

Version 1.0

HYDRA SD MAX STORAGE CARD

PROGRAMMING AND USER MANUAL

SECURE DIGITAL + 128K EEPROM

Andre' LaMothe

Nurve Networks LLC

Author

Andre' LaMothe

Editor/Technical Reviewer

The "Collective"

Printing

0001

ISBN

Pending

All rights reserved. No part of this user manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the user of the information contained herein. Although every precaution has been taken in the preparation of this user manual, the publisher and authors assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

Trademarks

All terms mentioned in this user manual that are known to be trademarks or service marks have been appropriately capitalized. Nurve Networks LLC cannot attest to the accuracy of this information. Use of a term in this user manual should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this user manual as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "*as is*" basis. The authors and the publisher shall have neither liability nor any responsibility to any person or entity with respect to any loss or damages arising from the information contained in this user manual.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

eBook License

This electronic user manual may be printed for personal use and (1) copy may be made for archival purposes, but may not be distributed by any means whatsoever, sold, resold, in any form, in whole, or in parts. Additionally, the contents of the CD this electronic user manual came on relating to the design, development, imagery, or any and all related subject matter pertaining to the HYDRA™ are copyrighted as well and may not be distributed in any way whatsoever in whole or in part. Individual programs are copyrighted by their respective owners and may require separate licensing.

Licensing, Terms & Conditions

NURVE NETWORKS LLC, . END-USER LICENSE AGREEMENT FOR HYDRA HARDWARE, SOFTWARE , EBOOKS, AND USER MANUALS

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PRODUCT. IT CONTAINS SOFTWARE, THE USE OF WHICH IS LICENSED BY NURVE NETWORKS LLC, INC., TO ITS CUSTOMERS FOR THEIR USE ONLY AS SET FORTH BELOW. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT USE THE SOFTWARE OR HARDWARE. USING ANY PART OF THE SOFTWARE OR HARDWARE INDICATES THAT YOU ACCEPT THESE TERMS.

GRANT OF LICENSE: NURVE NETWORKS LLC (the "Licensor") grants to you this personal, limited, non-exclusive, non-transferable, non-assignable license solely to use in a single copy of the Licensed Works on a single computer for use by a single concurrent user only, and solely provided that you adhere to all of the terms and conditions of this Agreement. The foregoing is an express limited use license and not an assignment, sale, or other transfer of the Licensed Works or any Intellectual Property Rights of Licensor.

ASSENT: By opening the files and or packaging containing this software and or hardware, you agree that this Agreement is a legally binding and valid contract, agree to abide by the intellectual property laws and all of the terms and conditions of this Agreement, and further agree to take all necessary steps to ensure that the terms and conditions of this Agreement are not violated by any person or entity under your control or in your service.

OWNERSHIP OF SOFTWARE AND HARDWARE: The Licensor and/or its affiliates or subsidiaries own certain rights that may exist from time to time in this or any other jurisdiction, whether foreign or domestic, under patent law, copyright law, publicity rights law, moral rights law, trade secret law, trademark law, unfair competition law or other similar protections, regardless of whether or not such rights or protections are registered or perfected (the "Intellectual Property Rights"), in the computer software and hardware, together with any related documentation (including design, systems and user) and other materials for use in connection with such computer software and hardware in this package (collectively, the "Licensed Works"). ALL INTELLECTUAL PROPERTY RIGHTS IN AND TO THE LICENSED WORKS ARE AND SHALL REMAIN IN LICENSOR.

RESTRICTIONS:

- (a) You are expressly prohibited from copying, modifying, merging, selling, leasing, redistributing, assigning, or transferring in any matter, Licensed Works or any portion thereof.
- (b) You may make a single copy of software materials within the package or otherwise related to Licensed Works only as required for backup purposes.
- (c) You are also expressly prohibited from reverse engineering, decompiling, translating, disassembling, deciphering, decrypting, or otherwise attempting to discover the source code of the Licensed Works as the Licensed Works contain proprietary material of Licensor. You may not otherwise modify, alter, adapt, port, or merge the Licensed Works.
- (d) You may not remove, alter, deface, overprint or otherwise obscure Licensor patent, trademark, service mark or copyright notices.
- (e) You agree that the Licensed Works will not be shipped, transferred or exported into any other country, or used in any manner prohibited by any government agency or any export laws, restrictions or regulations.
- (f) You may not publish or distribute in any form of electronic or printed communication the materials within or otherwise related to Licensed Works, including but not limited to the object code, documentation, help files, examples, and benchmarks.

TERM: This Agreement is effective until terminated. You may terminate this Agreement at any time by uninstalling the Licensed Works and destroying all copies of the Licensed Works both HARDWARE and SOFTWARE. Upon any termination, you agree to uninstall the Licensed Works and return or destroy all copies of the Licensed Works, any accompanying documentation, and all other associated materials.

WARRANTIES AND DISCLAIMER: EXCEPT AS EXPRESSLY PROVIDED OTHERWISE IN A WRITTEN AGREEMENT BETWEEN LICENSOR AND YOU, THE LICENSED WORKS ARE NOW PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR THE WARRANTY OF NON-INFRINGEMENT. WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO WARRANTY THAT (i) THE LICENSED WORKS WILL MEET YOUR REQUIREMENTS, (ii) THE USE OF THE LICENSED WORKS WILL BE UNINTERRUPTED, TIMELY, SECURE, OR ERROR-FREE, (iii) THE RESULTS THAT MAY BE OBTAINED FROM THE USE OF THE LICENSED WORKS WILL BE ACCURATE OR RELIABLE, (iv) THE QUALITY OF THE LICENSED WORKS WILL MEET YOUR EXPECTATIONS, (v) ANY ERRORS IN THE LICENSED WORKS WILL BE CORRECTED, AND/OR (vi) YOU MAY USE, PRACTICE, EXECUTE, OR ACCESS THE LICENSED WORKS WITHOUT VIOLATING THE INTELLECTUAL PROPERTY RIGHTS OF OTHERS. SOME STATES OR JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES OR LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY MAY LAST, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. IF CALIFORNIA LAW IS NOT HELD TO APPLY TO THIS AGREEMENT FOR ANY REASON, THEN IN JURISDICTIONS WHERE WARRANTIES, GUARANTEES, REPRESENTATIONS, AND/OR CONDITIONS OF ANY TYPE MAY NOT BE DISCLAIMED, ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION AND/OR WARRANTY IS: (1) HEREBY LIMITED TO THE PERIOD OF EITHER (A) Five (5) DAYS FROM THE DATE OF OPENING THE PACKAGE CONTAINING THE LICENSED WORKS OR (B) THE SHORTEST PERIOD ALLOWED BY LAW IN THE APPLICABLE JURISDICTION IF A FIVE (5) DAY LIMITATION WOULD BE UNENFORCEABLE; AND (2) LICENSOR'S SOLE LIABILITY FOR ANY BREACH OF ANY SUCH WARRANTY, GUARANTEE, REPRESENTATION, AND/OR CONDITION SHALL BE TO PROVIDE YOU WITH A NEW COPY OF THE LICENSED WORKS. IN NO EVENT SHALL LICENSOR OR ITS SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, THOSE RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT LICENSOR HAD BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE LICENSED WORKS. SOME JURISDICTIONS PROHIBIT THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING ANY FAILURE OF ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

SEVERABILITY: In the event any provision of this License Agreement is found to be invalid, illegal or unenforceable, the validity, legality and enforceability of any of the remaining provisions shall not in any way be affected or impaired and a valid, legal and enforceable provision of similar intent and economic impact shall be substituted therefore.

ENTIRE AGREEMENT: This License Agreement sets forth the entire understanding and agreement between you and NURVE NETWORKS LLC, supersedes all prior agreements, whether written or oral, with respect to the Software, and may be amended only in a writing signed by both parties.

NURVE NETWORKS LLC
12724 Rush Creek Lane
Austin, TX 78732

Version & Support/Web Site

This document is valid with the following hardware, software and firmware versions:

- HYDRA Game Console Revision A. or greater.
- Propeller Tool 1.0 or greater.

The information herein will usually apply to newer versions but may not apply to older versions. Please contact Nurve Networks LLC for any questions you may have.

Visit **www.xgamestation.com** for downloads, support and access to the XGameStation/HYDRA user community and more!

For technical support, sales, general questions, share feedback, please contact Nurve Networks LLC at:

support@nurve.net / nurve_help@yahoo.com

Table of Contents

LICENSING, TERMS & CONDITIONS	3
VERSION & SUPPORT/WEB SITE	4
TABLE OF CONTENTS	5
HYDRA SD MAX+128K CARD USER MANUAL	6
1.0 HYDRA SD MAX STORAGE CARD MANUAL OVERVIEW	6
2.0 PRODUCT CONTENTS.....	7
2.1 CD-ROM CONTENTS	7
3.0 INTRODUCTION AND QUICK START.....	8
3.1 QUICK START GUIDE	10
3.1.1 Re-programming the HYDRA SD Max Card with the Menu Demo Program ..	10
3.1.2 Re-formatting the SD Card with the Demo Binary Images.....	11
4.0 CIRCUIT DESIGN, ELECTRICAL AND MECHANICAL INTERFACE.....	12
4.1 ELECTRICAL INTERFACE DESIGN	12
4.2 MECHANICAL INTERFACE	15
5.0 INTERFACING TO THE HYDRA SD MAX CARD.....	16
5.1 SD CARD OVERIEW & HISTORY	17
5.2 SPI BUS BASICS	18
5.2.1 Basic SPI Communications Steps.....	20
5.3 SD CARD COMMUNICATIONS PROTOCOL	22
5.3.1 Placing the SD Card into SPI Mode	25
5.3.2 Reading a Sector.....	26
5.3.3 Writing a Sector	27
5.4 FAT16 FILE SYSTEM OVERVIEW	29
5.4.1 FAT16 SD Card Disk Structure	30
5.4.2 Master Boot Record	31
5.4.3 Partition Entries in the MBR	32
5.4.4 Parition Boot Record (PBR)	32
5.4.5 A Quick Recap and Locating the Primary FAT16 Data Structures	33
5.4.6 The Root Directory	34
5.4.7 The File Allocation Table	37
5.4.8 Understanding Directories.....	39
5.4.9 Final Aspects of Navigating the FAT16 File System.....	39
6.0 UNLEASHING THE SD MAX DRIVER API	40
6.1 Driver API Constants	42
6.2 Driver API Globals.....	44
6.3 Initializing the SD Max Driver	45
6.4 Master API Listing	46
7.0 THE HYDRA SD MAX DEMOS	61
7.1 THE MENU LOADER PROGRAM.....	62
7.1.1 The Menu Loader Software Architecture	62
7.1.2 Building the SD Card for the Loader	63
7.2 THE SD MAX DEMO API PROGRAM.....	65
7.2.1 The Main Menu.....	66
8.0 SUMMARY	66
APPENDICES	67
A. HYDRA SD MAX SCHEMATIC	67
B. PCB REFERENCE LAYOUT	69
C. API DRIVER SOURCE CODE LISTING.....	70
NOTES.....	89

HYDRA SD Max+128K Card User Manual

1.0 HYDRA SD Max Storage Card Manual Overview

Welcome to the user manual for the **HYDRA SD Max Storage Card**. The **HYDRA SD Max Storage Card** or “**SD Max**” for short, enhances the functionality of the HYDRA by adding the ability to read and write standard SD (Secure Digital) cards. The hardware interface is facilitated by a standard HYDRA expansion card with a SD card slot built in as well as a 128K EEPROM for program storage. The hardware interface exports lines for both **SPI** (serial peripheral interface) mode as well as proprietary **SD mode** (assuming you have a license).

Thus, you can use simple SPI routines to talk to the SD card or if you are a licensed SD card developer you can access the card at high speed. Also, you can always find “educational” notes on internet on how to access the high speed modes and then write code to do so, but if you ever produce a consumer product with the high speed protocol you better have a license, or you might get a visit from the “SD Police”. However, for the demos and drivers within we will stick with the free **SPI modes** of SD card operation.

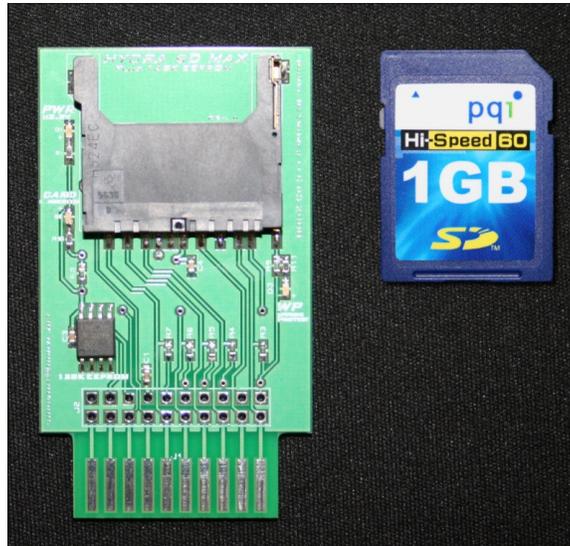
The HYDRA SD Max Storage Card not only has the SD card interface on it, but comes with a standard 128K serial EEPROM chip on-board, so that you can load your applications onto the 128K EEPROM and boot from the card as well. Thus, the card doubles as a 128K EEPROM memory expansion card when you’re not interested in using the SD slot.

NOTE

The high speed modes of the SD card interface is **not** free to use. There are two modes of operation; **SPI mode** and **SD mode**. In SPI mode there are no royalties, but in SD modes you must pay a fee to license to use (there are two version **1-bit** and faster **4-bit** protocol). Read more about it here: http://en.wikipedia.org/wiki/Sd_card. However, 99% of hobby or educational products simply use the SPI mode which is more than adequate.

There is a lot you need to know to write software yourself to access SD cards. From the bottom up, first, there is the electrical interface to the SD card itself which is a number of signals and power, then the actual communications to the device is based on the **SPI** (serial peripheral interface) which is a **3-line serial protocol** (data in, data out, clock) in SPI mode of course. That’s how you talk to the SD card. Then on top of that you need to send the SD card commands in **SD protocol format**. This gives you access to the card’s features, but not the file system. You can read and write sectors, but they are in raw form. If you want to really take advantage of the SD card then you have to emulate the **DOS FAT16** (or FAT32) file system standard. Alas, you have to write a version of the FAT16 file system on the HYDRA to access the card!

This is a lot of work no doubt, but hopefully this manual will give you insight into how to do this yourself. And of course, I will provide pre-made drivers and an API written in **SPIN** for ease of understanding along with a little menu driven demo. Nevertheless, I highly recommend that you try and conquer everything from the electrical interface, the SPI protocol, the SD card protocol and FAT16, so you know every single bit (no pun intended) of operation of these amazing little storage devices. The ability to hold 4Gigs in something the size of a stamp is amazing when you think about it. Assuming a page of text is about 4000 characters and the average book 350 pages, that means that a 4Gig SD card can hold $4,000,000,000 / 350 * 4000 = 2857$ books!!!! So you could easily store everything we know about math, physics, computers, electronics, chemistry, biology easily along with arts and a good helping of fiction if it was the end of the world and you had to leave the planet and could only take what you could fit in your pockets!

Figure 1.0 – The HYDRA SD Max Storage Card and Accessories.

2.0 Product Contents

The HYDRA SD Max Storage Card kit consists of the following items as shown in Figure 1.0:

- The HYDRA SD Max Storage Card itself with SD card slot and 128K EEPROM onboard.
- A 1GB SD card pre-formatted with a FAT16 file system (SD card may vary from model shown).
- CD-ROM with source, tools, API, etc. (not shown).
- Printed Quick Start Sheet (not shown).

NOTE

The 1GB SD card that comes with the kit is pre-format FAT16 and has a file system on it with a number of binaries for the pre-programmed demo that comes on the SD cards. You can erase and re-format the SD card at any time.

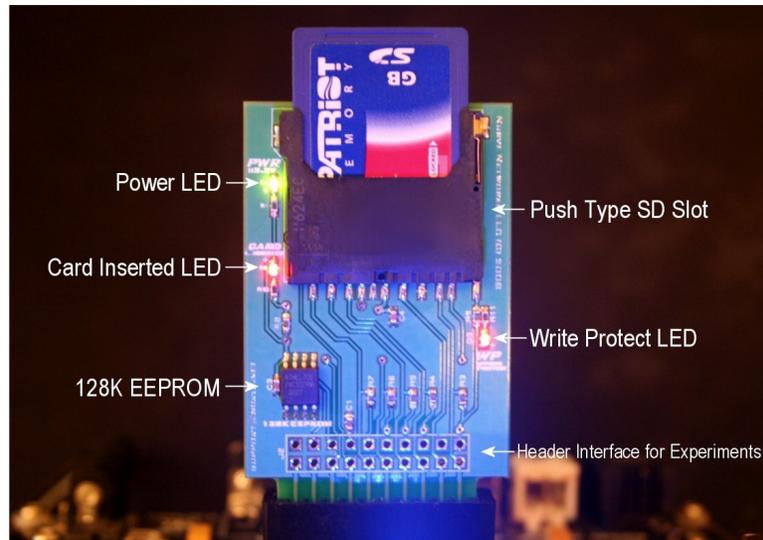
2.1 CD-ROM Contents

The CD-ROM contains the documentation, drivers, demos, and all source code for the SD Max. Additionally, there is a bonus sub-directory with the latest HYDRA demos and software that are user submitted. The CD-ROM layout is as follows:

```

CD_ROM:\
  README.TXT
  AUTORUN.SYS
  LICENSE.TXT
  SOURCES\
    \SD_MENU_IMAGE
  SCHEMATICS\
  DOCS\
    \DATASHEETS
    \FAT
    \SD
    \SPI
  TOOLS\
  GOODIES\
  
```

NOTE: "CD_ROM" is your CD-ROM drive letter; "D:", "E:", etc.

Figure 2.0 – Annotated close up of the HYDRA SD Max Storage Card.

3.0 Introduction and Quick Start

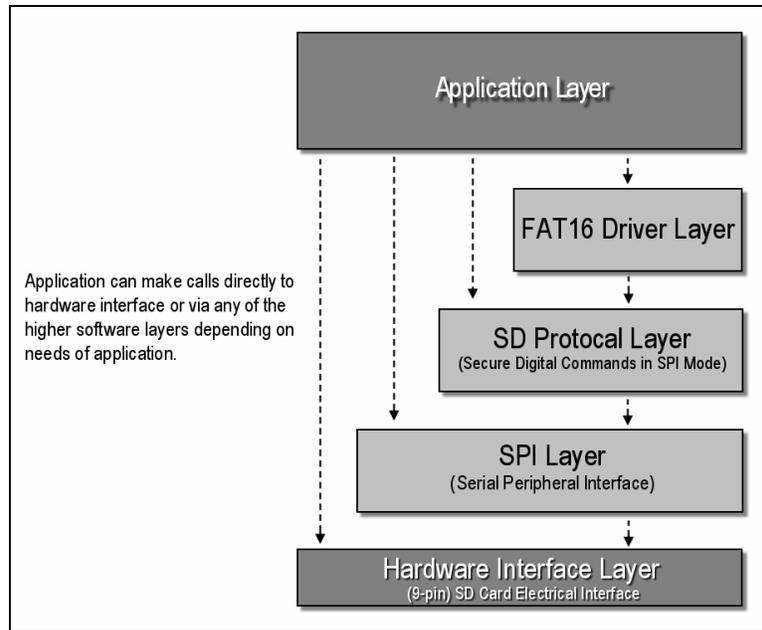
A close-up of the **HYDRA SD Max Storage Card** shown in Figure 2.0. The SD Max card has a simple friction type front loading standard SD card slot that accepts “**standard**” size SD cards (*mini* and *micro* cards can be used with adapters). The SD Max card also has a 128K serial EEPROM on it, so SD card programs can be stored directly on the card and not need to be loaded separately into the HYDRA’s base board 128K EEPROM.

Additionally, the card has 3 LED indicators:

- Power.
- Write Protect.
- Card Inserted.

Both the signals **write protect** and **card inserted** are actually interfaced to the I/O pins of the HYDRA’s expansion slot as well to help determine these if the inserted card is write protected, or that a card is inserted at all. The SD card can be queried as well with **software techniques** to determine if its inserted and if so write protected, but having the extra signals makes life a little easier.

The main idea of the SD card is augment the HYDRA with a complete file system for loading and storing large amounts of data. The SD card format is perfect since the hardware interface is very modest (only 3 lines needed in SPI mode). And the speed is quite respectable at **25MHz** even in **license free SPI mode**. The circuit design (discussed in the next section) supports high speed 4-bit modes as well for users that want to experiment with the other SD card modes for non-commercial applications.

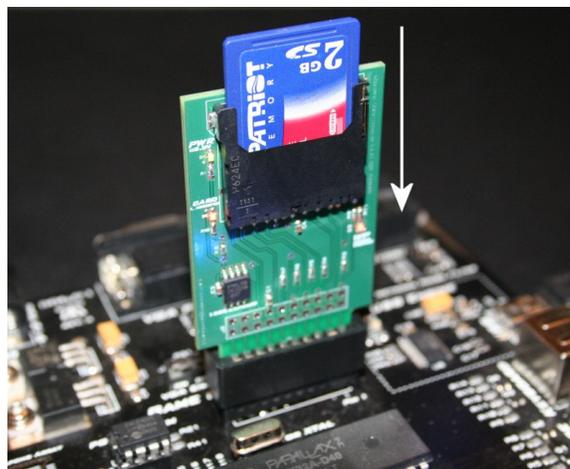
Figure 3.0 – A simplified illustration of the SD Max software model.

The software interface to the SD card is a layered model as shown in Figure 3.0. At the lowest level there is a **SPI driver** which does nothing except send and receive bytes to and from the SD card. On top of this protocol is the **SD SPI mode protocol** which can be used to completely control the SD card as well as read/write “sectors”. But, to interface the cards with the PC and be able to transparently swap the card from the PC to the HYDRA, a **FAT16** file system is needed since that’s what the PC uses to format and access the SD card. Thus, we need yet one more layer on top of the SD card protocol that implements FAT16 which is no easy feat.

NOTE

FAT16 is a file system used on DOS and Windows PCs, originally introduced in 1987 as a successor to the FAT12 system for floppy drives. FAT16 is simply a method of organizing a hard drive or floppy disk into sectors, clusters, and a directory system that allows a file system to be efficiently developed to access files, read and write. More on the FAT16 system later in the manual. A good overview can be found here: http://en.wikipedia.org/wiki/File_Allocation_Table.

The SD card can be used for all kinds of applications from simple file storage to virtual memory and advanced media applications. Later in the summary some ideas about cool things to do with the SD card will be discussed.

Figure 4.0 – Inserting the HYDRA SD Max card into the HYDRA.

3.1 Quick Start Guide

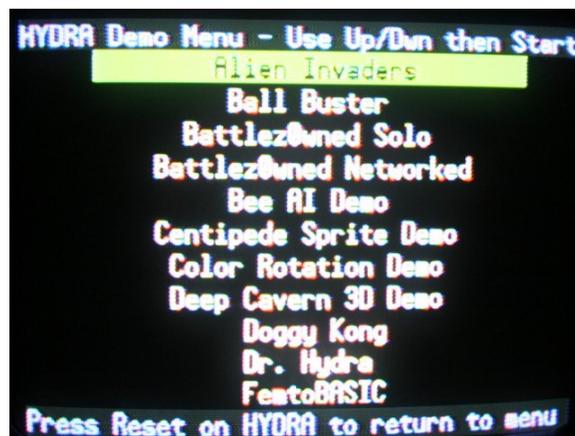
The HYDRA SD Max card's 128K EEPROM is pre-loaded with a little demo program that lists a number of demo, programs, games, and applications that you can scroll thru and select and load into the HYDRA and run as shown in Figure 5.0 below. If you are an old Atari, Commodore 64, or Apple programmer this will look very familiar to the old loaders that used to boot and give you a selection of games to play!

This functionality is accomplished by an SD card reader driver coupled with software that allows Propeller 32K EEPROM *"images"* with extension **.EEPROM** to be loaded directly into SRAM and then the Propeller to be re-booted. A binary image is simply a 32K byte image that is the result of compiling a standard SPIN/ASM program.

The demo binary images that the demo uses are pre-loaded onto the SD card's root directory, so all you have to do is insert the SD card itself into the HYDRA SD Max card slot and then insert the HYDRA SD Max card into the HYDRA itself. This is shown in Figure 4.0. Of course, you can hot slot the SD card itself (as well as the HYDRA SD Max card), thus, first you can insert the HYDRA SD Max card into the HYDRA and then insert the SD card into the SD card acceptor on the HYDRA SD Max card. Here are the steps to play with the pre-loaded demo and games on:

- ✓ **Step 1.** With the HYDRA hooked up to power, TV and the PC, simply insert the SD Max card into the expansion slot facing the front of HYDRA as shown in Figure 4.0, be careful not to force it. The HYDRA can be on or off. Make sure you have the game controller in the *left* controller port of the HYDRA.
- ✓ **Step 2.** Turn the HYDRA on and/or reset it and the card should reset and boot up the test suite. You will see the power LED on the top-left of the card light up as shown in Figure 4.0. If the SD Max card doesn't boot, leave the power on and simply remove and re-insert the card to get a better insertion connection and try hitting reset.
- ✓ **Step 3.** You should see the demo menu on screen as shown in Figure 5.0, use the up/down directional arrows on the game controller to scroll thru the demos, once you find something that sounds interesting press the Start button, the demo will load. When you want to try another demo, simply reset the HYDRA with the Reset button.

Figure 5.0 – The pre-loaded SD loader menu.



If you don't see the menu or the menu shows no programs available to load then either the pre-programmed demo program on the 128K EEPROM needs re-programming, or the SD card needs re-flashing. Refer to the sections below for steps to restore your product.

3.1.1 Re-programming the HYDRA SD Max Card with the Menu Demo Program

With the HYDRA SD Max card inserted into the HYDRA, simply load the program on the CD named **MENU_001.SPIN** located here:

```
CD_ROM:\sources\menu_001.spin
```

and write it to the 128K EEPROM on the HYDRA SD Max card with **F11** from the Propeller IDE. This should refresh the damaged program.

3.1.2 Re-formatting the SD Card with the Demo Binary Images

If the demo programs on the SD card get damaged somehow, you simply need to restore the SD card, which is trivial. You will need an SD card reader plugged into your PC. Then simply plug the SD card into the reader, open the card up and then inspect the files. You should see a long list of files with various names (each with the **.EEPROM** file extension) and a file named DIR.TXT that is used as an internal “**directory**” for the menu program. If you don’t then chances are the files got fragged, so you need to re-fresh them. Here are the steps:

Step 1: Format the SD card FAT16. Perform a full format, *not* a quick format.

Step 2: Locate the folder on the CD in the \Sources directory named \SD_Menu_Images located here:

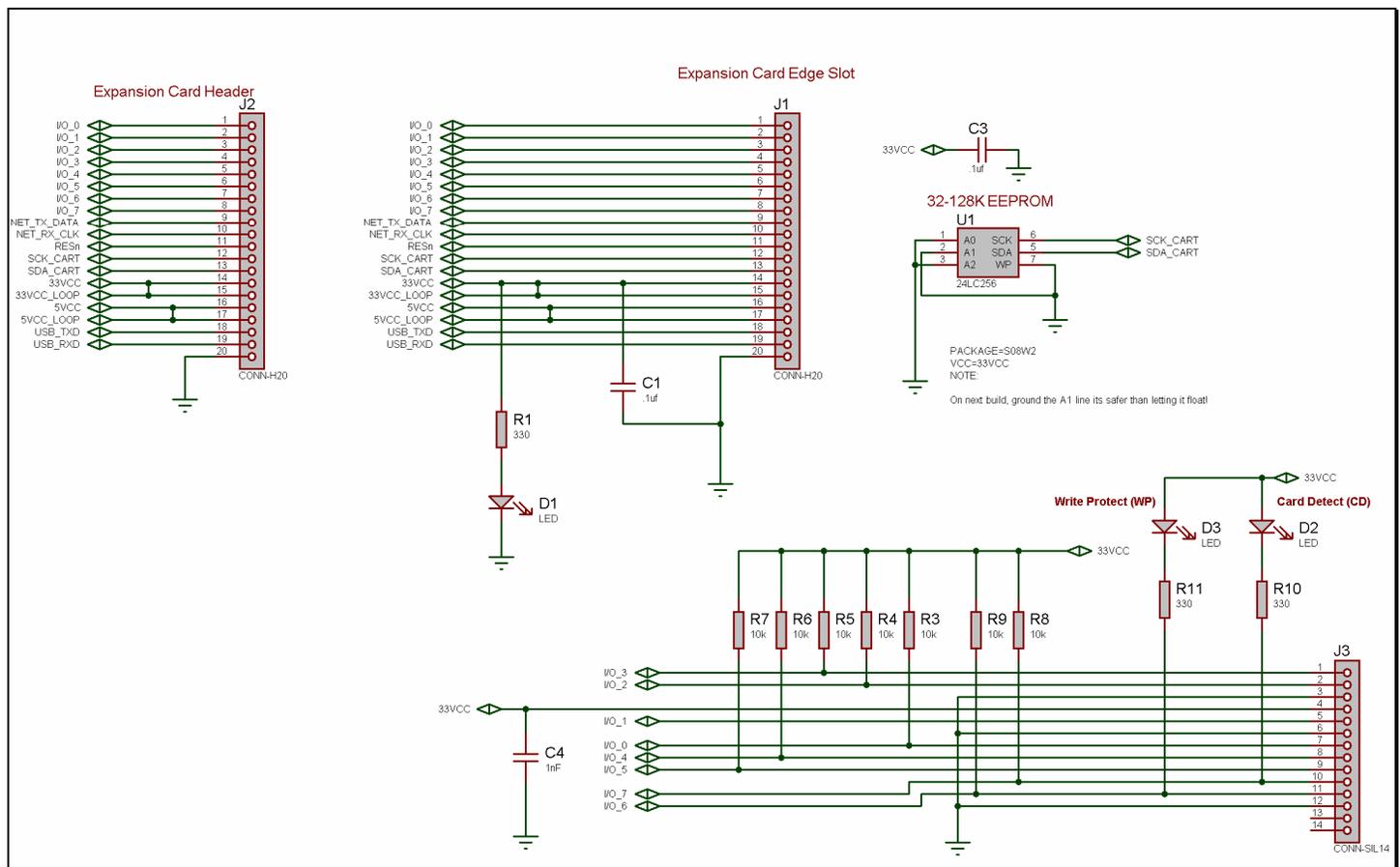
CD_ROM:\Sources\SD_Menu_Images

Enter the directory and **select all** the files and copy them into the SD card. This should re-fresh the card and you are back in business. Note: You will need to do this anytime you want to play with the SD card menu demo. Since the demo assumes the card is freshly formatted and has a root copy of all the files on it.

The menu demo is just a taste of what’s possible with the HYDRA SD Max card. You can load potentially thousands of demos and languages into the HYDRA with a single SD card and never have to remove the card from the HYDRA!

NOTE The menu program and underlying drivers are based on the hard work of **Mike Green’s** Femto BASIC and drivers developed by **Tomas Rokicki** of Radical Eye Software, they are free for use and sources available on the CD as well as the Propeller/HYDRA forums and object exchange at www.parallax.com. The formal HYDRA SD Max driver API written by yours truly is completely in SPIN for educational reasons, thus once you start really hacking away at the SD card you might want to use the faster lower level drivers by Mike Green and Radical Eye Software or write your own.

Figure 6.0 – The HYDRA SD Max Storage Card’s electrical design.



That's about the extent of the electrical design, very simple as you can see. Now, let's take a look at the actual signals themselves and see what's what. Table 1.0 lists the complete electrical interface for all modes of operation of the SD card.

Table 1.0 – SD card interface lines used.

Pin	SD 4-bit mode		SD 1-bit mode		SPI mode	
1	CD/DAT[3]	Data line 3	N/C	Not Used	CS	Card Select
2	CMD	Command line	CMD	Command line	DI	Data input
3	VSS1	Ground	VSS1	Ground	VSS1	Ground
4	VDD	Supply voltage	VDD	Supply voltage	VDD	Supply voltage
5	CLK	Clock	CLK	Clock	SCLK	Clock
6	VSS2	Ground	VSS2	Ground	VSS2	Ground
7	DAT[0]	Data line 0	DATA	Data line	DO	Data output
8	DAT[1]	Data line 1 or Interrupt (optional)	IRQ	Interrupt	IRQ	Interrupt
9	DAT[2]	Data line 2 or Read Wait (optional)	RW	Read Wait (optional)	NC	Not Used

Referring to Table 1.0, depending on the mode of operation the electrical interface of the SD card can change meaning. For example in SD 4-bit mode pin 8 is data line 1, but in SD 1-bit mode its an interrupt line. Therefore, by exporting all the lines to the HYDRA's expansion interface we can write drivers to support the 1-bit and 4-bit modes if we wish. However, we are more interested in the SPI mode which is shown on the right hand side of the table. In this mode, the interface is a straight SPI interface consisting of pin 1,2,3,4,5,6, and 7. Pin 8 and 9 are not needed. Also, note that pins 3 and 6 are ground and 4 is power, so in reality there are only 4 signal lines needed for SPI mode (CS, DI, DO, SCLK).

Nevertheless, we want to support all these modes for experimentation, so we need a connection to every single pin. If you count up all the signals there are a total of 6 signal lines and 3 power lines. Now, since the HYDRA's expansion bus has 8 available lines, we still have 2 more signals we can send thru the interface which is perfect for the **Card Detect** signal and the **Write Protect** signal which are implemented on the card slot itself by means of little switches implemented on the connector itself. That is, when you insert the SD card a short is made which senses as well as the state of the write protect tab. But, these signals are not part of the SD card interface and are solely at the discretion of the manufacturer of the particular SD card slot mechanical you choose for your design. But, we will get to that in a moment.

Right now, let's take a look at the interface between the SD card signals and the HYDRA expansion port itself. Basically, we have 6 data signals from the SD card plus 2 extra signals from the mechanical interface itself (indicating card insertion and write protect tab state) along with the power and grounds. Thus, we have to decide what signal to connect where on the HYDRA's I/O_0 to I/O_7 signal lines. This is mostly arbitrary, but I matched them to the most common connections that people have been using when interfacing SD cards to the HYDRA themselves. Table 2.0 below shows the final electrical interface from the SD card to the HYDRA expansion interface.

Table 2.0 – SD card interface to HYDRA expansion interface mapping.

SD Card Pin	Function	Name (EXP/PROP/LOG)
SD 1	(CS/DAT3 - chip select/DAT3)	I/O 3 (4/24/P19)
SD 2	(DI/CMD - data in/command)	I/O 2 (3/23/P18)
SD 3	GND	GND (20)
SD 4	3.3V	3.3V (14)
SD 5	(CLK/SCK – clock/serial clock)	I/O 1 (2/22/P17)
SD 6	GND	GND (20)
SD 7	(DO/DAT0 - data out/DAT0)	I/O 0 (1/21/P16)
SD 8	(DAT1/IRQ)	I/O 4 (5/25/P20)
SD 9	(DAT2/NC)	I/O 5 (6/26/P21)
Extra Mechanical SD card holder signals not part of the SD interface		
SD 10 ⁽¹⁾	(CD/Card Detect)	I/O 7 (8/28/P23)
SD 11 ⁽¹⁾	(WP/Write Protect)	I/O 6 (7/27/P22)

Note 1: SD 10 and 11 are not part of the SD interface, but simply pins that happen to be on the particular SD card mechanical chosen for the HYDRA SD Max (AVX Inc. makes them). Most SD card slots have similar signals.

The table is a little tricky to read especially in the right most column, so let's try and example. Let's trace SD card pin 7 which is the **D0** or **Data Out** line. Starting from the left most column, SD 7's function is **D0** or **Data Out**. Meaning, in 4-bit mode its **D0** of the 4-bit data, or in 1-bit mode it's the single "**Data Out**" line. Then moving to the right column, each entry is in the format "**signal name**" followed the three numbers in the order (**EXP**ansion port pin, **PROP**eller pin, **LOG**ical Propeller signal name).

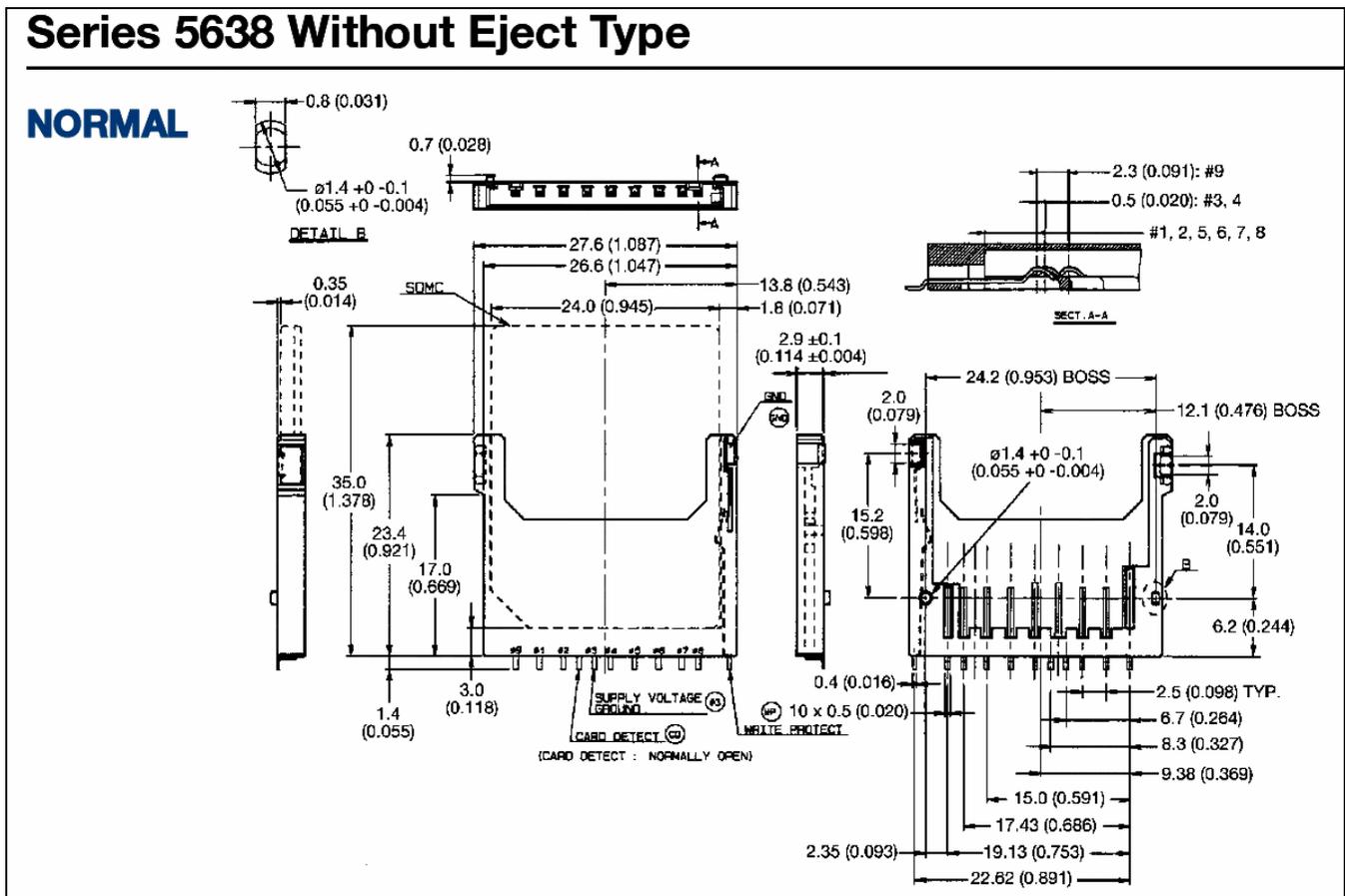
Thus, the SD card's pin 7 is connected thru the HYDRA expansion port's pin 1 which in turn is connected to the Propeller chip at **pin 21** which is named **P16** on the Propeller chips I/O interface. A little confusing, but that's what happens when you go from interface to interface to the chip to the final internal Propeller signal name. At the end of the day, as a programmer all you need to know are the Propeller's I/O pin assignments which are as follows:

```
' SPI (serial peripheral interface) interface pins to SD card
spiDO = 16 ' SD data out
spiCLK = 17 ' SD clock
spiDI = 18 ' SD data in
spiCS = 19 ' SD card select

' these two signals aren't really part of the SPI interface, but are defined in this block of code
sdWP = 22 ' write protect line
sdCD = 23 ' card detect line
```

Next, let's take a look at the actual mechanical SD card holder used in this project. I selected one from AVX Corporation based on price, interface, and availability.

Figure 8.0(a) – The HYDRA SD Max SD card connector from AVX Corporation.



4.2 Mechanical Interface

All SD cards must plug into a physical connector that is mounted to the PCB itself. There are hundreds of manufacturers that make these connectors and selecting one is more subjective than objective in many cases. However, some things that you should look out for if you decide to design an SD card reader are:

1. Is the connector a **normal** or **reverse** type?
2. Does the connector have a spring loaded release? Nice, but expensive.
3. Is the connector plastic or metal or have adequate shielding?
4. Cost, cost, cost! And did I mention cost?

TIP

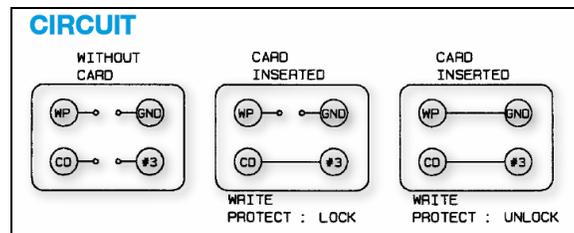
Normal or reverse connector indicates which way the SD card goes into the connector. **Normal** means label facing the **front** and the contacts facing the PCB. Reverse means the opposite. This is important depending on which way the PCB is oriented in the final product. In our case, the PCB of the SD Max card plugs face outward on the HYDRA, so when you plug in the SD card it faces outward as well.

In the case of the SD Max card we don't need all the bells and whistles, so we decided on a standard normal connector, plastic, without a spring eject mechanism (which is probably a good idea since its just one more thing that can break). The connector itself is manufactured by AVX Corporation and you can find the data sheet on the CD here:

CD_ROM:\docs\datasheets\avx_sd_mech_5638.pdf

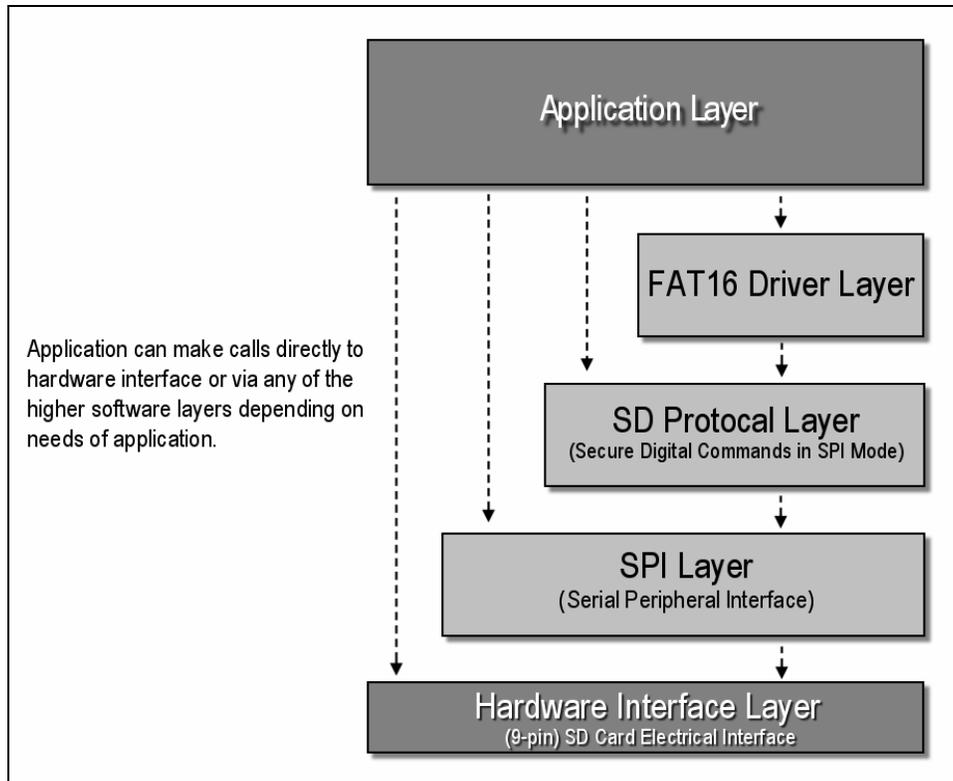
Figure 8.0(a) above shows an excerpt from the datasheet illustrating the mechanical specifications of the connector. Notice that there are a lot of specs and this is one of the biggest challenges when making an SD card reader; designing the PCB to accept the connector is no easy task due to the small tolerances. Also, notice the additional annotation relating to the Write Protect and Card Inserted circuits as shown in Figure 8.0(b).

Figure 8.0(b) – Card connector from AVX Corporation showing Write Protect and Card Inserted logic.



These extra signals are available on the SD connector, but not part of the SD specification, thus they aren't in any specific location manufacturer to manufacturer. Therefore, when designing SD card PCBs you will find that if you change the SD card connector, you will have to redesign the PCB footprint quite a bit.

Figure 9.0 – HYDRA SD Max software / hardware model.



5.0 Interfacing to the HYDRA SD Max Card

In this section we'll discuss interfacing to the SD card at all levels; electrical to FAT16. Referring to Figure 9.0 above you can see that the SD card software / hardware model looks a little like the 7-layer OSI model for networking. There is the **application layer** at the top. Then under it is the **FAT16 file system** which facilitates interfacing the SD card with the PC (optional). Next down is the **SD card protocol layer** which is what the SD card speaks and gives access to the internal controller in the SD card as well as raw sector reads and writes. This is followed by the actual communication layer to the SD card which is a **SPI interface** consisting of 4-lines; data out, data in, clock, and chip select. Then finally where the rubber meets the concrete are the **electrical connections** to the Propeller chip itself that we can toggle via port I/O commands. Thus, we have to implement every single one of these layers, so we have our work cut out for us!

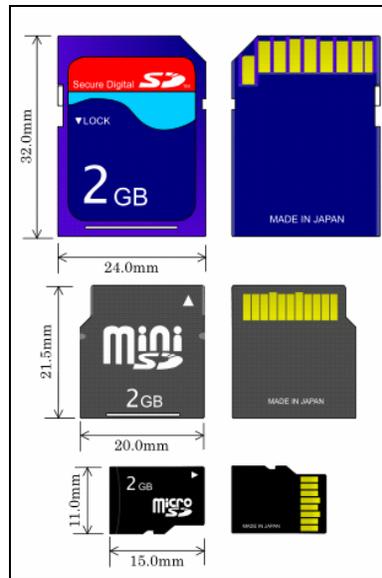
Our goal is to become familiar with writing a SPI driver all the way up to a DOS/Windows compliant FAT16 file system. There is a lot of material to cover on these subjects and each warrants a lot of information, so I suggest referring to the related links and documents on the CD for more in depth coverage. Take a look in these sub-directories for a number of relevant documents:

```

CD_ROM:\DOCS\
  \FAT - Documents about the FAT file system.
  \SD - Documents about the SD card protocol and design in general.
  \SPI - Documents about the SPI interface and protocol.
  
```

But, before we begin let's take an aside briefly and talk about some of the history and design of the SD card protocol.

Figure 10.0 – SD card physical packaging examples.



5.1 SD Card Overview & History

The SD card was originally invented in 1999 by a collaboration between **Panasonic**, **Toshiba**, and **SanDisk** (lead developer) to compete with **Sony's Memory Stick** technology which was proprietary and a closed standard. The secure digital aspect of SD card simply has to do with the added functionality supporting encryption that was missing in the **MMC** cards beforehand (**M**ulti **M**edia **C**ards) that the SD card was supposed to replace. Encryption was requested since the main purpose of the SD card was to hold media data and media providers wanted encryption and some kinds of digital rights management.

In any event, the SD has become the most popular solid state mass storage device along with Compact Flash storage devices. Due to this success and the continually shrinking size of consumer electronics new, smaller size, and faster SD cards are now available. Take a look at Figure 10.0 to see the **mini** and **micro** SD card footprints. The mini size is cute, but the micro is ridiculously small, similar to a cellular phone **SIM** (**S**ubscriber **I**dentify **M**odule) card. The size of the cards doesn't effect the storage capacity. Also, each smaller footprint is designed with the exact same electrical and signal interface, just smaller and adapters are available to plug each smaller size into each larger size, so even if you have a normal SD card reader, you can use a micro SD card with a little adapter.

Architecturally, each SD card has a microcontroller inside along with the actual flash memory storage area. Therefore, when communicating with an SD card you are more or less communicating with another computer. Thus, SD cards are rather complex pieces of technology that abstract the memory interface and the I/O interface. For example, before SD cards there were all kinds of memories; RAMs, ROMs, FLASH, EEPROM, etc. but the problem was that if you wanted to use one of these memories then you had to build a physical interface to them. SD cards removed this problem by separating the interface from the electronics/memory, so that a user could use a standardized set of communication lines to talk to a mass storage device. This is the genius of the SD card design. No matter what's inside the SD card; NAND flash, NOR flash, holographic memory or little elves, its always the same from the electrical point of view and the programmers interface. That is, 9 signals that always mean the same thing.

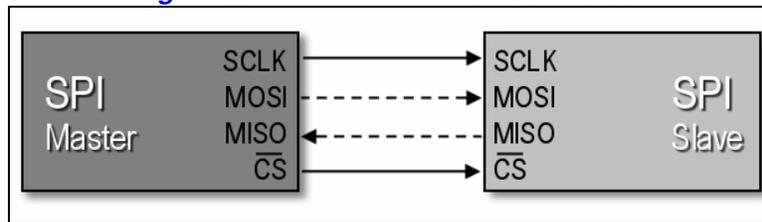
From a programmers point of view the SD card acts like a number of registers and a memory. Communication to the SD card is done thru a serial/parallel interface by issuing commands and sending and receiving data. The memory in an SD card is arranged as a number of sectors (512 bytes each usually) and currently SD cards max out at about 4GB of memory, but 8GB are available. The communication speeds of SD cards depend on the version of the card; 1.0, 1.01, 1.1, or 2.0. Most manufacturers stick with **version 1.1** specs which run at about 10-20 megabytes per second. The speed measuring of SD cards is based on CD ROM standard. Where **1X** CD speed is about **150 kB** (kilobytes) per second. Therefore, an SD card that runs at 66X speed means it runs at $66 \times (150 \text{ kB}) = 9.9\text{MB}$ per second. Currently, 66X and 133X are commonly available with even higher speeds if you're will to pay the money. However, these speeds are only attainable in licensed true SD protocol modes, not the free SPI mode (which we are using).

Finally, SD cards are nothing more than a collection of sectors, but to use with PCs, SD cards are setup or "formatted" to mimic the target file system of the PC they are connected to. Typically, this is FAT16 or FAT32 both Microsoft standards. Thus, when using an SD on the HYDRA if you want to plug it into the PC you must format in one of these standards.

Preferably **FAT16** since it's the only format that the drivers support. Of course, you can code FAT32 drivers if you like yourself.

Summing up, SD cards are memory **plus** microcontrollers. They are interfaced via a **9-pin** electrical interface which is always the same. Internally, they are organized as a collection of **sectors** that the programmer can access and read and write to via issuing commands to the internal microcontroller. If desired, one can layer a file system on top of the raw sectors like FAT16 or whatever is desired, but you the programmer must do this, SD cards have **no built in** internal support for a file system. The actual electrical communication to SD cards is performed via a 1-bit or 4-bit protocol. However, there is also a standard SPI protocol that can be used which is straightforward to implement. Therefore, the best place to start talking with SD cards is to build the SPI software and work our way up. In the following sections you will see SPIN code snippets excerpted from the API library, don't worry if they don't make complete sense since we will cover the API library in detail later in the manual, but for now if you see a global, constant that isn't defined, trust that it is and you will learn about it later.

Figure 11.0 – The SPI electrical interface.



5.2 SPI Bus Basics

SPI stands for **Serial Peripheral Interface** originally developed by **Motorola**. Its one of two very popular modern serial standards including **I²C** which stands for **Inter Integrated Circuit** by **Phillips**. SPI unlike I²C (which has no separate clock) is a clocked synchronous serial protocol that supports full duplex communication. However I²C only takes **2 wires** and a ground. Where SPI needs **3 wires**, ground, and potentially chip select lines to enable the slave devices. But, SPI is much faster, so in many cases speed wins over and the extra clock line is warranted. The advantage of I2C is that you can potentially hook hundreds of I²C devices on the same 2-bus lines since I²C devices have addresses that they respond to. SPI bus protocol on the other hand requires that every SPI slave has a chip select line.

Figure 11.0 shows a simple diagram between a **master** (left) and a **slave** (right) SPI device and the signals between them which are:

- **SCLK** - Serial Clock (output from master).
- **MOSI/SIMO** - Master Output, Slave Input (output from master).
- **MISO/SOMI** - Master Input, Slave Output (output from slave).
- **SS** - Slave Select (active low; output from master).

Note: You might find some devices with slightly different naming conventions, but the idea is there is data out and data in, a clock, and some kind of chip select.

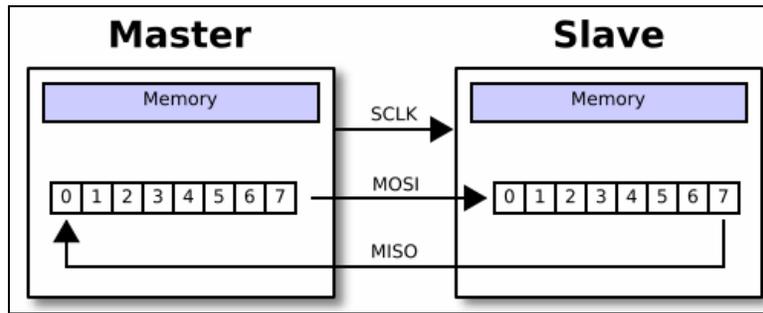
If you refer back to the SD Max interface signals then you will see that the SD card signals have slightly different names. Assuming the HYDRA is the Master and the SD card is the slave, the mapping of SPI signals are shown in Table 3.0 below:

Table 3.0 – Mapping of SD card signals to official SPI signal names.

SD Card Pin	SD Name	SPI Signal Name	Function
1	CS	SS	Chip select active low.
2	DI	MOSI	Master out, slave in.
5	CLK	SCLK	Clock signal.
7	DO	MISO	Master in, slave out.
<i>Note: Power and ground signals of course need to be connected as well.</i>			

SPI is very fast since not only is it clocked, but it's a simultaneous **full duplex** protocol which means that at you clock data out of the master into the slave, data is clocked from the slave into the master. This is facilitated by a transmit and receive bit buffer that constantly re-circulates as shown in Figure 12.0.

Figure 12.0 – Circular SPI buffers.



The use of the circular buffers means that you can send and receive a byte in only 8 clocks rather than clocking out 8-bits to send, then clocking in 8-bits to receive. Of course, in some cases the data clocked out or in is “dummy” data, meaning when you write data and you are **not** expecting a result the data you clock in is garbage and you can throw it away. Likewise when you do a SPI read, typically you would put a \$00 or \$FF in the transmit buffer as dummy data since something has to be sent and it might as well be predictable.

Sending bytes with SPI is similar to the serial RS-232 protocol, you place a bit of information on the transmit line, then strobe the clock line (of course RS-232 has no clock). As you do this, you also need to read the receive line since data is being transmitted in both directions. This is simple enough, but SPI protocol has some very specific details attached to it about **when** signals should be read and written that is, on the rising or falling edge of the clock as well as the polarity of the clock signal. This way there is no confusion about edge, level, or phase of the signals. These various modes of operation are logical called the SPI mode and are listed in Table 4.0 below:

Table 4.0 – SPI clocking modes.

Mode #	CPOL (Clock Polarity)	CPHA (Clock Phase)
0	0	0
1	0	1
2	1	0
3	1	1

Mode Descriptions

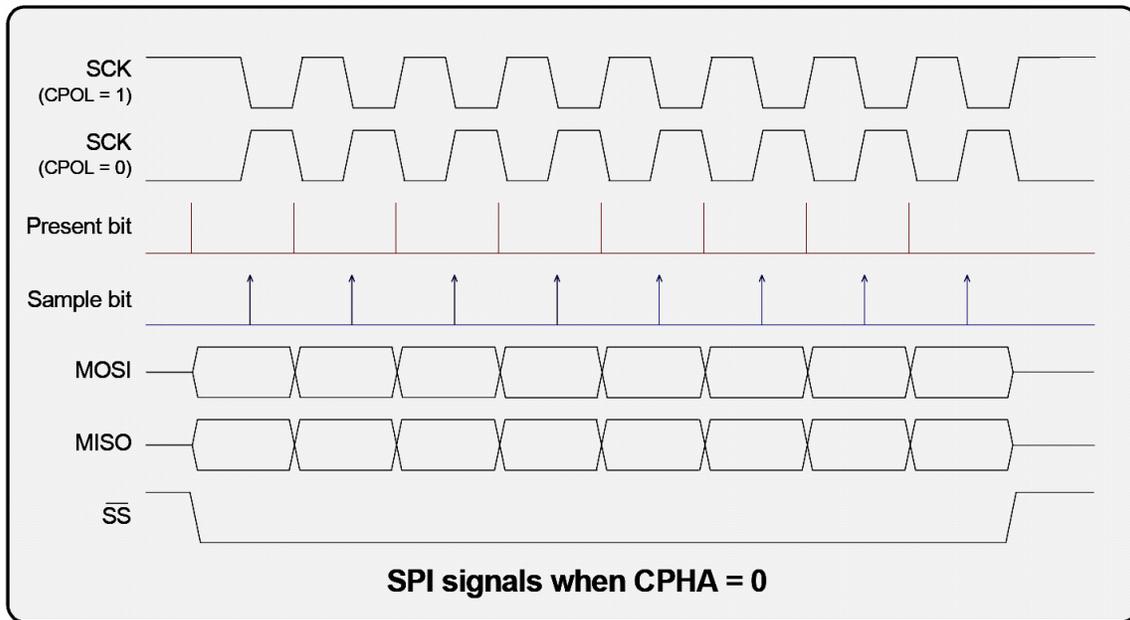
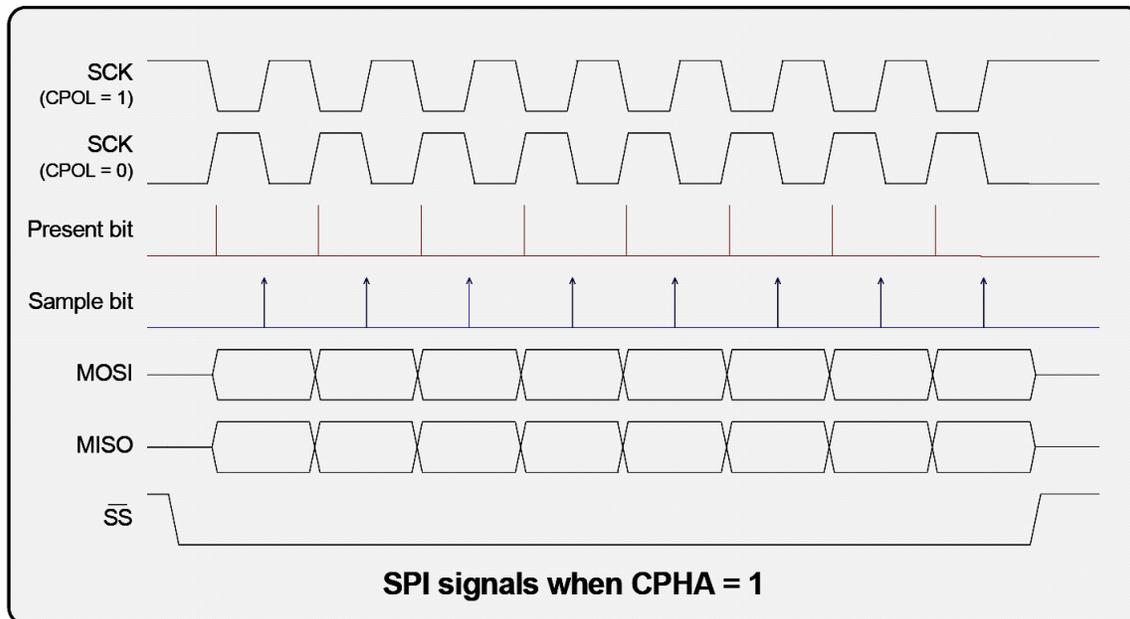
Mode 0 – The clock is **active** when **HIGH**. Data is **read** on the **rising edge** of the clock. Data is **written** on the **falling edge** of the clock (default mode for most SPI applications).

Mode 1 – The clock is **active** when **HIGH**. Data is **read** on the **falling edge** of the clock. Data is **written** on the **rising edge** of the clock.

Mode 2 – The clock is **active** when **LOW**. Data is **read** on the **rising edge** of the clock. Data is **written** on the **falling edge** of the clock.

Mode 3 – The clock is **active** when **LOW**. Data is **read** on the **falling edge** of the clock. Data is **written** on the **rising edge** of the clock.

Note: Most SPI slaves **default to mode 0**, so typically this mode is what is used to initiate communications with a SPI device.

Figure 13.0(a) – SPI timing diagrams for clock phase polarity (CPHA=0).**Figure 13.0(b) – SPI timing diagrams for clock phase polarity (CPHA=1).**

5.2.1 Basic SPI Communications Steps

Figures 13.0(a) and (b) show the complete timing diagrams for all variants of **clock polarity** (CPOL) and **clock phase** (CPHA). You must adhere to these timing constraints during communications. In most cases, you will use **mode 0** since it's the default that most SPI devices boot with. On the HYDRA there is no support for SPI buses, so we must bit bang the protocol ourselves by toggling the I/O lines connected to the SPI interface of the SD card. Once again these signals and Propeller I/O bits are:

```
spiDO = 16 ' SD data out from SD (input to HYDRA)
spiCLK = 17 ' SD clock (output from HYDRA)
spiDI = 18 ' SD data in (output from HYDRA)
spiCS = 19 ' SD card select (output from HYDRA)
```

Where the signals are from the perspective of the SD card (the slave in SPI terminology). For example spiDO is data out from the SD card, but data in to the HYDRA. Moving on, the first step to communicating with the SD SPI interface is to setup the I/O directions. In SPIN, the code looks like (excerpt from SD library):

```
PUB SPI_Init(mode)
' DESCRIPTION: this function initializes the hardware interface to the SPI bus. on the HYDRA SD MAX card
' (as with most SPI buses) there are only 4 signals; clk, data in, data out, and chip select
'
' INPUTS: mode - controls the phase of the clock and the logic levels
' mode 0 - default support
' mode 1 - not supported
' mode 2 - not supported
' mode 3 - not supported
'
' OUTPUTS: none
'-----
' set directions of SPI interface, be careful not to clock the SPI interface accidentally
' also, data in and data out are referenced from TARGET, so DI means the input to the SPI
' device which would be
' an OUTPUT from the propeller
OUTA [ spiDI ] := 0 ' set output LOW
DIRA [ spiDI ] := 1 ' set "data in" to output

DIRA [ spiDO ] := 0 ' set "data out" to input

OUTA [ spiCLK ] := 0 ' set output LOW
DIRA [ spiCLK ] := 1 ' set "clock" to output

OUTA [ spiCS ] := 1 ' set output HIGH (de-select the device)
DIRA [ spiCS ] := 1 ' set "chip select" to output
' end SPI_Init
```

Once the I/O interface is initialized then you can clock in/out data to the SPI bus. Here's a sample function to transmit and receive (excerpt from SD library):

```
PUB SPI_Send_Recv_Byte(spi_data8 | spi_result8, index)
' DESCRIPTION: this functions sends spi_data8 and receives spi_result8, remember all spi operations are
' circular, so when a byte is sent a byte is received at the same time, you can ignore the sent or received
' byte as you wish. For example, to do a read just send the dummy value $FF
' NOTE: assumes caller is controlling the chip select line
'
' INPUTS: spi_data8 - 8 bit data that is going to be sent
' OUTPUTS: returns the 8-bit data received at the same time
'-----
' reset result
spi_result8 := 0

' shift 8-bits of data out while shifting in 8-bits of data
' data is sent and receiving MSB to LSB, so bit 7,6,5,4,3,2,1,0
' assume SPI interface is in mode 0/1, later make more flexible
repeat index from 0 to 7
' place next bit on data in pin to device
OUTA [ spiDI ] := ((spi_data8 & $80) >> 7)

spi_data8 <<= 1 ' shift data to left and prepare next bit for transmission

' pulse the clock line
OUTA [ spiCLK ] := 1

' data is ready now on output line from device
spi_result8 <<= 1 ' shift result to left
spi_result8 |= INA [ spiDO ] ' read data bit and OR into result

' finish clock pulse
OUTA [ spiCLK ] := 0

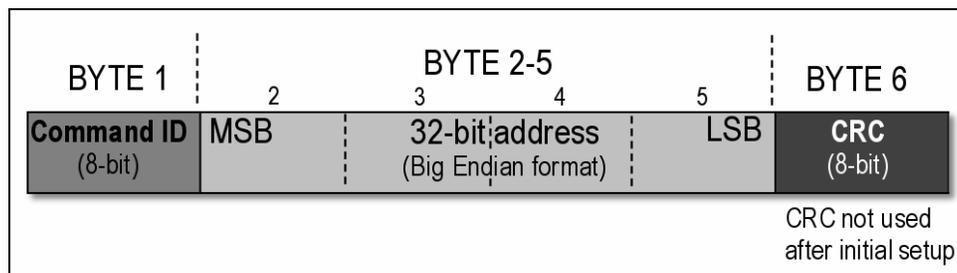
' return result
return (spi_result8)
' end SPI_Send_Receive_Byte
```

Notice that there is no timing related logic in the function. Since SPI protocol is totally **synchronous** the master in charge of the clock can run the clock as fast (to maximum speed) or slow (static if desired). Also, notice the data is sent out most significant bit (msb) to least significant bit (lsb).

These above two functions are the entire SPI software necessary to communicate with a SPI device! The only detail is that the **chip select** (CS) line must be toggled by the application as desired during operations. Later when we cover the API, we will see a couple more functions layered on top that read and write SPI data, but these are nothing but dummy functions that send dummy data or ignore the results for read and write operations respectively. The important point is that byte is always sent and received at the same time with SPI protocol.

Now, that we have the SPI protocol layer implemented, let's move on to the SD communications layer which is where things get a little more complicated.

Figure 14.0 – Format of SD card commands in SPI mode.



5.3 SD Card Communications Protocol

The complete SD card specification is a monster of a document, not because of its length, but because of its lack of clarity. For those interested, you can read all about in **SD_SDIO_specs_v1.pdf** located on the CD here:

CD_ROM:\docs\sd\SD_SDIO_specs_v1.pdf

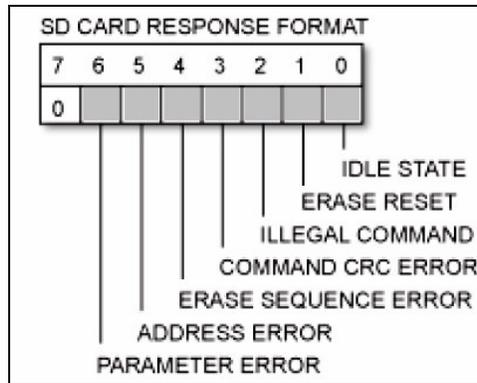
Luckily, we aren't using the SD protocol mode, but rather the SPI mode which is much easier to deal with. Of course, its not as fast as SD modes, nor does SPI mode have all the features. But, it allows to read and write sectors of the SD card and that's all that matters. Also, in SPI mode both SD cards and MMC cards work in the same way, so the software, drivers, and demos discussed here would theoretically work with MMC cards as well. Considering that, you can mod the SD Max if you wish to support them with some hardware hacking.

In any event, all communication with the SD cards is done via the SPI interface discussed earlier, so when we talk about writing and reading, the SPI interfaced is obviously used. This differs from "SPI" mode that we want to put the SD into. This is actually the first step when initializing the SD card. Figure 14.0 shows how commands are formed for the SD card. They are 6 bytes long and consist of:

Byte 1 : 8-bit Command ID.
 Byte 2,3,4,5 : 32-bit Address.
 Byte 6 : 8-bit CRC checksum.

The 32-bit address must be formatted in **Big Endian** format, that is high byte to low byte. This is the opposite of the Intel and Propeller format which are **Little Endian** machines. Therefore if you want to send \$FF_AA_00_11 in Big Endian then you would send \$FF, followed by \$AA, \$00, and finally \$11 to complete the address. Of course, the address word might not be important for the particular command, if not, always make it \$00_00_00_00. Finally, the **CRC** checksum byte (**Cyclic Redundancy Check**) is a byte that you the sender must compute based on bytes 1..5. The SD card after receiving bytes 1..5 compares the CRC you send to the CRC it computes and if not the same, will return an error. Also, when the SD card receives a command it sends back a response along with a CRC as well. You are free to inspect the CRC if you wish, but our library ignores it. After you send the 6-byte command to the SD card it always responds with a 1-byte response code. These response codes mean different things in different situations, but generally are used to catch errors. Table 5.0 below shows the bit encoding for the response codes:

Table 5.0 – Response code binary bit encoding format.

**NOTE**

When the SD is in SPI mode it doesn't require CRC bytes, therefore, they can be anything. However, before the SD card is in SPI mode, it does require the CRC be correct. If you're interested in how the computation of CRC bytes is done try this link: http://en.wikipedia.org/wiki/Cyclic_redundancy_check. SD and MMC cards use the CRC-7 method.

The actual code to send a command to the SD card is shown below. Notice that we always send a CRC of \$95, this is because its assumed the **first command** sent to the SD card is to **switch to SPI mode**. After which the CRC is ignored, but still necessary in the byte stream, so it doesn't hurt to send the same CRC each time, only the first time does the value actually matter.

```
PUB SD_Send_Command( command8, address32 ) | response, index
' DESCRIPTION: This function is the main workhorse of the SD interface and is used to send commands
' to the SD card via the SPI interface. All SD commands consist of 6 bytes in the following format:
'
' Command id      (1-byte) This you get from the SD specification.
' 32-bit address (4-bytes) In big endian form.
' CRC result      (1-byte) Error checking byte.
'
' The address is in big endian form, that is high byte to low byte (opposite of PC and propeller)
' The cycle redundancy check byte is sent in CRC7 or CRC16 format for all transactions
' except status tokens. CRC is of course a method of error detection, but not
' correction. If a CRC fails to match then the data must be sent or requested again.
' the algorithms for CRC7 and CRC16 are well documented and are basically division algorithms
' where the CRCs are the remainders. In our case, we are always going to send $95 which
' is the CRC for the first command that will be sent that matters, after that CRCs are
' ignored in SPI mode, later add code to compute the CRC for kicks if you like.
' Also, note the use of SD_RESPONSE_LIMIT, this allows the command to be sent over and over
' until there is a response, but if the attempts exceed this constant, there is something
' wrong with the SD card.
'
' INPUTS: command8 - 8-bit command from SD card specification refer "SD Card Spec" from SD Association.
'         address32 - 32-bit address little endian format (converts to big endian before transport)
'
' OUTPUTS: returns status of command to SD card, typically $FF means error,
'          but for each command response can be different
'
'-----
' first send command
SPI_Send_Recv_Byte(command8)

' now send 4-bytes of address in MS-Byte to LS-byte format
SPI_Send_Recv_Byte( (address32 >> 24) & $FF)
SPI_Send_Recv_Byte( (address32 >> 16) & $FF)
SPI_Send_Recv_Byte( (address32 >> 8)  & $FF)
SPI_Send_Recv_Byte( (address32 >> 0)  & $FF)

' finally the CRC
SPI_Send_Recv_Byte( $95 )

' now wait for response
response := $FF ' default response is $FF which means error or time out

repeat index from 0 to SD_RESPONSE_LIMIT
  read response by sending dummy data
  response := SPI_Send_Recv_Byte( $FF )

' DEBUG CODE - send out number of iterations the response took to get back
```

```

debug_wait_res_num_iter := index
' test if we have a non-$FF response
if (response <> $FF)
' send dummy data to allow SD card to finish operation, needed after all commands
SPI_Send_Recv_Byte ( $FF )
' return results
return ( response )

' if we got here then something bad happened!
return ( $FF )

' end SD_Send_Command

```

The command is straightforward, but there are some interesting error handling techniques in the code. First, notice that we send the bytes one at a time, then we continually send \$FF to the SPI interface at the same time reading back the response. Once we get a non-\$FF value then we exit the iteration loop. The idea here is that if we send \$FF then the SD card will send \$FF back to us (remember a circular buffer), until it finishes the command request at which point it changes the \$FF to something else. Then we can review the response up the call chain. Typically, **SD_RESPONSE_LIMIT** is typically set to 10-500 iterations. In the SD Max driver's its set to 256. If the card doesn't respond by then something is wrong.

The next question of course is what commands does the SD card support? Table 6.0 below lists the complete command set for SD SPI mode.

Table 6.0 – Command list for SD SPI mode.

Cmd ID	Abbreviation	SDMEM Mode	SDIO Mode	Comments
CMD0	GO_IDLE_STATE	Mandatory	Mandatory	Used to change from SD to SPI mode
CMD1	SEND_OP_COND	Mandatory		
CMD5	IO_SEND_OP_COND		Mandatory	
CMD9	SEND_CSD	Mandatory		CSD not supported by SDIO
CMD10	SEND_CID	Mandatory		CID not supported by SDIO
CMD12	STOP_TRANSMISSION	Mandatory		
CMD13	SEND_STATUS	Mandatory		Includes only SDMEM information.
CMD16	SET_BLOCKLEN	Mandatory		
CMD17	READ_SINGLE_BLOCK	Mandatory		
CMD18	READ_MULTIPLE_BLOCK	Mandatory		
CMD24	WRITE_BLOCK	Mandatory		
CMD25	WRITE_MULTIPLE_BLOCK	Mandatory		
CMD27	PROGRAM_CSD	Mandatory		CSD not supported by SDIO.
CMD28	SET_WRITE_PROT	Optional		
CMD29	CLR_WRITE_PROT	Optional		
CMD30	SEND_WRITE_PROT	Optional		
CMD32	ERASE_WR_BLK_START	Mandatory		
CMD33	ERASE_WR_BLK_END	Mandatory		
CMD38	ERASE	Mandatory		
CMD42	LOCK_UNLOCK	Optional		
CMD52	IO_RW_DIRECT		Mandatory	
CMD53	IO_RW_EXTENDED		Mandatory	Block mode is optional
CMD55	APP_CMD	Mandatory		
CMD56	GEN_CMD	Mandatory		
CMD58	READ_OCR	Mandatory		
CMD59	CRC_ON_OFF	Mandatory	Mandatory	
ACMD13	SD_STATUS	Mandatory		
ACMD22	SEND_NUM_WR_BLOCKS	Mandatory		
ACMD23	SET_WR_BLK_ERASE_COUNT	Mandatory		
ACMD41	SD_APP_OP_COND	Mandatory		
ACMD42	SET_CLR_CARD_DETECT	Mandatory		
ACMD51	SEND_SCR	Mandatory		SCR includes only SDMEM information.

Note: The command code IDs start at \$40 (64). For example, CMD0 = \$40+0 = \$40, CMD17 = \$40 + 17 = \$51.

Notice **CMD0**? This command is very important and literally the first command that we need to issue to the SD card. This command tells the SD card to switch into SPI mode. Therefore, we are going to issue this command first. After which the

SD card should respond with a response byte of **\$01** to indicate success. This brings us to Table 7.0 which is a short listing of SD SPI commands that we are going to support and their respective response bytes.

Table 7.0 – Subset of SD SPI commands needed to implement our SD card drivers.

Command	Mnemonic	Argument	Response ⁽²⁾	Description
0 (\$40)	GO_IDLE_STATE	None	\$01	Resets SD card and place in SPI mode.
1 (\$41)	EXIT_IDLE_STATE	None	\$01	Exits reset mode.
17 (\$51)	READ_SINGLE_BLOCK ⁽³⁾	Address	\$00	Reads a block at byte address.
24 (\$58)	WRITE_BLOCK ⁽⁴⁾	Address	\$00	Writes a block at byte address.
55 (\$77)	APP_CMD ⁽¹⁾	None	\$00	Prefix for application command.
41 (\$69)	SEND_OP_COND ⁽¹⁾	None	\$00	Application command.

Note 1: the last two command are not mandatory, but help differentiate if the card is SD or MMC. Only SD can reply to these commands.
Note 2: Refer to Table 5 for encoding of response byte.
Note 3: When reading a block, a data token of \$FE will be received after the initial response code of \$00. Then the next bytes will be the sector data.
Note 4: After the write command is sent, the SD card expects the data token \$FE to follow signifying the host is ready to send bytes.

5.3.1 Placing the SD Card into SPI Mode

Referring to Table 7.0 above the command to place the SD card into SPI mode is called **GO_IDLE_STATE** and the command id is equal to \$40. So the first command we must send to the SD card looks like:

```
Command id:  $40
Address:     $00_00_00_00
CRC:        $95
```

TIP

Notice that the CRC is \$95. This first command is the only time you need to send a valid CRC byte after the SD card switches to SPI mode, the incoming CRC is discarded by the SD card.

After the command completes and the function returns then the response byte should be **\$01** which means all went well. We could leave it at this, but the SD spec gives some other commands that one should send to make sure the card is a real SD card as well. This is facilitated with some extra commands that MMC cards don't respond to, thus a response means "SD card". Of course, if you wanted to support both SD and MMC cards in SPI mode you wouldn't care, but for our purposes let's make sure we are talking to a SD card.

The extra commands are **APP_CMD** (55) followed by **SEND_APP_OP_COND** (41). Each command is sent until a **\$00** response byte is returned at which time its guaranteed a legitimate SD card is connected. The code to initialize the SD card and "mount it" is shown below:

```
PUB SD_Mount | index, index2, sd_response, cmd_num_attempts
' DESCRIPTION: This function mounts the SD card which is complex process consisting of a number of steps.
' 1. Power up.
' 2. SD card must be placed into "idle state"
' 3. Then taken out of idle state.
' 4. Next, the card is tested to make sure its an SD and not MMC.
' 5. Finally, if all passes then the card is ready to go and "mounted".
'
' INPUTS: none
'
' OUTPUTS: returns $00 for success, $FF for failure
'-----
' de-assert chip select
OUTA [ spiCS ] := 1

' now clock the SD SPI interface a few times and let it initialize (send dummy data)
repeat index from 0 to SD_POWERUP_LIMIT
  SPI_Send_Recv_Byte( $FF )

' assert chip select
OUTA [ spiCS ] := 0

sd_response := $FF
cmd_num_attempts := 0
```

```

repeat while (sd_response <> $01)
  ' send SD command GO_IDLE_STATE = $40, which resets the SD card
  sd_response := SD_Send_Command( SD_GO_IDLE_STATE, 0 )

  ' check if we have tried this long enough
  if (++cmd_num_attempts > SD_RESPONSE_LIMIT)
    ' test for verbose mode
    return ($FF)

  ' test for verbose mode
  sd_response := $FF
  cmd_num_attempts := 0
  repeat while (sd_response <> $00)

    ' send SD command EXIT_IDLE_STATE = $41, which resets the SD card
    sd_response := SD_Send_Command( SD_EXIT_IDLE_STATE, 0 )

    ' check if we have tried this long enough
    if (++cmd_num_attempts > SD_RESPONSE_LIMIT)
      ' test for verbose mode
      return ($FF)

  ' send SD command APP_CMD = $40 + $37
  sd_response := SD_Send_Command( SD_APP_CMD, 0 )

  if ( sd_response <> $00 )
    return($FF)

  ' send SD command SEND_OP_CMD = $40 + $29
  sd_response := SD_Send_Command( SD_SEND_OP_CMD, 0 )

  if ( sd_response <> $00 )
    return($FF)

  ' return success
  return($00)
' end SD_Mount

```

The code is more or less straightforward, but a couple comments are in order. First, notice that when the function is entered the chip select line is de-asserted and the SD SPI interface is clocked a number of times. This clocking while de-asserted is necessary to initialize the SD card's internal electronics. While doing this you must send \$FF to the SPI bus. After the card is initialized then the chip select line is asserted (LOW) and the initialization process begins with the command to enter the idle state. This is followed by exit the idle state. At this point, we could use the SD card in SPI mode, but as mentioned, we want to make sure this is an SD card and not MMC card, so the commands **APP_CMD** and **SEND_APP_OP_CMD** are sent. When they return \$00 the card is initialized and definitely an SD card.

5.3.2 Reading a Sector

All SD cards are arranged as an array of **sectors**. Each sector is typically 512 bytes and each is addressed with a 32-bit address. Reading a sector from the SD card requires that the caller issue a read sector command to the SD card, then when the SD card is ready to reply, the caller reads the bytes back from the SD card. Reviewing the SD card command list in Table 6.0, we see that there are two potential commands that might serve this purpose:

- CMD17 – Read single block.
- CMD18 – Read multiple blocks.

We are going to use the **read single block** command since its easier to deal with. The command code for it is \$51. The format of the command is (6 bytes): \$51, 32-bit sector address, CRC byte. The CRC is not important, so we can send anything. But, the \$51 and the 32-bit sector address is important. Furthermore, if you refer to Table 7.0 you will see that the response and data token bytes from the read single block command are \$00 and \$FE. This means that after you send the SD command to the SD card, the SD card must respond back with a \$00 followed by a \$FE. Once this occurs, then you simple enter a loop and read 512 bytes from the SPI bus which is the 512 byte sector data (sectors are called **blocks** in SD terminology).

The next step after reading the data is to read the 16-bit CRC which is sent by the SD card for the 512-byte sector. You can review it if you wish and compare it against the 16-bit CRC of the data read, but its not needed. Therefore, we are just going to read 2-bytes and throw them away.

Finally, you must deselect the SPI chip select line and clock the SD's SPI bus 8 times with HIGH, that is, send \$FF. This allows the SD card to complete the operation internally. In review, the steps to reading a sector are:

Step 1: HYDRA enables chip select on the SD SPI bus, and send the command: **READ_SINGLE_BLOCK** (\$51), 4-byte sector address, 1-byte CRC.

Step 2: SD Card Returns Response Codes \$00, \$FE.

Step 3: HYDRA reads 512 bytes from the SPI bus.

Step 4: SD card sends final 16-bit CRC of 512 data bytes back to HYDRA.

Step 5: HYDRA disables chip select on SPI bus and performs an 8-bit dummy write to the SPI bus with value \$FF.

NOTE

During steps 1 and 5 the SD card is selected and deselected. I have found that you can keep the SD selected at all times and don't have to de-select it to perform the dummy \$FF writes. However, if you wish you can add this code change to your functions.

Here's a function to perform the task of reading a sector:

```
PUB SD_Read_Sector(sector32, sectorbuffer16) | response, index
' DESCRIPTION: this function reads a sector from the SD card into the sent sector buffer
'
' INPUTS: sector32      - 32-bit absolute sector number to read
'         sectorbuffer - 16-bit pointer to receive buffer for sector
'
' OUTPUTS: storage pointed to by sectorbuffer16 will be filled with byte data from sector
'         returns $00 if no errors, $FF if error
' -----
' send the read block command with the byte address computed by multiplying the
' requested sector by 512 (512 bytes per sector)
response := SD_Send_Command( SD_READ_SINGLE_BLOCK , sector32 * SECTOR_SIZE )
' test response, should be $00
if (response <> $00)
    return (response)
' wait for the SD card to fetch the data from the flash storage
repeat index from 0 to SD_RETRIEVE_SECTOR_LIMIT
    response := SPI_Send_Recv_Byte($FF)
' test for proper response code $FE, break out of loop
if (response == $FE)
    quit
' // end repeat index loop
' return error if not responding
if (response <> $FE)
    return ($FF) ' there was an error
' now, read the data bytes into the sent buffer
repeat index from 0 to 511
    byte[sectorbuffer16][index] := SPI_Send_Recv_Byte($FF)
' next read in dummy data
SPI_Send_Recv_Byte($FF) ' ignore this data
SPI_Send_Recv_Byte($FF) ' ignore CRC value
SPI_Send_Recv_Byte($FF) ' finally 8 more clocks for the card to finish
' return $00, no errors code
return ($00)
' end SD_Read_Sector
```

5.3.3 Writing a Sector

Writing a sector is nearly identical to reading a sector. But, instead of issuing a **READ_SINGLE_BLOCK** command to the SD card, we must issue **CMD24** which is **WRITE_BLOCK** (\$58). There is also a **write multiple block** command, but we aren't supporting it. The **WRITE_BLOCK** command takes the 32-bit sector address of the sector/block you want to write followed by the 1-byte CRC (the CRC is not used, so it can be anything). Thus, the write block command is 6-bytes long and looks like:

\$58 (1-byte), 32-bit address (4-bytes), CRC (1-byte)

And remember the 32-bit write sector address must be **Big Endian**, that is, **most** significant byte to **least** significant byte.

After sending the **WRITE_BLOCK** command, the SD card needs a moment to prepare for the write action, so you must poll for a response code of \$00, once the SD card returns \$00, then you can move onto the next step which is clocking the SD 8 more times with dummy data \$FF followed by a data token of \$FE. At this time, the 512 bytes can finally be written.

After the data bytes are written to the SD card they actually are held in a temporary buffer internally then the real write process takes place which can take a considerable amount of time. During this time, we have to poll the SD card and wait for it to complete the flash write operation. In the code you will see a call to **SD_Wait_Write_Complete** this function is rather complex, so we will take a look after the code for the write sector which is listed below:

```
PUB SD_Write_Sector(sector32, sectorbuffer16) | response, index
' DESCRIPTION: send the write block command with the byte address computed by multiplying the
' requested sector by 512 (512 bytes per sector). Be CAREFUL, you can destroy the MBR etc. by
' writing and some SD cards can stop functioning, so know what you're doing if you use the write
' operation.
'
' INPUTS: sector32 - sector to write
'         sectorbuffer16 - pointer to sector to write (512 bytes)
'
' OUTPUTS: returns response code from write operation, $00 no error, $FF error
'-----
response := SD_Send_Command( SD_WRITE_BLOCK , sector32 * SECTOR_SIZE )
' test response, should be $00
if (response <> $00)
    return (response)
' send $FF and $FE
SPI_Send_Recv_Byte($FF)
SPI_Send_Recv_Byte($FE)
' now, write the bytes to card from buffer
repeat index from 0 to 511
    SPI_Send_Recv_Byte(byte[sectorbuffer16][index])
' next read in dummy data
SPI_Send_Recv_Byte($FF) ' ignore this data
SPI_Send_Recv_Byte($FF) ' ignore CRC value
' wait for write to complete
response := SD_Wait_Write_Complete
SPI_Send_Recv_Byte($FF) ' finally 8 more clocks for the card to finish
' return response from wait call, $00 = no error, $FF = error
return (response)
' end SD_Write_Sector
```

The **SD_Write_Sector** function call returns from the **SD_Wait_Write_Complete** then clocks out a \$FF allowing the SD to finish up any last minute processing and returns. The **SD_Wait_Write_Complete** function was broken out separately since it's a bit contrived and didn't quite fit in the clean code from **SD_Write_Sector**. In any event, let's take a look at the code first then cover its internals:

```
PUB SD_Wait_Write_Complete | num_attempts, response
' DESCRIPTION: This function is called after a write operation to "wait" for the write to complete. As
' with any flash technology the write operation can be many times slower than read, thus you must wait for
' it to complete. The function waits SD_WRITE_SECTOR_LIMIT iterations by checking the status response from
' the SD card, if the proper response is given, then its assumed there was a failure.
'
' INPUTS: none, a previous sector write is assumed
'
' OUTPUTS: returns $00 for success, $FF for failure
'-----
num_attempts := 0
repeat while (num_attempts++ < SD_WRITE_SECTOR_LIMIT )
    ' wait for response
    response := SPI_Send_Recv_Byte($FF)
    ' looking for $05 for success, $0B or $0D problem
    if (response == $05)
        quit
    elseif( response == $0B or response == $0D)
```

```

return $FF

' test for took too long
if ( num_attempts => SD_WRITE_SECTOR_LIMIT)
return $FF

' finally wait for first non-$00 response, then done
num_attempts := 0
repeat while( (num_attempts++ < SD_WRITE_SECTOR_LIMIT) and (SPI_Send_Recv_Byte($FF) == $00) )

' did it take too long to get a response
if ( num_attempts < SD_WRITE_SECTOR_LIMIT)
return $00
else
return $FF

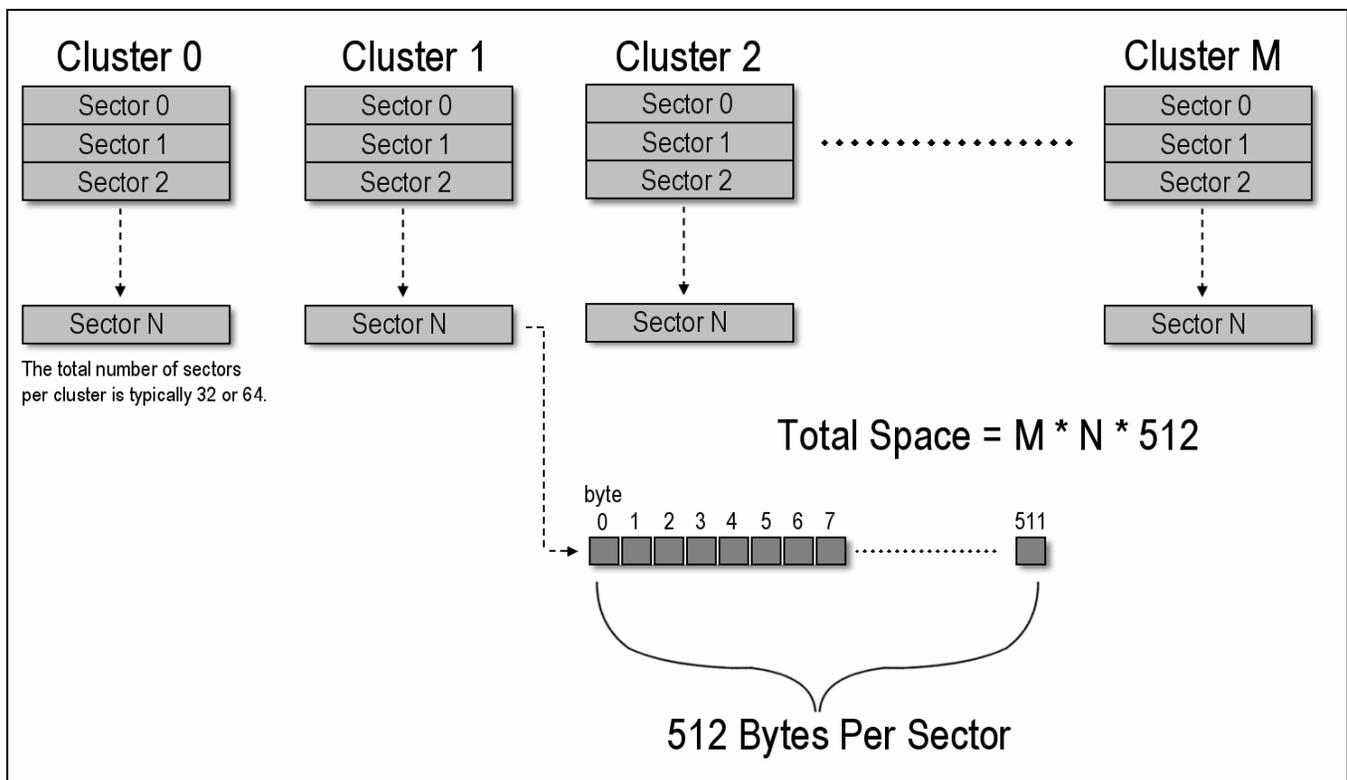
' end SD_wait_Write_Complete

```

The function has two main parts; the initial wait for a successful response code from the SD card indicating that the write sector operation was a success, and secondly the function has to wait for the SD card to get back to “neutral” or an idle state which is indicated by a \$00 response code. On top of all this, the function needs to make a certain number of attempts in both cases to give the SD card time, but at some point has to give up. This is what the loops are for and the **SD_WRITE_SECTOR_LIMIT** constant which is the number of iterations or attempts to give the SD card to finish.

This completes the main operations that can be performed with the SD protocol that we need to be concerned about. Of course, you can implement every single operation that is supported in the SD SPI protocol subset, but that would be overkill for our needs. Next up is the dreaded FAT16 file system!

Figure 15.0 – Clusters, sectors, and bytes.



5.4 FAT16 File System Overview

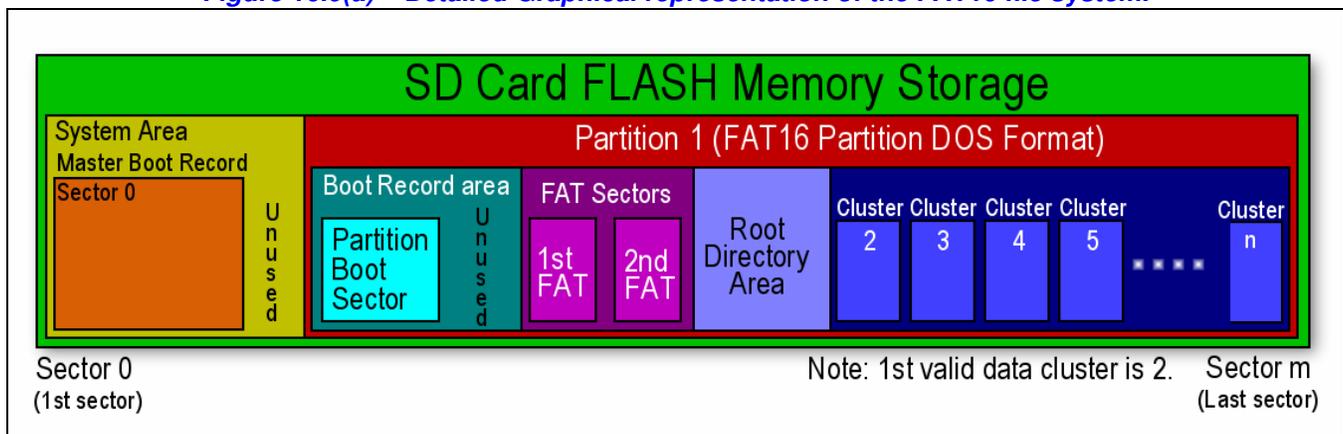
The **FAT** file system was invented by **Microsoft** in the early 70's for the **DOS operating system**. Of course, “invented” is a strong word since the FAT file system is very similar to many file systems that were already in place on CPM machines as well and UNIX. In any event, the first FAT system was FAT12 which was designed to support floppy disks since hard drives were unheard of in 1977 (available, but extremely expensive). The “12” in FAT 12 has to do with the number of “**clusters**” that can be addressed in the file system. Each cluster contains a number of sectors, each sector contains a number of bytes as shown in Figure 15.0. Therefore, with FAT12 - 2^{12} clusters could be addressed for a total of 4096.

Then each cluster would contain one or more sectors. Of course, once hard drives became common place the storage capacity of FAT12 was woefully inadequate. Microsoft's response to this was the **FAT16** file system released in **1984** and finalized in **1987**. FAT16 had more features, the most important of which was a **16-bit cluster address**. This increased the cluster range to **65536**. Along with this support for up to 64 sectors per cluster and 512 bytes per sector raised the maximum stored to 2 Gigabytes.

Inevitably, FAT16 was once again upgraded to **FAT32** in **1996** which used a **28-bit address** for the number of clusters, this increased the theoretical address space to 8 Terabytes with 32K clusters. However, the FAT16 file system is what the majority of SD cards use (although you can format them any way you like). Nonetheless, FAT16 supports SD cards up to 2 Gigabytes which is more than enough for any embedded system. In the following sections we are going to cover the basics of the FAT16 file system, at least enough so you can implement rudimentary file I/O and SD card access.

Writing a complete FAT16 file system with support for every single feature is a rather large undertaking and not necessary. Considering that, the following sections outline FAT16, refer to Figure 16.0 below as you read the sections for a graphical map of what's going on since it gets a little confusing. Also, from here on when "FAT" is referred to, its understood that we mean "FAT16" unless otherwise noted. Thus, make sure that you always format your SD card as FAT16 not FAT32 when using it with the API developed here. You can develop your own FAT32 support if you wish later.

Figure 16.0(a) – Detailed Graphical representation of the FAT16 file system.



5.4.1 FAT16 SD Card Disk Structure

The first sector on an SD card formatted with the FAT16 system is called the **Master Boot Record (MBR)**. The MBR contains information about different logical subdivisions on the SD card, called **partitions**. Each of these partitions can be formatted with a unique file system. As noted above, usually an SD card only has **one** active partition (which is not bootable usually), which is comprised of the following parts:

- Boot Sector
- FAT Regions
- Root Directory Region
- Data Region

Referring to Figure 16.0(a), the **boot sector** is the first sector of the partition and contains basic information about the file system type. The FAT region is actually a map of the card, indicating how the clusters are allocated in the data region (clusters make up files). Generally, there are two copies of the FAT in the FAT region, to provide redundancy in case of data corruption. However, most SD card systems do not keep the 2nd copy up to date. The root directory region follows the FAT region and is composed of a directory table that contains an entry for every directory and file on the card. Collectively, the first three sections are called the **system area**. The remaining space is the **data region**.

Data stored in this region remains intact, even if it is deleted, until it is overwritten by new data. This is because a file is a linked list of clusters. When a file is deleted only the linked list is deleted, but the actual clusters are still there. As mentioned before, the FAT16 system uses 16-bit FAT entries, allowing approximately 65,536 (2^{16}) clusters to be represented. A signed byte in the boot sector defines the number of sectors per cluster for a disk. This byte has a range of -128 to 127. Thus, the only usable values in the FAT16 file system are positive, power-of-two values (1, 2, 4, 8, 16, 32, 64), so 64 is the largest sector per cluster value. This means that with the standard 512-byte sector size, the FAT16 file system can support a maximum of:

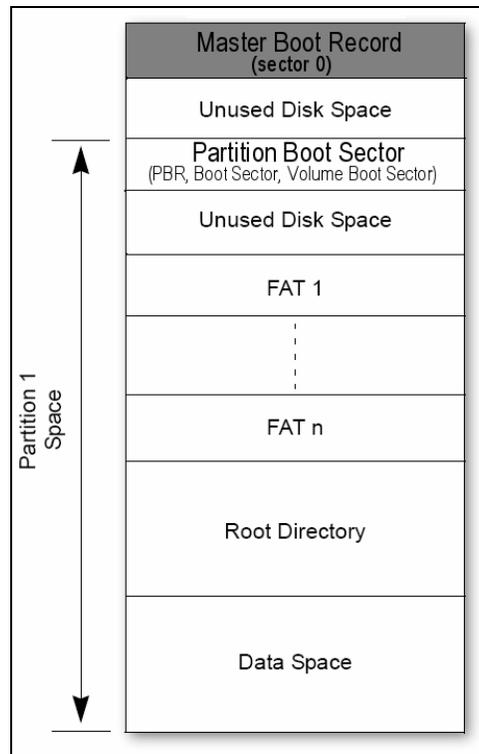
65536 clusters * 64 sectors per cluster * 512 bytes per sector = 2 GB of disk space.

There are 4GB SD cards that support FAT16 with 64K clusters, but not common. Usually cluster size is 32K which is 64 sectors per cluster.

Now, let's drill down into each of the components making up the various area of the SD card/disk starting with the Master Boot Record or MBR.

NOTE All FAT16 data is always in Little Endian format. So values encoded in 2-bytes, are formatted low byte, high byte. Values that are 4-bytes are encoded low byte...high byte.

Figure 16.0(b) – Simplified representation of the FAT16 file system.



5.4.2 Master Boot Record

The MBR (as shown in Figure 16.0(b)) contains information that is used to boot the card, as well as information about the partitions on the card. The information in the master boot record is programmed when the card is manufactured, and any attempt to write to the MBR could render the disk unusable. The contents of the MBR are listed in Table 8.0 below.

WARNING! Never write the MBR unless you absolutely know what you're doing. Reading it is perfectly safe, but if you write to it and make a mistake the on-board SD card controller might take action like moving it to another location, assuming its been damaged.

Table 8.0 – The contents of the Master Boot Record (MBR).

Offset	Description	Size in bytes
\$000	Boot Code (machine code and data)	446
\$1BE	Partition Entry 1 (active)	16
\$1CE	Partition Entry 2	16
\$1DE	Partition Entry 3	16
\$1EE	Partition Entry 4	16
\$1FE	Boot Signature Code (0x55AA)	2

The first **446 bytes** of the MBR are quite interesting. They actually can hold **executable code** that computer will load and execute if the SD card (or disk) is tagged as bootable. In our case, this will never happen. But, on floppies or hard drives, this 446 byte program is what **“boots”** the computer, that’s where the name **“Master Boot Record”** originates from.

The next 4 entries are 16 bytes each and each represent a **Partition Entry**. This is not to be confused with a **disk partition**, which is physical space. A partition entry is a descriptor of this space. Typically, on SD cards only one partition is active (the first), since SD cards are typically not partitioned into multiple drives.

Finally, the MBR ends with a special 2-byte code called the **“Boot Signature Code”** (\$55AA). On drives that have the MBR at a different location than sector 0 (which is possible), the only way to find it, is to scan for the code \$55AA until its found, then you know you have found the MBR (or at least one of them).

5.4.3 Partition Entries in the MBR

Information about each partition on the SD card/disk is contained in the respective 16-byte **partition table entry** of the **master boot record**. A **file system descriptor** is included in each entry to indicate which type of file system was specified when the partition was formatted. The following file descriptor values indicate FAT16 formatting:

- \$04 (16-bit FAT, < 32M) – very rare since its hard to find a 32MB SD card these days.
- \$06 (16-bit FAT, > 32M) – most common case.
- \$0E (DOS CHS mapped) – Cylinder, head, sector, not relevant for SD cards.

Therefore, its always good to verify what kind of file system the partition was formatted with by reviewing the file system descriptor value. In general, SD cards contain a single active partition. The contents of a **partition table entry** are listed in Table 9.0 below.

Table 9.0 – Partition Table Entry.

Offset	Description	Size in bytes
\$00	Boot Descriptor (\$80 if active partition, \$00 if inactive)	1
\$01	First Partition Sector (head, cylinder, sector format?)	3
\$04	File System Descriptor	1
\$05	Last Partition Sector (head, cylinder, sector format?)	3
\$08	Number of sectors between the MBR and the first sector of the partition	4
\$0C	Number of sectors in the partition	4

Reviewing the elements in the partition entry, you see that it starts off with the boot description and is followed with values that help locate the actual partition data itself. All values are 0-based. For example, **first partition sector** means the 0-based first sector that the partition referred to by this entry starts at; however, this value is in head, cylinder, sector format which might not be linear. Thus, there is a backup value at \$08 which is the absolute number of sectors between the MBR and the first partition data sector. Typically, the 3-byte value at \$08 is used to locate the first sector. Additionally, the entry at \$05 has the same potential problem, it might encoded as head, cylinder, sector. Therefore, SD cards will usually look at the last entry \$0C which is an absolute number that indicates the real number of sectors in the partition. As you can see, mapping the SD card to the FAT16 system can be a little confusing since these data structures were obviously intended for physical disk drives which have a different structure. So mapping the solid state SD cards to FAT16 you have to be careful when interpreting and decoding the FAT16 data structures sometimes.

Once you have interrogated the partition entry and located the first sector of the partition data then you can start reading in the partition itself. The first sector of the partition is called the **“Boot Record”** or **“Partition Boot Record”** or **“Volume Boot Record”**, none of which you should confuse with the **Master Boot Record** (sector 0). The way to think about the partition boot record is that it’s the boot record solely for that particular partition, whereas the Master Boot Record is the boot record for the **whole disk**. With that in mind, let’s look at what the partition boot record looks like.

5.4.4 Partition Boot Record (PBR)

The **“Partition Boot Record”** is also know as the **“Volume Boot Record”**, **“Boot Record”**, **“Boot Sector”**, and other variants. The main idea is that this record (sector) is the boot information for the particular partition **not** the disk/SD drive . Thus, a disk with multiple partitions will have multiple partition boot records, but only **one** Master Boot Record. That said, the partition boot record contains everything you need to know about the partition itself. Many of the entries are for

physical disk drives and have no meaning for a solid state SD card, but some of the more important entries are bytes per sector, sectors per cluster, total number of sectors, volume label, and so on. Table 10.0 below lists the format of the data structure and descriptions of each entry. And remember, the data structure is a single sector of 512 bytes.

Table 10.0 – Partition Boot Record Format

Offset	Description	Size in bytes
\$00	Jump Command	3
\$03	OEM Name	8
\$0B	Bytes per Sector	2
\$0D	Sectors per Cluster	1
\$0E	Total Number of Reserved Sectors	2
\$10	Number of File Allocation Tables	1
\$11	Number of Root Directory Entries	2
\$13	Total Number of Sectors (bits 0-15 out of 48)	2
\$15	Media Descriptor	1
\$16	Number of Sectors per FAT	2
\$18	Sectors per Track	2
\$1A	Number of Heads	2
\$1C	Number of Hidden Sectors	4
\$20	Total Number of Sectors (bits 16-47 out of 48)	4
\$24	Physical Drive Number	1
\$25	Current Head	1
\$26	Boot Signature	1
\$27	Volume ID (Serial Number)	4
\$2B	Volume Label	11
\$36	File System Type (not for determination)	8
\$3E	Executable Code	448
\$1FE	Signature (\$55hAA)	2

There is a lot of interesting information in the **Partition Boot Record (PBR)**. To begin with at address \$00 you will notice this is where a jump command can be placed. This jump command allows the program counter to begin execution of boot code somewhere else on the disk (not relevant for SD cards). Additionally, more code space is down at \$3E where there is room for 448 bytes of executable code. Again, this is for bootable drives, so not relevant. The most important entries have to do with:

- Bytes per sector
- Sectors per cluster
- Number of reserved sectors
- Number of file allocation tables
- Number of root directory entries
- Total number of sectors

And that's about it. Near the end of the record there are some identifying entries such as the volume id, file system type, and lastly the boot signature \$55AA which is identical to the value in the MBR.

INTERESTING FACT

As you can see, the MBR and PBR are great places to put virus code! This is why a virus that infects the MBR or PBR is very lethal since the designed write very short viruses that can fit in the boot code and damage the system. Also, the viruses are very hard to remove since they are the first thing booted by the ROM BIOS!

You will notice that some of the information is redundant and is found in the partition table entry for the partition. This is by design since in many cases the FAT16 code API might have to navigate around just using the MBR's partition table entries and doesn't want to have to read in the complete partition boot record to figure out certain geometries of the disk. In any event, we absolutely must read this data structure in.

5.4.5 A Quick Recap and Locating the Primary FAT16 Data Structures

At this point, let's take moment to review where we are so far in the FAT16 file system hierarchy. The first step to accessing the FAT16 file system on the SD card is to load in **sector 0** which is the MBR or **Master Boot Record**. This record contains boot code for bootable drives that is ignored, but more importantly it contains **4 partition table** entries that

are **16-bytes** each that each describe the whereabouts of **4 drive partitions** and some details about each. In most cases, we only need to review **partition 1** and then locate it. To locate the actual sector of the partition you would load in **sector 0**, the MBR, then starting at location **\$1BE** are the **16-bytes** that make up the first partition boot record. So you interrogate the data and look at the data at the byte offset **\$08** within the 16-byte record, this is the number of sectors between the Master Boot Record and the first sector of the partition and is 4 bytes long. Since the MBR is always at sector 0, you simply go and retrieve this sector and load it into your 512 byte working buffer. Now, we have the partition boot record as outlined in Table 10.0 From this record you would extract all the information you need and place them into variables.

At this point you have enough information to locate any data structure on the SD card (or disk). Of course, we still haven't discussed how files are stored, or the FAT tables themselves or the directory of files -- in good time. But, to get there we need to know how to find this data. So at this point, we have to do a little work to find the three important things:

- The starting FAT table sector. This is where the actual file allocation tables are that describe the linked lists of files.
- The root directory first sector. This is the actually root directory of the SD card that has the file list in it and pointers back to the FAT where each starts.
- The starting sector of the file storage itself. This is where the actual bytes of the files are stored. They are stored in large collections of sectors (32-64) called clusters, therefore the smallest unit of file storage is actually a cluster.

In the next sections we will see how each of these data areas is traversed and how they work, but first we need to figure out how to find them. Please refer back to Figure 16.0 for a complete graphic of the relationships between each data area, so its easy to follow the formulas that follow.

The **first** data structure that has to be located is the **Partition Boot Record** itself. The **Master Boot Record** and the **partition entries** themselves are all in **sector 0**, but the partition boot records themselves have to be located. As explained in the paragraphs above you need to use the “**number of sectors between the Master Boot Record and the first sector of the partition boot record**” entry at **\$08** to find the sector number of the partition boot record. Then you load all data in and parse apart each of the entries in Table 10.0 above. Then the first sector of the FAT, root directory, and file data storage area can be found with these formulas:

```
first_fat_sector := mbr_num_sect_to_part + pbr_num_reserved_sect
first_dir_sector := mbr_num_sect_to_part + pbr_num_reserved_sect + pbr_num_fats*pbr_num_sect_per_fat
first_data_sector := mbr_num_sect_to_part + pbr_num_reserved_sect +
                    pbr_num_fats*pbr_num_sect_per_fat + (pbr_num_root_dir_entries / 16)
```

Note: The 3rd formula is on two lines. Also, “mbr” stands for Master Boot Record, “pbr” stands for Partition Boot Record.

Let's take a look at these formulas and see if they make sense. For example, to locate the first FAT sector, referring to Figure 16.0(a,b), we see that it comes after the Partition Boot Record, but there is some “**reserved space**” between the two. This space is referred to by **pbr_num_reserved_sect** and if the number of sectors reserved after the PBR, but before the FATs. Thus, to get to the FAT's first sector we need to add this value to the initial sector number of the PBR itself. The remaining formulas work in a similar fashion and simply use all the various constants in the MBR, and PBR to locate the first sectors of the root directory and the file storage data area respectively. Now, that we can find the FATs and the root directory, the next step is to look at the format of the root directory since this is where all the file names are stored for the SD card/disk.

5.4.6 The Root Directory

The Root Directory, located after the FAT region on the SD card/disk, is a table that stores file and directory information in 32-byte entries. Each entry includes the file name, file size, the first cluster of the file and the time the file was created and/or modified. This information is used to index into the FAT to traverse the cluster chain for the file and ultimately access the sectors of the file in the data area.

NOTE

Generally, each file entry conforms to the 8.3 short file name format used in DOS. Only digits 0 to 9, letters A to Z, the space character and special characters, ! # \$ % & () - @ ^ _ ' { } ~ ` , are used. Although it is customary to consider the period (.) and extension as elements of the file name, in this case, none of the characters after the initial name are used as part of the actual file name in the directory entry. For example, a file named “FOO.TXT” would have the file name “FOO_____” in the root directory, with the final 5 characters replaced by 5 instances of the space character \$20. Then the extension would store “TXT”. Therefore, when scanning for file names, care has to be taken in the string compare functions.

The **Root Directory** typically contains up to **512** entries which means that on the “root” of the SD card there can be 512 files or directories, no more. Of course, each directory can have files in it as well. But, if you perform a “DIR” on the root, there can be no more than 512 entries. This is important since the FAT16 API developed for the SD Max only support root directory entries and no sub-directories. Of course, you can add these features yourself. But, for embedded systems you can get away with all the files in the root directory along with 8.3 filenames. Table 11.0 lists the structure of each 32-byte directory entry.

Table 11.0 –Root Directory Entry Structure.

Offset	Description	Size in bytes
\$00	File Name	8
\$08	File Extension	3
\$0B	File Attributes	1
	Format: (msb) 7 6 5 4 3 2 1 0 (lsb) 0 0 A R S H D V	
	0 – Unused bit. A – Archive bit. R – Read only bit. S – System bit. D – Directory bit. V – Volume bit.	
\$0C	Reserved	1
\$0D	File Creation Time (ms portion)	1
\$0E	File Creation Time (hours, minutes and seconds)	2
\$10	File Creation Date (year, month, day)	2
\$12	Last Access Date (year, month, day)	2
\$14	Extended Address-Index	2
\$16	Last Update Time (hours, minutes and seconds)	2
\$18	Last Update Date (year, month, day)	2
\$1A	First Cluster of the File	2
\$1C	File Size	4

Each entry contains everything you could want to know about a file including; name, size, creation and modification times and dates, along the file’s starting cluster location on the disk. There are some important details about each entry, so let’s briefly touch on each entry.

Filenames

The filename is in 8.3 format, but as its stored in the entry there are 8 bytes for the filename and 3 bytes for the extension. There is no dot “.” stored in the actual filename. Moreover, file name are always stored as **left-justified**, padded with spaces to the right.

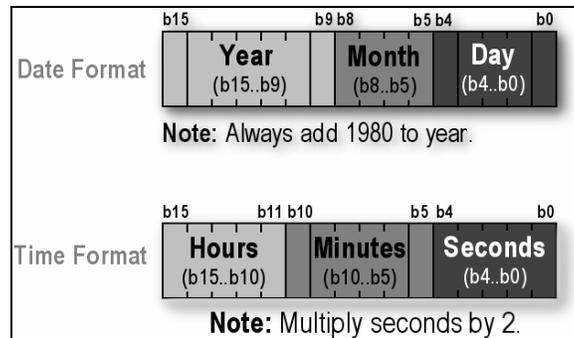
When a file is **deleted** then **only** the first character of the file is modified to indicate this, the file clusters are not harmed. Alas, this is why you can undelete files. Until something actually overwrites the clusters of a file its intact and only a single character has been changed in the filename portion of the entry to indicate its been deleted. Also, **empty** entries that have never been used are indicated by the first character as well. Thus, the first byte of the directory entry is very telling. Table 12.0 below lists the values that the first character of a filename which is also the first byte of each directory entry.

Table 12.0 – Directory file entry first character codes.

Value	Description
\$00	This entry is available and no subsequent entry is in use. Directory scanning can be terminated.
\$E5	The file in this entry was deleted and the entry is available. The data clusters are still intact.
\$05	The first character in the file name is \$E5h. This is a hack due to the \$E5 code above.
\$2E	This entry points to the current or previous directory.

For our purposes we more or less just want to check for \$00 (empty/last directory entry), and \$E5. The value \$2E shouldn't be there since we are assuming no directories, and \$05 is only needed when the filename starts with \$E5 which is decimal 229 an extended ASCII angstrom character \AA that will rarely be in a filename.

Figure 17.0 – Time and date formats.



Time and Date Formats

The file creation dates are in the format year/month/day where the bit encoding is 7 bits for year, 4 bits for month, and 5 bits for day. As shown in Figure 17.0. Also, the year is relative to 1980, so you always need to add 1980 to the year. The SPIN code snippet below shows how to extract the year, month, and day from the 16-bit value, in this example the creation date:

```
year      := 1980 + (dir_create_date_ymd & %1111_1110_0000_0000) >> 9
month     := (dir_create_date_ymd & %0000_0001_1110_0000) >> 5
day       := (dir_create_date_ymd & %0000_0000_0001_1111)
```

The time stamps are in the format hours/minute/second where the bit encoding is 5 bits for hours, 6 bits for minutes, and 5 bits for seconds. Assuming you have extracted the creation time into a 16-bit word then here's a SPIN snippet that would decode it:

```
hours     := (dir_create_hms & %1111_1000_0000_0000) >> 11
minutes   := (dir_create_hms & %0000_0111_1110_0000) >> 5
seconds   := (dir_create_hms & %0000_0000_0001_1111) << 1 ' resolution 2 seconds, so
multiply by 2
```

Notice that the **seconds** fields only has 5 bits of resolution, thus not enough to resolve 60 seconds, so you must multiply it by 2.

File Attributes

The file attributes byte helps decipher what kind of file entry is at the current location. Referring to the embedded table in Table 12.0, there are bits for read-only, archive, volume, directory, system, and so forth. Typically, the **attribute** byte should be **00000000b** or maybe the read-only bit is set **00100000b**. Also, there will be a single disk volume label entry which isn't really a file, but gives the name of the partition. In this case, the attribute will have the **volume** bit 0 set like so, **0000001b**. And then the filename and extension are concatenated together to form one continuous 11 byte volume name like "C_DRIVE" or whatever you name it.

First Cluster

This is a 16-bit value that indicates the absolute first cluster of a file. This is always relative to the beginning of the file data area. For example, say a file starts at cluster 10 and each cluster has 32 sectors. Moreover, assume that the first sector of the file data area is 10,000. Therefore, the first sector of the file would be:

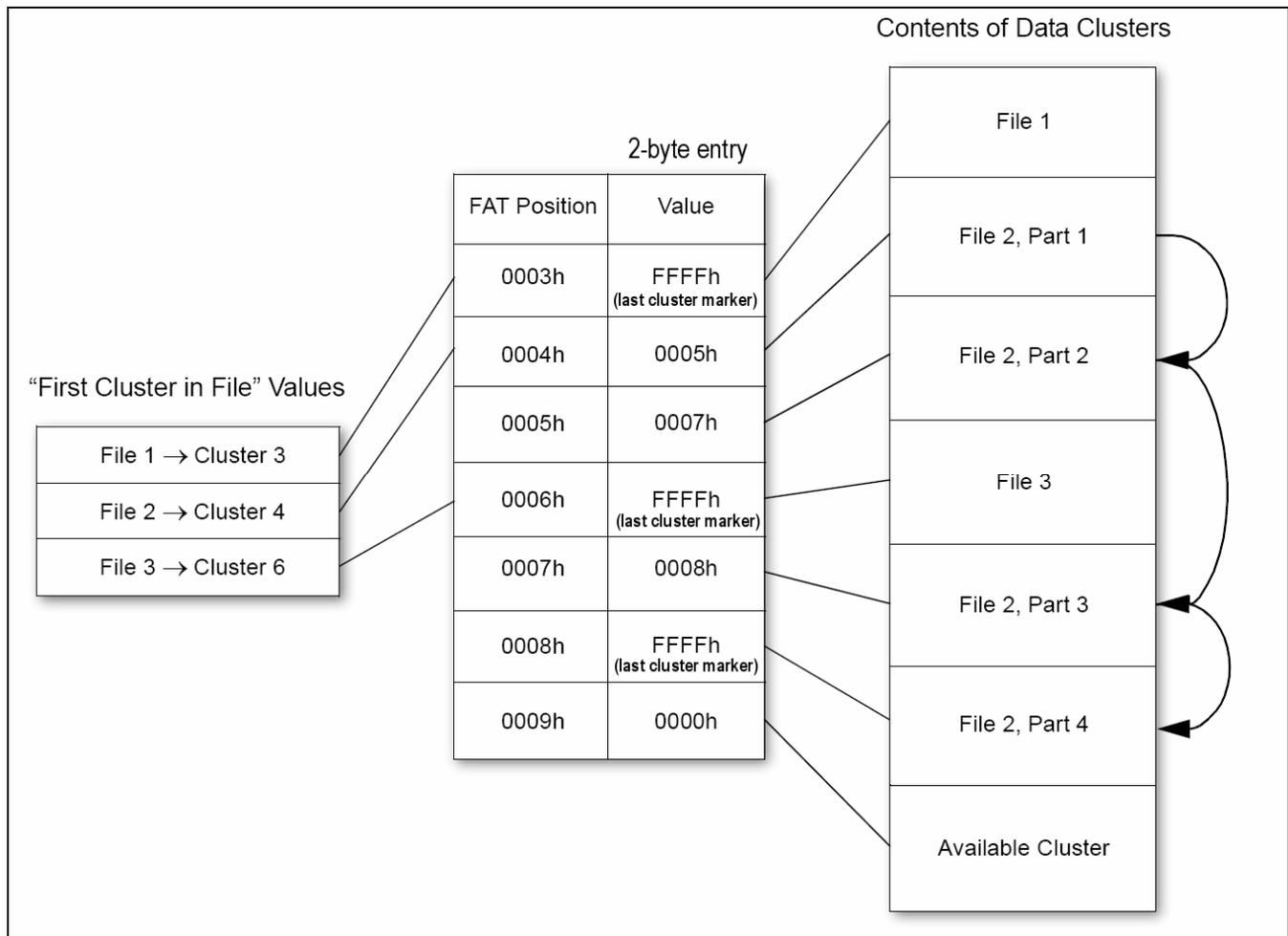
$10,000 + 10 * 32 = 10,320$ is first sector of file

Of course, we still have to figure out exactly where the first sector of the data area is, but we are getting there.

File Size

File size is simply a 4-byte, 32-bit value that indicates how large the file is 0 to 2GB. This is important since as a file is traversed in the FAT to seek each of its clusters, we know that we are done with the file when we have read in the number of clusters that equal the total file size of the file. Where the total number of bytes of a cluster is equal to the number of sectors per cluster multiplied by the bytes per sector (512 in 99% of cases).

Figure 18.0 – An illustration of the FAT cluster chain.



5.4.7 The File Allocation Table

Here's where things get interesting. The file allocation table or FAT is the map of every file on the SD card/disk. There can be more than one FAT, but they are for redundancy and rarely kept up on SD cards. The FAT is one of the most confusing things about the FAT16 file system since its an indirect data structure and a linked list in an array. Theoretically, its not hard to understand, but when it comes time to implement FAT access, many people get caught up with the details, lots of "off by one errors" and so forth; therefore, its important to draw a lot of pictures.

With that in mind, the FAT is organized as an array of 2-byte entries as shown in the center of Figure 18.0. Where each entry correspond to every cluster in the data cluster section of the partition. For example, the third set of two bytes in the FAT will correspond to the first cluster in the data region (the first two cluster entries are reserved). The FAT itself can be many sectors long, so you must traverse the cluster chains over sector boundaries as well which can make the algorithms messy. The values placed in each FAT cluster entry indicate the next cluster of the file, or end of cluster list, along with

other codes. For example, is this the last cluster in the chain? Is the file deleted and so forth. Table 13.0 below lists the possible encodings for cluster values.

Table 13.0 – FAT cluster entry values.

Value Codes	Description
\$0000	Cluster is available for use.
\$0001	Cluster is reserved.
\$0002-FFEF	Points to next cluster in the file.
\$FFF0-FFF6	Cluster is reserved.
\$FFF7	Cluster is bad.
\$FFF8-FFFF	Last cluster of a file.

Note: Values \$0002 – FFEF are literal cluster indices, while the other possible values are “codes”.

Referring to Table 13.0, some of the entries are useful when writing a file, some are useful when reading a file. For example, when reading a file, the directory entry is going to point you to the **first** cluster in the file. You would locate that cluster’s sectors in the data area and load the data, then its time for the **next** cluster in the file. You would then look at the first cluster entry that the directory entry pointed to and see what the value is. If the value is \$FFF8-FFFF then the file must fit into a single cluster and be smaller than 32-64K since that value range means last cluster. Otherwise, the entry is a pointer or link to the next cluster than makes up the file. The value will be \$0002-FFEF. Then you simply load this value and move to that 2-byte cluster entry in the FAT, at which point you locate the cluster in the data region of the disk, load the sectors and continue the process.

There are a couple important points here. First, a cluster chain might not be linear, that is a file could have the cluster chain:

Format: Cluster entry index (value in cluster entry)

2 (3) → 3 (4) → 4 (5) → 5 (\$FFFF end of file, last cluster)

Which means there are 4 clusters all nice and linear, but files could be deleted, moved, resized, in which case its possible that the same 4 cluster chain might look like:

Format: Cluster entry index (value in cluster entry)

2 (100) → 100 (5) → 5 (3456) → 3456 (end of file, last cluster)

NOTE	The first valid data cluster starts with cluster 2. Clusters 0,1 are used by the file system and reserved. Thus, a file directory entry will never start with a cluster less than 2. The FAT entries corresponding to clusters 0 and 1 contain the media descriptor, followed by bytes containing the value, \$FF.
-------------	--

Now, writing a file presents a new set of challenges (and the drivers I developed, don’t support writing, I leave it to you to add). To write a file, first you have to find an open directory entry into the directory first byte (\$00), then you would build up the directory entry’s 32-bytes in the proper format. Then it comes time to actually store the data of the file on the SD card/disk in the data area. To do this, we need to traverse the cluster map in the FAT and find some available clusters. So the first thing we do is start at cluster entry 2 in the FAT table and check if its \$0000, if so then that will be the first cluster of our new file, else, we continue looking for available clusters. Once we find one, we then take that cluster number, call it c1, and locate the starting sector in the data area of the SD card/disk and write the first set of sectors that make up the first cluster of the file to that cluster. So far, so good. Next, we have to continue scanning for another available cluster, once we find it then we set cluster c1’s value to point to it, we then go write out the next set of sectors that this new cluster c2 point to, and the process continues. This way you more or less scan for available clusters and then write your file to the SD card/disk one cluster at a time, all the while stitching the cluster linked list together once entry at a time. At the last cluster you need for your file, you write \$FFFF into the location, write the cluster data itself and you’re done!

The example in Figure 18.0 shows a concrete example of a file on the SD card/disk. The “**First Cluster in File**” values are of course indicated by three entries in the root directory that indicate the start of these three files. The FAT

demonstrates the links between the files. File 1 and File 3 are smaller than the size of a cluster, so they are only assigned one cluster. The cluster entries in the FAT that correspond to these files contain only the End-Of-File value \$FFFF. File 2 is larger than three clusters, but smaller than four, so it is assigned four clusters. Since there were not three consecutive clusters available when File 2 was created, it was assigned non-consecutive clusters. This is called **"fragmentation"**. The values of the cluster entries in the FAT for File 2 point to the next cluster in the file. The last cluster entry in the FAT for File 2 contains the End-Of-File value \$FFFF.

TIP

It should be obvious why a fragmented drive becomes so slow; it has to potentially move the head all over the place to find clusters of the file. And moving the disk drive head can take milliseconds, do that a few thousand times then the file takes seconds to access!

Although, the SD Max demo API does **not** support directories, the following section outlines the concept. They aren't supported along with file writing support simply to try and keep the SPIN code API a manageable size since memory is at a premium. You may want to add support for writing and directories yourself.

5.4.8 Understanding Directories

Directories in the FAT16 file system, with the exception of the root directory, are written in the same way that files are written. Each directory occupies one or more clusters in the data section of the partition, and each has its own directory entry and chain of FAT entries. Bit **four (4)** of the **attribute** field in the directory entry of a directory is set, indicating that the entry belongs to a directory. Directory names follow the 8.3 format as do normal files. Directories differ from files in that they have **no extension**, though, thus each directory name is allowed 11 characters. Each sub-directory can contain as many files as it wishes. Two special directory entries, the dot entry "." and the dot dot entry "..", are present in every directory except the root directory. The dot entry is the first entry in any sub-directory. The name value in this entry is a single dot (\$2E) followed by ten space characters (\$20). The first-cluster-in-file value of this entry will point to the cluster that the entry is in. The dot dot entry is similar, except the name contains two dots followed by nine spaces, and the first-cluster-in-file value points to the directory that contains entry for the directory the dot dot entry is in (the previous directory). More or less, each sub-directory is represented by a single cluster of data that contains an embedded directory. This directory is of the same format as the root directory. Thus, once you traverse into a "directory", you read the directory entries in the cluster pointed to by the parent first-cluster-in-file, then you recursively access the file clusters pointed to by any of the directory entries. This can go on level after level as deep as you wish it to.

For more information on sub-directories and the FAT system in general here's a nice white paper by Microsoft located here on the CD:

CD_ROM:\docs\fat\MS_fat_white_paper.doc

5.4.9 Final Aspects of Navigating the FAT16 File System

The hardest part of the FAT16 file system is simple keeping track of all the different data structures. There is the Master Boot Record at sector 0 that contains the 4 possible Partition Entries. Each Partition Entry is 16-bytes long and indicates how to locate the first sector of each respective partition. Then within each partition is the Partition Boot Record (first sector), followed by some reserved sectors, then the FAT tables (1 or more copies), then the Root Directory, and finally by the actual file data sectors represented by blocks of clusters. Then files are found by inspecting the Root Directory entries, parsing them out, finding the first cluster of any file, then using that information to traverse the FAT table itself and follow the cluster list. The trick to FAT16 is taking your time, loading all the data structures, parsing out everything into variables, so you can work with it. Additionally, there are many ways to locate each data area; the FATs, root directory, files data area, but in general these formulas are what you want to use:

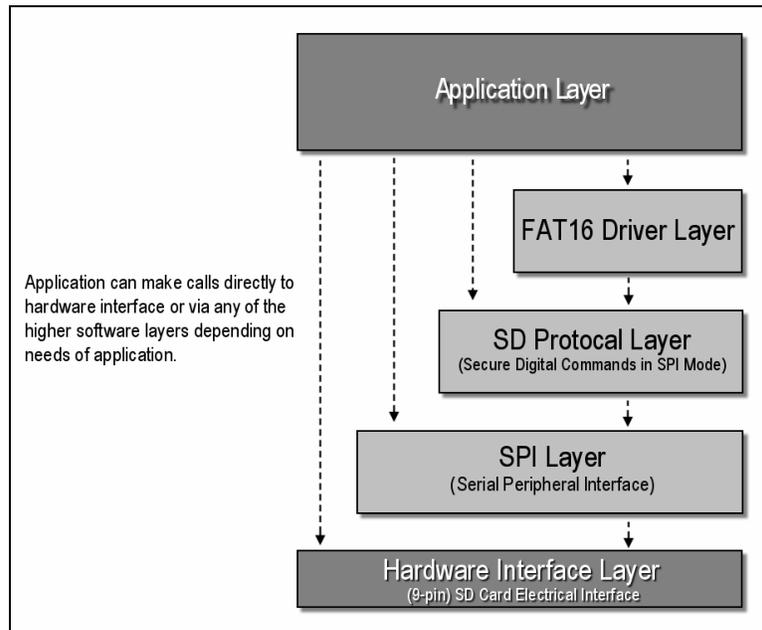
```

first_fat_sector := mbr_num_sect_to_part + pbr_num_reserved_sect
first_dir_sector := mbr_num_sect_to_part + pbr_num_reserved_sect +
pbr_num_fats*pbr_num_sect_per_fat
first_data_sector := mbr_num_sect_to_part + pbr_num_reserved_sect +
                    pbr_num_fats*pbr_num_sect_per_fat + (pbr_num_root_dir_entries / 16)

```

The SD Max card API that we are going to cover next was written 100% in SPIN, so its easy follow The API supports SPI communications, SD card protocol and finally a FAT16 layer that supports directory listings, and reading files. There is no write file support or sub-directory support, that's I leave to you.

Figure 19.0 – The overall system software model with SD Max Driver API.



6.0 Unleashing the SD Max Driver API

The SD Max driver is a starting point for you to begin experimenting with SD card communications and the FAT16 file system. Of course, no one said you have to use FAT16, you can use any file system you wish on the card. In fact, for some applications you might just write/read sectors directly with the SD card protocol and this is totally acceptable and beneficial in many cases. The overall driver architecture and its relationship to the application and the hardware is shown in Figure 19.0. As you can see, you make call to the driver and then are passed down thru the SPI interface to the SD card itself. The driver API has these main components:

- SPI Interface
- SD Card Functions
- FAT16 Support

The SPI is very simply and more or less consists of functions to read and write, that's about all there is to SPI. The SD card functions are much more complex and as outlined in the previous sections on the subject there are many, many functions SD cards supports, but the API here only supports reading and writing. You can always add more function calls. Finally, FAT16 is a rather large specification and I only supported basic functions like directory listing, reading files only, root directory support, and no sub-directories. The functions are well documented and hopefully you will optimize them, add, subtract and change them for you application.

Initially, I was just going to create a single application with the functions as part of the application that was a demo of them, but later I decided to put them into a separate object for you. The object's name is **SDMAX_DRV_001.SPIN** and is located on the CD here:

CD_ROM:\sources\SDMAX_DRV_001.SPIN

Additionally, there is a “**no debug**” version of the driver with less error printing support called **SDMAX_DRV_ND_001.SPIN** which is located on the CD here:

CD_ROM:\sources\SDMAX_DRV_ND_001.SPIN

The “no debug” version is simply pruned of its debug output conditions all over the place since SPIN has no support for conditional compilation, you are forced to sprinkle actual conditional compiled code to perform debug printing.

Figure 20.0 – The SD Max Driver model.

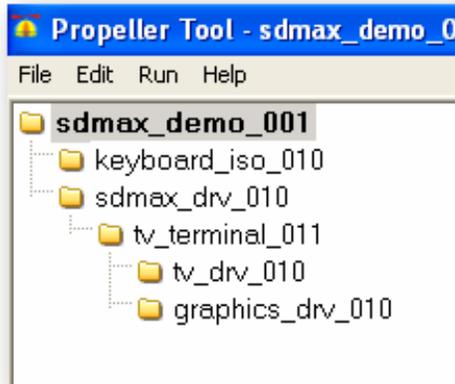
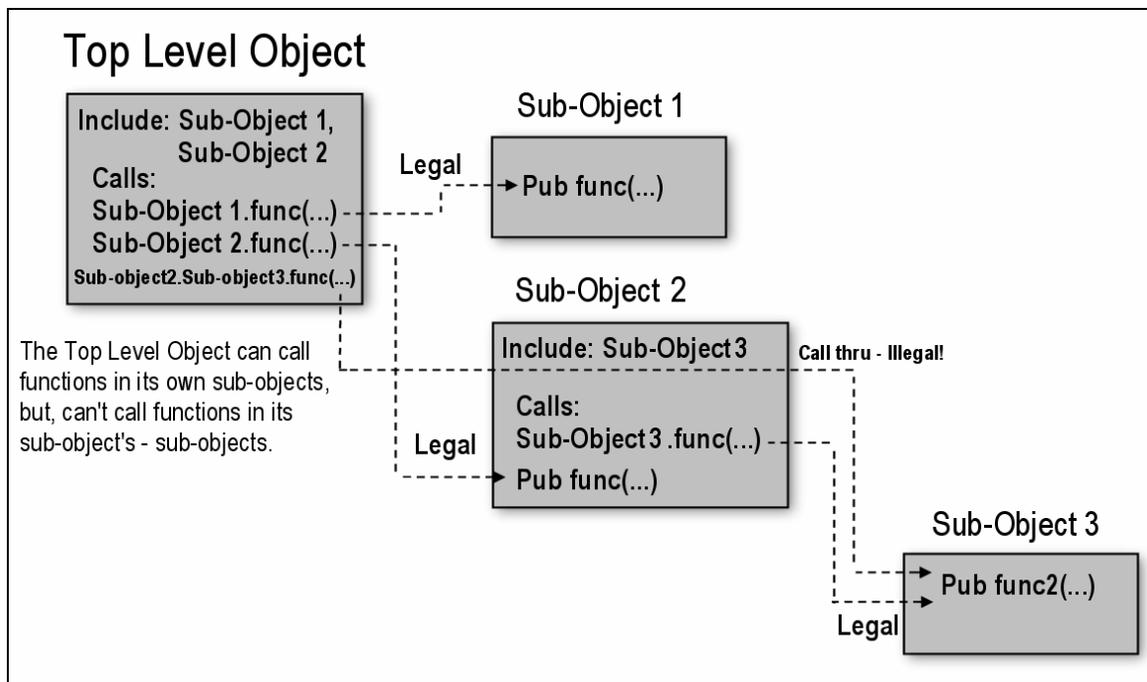


Figure 20.0 shows a close up of the driver and its constituent parts as outlined in the Propeller IDE object view. The driver itself uses the NTSC TV terminal object to print to the screen for debugging and directory print out etc. Thus, the SD Max driver itself includes the TV terminal object, which in turn includes the graphics and TV objects. This pulls in a lot of code and uses a lot of space, especially the TV driver and graphics driver since they are bitmapped based. Thus, when you start to run out of room, then you will probably want to go into the driver and remove all printing calls. Of course functions like the directory listing FAT16 function will have to be modified to pass back strings of filename rather than print to the screen directly.

Figure 21.0 – SPIN can't call a sub-object's objects.



Additionally, since the TV driver is to be used in the demo to print information to screen, we need to get to the terminal driver sub-object itself. But, SPIN only allows one level of sub-object access. Meaning, say you have a top level object A. Then say that A includes a sub-object B, like this:

```

' object A, top level object
OBJ
    B : object.spin ' include object B in A
    
```

Now, A can call functions of B with this syntax:

From Object A, we make the call:

```
B.function_name(...)
```

Which is fine and dandy. But, let's say that Object B includes Object C (a math library for example that is very large) as shown below:

```
' object A, top level object
OBJ
  B : bobject.spin ' include object B in A
```

```
' object B, includes object C
OBJ
  C : math.spin ' include object C in B
```

Now, say from the top level Object A, we want math support. But, we don't want to include the math object again since we already know B included it, this we would expect to be able to access the math functions in C thru B like this:

From Object A, we make the call:

```
B.C.math_function(...)
```

But, this is **not** the case! You have to include the math Object C in every object that is going to use it. This is a huge waste of memory and is being addressed by Parallax for the next implementation of the SPIN compiler. In the meantime, there are numerous hacks to expose functions in a sub-object. For example, one trick is to create wrapper function **"interface"** so that from Object B, there are functions that **"wrap"** the sub-functions in C, therefore from the top level Object A, you make calls to B, but these call then make calls to C finally calling the function for you. Therefore, we get a call structure that looks like:

From Object A, we make the call:

```
' this code is in Object A
B.interface_function_math_wrapper(parms...)
```

```
' this code is inside of Object B, and "interfaces" or exposes the math calls to B, so A can see them
PUB interface_function_math_wrapper(parms...)
  C.math_function(parms...)
```

Referring back to Figure 21.0(b), this is the technique that is used in the SD Max driver API to expose the terminal object print functionality back to the top level object. Its ugly, but no way around it. Of course, we could have zero printing in the driver itself and return strings back for the caller to perform printing, but then when you try to develop code you are going to have a hard time fitting the terminal driver at the top level and at the bottom level, so I decided to leave the terminal in one place within the SD Max driver, so it could access it as you hack my code. Plus, this is a good example to show off the "interface" technique to gain access to sub-objects and sub-functions when needed. In any event, let's continue with the API overview beginning with the constants.

6.1 Driver API Constants

The constants for the SDMAX driver cover the hardware interface, SPI, and FAT16. They are shown below:

```

CON

' SPI (serial peripheral interface) interface pins
spiDO = 16
spiCLK = 17
spiDI = 18
spiCS = 19

' SD card (secure digital) constants (you can reduce these to speed things up and see if the SD card still works)
SD_RESPONSE_LIMIT = 256 ' max number of iterations that SD card queried for response after a command sent
SD_POWERUP_LIMIT = 256 ' max number of iterations to initially clock SD card to power up
SD_RETRIEVE_SECTOR_LIMIT = 4096 ' max number of iterations for SD to retrieve a requested sector/block
SD_WRITE_SECTOR_LIMIT = 128 ' max number of iterations for SD to write a requested sector/block

' SD commands (based on SD spec names)
SD_GO_IDLE_STATE = $40 ' resets the SD card, places it into SPI mode, idle
SD_EXIT_IDLE_STATE = $41 ' takes SD out of idle state ready for commands

SD_APP_CMD = ($40 + $37) ' makes sure card is in SD mode
SD_SEND_OP_CMD = ($40 + $29) ' makes sure card is in SD mode
SD_READ_SINGLE_BLOCK = ($40 + $11) ' reads a single block (512 bytes) at byte address sent in command address
SD_WRITE_BLOCK = ($40 + $18) ' write a single block (512 bytes) at byte address sent in command address

SECTOR_SIZE = 512 ' should always be 512 for SD cards
MAX_SECTORS_BUFFER = 1 ' number of sectors buffer can hold, make larger if needed

' fat constants for various functions
FILE_SEEK_ABS = 0 ' interpret seek request with absolute position
FILE_SEEK_REL = 1 ' interpret seek request as relative position added to current position
FILE_SEEK_END = 2 ' seek to end of file (file_size - 1)
FILE_SEEK_START = 3 ' seek to start of file (0)

' debugging constants for function terminal debug outputs
DEBUG_OFF = 2 ' no output at all
DEBUG_VERBOSE = 1 ' verbose mode for functions debug output
DEBUG_BRIEF = 0 ' brief mode for functions debug output

```

The constants are well documented, so you shouldn't have any trouble understanding what they are for. But, let's take a quick look at them as a whole. The first **4 constants** are the **I/O ports** used by the **SPI** interface. Change these if you decide to use the driver for something else or build your own SD card interface and you need to use other I/O pins.

Next up are the **SD command protocol** constants. The first set of them are used as time limits to wait for responses for various operations. For example, when writing a sector, the code will wait 128 iterations and then exit with an error. These constants can be changed to speed up some of the algorithms, but as you reduce their values some SD cards might not respond. I have tested these with 5-6 major manufacturers and they work 100%. But, as I reduce the constants more, some of the SD cards begin to fail.

After the SD timing constants, you see the **SD commands** themselves that are supported downstream in the code. As you see, there are very few that we support. You can add others yourself by reviewing the SD card specifications and implementing more functions if you wish.

The next constants define sector size and the maximum number of sector buffers. The second constant **MAX_SECTORS_BUFFER** is worth mentioning. The driver is going to create a number of local temporary sector buffers to hold the Master Boot Record, Partition Boot Record, FAT sectors etc. But, a general "working" buffer is also created in the driver. The sector size of this buffer is controlled by **MAX_SECTORS_BUFFER** and this is important since the caller that instantiates the driver object gets access to this buffer since it would be wasteful to create a large buffer for the driver alone and not let the caller use it, especially on such a limited memory chip as the Propeller with only 32K of RAM. The buffer is set to 1 sector right now, since you can do everything 1 sector at a time, but later if you want to increase its size this is where you will do it. Moreover, later in the global's discussion and the start up sequence you will see how the caller (top level object) passes a pointer array to the driver object, where the driver object stores the actual local addresses of the local buffers, so the call can get access to this orphaned memory in the driver if needed. Again, since SPIN code is so large and the Propeller has such a small amount of memory, we have to be frugal where possible.

TIP

One great improvement for the demo program later in the manual is to use a custom terminal program that doesn't use the standard Parallax TV and graphics driver. This way, no video memory would be needed for a pure character mode and around 20K bytes would be freed by the VRAM going away.

Moving on, the next constants are simply **commands for the FAT16** file seek functions. Nothing too interesting there. The final constants are important. These control the driver's internal debugging output level. In other words, how verbose the driver is. The idea of the driver is to help you experiment with SD cards, so I have sprinkled lots of debug code in to spit out interesting things as the functions are called. This is controlled by a global variable called **debug_level** in the driver itself. The assignment of **debug_level** determines the amount of printing.

6.2 Driver API Globals

The driver API globals store important information like the **Master Boot Record**, and **Partition Boot Record**. Keep track of various values as the driver executes, and also since SPIN has limited syntax for local variables, that is, its hard to put a lot of local in a function without a huge single line list of variables off the right hand of the screen, many variables were made global to make them more accessible. In any event, here's the complete globals section from the driver for your review:

```
VAR
' API GLOBALS-----
' debug values
long debug_level          ' setting of printout for debug level (DEBUG_OFF, DEBUG_VERBOSE, DEBUG_BRIEF)
long debug_wait_res_num_iter ' the number of times the sd send command function internally waits for a SPI response
' these are the local working buffers to hold the master boot record, partition boot record, fat sectors, and general diskbuffer to
' read sectors, in general the driver needs and uses all of these, but to save memory, so the caller doesn't have to instantiate his
' own buffers, pointers to these are passed BACK to the caller when he calls the Start method, then he can use the storage buffers
' and access, inspect them as well, currently 2K bytes is used by all buffers. Technically, once the MBR and PBR have been loaded
' they can be destroyed and the memory used, but then you would have to reload them to review or inspect, thus, suggest leave them
' alone after loading...fatbuff is used internally by the file functions, thus between calls, diskbuff is not used by any of the calls, so
' will remain intact call to call. However, when debug_level is verbose then diskbuff is used in a couple function to load sector data to
' print therefore, it will NOT remain intact call to call, but in debug level brief and off, diskbuff is NOT used by the driver internally
' and can be used by the caller for storage during the file read operations

byte diskbuff[ SECTOR_SIZE*MAX_SECTORS_BUFFER ] ' generic disk sector buffer
byte mbrbuff[ SECTOR_SIZE ] ' master boot record buffer
byte pbrbuff[ SECTOR_SIZE ] ' partition boot record
byte fatbuff[ SECTOR_SIZE] ' holds one fat sector at a time

' global master boot record variable template for partition entry (subset, important only)
long mbr_boot_descriptor ' boot descriptor (determines if entry is active, $80=active, $00=inactive), offset $00, 1-byte
long mbr_first_part_sect ' first partition sector, offset $01, 1-byte
long mbr_file_sys_desc ' fat system descriptor ($04 means fat16 system < 32MB, $06 means fat16 > 32MB, offset $4, 1-byte)
long mbr_last_part_sect ' last partition sector, offset $05, 3-bytes
long mbr_num_sect_to_part ' number of sectors between the MBR and the first sector of the partition, offset $08, 4-bytes
long mbr_num_sect_in_part ' number of sectors in the partition, offset $0C, 4-bytes

' global partition boot sector variable template (subset, important only)
long pbr_bytes_per_sect
long pbr_sect_per_clust
long pbr_num_res_sect
long pbr_num_fats
long pbr_num_root_dir_entrys
long pbr_total_num_sect
long pbr_num_sect_per_fat
long pbr_num_vol_id
byte pbr_volume_label[12]

' global FAT tracking globals
long first_data_sector
long first_dir_sector
long first_fat_sector

' temp global FAT directory entry (subset, important only)
long dir_attr
long dir_reserved
long dir_create_time_ms
long dir_create_hms
long dir_create_date_ymd
long dir_last_access_date_ymd
long dir_ea_index
long dir_last_update_time_hms
long dir_last_update_date_ymd
long dir_first_cluster
long dir_file_size
long hours, minutes, seconds
long year, month, day

byte tfilename[16] ' temporary string buffer, used for various string operations by functions internally
```

The globals begin with a couple debugging values, the first we have discussed, it simply control how much informational printing the driver does. The second debug global is internal for debugging and not important. The next four arrays are though. These are storage space for the **Master Boot Record**, the **Partition Boot Record**, the **current FAT table sector**, as well as a general “**working**” buffer. The driver could have been written so that all of these were transient, but I decided to cache them for easy inspection and to make the code cleaner rather than constantly reloading them every time we need to inspect them. **diskbuff[]** is the only buffer that is transient and it along with the starting address of it along with the others are passed back to the caller when the driver is started up, so the caller can inspect them. These buffers are a slight waste of space, but the ease of access outweighs the 2K bytes they use up.

Next are the **Master Boot Record globals** used by the driver. The MBR is sector 0, so the driver at some points load it in, but the problem is that all the data is encoded, so rather than have a bunch of functions that constantly have to do bit twiddling, the MBR is decoded once and important values are stored in these human readable values starting with **mbr_***.

The MBR globals are followed by the **Partition Boot Record globals** which serve the same purpose. These are all internal to the driver and make the code easier to understand. Finally, there are 3 very important globals that the driver computes at some point; the first data sector, first FAT sector, and first directory sector. Again, these could be computed on the fly, but much cleaner to compute them and store in globals.

Last up are some globals to help with the directory listing function, without these globals the code would be cryptic and there is no room for them as locals due to SPIN's local syntax forces locals on the same line as the function itself, making for very long and unreadable lines. So, the globals starting with "**dir_**" are all used in the directory listing function.

The only globals the caller has access to in a round about way are the disk buffers. But, more on that later, when we see the functions that initialize the API driver.

6.3 Initializing the SD Max Driver

Using the SD driver in your code is very simple you only need to create some globals, include the driver object, and then start the object up. The top level object needs a few globals in the VAR section to pass to the startup function. These will be written with pointers to the driver's local buffers. Here's an example excerpted from the demo program:

```
' DRIVER SUPPORT VARS -----
'
'application calls -----> SDMAX_DRV_010.Start with address of 1st pointer @diskbuff_ptr
'
'      Application locals      |      Driver Local buffers
'      diskbuff_ptr <----- @ diskbuff [ ... ]
'      mbrbuff_ptr  <----- @ mbrbuff  [ ... ]
'      pbrbuff_ptr  <----- @ pbrbuff  [ ... ]
'
long diskbuff_ptr ' generic disk sector buffer, initialized by call to SD driver's Start
long mbrbuff_ptr  ' master boot record buffer , initialized by call to SD driver's Start
long pbrbuff_ptr  ' partition boot record   , initialized by call to SD driver's Start
'
' END DRIVER SUPPORT VARS -----
```

The variables can have any name, but its suggested using names above. These variables will point to the sector buffers in the driver itself, when the initial call to the SD driver's **start(..)** method is made the address of the first pointer **@diskbuff_ptr** is passed to the function, then the function fills in each pointer with the address of the SD drivers local memory buffers, so that the caller can access the buffers and not need to allocate more buffers itself. The **Master Boot Record** and **Partition Boot Record** buffers are **volatile** and should only be read or re-loaded with appropriate driver calls, but the **diskbuff_ptr** points to a temporary buffer in the driver and can be used between calls as storage, but the driver "may" destroy the contents call to call, so watch out. Each buffer is a byte buffer, so when you want to access the actual bytes, you must cast to byte and use syntax such as:

```
byte[diskbuff_ptr][index]
```

Then you need to make sure to include the object in your top level program:

```
OBJ
sd : "sdmax_drv_010.spin" ' instantiate SD card driver object
    ' use "sdmax_drv_nd_010.spin" version to minimize debugging output
```

At which point, you can start the object up and pass it the starting address of your pointer receiver area like this:

```
' start the SD driver up, also starts up tv terminal for local printing (necessary)
' notice that the call MUST be made with the address of the receiver pointer area
sd.Start(@diskbuff_ptr)
```

After this call the driver will point all the pointers in the local globals at the buffers in the driver's local memory, so the variables:

```
long diskbuff_ptr ' generic disk sector buffer, initialized by call to SD driver's Start
```

```
long mbrbuff_ptr ' master boot record buffer , initialized by call to SD driver's Start
long pbrbuff_ptr ' partition boot record      , initialized by call to SD driver's Start
```

Will be valid pointers which you should not modify, but simply use in your calls.

6.4 Master API Listing

In the following sections we will discuss each function call with a prototype and example usage. As a reference Table 14.0 below is a complete API listing sorted by group for ease of reference.

Table 14.0 – Master API function call reference.

Initialization Functions

Start (buffer_ptrs) - Initializes the driver and writes the callers pointer storage space with local disk buffer addresses .

SPI Functions

SPI_Init(mode) - Initializes the hardware SPI interface I/O pins from propeller to card.
SPI_Send_Recv_Byte(spi_data8) - Sends a single byte to SPI interface and receives one at the same time.
SPI_Read_Byte - Reads the SPI buffer, writes a dummy values of \$FF.
SPI_Write_Byte(spi_data8) - Writes a byte to the SPI interface.

SD Functions

SD_Mount - Mounts the SD card by initializing and placing it into SPI mode.
SD_Unmount - Unmounts the previously mounted SD card.
SD_Send_Command(command8, address32) - Sends a generic command to the SD card.
SD_Read_Sector(sector32, sectorbuffer16) - Reads a sector from the SD card.
SD_Write_Sector(sector32, sectorbuffer16) - Writes a sector to the SD card.
SD_Wait_Write_Complete - Internal function used in the write command to make sure the flash has been updated.
SD_Print_Sector(sector32, sect_ptr, start_byte, end_byte, base, print_addr) - Diagnostic function that prints the contents of a sector to terminal.
SD_Read_WP - Reads the single bit "write protect" signal on the SD card mechanical.
SD_Read_CD - Reads the single bit "card inserted" signal on the SD card mechanical.

FAT16 Functions

** Lower level functions (initialization of FAT16 system)*

FAT_Read_MBR(mbr_ptr) - Reads the MBR (Master Boot Record) from the SD card, sector 0.
FAT_Load_Partition_Entry(partition, mbr_ptr) - Loads the requested 16-byte partition entry from the loaded MBR.
FAT_Print_Partition_Entry(partition, mbr_ptr) - Pretty prints the partition entry to terminal.
FAT_Load_Partition_Boot_Rec(pbr_ptr) - Loads the partition boot record referred to by loaded partition entry.
FAT_Print_Partition_Boot_Rec - Pretty prints the partition boot record (aka volume record or simply boot record).

** High level functions (general file I/O)*

FAT_Print_Directory - Print the root file directory to the terminal (like DOS DIR command).
FAT_File_Open(filename_ptr, file_handle_ptr) - Opens filename and fills in the sent file handle structure.
FAT_File_Close(file_handle_ptr) - Closes the file handle.
FAT_File_Read(file_handle_ptr, buffer_ptr, count) - Reads bytes from file referred to by the sent file handle.
FAT_File_Seek(file_handle_ptr, count, mode) - Seeks file pointer to a specific location in file for reading.

Utility Functions

To_ASCII(inchar, replace_char) - Used to map non-printable characters to printable.
itoa(value, sptr) - Converts an integer to an NULL terminated ASCII string.

ToUpper(char) - Converts lower case ASCII to upper case.
Strncmp(string_ptr1, string_ptr2, length) - Compares strings.
_Min(a,b) - returns the min.
_Max(a,b) - returns the max.

Exported Object Functions

tvb_bin(value, digits) - Prints a binary number to the terminal with specific number of digits.
tvb_out(char) - Outputs a single character to terminal.
tvb_dec(value) - Prints a number in decimal format to terminal.
tvb_hex(value, digits) - Prints a hex number to the terminal with specific number of digits.
tvb_setx(new_x) - Sets the x cursor position on terminal.
tvb_sety(new_y) - Sets the y cursor position on terminal.
tvb_getx - Gets the current x cursor position in terminal.
tvb_gety - Gets the current y cursor position in terminal.
tvb_pstring(string_ptr) - Prints a NULL terminated string to the terminal.
tvb_erase - Erases completely the character under the current cursor position.

In the following sections we are going to cover each of the API functional sub-classes with a brief overview of each followed by a function listing and example of each function. Use these as a reference along with the final demo program which puts them all together, so you can see them in action!

6.4.1 SPI Function Listing

The **SPI** (Serial Peripheral Interface) functions are the simplest to use since they really have no intelligence. If the SD card is plugged in then you can use these functions to talk to. Of course, unless you use them to send properly formatted SD protocol commands then they won't do much good. However, you can always use these SPI functions for other projects simply by making sure to connect the correct I/O pins to MISO, MOSI, CS, and SCLK.

Function Prototype:

SPI_Init(mode)

Description:

Initializes the hardware SPI interface I/O pins from Propeller to SD Max card. This is the **first** call you must make in all your programs even if you don't use the SPI function directly. The single parameter is ignored currently, set it to 0. In the future, the parameter will control the SPI **mode**.

Returns: Nothing.

Example(s): This example shows the steps to initializing the SD Max driver including the call to **SPI_Init**.

```
VAR
    long diskbuff_ptr ' generic disk sector buffer, initialized by call to SD driver's Start
    long mbrbuff_ptr  ' master boot record buffer , initialized by call to SD driver's Start
    long pbrbuff_ptr  ' partition boot record      , initialized by call to SD driver's Start
OBJ
    sd      : "sdmax_drv_010.spin"          ' instantiate SD card driver object
    ' start the SD driver up, also starts up tv terminal for local printing (necessary)
    ' notice that the call MUST be made with the address of the receiver pointer area
    sd.Start(@diskbuff_ptr)
    ' initialize SPI interface in mode 0 (necessary)
    sd.SPI_Init(0)
```

Function Prototype:

SPI_Send_Recv_Byte(spi_data8)

Description:

Sends a single byte to SPI interface and receives one at the same time.

Returns: The received 8-bit data.

Example(s): Sent a \$55 to the SPI interface and capture the received value in *result*:

```
result := sd.SPI_Send_Recv_Byte( $55 )
```

Function Prototype:

SPI_Read_Byte

Description:

Reads the SPI buffer, writes a dummy values of \$FF.

Returns: The received 8-bit data.

Example(s): Read the SPI bus until a \$E0 is received:

```
repeat while (sd.SPI_Read_Byte <> $E0 )
```

Function Prototype:

SPI_Write_Byte(spi_data8)

Description:

Writes a byte to the SPI interface. Simply a wrapper around the *SPI_Send_Recv_Byte* function.

Returns: The bytes from the SPI interface.

Example(s): Write 0..255 to the SPI interface, ignore the value sent back:

```
repeat data from 0 to 255
  sd.SPI_Write_Byte( data )
```

6.4.2 SD Protocol Function Listing

The SD protocol API layer is really the *most important* part of the API. With the SD protocol layer you can more or less communicate with the SD card, read and write sectors and implement any file system you wish. The SD protocol was covered in the section above, so please refer to that as you read the listing.

The SD layer is very easy to use, you simply need to “**mount**” the SD card before attempting any communication to the card. Once mounted, you are free to send commands to it for reading, writing, and so forth. When you are done with an SD card, you can unmount it as well. Also, you are free to re-mount cards at any time. That is, if the user pulls the current SD card from the SD Max and places another one in there, you simply need to call the *SD_Mount* call again, and all the internal data structures will be refreshed. The FAT16 API layer is not that simple. If a user pulls the SD card the a number of calls have to be made in addition to re-mounting the card to re-load the boot sectors, etc. But, more on that in the FAT API listing discussion.

Function Prototype:

SD_Mount

Description:

Mounts the SD card by initializing and placing it into SPI mode.

Returns: \$FF for error, \$00 success.

Example(s): Assuming the SD Max card is plugged into the HYDRA with an SD card, this example will mount the SD Max card:

```
if (sd.SD_Mount == $00)
    success
else
    failure
```

Function Prototype:

SD_Unmount

Description:

Unmounts the previously mounted SD card. Always make sure to call the unmount function when you exit your file code. Even though, the unmount doesn't do much in this version, later it might maintain some data structures that you would want to free up. Currently, disables the chip select line to the SD card, that's all.

Returns: Nothing.

Example(s):Mount and unmount an SD card.

```
SD_Mount
' do work...
' end of program
SD_Unmount
```

Function Prototype:

SD_Send_Command(command8, address32)

Description:

Sends a generic command to the SD card. The commands are listed back in Table 7.0 for reference. The parameters are **command8** which is the 8-bit command code and **address32** which is a 32-bit address for commands that needed addresses. This function is relatively low level and you wouldn't call it unless you are implementing new functions.

Returns: \$FF means error, other values depend on the command sent.

Example(s): Place the SD card into "idle" mode:

```
' send SD command GO_IDLE_STATE = $40, which resets the SD card
sd_response := sd.SD_Send_Command( SD_GO_IDLE_STATE, 0 )
```

Function Prototype:

SD_Read_Sector(sector32, sectorbuffer16)

Description:

Reads a sector from the SD card and stores it into the buffer. **Sector32** is the 32-bit sector address you wish to read, **sectorbuffer16** is a pointer to a 512 byte buffer that the sector bytes will be stored in.

Returns: \$00 if no errors, \$FF if error.

Example(s): Read sector 0 in and scan for the boot signature \$55AA.

```
VAR
    byte buffer [ 512 ] ' we need a buffer to hold the sector
' read boot sector 0 (the master boot record)
```

```
sd.SD_Read_Sector( 0, @buffer[0] )
found :=0 ' set flag to not found
' now scan for $55AA using WORD pointer movements, 256 words = 512 bytes
repeat index from 0 to 256
  if ( word [ @buffer[0] ] [ index ] == $55AA )
    found := 1
    quit
' test found flag now...
```

Function Prototype:

SD_Write_Sector(sector32, sectorbuffer16)

Description:

Writes a sector to the SD card. **Sector32** is the 32-bit sector address you wish to write and **sectorbuffer16** is a pointer to the 512 byte sector data you want to write. The function is slow compared to read since it must wait for the write to complete.

Returns: \$00 success, \$FF error.

Example(s): Zero out sector 10_000 to 20_000 inclusive

```
VAR
byte buffer[512] ' sector data buffer
' clear the buffer to 0's
bytefill( @buffer[0], 0, 512 )
repeat sector from 10_000 to 20_000
  sd.SD_write_Sector( sector, @buffer[0] )
```

Function Prototype:

SD_Wait_Write_Complete

Description:

Internal function used in the write command to make sure the flash memory write has completed, blocking call.

Example(s): N/A

Function Prototype:

SD_Print_Sector(sector32, sect_ptr, start_byte, end_byte, base, print_addr)

Description:

Diagnostic function that prints the contents of a sector to terminal. **Sector32** is the 32-bit sector address to print, **sect_ptr** points to storage sector buffer you want to the function to use for buffering, **start_byte** to **end_byte** form the range to print, **base** is the numeric base to print in (10=decimal, 16=hex, 127=ASCII), **print_addr** is simply an offset to add to the starting address to make the display match to whatever you want since its always 0 based. Use this function when you are debugging and want to see bytes in a nice tabular form in decimal, hex, or ASCII.

Returns: N/A.

Example(s): Load in sector 125 and print the first 128 bytes to the screen in hex format:

```
VAR
byte buffer [ 512 ] ' we need a buffer to hold the sector

' now print the sector
sd.SD_Print_Sector(125, @buffer[0], 0, 127, 16, $0000)
```

Function Prototype:

SD_Read_WP

Description:

Reads the single bit "write protect" signal on the SD card mechanical and returns its value.

Returns: 0 = write protect disabled, 1=write protect enabled.

Example(s): Test if the card is write protected, if so, tell user.

```
if (sd.SD_Read_WP == 1)
    ' write protected!
else
    ' ok to write
```

Function Prototype:

SD_Read_CD

Description:

Reads the single bit "card inserted" signal on the SD card mechanical.

Returns: 0=card inserted, 1=card not inserted.

Example(s): Test if card is inserted, if so continue with program, otherwise exit.

```
if (sd.SD_Read_CD == 1)
    quit
else
    ' card is inserted, go ahead and do whatever
```

6.4.3 FAT16 Function Listing

The FAT16 file functions supported by the driver are the bare minimum needed to display directory information and read files. Write file functionality is left for you to implement yourself, but the overall algorithm was outlined in the discussion on the FAT16 system in the sections above.

Working with the FAT16 functions assumes that the **SPI system has been initialized** as well as the **SD card mounted** successfully. Also, if you recall from the discussion on FAT16 the call that starts the driver must be passes the starting address of pointer array that the driver will write back to giving the caller access to its local buffers to save memory. The code sequence was:

```
' start the SD driver up, also starts up tv terminal for local printing (necessary)
' notice that the call MUST be made with the address of the receiver pointer area
sd.Start(@diskbuff_ptr)
```

After this call the driver will point all the pointers in the local globals at the buffers in the driver's local memory, so the variables:

```
long diskbuff_ptr ' generic disk sector buffer, initialized by call to SD driver's Start
long mbrbuff_ptr  ' master boot record buffer , initialized by call to SD driver's Start
long pbrbuff_ptr  ' partition boot record    , initialized by call to SD driver's Start
```

This step is always the very first, and if you use the same variable names then it will be consistent with demo that comes up later as well as the examples here. After the initial driver **Start**, SPI, and SD mount calls then in most cases you will follow these steps to get the FAT16 up and running:

Step 1: Read the Master Boot Record in with **FAT_Read_MBR(...)**.

Step 2: Load the first partition entry into the globals with a call to **FAT_Load_Partition_Entry(...)**.

Step 3: Load the Partition Boot Record also known as the Boot Sector with **FAT_Load_Partition_Boot_Rec(...)**.

At this point, the FAT16 system and all the globals are ready to go. Then the majority of your time you will spend making calls to the file open, close, read and seek functions. Also, you can list the directory of the SD card at any time with a call to the directory function. With that in mind, the following functional listing starts with the lower level functions and ends with the higher level functions.

Lower level functions (initialization of FAT16 system)

Function Prototype:

```
FAT_Read_MBR(mbr_ptr)
```

Description:

Reads the MBR (Master Boot Record) from the SD card, sector 0. The function expects a pointer in **mbr_ptr** to a 512 byte buffer to receive sector 0 data.

Returns: \$00 if no errors, \$FF if error.

Example(s): Load in the MBR into the proper buffer for the driver to access it.

```
' read Master Boot Record, pass the pointer the driver sent back as the local storage for the MBR
' redundant, but gives you the flexibility to load the MBR into another buffer
sd.FAT_Read_MBR(mbrbuff_ptr)
```

Function Prototype:

```
FAT_Load_Partition_Entry(partition, mbr_ptr)
```

Description:

Loads the requested 16-byte partition entry from the loaded MBR. This function assumes the MBR has been previously loaded (since the driver refers to the MBR to extract the partition entries). **Partition** is the partition you want to load 0..3, but you will always load 0 since in 99.9% of cases there is only a single partition on an SD card. **mbr_ptr** is the global pointer that the driver wrote to as a call back, and you simply pass this value again. When the function completes, the driver will have updated some local variables with the partition entry data, but you still must use that information to actually load the partition boot record.

Returns: Success 1, or failure 0.

Example(s): Load the MBR, then partition entry 0.

```
' read Master Boot Record
sd.FAT_Read_MBR(mbrbuff_ptr)
' load the values from the first partition into the globals
sd.FAT_Load_Partition_Entry(0, mbrbuff_ptr)
```

Function Prototype:

```
FAT_Print_Partition_Entry(partition, mbr_ptr)
```

Description:

Pretty prints the partition entry to terminal display with labels. Parameters are **partition** 0..3 which indicates which partition entry to print, **mbrbuff_ptr** must point to the pre-loaded MBR.

Returns: N/A.

Example(s): Print partition entry 0..3, assumes that the code has already loaded in the MBR into the memory storage pointed to by *mbr_ptr*.

```
' iterate thru all 16-byte partition entries and print them out to screen
repeat partition from 0 to 3
  FAT_Print_Partition_Entry(partition, mbr_ptr)
```

Function Prototype:

FAT_Load_Partition_Boot_Rec(pbr_ptr)

Description:

Loads the partition boot record referred to by loaded partition entry. This function works in tandem with the **FAT_Load_Partition_Entry** function. The aforementioned function must be called **before** **FAT_Load_Partition_Boot_Rec** is called. The “partition entry” is a completely different data structure that the “Partition Boot Record” and shouldn’t be confused. The partition entry describes **where** the Partition Boot Record is for that particular partition 0..3, and thus, this function uses that information to load the Partition Boot Record itself. **Pbr_ptr** is simply the pointer that will receive the data from the boot record, in most cases you will use the called back set pointer from the initial call to the driver’s **Start** function.

Returns: Assigns the global template variables *pbr_** with the values found in the partition record

Example(s): Here’s an example of loading the MBR, partition entry 0, and finally Partition Boot Record 0.

```
' read Master Boot Record (necessary)
sd.FAT_Read_MBR(mbrbuff_ptr)

' load the values from the first partition into the globals (necessary)
sd.FAT_Load_Partition_Entry(0, mbrbuff_ptr)

' read in the partition boot record selected previously with the loaded partition entry
sd.FAT_Load_Partition_Boot_Rec(pbrbuff_ptr)
```

Function Prototype:

FAT_Print_Partition_Boot_Rec

Description:

Pretty prints the partition boot record (aka volume record or simply boot record). Assumes the Partition Boot Record has previously been loaded with a call to **FAT_Load_Partition_Boot_Rec**.

Returns: N/A.

Example(s): Print out the important parts of the Partition Boot Record for review.

```
' assumes the PBR has been loaded into pbr_ptr
sd.FAT_Print_Partition_Boot_Rec
```

High level functions (general file I/O)

Function Prototype:

FAT_Print_Directory

Description:

Print the root file directory to the terminal (like DOS DIR command). This function pretty prints the directory to the terminal screen. There is no pause, so files will scroll by if there are more than 24 lines worth. If you wish, you can modify the function source to be more intelligent. But, its a great tool to “see” what’s on the SD card. Assumes, the file system has been mounted and the MBR and PBR have been loaded.

Returns: N/A.

Example(s): Print the current root directory of the SD card to the screen.

```
FAT_Print_Directory
```

Function Prototype:

```
FAT_File_Open(filename_ptr, file_handle_ptr)
```

Description:

Opens the file in the root directory with the sent filename in 8.3 format and fills in the sent file handle structure.

Filename_ptr should point to an ASCII string filename in **8.3 format** with a **NULL** terminator, **file_handle_ptr** should point to the beginning of a zero'ed out "**file handle**" data structure which is 4 longs in size and has the following memory footprint:

```
long filehandle[4] ' generic file handle first_cluster, file_length, curr_pos, directory_entry
' long first_cluster ' first cluster of file
' long filesize ' size in bytes of file
' long curr_read_pos ' current read position, -1 means end of file, or anything past the size
' ' convention is that the pointer is always incremented then read
' ' from, thus if curr_read_pos = 0 then the byte at location 0 has
' ' already been read, the next byte read will be at 1
' long dir_entry ' the directory index of the entry
```

Each of the 4 elements of the file handle stores important information about the file to help access it in the future. If you have ever programmed in an advanced language like C/C++, JAVA, Pearl, PHP, etc. then the concept of "file handles" should be familiar to you. They are simply internal data structures the operating systems uses to track the file and the current read/write position an so forth. For our little driver, we don't need a lot of bells and whistles, so the 4 fields that make up the file handle are more than adequate. The comments above outline each field, but below is a more detailed description:

File Handle Structure

long [0] : **first_cluster** – This is the actually first cluster index that the file data starts. It will always be greater than 2.

long [1] : **filesize** – This is the size of the file in bytes.

long [2] : **curr_read_pos** – The is the current file pointer from 0 to filesize. This is where the next read operation takes place.

long [3] : **dir_entry** - This is the "directory entry" index in the root directory. The directory has 512 entries usually, this index is used internally to help locate the original 32-byte directory entry if more file information needs to be accessed like the creation date, time stamps, archive bits etc. This helps so we don't have to scan for the file name again in the directory.

After the **FAT_File_Open** function returns the 4 longs pointed to by **file_handle_ptr** will be filled in with the relevant information about the file. This file handle then is used to read, seek, or close the file in later calls.

Returns: -1 if failure, else length of file in bytes.

Example(s): Open the file named "data.txt".

```
VAR
filehandle1[4]
' clear the fill handle out, good habit
```

```

longfill( @filehandle,0, 4)
‘ try and open the file
size := sd.FAT_File_Open( string (“data.txt), @filehandle[0] )
‘ check for success
if (size == -1)
‘ file not found!
else
‘ file found, do work...

```

Function Prototype:

FAT_File_Close(file_handle_ptr)

Description:

Closes the file handle. This functions “**closes**” the previously opened file. However, since there is no operating system to speak of, and the SD card has no concept of “**open**” or “**closed**” files thus, all the function really does is invalidate the file handle, so if you try to make calls with it, nothing will happen. However, for future upgrades, you should always close files when you are done with them, so that if the API is modified to have a limit on “**open**” files that tactic will protect against memory leaks.

Returns: Returns 1 if successful, 0 if failed or file handle NULL.

Example(s): Open the file “database.dat” and then close it.

```

‘ try and open the file
sd.FAT_File_Open( string (“database.dat), @filehandle[0] )
‘ .. do some work with file
‘ close the file
sd.FAT_Close_File(@filehandle[0])

```

Function Prototype:

FAT_File_Read(file_handle_ptr, buffer_ptr, count)

Description:

Reads bytes from file referred to by the sent file handle and stores them in the buffer pointed to by **buffer_ptr**. On entry, **file_handle_ptr** should be a valid file handle, **buffer_ptr** should point to a buffer to receive the data, and **count** is the number of bytes to read. The function will read the bytes requested up to the end of the file and store them into the buffer. If you request more bytes that are available in the file, the read will stop at the end of the file. In all cases, the function returns the number of bytes read and updates the file handle’s read/write file position. Also, its very important to keep in mind that the buffer you pass the function to receive the bytes must be large enough to hold the data, otherwise data will overwrite memory and typically crash the Propeller chip. That said, in most cases its best to use a single sector buffer and then perform reads in 512 byte chunks, so you can minimize the size of the required buffer.

Returns: Number of bytes read, can potentially be 0.

Example(s): Open a file of unknown size and sum the values in the file and average to create a checksum. Notice that the algorithm uses the **diskbuff_ptr** which we know points to a 512 byte storage buffer in the driver itself which can be used for general purpose. This way we do **not** have to allocate local memory for buffering data.

```

‘ open the file
filesize := sd.FAT_File_Open( string (“data.dat), @filehandle[0] )
‘ reset sum
checksum := 0
‘ read in file a sector at a time, at some point we are going to read < a sector, so be careful!
fp := 0
repeat while (fp < filesize )
‘ read next sector
bytesread := sd.FAT_File_Read(@filehandle[0], diskbuff_ptr, 512)
‘ now sum the bytes into the total

```

```

repeat index from 0 to bytesread-1
checksum += byte [diskbuff_ptr][index]

' update fp with actual number of bytes read this time around
fp += bytesread

' at this point we are done, now average
checksum := checksum / filesize

' close file
sd.FAT_Close_File( @filehandle[0] )

```

Function Prototype:

```
FAT_File_Seek(file_handle_ptr, count, mode)
```

Description:

Seeks file pointer to a specific location in file for reading. All this function does is update the 3rd long in the file handle data structure to the new value. However, its safer than modifying it directly since it does some bounds checking for you. Also, you can send the function a couple different command modes which are reminiscent of C/C++ file operations to perform absolute, relative, seek to start, seek to end functionality. The parameters are **file_handle_ptr** which should point to a valid file handle, **count** which is the value to seek (could be relative or absolute depending on the next parameter), and finally **mode** which controls the mode of the seek function as outlined below in the table below:

Seek Mode	Description
FILE_SEEK_ABS	Interpret seek request with absolute position.
FILE_SEEK_REL	Interpret seek request as relative signed position offset added to current position.
FILE_SEEK_END	seek to end of file (file_size - 1), count is ignored, set to 0.
FILE_SEEK_START	seek to start of file position (0), count is ignored, set to 0.

Note: These constants are located in the driver itself, thus to access them use the syntax **object_name#constant_name**.

Returns: Updated file position.

Example(s):

Open a file and seek to the end of it.

```

' open the file
filesize := sd.FAT_File_Open( string ("data.dat), @filehandle[0] )

' seek to end
sd.FAT_File_Seek( @filehandle[0], 0, sd#FILE_SEEK_END )

```

Open a file and move the file pointer 10 bytes at a time until the end of the file is reached.

```

' open the file
filesize := sd.FAT_File_Open( string ("data.dat), @filehandle[0] )

repeat index from 0 to filesize - 1 step 10

' seek 10 bytes from current position
sd.FAT_File_Seek( @filehandle[0], 10, FILE_SEEK_REL )

```

6.4.5 Utility Function Listing

The utility function listing covers a number of helper functions that were needed in the development of the SPI, SD, and FAT16 API. You might find them useful, so they are public and accessible from the top level object.

Function Prototype:

```
To_ASCII(inchar, replace_char)
```

Description:

Used to map non-printable characters to printable. ***Inchar*** is the character you want tested, ***replace_char*** is used to replace the character if it's a non-printable control code. You would use this macro to filter characters before you send them to the terminal driver, so they don't cause the screen to go crazy.

Returns: ***inchar*** if printable, else ***replace_char***.

Example(s):

Send the characters of a string to a printing function that won't tolerate non-printable ASCII codes.

```
' assume string_ptr is pointing to ASCII string that is NULL terminated, but might have non-printable characters
' replace them with SPACE $20 before passing on to print function
index := 0

repeat while (byte [string_ptr][index++] <> 0 )
' call print function elsewhere
print ( sd.To_ASCII (byte [string_ptr][index], $20 ) )
```

Function Prototype:

itoa(value, sptr)

Description:

Converts an integer to a NULL terminated ASCII string. ***Value*** is the signed 32-bit decimal integer to be converted, ***sptr*** is a pointer to the buffer to receive the ASCII converted string. If the buffer isn't large enough to hold the result, the function will overwrite into memory.

Returns: Writes string to sptr and returns pointer to string as well

Example(s): Convert the 32-bit binary variable ***power*** into an ASCII string for printing or formatting:

```
VAR
long power          ' the variable we are converting from int to ascii
byte string_buffer[11] ' enough storage for 10 digits plus NULL

' and down in the main code somewhere
sd.itoa ( power, @string_buffer[0] )
```

Function Prototype:

ToUpper(char)

Description:

Converts lower case ASCII to upper case. If ***char*** is a character from a..z then it upcases it; otherwise, it passed the character thru unscathed.

Returns: The ASCII upper case value of the character.

Example(s): Convert a previously input string stored in ***input_str*** into all upper case:

```
length := 0
repeat while (byte [input_str][length] <> 0)
' convert to upper case
byte [input_str][length] := sd.ToUpper (byte [input_str][length++])
' string input_str is converted to all uppercase, and length holds its length!
```

Function Prototype:

Strncmp(string_ptr1, string_ptr2, length)

Description:

Compares strings pointed to by *string_ptr1* and *string_ptr2*, up to *length* characters. The comparison is *not* case sensitive. So “hello” and “hELLO” would be a match.

Returns: 1 if equal, 0 if not equal.

Example(s): Compare the first 5 characters of two strings.

```
if ( sd.Strncmp ( string ("Hello world"), string ("hello world series"), 5 ) == 1)
  ' strings are equal, at least first 5 characters are
else
  ' strings aren't equal
```

Function Prototype:

`_Min(a, b)`

Description:

Returns the signed min of (a,b).

Example(s): Find the minimum value in a 1000 element array named *data[]*.

```
min := 2_000_000_000 ' set to large number
' index thru array and check if current data is smaller than current minimum
repeat index from 0 to 999
  min := sd._Min( min, data[ index ] )
```

Function Prototype:

`_Max(a, b)`

Description:

Returns the signed max of (a,b)

Example(s): Find the maximum value in a 1000 element array named *data[]*.

```
max := -2_000_000_000 ' set to small number
' index thru array and check if current data is larger than current maximum
repeat index from 0 to 999
  min := sd._Max( max, data[ index ] )
```

6.4.6 TV Terminal Interface Listing

The TV terminal interface is a sub-object of the SD Max driver itself. However, its very useful to be able to print to the terminal from the top level object, so the following “interface” functions let you make calls from the top level object “thru” the driver object to the TV terminal sub-object. All this maneuvering is due to the fact that SPIN has no support to access functions of a sub-object’s, sub-objects. In any event, the TV terminal itself is slightly modified from the original shipped with the HYDRA. I had to add a couple cursor positioning functions to make text printing easier. The new driver is named **TV_TERMINAL_011.SPIN** and is located on the CD here:

CD_ROM:\sources\TV_TERMINAL_011.SPIN

The TV terminal driver includes sub-objects **TV_DRIVER_010.SPIN** as well as **GRAPHICS_DRV_010.SPIN**. Both of which are not accessible.

Now, at some point you may want to completely remove all the printing ability from the SD Max driver, when you do so these functions would be useless as well. But, for now, they allow both the SD Max driver to print to the screen as well as the top level program without incurring any more program memory overhead. The functions are simply wrappers that pass the parameters to the real TV driver’s functions. So please review the file **TV_TERMINAL_011.SPIN** for the actual code.

Function Prototype:

```
tvb_bin(value, digits)
```

Description:

Prints a number in binary 1's 0's format **value** to the terminal with specific number of **digits**.

Returns: N/A.

Example(s): Print %1111_0000 to the screen.

```
sd.tvb_bin(%1111_0000, 8)
```

Function Prototype:

```
tvb_out(char)
```

Description:

Outputs a single character **char** to terminal.

Returns: N/A.

Example(s): Print the entire ASCII character set from \$20 (space) to \$7E (~) to the screen.

```
repeat ch from $20 to $7E
sd.tvb_out( ch )
```

Function Prototype:

```
tvb_dec(value)
```

Description:

Prints a number **value** in decimal format to terminal.

Returns: N/A.

Example(s): Print the first Fibonacci numbers 1 to 25 to the screen, 1,1,2,3,5,...(fn-2 + fn-1)

```
fn := 1
fn_1 := 1
fn_2 := 0

repeat index from 0 to 24
  ' print the fib number to screen
  sd.tvb_dec ( fn )

  ' compute next Fibonacci number fn = fn-1 + fn-2
  fn := fn_1 + fn_2
  fn_2 := fn_1
  fn_1 := fn
```

Function Prototype:

```
tvb_hex(value, digits)
```

Description:

Prints the hex number **value** to the terminal with specific number of digits.

Returns: N/A.

Example(s): Print a 16-bit hex value to screen.

```
sd.tvt_hex( number, 4 )
```

Function Prototype:

tv_tsetx(new_x)

Description:

Sets the x cursor position on terminal. The terminal program is 40 characters by 24 rows. This function sets the cursor position, so the next print operation occurs at the x column location defined by **new_x**.

Returns: N/A.

Example(s): Print the first 8 powers of 2 in decimal along the top edge of the screen, assumes that the initial cursor position is at (0,0)

```
' initialize to 1, that's 2 ^ 0
powerof2 := 1

repeat x from 0 to 7
  ' print power of 2
  sd.tvt_dec( powerof2 )

  ' compute next power of 2
  powerof2 *= 2

  ' advance cursor
  sd.tvt_setx( x*5 )
```

Function Prototype:

tv_tsety(new_y)

Description:

Sets the y cursor position on terminal. The terminal program is 40 characters by 24 rows. This function sets the cursor position, so the next print operation occurs at the y row location defined by **new_y**.

Returns: N/A.

Example(s): Draw a diagonal line of "*" from top left to bottom right of screen.

```
repeat t from 0 to 20
  ' position cursor at (t,t)
  sd.tvt_setx( t )
  sd.tvt_sety( t )

  ' output character
  sd.tvt_out( '*' )
```

Function Prototype:

tv_getx

Description:

Returns the current x cursor position in terminal.

Returns: N/A.

Example(s): Move the cursor on the same row, but 5 columns to the right:

```
sd.setx( sd.getx + 5 )
```

Function Prototype:

```
tv_t_gety
```

Description:

Returns the current y cursor position in terminal.

Returns: N/A.

Example(s): Move up a single line:

```
sd.sety( sd.gety - 1 )
```

Function Prototype:

```
tv_t_pstring(string_ptr)
```

Description:

Prints the NULL terminated string pointed to by *string_ptr* to the terminal.

Returns: N/A.

Example(s): Print my name to the screen at cursor position (0,0):

```
' set cursor to (0,0)
sd.tv_t_setx ( 0 )
sd.tv_t_setx ( 0 )

' print my name
sd.tv_t_pstring( string ("Andre' LaMothe" ) )
```

Function Prototype:

```
tv_t_erase
```

Description:

Erases completely the character under the current cursor position.

Returns: N/A.

Example(s):

Print a diagnostic value on the screen at (0,0), but make sure the previous data is erased.

```
' erase characters
repeat x from 0 to 7
  sd.tv_t_setx(x)
  sd.tv_t_erase

' print new data
sd.tv_t_setx(0)
sd.tv_t_dec( value )
```

7.0 The HYDRA SD Max Demos

There are two main demos to explore the HYDRA SD Max Card. The *first demo* is the one that is pre-loaded on the SD Max card with the top file name **MENU_001.SPIN**. It's a menu loader program that accesses the SD card and allows you to load in a demo program by reading a binary EEPROM image of the demo into the RAM of the Propeller chip then

issuing a soft reset to the Propeller chip. This way you can play hundreds, even thousands of games with the single SD card. Currently, there are dozens of games and demos on the SD card in binary format. The majority of source code is available for the demos as well and can be found in the Parallax.com's HYDRA Forums Master Project link as well as in the "goodies" directory on the CD here:

CD_ROM:\goodies\user_programs

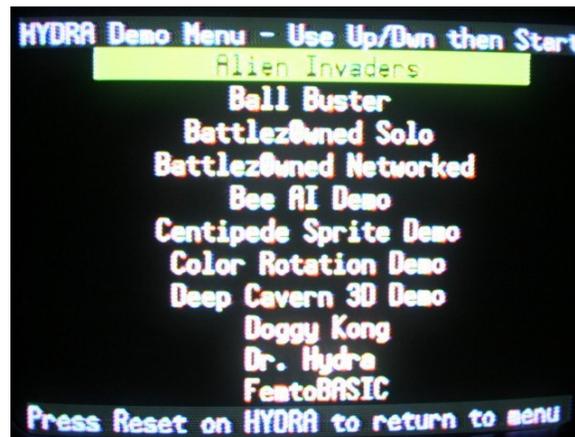
The **second demo** program is based on the SD Max driver developed in this manual and illustrates the use of all the SD card API commands in detail with a complete example that allows you to mount/unmount SD cards, display the directory, load files and print to the screen and more. The name of the demo is **SDMAX_DEMO_001.spin**, I suggest you use this to experiment with the SD Max API as a template to try things.

In the next sections, we will discuss the details of the demos, the source files, and so forth. But, before we start one very important thing about SD cards that I want to emphasize:

WARNING!

When working with SD cards, if you want to erase the SD card, its not enough to just erase the files, make sure to **FORMAT** the SD card for FAT16 ALWAYS. This way the SD card is in a clean state to work with. If you delete and write files over and over, sometimes the FAT16 drivers might not work right, so best to start with a freshly formatted SD card.

Figure 22.0 – The menu loader program in action.



7.1 The Menu Loader Program

The menu loader (as shown in Figure 22.0) was discussed in the **Quick Start** section of the manual and is the program that comes pre-loaded on the SD Max Card's EEPROM. The program displays a menu on the TV screen listing all the demos, games, and programs on the SD card. Then with the gamepad you can scroll up/down and load and execute any game by pressing the Start button on the controller. When you're done press the **Reset** button on the HYDRA to re-display the menu and try again.

Additionally, the SD card itself shipped with the SD Max Card comes pre-loaded with all the binary files for all the games. In this section, we are going to briefly talk about the loader program and how to experiment with it.

NOTE

The loader program is the combined work of many people since it uses some basic video drivers and so forth. But, the core of the code (the hard stuff) was written by **Mike Green** and **Tomas Rokicki** of **Radical Eye Software**. Tom originally wrote a little BASIC interpreter which Mike Green took and developed into "**FemtoBASIC**", this then turned into the platform that Tom and others have added on modules to do things like SPI, I2C, and FAT16. In any case, credit goes to Tom and Mike for developing the menu program, and their flavor of SPI, and FAT16 file system drivers.

7.1.1 The Menu Loader Software Architecture

The menu loader program consists of a main top level file called **MENU_001.SPIN**, its located on the CD here:

CD_ROM:\sources\MENU_001.SPIN

The menu loader pulls in a number of sub-objects directly or indirectly, they are:

FSRWFEMTO.SPIN	- FAT file system support.
SDSPIFEMTO.SPIN	- The SPI interface to the SD card.
TV_WTEXT.SPIN	- Windowed TV driver based on Parallax TV_TEXT.SPIN modified by Phil Pilgrim.
TV.SPIN	- The Parallax TV driver.
GAMEPAD_DRV_001.SPIN	- The Nurve gamepad driver.

If you want to use these driver for your SD card work then all you need are **FSRWFEMTO.SPIN** and **SDSPIFEMTO.SPIN** which gives you the high level FAT16 support and the low level SPI support need to talk to the SD card. These drivers are much faster since they use assembly language, but there are no docs other than the comments on “how they work”, but the source is well commented. So, if you need a more complete FAT16 system and need more speed these drivers are the way to go.

To burn the menu program back into the SD Max card simple load **MENU_001.SPIN** into the Propeller IDE and press F11.

7.1.2 Building the SD Card for the Loader

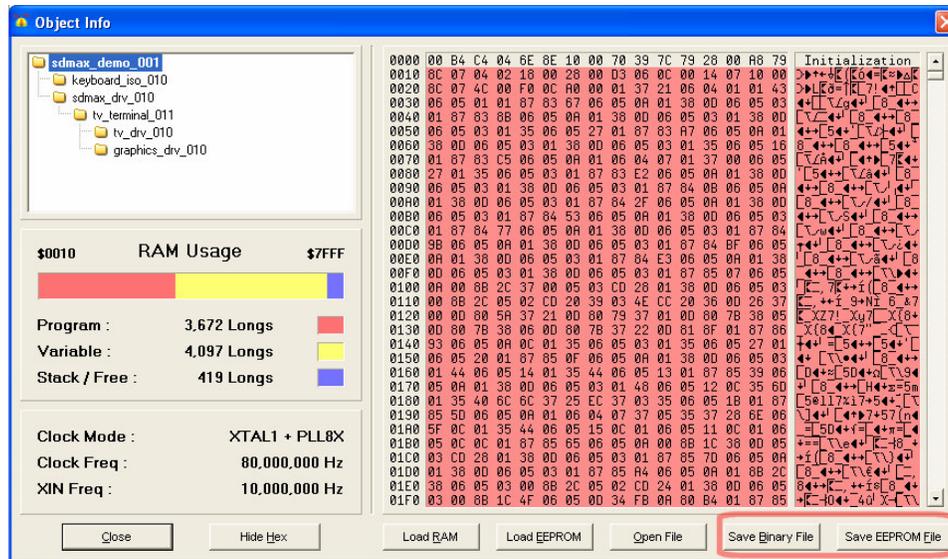
The menu loader program consists of SPI and FAT16 code, but the really cool part is its ability to load in a program from the SD card into the Propeller’s 32K and then soft boot to execute it. This isn’t that complex. All you need to do is run a program on a COG that can load the 32K SRAM externally and then call a boot on the Propeller. In theory, not difficult, but a bit tricky to code in practice (isn’t everything!). The **FSRWFEMTO.SPIN** file supports this functionality in you will see a call named **Execute(..)** in **MENU_001.SPIN** that calls a couple functions including **bootSDCard** from the driver. The code is shown below:

```
PRI Execute( filename )
  if \fsrw.mount(spiDO,spiClk,spiDI,spiCS) < 0
    abort string("Can't mount SD card")
  if \fsrw.popen( filename, "r" )
    abort string("Can't open file")
  fsrw.bootSDCard
  return 0
```

You will have to follow these functions back to the **FSRWFEMTO.SPIN** driver and read the source to figure out how they work, but this is the starting place to that.

In any event, it all works just fine. So the next question is, what format do the files on the SD card need to be in for the menu program to see them? Normally, you write your source code in ASCII text source files and then save the program’s as SPIN files. However, when you press F10 or F11, your code and all its sub-objects are compiled into a single 32K byte binary image. This is then downloaded to the Propeller chip, loaded into the 32K SRAM, then the Propeller boots the code. Thus, if we can create this 32K images from the Propeller tool then in theory with the right code running on the Propeller we should be able to “fake” the boot sequence and consequently load any image of SD (or any device) and then execute it from the Propeller chip. This will all work as long as we have a true Propeller 32K binary image.

Figure 23.0 – The Propeller IDE's Object Info Viewer Tool.



So how do we make these images? Well, in the Propeller IDE there is menu option called “**View Info**” its located under the Run menu here: < **Run** → **Compile Current** → **View Info** >. Or you can get to it by pressing F8. When you do so you will see a dialog box similar to that shown in Figure 23.0. This is called the Object Info viewer and shows the memory usage as well as the compiled program data to the right. Additionally, there are some interesting buttons to the bottom right:

- <Save Binary File> - Saves a binary of the compiled program.
- <Save EEPROM File> - Saves a 32K EEPROM image of the compiled program.

We are interested in the <Save EEPROM File> button which allows you to save the complete 32K image that represents your program. Even if you program is only 1K long the Propeller *always* makes a 32K EEPROM image. If you take the EEPROM image and copy into the HYDRA EEPROM then the Propeller would be able to read it and boot it. If you copy the binary into the EEPROM the HYDRA wouldn't know what to do. The binary is not an image for the EEPROM, but the actual short binary of the program code. This has to be loaded in a different way, alas, we are not interested in this.

Alright, so the trick is that for every program you want the menu loader to recognize you have to load the source into the Propeller IDE, compile it with F8 and then save it to the HD in EEPROM format. Then you will have a collection of 32K images that are directly bootable from the Propeller chip via the EEPROM.

Next, you would take all these EEPROM images and copy them to an SD card. But, the menu loader program needs a little help to know what you want the menu to look like, thus, after you copy all the *.EEPROM image files on the SD card, then you must create one more file – a directory file named **DIR.TXT** which is a standard ASCII readable text file that simply has your menu list in it that refers to the files on the SD card. Each entry in the DIR.TXT file must be on two lines and must look like this:

File Contents:

```
1st line of entry: Text You want to Display On the Menu Loader for this file
2nd line of entry: FILENAME.EEP
```

The first line is the text you want displayed, the second is the actual filename of the **EEPROM** image. Now there are a couple rules here; first the filenames **must** be in **8.3 format**, so if you have a really long file names for the EEPROM images on the SD card, that's fine, but when you refer to them in the SD Max driver or this one you have to convert them to 8.3. The best way to do this is to see the listing from the CMD shell, since there is an algorithm to shorten names. But, one thing is for sure the extension will always be .EEP rather than EEPROM.

NOTE

To see the exact names that FAT16 converts long files to launch a Command shell from Windows, CMD.EXE should do it. Then navigate to your SD card or wherever you have your long file names that are going to end up on the SD card that you must figure out the FAT16 name for in 8.3 format. Then type the command “**dir /x**”. This will run the directory listing command, but also show the short 8.3 versions of the filenames.

As an example, let's say that we have a game called **asteroids_demo_014.SPIN**, we compile the program and save the EEPROM to the SD card as **asteroids_demo_014.eeprom**. So far, so good, but we have to add an entry to the DIR.TXT directory file also on the SD card. But, we need the exact 8.3 filename? No problem, as outlined in the TIP, just request a directory listing of the SD card from the CMD shell and see what the same is. If you do this, you will find that the name will be something like:

ASTERO~1.EEP

Which is 8.3. Now, let's say that we want the loader to print on the screen "**Asteroids Demo Dude**". So we open up the **DIR.TXT** file on the SD card (or your SD card staging area) and then add these two lines exactly:

File Contents:

```
Asteroids Demo Dude
ASTERO~1.EEP
```

Obviously, above and below this, there might be numerous other menu names and file names. This is how you create images for the SD card menu loader program. The steps are:

- **Step 1:** Format the SD card initially FAT16 to clean it up.
- **Step 2:** Compile your demo into a .EEPROM file.
- **Step 3:** Copy the .EEPROM file onto the SD card.
- **Step 4:** Add an entry in the DIR.TXT file on the SD card with the human readable text that represents the file along with the filename in 8.3 format.

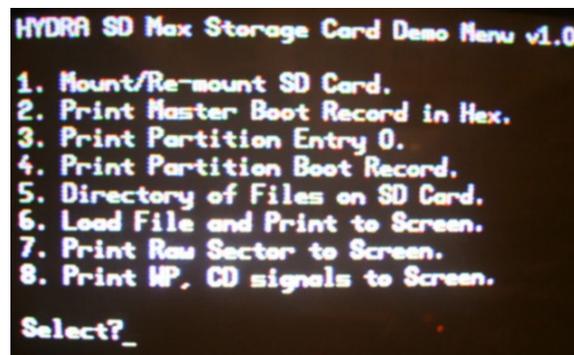
Of course, all this has been done for you already since the SD card that ships with the SD Max Storage Card already has all the EEPROM images and a **DIR.TXT** file on it. Also, if you damage the file system on the SD card and want to simply "Refresh" the games and DIR.TXT file back to their original state, format your SD card FAT16 then copy the contents of this directory from the CD to the SD card:

```
CD_ROM:\Sources\sd_menu_images \ *.*
```

NOTE

With a 2GB SD card you can hold approximately 65,000, 32K images! I think that's enough to hold a few HYDRA games and applications ☺

Figure 24.0 – The SD Max Storage Card Demo running.



```
HYDRA SD Max Storage Card Demo Menu v1.0
1. Mount/Re-mount SD Card.
2. Print Master Boot Record in Hex.
3. Print Partition Entry 0.
4. Print Partition Boot Record.
5. Directory of Files on SD Card.
6. Load File and Print to Screen.
7. Print Raw Sector to Screen.
8. Print MP, CD signals to Screen.
Select?_
```

7.2 The SD Max Demo API Program

The SD Max API demo is a simple program I put together to show off the bare minimum API code you need to mount SD cards, read the directory, open files, etc. The demo program is primarily made of the top level file itself located on the CD here:

CD_ROM:\sources\SDMAX_DEMO_001.SPIN

Which of course loads in the driver itself **SDMAX_DRV_010.SPIN** located on the CD here:

CD_ROM:\sources\SDMAX_DEMO_001.SPIN

The demo program also uses the TV, keyboard, and terminal drivers which are located in the same sources directory on the CD. Figure 24.0 shows the demo running, as you can see there are a number of menu options that will help you experiment with SD cards as well as give you some base code to experiment with, so you can add more options.

To truly learn about SD cards and the FAT16 file system there is no better way that to explore the MBR, PBR, FAT sectors and so forth. This demo will help you do that as well as show a model of how to call the API.

7.2.1 The Main Menu

The main menu lists 8 options for the program. Let's cover what each does.

1. **Mount/Re-mount SD Card** – Whenever you insert a new SD card into the SD max you must “mount” the SD card. Selecting this option, does just that.
2. **Print Master Boot Record in Hex** – This option prints out boot sector 0 in hex format, so you can investigate the data.
3. **Print Partition Entry 0** – This prints out the 16-byte partition entry 0 in hex and parses it for viewing.
4. **Print Partition Boot Record** - This prints out the partition boot record in human readable/non-binary form.
5. **Directory of Files on SD Card** – Simply lists the root directory of the SD card, shows the file names, sizes, etc.
6. **Load File and Print to Screen** – This function allows you to load a file and print it to the screen in decimal, hex, or ASCII format.
7. **Print Raw Sector to Screen** – This is a good diagnosis and investigation tool allowing you to load and display single sectors to the screen.
8. **Print WP, CD signals to Screen** – This prints the SD card's Write Protect and Card Detected signals to the screen for inspection.

That's about all there is to the menu program. When using the demo to experiment and see what's on your SD card, make sure that you always “mount” the SD card with **Option 1**, every time you remove/insert an SD card.

8.0 Summary

Well, we are at the end of this manual. If you read it all the way thru and knew nothing about SD cards, SPI communication, nor the FAT file system then hopefully you learned a lot. I suggest you experiment with the demo programs. The demo menu loader is a lot of fun and basically you can play hundreds of games on your HYDRA with a single card which is very cool. The SD Max API library should be more than enough to get you started. At some point we will add a FAT write file, but for now, it really is a good exercise to try and implement it just once in your lifetime. We take for granted all the technology around us, but few engineers or programmers these days every do any “heavy lifting”, they just connect things other people made and don't have a real appreciation for the technology from A to Z. Hopefully, this document has opened your eyes to the massive amount of work that went into the SD card and how all that work has such an impact on products, storage, and convenience for all of us.

Moving forward, some cool projects you might try implementing are a video or MP3 player for the HYDRA using the SD card. A simple audio recorder would be nice. The Parallax Object Exchange file repository has numerous recording and A/D converters that could be used for a continuous speech recorder system on the HYDRA. All you have to do is hook up a mic and a couple passive components to the free I/Os that aren't being used by other peripherals on the HYDRA.

If you do create some cool for the SD Max Card make sure to send it to us at support@nurve.net and we will include it with the next CD revision.

Good luck!

Appendices

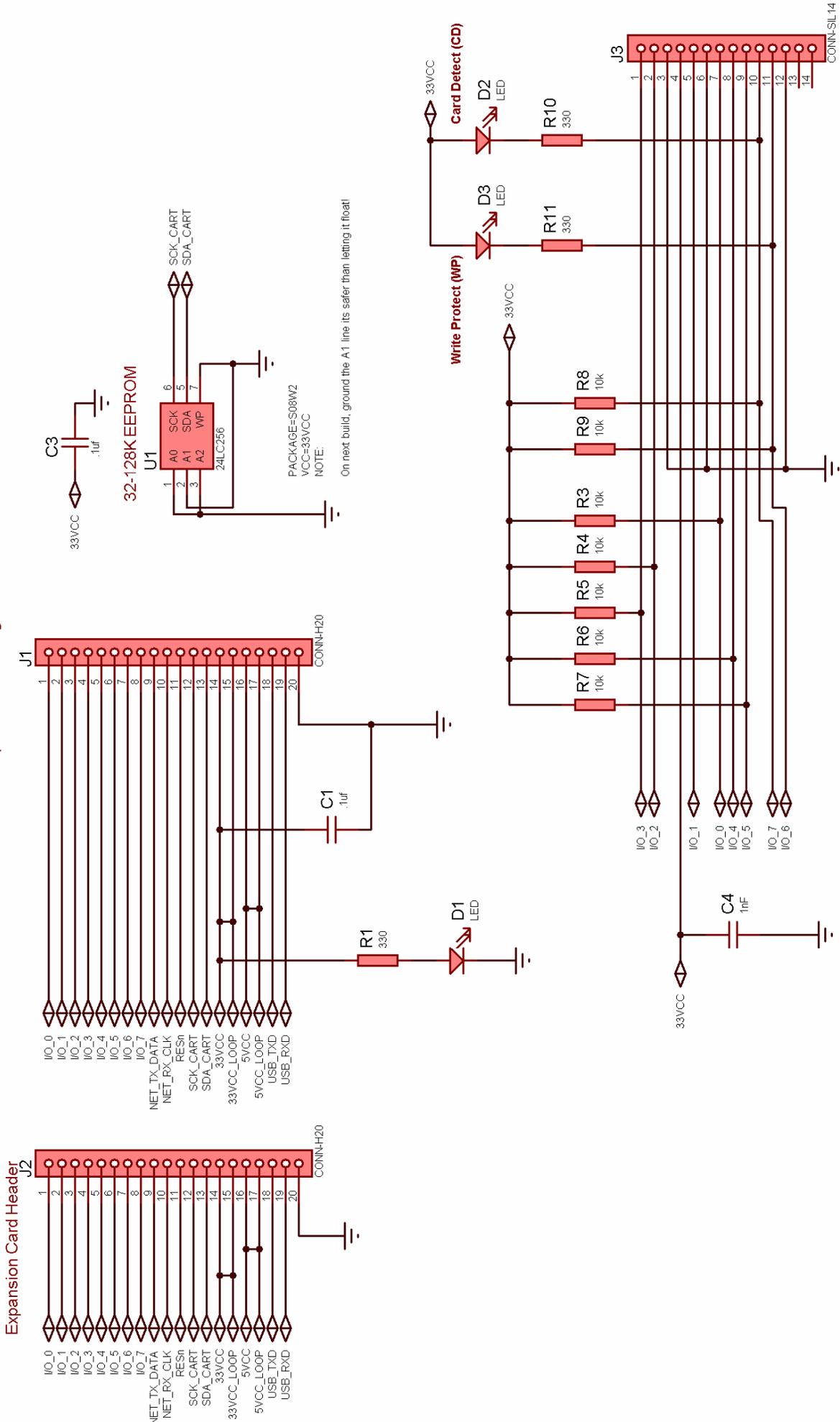
The following are a short list of appendices that cover the HYDRA SD Max schematics, PCB, and driver source code listing.

- Appendix A -HYDRA SD Max Schematic
- Appendix B - PCB Reference Layout
- Appendix C - API Driver Source Code Listing

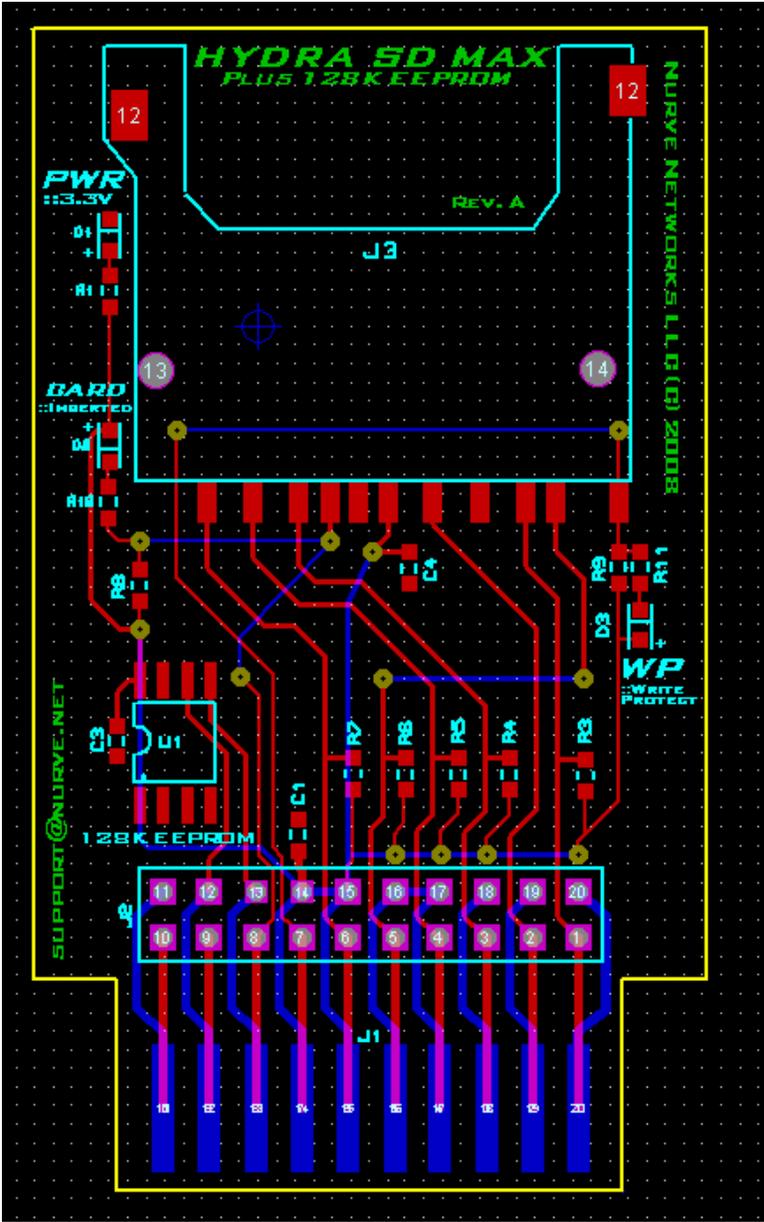
A. HYDRA SD Max Schematic

Revision A. (Next Page)

Expansion Card Edge Slot



B. PCB Reference Layout



Revision A. (Not actual size)

C. API Driver Source Code Listing

The following is a listing of the entire SD Max driver. Normally, I would never put a code dump like this in a printed book since it's a huge waste of paper, but since this is an eBook, it's nice to have the listing in the same document for reference.

```
{
///////////////////////////////////////////////////////////////////
FILENAME: SDMAX_DRV_010.SPIN
AUTHOR: Copyright 2008 Nurve Networks LLC, Andre' LaMothe
        www.xgamestation.com, support@nurve.net, Ph: 512.266.2399 / Fax: 512.266.0599
LICENSE: Free for personal and commercial use. You can freely distribute these drivers,
        bundle/modify them with your projects, use them in any way you like as long as Author
        and License information is intact. Software in whole or in part not to be sold
        in any way.
LAST MODIFIED: 1.xx.08
I. DESCRIPTION
This driver contains 3 layers of software to facilitate the interfacing of SD cards,
specifically the HYDRA SD MAX Storage Card to the HYDRA. However, the functions can be used with
any propeller chip interface simply by changing the SPI bus pins.
The 3 layers to communicate to the SD card are:
-----
| FAT16 (High Level - read DOS formatted FAT16 file system files on card) |
| SD Card Interface (Medium Level - initialize card, read, write sectors, etc.) |
| SPI Interface (Low Level - send and receive bytes directly to the SPI interface) |
-----
And of course sitting on top of the FAT16 layer is the application itself.
The driver consists of function calls to access all 3 layers of the model, so you can do
things like access the SD card as if it were a disk drive, print the directory, load files
of any size, etc. Or you might want to forget using DOS FAT16 format, and just read and write
sectors directly to the SD card, lastly, you might want to modify the SPI routines and code
them in assembly language.
Everything is written in SPIN for ease of use and review to see what's going on and the functions
are heavily commented, but please refer to the HYDRA SD MAX Storage Card manual for the complete
API listing as well as some general primers on FAT16, SD cards, and SPI programming. Also, I highly
recommend reading the reference links below and documents that are on the CD in the DOCS directory relating
to these subjects since they are tricky. Also, SPIN lacks any data structure support, so bare
with the use of arrays and pointers to get to everything. Also, since the propeller has such a small
amount of memory everything is cached a sector at a time to conserve working memory, this is good
for the memory, but makes the code messy since as a data structure is traversed and a sector boundary
is breached the next fragment of the data structure must be loaded. In a C/C++ program on a PC, you would
typically just load the entire data structure into a buffer and work with in place. However, one
approach is to think in terms of a virtual memory system on the propeller and create "smart" arrays
that you can access from 0 to size, then as you attempt to access data that is not in the local
cache then that data is loaded, so from the callers point of view the array is contiguous and always
there. This is worth the extra work if you continue to do a lot of "file system" programming with large
data structures.
SPI Interface Links:
http://www.vti.fi/midcom-serveattachmentguid-b469d17c67e60d4e3dbb25b0d099ad68/TN15\_SPI\_Interface\_Specification.pdf
SD Card Links:
http://www.cs.ucr.edu/~amitra/sdcard/ProdManualSDCardv1.9.pdf
http://www.sandisk.com/Assets/File/OEM/Manuals/SD\_SDIO\_specs\_v1.pdf
FAT16 Links:
http://www.maverick-os.dk/FileSystemFormats/FAT16\_FileSystem.html
http://www.compuphase.com/mbr\_fat.htm
II. API OVERVIEW
The following is a short list of the API functions with single line explanations. Refer to the CON and VAR sections
below for insight into the data structures used by the API.
*** Initialization Functions ***
PUB Start ( buffer_ptr ) - Initializes the driver and writes the callers pointer storage space with local disk buffer addresses
*** SPI Functions ***
PUB SPI_Init(mode) - Initializes the hardware SPI interface I/O pins from propeller to card.
PUB SPI_Send_Recv_Byte(spi_data8) - Sends a single byte to SPI interface and receives one at the same time.
PUB SPI_Read_Byte - Reads the SPI buffer, writes a dummy values of $FF.
PUB SPI_Write_Byte(spi_data8) - Writes a byte to the SPI interface.
*** SD Functions ***
PUB SD_Mount - Mounts the SD card by initializing and placing it into SPI mode.
PUB SD_Unmount - Unmounts the previously mounted SD card.
PUB SD_Send_Command( command8, address32 ) - Sends a generic command to the SD card.
PUB SD_Read_Sector(sector32, sectorbuffer16) - Reads a sector from the SD card.
PUB SD_Write_Sector(sector32, sectorbuffer16) - Writes a sector to the SD card.
PUB SD_Wait_Write_Complete - Internal function used in the write command to make sure the flash has been updated.
PUB SD_Print_Sector(sector32, sect_ptr, start_byte, end_byte, base, print_addr) - Diagnostic function that prints the contents of a sector to
terminal.
```



```

' MAIN ENTRY POINT //////////////////////////////////////
'////////////////////////////////////

PUB Start (buffer_ptrs)
' starts the SD driver up, does any global initialization that is needed (not much)
' also this function takes as an input a pointer to a receiver area that the function writes the local addresses of the
' sector buffers, so the calling object can use the memory and not create his own local buffers. the data structure
' is assumed to look like this from the callers point of view:
' long diskbuff_ptr ' generic disk sector buffer
' long mbrbuff_ptr ' master boot record buffer
' long pbrbuff_ptr ' partition boot record
'
' This function expects the address of the first pointer @diskbuff_ptr to be sent from caller, then the function will
' fill in each of the pointers and point them to the local memory buffers, so the caller can access them directly
'
' INPUTS: buffer_ptrs - pointer to starting address of diskbuff_ptr, mbrbuff_ptr, and pbrbuff_ptr in callers VAR space
' OUTPUTS: setups buffer_ptrs and returns $00 for success, $FF for failure
'-----
' set global debug level
debug_level := DEBUG_BRIEF

'start keyboard on pingroup 3, when doing local debugging
'key.start(3)

'start the tv terminal
term.start

' assign caller's pointers to local memory buffers, so caller can access them and use them
long [buffer_ptrs][0] := @diskbuff ' generic disk sector buffer
long [buffer_ptrs][1] := @mbrbuff ' master boot record buffer
long [buffer_ptrs][2] := @pbrbuff ' partition boot record
' end start

'////////////////////////////////////
' OBJECT EXPORTS
'////////////////////////////////////

' export TV terminal interface to higher level object, so caller doesn't have to instantiate TV terminal object
' crude hack, but what can you do!

PUB tvt_bin( value, digits )
term.bin( value, digits )

PUB tvt_out( char )
term.out( char )

PUB tvt_dec( value )
term.dec( value )

PUB tvt_hex( value, digits )
term.hex( value, digits )

PUB tvt_setx( new_x )
term.setx( new_x )

PUB tvt_sety( new_y )
term.sety( new_y )

PUB tvt_getx
return( term.getx )

PUB tvt_gety
return( term.gety )

PUB tvt_pstring( string_ptr )
term.pstring( string_ptr )

PUB tvt_erase
term.erase

'////////////////////////////////////
' FUNCTIONS
'////////////////////////////////////

' FAT16 DOS File System Interface
'////////////////////////////////////

PUB FAT_Print_Directory | sector_index, dir_index, data8, index, index2, hours_12, total_file_size, total_num_files

' DESCRIPTION: prints out the root directory entries to the terminal screen, see FAT_File_Open for explanation of how this works
' INPUTS: none
' OUTPUTS: returns the total number of files found
'-----

term.pstring(STRING("SD Card FAT16 Directory Listing..."))
term.out($0D)
term.out($0D)
term.out($0D)

' reset aggregate file size
total_file_size := 0
total_num_files := 0

repeat sector_index from first_dir_sector to (first_dir_sector + 32 - 1)
' read the sector in then traverse each directory entry, there are 16 entries
' per sector, once we find the file, or we find an empty entry then we can quit
SD_Read_Sector(sector_index, @diskbuff)
term.sety( term.gety - 1) ' HACK - for some reason everytime a sector is read, the inner loop is entered/exited another newline is printed!!!

' check each directory entry 0..15 in this sector
repeat dir_index from 0*32 to 15*32 step 32

' is this the last entry?
if ( diskbuff[dir_index] == $00)
' print final stats
' output file size and number of files
term.out($0D)
term.setx(7)
term.dec(total_num_files)
term.pstring(STRING(" File(s)"))
term.setx(9+7+4)
term.dec(total_file_size)
term.pstring(STRING(" Bytes"))

```

```

' return total number of entries
return(total_num_files )

' first file name, test first byte for deleted, empty, or first character of filename
elseif ( (diskbuff[dir_index] == $E5) or (diskbuff[dir_index] == $2E) or ( diskbuff[ dir_index + $0B ]==$0F ) or ( diskbuff[ dir_index + $0B ]==$08 ) )
' skip to next entry
next

' this is a valid entry, two special cases, volume label and root subdirectory

' now other parts of file
dir_attr := ( diskbuff[ dir_index + $0B ] )
dir_reserved := ( diskbuff[ dir_index + $0C ] )
dir_create_time_ms := ( diskbuff[ dir_index + $0D ] )
dir_create_hms := ( diskbuff[ dir_index + $0E ] ) + ( diskbuff[ dir_index + $0E + $01 ] << 8 )
dir_create_date_ymd := ( diskbuff[ dir_index + $10 ] ) + ( diskbuff[ dir_index + $10 + $01 ] << 8 )
dir_last_access_date_ymd := ( diskbuff[ dir_index + $12 ] ) + ( diskbuff[ dir_index + $12 + $01 ] << 8 )
dir_ea_index := ( diskbuff[ dir_index + $14 ] ) + ( diskbuff[ dir_index + $14 + $01 ] << 8 )
dir_last_update_time_hms := ( diskbuff[ dir_index + $16 ] ) + ( diskbuff[ dir_index + $16 + $01 ] << 8 )
dir_last_update_date_ymd := ( diskbuff[ dir_index + $18 ] ) + ( diskbuff[ dir_index + $18 + $01 ] << 8 )
dir_first_cluster := ( diskbuff[ dir_index + $1A ] ) + ( diskbuff[ dir_index + $1A + $01 ] << 8 )
dir_file_size := ( diskbuff[ dir_index + $1C ] ) + ( diskbuff[ dir_index + $1C + $01 ] << 8 ) + ( diskbuff[ dir_index + $1C + $02 ] << 16 ) + ( diskbuff[ dir_index + $1C + $03 ] << 24 )

' use these commented out sections to print out and format other details about the file entry that you might want to display
{
term.pstring(STRING("dir_attr:          "))
term.bin(dir_attr, 8)
term.out($0D)
}

{
term.pstring(STRING("dir_reserved:      "))
term.dec(dir_attr)
term.out($0D)
}

{
term.pstring(STRING("dir_create_time_ms:    "))
term.dec(dir_create_time_ms)
term.out($0D)
}

{
term.pstring(STRING("dir_create_hms:          "))
hours := (dir_create_hms & %1111_1000_0000_0000) >> 11
minutes := (dir_create_hms & %0000_0111_1110_0000) >> 5
seconds := (dir_create_hms & %0000_0000_0001_1111) << 1 ' resolution 2 seconds, so multiply by 2

term.dec(hours)
term.out(":")
term.dec(minutes)
term.out(":")
term.dec(seconds)
term.out($0D)
}

{
term.pstring(STRING("dir_create_date_ymd:    "))
year := 1980 + (dir_create_date_ymd & %1111_1110_0000_0000) >> 9
month := (dir_create_date_ymd & %0000_0001_1110_0000) >> 5
day := (dir_create_date_ymd & %0000_0000_0001_1111)

term.dec(month)
term.out("/")
term.dec(day)
term.out("/")
term.dec(year)
term.out($0D)
}

{
term.pstring(STRING("dir_last_access_date_ymd:"))
year := 1980 + (dir_last_access_date_ymd & %1111_1110_0000_0000) >> 9
month := (dir_last_access_date_ymd & %0000_0001_1110_0000) >> 5
day := (dir_last_access_date_ymd & %0000_0000_0001_1111)

term.dec(month)
term.out("/")
term.dec(day)
term.out("/")
term.dec(year)
term.out($0D)
}

{
term.pstring(STRING("dir_ea_index:          "))
term.dec(dir_ea_index)
term.out($0D)
}

' term.pstring(STRING("dir_last_update_date_ymd:"))
year := 1980 + (dir_last_update_date_ymd & %1111_1110_0000_0000) >> 9
month := (dir_last_update_date_ymd & %0000_0001_1110_0000) >> 5
day := (dir_last_update_date_ymd & %0000_0000_0001_1111)

term.dec(month)
term.out("/")
term.dec(day)
term.out("/")
if ((year-2000) < 10)
term.out("0")
term.dec(year-2000)
term.setx(9)
' term.pstring(STRING(" "))
' term.out($0D)

' term.pstring(STRING("dir_last_update_time_hms:"))
hours := (dir_last_update_time_hms & %1111_1000_0000_0000) >> 11
minutes := (dir_last_update_time_hms & %0000_0111_1110_0000) >> 5
seconds := (dir_last_update_time_hms & %0000_0000_0001_1111) << 1 ' resolution 2 seconds, so multiply by 2

' convert to am/pm
hours_12 := hours // 12
if (hours_12 == 0)

```

```

hours_12 := 12

term.dec(hours_12)
term.out(":")

' add a leading 0
if (minutes < 10)
  term.out("0")

term.dec(minutes)
term.out(" ")
term.out(":")
term.dec(seconds)
if ((hours / 12) == 0)
  term.pstring(STRING("AM"))
else
  term.pstring(STRING("PM"))

term.setx(9+9)
' term.pstring(STRING(" "))
' term.out($0D)

{
term.pstring(STRING("dir_first_cluster:      "))
term.dec(dir_first_cluster)
term.out($0D)
}

' test for directory entry eventhough we can't traverse, show root directories
if (dir_attr & $10)
  term.pstring(STRING("<DIR>"))
  term.setx(9+7+10)
else
  ' normal file
  ' update file tracking
  total_file_size += dir_file_size
  total_num_files++
  ' term.pstring(STRING("dir_file_size:      "))
  term.dec(dir_file_size)
  term.setx(9+7+10)
  ' term.pstring(STRING(" "))
  ' term.out($0D)

' print file name
repeat index from dir_index to dir_index+7
  ' test for non-space character
  if (diskbuff[index] <> $20)
    term.out(diskbuff[index])
  else
    quit

' test for extension
if (diskbuff[dir_index+8] <> $20)
  ' put dot for extension
  term.out(".")

  repeat index from dir_index+8 to dir_index+10
    if (diskbuff[index] <> $20)
      term.out(diskbuff[index])
    else
      quit

term.out($0D)

' end repeat dir_index
term.out($0D)

' end FAT_Print_Directory

' ////////////////////////////////////////////////////////////////////
PUB FAT_File_Open(filename_ptr, file_handle_ptr) | char, file_found, fnlength, findex, findex2, sector_index, dir_index, dir_entry, string_ptr1,
string_ptr2
' DESCRIPTION: scans the root directory and if the file pointed to by the NULL terminated filename filename_ptr
' is found then opens the file for reading and stores a data structure referring to the state of the file
' in file_handle_ptr which is points to a generic "file descriptor" data structure consisting of 4 longs
' in the following format:
'
' long first_cluster ' first cluster of file
' long filesize      ' size in bytes of file
' long curr_read_pos ' current read position
' long dir_entry     ' the directory index of the entry
'
' thus, the caller must create a zero'ed out 4 long storage area and pass the pointer to this function to work
' properly. This function is complex and the explanation is best read about in the user manual, but a brief
' overview is as follows: the FAT16 file system consists of a master boot record MBR (which would have already been read in)
' the MBR has partition entries that refer to each of the partitions on the drive (SD card in our case). Each partition entry
' points to the actual partition data of the logical drive, so first step is to load the MBR then located the first partition entry
' within it. Each entry is 16-bytes and the most important thing in the partition entry is the first sector of the partition along
' with the number of sectors from the MBR and the first sector of the partition (redundant more or less). Now, with the first
' sector of the partition, we load that sector in and it is the partition boot sector or volume boot sector or FAT16 boot sector,
' all mean the same thing. In the boot sector is all kind of information about the physical nature of the drive, however, in the
' case of SD cards, most of the information is irrelevant since there or no heads, tracks, etc. But, there are some entries like
' total number of sectors, bytes per sector, sectors per cluster that are needed for calculations and change depending on the size
' of the SD card. Next, up is the actual FAT data structure this is a linked list of clusters for each file, this is followed by the
' actual root DIRECTORY of files, each directory entry is 32-bytes and has name, length, etc. of each file. So there are a lot of data
' structures that have to be traversed, add to that, we only cache one sector at a time (512 bytes), and file access becomes quite
' a challenge on the propeller chip since not only is the memory limited, but SPIN has zero support for data structures, so everything
' in an array. In any event, to "open" a file, the following actions are taken; the root file directory is traversed, when/if the requested
' file name is found then the 32-byte file entry is accessed and the first cluster and size are returned in the sent
' "file handle" data structure. Interestingly, all we need is the size of the file and the first cluster to read it in since
' once we have this information then the FAT16 linked list is all we need. Of course, it might be prudent to also return the actual
' directory entry number, since once the file is opened other functions might refer to it, hence, the entry index is returned as well.
' The function FAT_Print_Directory works exactly as this function does, except it simply prints all valid entries
'
' INPUTS: filename_ptr - NULL terminated 8.3 filename to attempt to open (only in root directory)
'         file_handle_ptr - pointer to 4 longs where the file handle information will be stored if successful
' OUTPUTS: Returns length of file (which could be 0), or -1 if file not found.
'-----

' convert null terminated search filename into same format 8.3 with spaces padding right side
' this will make comparison less complex
' extract root filename

' prepare test filename with all spaces
bytefill(@tfilename, " ", 11)

```

```

tfilename[11] := 0 ' null terminate for compatibility

' compute length of filename being tested
fnlength := strsize( filename_ptr )

' parse filename into root and extension with padding
repeat findex from 0 to (fnlength-1)
  ' test for extension
  if (byte[filename_ptr][findex] == ".")
    repeat findex2 from (findex+1) to (fnlength-1)
      tfilename[(findex2 - (findex + 1)) + 8] := Toupper(byte[filename_ptr][findex2])
    quit
  else
    tfilename[findex] := Toupper(byte[filename_ptr][findex])

' set file not found
file_found := 0

' step 1: scan the directory for the file, see if it exists
' there are 512 directory entries, each entry is 32 bytes, thus, 32 sectors to
' hold the entire root directory, scan thru all of them

repeat sector_index from first_dir_sector to (first_dir_sector + 32 - 1)
  ' read the sector in then traverse each directory entry, there are 16 entries
  ' per sector, once we find the file, or we find an empty entry then we can quit
  SD_Read_Sector(sector_index, @diskbuff)

  ' check each directory entry 0..15 in this sector
  repeat dir_index from 0*32 to 15*32 step 32

    ' first file name, test first byte for deleted, empty, or first character of filename
    if (diskbuff[dir_index] == $00)
      ' last directory entry, no need to search any more entries or sectors
      ' exit, file not found
      ' set out loop past index to exit
      sector_index := first_dir_sector + 32
      ' break out of inner loop
      quit

    elseif ( ( diskbuff[dir_index] == $E5 ) or ( diskbuff[dir_index] == $2E ) or ( diskbuff[ dir_index + $0B ] == $0F ) or ( diskbuff[ dir_index + $0B ] == $08 ) )
      ' file deleted, folder entry, long filename, volume entry move on to next entry in inner loop
      next

    ' else non-empty, deleted, last, thus must be a real entry
    ' at this point, this is a file of some sort, figure out which?

    ' compare the file name to the entry and see if we have a match
    ' entry format 8.3: filename(8): FFFFxxx Extension(3): EEX
    ' where the names are LEFT justified and padded with spaces $20 (32) to the right.

    ' first compare filename
    if ( Strncmp( @tfilename[0], @diskbuff[dir_index], 11 ) == 1)
      ' filename found, exit out of loop

      ' set found flag
      file_found := 1

      ' compute absolute directory entry 0 .. 511
      dir_entry := (sector_index - first_dir_sector) * 32 + dir_index / 32

      sector_index := first_dir_sector + 32
      ' break out of inner loop
      quit

    ' end repeat dir_index

  ' end repeat sector_index

' did we find the file?
if (file_found == 0)
  ' file not found, return 0 for error, otherwise length of file
  if (debug_level == DEBUG_BRIEF)
    term.out($0D)
    term.pstring( STRING("File not found") )
    term.out($0D)
  return (-1)
else
  if (debug_level == DEBUG_BRIEF)
    if ($0D)
      term.pstring( STRING("File found at directory entry [") )
      term.dec(dir_entry)
      term.pstring( STRING("]") )
      term.out($0D)

    ' print filename
    repeat findex from 0 to 10
      term.out( Toupper( diskbuff[ dir_index + findex ] ) )
    term.out($0D)

  ' extract entire data structure incase we want it later
  dir_attr := ( diskbuff[ dir_index + $0B ] )
  dir_reserved := ( diskbuff[ dir_index + $0C ] )
  dir_create_time_ms := ( diskbuff[ dir_index + $0D ] )
  dir_create_hms := ( diskbuff[ dir_index + $0E ] ) + ( diskbuff[ dir_index + $0E + $01 ] << 8 )
  dir_create_date_ymd := ( diskbuff[ dir_index + $10 ] ) + ( diskbuff[ dir_index + $10 + $01 ] << 8 )
  dir_last_access_date_ymd := ( diskbuff[ dir_index + $12 ] ) + ( diskbuff[ dir_index + $12 + $01 ] << 8 )
  dir_ea_index := ( diskbuff[ dir_index + $14 ] ) + ( diskbuff[ dir_index + $14 + $01 ] << 8 )
  dir_last_update_time_hms := ( diskbuff[ dir_index + $16 ] ) + ( diskbuff[ dir_index + $16 + $01 ] << 8 )
  dir_last_update_date_ymd := ( diskbuff[ dir_index + $18 ] ) + ( diskbuff[ dir_index + $18 + $01 ] << 8 )
  dir_first_cluster := ( diskbuff[ dir_index + $1A ] ) + ( diskbuff[ dir_index + $1A + $01 ] << 8 )
  dir_file_size := ( diskbuff[ dir_index + $1C ] ) + ( diskbuff[ dir_index + $1C + $01 ] << 8 ) + ( diskbuff[ dir_index + $1C + $02 ] <<
16 ) + ( diskbuff[ dir_index + $1C + $03 ] << 24 )

  ' build up open file descriptor in following format, this is the minimum we need to keep track of the open file
  ' long first_cluster ' first cluster of file
  ' long filesize ' size in bytes of file
  ' long curr_read_pos ' current read position
  long[ file_handle_ptr ] [0] := dir_first_cluster
  long[ file_handle_ptr ] [1] := dir_file_size
  long[ file_handle_ptr ] [2] := 0 ' set file pointer current pos to first byte of file
  long[ file_handle_ptr ] [3] := dir_entry ' the directory index of the entry

  ' return length of file (could be 0, which is valid)
  return ( dir_file_size )

```

```
' end FAT_File_Open

' ////////////////////////////////////////////////////////////////////
PUB FAT_File_Read(file_handle_ptr, buffer_ptr, count) | fat_sector, fat_offset, bytes_read, index, sector_offset, data_sector, num_bytes_copy,
next_cluster, prev_cluster, fp, fp_file_size, fp_curr_pos, fp_end_pos, fp_first_cluster, logical_sector, logical_cluster, physical_cluster,
cluster_index

' DESCRIPTION: this function reads count bytes from the file referred to by the file handle and stores them into the sent buffer
' if count is greater than the total number of bytes left in the file, or the current file pointer plus count is greater
' than the length of the file, the function will read to the end of the file and return the ACTUAL number of bytes read always
'
' This function is VERY VERY complex and the explanation is best read about in the user manual, but a brief
' overview is as follows: the FAT16 file system consists of a master boot record MBR (which would have already been read in)
' the MBR has partition entries that refer to each of the partitions on the drive (SD card in our case). Each partition entry
' points to the actual partition data of the logical drive, so first step is to load the MBR then located the first partition entry
' within it. Each entry is 16-bytes and the most important thing in the partition entry is the first sector of the partition along
' with the number of sectors from the MBR and the first sector of the partition (redundant more or less). Now, with the first
' sector of the partition, we load that sector in and it is the partition boot sector or volume boot sector or FAT16 boot sector,
' all mean the same thing. In the boot sector is all kind of information about the physical nature of the drive, however, in the
' case of SD cards, most of the information is irrelevant since there are no heads, tracks, etc. But, there are some entries like
' total number of sectors, bytes per sector, sectors per cluster that are needed for calculations and change depending on the size
' of the SD card. Next, up is the actual FAT data structure this is a linked list of clusters for each file, this is followed by the
' actual root DIRECTORY of files, each directory entry is 32-bytes and has name, length, etc. of each file. So there are a lot of data
' structures that have to be traversed, add to that, we only cache one sector at a time (512 bytes), and file access becomes quite
' a challenge on the propeller chip since not only is the memory limited, but SPIN has zero support for data structures, so everything
' in an array. In any event, after the file has been "opened" with a call to FAT_File_Open the file handle descriptor is used by this
' function as a locator of the actual file data cluster and its length. Since we know the first cluster and the length of the file we
' simply need to read in the data sector by sector of the file. Now, there are a myriad of problems with this. First, each sector is 512
' bytes, so file reads that are greater than 512 bytes will span a sector, also, file reads that straddle sectors will need to read multiple
' sectors. There are typically 32-64 sectors per cluster, which means a single cluster holds 32-64K bytes, thus if a file is larger than
' 32-64K bytes then more than one cluster will have to be accessed. This means accessing the FAT and traversing the linked list of clusters
' for the file as it passes cluster boundaries. This problem is further compounded by the fact that each FAT sector only holds 256 cluster links,
' thus if a file is greater than 256*cluster size this means that multiple FAT sectors have to be loaded!! All this ends up being a
' brutally painful experience to code in SPIN due to its lack of data structure support and of course the propeller has no memory,
' so everything is cached a sector at a time. Anyway, the trick to writing FAT16 (or 32) functions is to think in terms of logical and
' physical values, so you can write a routine that deals with bytes, then one that deals with sectors, then as the fragmented data structures
' on the SD (or disk) are needed they are buffered transparently for you, not fun, no matter how many times your code it!

' INPUTS:
file_handle_ptr - pointer to 4 longs of file handle previously opened with call to FAT_File_Open in following format:
'
'         long first_cluster ' first cluster of file
'         long filesize      ' size in bytes of file
'         long curr_read_pos  ' current read position
'         long dir_entry     ' the directory index of the entry
'
' buffer_ptr - pointer to memory where the data will be read, make sure its big enough!
' count      - number of bytes to read.

' OUTPUTS: Returns actual number of bytes read
-----

' make sure this is a valid file
if (file_handle_ptr == 0)
  if (debug_level == DEBUG_BRIEF)
    term.out($0D)
    term.pstring( STRING("FAT16 Error: NULL file handle.") )
  return (0) ' null pointer

' extract cluster
fp_first_cluster := long[ file_handle_ptr ][0]

' compute fat sector offset that cluster index exists in, each fat sector holds 256 entries (assuming a 512 byte sector)
' for example if the cluster index is from 0..255 then we simply load the first FAT sector, 1..511, then the 2nd FAT sector etc.
fat_offset := fp_first_cluster / 256

' compute current fat sector cached
fat_sector := first_fat_sector + fat_offset

if (debug_level == DEBUG_VERBOSE)
  term.pstring( STRING("fat offset = ") )
  term.dec(fat_offset)
  term.out($0D)

' test if first cluster valid
if ( fp_first_cluster < 2)
  if (debug_level == DEBUG_BRIEF)
    term.out($0D)
    term.pstring( STRING("FAT16 Error: Cluster not valid.") )
  return (0) ' file handle error

fp_file_size := long[ file_handle_ptr ][1]
fp_curr_pos  := long[ file_handle_ptr ][2]
fp_end_pos   := _Min( fp_curr_pos + count - 1, fp_file_size - 1)

' recomputed count now that its been bounded
count      := fp_end_pos - fp_curr_pos + 1
bytes_read := count

' test if file pointer is at end of file (-1 or => file_size), if so simply return, user needs to seek to the start
if ( (fp_curr_pos == -1) or (fp_curr_pos => fp_file_size))
  ' 0 bytes read
  if (debug_level == DEBUG_BRIEF)
    term.out($0D)
    term.pstring( STRING("End Of File.") )
  return(0)

' compute working variables to make reading file easier (if there is such a thing with FAT!)

' load the FAT sector that has the cluster link in it, we might have to load multiple FAT sectors to traverse entire FAT
if (debug_level == DEBUG_VERBOSE)
  SD_Print_Sector(fat_sector, @diskbuff, 0,64, 16, 0)
  SD_Read_Sector(fat_sector, @fatbuff)

' compute logical cluster, simply divide filepointer by size of sector, 0 based value
logical_sector := fp_curr_pos / pbr_bytes_per_sect

' now the logical_cluster, this means the 0 based cluster that refers to the cluster holding the first byte referred to
' by the current file pointer, fp_curr_pos
logical_cluster := logical_sector / pbr_sect_per_clust

if (debug_level == DEBUG_VERBOSE)
  term.out($0D)
  term.pstring( STRING("pbr_bytes_per_sect=") )
```

```

term.dec(pbr_bytes_per_sect)

term.out($0D)
term.pstring( STRING("pbr_sect_per_clust=") )
term.dec(pbr_sect_per_clust)

term.out($0D)
term.pstring( STRING("Logical sector=") )
term.dec(logical_sector)

term.out($0D)
term.pstring( STRING("Logical cluster=") )
term.dec(logical_cluster)
term.out($0D)
' key.newkey

' traverse FAT cluster array to find the physical cluster we are interested, a file is broken into clusters, which are each n contiguous sectors
' but the cluster need NOT be contiguous, thus we must traverse the FAT to find the actual physical cluster the data is in

' this is the first cluster, it points to the next cluster, etc.
prev_cluster := fp_first_cluster
next_cluster := fp_first_cluster

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("Finding 1st cluster of file...") )
term.out($0D)
' key.newkey

' traverse list
cluster_index := 0

repeat while (cluster_index < logical_cluster )
if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING(" Clust ") )
term.dec(prev_cluster)

' at this point we need to make sure we haven't crossed a FAT sector boundary since we are only caching one sector
' of the FAT which is 512 bytes, 2 bytes per entry, thus we need to test the desired cluster entry to determine if its
' still in the current FAT sector, if not, load the needed FAT sector, which COULD be non-linear and somewhere else
' if user has deleted / created a lot of files on the SD card
if ( (( prev_cluster / 256) + first_fat_sector) <> fat_sector )
' re-compute proper FAT sector and load to access next 256 FAT cluster entries
fat_offset := prev_cluster / 256

' compute current fat sector cached
fat_sector := first_fat_sector + fat_offset

' and finally load FAT sector into cache
if (debug_level == DEBUG_VERBOSE)
SD_Print_Sector(fat_sector, @diskbuff, 0,64, 16, 0)

SD_Read_Sector(fat_sector, @fatbuff)

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("Loading new FAT sector ") )
term.dec(fat_offset)
term.pstring( STRING(" at final sector ") )
term.dec(fat_sector)
term.out($0D)

' always index into the FAT buffer mod 256, we can do this since we know that we have the correct FAT sector already
next_cluster := word[ @fatbuff ][ prev_cluster // 256 ]

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING(" -> ") )
term.dec(next_cluster)

' test if this is last cluster
if (next_cluster == $FFFF)
next_cluster := prev_cluster
quit

' advance link to next cluster
prev_cluster := next_cluster

' increment logical cluster index
cluster_index++

' finally we have the physical cluster
physical_cluster := next_cluster

if (debug_level == DEBUG_VERBOSE)
term.out($0D)
term.pstring( STRING("Logical cluster ") )
term.dec(logical_cluster)
term.pstring( STRING(" at physical ") )
term.dec(physical_cluster)
term.out($0D)

' key.newkey

' ready to copy file to buffer, the idea here is to use the logical byte index to find the cluster, sector, and byte on the SD then copy it
' this will be done in sector size chunks for speed
fp := fp_curr_pos

repeat while (count > 0)

' compute the actual sector holding the data, cluster and file pointer need to be considered
data_sector := ((fp / SECTOR_SIZE) // pbr_sect_per_clust ) + (first_data_sector + ((physical_cluster-2) * pbr_sect_per_clust) )

if (debug_level == DEBUG_VERBOSE)
term.out($0D)
term.pstring( STRING("data sector=") )
term.dec(data_sector)
term.out($0D)
'key.newkey

' now read the sector in that the current file pointer is within
SD_Read_Sector(data_sector, @diskbuff)

' compute offset into current sector (this happens on first sector, last sector is a similar special case)
sector_offset := fp // SECTOR_SIZE

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("sector offset=") )
term.dec(sector_offset)
term.out($0D)

```

```

'key.newkey

' at most we can copy (SECTOR_SIZE - sector_offset) bytes since we can't cross a sector boundary without another sector read
' now copy the data from the sector to the output buffer, sector_size bytes at a time (max)
' test for special cases first
if ( (sector_offset + count) > SECTOR_SIZE)
' data crosses a sector boundary, copy the remainder of the sector
num_bytes_copy := (SECTOR_SIZE - sector_offset)

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING(" > sector size ") )
term.out($0D)
'key.newkey
else
' data doesn't cross another sector boundary, this is it
num_bytes_copy := count

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING(" < sector size ") )
term.out($0D)
'key.newkey

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("num_bytes_copy=") )
term.dec(num_bytes_copy)
term.out($0D)
'key.newkey

' copy the bytes now
bytemove( buffer_ptr, @diskbuff + sector_offset, num_bytes_copy )

' print to screen
if (debug_level == DEBUG_BRIEF)
repeat index from 0 to num_bytes_copy - 1
term.out( byte [buffer_ptr][index] )

' update count and src/dest pointers
count -= num_bytes_copy
buffer_ptr += num_bytes_copy

' advance file pointer to next sector always
fp += num_bytes_copy

' test if we need to advance to next cluster, there are pbr_sect_per_clust sectors per cluster, if the fp is at a cluster
' boundary, then this expression will equal 0
if ( (fp // (pbr_bytes_per_sect * pbr_sect_per_clust)) == 0)
if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("Cluster boundary at fp = ") )
term.dec(fp)
term.out($0D)

' advance to next cluster in FAT chain
physical_cluster := word[ @fatbuff ][ physical_cluster // 256 ]

' NEW CODE -----
' now test if new physical cluster causes a FAT sector fault and we have to load in a new FAT sector!
if ( (( physical_cluster / 256) + first_fat_sector) <> fat_sector )
' re-compute proper FAT sector and load to access next 256 FAT cluster entries
fat_offset := physical_cluster / 256

' compute current fat sector cached
fat_sector := first_fat_sector + fat_offset

' and finally load FAT sector into cache
if (debug_level == DEBUG_VERBOSE)
SD_Print_Sector(fat_sector, @diskbuff, 0,64, 16,0)

SD_Read_Sector(fat_sector, @fatbuff)

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("Load new FAT sect ") )
term.dec(fat_offset)
term.pstring( STRING(" at final sect ") )
term.dec(fat_sector)
term.out($0D)
'key.newkey

' END NEW CODE -----

if (debug_level == DEBUG_VERBOSE)
term.pstring( STRING("Advancing to cluster = ") )
term.dec(physical_cluster)
term.out($0D)

'key.newkey

' write new file position
long[ file_handle_ptr ][2] := fp

if (debug_level == DEBUG_BRIEF)

term.out($0D)
term.pstring( STRING("fp = ") )
term.dec(long[ file_handle_ptr ][2])

term.pstring( STRING(" size = ") )
term.dec(long[ file_handle_ptr ][1])

term.out($0D)

' return number of bytes read
return ( bytes_read )

' end FAT_File_Read

' ////////////////////////////////////////
PUB FAT_File_Close(file_handle_ptr)

' DESCRIPTION: simply "closes" the file by zero'ing out the file handle data structure which once again looks like:
'
' file descriptor in following format
' long first_cluster ' first cluster of file
' long filesize ' size in bytes of file

```

```

' long curr_read_pos ' current read position
' long dir_entry    ' the directory index of the entry
'
' INPUTS: file_handle_ptr - pointer to file handle
' OUTPUTS: returns 1 if successful, 0 if failed or file handle NULL
-----
' test pointer, some modicum of error handling
if (file_handle_ptr == 0)
    return ( 0 ) ' there was a problem, this was null already
    else...
long[ file_handle_ptr ][0] := 0
long[ file_handle_ptr ][1] := 0
long[ file_handle_ptr ][2] := 0
long[ file_handle_ptr ][3] := 0
' return success
return ( 1 )
' end FAT_File_Close
'
' ////////////////////////////////////////////////////////////////////
PUB FAT_File_Seek(file_handle_ptr, count, mode)
' DESCRIPTION: this function updates the current read position of the sent file handle either
' with a new absolute value, or an offset to advance the pointer, controlled by mode (absolute or relative)
' file descriptor in following format as usual:
' long first_cluster ' first cluster of file
' long filesize      ' size in bytes of file
' long curr_read_pos ' current read position
' long dir_entry     ' the directory index of the entry
'
' INPUTS: file_handle_ptr - file handle to update
'         count          - value to set file position to
'         mode           - mode of operation relative (FILE_SEEK_REL) or absolute(FILE_SEEK_ABS)
'
' OUTPUTS: returns new file position
-----
' test file handle
if (file_handle_ptr == 0)
    return (-1)
' test which mode user is requesting
if (mode == FILE_SEEK_ABS)
    ' update current file position to the minimum of the requested value and the max size
    ' of file - 1
    long[ file_handle_ptr ][2] := _Min(count, long[ file_handle_ptr ][1] - 1)
elseif (mode == FILE_SEEK_REL)
    ' requesting relative mode, so add count to current position, test for overflow
    ' if so, set file read position to end of file
    long[ file_handle_ptr ][2] += count
    if ( long[ file_handle_ptr ][2] => long[ file_handle_ptr ][1])
        long[ file_handle_ptr ][2] := long[ file_handle_ptr ][1] - 1
elseif (mode == FILE_SEEK_END)
    ' set file pointer to last byte of file
    long[ file_handle_ptr ][2] := (long[ file_handle_ptr ][1] - 1)
else
    ' assume FILE_SEEK_START, set file pointer to first byte of file
    long[ file_handle_ptr ][2] := 0
' return new fileposition
return( long[ file_handle_ptr ][2] )
' end FAT_File_Seek(file_handle_ptr, count, mode)
'
' ////////////////////////////////////////////////////////////////////
PUB FAT_Print_Partition_Boot_Rec | index, data8
' DESCRIPTION: this function pretty prints the loaded partition boot record out in human readable form
' it assumes the partition information has been loaded already with a call to
' FAT_Load_Partition_Boot_Rec
'
' INPUTS: none
' OUTPUTS: prints to terminal
-----
term.pstring(STRING("Partition Boot Record Entry:"))
term.out($0D)
term.pstring(STRING("pbr_bytes_per_sect = "))
term.dec(pbr_bytes_per_sect)
term.out($0D)
term.pstring(STRING("pbr_sect_per_clust = "))
term.dec(pbr_sect_per_clust)
term.out($0D)
term.pstring(STRING("pbr_num_res_sect = "))
term.dec(pbr_num_res_sect)
term.out($0D)
term.pstring(STRING("pbr_num_fats = "))
term.dec(pbr_num_fats)
term.out($0D)
term.pstring(STRING("pbr_num_root_dir_entries = "))
term.dec(pbr_num_root_dir_entries)
term.out($0D)
term.pstring(STRING("pbr_total_num_sect = "))
term.dec(pbr_total_num_sect)
term.out($0D)
term.pstring(STRING("pbr_num_sect_per_fat = "))
term.dec(pbr_num_sect_per_fat)
term.out($0D)
term.pstring(STRING("pbr_volume_label = "))
term.pstring(@pbr_volume_label)
term.out($0D)
' end FAT_Print_Partition_Boot_Rec

```

```

' ////////////////////////////////////////////////////////////////////
PUB FAT_Load_Partition_Boot_Rec(pbr_ptr) | index
' DESCRIPTION: this function reads in the partition boot record (aka volume record, fat boot record),
' this is different than the MBR master boot record in that THIS is the boot record/entry for a
' specific partition, in this case, the partition previously loaded in via the load FAT_Load_Partition_Entry
' call which simply loads one of the 4 possible partition entries in the MBR, each entry is 16 bytes, these
' partition entries help locate the actual partition boot record which has all the information for the physical
' partition. This function loads the data and then parses it apart and sets the global template variables
'
' INPUTS: pbr_ptr - pointer to the buffer to receive the partition boot record data
' OUTPUTS: assigns the global template variables pbr_* with the values found in the partition record
'-----
' read partition boot record, its located mbr_num_sect_to_part sectors from sector 0 (the MBR)
SD_Read_Sector(mbr_num_sect_to_part, pbr_ptr)

if (debug_level == DEBUG_BRIEF)
  term.pstring(STRING("Loading partition boot record from "))
  term.dec(mbr_num_sect_to_part)
  term.out($0D)

' now process record and parse out variables and store to globals
' extract partition boot record values into template for easy access
pbr_bytes_per_sect := ( byte[pbr_ptr][ $0B ] ) + ( byte[pbr_ptr][ $0B + $01 ] << 8 )
pbr_sect_per_clust := ( byte[pbr_ptr][ $0D ] )
pbr_num_res_sect := ( byte[pbr_ptr][ $0E ] ) + ( byte[pbr_ptr][ $0E + $01 ] << 8 )
pbr_num_fats := ( byte[pbr_ptr][ $10 ] )
pbr_num_root_dir_entries := ( byte[pbr_ptr][ $11 ] ) + ( byte[pbr_ptr][ $11 + $01 ] << 8 )

' check for 32MB partition?
if ( ( byte[pbr_ptr][ $13 ] == 0 ) and ( byte[pbr_ptr][ $13 + $01 ] == 0 )
  ' partition greater than 32M
  pbr_total_num_sect := ( byte[pbr_ptr][ $20 ] ) + ( byte[pbr_ptr][ $20 + $01 ] << 8 ) + ( byte[pbr_ptr][ $20 + $02 ] << 16 ) + ( byte[pbr_ptr][
$20 + $03 ] << 24 )
  else
  ' partition less than 32M
  pbr_total_num_sect := ( byte[pbr_ptr][ $13 ] ) + ( byte[pbr_ptr][ $13 + $01 ] << 8 )

pbr_num_sect_per_fat := ( byte[pbr_ptr][ $16 ] ) + ( byte[pbr_ptr][ $16 + $01 ] << 8 )
pbr_num_vol_id := ( byte[pbr_ptr][ $27 ] ) + ( byte[pbr_ptr][ $27 + $01 ] << 8 ) + ( byte[pbr_ptr][ $27 + $02 ] << 16 ) + ( byte[pbr_ptr][
$27 + $03 ] << 24 )

bytemove( @pbr_volume_label, @pbrbuff[ $2B ], 11)
pbr_volume_label[11] := 0 ' null terminate

' convert any non-printable characters to ascii "."
repeat index from 0 to 10
  if (pbr_volume_label[index] < $20 or pbr_volume_label[index] > $7E)
    pbr_volume_label[index] := "."

' compute location of relevant data FAT directory structures
first_fat_sector := mbr_num_sect_to_part + pbr_num_res_sect
first_dir_sector := mbr_num_sect_to_part + pbr_num_res_sect + pbr_num_fats*pbr_num_sect_per_fat
first_data_sector := mbr_num_sect_to_part + pbr_num_res_sect + pbr_num_fats*pbr_num_sect_per_fat + (pbr_num_root_dir_entries / 16)

if (debug_level == DEBUG_BRIEF)
  term.out($0D)
  term.pstring(STRING("first_fat_sector = "))
  term.dec(first_fat_sector)
  term.out($0D)

  term.pstring(STRING("first_dir_sector = "))
  term.dec(first_dir_sector)
  term.out($0D)

  term.pstring(STRING("first_data_sector = "))
  term.dec(first_data_sector)
  term.out($0D)

' end FAT_Load_Partition_Boot_Rec
' ////////////////////////////////////////////////////////////////////
PUB FAT_Load_Partition_Entry(partition, mbr_ptr) | part_base
' DESCRIPTION: parses the sent mbr buffer and requested partition entry (0..3) and loads global variables
' with the partition entry, further FAT calls will use this data to locate files
' you can change to any partition at any time, by loading in another partition entry
' of course partition 1 us usually all that SD support
'
' INPUTS: partition - integer index of entry to load 0..3 (99% of the time will be 0)
' mbr_ptr - pointer to previously loaded master boot record
'
' OUTPUTS: parses and assigns global template variables mbr_*
' returns success 1, or failure 0
'-----
' test mbr_ptr
if (mbr_ptr == 0)
  return (0) ' failure

' compute base address of 16 byte partition record
part_base := $1BE + partition*16

' compute all globals associated with MBR

' boot descriptor (determines if partition entry is active, $80=active, $00=inactive), offset $00, 1-byte
mbr_boot_descriptor := ( byte[mbr_ptr][ $04 + part_base ] )

' first partition sector, offset $01, 1-byte
' 3-bytes in little endian format, build up LONG from bytes
mbr_first_part_sect := ( byte[mbr_ptr][ $01 + part_base ] ) + ( byte[mbr_ptr][ $01 + part_base ] << 8 ) + ( byte[mbr_ptr][ $01 + part_base ] << 16 )

' fat file system descriptor ($04 means fat16 system < 32MB, $06 means fat16 > 32MB, offset $4, 1-byte
mbr_file_sys_desc := ( byte[mbr_ptr][ $04 + part_base ] )

' last partition sector, offset $05, 3-bytes
' 3-bytes in little endian format, build up LONG from bytes
mbr_last_part_sect := ( byte[mbr_ptr][ $05 + part_base ] ) + ( byte[mbr_ptr][ $06 + part_base ] << 8 ) + ( byte[mbr_ptr][ $07 + part_base ] << 16 )

' number of sectors between the MBR and the first sector of the partition, offset $08, 4-bytes
' extract number of sectors to partition 1 data from MBR, 4-bytes in little endian format, build up LONG from bytes
mbr_num_sect_to_part := ( byte[mbr_ptr][ $08 + part_base ] ) + ( byte[mbr_ptr][ $09 + part_base ] << 8 ) + ( byte[mbr_ptr][ $0A + part_base ] << 16 )
+ ( byte[mbr_ptr][ $0B + part_base ] << 24 )

```

```

' return success
return (1)
' end FAT_Load_Partition_Entry
'////////////////////
PUB FAT_Read_MBR(mbr_ptr)
' DESCRIPTION: simply reads in sector 0 which is the MBR or master boot record, this could be done with the SD_Read_Sector
' function as well.
'
' INPUTS: mbr_ptr - sector pointer to where the data should be stored (one sector 512 bytes)
' OUTPUTS: returns $FF if there was an error or $00 if successful
'
' read the sector in and return result of read
return ( SD_Read_Sector(0, mbr_ptr) )
' end FAT_Read_MBR
'////////////////////
PUB FAT_Print_Partition_Entry(partition, mbr_ptr) | part_base, index, data8, _mbr_boot_descriptor, _mbr_first_part_sect, _mbr_file_sys_desc,
_mbr_last_part_sect, _mbr_num_sect_to_part
' DESCRIPTION: This function simply prints the 16 bytes that make up a partition entry in a tabular hex format "index:[value]"
' this function does NOT load the partition entry into the globals, it just prints them out for inspection
' typically SD cards only use the first partition entry, but the function support printing all 4 partition entries 0..3
' for completeness. This function is mostly for debugging purposes and not needed, it just helps when you are trying to understand
' what's in the partition entries and to print them to the screen in hex. You must have previously loaded the MBR (sector 0) into
' mbr_ptr before calling this function.
'
' INPUTS: partition - 0..3 the partition to print
'         mbr_ptr - pointer to local buffer holding mbr
'
' OUTPUTS: Just prints to screen for inspection.
'-----
' compute base address of 16 byte partition record
part_base := $1BE + partition*16
term.pstring(STRING("Partition entry "))
term.dec(partition)
term.pstring(STRING(" at address $"))
term.hex(part_base, 4)
term.out($0D)
' print out the requested partition entry, 16 bytes, starts at $1BE
repeat index from 0 to 15
term.hex(index,2)
term.pstring(STRING(":[")
data8 := byte[mbr_ptr][ index + part_base ]
term.hex (data8, 2)
term.pstring(STRING("]")
if ((index // 4)==3)
term.out($0D)
' now print out parsed data from partition entry in human readable form
' compute base address of 16 byte partition record
part_base := $1BE + partition*16
' compute some locals for printing, use same name as globals, but with underscores
' boot descriptor (determines if partition entry is active, $80=active, $00=inactive), offset $04, 1-byte
_mbr_boot_descriptor := ( byte[mbr_ptr][ $04 + part_base ] )
' first partition sector, offset $01, 1-byte
' 3-bytes in little endian format, build up LONG from bytes
_mbr_first_part_sect := ( byte[mbr_ptr][ $01 + part_base ] ) + ( byte[mbr_ptr][ $01 + part_base ] << 8 ) + ( byte[mbr_ptr][ $01 + part_base ] << 16 )
' fat file system descriptor ($04 means fat16 system < 32MB, $06 means fat16 > 32MB, offset $4, 1-byte
_mbr_file_sys_desc := ( byte[mbr_ptr][ $04 + part_base ] )
' last partition sector, offset $05, 3-bytes
' 3-bytes in little endian format, build up LONG from bytes
_mbr_last_part_sect := ( byte[mbr_ptr][ $05 + part_base ] ) + ( byte[mbr_ptr][ $06 + part_base ] << 8 ) + ( byte[mbr_ptr][ $07 + part_base ] << 16 )
' number of sectors between the MBR and the first sector of the partition, offset $08, 4-bytes
' extract number of sectors to partition 1 data from MBR, 4-bytes in little endian format, build up LONG from bytes
_mbr_num_sect_to_part := ( byte[mbr_ptr][ $08 + part_base ] ) + ( byte[mbr_ptr][ $09 + part_base ] << 8 ) + ( byte[mbr_ptr][ $0A + part_base ] <<
16 ) + ( byte[mbr_ptr][ $0B + part_base ] << 24 )
term.pstring(STRING("mbr_boot_descriptor = "))
term.hex(_mbr_boot_descriptor, 2)
term.out($0D)
term.pstring(STRING("mbr_first_part_sect = "))
term.dec(_mbr_first_part_sect)
term.out($0D)
term.pstring(STRING("mbr_file_sys_desc = $"))
term.dec(_mbr_file_sys_desc)
term.out($0D)
term.pstring(STRING("mbr_last_part_sect = "))
term.dec(_mbr_last_part_sect)
term.out($0D)
term.pstring(STRING("mbr_num_sect_to_part = "))
term.dec(_mbr_num_sect_to_part)
term.out($0D)
' end FAT_Print_Partition_Entry
'////////////////////
'////////////////////
' SPI (serial peripheral interface) interface code
'////////////////////
'////////////////////
PUB SPI_Init(mode)

```

```

' DESCRIPTION: this function initializes the hardware interface to the SPI bus. on the HYDRA SD MAX card
' (as with most SPI buses) there are only 4 signals; clk, data in, data out, and chip select

' INPUTS: mode - controls the phase of the clock and the logic levels
' mode 0 - default support
' mode 1 - not supported
' mode 2 - not supported
' mode 3 - not supported
'
' OUTPUTS: none
'-----
' set directions of SPI interface, be careful not to clock the SPI interface accidentally
' also, data in and data out are referenced from TARGET, so DI means the input to the SPI device which would be
' an OUTPUT from the propeller
OUTA [ spiDI ] := 0 ' set output LOW
DIRA [ spiDI ] := 1 ' set "data in" to output

DIRA [ spiDO ] := 0 ' set "data out" to input

OUTA [ spiCLK ] := 0 ' set output LOW
DIRA [ spiCLK ] := 1 ' set "clock" to output

OUTA [ spiCS ] := 1 ' set output HIGH (de-select the device)
DIRA [ spiCS ] := 1 ' set "chip select" to output

' end SPI_Init

'//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
PUB SPI_Send_Recv_Byte(spi_data8) | spi_result8, index
' DESCRIPTION: this functions sends spi_data8 and receives spi_result8, remember all spi operations are circular, so
' when a byte is sent a byte is received at the same time, you can ignore the sent or received byte as you
' wish. For example, to do a read just send the dummy value $FF
' NOTE: assumes caller is controlling the chip select line
'
' INPUTS: spi_data8 - 8 bit data that is going to be sent
' OUTPUTS: returns the 8-bit data received at the same time
'-----
' reset result
spi_result8 := 0

' shift 8-bits of data out while shifting in 8-bits of data
' data is sent and receiving MSB to LSB, so bit 7,6,5,4,3,2,1,0
' assume SPI interface is in mode 0/1, later make more flexible
repeat index from 0 to 7
  ' place next bit on data in pin to device
  OUTA [ spiDI ] := ((spi_data8 & $80) >> 7)
  term.dec((spi_data8 & $80) >> 7)

  spi_data8 <<= 1 ' shift data to left and prepare next bit for transmission

' pulse the clock line
OUTA [ spiCLK ] := 1

  ' data is ready now on output line from device
  spi_result8 <<= 1 ' shift result to left
  spi_result8 |= INA [ spiDO ] ' read data bit and OR into result

' finish clock pulse
OUTA [ spiCLK ] := 0

' term.pstring(STRING("RECEIVED = "))
' term.hex(spi_result8, 2)
' term.out($0D)

' return result
return (spi_result8)
' end SPI_Send_Receive_Byte

'//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
PUB SPI_Read_Byte
' DESCRIPTION: reads a single byte from the SPI interface, simply a dummy call to SPI_Send_Recv_Byte
'
' INPUTS: none.
' OUTPUTS: returns the byte read from the SPI interface
'-----
return ( SPI_Send_Recv_Byte($FF) )
' end SPI_Read_Byte

'//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
PUB SPI_write_Byte(spi_data8)
' DESCRIPTION: sends a single byte to the SPI interface, returns the read value
'
' INPUTS: spi_data8 - 8-bit data to send
' OUTPUTS: returns the 8-bit data from the SPI interface
'-----
' return the data from the cyclic buffer regardless...
return( SPI_Send_Recv_Byte(spi_data8) )
' end SPI_write_Byte

'//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
SD (secure digital) card interface code
'//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
PUB SD_Read_WP
' DESCRIPTION: reads the single bit "write protect" signal on the SD card mechanical
'
' INPUTS: none
' OUTPUTS: returns write protect bit, 0=write enabled, 1=write protected
'-----
DIRA [ sdWP ] := 0 ' set to LOW for input (should be default)
return ( INA [ sdWP ] )

```

```

' end PUB_SD_ReadWP

PUB_SD_Read_CD
'DESCRIPTION: reads the single bit "card inserted" signal on the SD card mechanical
'
' INPUTS: none
'
' OUTPUTS: returns card inserted bit, 0=card inserted, 1=card not inserted
'-----
DIRA [ sdCD ] := 0 ' set to LOW for input (should be default)
return ( INA [ sdCD ] )
' end PUB_SD_Read_CD
'
' ////////////////////////////////////////////////////////////////////
PUB_SD_Send_Command( command8, address32 ) | response, index
'DESCRIPTION: This function is the main workhorse of the SD interface and is used to send commands
' to the SD card via the SPI interface. All SD commands consist of 6 bytes in the following format:
'
' Command id      (1-byte) This you get from the SD specification.
' 32-bit address (4-bytes) In big endian form.
' CRC result      (1-byte) Error checking byte.
'
' The address is in big endian form, that is high byte to low byte (opposite of PC and propeller)
' The cycle redundancy check byte is sent in CRC7 or CRC16 format for all transactions
' except status tokens. CRC is of course a method of error detection, but not
' correction. If a CRC fails to match then the data must be sent or requested again.
' the algorithms for CRC7 and CRC16 are well documented and are basically division algorithms
' where the CRCs are the remainders. In our case, we are always going to send $95 which
' is the CRC for the first command that will be sent that matters, after that CRCs are
' ignored in SPI mode, later add code to compute the CRC for kicks if you like.
' Also, note the use of SD_RESPONSE_LIMIT, this allows the command to be sent over and over until there is a response.
' but, if the attempts exceed this constant, there is something wrong with the SD card.
'
' INPUTS: command8 - 8-bit command from SD card specification refer to document "SD Card Specification" from SD Association.
'         address32 - 32-bit address in normal little endian format, function converts to big endian before transport
'
' OUTPUTS: returns status of command to SD card, typically $FF means error, but for each command response can be different
'-----
' first send command
SPI_Send_Recv_Byte(command8)
' now send 4-bytes of address in MS-Byte to LS-byte format
SPI_Send_Recv_Byte( (address32 >> 24) & $FF)
SPI_Send_Recv_Byte( (address32 >> 16) & $FF)
SPI_Send_Recv_Byte( (address32 >> 8) & $FF)
SPI_Send_Recv_Byte( (address32 >> 0) & $FF)
' finally the CRC
SPI_Send_Recv_Byte( $95 )
' now wait for response
response := $FF ' default response is $FF which means error or time out
repeat index from 0 to SD_RESPONSE_LIMIT
' read response by sending dummy data
response := SPI_Send_Recv_Byte( $FF )
' DEBUG CODE - send out number of iterations the response took to get back
debug_wait_res_num_iter := index
' test if we have a non-$FF response
if (response <> $FF)
' send dummy data to allow SD card to finish operation, needed after all commands
SPI_Send_Recv_Byte( $FF )
' return results
return ( response )
' if we got here then something bad happened!
return ( $FF )
' end SD_Send_Command(command8, address32)
'
' ////////////////////////////////////////////////////////////////////
PUB_SD_Print_Sector(sector32, sect_ptr, start_byte, end_byte, base, print_addr) | index, data8
'DESCRIPTION: prints out the requested sector from start_byte to end_byte inclusive in various base formats
' for debugging purposes. In decimal and hex modes the printing is done in a tabular form with the starting address
' of the row printed to the left, this address usually would be 0 based, but if the caller wants to offset
' it by a constant value, then print_addr is used for this purpose, print_addr is added to the
' current printed address that is displayed. Also if sector32 == -1, then the function prints the
' data already pointed to by sect_ptr
'
' INPUTS: sector32 - sector to print out, if -1 then function is used to pretty print the sent buffer
'         sect_ptr - buffer space that will be used by function to load sector to print it, will be overwritten
'         start_byte - 0 based start byte index to begin printing
'         end_byte - last byte index to print
'         base - numerical base to print in, 16 for hex, 10 for decimal format, 127 for ASCII
'         print_addr - used to offset the left column base address in decimal and hex tabular forms, set to 0 usually
'
' OUTPUTS: prints to terminal screen
'-----
' read sector into buffer
if (sector32 <> -1)
SD_Read_Sector(sector32, sect_ptr)
' else just print data pointed to by sect_ptr
' test for ASCII format, if so just print it
if (base == 127)
repeat index from 0 to (end_byte - start_byte)
' ascii format
term.out( byte[sect_ptr][ start_byte + index ] )
term.out($0D)
return
' print requested data out in tabular decimal or hex
repeat index from 0 to (end_byte - start_byte)
if ((index // 8)==0)

```

```

term.out($0D)

if (base == 16)
  term.pstring(STRING("$"))
  term.hex(print_addr + start_byte + index,4)
else
  term.dec(print_addr + start_byte + index)

term.pstring(STRING(": "))

' print data now
data8 := byte[sect_ptr][ start_byte + index ]

if (base == 16)
  ' print in hex format
  term.pstring(STRING("$"))
  term.hex(data8,2)
else
  ' decimal format
  term.dec(data8)

term.pstring(STRING(", "))

term.out($0D)

' ////////////////////////////////////////////////////////////////////
PUB SD_Read_Sector(sect32, sectorbuffer16) | response, index
' DESCRIPTION: this function reads a sector from the SD card into the sent sector buffer
'
' INPUTS: sect32      - 32-bit absolute sector number to read
'         sectorbuffer - 16-bit pointer to receive buffer for sector
'
' OUTPUTS: storage pointed to by sectorbuffer16 will be filled with byte data from sector
'         returns $00 if no errors, $FF if error
'-----

' send the read block command with the byte address computed by multiplying the
' requested sector by 512 (512 bytes per sector)
response := SD_Send_Command( SD_READ_SINGLE_BLOCK , sect32 * SECTOR_SIZE )

' test response, should be $00
if (response <> $00)
  return (response)

' wait for the SD card to fetch the data from the flash storage
repeat index from 0 to SD_RETRIEVE_SECTOR_LIMIT
  response := SPI_Send_Recv_Byte($FF)

  ' test for proper response code $FE, break out of loop
  if (response == $FE)
    quit

  ' // end repeat index loop

' return error if not responding
if (response <> $FE)
  return ($FF) ' there was an error

' now, read the data bytes into the sent buffer
repeat index from 0 to 511
  byte[sectorbuffer16][index] := SPI_Send_Recv_Byte($FF)

' next read in dummy data
SPI_Send_Recv_Byte($FF) ' ignore this data
SPI_Send_Recv_Byte($FF) ' ignore CRC value

SPI_Send_Recv_Byte($FF) ' finally 8 more clocks for the card to finish

' return $00, no errors code
return ($00)

' end SD_Read_Sector

' ////////////////////////////////////////////////////////////////////
PUB SD_write_Sector(sect32, sectorbuffer16) | response, index
' DESCRIPTION: send the write block command with the byte address computed by multiplying the
' requested sector by 512 (512 bytes per sector). Be CAREFUL, you can destroy the MBR etc. by
' writing and some SD cards can stop functioning, so know what you're doing if you use the write
' operation.
'
' INPUTS: sect32 - sector to write
'         sectorbuffer16 - pointer to sector to write (512 bytes)
'
' OUTPUTS: returns response code from write operation, $00 no error, $FF error
'-----

response := SD_Send_Command( SD_WRITE_BLOCK , sect32 * SECTOR_SIZE )

' test response, should be $00
if (response <> $00)
  return (response)

' send $FF and $FE
SPI_Send_Recv_Byte($FF)
SPI_Send_Recv_Byte($FE)

' now, write the bytes to card from buffer
repeat index from 0 to 511
  SPI_Send_Recv_Byte(byte[sectorbuffer16][index])

' next read in dummy data
SPI_Send_Recv_Byte($FF) ' ignore this data
SPI_Send_Recv_Byte($FF) ' ignore CRC value

' wait for write to complete
response := SD_wait_Write_Complete

SPI_Send_Recv_Byte($FF) ' finally 8 more clocks for the card to finish

' return response from wait call, $00 = no error, $FF = error
return (response)

' end SD_write_Sector

```

```

'//////////////////////////////////////////////////////////////////
PUB SD_wait_write_Complete | num_attempts, response
' DESCRIPTION: This function is called after a write operation to "wait" for the write to complete. As
' with any flash technology the write operation can be many times slower than read, thus you must wait for
' it to complete. The function waits SD_WRITE_SECTOR_LIMIT iterations by checking the status response from
' the SD card, if the proper response is given, then its assumed there was a failure.
'
' INPUTS: none, a previous sector write is assumed
'
' OUTPUTS: returns $00 for success, $FF for failure
'-----
num_attempts := 0
repeat while (num_attempts++ < SD_WRITE_SECTOR_LIMIT )
  ' wait for response
  response := SPI_Send_Recv_Byte($FF)

  ' looking for $05 for success, $0B or $0D problem
  if ( response == $05)
    quit
  elseif( response == $0B or response == $0D)
    return $FF

' test for took too long
if ( num_attempts => SD_WRITE_SECTOR_LIMIT)
  return $FF

' finally wait for first non-$00 response, then done
num_attempts := 0
repeat while( (num_attempts++ < SD_WRITE_SECTOR_LIMIT) and (SPI_Send_Recv_Byte($FF) == $00) )

' did it take too long to get a response
if ( num_attempts < SD_WRITE_SECTOR_LIMIT)
  return $00
else
  return $FF

' end SD_wait_write_Complete
'//////////////////////////////////////////////////////////////////
PUB SD_Unmount
' DESCRIPTION: unmounts the SD card, simply releases the SD card chip select line.
' In a more advanced file system there might be numerous data structure that are created when
' a file system is "mounted", but in our case there are none, so nothing to unmount!
'
' INPUTS: none
'
' OUTPUTS: none
'-----
if (debug_level == DEBUG_BRIEF)
  term.pstring(STRING("SD card unmounted"))
  term.out($0D)

' de-assert chip select
OUTA [ spiCS ] := 1
'//////////////////////////////////////////////////////////////////
PUB SD_Mount | index, index2, sd_response, cmd_num_attempts
' DESCRIPTION: This function mounts the SD card which is a fairly complex process consisting of a number of steps.
' 1. Power up.
' 2. SD card must be placed into "idle state"
' 3. Then taken out of idle state.
' 4. Next, the card is tested to make sure its an SD and not MMC.
' 5. Finally, if all passes then the card is ready to go and "mounted".
'
' INPUTS: none
'
' OUTPUTS: returns $00 for success, $FF for failure
'-----
if (debug_level == DEBUG_BRIEF)
  term.pstring(STRING("Mounting SD card.."))
  term.out($0D)

' de-assert chip select
OUTA [ spiCS ] := 1

' now clock the SD SPI interface a few times and let it initialize (send dummy data)
repeat index from 0 to SD_POWERUP_LIMIT
  SPI_Send_Recv_Byte( $FF )

' assert chip select
OUTA [ spiCS ] := 0

' test for verbose mode
if (debug_level == DEBUG_VERBOSE)
  term.pstring(STRING("Placing SD card into idle state.."))
  term.out($0D)

sd_response := $FF
cmd_num_attempts := 0
repeat while (sd_response <> $01)

  ' send SD command GO_IDLE_STATE = $40, which resets the SD card
  sd_response := SD_Send_Command( SD_GO_IDLE_STATE, 0 )

  ' test for verbose mode
  if (debug_level == DEBUG_VERBOSE)
    ' print response
    term.pstring(STRING("Response = "))
    term.hex(sd_response, 2)
    term.out($0D)

    term.pstring(STRING("Number of iterations = "))
    term.dec(debug_wait_res_num_iter)
    term.out($0D)

    term.pstring(STRING("Number of attempts = "))
    term.dec(cmd_num_attempts)
    term.out($0D)

```

```

' check if we have tried this long enough
if (++cmd_num_attempts > SD_RESPONSE_LIMIT)
  ' test for verbose mode
  if (debug_level == DEBUG_VERBOSE)
    term.pstring(STRING("SD card not responding to idle state command. "))
    term.out($0D)

  if (debug_level == DEBUG_BRIEF)
    term.pstring(STRING("Couldn't mount SD card!"))
    term.out($0D)

  return ($FF)

' test for verbose mode
if (debug_level == DEBUG_VERBOSE)
term.pstring(STRING("Bringing SD card out of idle state..."))
term.out($0D)

sd_response := $FF
cmd_num_attempts := 0
repeat while (sd_response <> $00)

  ' send SD command EXIT_IDLE_STATE = $41, which resets the SD card
  sd_response := SD_Send_Command( SD_EXIT_IDLE_STATE, 0 )

  ' test for verbose mode
  if (debug_level == DEBUG_VERBOSE)
    ' print response
    term.pstring(STRING("Response = "))
    term.hex(sd_response, 2)
    term.out($0D)

    term.pstring(STRING("Number of iterations = "))
    term.dec(debug_wait_res_num_iter)
    term.out($0D)

    term.pstring(STRING("Number of attempts = "))
    term.dec(cmd_num_attempts)
    term.out($0D)

  ' check if we have tried this long enough
  if (++cmd_num_attempts > SD_RESPONSE_LIMIT)
    ' test for verbose mode
    if (debug_level == DEBUG_VERBOSE)
      term.pstring(STRING("SD card not responding to exit idle state command. "))
      term.out($0D)

    if (debug_level == DEBUG_BRIEF)
      term.pstring(STRING("Couldn't mount SD card!"))
      term.out($0D)

    return ($FF)

  ' test for verbose mode
  if (debug_level == DEBUG_VERBOSE)
    term.pstring(STRING("Verify SD card mode (not a MMC)..."))
    term.out($0D)

  ' send SD command APP_CMD = $40 + $37
  sd_response := SD_Send_Command( SD_APP_CMD, 0 )

  if ( sd_response <> $00 )
    if (debug_level == DEBUG_BRIEF)
      term.pstring(STRING("Couldn't mount SD card!"))
      term.out($0D)

    return($FF)

  ' send SD command SEND_OP_CMD = $40 + $29
  sd_response := SD_Send_Command( SD_SEND_OP_CMD, 0 )

  if ( sd_response <> $00 )
    if (debug_level == DEBUG_BRIEF)
      term.pstring(STRING("Couldn't mount SD card!"))
      term.out($0D)

    return($FF)

  if (debug_level == DEBUG_BRIEF)
    term.pstring(STRING("SD card successfully mounted. "))
    term.out($0D)

  ' return success
  return($00)

' end SD_Mount

' ////////////////////////////////////////////////////////////////////
' ////////////////////////////////////////////////////////////////////
' General Utility Functions
' ////////////////////////////////////////////////////////////////////
' ////////////////////////////////////////////////////////////////////

PUB Toupper(char)

' DESCRIPTION: this function upcases the sent character, if its not lowercase character then it passes it thru
' INPUTS: char - the character to upcase
' OUTPUTS: the upcased character
' -----

if ( (char => 97) and (char =< 122))
  char -= 32
  return (char)

' ////////////////////////////////////////////////////////////////////

PUB Strncmp(string_ptr1, string_ptr2, length) | index, char

' DESCRIPTION: this function compares the two sent strings up to the nth character
' the compare is case insensitive
' INPUTS: string_ptr1 - 1st NULL terminated string to compare
'         string_ptr2 - 2nd NULL terminated string to compare
' OUTPUTS: returns 1 if strings equal, 0 if not (case insensitive)
' -----

repeat index from 0 to length-1
  characters are the same?
  if ( Toupper(byte[string_ptr1][index]) <> Toupper(byte[string_ptr2][index]) )

```

```

' strings differ, return failure
return( 0 )

' strings must be equal
return( 1 )
end Strncmp

'
' ///////////////////////////////////////////////////////////////////
PUB itoa(value, sptr) | i, z
' DESCRIPTION: converts the sent integer to an ASCII null terminated string
'
' INPUTS: value - positive or negative integer to convert
'         sptr - storage for converted string
'
' OUTPUTS: writes string to sptr and returns pointer to string as well
'-----

' test for negative
if (value < 0)
' add leading negative sign
-value
byte[ sptr++ ] := "-"

' set up largest divisor
i := 1_000_000_000
z~

' compute next digit
repeat 10
if value => i
byte[ sptr++ ] := value / i + "0"
value /= i
z~~
elseif z or i == 1
byte[ sptr++ ] := "0"
i /= 10

' return string
return (sptr)

' end itoa

' ///////////////////////////////////////////////////////////////////
PUB To_ASCII(inchar, replace_char)
' DESCRIPTION: converts input into a ASCII printable character, this macro function is
' is used to simply turn non-printable garbage to printable text, so the tv terminal doesn't
' get sent garbage.
'
' INPUTS: inchar - input character to test.
'         replace_char - if the input char isn't printable then replace with this character
'         (usually a dot or space, so you can see what changed)
'
' OUTPUTS: returns the converted character
'-----

' test the character for printable range
if (inchar < $20 or inchar > $7E)
inchar := replace_char
return(inchar)

' end To_ASCII

' ///////////////////////////////////////////////////////////////////
PUB _Min(a,b)
' DESCRIPTION: returns signed minimum of a,b
' INPUTS: a,b - values to be compared
' OUTPUTS: the minimum
'-----

if (a < b)
return a
else
return b

' ///////////////////////////////////////////////////////////////////
PUB _Max(a,b)
' DESCRIPTION: returns signed maximum of a,b
' INPUTS: a,b - values to be compared
' OUTPUTS: the maximum
'-----

if (a > b)
return a
else
return b

' ///////////////////////////////////////////////////////////////////
PUB Set_Debug_Level ( level )
' DESCRIPTION: this function sets the driver debug level
' INPUTS: level - level to set output at DEBUG_BRIEF, DEBUG_OFF, DEBUG_VERBOSE
' OUPUTS: none

' set global debug level
debug_level := level

' end Set_Debug_Level

' ///////////////////////////////////////////////////////////////////
' DATA SECTION
' ///////////////////////////////////////////////////////////////////

DAT
str_newline byte 10,13

```

NOTES