mode
7. 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 19.1. The command codes themselves are listed in table 19.3. The F256jr will respond to each command packet with a response packet as shown in table 19.2. 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, except for 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: 19.3)

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.

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.

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 19.1: USB Debug Port 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.

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

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

Table 19.2: USB Debug Port Command Packet

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.

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.

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
0x90	Set the MMU to boot in RAM (F256jr Rev A only)
0x91	Set the MMU to boot in flash (F256jr Rev A only)
0xFE	Fetch the revision number of the debug interface

Table 19.3: USB Debug Port Commands

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.

20 Expansion Connector

The F256jr RevB includes a PCIe x1 style expansion connector. This expansion connector is designed to support both RAM expansion and game style ROM cartridges. The pin assignments for the expansion connector can be seen in figure 20.1, while the size information for the card itself can be seen in figure 20.2.

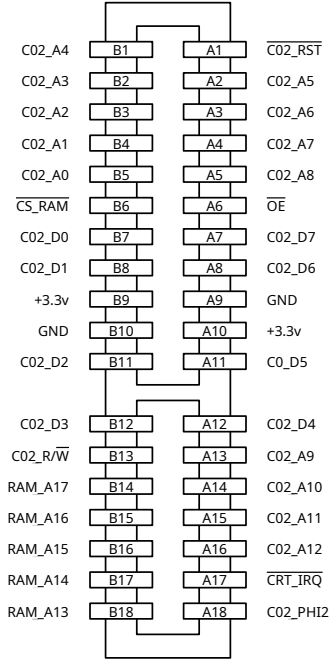

Figure 20.1: Expansion Port

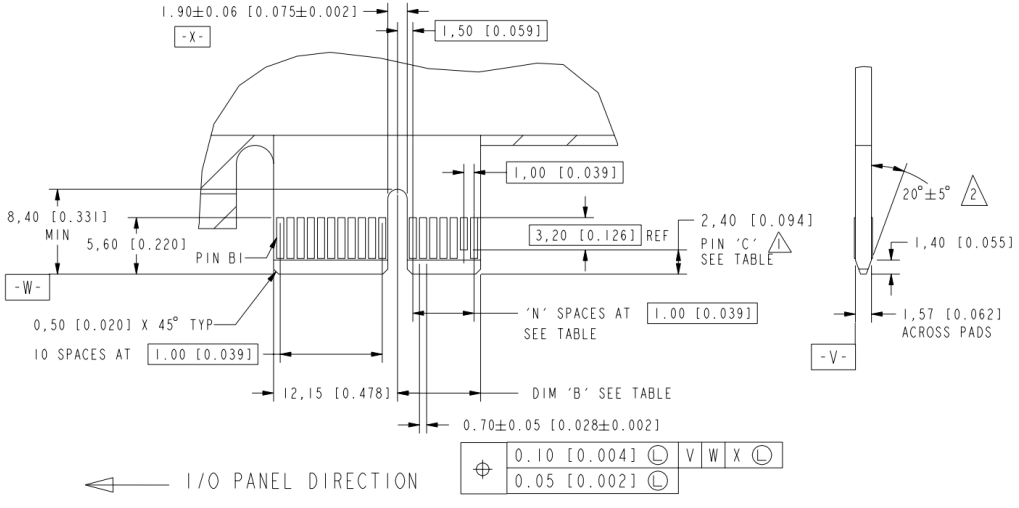

Figure 20.2: Expansion Port Physical Size

21

I/O Connectors and Jumpers

Connectors

The F256jr has several connectors on its board for I/O devices beyond the standard connections on the back. Some of these connectors are the only way to access that particular I/O device, but some are auxiliary connectors that provide alternate forms of access. All are IDC header pins. The diagrams that follow show the pin assignments for these connectors. In these diagrams, the views are top-down onto the board, with the board arranged so that the main connectors are towards the top and the power connector is towards the bottom.

Game Connectors

There are two connectors for game controllers. There are two connectors for Atari style joysticks (figure 21.1). And two for NES or Super NES gamepads (figure 21.2).

For the Atari style joysticks, an adapter cable will be needed to convert from IDC to the standard DB-9 connector. There are two key differences between the F256jr and Atari or Commodore devices: first, the F256jr provides 3.3 volts instead of 5 volts, and second there are no paddle inputs supported on the F256jr.

For the NES and SNES style gamepads, an adapter box will be needed to provide the correct interfacing for a controller. This adapter should be available from Foenix Retro Systems in the near future (at the time of writing this manual).

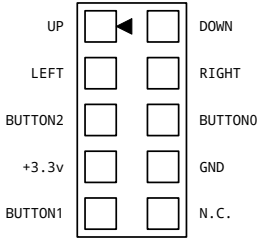

Figure 21.1: Joystick Port Pinouts

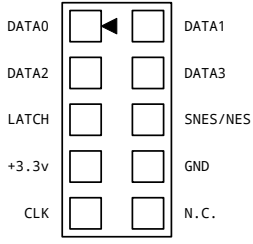

Figure 21.2: NES/SNES Gamepad Port Pinouts

PS/2 Port

An internal connector is included for the PS/2 mouse and keyboard port (figure 21.3). This connector just provides an alternate way of accessing the PS/2 signals. It could be useful in building an integrated case for the F256jr that includes a PS/2 keyboard.

UART

This connector provides access to the serial in and out signals for the F256jr's UART (figure 21.4). The TxD and RxD signals are compatible with standard ±12 volt RS-232 signals. The signals on this connector can be brought out to a DB-9 connector to provide a standard 3 wire RS-232 serial port.

NOTE: a ESP32 Feather board can be installed on the F256jr to provide access to Wi-Fi, but using

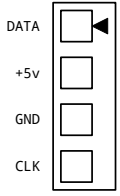

Figure 21.3: Auxiliary PS/2 Pinouts

this board will take over the supplied UART. Therefore, the F256jr can provide either an RS-232 serial port or access to Wi-Fi, but not both at the same time. A jumper on the board selects which is active.

USB Debug Port

While there is a USB Mini-B connector (figure: 21.5) on the port to access the USB debug interface, there is an IDC header connector on the board to provide access to this for the case USB connector, if desired. This is not compatible with all USB case connectors, as many of them are dual connectors and require pins for two USB ports. It should be compatible with cases with single connectors or with panel mounted USB cables.

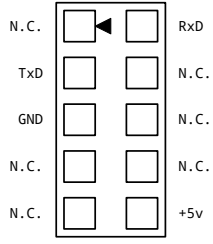

Figure 21.4: UART Pinouts

Case Connectors

There are two connectors for the usual PC case connections. There is one for the headphone jack (figure: 21.6), and one for the various buttons, LEDs, and case speaker (figure: 21.7): power, reset, hard drive/SD card activity. Note that while the speaker and buttons are not polarized, the power LED and hard-drive activity LED are and must be wired in the correct orientation.

NOTE: not shown is a small secondary connector for an SPST switch for power (the connector is located at the front-right corner of the F256jr RevB board). This connector provides an alternative to the usual case power push button. The two buttons cannot be used together. Either the case push button shown in 21.7 is used, or the other SPST switch is used, but not both.

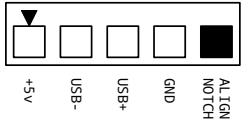

Figure 21.5: USB Debug Pinouts

Jumpers

There are several IDC header pins with jumpers to configure various options on the F256jr. All the jumper headers are three pin headers, where the center pin is the common. The header is used to select the appropriate routing of a signal or voltage level. So the jumper is always used to connect the center common pin to either the left or right pins (or the top or bottom, depending on the orientation of the header).

SID Jumpers

The SID chips have two sets of jumpers each. The first pair are the voltage selection jumpers. They can be used to select the appropriate voltage for the SID chip: 12 volts for the original 6581, and 9 volts for the later 8580 (see figure 21.8). If you are using a modern replacement, check the instructions as to which voltage to use: some replacements work with either voltage.

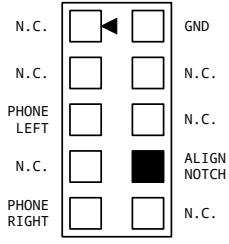

Figure 21.6: Case Audio Pinouts

The second pair of SID jumpers are the channel selectors (see figure 21.9). These jumpers select the source of the left and right channels for the CODEC. With each one, you can select either the right or the left SID as an input to the CODEC for that stereo channel. If a F256jr is using both SIDs, the left channel should select the left SID, and the right channel select the right SID. But if only a single SID is used, both channels can be set to that SID. For instance, if the SID is in the left socket, both left and right channels can select the left SID as the source to get a balanced monaural sound.

Boot Source

For the F256jr RevB, there is a jumper to select boot source (see figure 21.10). With the jumper in the left position, the F256jr will boot off the flash, using the last 8KB bank of flash memory as the last 8KB

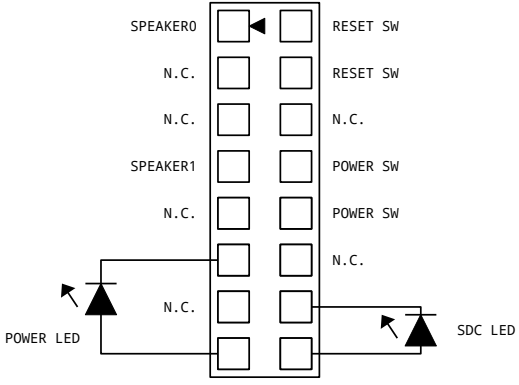

Figure 21.7: Case Button and LED Pinouts

of CPU address space. With the jumper in the right position, the F256jr will boot using RAM. On the RevA boards, this jumper is not present, and so this feature is controlled by a command on the USB debug port, which the RevB board does not support. See page 2 for a more complete description of boot sources.

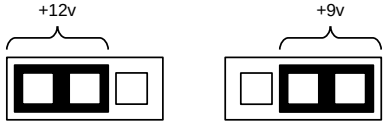

Figure 21.8: SID Voltage Jumper

UART Configuration

There are two headers for selecting how the UART is used—selecting between the serial port or the ESP32 Wi-Fi module (see figure 21.11). One jumper controls the routing of the TxD line, while the other controls the routing of the RxD line. With the jumper positioned across the “back” pair of pins, the signal is routed to the Wi-Fi module. With the jumper positioned across the “front” pair of pins, the signal is routed to the DB-9 port connected to the UART connector. Note that both the TxD and RxD jumpers should be in the same relative position so that they both use the same device.

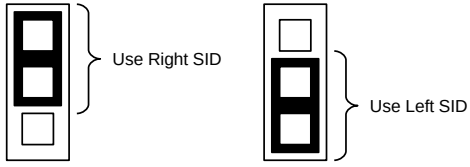

Figure 21.9: SID Stereo Channel Source Jumper

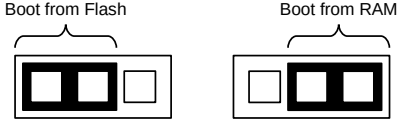

Figure 21.10: Boot Source Jumper

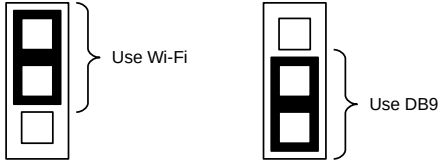

Figure 21.11: UART Device Jumpers

22

Memory Maps

Address	Purpose
0x000000	System RAM for programs, data, and graphics (512 KB)
0x07FFFF	
0x080000	Flash memory (512 KB)
0x0FFFFF	
0x100000	Expansion RAM for programs, and data (256 KB)
0x13FFFF	
0x140000	Reserved
0x1FFFFF	

Table 22.1: System Memory Map for the F256jr

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

Table 22.2: System Memory Map for the F256jr RevA

Bank	Address	Purpose
0	0x0000	MMU Memory Control Register
	0x0001	MMU I/O Control Register
	0x0002	RAM or Flash
	0x0007	
	0x0008	RAM, Flash, or MMU LUT Registers
	0x000F	
	0x0010	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	

Table 22.3: CPU Memory Map for the F256jr (Banks 0–3)

Bank	Address	Purpose
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 22.4: CPU Memory Map for the F256jr (Banks 4–7)

Start	End	Purpose	Page
0xC000	0xC3FF	Gamma Table Blue	101
0xC400	0xC7FF	Gamma Table Green	
0xC800	0xCBFF	Gamma Table Red	
0xCC00	0xCFFF	Reserved	
0xD000	0xD0FF	VICKY Master Control Registers	43
0xD100	0xD1FF	VICKY Bitmap Control Registers	53
0xD200	0xD2FF	VICKY Tile Control Registers	77
0xD300	0xD3FF	Reserved	
0xD400	0xD4FF	SID Left	117
0xD500	0xD5FF	SID Right	
0xD600	0xD607	PSG Left	109
0xD608	0xD60F	PSG Left and Right	
0xD610	0xD61F	PSG Right	
0xD620	0xD62F	CODEC	107
0xD630	0xD63F	UART	163
0xD640	0xD64F	PS/2 Interface	156
0xD650	0xD65F	Timers	139
0xD660	0xD66F	Interrupt Controller	119
0xD670	0xD67F	DIP Switch	90

Table 22.5: I/O Page 0 Addresses (0xC000–0xD67F)

Start	End	Purpose	Page
0xD680	0xD68F	IEC Controller	150
0xD690	0xD69F	Real Time Clock	140
0xD6A0	0xD6AF	System Control Registers	181
0xD6B0	0xD6DF	Reserved	
0xD6E0	0xD6EA	Mouse Pointer Registers	158
0xD6EB	0xD7EF	PCB Version Register	186
0xD6EF	0xD7FF	Reserved	
0xD800	0xD83F	Text Foreground Color LUT	42
0xD840	0xD87F	Text Background Color LUT	
0xD880	0xD8FF	Reserved	
0xD900	0xDAFF	VICKY Sprite Control Registers	64
0xDB00	0xDBFF	Reserved	
0xDC00	0xDCFF	65C22 VIA Control Registers	150
0xDD00	0xDDFF	SD Card Controller	156
0xDE00	0xDE1F	Integer Math Coprocessor	191
0xDE20	0xDEFF	Reserved	
0xDF00	0xDF13	DMA Controller	169

Table 22.6: I/O Page 0 Addresses (0xD680–0xDF13)

Start	End	Purpose	Page
0xC000	0xC7FF	Text Mode Font Set 0 Memory	39
0xC800	0xCFFF	Text Mode Font Set 1 Memory	39
0xD000	0xD3FF	Graphics Color LUT 0	47
0xD400	0xD7FF	Graphics Color LUT 1	
0xD800	0xDBFF	Graphics Color LUT 2	
0xDC00	0xDFFF	Graphics Color LUT 3	

Table 22.7: Memory Map for I/O Page 1

23

The TinyCore 65c02 MicroKernel F256(K) Edition

NOTE: This chapter was written by Jessie Oberreuter to describe her MicroKernel for the F256 family and is covered under her project's copyright. It is included here by her kind permission. The link to get the API files and sample code for the kernel is listed below and in the References chapter at the end of the manual.

The TinyCore 6502 MicroKernel for the Foenix F256/F256K line of computers is a powerful alternative to the typical BIOS style kernels that come with most 8-bit computers. This kernel offers the following features:

- An event-based programming model for real-time games and VMs.
- A network stack supporting multiple concurrent TCP and UDP sockets.
- Drivers for optional ESP8266 based wifi modules.
- Drivers for IEC drives.
- Drivers for Fat32 formatted SD Cards.
- Support for PS2 keyboards, Foenix keyboards, and CBM keyboards
- Support for PS2 scroll mice.
- Preemptive kernel multi-tasking (no need to yield).

There's a lot of love in here; I hope you will enjoy using it as much as I have enjoyed writing it! — Gadget.

Setup and Installation

Getting the Kernel

Kernel binaries may be obtained from either of the following repos:

- https://github.com/ghackwrench/F256_Jr_Kernel_DOS
- https://github.com/paulscottrobson/superbasic

The ‘ghackwrench’ repo contains the latest release. The ‘paulscottrobson’ repo contains the latest version verified to work with SuperBASIC.

Flashing the Kernel

The kernel consists of four 8k blocks which must be flashed into the last four blocks of the F256. Using the F256 Programmer Tool (Windows), the kernel needs to be flashed into Flash Blocks 0x3C–0x3F. Alternately, the SuperBASIC repo contains a script which uses a python based loader to install SuperBASIC along with the Kernel.

DIP Switches

The kernel assigns the DIP switches as follows:

1. Enable boot-from-RAM
2. Reserved (potentially for SNES gamepad support)
3. Enable SLIP based networking
4. Feather board installed (*e.g.* Huzzah 8266 WiFi)

5. SIDs are installed
6. F256: CBM keyboard installed; F256k: audio expansion installed.
7. ON: 640x480, OFF: 640x400 (*not yet implemented*)
8. Enable gamma color correction

When boot-from-RAM is enabled, the kernel will search the first 48k of RAM for pre-loaded programs before starting the first program on the expansion cartridge or the first program in flash.

Support Software

- The superbasic repo contains a powerful BASIC programming language which has been ported to run on the MicroKernel.
- The Kernel_DOS repo includes a simple DOS CLI which demonstrates most kernel functions.
- The Kernel_DOS repo also contains library and config files for compiling C programs with cc65 for use with the MicroKernel.

Memory Model

The F256 series machines minimally reserve addresses 0x00 and 0x01 for memory and I/O control. Programs may optionally enable additional hardware registers from 0x08–0x0f.

The MicroKernel uses 16 bytes of the Zero page, from 0xf0–0xff, for the `kernel.args` struct. Kernel calls do not expect their arguments to come in registers. Instead, all kernel arguments are passed by writing them to variables in `kernel.args`. The `A` register is used by all kernel calls. The `X` and `Y` registers, however, are always preserved. All kernel calls clear the carry on success and set the carry on failure. A few calls return a stream id in `A` as a convenience, but most of the time, `A` should just be considered undefined upon return from the kernel.

Outside of the above, user programs are free to use all addresses up to 0xC000 as they wish.

0xC000–0xDFFF is the I/O window. The I/O window can be disabled to reveal RAM underneath, but programs should refrain from doing so, as this RAM is used by the kernel. Programs are generally free to use the hardware as they see fit, but note that, by default, the kernel takes ownership of the PS2 ports, the frame interrupt, and the RTC interrupt, and will take ownership of the serial port if either the SLIP or Feather DIP switches are set.

0xE000–0xFFFF contains the kernel itself. The kernel is considerably larger than 8kB, but it uses the F256 MMU hardware to keep its footprint in the user's memory map to a relative minimum.

The F256 machines support four concurrent memory maps. The kernel reserves map zero for itself, and map one for the Fat32 drivers; user programs are run from map three. Map two is potentially reserved for a hypervisor.

The kernel uses only two 8k blocks of RAM: block 6 (nominally at 0xC000), and block 7 (nominally at 0xE000). Everything else is free for use by user software.

Startup

On power-on or reset, the kernel initializes the hardware according to the DIP switches and then searches various memory regions for the first program to run. If DIP1 is on, it first checks RAM blocks 1–5 for a pre-loaded binary. After that, it searches the expansion RAM/ROM blocks, and, finally, the on-board flash blocks. The first block found with a valid header will be mapped into MMU 3 and started. The header must appear at the very start of the block and contains the following fields:

```
Byte  0    signature: 0xF2
Byte  1    signature: 0x56
Byte  2    the size of program in 8k blocks
Byte  3    the starting slot of the program (ie where to map it)
Bytes 4-5  the start address of the program
Bytes 6-9  reserved
Bytes 10-  the zero-terminated name of the program.
```

Programming

With a 65c02 installed, the F256 machines can generally be treated as simple 6502 machines. Programs have full access to the RAM from 0x0000–0xBFFF, and full access to I/O from 0xC000–0xDFFF. More adventurous programs can enable the MMU registers and bank additional memory into any 8k slot below 0xC000. Really adventurous programs are free to disable interrupts, map out the kernel, and take complete control over the machine. Programs wishing to use the kernel, however, should be sure

to keep interrupts enabled as much as possible, call `NextEvent` often enough that the kernel doesn't run out of event objects, and refrain from trashing MMU LUT0, MMU LUT1, and the RAM between 0xC000 and 0xFFFF.

The kernel interfaces are described below using their symbolic names. The actual values and addresses must be obtained by including either `api.asm` or `api.h` in your project.

Events

The F256 machines were designed for games, and the 6502 TinyCore MicroKernel was designed to match.

Games generally run in a simple loop:

1. Update the screen
2. Read the controls
3. Update the game state
4. Goto 1

The kernel supports this mode of operation by replacing step 2 above (read the controls) with a generic kernel call: `NextEvent`. `NextEvent` gets the next I/O event from the kernel's queue and copies it into a user provided buffer.

Setup

Events are 8-byte packets. Each packet contains four common bytes, and up to four event-specific bytes. The user's buffer for these bytes may be placed anywhere, but since they are accessed frequently, the zero page is a good place.

Before a program can receive events, it must tell the kernel where events should be copied. It does this by writing the address of an 8-byte buffer into the kernel's `arg` block:

```
    .section zp
event   .dstruct  kernel.event.event_t
    .send

    .section code

init_events

    lda     #<event
    sta     kernel.args.events+0
    lda     #>event
    sta     kernel.args.events+1
    rts
```

Handling

For games, and other real-time applications, a program will typically contain a single "handle_events" routine:

```
handle_events

    ; Peek at the queue to see if anything is pending
    lda kernel.events.pending  ; Negated count
    bpl _done

    ; Get the next event.
    jsr kernel.NextEvent
    bcs _done

    ; Handle the event
    jsr _dispatch

    ; Continue until the queue is drained.
    bra handle_events

_done

    rts
```

```
_dispatch

    ; Get the event's type
    lda event.type

    ; Call the appropriate handler

    cmp #kernel.event.key.PRESSED
    beq _key_pressed

    cmp #kernel.event.mouse.DELTA
    beq _mouse_moved

    ...
    rts     ; Anything not handled can be ignored.
```

Other types of programs may eschew the use of a single central event handler, and instead work the queue only when waiting for events, and then only to handle the event types expected for the operation. The F256 cc65 kernel library does just this: when waiting for keypresses, it only handles key events; when waiting for data from a file, it only handles file.DATA/EOF/ERROR events and ignores all others. This approach is considerably simpler but also considerably less powerful.

Event types

Events belong to one of two categories: *Solicited Events* and *Unsolicited Events*.

Solicited Events are events which are generated in response to I/O requests (*e.g.* file Open/Read/Close calls). *Unsolicited Events* are generated by external devices such as keyboards, joysticks, and mice. *Unsolicited Events* are also generated whenever packets are received from the network.

Solicited Events

The *solicited events* are described in the *Kernel Calls* section below. The documentation for each kernel call that queues an event includes a description of the possible events.

Unsolicited Events

`event.key.PRESSED`
`event.key.RELEASED`

These events occur whenever a key is pressed or released:

- `event.key.keyboard` contains the id of the keyboard.
- `event.key.raw` contains the raw key-code (see `kernel/keys.asm`).
- `event.key.flags` is negative if this is a meta (non-ascii) key.
- `event.key.ascii` contains the ascii interpretation if available.

`event.mouse.DELTA`

This event is generated every time the mouse is moved or a button changes state.

- `event.mouse.delta.x` contains the x delta.
- `event.mouse.delta.y` contains the y delta.
- `event.mouse.delta.z` contains the z (scroll) delta.
- `event.mouse.delta.buttons` contains the button bits.

The buttons are encoded as follows:

- Bit 0 (1) is the inner-most button
- Bit 1 (2) is the middle button
- Bit 2 (4) is the outer-most button

The bits are set when the button is pressed and cleared when the button is released.
To change the "handedness" of the mouse, simply place it in whichever hand you prefer, and double-click!

`event.mouse.CLICKS`

This event reports mouse click (press-and-release) events. Whenever a mouse button is pressed, the kernel starts a 500ms timer and counts the number of times each button is both pressed and released. At the end of the 500ms, the counts are reported:

- `event.mouse.clicks.inner` contains the count of inner clicks.
- `event.mouse.clicks.middle` contains the count of middle clicks.
- `event.mouse.clicks.outer` contains the count of outer clicks.

A report of all zeros indicates that a press-and-hold is in progress; programs should consult the most recent event.`mouse.DELTA` event to determine which button(s) are being held.

`event.JOYSTICK+`

This event returns the state of the "buttons" for each of the two joysticks whenever either's state changes:

- `event.joystick.joy0` contains the bits for Joystick 0.
- `event.joystick.joy1` contains the bits for Joystick 1.

The bits are set when the associated switch is pressed, and clear when released.

`event.TCP`
`event.UDP`
`event.ICMP`

These events are generated whenever a network packet of the given type is received. For TCP and UDP packets, a program can use the `kernel.Net.Match` call to see if the packet matches an open

socket. Non-matching UDP packets can be ignored; network aware programs should respond to unmatched TCP packets by calling `kernel.Net.TCP.Reject`. TCP and UDP payloads may be read by calling `kernel.Net.TCP.Recv` and `kernel.Net.UDP.Recv` respectively. For ICMP packets, programs can get the raw data by calling `kernel.ReadData`.

Kernel Calls

Generic Calls

NextEvent

Copies the next event from the kernel's event queue into a user provided event buffer.

Input

- `kernel.args.events` points to an 8-byte buffer.

Output

- Carry cleared on success.
- Carry set if no events are pending.

Effect

- If there is an event in the queue, it is copied into the provided buffer.

Notes

- `kernel.args.events` is a reserved field in the argument block; nothing else uses this space, so you need only initialize this pointer once on startup.
- `kernel.args.pending` contains the count of pending events (negated for ease of testing with BIT). You can save considerable CPU time by testing `kernel.args.pending` and only calling `NextEvent` if it is non-zero.
- The kernel reports almost everything through events. Consider keeping the event buffer in the zeropage for efficient access.

ReadData

Copies data from the kernel buffer associated with the current event into the user's address space.

Input

- `kernel.args.buf` points to a user buffer.
- `kernel.args.buflen` contains the number of bytes to copy (0=256).

Output

- Carry cleared.

Effect

- The contents of the current event's primary buffer are copied into the provided user buffer.
- If the event doesn't contain a buffer, zeros are copied.

ReadExt

Copies extended data from the kernel buffer associated with the current event into the user's address space.

Input

- `kernel.args.buf` points to a user buffer.
- `kernel.args.buflen` contains the number of bytes to copy (0=256).

Output

- Carry cleared.

Effect

- The contents of the current event's primary buffer are copied into the provided user buffer.
- If the event doesn't contain an extended data buffer, zeros are copied.

Notes

- Events with extended data are relatively rare. Typical examples include file meta-information and bytes 256...511 of a 512 byte raw-sector read.

Yield

Yields the CPU to the kernel. This is typically used to expedite the processing of network packets. It is never required.

Input

- none

Output

- none

Effects

- none

RunBlock

Transfers execution to the program found in the given memory block.

Input

- `kernel.args.run.block_id` contains the number of the block to execute.

Output

- On success, the call doesn't return.
- Carry set on error (block doesn't contain a program).

RunNamed

Transfers execution to the first program found with the given name.

Input

- `kernel.args.buf` points to a buffer containing the name of the program to run.
- `kernel.args.buflen` contains the length of the name.

Output

- On success, the call doesn't return.
- Carry set on error (a program with the provided name was not found).

Notes

- The name match is case-insensitive.

FileSystem Calls

FileSystem.MkFS

Creates a new filesystem on the given device.

Input

- `kernel.args.fs.mkfs.drive` contains the device number (0 = SD, 1 = IEC #8, 2 = IEC #9)
- `kernel.args.fs.mkfs.label` points to a buffer containing the new drive label.
- `kernel.args.fs.mkfs.label_len` contains the length of the label buffer (0=0).
- `kernel.args.fs.mkfs.cookie` contains a user-provided cookie for matching against completion events.

Output

- Carry cleared on success.
- Carry set on error (device doesn't exist, kernel is out of streams).

Events

- The kernel will queue an `event.fs.CREATED` event on success.
- The kernel will queue an `event.fs.ERROR` event on failure.

Notes

- This is presently an atomic, blocking call on both IEC and Fat32, and it can take a *long* time to complete. While running, your program will not be able to work the event queue, so pretty much the whole system will grind to a halt. Until this changes, it's best if interactive programs and operating systems avoid this call.

File Calls

Open

Opens a file for read, append, or create/overwrite. The file should not be concurrently opened in another stream.

Input

- `kernel.args.file.open.drive` contains the drive ID (0 = SD card, 1 = IEC #8, 2 = IEC #9).
- `kernel.args.file.open.fname` points to the name of the file.
- `kernel.args.file.open.fname_len` contains the length of the name.
- `kernel.args.file.open.mode` contains the access mode (0 = read, 1 = write, 2 = append).
- `kernel.args.file.open.cookie` contains an optional, user provided value which will be returned in subsequent events related to this file.

Output

- Carry cleared on success, and `A` contains the stream ID for the file.
- Carry set if the drive isn't found, or if the kernel lacks sufficient resources to complete the call.

Events

- On a successful open/create/append, the kernel will queue a `file.OPENED` event.
- For read or append, if the file does not exist, the kernel will queue a `file.NOT_FOUND` event.
- File events contain the stream id (in `event.file.stream`) and the user supplied cookie (in `event.file.cookie`).

Notes

- The kernel supports a maximum of 20 concurrently opened files (including directories and rename/delete operations) across all devices.
- The IEC driver supports a maximum of 8 concurrently opened files (not counting directories and rename/delete operations) per device.
- Fat32 preserves case when creating files, and uses case-insensitive matching when opening files.
- IEC devices vary in their handling of case. Users will just need to live with this.
- Unlike other kernel calls, most file operations are blocking. In the case of IEC, this is due to the fact that an interrupt driven interface was not available at the time of writing (it is now, so IEC can be improved). In the case of Fat32, the fat32.s package we're using contains its own, blocking SPI stack (which may be replaced in the future). Fortunately, most operations are fast enough that events won't be dropped.
- The kernel does not lock files and does not check to see if a file is already in use. Should you attempt to concurrently open the file in two or more streams, the resulting behavior is entirely up to the device (IEC) or the file-system driver (fat32.s). The above statement that the file should not be concurrently opened should be treated as a strong warning.
- *Open for append is not yet implemented*

File.Read

Reads bytes from a file opened for reading.

Input

- `kernel.args.file.read.stream` contains the stream ID of the file (the stream is returned in the `file.OPENED` event and from the `File.Open` call itself).
- `kernel.args.file.read.buflen` contains the requested number of bytes to read.

Output

- Carry cleared on success.
- Carry set on error (file is not opened for reading, file is in the EOF state, the kernel is out of event objects).

Events

- On a successful read, the kernel will queue an `event.file.DATA` event (see `ReadData`). `event.file.data.requested` contains the number of bytes requested; `event.file.data.read` contains the number of bytes actually read.
- On EOF, the kernel will queue an `event.file.EOF` event.

- On error, the kernel will queue an `event.file.ERROR` event.
- In all cases, the event will also contain the stream id in `event.file.stream` and the the user's cookie in `event.file.cookie`.

Notes

- As with POSIX, the kernel may return fewer bytes than requested. This is not an error, and does not imply that the file has reached EOF (the kernel will queue an `event.file.EOF` on EOF). When this happens, the user is expected to simply call `File.Read` again to get more data.
- IEC devices don't report file-not-found until a read is attempted. To work around this issue, when talking to an IEC device, File.Open performs a one-byte read during the open. This single byte is later returned in the file's first `event.file.DATA` response (it does not attempt to merge this byte into the next full read).
- To reduce the risk of losing events while reading data from IEC devices, IEC read transfers are artificially limited to 64 bytes at a time. This limit may be lifted once we have interrupt driven IEC transfers.

File.Write

Writes bytes to a file opened for writing.

Input

- `kernel.args.file.write.stream` contains the stream ID of the file (the stream is returned in the `file.OPENED` event and from the `File.Open` call itself).
- `kernel.args.file.write.buf` points to the buffer to write.
- `kernel.args.file.write.buflen` contains the requested number of bytes to write.

Output

- Carry cleared on success.
- Carry set on error (file is not opened for writing, the kernel is out of event objects).

Events

- On a successful write, the kernel will queue an `event.file.WROTE` event.
  `event.file.wrote.requested` contains the number of bytes requested;
  `event.file.wrote.wrote` contains the number of bytes actually written.
- On error, the kernel will queue an `event.file.ERROR` event.
- In all cases, the event will also contain the stream id in `event.file.stream` and the user's cookie in `event.file.cookie`.

Notes

- As with POSIX, the kernel may write fewer bytes than requested. This is not an error. When it happens, the user is expected to simply call `File.Write` again to write more data.
- To reduce the risk of losing events while reading data from IEC devices, IEC write transfers are artificially limited to 64 bytes at a time. This limit may be lifted once we have interrupt driven IEC transfers.

File.Close

Closes an open file.

Input

- `kernel.args.file.close.stream` contains the stream ID of the file.

Output

- Carry cleared on success.
- Carry set on error (kernel is out of event objects).

Events

- The kernel will always queue an `event.file.CLOSED`, even if an I/O error occurs during the close. The event will contain the stream id in `event.file.stream` and the user's cookie in `event.file.cookie`.

Notes

- The kernel will always queue an `event.file.CLOSED`, even if an I/O error occurs during the close.
- Upon a successful call to `File.Close`, no further operations should be attempted on the given stream (the stream ID will be returned to the kernel's free pool for use by subsequent file operations).

File.Rename

Renames a file. The file should not be in use.

Input

- `kernel.args.file.rename.drive` contains the drive ID (0 = SD, 1 = IEC #8, 2 = IEC #9)
- `kernel.args.file.rename.cookie` contains a user supplied cookie for matching the completed event.

- `kernel.args.file.rename.old` points to a file path containing the name of the file to rename.
- `kernel.args.file.rename.old_len` contains the length of the path above.
- `kernel.args.file.rename.new` points to the new name for the file (*NOT* a new *path*!)
- `kernel.args.file.rename.new_len` contains the length of the new name above.

Output

- Carry cleared on success.
- Carry set on error (device not found; kernel out of events).

Events

- On successful completion, the kernel will queue a `file.RENAMED` event.
- On failure, the kernel will queue a `file.ERROR` event.
- In either case, `event.file.cookie` will contain the cookie supplied above.

Notes

- Rename semantics are up to the device (in the case of IEC) or the fat32.s driver (in the case of Fat32). The kernel doesn't even look at the arguments to see if they are sane. This means you may or may not be able to, say, change the case of letters in a filename with a single rename call; you may need to rename to a temp name, and then rename to the case-corrected name.

- Similarly, the kernel doesn't actually check if the file is in-use or not. Whether this matters is up to the device or file-system driver; treat the above statement that the file should not be in use as a strong recommendation.
- Rename is *NOT* a 'move'. You can only rename a file in place. If you want to move it, you will need to copy and then delete.

File.Delete

Deletes a file. The file should not be in use.

Input

- `kernel.args.file.delete.drive` contains the drive ID (0 = SD, 1 = IEC #8, 2 = IEC #9)
- `kernel.args.file.delete.cookie` contains a user supplied cookie for matching the completed event.
- `kernel.args.file.delete.fname` points to a file path containing the name of the file to delete.
- `kernel.args.file.delete.fname_len` contains the length of the path above.

Output

- Carry cleared on success.
- Carry set on error (device not found; kernel out of events).

Events

- On successful completion, the kernel will queue a `file.DELETED` event.
- On failure, the kernel will queue a `file.ERROR` event.
- In either case, `event.file.cookie` will contain the cookie supplied above.

Notes

- Case matching is up to the device (IEC) or the file-system driver (fat32.s).
- Delete semantics are up to the device (in the case of IEC) or the fat32.s driver (in the case of Fat32). The kernel doesn't even look at the path to see if it is sane.
- Similarly, the kernel doesn't actually check if the file is in-use or not. Whether this matters is up to the device or file-system driver; treat the above statement that the file should not be in use as a strong recommendation.

Directory Calls

Directory.Open

Opens a directory for reading.

Input

- `kernel.args.directory.open.drive` contains the device id (0 = SD, 1 = IEC #8, 2 = IEC #9).
- `kernel.args.directory.open.path` points to a buffer containing the path.
- `kernel.args.directory.open.lan_len` contains the length of the path above. May be zero for the root directory.
- `kernel.args.directory.open.cookie` contains a user-supplied cookie for matching the completed event.

Output

- Carry cleared on success; `A` contains the stream id.
- Carry set on error (device not found, kernel out of event or stream objects).

Events

- On successful completion, the kernel will queue an `event.directory.OPENED` event.
- On error, the kernel will queue an `event.directory.ERROR` event.
- In either case, `event.directory.cookie` will contain the above cookie.

Notes

- The IEC protocol only supports reading one directory per device at a time. The kernel does not, however, prevent you from trying. Consider yourself warned.

Directory.Read

Reads the next directory element (volume name entry, file entry, bytes-free entry).

Input

- `kernel.args.directory.stream` contains the stream id.

Output

- Carry cleared on success.
- Carry set on error (EOF has occurred or the kernel is out of event objects).

Events

- The first read will generally queue an `event.directory.VOLUME` event. `event.directory.volume.len` will contain the length of the volume name. Call `ReadData` to retrieve the volume name.

- Subsequent reads will generally queue an `event.directory.FILE` event. `event.directory.file.len` will contain the length of the filename. `ReadData` will retrieve the file name; `ReadExt` will retrieve the meta-data (presently just the sector count).
- The last read before EOF will generally queue an `event.directory.FREE` event. `event.directory.free.free` will contain the number of free sectors on the device.
- The final read will queue an `event.directory.EOF` event.
- Should an error occur while reading the directory, the kernel will queue an `event.directory.ERROR` event.

Notes

- The IEC protocol does not support multiple concurrent directory reads.
- Attempting to open files while reading an IEC directory have been known to result in directory read errors (cc65 errata).
- SD2IEC devices cap the sectors-free report at 65535 sectors.

Directory.Close

Input

- `kernel.args.directory.stream` contains the stream id.

Output

- Carry cleared on success.
- Carry set on error (kernel is out of event objects).

Events

- The kernel will always queue an `event.directory.CLOSED` event, even if an error should occur.

Notes

- Do not attempt to make further calls against the same stream id after calling `Directory.Close`—the stream will be returned to the kernel for allocation to subsequent file operations.

Network Calls—Generic

Network.Match

Determines if the packet in the current event belongs to the provided socket.

Input

- `kernel.args.net.socket` points to either a TCP or UDP socket.

Output

- Carry cleared if the socket matches the packet.
- Carry set if the socket does not match the packet.

Network Calls—UDP

Network.UDP.Init

Initializes a UDP socket in the user's address space.

Input

- `kernel.args.net.socket` points to a 32 byte UDP socket structure.
- `kernel.args.net.dest_ip` contains the destination address.
- `kernel.args.net.src_port` contains the desired source port.
- `kernel.args.net.dest_port` contains the desired destination port.

Output

- Carry clear (always succeeds).

Events

- None

Notes

- Opening a UDP socket *does not* create a packet filter for the particular socket. Instead, ALL UDP packets received by the kernel are queued as events; it is up to the user program to accept or ignore each in turn.

Network.UDP.Send

Writes data to a UDP socket.

Input

- `kernel.args.net.socket` points to a 32 byte UDP socket structure.
- `kernel.args.net.buf` points to the send buffer.
- `kernel.args.net.buf_len` contains the length of the buffer (0 = 256, but see the notes section)

Output

- Carry cleared on success.
- Carry set on error (kernel is out of packet buffers).

Events

- None

Notes

- By design, the kernel limits all network packets to 256 bytes. This means the maximum payload for a single UDP packet is 228 bytes.
- **Bug**: the kernel doesn't currently stop you from trying to send more than 228 bytes. Attempts to do so will result in corrupt packets.

Network.UDP.Recv

Reads the UDP payload from an `event.network.UDP` event.

Input

- `kernel.args.net.buf` points to the receive buffer.
- `kernel.args.net.buflen` contains the size of the receive buffer.

Output

- Carry cleared on success; `kernel.args.net.accepted` contains the number of bytes copied from the event (0 = 0).

- Carry set on failure (event is not a network.UDP event).

Notes

- The full packet may be read into the user's address space by calling `kernel.ReadData`.

Network Calls—TCP

Network.TCP.Open

Initializes a TCP socket in the user's address space.

Input

- `kernel.args.net.socket` points to a 256 byte TCP socket structure (includes a re-transmission queue).
- `kernel.args.net.dest_ip` contains the destination address.
- `kernel.args.net.dest_port` contains the desired destination port.

Output

- Carry clear (always succeeds).

Events

- None

Notes

- Opening a TCP socket *does not* create a packet filter for the particular socket. Instead, ALL TCP packets received by the kernel are queued as events; it is up to the user program to accept or reject each in turn; see `Network.Match`.

Network.TCP.Accept

Accepts a new connection from the outside world.

Notes

- *Not yet implemented.*

Network.TCP.Reject

Rejects a connection from the outside world. May also be used to forcibly abort an existing connection.

Notes

- *Not yet implemented.*

Network.TCP.Send

Writes data to a TCP socket.

Input

- `kernel.args.net.socket` points to the socket.
- `kernel.args.net.buf` points to the send buffer.
- `kernel.args.net.buf_len` contains the length of the buffer (0 = 0; may be used to force retransmission).

Output

- Carry cleared on success; `kernel.args.net.accepted` contains the number of byte accepted.
- Carry set on error (socket not yet open, user has closed this side of the socket, kernel is out of packet buffers).

Events

- None

Notes

- The socket presently contains a 128 byte transmit queue. This queue is forwarded by both `Network.TCP.Send` and by `Network.TCP.Recv`. The remote host must ACK the bytes in the transmit queue before more bytes become available in the queue.

Network.TCP.Recv

- Reads the TCP payload from an `event.network.TCP` event.
- ACKs valid packets from the remote host.
- Maintains the state of the socket.

Input

- `kernel.args.net.socket` points to the socket.
- `kernel.args.net.buf` points to the receive buffer.
- `kernel.args.net.buflen` contains the size of the receive buffer.

Output

- Carry cleared on success; `kernel.args.net.accepted` contains the number of bytes copied from the event (0 = 0). `A` contains the socket state.
- Carry set on failure (event is not a network.TCP event).

Notes

- The full packet may be read into the user's address space by calling `kernel.ReadData`.

Network.TCP.Close

Tells the remote host that no more data will be sent from this side of the socket. The remote host is, however, free to continue sending until it issues a close.

Input

- `kernel.args.net.socket` points to the socket.

Output

- Carry cleared on success.
- Carry set on failure (kernel is out of buffers).

Display Calls

Display.Reset

Resets the display resolution and colors and disables the mouse and cursor. Does NOT reset the font.

Input

- None

Output

- None

Notes

- In the future, the kernel should include support for re-initializing the default fonts.

Display.GetSize

Returns the size of the current text display.

Input

- None

Output

- `kernel.args.display.x` contains the horizontal size.
- `kernel.args.display.y` contains the vertical size.

Display.DrawRow

Copies the user provided text buffer to the screen (from left to right) starting at the provided coordinates and using the provided color buffer.

Input

- `kernel.args.display.x` contains the starting x coordinate.
- `kernel.args.display.y` contains the starting y coordinate.
- `kernel.args.display.text` points to the text data.
- `kernel.args.display.color` points to the color data.
- `kernel.args.buflen` contains the length of the buffer.

Output

- Carry cleared on success.
- Carry set on error (x/y outside of the screen)

Notes

- Probably isn't yet checking the coordinate bounds or clipping.

Display.DrawColumn

Copies the user provided text buffer to the screen (from top to bottom) starting at the provided coordinates and using the provided color buffer.

Input

- `kernel.args.display.x` contains the starting x coordinate.
- `kernel.args.display.y` contains the starting y coordinate.
- `kernel.args.display.text` points to the text data.
- `kernel.args.display.color` points to the color data.
- `kernel.args.buflen` contains the length of the buffer.

Output

- Carry cleared on success.
- Carry set on error (x/y outside of the screen)

Notes

- *Not yet implemented.*

24 References

It is not possible to cover all the details of all the chips that are part of the F256jr in this one reference manual. This chapter lists links to all the data sheets for each chip used or implemented as well as some other useful websites. These chips include the 65C02 (CPU), 65C22 (VIA), WM8776 (sound CODEC), bq4802ly (real time clock), 6581 (SID), and the SN76489 (PSG).

CPU https://www.westerndesigncenter.com/wdc/documentation/w65c02s.pdf

VIA https://www.westerndesigncenter.com/wdc/documentation/w65c22.pdf

CODEC https://web.archive.org/web/20140801092610/http://www.wolfsonmicro.com/media/

76476/WM8776.pdf

RTC https://www.ti.com/lit/ds/symlink/bq4802y.pdf

SID http://archive.6502.org/datasheets/mos_6581_sid.pdf

PSG http://www.vgmpf.com/Wiki/images/7/78/SN76489AN_-_Manual.pdf

The website for all Foenix information is https://c256foenix.com, and the latest electronic copy of this manual is at https://github.com/pweingar/C256jrManual

25 Copyright