REU PLUS DESCRIPTION

Table of Contents

1. Introduction 3
2. REU 3
3. I2C Peripheral 3
   3.1. Example code 4
      3.1.1. START command 4
      3.1.2. Writing Multiple Bytes 4
      3.1.3. Reading Multiple Bytes 6
   3.2. I2C C64 Registers 7
4. Text Mode Video Peripheral 9
   4.1. Description 9
      4.1.1. Writing to the Video Interface 9
      4.1.2. Video C64 Registers 10
      4.1.3. Text Color Jumpers 11
   4.2. Control Codes 12
5. Real Time Clock 13
6. I2C EEPROMs 13
7. Boot ROM 13
8. Support Software 14
   8.1. BIN2MIF 14
9. Register Map 15
   9.1. REU Registers 17
      9.1.1. REU Status $DF00 R 17
      9.1.2. REU Command $DF01 R/W 17
      9.1.3. REU C64 Address High/Low $DF03/$DF02 R/W 18
      9.1.4. REU Expansion Address High/Mid/Low $DF06/$DF05/$DF04 R/W 18
      9.1.5. REU Transfer Length High/Low $DF08/$DF07 R/W 18
      9.1.6. REU Interrupt Mask Register $DF09 R/W 19
      9.1.7. REU Address Control Register $DF0A R/W 19
      9.1.8. REU Expansion Address $DF0E R 19
      9.1.9. REU Superbank $DF0F R/W 20

9.2. ROM/RAM Mapping Registers 20
9.2.1. ROML Base Address $DF11/DF10 R/W 20
9.2.2. ROMH Base Address $DF13/DF12 R/W 21
9.2.3. RAM Base Address $DF16/DF15/DF14 R/W 21
9.2.4. ROM Select Register $DF28 R/W 21
9.3. Video Peripheral Registers 22
9.3.1. VID_STAT_REG $DF20 R 22
9.3.2. VID_CHAR_REG $DF21 R/W 22
9.4. Coprocessor Control Registers 24
9.4.1. Mailbox Bank Register $DF26 R/W 24
9.4.2. Mailbox Bank Register $DF30..$DF3F R/W 24
9.4.3. Coprocessor Control Register $DF2F R/W 24
9.5. I2C Control Registers 24
9.5.1. I2C Command Register $DF2A R/W 24
9.5.2. I2C_DATARD_REG $DF2B R/W 26
9.5.3. I2C Data Write Register $DF2C R/W 26
9.6. Miscellaneous Registers 26
9.6.1. FPGA Revision Register $DF2E R 26

1. Introduction

2. REU

3. I2C Peripheral

The I2C peripheral is used to communicate with the real time clock (RTC) and the 256k byte EEPROMs. The peripheral is implemented as a finite state machine (FSM) that is controlled by writing a command to a C64 register. The I2C bus operates at a clock speed of 400kHz which means that transferring a byte takes a minimum of 22.5µs.

In general, an I2C transaction consists of four major types of operations: START, STOP, SEND, and RECEIVE.

- The START operation is accomplished by writing $01 to the I2C_CMD_REG. The FSM waits until the C64 clock goes low and then performs the operations necessary to generate the START command. When the FSM has completed all operations, the FSM sets the I2C_CMD_REG to $00 (NOP) and then stops.

- The STOP operation is accomplished by writing $02 to the I2C_CMD_REG. The FSM waits until the C64 clock goes low and then performs the operations necessary to generate the STOP command. When the FSM has completed all operations, the FSM sets the I2C_CMD_REG to $00 (NOP) and then stops.

- The SEND command is started by writing $03 to the I2C_CMD_REG. The FSM waits until the C64 clock goes low and then transitions to wait for data to be written to the I2C_DATAWR_REG. When a byte has been written to the write data register, the FSM waits until the C64 clock goes low and then starts shifting the value in the I2C_DATAWR_REG out on the I2C bus MSB first. The FSM then issues a 9th clock to allow the I2C peripheral to provide an ACK bit to the FSM. The FSM latches the ACK bit into a register and then waits for the C64. If the C64 writes another value to the I2C_DATAWR_REG, that data will be clocked out on the I2C bus and the new ACK bit latched.

  To terminate the SEND state, the C64 must write a different command to I2C_CMD_REG (usually NOP). This will cause the FSM to exit the SEND state and enter the next command state (including the NOP command state).

- Reading from the I2C bus requires two different types of operations that differ by the logic value sent to the I2C peripheral during the ACK phase of the read. As a consequence, the I2C FSM has a command that corresponds to these two conditions.

  When the C64 writes either the RDACK or RDNACK command to the I2C peripheral, the FSM enters the state that will receive a data byte from the I2C peripheral. When the

C64 reads from the I2C_DATARD_REG, the FSM will send eight clock pulses to the I2C bus to read in a data byte from the I2C peripheral. If the command is RDACK, the FSM will issue a '0' (I2C bus ACK) to the I2C peripheral during the ACK phase of the read. Alternatively, if the command is RDNACK, the FSM issues a '1' (I2C bus NACK) to the I2C peripheral. The NACK is required to complete reading data from the I2C peripheral even if only one byte is to be read.

If multiple bytes are to be read, the program must read all but the last byte using the RDACK command and then read the final byte with the RDNACK command. Alternatively, the program can read all bytes with the RDACK command and perform a dummy read with the RDNACK command and discard the data byte. Writing the RDNACK command to the FSM can be done as required and does not require writing a NOP command.

It is important to know that the first byte read from the I2C_DATARD_REG usually will not be the desired data byte since the FSM has not shifted anything from the I2C device into its receive register. This means that the program must read from the I2C_DATARD_REG register and discard the result. Subsequent reads from the register will provide data that has been shifted in from the I2C peripheral.

Accessing the I2C FSM from BASIC will generally not require testing to determine if the I2C FSM has completed an operation as BASIC is slow enough that the operation will have completed before BASIC can issue another request. Testing of the ACK bit will still be required during C64 reads, however.

3.1. Example code

3.1.1. START command

```
    lda #$01
    sta I2C_CMD_REG
waitLoop:
    lda I2C_CMD_REG
    ora #7
    bne waitLoop
```

The first two instructions write the command to the I2C FSM to initiate the start command. The loop reads the command register and masks off the command bits. If the command is not NOP (zero) the program loops until it is. This works since the FSM will set the command to NOP when the START command has completed.

The same code can be used to issue a STOP command by loading A with $02

3.1.2. Writing Multiple Bytes

```
    lda #$03
```

```
        sta I2C_CMD_REG

sendLoop:
        lda (buffer), y
        sta I2C_DATAWR_REG

waitLoop:
        lda I2C_CMD_REG
        and #$80
        beq waitLoop

        lda I2C_CMD_REG
        and #$40
        bne ackErr

        iny
        dex
        bne sendLoop

ackErr:
        pha
        lda #$00
        sta I2C_CMD_REG
        pla
        rts
```

This code assumes:

- (buffer) points to the where the transmit data resides
- Y points to the offset in the buffer at which to start
- X is the number of bytes to send
- All other setup required for the I2C device has been done. This only sends the bytes.

Summary of what the code is doing:

- The first two lines set up the FSM to send the bytes to the I2C device.
- The sendLoop bytes write the data to the output shift register. The FSM then starts shifting out the byte.
- waitLoop reads the status byte and masks off the i2cShifterRdy bit and if it is not set branches back to read it again. When the FSM has completed shifting out the byte, it will set the i2cShifterRdy and we can move on.
- The next three instructions read the ACK bit back and if it is not zero the code branches to an error exit. If the ACK was good, we adjust the variables and then loop back for more bytes.
- The exit routine saves the value in A and then writes a NOP to the I2C FSM to end the command. It then restores A and returns. For this routine, A will be $00 if all went well or $40 if there was an ACK error.

3.1.3. Reading Multiple Bytes

```
        lda #$04
        sta I2C_CMD_REG

        lda I2C_DATARD_REG
        jsr waitLoop

getByte:
        lda I2C_DATARD_REG
        sta (buffer), y
        jsr waitLoop

        iny
        dex
        bne getByte

        lda #$05
        sta I2C_CMD_REG
        jsr waitLoop
        rts

waitLoop:
        lda I2C_CMD_REG
        and #$80
        beq waitLoop
        rts
```

This code assumes:

- (buffer) points to the where the transmit data resides
- Y points to the offset in the buffer at which to start
- X is the number of bytes to read
- All other setup required for the I2C device has been done. This only sends the bytes.
- A dummy read to complete the transaction is acceptable.

Summary of what the code is doing:

- The first two instructions set up the I2C FSM to read data and provide an ACK to the I2C device.
- The next two instructions perform a dummy read to get the first byte of data from the I2C device. This value is ignored by the code but, importantly, the I2C receive register will have the first data byte after this read completes.

- The getByte loop reads a byte from the I2C FSM receive register and stores it in the buffer. It then updates the Y pointer and the X byte counter and then loops back to read more bytes if necessary.
- Once all of the bytes have been read, the I2C FSM is set up to do a read with a NACK and a byte is read. This will terminate the read transaction and the code then exits.

3.2. I2C C64 Registers

3.2.1.1. I2C_CMD_REG DF2A (57130) Read/Write

<table>
<tr><th colspan="4">I2C_CMD_REG DF2A (57130) Write</th></tr>
<tr><td>Bit</td><td colspan="3">Function</td></tr>
<tr><td>7</td><td colspan="3">unused</td></tr>
<tr><td>6</td><td colspan="3">unused</td></tr>
<tr><td>5</td><td colspan="3">unused</td></tr>
<tr><td>4</td><td colspan="3">unused</td></tr>
<tr><td>3</td><td colspan="3">unused</td></tr>
<tr><td>2
1</td><td>Command Code</td><td>Name</td><td>Function</td></tr>
<tr><td rowspan="5">0</td><td>$00</td><td>NOP</td><td>The I2C FSM does nothing</td></tr>
<tr><td>$01</td><td>START</td><td>Sends an I2C START command. The I2C FSM resets the command code to NOP after the START command completes</td></tr>
<tr><td>$02</td><td>STOP</td><td>Sends an I2C STOP command. The I2C FSM resets the command code to NOP after the STOP command completes</td></tr>
<tr><td>$03</td><td>SEND</td><td>Sets the I2C FSM to send bytes on the I2C bus. The SEND command also performs the ACK operation and the ACK bit is latched into the status byte.
Note: no byte is actually clocked out until it is written to the I2C_DATWR_REG.
Note: the I2C FSM will stay in this state until another command code is written to this register.</td></tr>
<tr><td>$04</td><td>RDACK</td><td>Sets the I2C FSM to receive bytes on the I2C bus. The RDACK command also sends an ACK to the I2C peripheral.
Note: no byte is actually clocked in until a byte is read from the I2C_DATRD_REG.</td></tr>
</table>

			Note: the I2C FSM will stay in this state until another command code is written to this register.
	$05	RDNACK	Sets the I2C FSM to receive bytes on the I2C bus. The RDACK command also sends an NACK to the I2C peripheral. Note: no byte is actually clocked in until a byte is read from the I2C_DATRD_REG. Note: the I2C FSM will stay in this state until another command code is written to this register.
	$06 & $07	unused	Either of these values will do nothing or will cause the I2C FSM to return to the idle state.
Reset value: $00			

I2C_CMD_REG DF2A (57130) Read	
Bit	Function
7	'1' – if the I2C shifter is ready to send/receive a byte '0' – if the I2C is busy shifting out a byte
6	'1' – the I2C peripheral returned a NACK during a write to it '0' – the I2C peripheral returned an ACK during a write to it
5	'1' – the I2C FSM is in the idle state
4	The logical state of the I2C SDA pin on the FPGA. This allows the C64 to interrogate the actual state of the I2C bus.
3	The logical state of the I2C SCL pin on the FPGA. This allows the C64 to interrogate the actual state of the I2C bus.
2	Returns the command value
1	
0	
Reset value: N/A	

3.2.1.2. I2C_DATARD_REG DF2B (57131) Read

I2C_DATARD_REG DF2B (57131) Read	
Bits	Function
7..0	Reading from this register when the I2C FSM is in the RDACK or RDNACK modes, will return the previous value shifted in on the I2C bus and will shift in the next byte from the I2C bus.
Reset value: N/A	

3.2.1.3. I2C_DATAWR_REG DF2C (57132) Read/Write

I2C_DATAWR_REG DF2C (57132) Read/Write	
Bits	Function
7..0	Writing to this register when the I2C FSM is in the SEND mode will shift out the C64 byte onto the I2C bus. Reading from this register will return any previously written value.
Reset value: $00	

4. Text Mode Video Peripheral

4.1. Description

The text video module is a monochrome 80 column x 25 line text video that outputs a 640X480 VGA compatible signal. The module is adapted from Grant Searle's Multicomp terminal peripheral (http://searle.x10host.com/Multicomp/index.html). The modification stripped out the keyboard section and modified the video section for monochrome video.

The font is set at FPGA compile time and is fixed.

4.1.1. Writing to the Video Interface

Under most conditions, writing a character to the video interface is relatively fast and would not require the C64 wait for the video peripheral but the response will slow considerably if scrolling or other actions such as deleting or adding lines is requested. It is recommended that when a character is written to VID_CHAR_REG that the C64 check bit 1 of VID_STAT_REG for a '1.' If the value of the bit is zero, the C64 should wait for the bit to go to a '1' before sending the next character. For example, this code snippet shows an example:

This assumes that the subroutine is called with A containing the character to write and that A can be clobbered.

```
writeChar:
        sta VID_CHAR_REG
waitLoop:
        lda VID_STAT_REG
        bpl waitLoop
        rts
```

This code could be modified to perform the availability check prior to writing the character.

4.1.2. Video C64 Registers

The video peripheral is controlled by registers in the IO2 space.

Address	Name	Function
DF2F (57135)	COP_SEL_REG	Selects whether the coprocessor or the C64 has access to the video peripheral
DF20 (57120)	VID_STAT_REG	Shows the status of the video peripheral
DF21 (57121)	VID_CHAR_REG	Used to write data to the video peripheral

4.1.2.1. COP_SEL_REG DF2F (57135) Read/Write

COP_SEL_REG DF2F (57135) Read/Write	
Bit	Function
7	'0' – the C64 and REU have control of the SDRAM '1' – the coprocessor has control of the SDRAM
6	'0' – the C64 has control of the video peripheral '1' – the coprocessor has control of the video peripheral
5	'0' – resets the Z80 '1' – removes the reset from the Z80
4	'0' – prevents the Z80 from running '1' – allows the Z80 to run This bit does NOT reset the Z80
3	Read: the status of the CEN signal that goes to the Z80. See description of what this does
2	unused
1	unused
0	'0' – the REU is enabled '1' – the REU is disabled
Reset value: $00	

4.1.2.2. VID_STAT_REG DF20 (57120) Read

VID_STAT_REG DF20 (57120) Read	
Bit	Function
7	'1' – the video peripheral is available
6	Unused
5	Unused
4	Unused

VID_STAT_REG DF20 (57120) Read	
Bit	Function
3	‘0’
2	‘0’
1	‘1’ – the video peripheral has issued an interrupt This bit is not connected in the FPGA
0	unused

4.1.2.3. VID_CHAR_REG DF21 (57121) Read/Write

Write – the character to be written to the video peripheral

Read

VID_CHAR_REG DF21 (57121) Read	
Bit	Function
7	‘0’
6	‘0’
5	‘0’
4	‘0’
3	‘1’ – flash attribute is active
2	‘1’ – strikethrough attribute is active
1	‘1’ – underline attribute is active
0	‘1` - inverse attribute is active

4.1.3. Text Color Jumpers

The color of the text is selectable using the jumpers on the board. The background color is always black while the foreground colors selected by connecting the OUT pins to either the IN or GND pins.

Color	R	G	GL	B
Red	IN	GND	OPEN	GND
Green	GND	IN	OPEN	GND
Blue	GND	GND	OPEN	IN
White	IN	IN	OPEN	IN
Yellow	IN	IN	OPEN	GND
Amber	IN	OPEN	IN	GND
Cyan1	GND	IN	OPEN	IN
Cyan2	GND	OPEN	IN	IN

Color	R	G	GL	B
Magenta	IN	GND	OPEN	IN

The best colors will likely be white, amber and green and possibly yellow.

4.2. Control Codes

The following table lists the control codes that the text video uses to control it operation.

Code	Name	Result
$12	Clear screen	Clear the screen and set the cursor to the top left corner of the screen.
$0A	Line feed	Line feed – moves the cursor down one line in the current column
$0D	Carriage return	Carriage return – moves the cursor to the start of the current line.
ESC[H	Cursor home	Home the cursor
ESC[K	Erase EOL	Erase to then end of the current line
ESC[s	Save cursor position	Save the current cursor row and column. The current cursor position overwrites any previously save cursor position.
ESC[u	Restore cursor position	Returns the cursor to the save position
ESC[2J	Clear screen	Clear the screen and set the cursor to the top left corner of the screen.
ESC[0J or ESC[J	Clear to end of screen	Fill the current line with spaces from the cursor position to the end of the screen.
ESC[{val}A	Cursor Up	Move the cursor up {val} places
ESC[{val}B	Cursor Down	Move the cursor down {val} places
ESC[{val}C	Cursor forward	Move the cursor to the right {val} places
ESC[{val}D	Cursor back	Move the cursor to the left {val} places
ESC[{row};{col}H	Set cursor position	Directly move the cursor to {row}, {col}. The top left corner is at 1;1.
ESC[L	Delete line	Delete the line on which the cursor resides. All lines below are move up. This command only deletes one line.
ESC[M	Insert line	Insert a blank line at the current cursor row. All lines below the cursor line are moved down. This command only adds one line.
ESC[{val}m	Change attribute	Set or clear the text mode attributes: val: 7 – enable inverse val: 8 – disable inverse

Code	Name	Result		
		9	enable underline	
		10	disable inverse	
		11	enable strikethrough	
		12	disable strikethrough	
		13	enable flashing	
		14	disable flashing	
Notes: 1. ESC is sent as $1B 2. All numeric values in the curly brackets ({}) are ASCII digits and not binary values. For instance, to move the cursor up two lines, send ESC[2A ($1B$5B$32$41)				

5. Real Time Clock

The real time clock uses a DS3231SN# integrated circuit. The RTC incorporates an internal crystal and only requires power and a 3V lithium battery to maintain time when the board is powered down. The RTC keeps time to the second in 12 or 24 hour format and has a calendar function. The RTC also has two alarms that generate an interrupt when they expire. Since the REUPlus does not connect the interrupt to the C64, this functionality is not used.

The RTC is controlled via the I2C peripheral. See the DS3231SN# data sheet for detailed instructions on managing the device.

A program is included that allows reading and setting the time and calendar registers in the DS3231SN#. The program has no provision for setting or reading the alarm registers since they are not useful in the REUPlus2C. Both the executable and CC65 compatible source files are included.

6. I2C EEPROMs

The REUPlus incorporates a pair of AT24CM02-SSHM-T 256k byte EEPROMs for data storage providing a total of 512k bytes of erasable non-volatile storage. The devices are accessed by the I2C bus.

See the AT24CM02-SSHM-T data sheet for details on accessing the EEPROMS.

7. Boot ROM

The FPGA contains a small 1kByte ROM this is enabled as EXROM when the REUPlusC2 is reset. This ROM is used primarily to load an executable program from the EEPROM.

Note: This functionality is currently disabled but the capability is still available.

8. Support Software

The support software is used to generate various files used during development. All of the software was written in Power Basic as console applications and they run under Windows.

8.1. BIN2MIF

This utility converts a binary or Intel Hex file to a format compatible with the Quartus tools for initializing the boot ROM in the FPGA. The program accepts two required parameters and one optional parameter.

Usage: bin2Mif -fFileName -rRamSize [-b|-h]
Note: all parameters are case insensitive

-f = The name of the file to convert (required).
-r = The size of the FPGA ROM (required).
This value must be at least 1 and less than 4097.
Decimal or hexadecimal values are accepted with hexadecimal values being preceded with a $ or 0x.

The size of output file will be rounded up to the next 256 byte page value but will not exceed the size given in the -r parameter

These are optional:
-b = The input file is binary (default).
-h = The input file is Intel hex format.
The program assumes the hex file is valid and well formatted. Nonsense will be produced if this is not true.

Any binary code exceeding the size given in the -r parameter will be truncated

8.1. Concat

This utility concatenates a data file to a 6502 assembler routine the writes the file data to the REUPlus EEPROM. Because of this and the maximum memory of the C64, the maximum size file one can concatenate will be 32768 bytes and the program will enforce this limit by refusing to concatenate a larger file

All command line parameters are case insensitive as they are converted to uppercase in the program. This is Windows, eh?

This file uses pgmEeprom.bin, so ensure it is in the same directory as this file.

Usage:

Example: concat -n foo -e $001234

-n = the input file name
-e = the EEPROM address where the file is to be written
The program willa accept addresses in decimal or hexadecimal if the hex values are preceded with $ or 0x

9. Register Map

Address	Address (Hex)	Name
57088	DF00	REU Status
57089	DF01	REU Control
57090	DF02	REU C64 Address Low
57091	DF03	REU C64 Address High
57092	DF04	REU Expansion Address Low
57093	DF05	REU Expansion Address Mid
57094	DF06	REU Expansion Address High
57095	DF07	REU Transfer Length Low
57096	DF08	REU Transfer Length Mid
57097	DF09	REU interrupt Mask Register
57098	DF0A	REU Address Control Register
57099	DF0B	Unused
57100	DF0C	Unused
57101	DF0D	Unused
57102	DF0E	REU Expansion Address
57103	DF0F	REU Superbank
57104	DF10	ROML_BASELOW
57105	DF11	ROML_BASEHIGH
57106	DF12	ROMH_BASELOW
57107	DF13	ROMH_BASEHIGH
57108	DF14	RAM_BASELOW
57109	DF15	RAM_BASEMID
57110	DF16	RAM_BASEHI
57111	DF17	Unused
57112	DF18	Unused
57113	DF19	Unused
57114	DF1A	Unused
57115	DF1B	Unused
57116	DF1C	Unused
57117	DF1D	Unused
57118	DF1E	Unused
57119	DF1F	Unused

Address	Address (Hex)	Name
57120	DF20	VID_STAT_REG
57121	DF21	VID_CHAR_REG
57122	DF22	Unused
57123	DF23	Unused
57124	DF24	Unused
57125	DF25	Unused
57126	DF26	MBOX_BANKREG
57127	DF27	Z80CTL_REG
57128	DF28	ROMSEL_REG
57129	DF29	Unused
57130	DF2A	I2C_CMD_REG
57131	DF2B	I2C_DATRD_REG
57132	DF2C	I2C_DATWR_REG
57133	DF2D	Unused
57134	DF2E	FPGA_REV_REG
57135	DF2F	COP_SEL_REG
57136	DF30	MB0_REG
57137	DF31	MB1_REG
57138	DF32	MB2_REG
57139	DF33	MB3_REG
57140	DF34	MB4_REG
57141	DF35	MB5_REG
57142	DF36	MB6_REG
57143	DF37	MB7_REG
57144	DF38	MB8_REG
57145	DF39	MB9_REG
57146	DF3A	MB10_REG
57147	DF3B	MB11_REG
57148	DF3C	MB12_REG
57149	DF3D	MB13_REG
57150	DF3E	MB14_REG
57151	DF3F	MB15_REG

9.1. REU Registers

9.1.1. REU Status $DF00 R

Reset value: $10
Status register, read-only, bits 7-5 are automatically cleared on reading the register

REU Status $DF00			
Bit #	Read/Write	Function	Mode
7	R/Clear	Interrupt pending	1 – if any interrupt is pending (bits 6-5), this requires configuration of an interrupt in the interrupt mask register 0 – otherwise
6	R/Clear	End of block	1 – if a transfer is completed 0 – otherwise
5	R/Clear	Verify error	1 – if a verify operation stopped with an error 0 – otherwise
4	R	Size	Always 1
3..0	R	Chip version	Always '0000'

9.1.2. REU Command $DF01 R/W

Reset value: $10
Command register, read/write

REU Command $DF01			
Bit #	Read/Write	Function	Mode
7	R/W	Execute	1 – a transfer is started either immediately or deferred (using 0xFF00 trigger option) 0 – transfer disabled. The bit is automatically set back to 0, when a transfer took place.
6	R/W	Reserved	The bit is backed by a register. The value last written to it is given back on reading again. Any REU operation does not change its state.
5	R/W	Autoload	1 – if autoloading registers (addresses, length) should take place after a transfer 0 – if the registers should remain on their last counting state
4	R/W	FF00 Trigger	1 – 0xFF00 trigger option is disabled, transfers are started immediately by setting bit 7 0 – 0xFF00 trigger option is enabled, transfers are started, when bit 7 is set also and a write access to address 0xFF00 occurs. The bit is automatically set back to 1, when a transfer took place.

REU Command $DF01			
Bit #	Read/Write	Function	Mode
3..2	R/W	Reserved	The bits are backed by a register. The values last written to is given back on reading again. Any REU operation does not change their state.
1..0	R/W		00 – data is transferred from C64/C128 to REU 01 – data is transferred from REU to C64/C128 10 – data is exchanged between C64/C128 and REU 11 – data is verified between C64/C128 and REU

9.1.3. REU C64 Address High/Low $DF03/$DF02 R/W

Reset to $0000
These registers hold the start address in the C64 at which the REU start will its operation

REU C64 Address Low $DF02		
Bit #	Read/Write	Function
7..0	R/W	C64 base address LSB, low 8 bits of C64 base address

REU C64 Address High $DF03		
Bit #	Read/Write	Function
7..0	R/W	C64 base address MSB, high 8 bits of C64 base address

9.1.4. REU Expansion Address High/Mid/Low $DF06/$DF05/$DF04 R/W

Reset to $F80000
These registers hold the 24 bit start address in the SDRAM at which the REU will start its operation

REU Expansion Address Low $DF04		
Bit #	Read/Write	Function
7..0	R/W	SDRAM base address LSB, low 8 bits of SDRAM base address

REU Expansion Address Mid $DF05		
Bit #	Read/Write	Function
7..0	R/W	SDRAM base address MIDDLE, middle 8 bits of SDRAM base address

REU Expansion Address HIgh $DF06		
Bit #	Read/Write	Function
7..0	R/W	SDRAM base address MSB, high 8 bits of SDRAM base address
7..3	R	Always returns ‘11111’ This is for compatibility with legacy software. Read the REU Expansion register ($DF0E) to read all 8 bits
7..0	W	Sets all 8 high order bits of the 24 bit address

9.1.5. REU Transfer Length High/Low $DF08/$DF07 R/W

Reset to $FFFF
These registers hold the 24 bit start address in the SDRAM at which the REU will start its operation. Bit 25 is held in the Superbank register ($DF0F).

REU Transfer Length Low $DF07		
Bit #	Read/Write	Function
7..0	R/W	Transfer length LSB, lower 8 bits of the byte counter

REU Transfer Length Low $DF08		
Bit #	Read/Write	Function
7..0	R/W	Transfer length MSB, upper 8 bits of the byte counter

9.1.6. REU Interrupt Mask Register $DF09 R/W

Reset to $1F
Interrupt mask register

REU Interrupt Mask $DF09			
Bit #	Read/Write	Function	Mode
7	R/W	Interrupt enable	1 – if REU interrupts are enabled 0 – otherwise
6	R/W	End of block mask	1 – if end of block interrupts is enabled 0 – otherwise
5	R/W	Verify error mask	1 – if verify error interrupt is enabled 0 – otherwise
4..0	R/W	Unused	Always 1

9.1.7. REU Address Control Register $DF0A R/W

Reset to $3F
Interrupt mask register

REU Address Control $DF0A			
Bit #	Read/Write	Function	Mode
7..6	R/W	Addressing Type	00 – both addresses are incremented on transfers (default) 01 – the REU base address is fixed 10 – the C64/C128 base address is fixed 11 – both addresses are fixed
5..0	R/W	Unused	Always 1

9.1.8. REU Expansion Address $DF0E R

Reset to $00
This is an 8 bit mirror of $DF06

REU Expansion Address $DF0E		
Bit #	Read/Write	Function
7..0	R	Reads back the 8 bit value written to $DF06. This is the upper 8 bits of the 24 bit SDRAM address.

9.1.9. REU Superbank $DF0F R/W

Reset to $00
This register holds bit 25 of the SDRAM address

REU Superbank $DF0F		
Bit #	Read/Write	Function
7..1	R/W	Unused bits, but have registers to hold values. These bits could be used by programmers for their own purposes.
0	R/W	Bit 25 of the SDRAM address

9.2. ROM/RAM Mapping Registers

9.2.1. ROML Base Address $DF11/DF10 R/W

Reset to $0000
These registers hold the 16 bit offset into the SDRAM for a ROML image. Note that the ROM images are on 8192 byte boundaries in the SDRAM so only the lower 12 bits of the 16 bit value is actually used.

Notes:
1. The ROMH image must be loaded into SDRAM before it can be used.
2. Using the ROMH image requires setting the proper bit in the ROMSEL register ($DF28).

REU ROML Base Low $DF10		
Bit #	Read/Write	Function
7..0	R/W	Low 8 bits of the ROML offset address in SDRAM

REU ROML Base High $DF11		
Bit #	Read/Write	Function
7..0	R/W	High 8 bits of the ROML offset address in SDRAM

9.2.2. ROMH Base Address $DF13/DF12 R/W

Reset to $000
These registers hold the 16 bit offset into the SDRAM for a ROMH image. Note that the ROM images are on 8192 byte boundaries in the SDRAM so only the lower 12 bits of the 16 bit value is actually used.

Notes:
1. The ROMH image must be loaded into SDRAM before it can be used.
2. Using the ROMH image requires setting the proper bit in the ROMSEL register ($DF28).

REU ROMH Base Low $DF12		
Bit #	Read/Write	Function
7..0	R/W	Low 8 bits of the ROMH offset address in SDRAM

REU ROMH Base High $DF13		
Bit #	Read/Write	Function
7..0	R/W	High 8 bits of the ROMH offset address in SDRAM

9.2.3. RAM Base Address $DF16/DF15/DF14 R/W

Reset to $000000
These registers hold the 17 bit to map a 256 byte window at $DE00 into the SDRAM memory space.

REU RAM Base Low $DF14		
Bit #	Read/Write	Function
7..0	R/W	Low 8 bits of the ROMH offset address in SDRAM

REU RAM Base Mid $DF15		
Bit #	Read/Write	Function
7..0	R/W	High 8 bits of the ROMH offset address in SDRAM

REU RAM Base High $DF16		
Bit #	Read/Write	Function
7..1		Unused
0	R/W	Bit 17 of the window address

9.2.4. ROM Select Register $DF28 R/W

Reset: $00
This register controls the EXROM and GAME inputs to the C64

ROM Select Register $DF28		
Bit #	Read/Write	Function
7	R/W	'1' – disable the boot ROM '0' – enable the boot ROM
6	R/W	unused
5	R/W	unused

ROM Select Register $DF28		
Bit #	Read/Write	Function
4	R/W	unused
3	R/W	unused
2	R/W	unused
1	R/W	'1' – set EXROM line low '0' – set the EXROM line high
0	R/W	'1` - set GAME line low '0' – set the GAME line high

9.3. Video Peripheral Registers

9.3.1. VID_STAT_REG $DF20 R

Returns the status of the video peripheral. The most useful bit is bit 7 as this indicates the video peripheral is available for another operation. This is useful since scrolling or other operations may take a considerable amount of time in CPU terms.

Video Status Register $DF20		
Bit #	Read/Write	Function
7	R	'1' – the video peripheral is available for another operation.
6	R	unused
5	R	unused
4	R	unused
3	R	unused
2	R	unused
1	R	'1' – the video peripheral has issued an interrupt This bit is not connected in the FPGA
0	R	unused

9.3.2. VID_CHAR_REG $DF21 R/W

Write – the character to be written to the video peripheral

Read
Returns the current attribute data

Video Character Register $DF21		
Bit #	Read/Write	Function
7	R	'0'
6	R	'0'
5	R	'0'
4	R	'0'
3	R	'1' – flash attribute is active
2	R	'1' – strikethrough attribute is active
1	R	'1' – underline attribute is active

Video Character Register $DF21		
Bit #	Read/Write	Function
0	R	‘1` - inverse attribute is active

9.4. Coprocessor Control Registers

These registers allow the C64 to control and communicate with the coprocessor

9.4.1. Mailbox Bank Register $DF26 R/W

Reset: $00
This register controls which of the 32 banks of 16 mailbox registers are available to the C64.

Mailbox Bank Register $DF26		
Bit #	Read/Write	Function
7..5	R/W	Unused
4..0	R/W	Bank number

9.4.2. Mailbox Bank Register $DF30..$DF3F R/W

Reset: random
These addresses access a 16 byte bank in the coprocessor mailbox space

9.4.3. Coprocessor Control Register $DF2F R/W

Reset: $00
This register controls the operation or the coprocessor.

Mailbox Bank Register $DF2F		
Bit #	Read/Write	Function
7	R/W	'1' – allows the coprocessor access to the SDRAM '0' – allows the C64 access to the SDRAM
6	R/W	'1' – allows the coprocessor access to the video peripheral '0' – allows the C64 access to the video peripheral
5	R/W	'0' – resets the coprocessor and all of its peripherals
4	R/W	'0' – disable Z80 via its chip enable '1' - enable Z80 via its chip enable
3	R/W	'0' – reset only the coprocessor
2	R	Reports the state of the internal Z80 CEN signal. The state of the internal CEN may be delayed since it must by synchronized with the Z80 operation,
1	R/W	unused
0	R/W	'1' – disables the REU

9.5. I2C Control Registers

9.5.1. I2C Command Register $DF2A R/W

Reset: $00
This register is used to control the I2C FSM and read back status bits from the FSM

I2C Command Register **$DF2A** **Write**

Bit #	Read/Write	Function
7	W	unused
6	W	unused
5	W	unused
4	W	unused
3	W	unused
2..0	W	

Command Code	Name	Function
$00	NOP	The I2C FSM does nothing
$01	START	Sends an I2C START command. The I2C FSM resets the command code to NOP after the START command completes
$02	STOP	Sends an I2C STOP command. The I2C FSM resets the command code to NOP after the STOP command completes
$03	SEND	Sets the I2C FSM to send bytes on the I2C bus. The SEND command also performs the ACK operation and the ACK bit is latched into the status byte. Note: no byte is actually clocked out until it is written to the I2C_DATWR_REG. Note: the I2C FSM will stay in this state until another command code is written to this register.
$04	RDACK	Sets the I2C FSM to receive bytes on the I2C bus. The RDACK command also sends an ACK to the I2C peripheral. Note: no byte is actually clocked in until a byte is read from the I2C_DATRD_REG. Note: the I2C FSM will stay in this state until another command code is written to this register.
$05	RDNACK	Sets the I2C FSM to receive bytes on the I2C bus. The RDACK command also sends an NACK to the I2C peripheral. Note: no byte is actually clocked in until a byte is read from the I2C_DATRD_REG. Note: the I2C FSM will stay in this state until another command code is written to this register.
$06 & $07	unused	Either of these values will do nothing or will cause the I2C FSM to return to the idle state.

I2C Command Register $DF2A Read		
Bit #	Read/Write	Function
7	R	'1' – if the I2C shifter is ready to send/receive a byte '0' – if the I2C is busy shifting out a byte
6	R	'1' – the I2C peripheral returned a NACK during a write to it '0' – the I2C peripheral returned an ACK during a write to it
5	R	'1' – the I2C FSM is in the idle state
4	R	The logical state of the I2C SDA pin on the FPGA. This allows the C64 to interrogate the actual state of the I2C bus.
3	R	The logical state of the I2C SCL pin on the FPGA. This allows the C64 to interrogate the actual state of the I2C bus.
2..0	R	Returns the command value

9.5.2. I2C_DATARD_REG $DF2B R/W

Reset: $00

Reading from this register will provide the byte last read from the I2C bus if the command register (#DF2A) contains a RDACK or RDNACK command.

Note: if there has been no previous read from the I2C device, a dummy read must be performed by the C64 to actually shift in the first byte from the I2C device. All successive reads will then be valid data.

I2C Data Read Register $DF2B		
Bit #	Read/Write	Function
7..0	R/W	The last data read from the I2C device and will shift in the next byte from the I2C device.

9.5.3. I2C Data Write Register $DF2C R/W

Reset: $00

Writing to this register will send a byte on the I2C bus if the command register (#DF2A) contains a SEND command

I2C Data Read Register $DF2C		
Bit #	Read/Write	Function
7..0	R/W	The data value to send on the I2C bus

9.6. Miscellaneous Registers

9.6.1. FPGA Revision Register $DF2E R

Reset: FPGA version

Reading this register provides the version of the FPGA in BCD format

FPGA Revision Number		$DF2E
Bit #	Read/Write	Function
7..4	R	FPGA major revision
3..0	R	FPGA minor revision