REU PLUS2C REFERENCE GUIDE

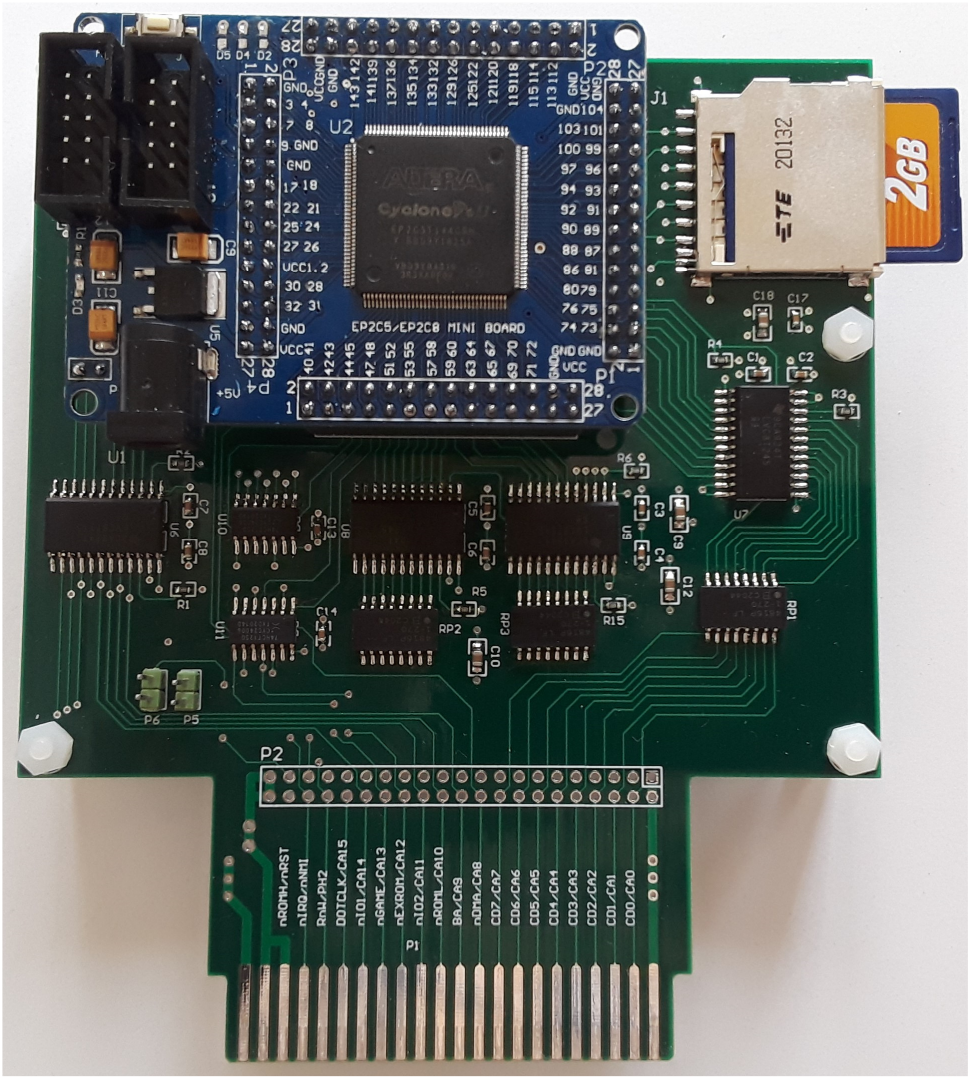


Table of Contents

1. Acknowledgements

The design of the REU Plus2C was based on previous work by many people and was made successful by them. Any remaining bugs or errors in the FPGA are the fault of the author alone and not of the authors of the referenced works.

Wolfgang Moser – ***Commodore RAM Expansion Unit Controller 8726R1***

This document provides a detailed description of the inner workings of the original Commodore REU controller IC. This formed the basis of the VHDL code for the REU portion of the FPGA.

Thomas Giesel – ***The C64 PLA Dissected***

This document provides a detailed description of the intricacies of the C64 PLA. This document was invaluable for understanding the timing and addressing of the C64 bus.

Commodore Business Machines – ***The Commodore 64 Programmer's Reference Guide***

Need I say more?

Jim Brain – ***RETRO Innovations***

Jim is impetus of this project. When I contacted him about making a C64 peripheral, he suggested an inexpensive REU for the Commodore community. While this implementation is not horrendously expensive, it hews to the original concept.

Grant Searle and Rienk Koolstra

The SD card controller was lifted wholesale from the MiSTER implementation of Grant's Multicomp system. I have made some very minor changes, but the vast majority of the work is theirs.

Scott Hunter and Paul Goodrich

They were willing to risk their precious C64s to test the prototype hardware. Thanks for your help.

2. Introduction

3. REU

The REU Plus2C is a device that plugs into the C64 expansion connector and provides enhanced REU 1764 functions for the C64. The REU Plus2C is a relatively low-cost expansion module that is a feature subset of the more expensive REU Plus4C device. The majority of the logic for the REU Plus2C is implemented using an inexpensive Intel EP2C5 FPGA development board.

The features of the REU Plus2C board are:

- 32MB of SDRAM expansion memory for the C64
- Full but expanded REU 1764 functionality

    - Full REU functionality but with 32MB of expansion memory. The REU register set has been expanded to accommodate the additional memory
    - Full software control of the GAME and EXROM lines
    - Ability to use the REU memory for software loaded ROMs
    - Supports basic Ultimax mode using the expansion memory
- A 256 byte RAM window at $DExx for accessing the SDRAM memory
- SD Card interface using a smart embedded SD card peripheral
- Real time clock with battery backup
- 256 byte EEPROM for storing configuration information
- 8/16/32 bit integer ALU
- 1024 byte stack located in the FPGA hardware for temporary data storage.

The main board contains the connector for the C64 expansion board as well as the buffers required to translate between the 3.3V logic of the FPGA and the +5V logic of the C64. The main board requires less than 100mA of power so is well within the capabilities of the C64 bus.

4. REU Functions

The REU Plus2C utilizes a 16MBx16 SDRAM clocked at 100MHz for the REU RAM. The SDRAM controller allows the full 32MB of ram to be accessed by the FPGA logic. The SDRAM access time is fast enough that it can be successfully accessed during both phases of the 1MHz system clock which allows both the VIC and the 6510 processor to read and write data to the SDRAM without contention.

Depending upon what the programmer wants to do, accessing the memory can be a simple affair or can be relatively complex. The following sections will explain how the REU Plus2C must be programmed to use the various features in the FPGA.

It is important to note that the Ultimax and REU functions are mutually exclusive so that when Ultimax mode is enabled the REU functions are disabled and vice versa.

4.1. RAM Access

Note: The 256 byte window at $DExx is accessible under both Ultimax and REU modes of the FPGA.

RAM window accesses are allowed in any 256 byte page located anywhere in the 32MB memory space but the hardware enforces each page to be aligned on 256 byte boundaries in the SDRAM (SDRAM addresses end in $00). The 6510 provides the lower 8 bits of the SDRAM address while the FPGA contains three address window registers at $DF1A, $DF19, and $DF18 that provide the upper 17 bits required for the SDRAM address. The register at $DF1A contains the high byte of the SDRAM address and $DF18 contains the low byte.

To use this mode, program the address window registers with the desired memory offset into the SDRAM. All accesses to the addresses in the $DFxx range will then read or write to the SDRAM in the selected window.

4.2. REU Functions

Note: REU functions are NOT available in Ultimax mode.

This document will not cover the REU operations in detail but only the differences between the REU Plus2C and the standard 1764 REU.

A copy of the 1764 REU manual can be obtained at:
http://www.zimmers.net/anonftp/pub/cbm/manuals/peripherals/1764_Ram_Expansion_Module_Users_Guide.pdf

The REU Plus2C implements the same registers as in the 1764 REU but adds two registers at addresses $DF0E and $DF0F. The function of these registers will be described below.

Because the REU Plus2C has substantially more RAM than the 1764 REU, additional address bits are required. The original 1764 only utilized the lower two bits of register $DF06 for the four banks it contained. The REU Plus2C utilizes all 8 bits of this register as well as bit 0 of the new superbank register at $DF0F to contain the additional address bits. Like the 1764, the REU Plus2C handles the 25 bit SDRAM address as a flat address space so that all 25 bits are adjusted as required when accessing the SDRAM. This means that there are no “banks” for the programmer to keep track of.

One wrinkle about the register at $DF06, is that when read back, the upper six bits will be set to zero to maintain compatibility with the original 1764 REU. If all 8 bits are required, read from $DF0E instead of $DF06. Remember, all 25 bits of the SDRAM address are maintained internally so register reloads and readbacks will operate as expected.

As an example, if the programmer wishes to use SDRAM memory at address $1BEEFA5, they would load register $DF0F with $01, $DF06 with $BE, $DF05 with $EF, $DF04 with $A5. The REU engine will then perform the REU operation as expected.

4.3. EXROM and GAME

Note: ROM functionality is available in Ultimax and REU modes.

4.3.1. REU Mode

The REU Plus2C does not have on board ROM, but the SDRAM memory can be loaded from disk or the SD card and the memory then used as ROM. The C64 asserts two signals (ROMH and ROML) to access cartridge 8K ROM banks. The REU Plus2C divides the 8K ROM banks into two 4k banks. This is done to allow software the ability to increase the effective available memory space for ROMH and ROML.

For the lower 4k bank of ROML the two registers at $DF11:$DF10 provide the upper address bits for the SDRAM address. The upper 4k bank of ROML uses registers $DF13:$DF12 for this purpose.

Similarly, for the lower 4k bank of ROMH $DF15:$DF14 are used and $DF17:$DF16 are used for the upper 4k bank.

4.3.1.1. Memory Example

As an example of how this may be used, we will assume that the programmer wishes to add an extension to BASIC similar to Simon's BASIC.

Assumptions:

1. The expansion takes a bit over 20k of space and that it has been partitioned to be banking friendly.
2. The new expansion code is overlaid in the original BASIC ROM area (LHGX = 0100)
3. The BASIC wedge and/or re-revectoring goes to some other memory area in C64 main memory (NOT in the extension memory).

Your loader does the following:

The wedge and all revectoring code is read from disk and transferred to where it needs to go in C64 main memory.

The common code for the BASIC extension is loaded into SDRAM at $17A00 and the other 4k block of code are loaded into the SDRAM at the addresses in the illustration. This is an extreme example since the code would likely be loaded into consecutive 4k areas, but it illustrates the point that the code can be at any SDRAM address as long as the extension knows where its code is and your code does not stomp on other code in SDRAM.

The loader then writes $17A into $DF15:$DF14. This points the ROMH vector to your common code and queues it up to be used. Registers at $DF17:$DF16 can also be loaded at this time as well.

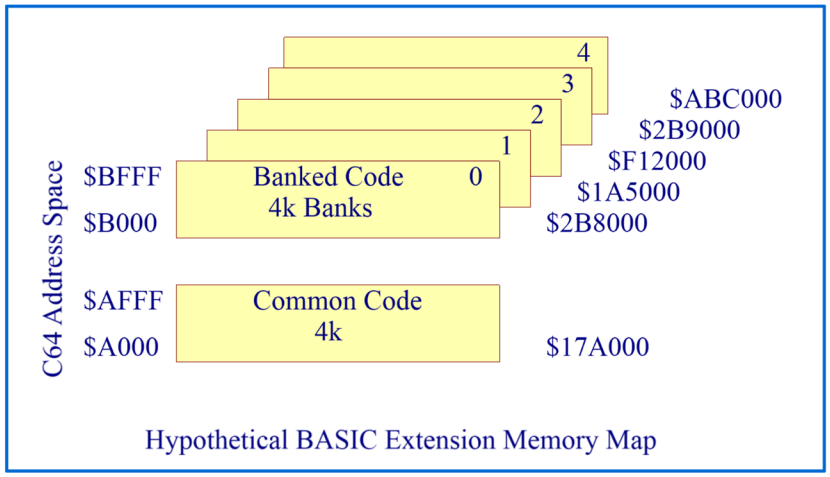
At this point the loader is done, so it exits to BASIC.

When the wedge determines it needs to activate the extension, it reads and saves the current state of the LGHX bits and writes the LGHX bits with 0100 and jumps to the extension dispatcher in the extension common area. The basic extension maps in whichever of its banks (one or more) are required to deal with the request and when done, it jumps back to the wedge code where the LGHX bits are restored and the program goes on.

If the extension needs scratchpad memory, it can use either the FPGA stack or revector the page at $DExx for its use. Obviously, the extension would need to clean up after itself to ensure any other users of these resources are not affected.

This is only one example but was chosen because it allows the original BASIC to be used as well as minimizing any reduction in RAM for BASIC programs. If the programmer is willing to completely control the C64 simpler and faster banking schemes can be implemented that would, for example, allow a modified version of the C128 BASIC 7 to be run on the C64.

It should be noted, that while using this scheme, programs are limited to writing only to the C64 motherboard RAM and the page at $DExx. Ultimax mode can eliminate this barrier with some limitations to VIC II RAM access.



Hypothetical BASIC Extension Memory Map

4.3.2. Ultimax Mode

See the Ultimax section

4.4. Ultimax

Ultimax mode configures the C64 memory map so that the lower 4k of C64 RAM, and the I/O space at Dxxx is permanently mapped in. ROML is positioned at $8000..$9FFF and ROMH is at $E000..$FFFF. All other addresses are usually described as “open” meaning that the bus is essentially floating and any data on the bus is indeterminate. The REU Plus2C provides the data for both the 6510 and VIC II for all memory areas except the lower 4k of memory and provides data in the I/O area whenever it falls in the range to which the FPGA responds.

The REU Plus2C provides the 16 bit registers required to expand the C64 address bus as to access the full SDRAM. The FPGA uses block RAM to store these registers which provides for storing several mapping register sets. Each register set consists of 16 registers each of 16 bits. The size of the block RAMs allows up to 32 of these mapping register sets to be maintained. See the figure below for a pictorial representation of how the register sets are implemented.

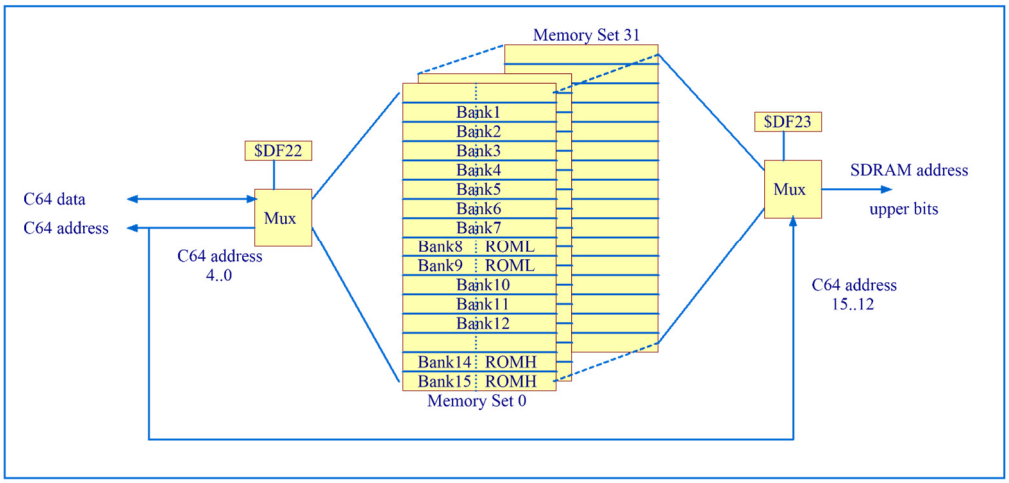
The FPGA reserves 32 addresses in the range of $DF40..DF5F to allow the 6510 to write or read a bank of sixteen mapping registers at a time. A register at address $DF22 selects which set of 16 registers is being accessed by the 6510 at any one time. To read or write a value to a given register set, the 6510 first writes a value between 0 and 31 to $DF22 which selects one of the mapping register sets to access. It then writes the two bytes necessary for that memory bank and repeats until it is complete.

Once the memory mapping sets have been initialized, they can be used by writing the appropriate value to the register at address $DF23. The upper four bits of the 6510 address bus are then used to select one of the 16 mapping registers from the selected mapping set.

One important caveat to this is that addresses in the range of $0000..0FFF are ignored by the FPGA since they correspond to the addresses in the C64 memory area used during Ultimax mode. The Ultimax mapping register for the I/O area is ignored, however the FPGA responds appropriately to accesses in the I/O area ($D000..$DFFF) including allowing access via the memory page at $DExx.

The use of Ultimax mode can provide some real advantages to the programmer since very large programs can be written as long as they are written with memory banking in mind. It is also possible to implement a versatile multi-tasking environment that switches to different tasks by changing the map select register at $DF23.

The tradeoff in Ultimax mode is that it is more difficult to interact with the memory the VIC II can access. Depending on the program, it may require dropping into and out of Ultimax mode.



5. I2C Peripheral

The I2C peripheral is used to communicate with the real time clock (RTC) and the 256 byte EEPROM. 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.

5.1. Example code

5.1.1. 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

5.1.2. 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.

5.1.3. 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.

5.2. I2C C64 Registers

5.2.1.1. 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.
Note: the I2C FSM will stay in this state until another command code is written to this register.</td></tr>
</table>

	$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	

5.2.1.2. 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	

5.2.1.3. 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	

5.2.1.4. VID_STAT_REG DF20 (57120) Read

6. 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.

7. I2C EEPROM

The REUPlus incorporates a pair of AT24C02C-STUM-T 256 byte EEPROMs for storage of configuration information. The devices are accessed by the I2C bus.

See the AT24C02C-STUM-T data sheet for details on accessing the EEPROMS.

8. 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.

9. 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.

10. 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	ROMLBL_BASELOW
57105	DF11	ROMLBL_BASEHIGH
57106	DF12	ROMLBH_BASELOW
57107	DF13	ROMLBH_BASEHIGH
57108	DF14	ROMHBL_BASELOW
57109	DF15	ROMHBL_BASEHIGH
57110	DF16	ROMHBH_BASELOW
57111	DF17	ROMHBH_BASEHIGH
57112	DF18	RAM_BASELOW
57113	DF19	RAM_BASEMID
57114	DF1A	RAM_BASEHI
57115	DF1B	Unused
57116	DF1C	Unused
57117	DF1D	Unused

Address	Address (Hex)	Name
57118	DF1E	Unused
57119	DF1F	GAMEROM
57120	DF20	FPGAREV
57121	DF21	SDCRST
57122	DF22	UMAXWRBANK
57123	DF23	UMAXRDBANK
57124	DF24	STACK_RD
57125	DF25	STACK_WR
57126	DF26	Unused
57127	DF27	Unused
57128	DF28	ROMSEL_REG
57129	DF29	Unused
57130	DF2A	I2C_CMD_REG
57131	DF2B	I2C_DATRD_REG
57132	DF2C	I2C_DATWR_REG
57133	DF2D	TIMER10MS
57134	DF2E	FPGA_REV_REG
57135	DF2F	Unused
57136	DF30	Unused
57137	DF31	Unused
57138	DF32	Unused
57139	DF33	Unused
57140	DF34	Unused
57141	DF35	Unused
57142	DF36	Unused
57143	DF37	Unused
57144	DF38	Unused
57145	DF39	Unused
57146	DF3A	Unused
57147	DF3B	Unused
57148	DF3C	Unused
57149	DF3D	Unused
57150	DF3E	Unused
57151	DF3F	Unused
57152	DF40	Umax0Low
57153	DF41	Umax0High
57154	DF42	Umax1Low
57155	DF43	Umax1High
57156	DF44	Umax2Low
57157	DF45	Umax2High

Address	Address (Hex)	Name
57158	DF46	Umax3Low
57159	DF47	Umax3High
57160	DF48	Umax4Low
57161	DF49	Umax4High
57162	DF4A	Umax5Low
57163	DF4B	Umax5High
57164	DF4C	Umax6Low
57165	DF4D	Umax6 High
57166	DF4E	Umax7Low
57167	DF4F	Umax7High
57168	DF50	Umax8Low
57169	DF51	Umax8High
57170	DF52	Umax9Low
57171	DF53	Umax9High
57172	DF54	UmaxALow
57173	DF55	UmaxAHigh
57174	DF56	UmaxBLow
57175	DF57	UmaxBHigh
57176	DF58	UmaxCLow
57177	DF59	UmaxCHigh
57178	DF5A	UmaxDLow
57179	DF5B	UmaxDHigh
57180	DF5C	UmaxELow
57181	DF5D	UmaxDHigh
57182	DF5E	UmaxFLow
57183	DF5F	UmaxFHigh
57200	DF70	SDCDATWR
57201	DF71	SDCSTATUS
57202	DF72	SDCADRLO
57203	DF73	SDCADRMID
57204	DF74	SDCADRHI
57205	DF75	SDCDATRD

10.1. REU Registers

10.1.1. 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'

10.1.2. 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.

REU Command $DF01			
Bit #	Read/Write	Function	Mode
			The bit is automatically set back to 1, when a transfer took place.
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

10.1.3. 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

10.1.4. 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

10.1.5. 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

10.1.6. 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

10.1.7. 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

10.1.8. 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.

10.1.9. 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

10.2. ROM/RAM Mapping Registers

10.2.1. ROML Low Bank Base Address $DF11/DF10 R/W

Reset to $0000
These registers hold the 16 bit offset into the SDRAM for the low 4k of a ROML image. Note that the ROM image pages are on 4096 byte boundaries in the SDRAM so only the lower 13 bits of the 16 bit value is actually used.

Notes:
1. The ROML image must be loaded into SDRAM before it can be used.
2. Using the ROML 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

10.2.2. ROML HIgh Bank Base Address $DF13/DF12 R/W

Reset to $0000
These registers hold the 16 bit offset into the SDRAM for the high 4k of a ROML image. Note that the ROM image pages are on 4096 byte boundaries in the SDRAM so only the lower 13 bits of the 16 bit value is actually used.

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

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

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

10.2.3. ROMH Low Bank Base Address $DF13/DF12 R/W

Reset to $000
These registers hold the 16 bit offset into the SDRAM for the low 4k of a ROMH image. Note that the ROM image pages are on 4096 byte boundaries in the SDRAM so only the lower 13 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

10.2.4. ROMH HIgh Bank Base Address $DF15/DF14 R/W

Reset to $000
These registers hold the 16 bit offset into the SDRAM for the high 4k of a ROMH image. Note that the ROM image pages are on 4096 byte boundaries in the SDRAM so only the lower 13 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

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

Reset to $000000
These registers hold the 17 bits 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

10.2.6. ROM Select Register $DF1F R/W

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

ROM Select Register $DF1F		
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
4	R/W	unused
3	R/W	unused
2	R/W	unused
1	R/W	'1' – set EXROM line low (asserts EXROM) '0' – set the EXROM line high
0	R/W	'1` - set GAME line low (asserts GAME) '0' – set the GAME line high

10.3. Ultimax Mapping Registers

10.3.1. Ultimax I/O Bank Select $DF22 R/W

Reset to $00

This register selects which of the 32 Ultimax bank sets are selected for reading/writing by the C64. These reads and writes are how the C64 initializes the bank sets for use.

Ultimax I/O Bank Select $DF22		
Bit #	Read/Write	Function
7..5	R/W	Ignored
4..0		The bank to access

10.3.1. Ultimax Address Bank Select $DF23 R/W

Reset to $00

This register selects which of the 32 Ultimax bank sets are selected for memorya access by the C64..

Ultimax Address Bank Select $DF23		
Bit #	Read/Write	Function
7..0	R/W	Low 8 bits of the ROMH offset address in SDRAM

10.3.1. Ultimax Address Bank Registers $DF40…$DF5F R/W

Reset to --
These registers hold the 13 bits to map a 4k byte window into the C64 address space.

C64 Addr	BankH	BankL	Read/Write	Function
$0000-$0FFF	$DF41	$DF40	R/W	Not used for addressing but available for general use
$1000-$1FFF	$DF43	$DF42	R/W	Bank pointer into SDRAM for this 4k area
$2000-$2FFF	$DF45	$DF44	R/W	Bank pointer into SDRAM for this 4k area
$3000-$3FFF	$DF47	$DF46	R/W	Bank pointer into SDRAM for this 4k area
$4000-$4FFF	$DF49	$DF48	R/W	Bank pointer into SDRAM for this 4k area
$5000-$5FFF	$DF4B	$DF4A	R/W	Bank pointer into SDRAM for this 4k area
$6000-$6FFF	$DF4D	$DF4C	R/W	Bank pointer into SDRAM for this 4k area
$7000-$7FFF	$DF4F	$DF4E	R/W	Bank pointer into SDRAM for this 4k area
$8000-$8FFF	$DF51	$DF50	R/W	Bank pointer into SDRAM for this 4k area
$9000-$9FFF	$DF53	$DF52	R/W	Bank pointer into SDRAM for this 4k area
$A000-$AFFF	$DF55	$DF54	R/W	Bank pointer into SDRAM for this 4k area
$B000-$BFFF	$DF57	$DF56	R/W	Bank pointer into SDRAM for this 4k area
$C000-$CFFF	$DF59	$DF58	R/W	Bank pointer into SDRAM for this 4k area
$D000-$DFFF	$DF5B	$DF5A	R/W	Not used for addressing but available for general use
$E000-$EFFF	$DF5D	$DF5C	R/W	Bank pointer into SDRAM for this 4k area
$F000-$FFFF	$DF5F	$DF5E	R/W	Bank pointer into SDRAM for this 4k area

10.4. I2C Control Registers

10.4.1. 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

10.4.2. 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.

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

10.5. SD Card Registers

10.5.1. SD Card Reset Register $DF21 R/W

Reset: $00

Writing $01 to this register resets the SD controller and writing $00 releases the reset. This is useful in the event that the programmer wishes to re-intialize the card such as if the card has been removed and reinserted. This will cause the SD controller to re-initialze the card.

Reading the register returns its current state

SD Card Controller Reset $DF21		
Bit #	Read/Write	Function
7..1	W R	Ignored Always zero
3..0	W R	'1' resets the SD controller, '0' releases the reset The current state of the reset bit.

10.5.1. SD Card Data Write $DF70 R/W

Reset: $00

Writing to this register will send a byte to the SD card via the SD controller IF the SD controller is in the block write state AND if the transmitter is available (the status register should be interrogated for this information). If the SD controller is not in the block write state, the write will have no effect other than to load the register.

Reading the register returns its current state

SD Card Data Write $DF70		
Bit #	Read/Write	Function
7..0	W R	Data to send Current value in the register

10.5.1. SD Card Control/Status $DF71 R/W

Reset: $00

Writing a valid command to this register while the SD controller is idle will advance the controller FSM to the desired state. If the SD controller is not in its idle state, a write here will have no effect.

NOTE: The action of writing the value is required to initiate the desired action. For instance, if the register value is $00 and the SD controller is in the idle state, the FSM will not advance to the block read state. If, however, the C64 writes a $00 to the register, the FSM will advance to the block read state.

Reading the register returns status information

SD Controller Control $DF21		
Bit #	Read/Write	Function
7..0	W	$00 – go to the block read state if in the idle state $01 – go to the block write state if in the idle state

SD Controller Status $DF21		
Bit #	Read/Write	Function
7	R	'1' – the output shift register is idle and can accept a data byte
6	R	'1' – the receive shift register has a byte available
5	R	'1' – the FSM is in either the block receive or block transmit mode The programmer is responsible for knowing which mode was requested
4	R	'1' – the SD controller is initializing an SD card.
3	R	'1' – the SD controller is in its idle state
2	R	'1' – the SD card has been removed and replaced since the last reset If this bit is set, the programmer should reset the SD controller because the card may have been changed and writing to it will likely corrupt it
1	R	'1' – the card is write protected NOTE: this bit is permanently set to '0' by the REU Plus2C hardware
0	R	'1' – an SD card is in the connector

10.5.1. SD Card Sector Address $DF74/$DF73/$DF72 R/W

Reset: $00

Writing to this register will send a byte to the SD card via the SD controller IF the SD controller is in the block write state. If the SD controller is not in the block write state, the write will have no effect other than to load the register.

Reading the register returns its current state

SD Card Sector High Byte $DF74		
Bit #	Read/Write	Function
7..0	W	The high byte of the SD card sector to read or write

SD Card Sector Middle Byte $DF73		
Bit #	Read/Write	Function
7..0	W	The middle byte of the SD card sector to read or write

SD Card Sector Low Byte $DF72		
Bit #	Read/Write	Function
7..0	W	The low byte of the SD card sector to read or write

10.5.1. SD Card Data Read $DF75 R/W

Reset: $00

Reading from this register will retrieve the byte read from the SD card via the SD controller IF the SD controller is in the block read state AND if the receiver indicates a byte is available (the status register should be interrogated for this information). If the SD controller is not in the block read state, the read will have no effect other than to load the register.

Reading the register does nothing

SD Card Data Read $DF75		
Bit #	Read/Write	Function
7..0	R	The received byte

10.6. Miscellaneous Registers

10.6.1. FPGA Revision Register $DF20 R

Reset: FPGA version

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

FPGA Revision Number $DF20		
Bit #	Read/Write	Function
7..4	R	FPGA major revision
3..0	R	FPGA minor revision

10.6.1. Stack Read $DF24 R

Reset: --

Reading this register returns the value on the top of the 1024 byte hardware stack. After the read, the internal stack pointer is adjusted as needed.

Stack Read $DF24		
Bit #	Read/Write	Function
7..0	R	The value at the top of the hardware stack

10.6.1. Stack Write $DF25 R

Reset: --

Reading this register pushes the value onto the top of the 1024 byte hardware stack. After the write, the internal stack pointer is adjusted as needed.

Stack Write $DF25		
Bit #	Read/Write	Function
7..0	W	The value to push on the stack