SanDisk SD Software Tool Kit User's Guide

This Manual covers the SD Device Driver and File System (SDDK-05)

SanDisk Corporation
140 Caspian Court
Sunnyvale, CA 94089
TEL: 408-542-0500 FAX: 408-542-0503
URL: http://www.sandisk.com

Lit. No. 80-36-00144 Rev. 1 2001 *Printed in U.S.A.*

Revision History
- *Revision 1 – initial release.*

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

1.0 Overview

The Secure Digital (SD) Device Driver and File System (SDDK-05) Software Development Kit (SDK) is an integrated solution for managing high-level data on flash storage devices. The SDDK-05 requires minimal memory resources for both ROM and RAM while maximizing system performance.

Once integrated into an application or operating system, the SDDK-05 provides full File System functionality to manage data on storage devices. The SDDK-05 is a FAT (File Allocation Table) File System, fully compatible with DOS operating systems. The media is interchangeable between many DOS and Windows operating environments.

Also, in many embedded applications where the File System is not needed, the SDDK-05 can provide a way to access the Flash storage devices directly through its low level drivers.

1.1 Features

The SDDK-05 offers the following features:

- Full FAT File System interface with Application Programmer's Interface (API) functions such as create, delete, insert, merge files, sub-directories, file date/time, file attributes and volume labels.
- The FAT File System is optional and can be removed.
- Support for FAT12 and FAT16.
- Supports long and short file names.
- Single or multiple socket compatibility.
- Removable or fixed media support.
- Absolute sector access supported.
- Tunable options for different target environments.
- Full functionality on systems where byte access is not permitted.
- Distribution of C source code for specific environment.
- Extensive examples.

1.2 Target Applications

- Customizable to all CPUs
  - 8-bit, 16-bit and 32-bit processors supported
  - Little-endian and big-endian integer formats supported
  - TI DSPs supported
- Customizable to all ANSI-C or C++ compilers
- Customizable to all socket adapters and controllers
  - Memory or I/O mapped base supported
  - Multiple sockets and multiple adapters supported

1.3 Customization

The SDDK-05 is distributed in C source code format and intended to be customized on the target application. Documentation and examples are provided to guide all aspects of the customization process such as:

- Target CPUs
- Compilers
- Peripheral controllers: Serial Peripheral Interface (SPI) and SecureDigital/MultiMediaCard (SD/MultiMediaCard)
- System hardware such as, interrupts, timer, user interaction
- Parameters and functionality suitable for the target application

2.0 Introduction

The SanDisk SDDK-05 contains everything developers need to integrate SanDisk flash data storage products into any platform. The SanDisk SDDK-05 provides a native FAT File System and a stand-alone low level Peripheral Bus device driver. Platforms with limited software support can take advantage of this drop-in software component which adds complete disk subsystem functionality to the system. For systems that do not need the FAT File System, the Peripheral Bus device driver offers complete low level I/O access to the SanDisk flash products.

2.1 Components

The SDDK-05 includes the following components:

- API —Similar to POSIX/UNIX/DOS, this easy-to-use interface links the FAT File System or the Peripheral Bus Interface and the host's application software. It manages all aspects of storing and retrieving files, using a SanDisk device driver to perform low level I/O.
- FAT File System—This fully functional, DOS compatible (FAT) file system is contained in a portable 'C' source code library. It is re-entrant, and provides disk directory management as well as high performance file I/O. This File System is optional and can be removed.
- Sample Programs—These clearly demonstrate the use of all APIs. Moreover, because these programs are more sophisticated than just samples, they can serve as the basis for application development, in most cases.
- Source Code—Complete, highly portable, 'C' source code is provided for the entire SDDK-05 with a selected peripheral bus.
- SPI Device Driver —This device driver handles all low-level I/O access to SanDisk SD/MultiMediaCard storage products in SPI mode.
- SD/MultiMediaCard Device Driver —This device driver handles all low-level I/O access to SanDisk SD/MultiMediaCard storage products in SD/MultiMediaCard mode.

The SDDK-05 is designed with the flexibility to be configured as a stand-alone low level driver or used with the FAT File System. It is divided into these areas:

- FAT File System
- SanDisk API
- System Specific Section
  - System Enhancement Layers
  - System Abstraction Layers
  - Device Specifics Layers

Figure 2-1 shows how these modules communicate with each other, depicting a high level architectural view of the SDDK-05.

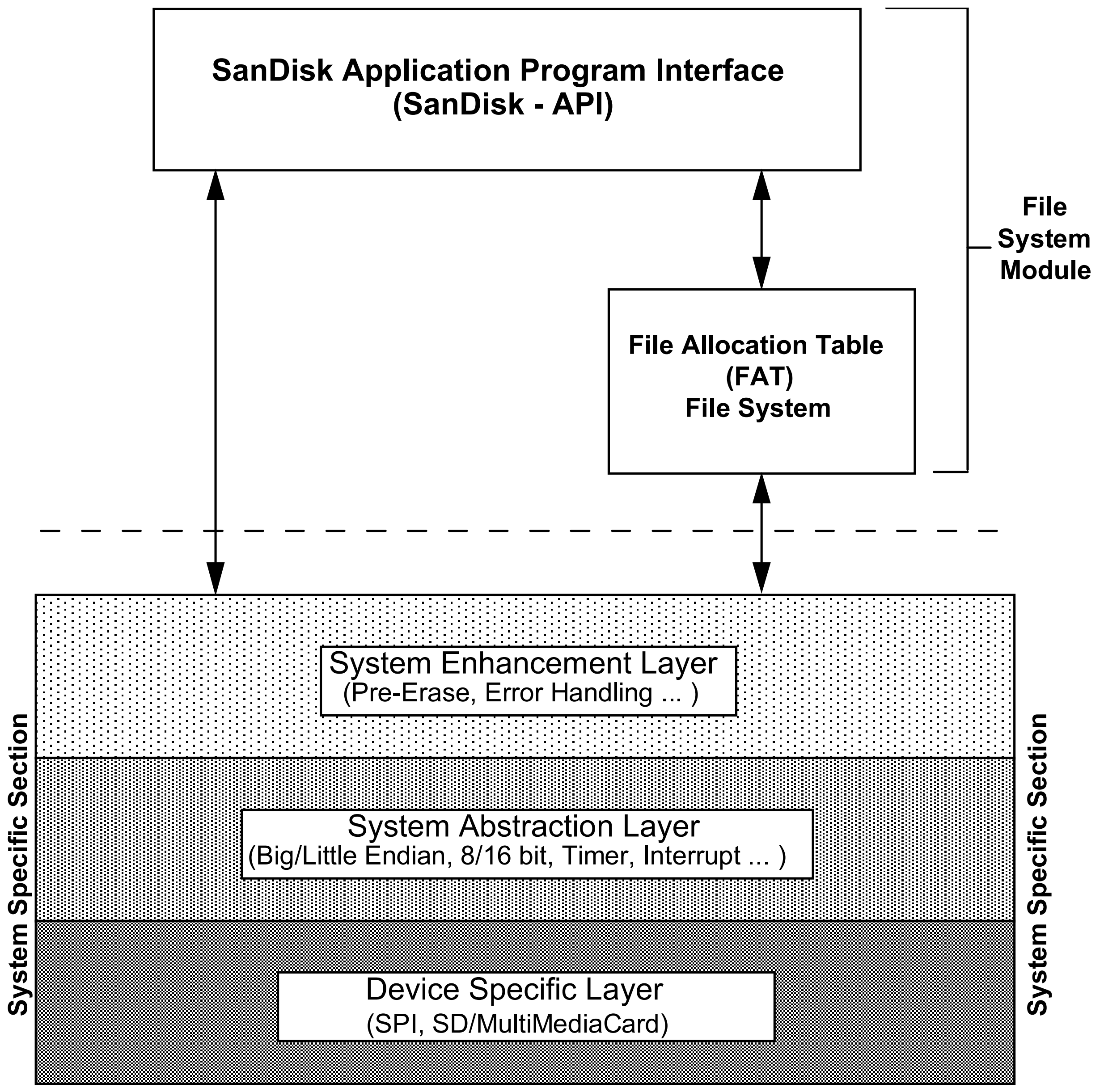

Figure 2-1 SDDK-05 Block Diagram

2.1.1 File System

The File System module is the highest level that manipulates data on the storage device. It communicates with the System Specific Section through the software layers contained in the Interface and Platform modules listed in Figure 3-1. Because the SDDK-05 is very flexible, designers can remove the FAT File System from the File System Module and incorporate a different File System for use with the low level peripheral interface driver.

2.1.2 API (Application Programmer's Interface)

The API provides a way for an application to communicate with flash devices through defined routines in the SDDK-05. This function set provides complete access to the flash device from the high level File System to the low level hardware driver.

2.1.3 System Specifics

The System Specific Section is divided into three layers:

- **Device Specific Layer** - Contained in the Interface Module (shown in Figure 3-1). This layer isolates product and defines mode selections (i.e. SD, SPI or MultiMediaCard). The Device Specific Layer provides the low level device driver that directly accesses the storage devices. It can access the platform module to perform system specific tasks.
- **System Abstraction Layer** - contained in the Platform Module (shown in Figure 3-1). This layer hides system specifics and provides ease of portability into various host target environments (i.e. processors, compilers). The System Abstraction Layer holds all system-specific routines such as interrupts, timer, error handler, hardware abstraction layer, compilation tools, etc.
- **System Enhancement Layer** - contains added features such as Critical Error Handling, High Performance Pre-Erase, etc.

3.0 SDDK-05 Source Directories

To better understand the SDDK-05 design, the overall structure and software modularity of the SDDK-05 are presented as a source tree structure. This source tree shows a list of all the files in a general view of the SDDK-05.

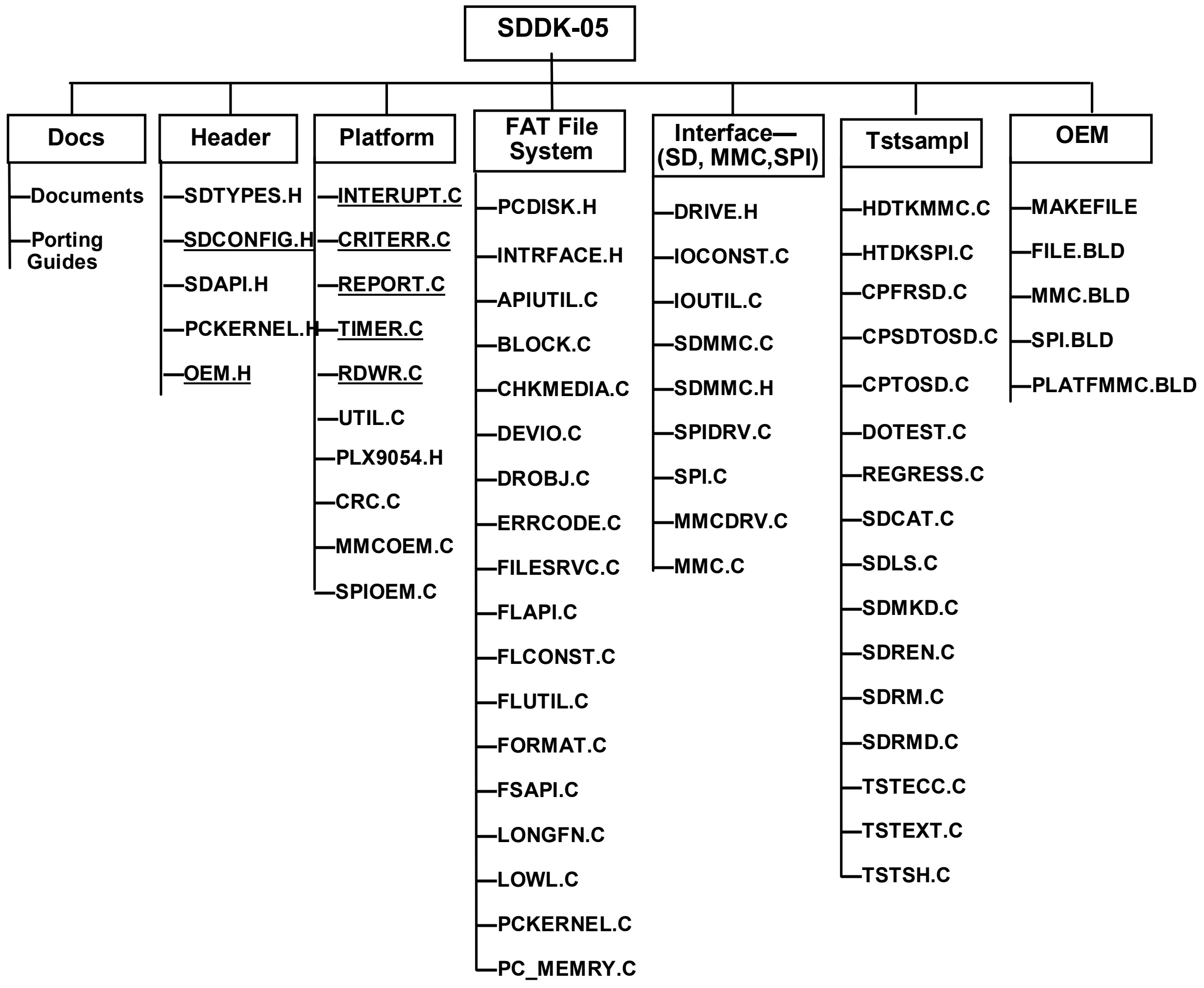

Figure 3-1 SDDK-05 Source Tree

Note: Files with underlined file names may need to be modified during the porting process.

The files associated with the directories are described below.

The **Docs** directory includes general documentation and specific porting guides.

The **Header** directory contains the following files:

SDTYPES.H	Data Type definitions.
SDCONFIG.H	Environment tuning configuration options.
SDAPI.H	File System and Peripheral Bus API.
PCKERNEL.H	Support multitasking routines.
OEM.H	OEM specific routines.

The **Platform** directory contains the following files:

INTERUPT.C	Interrupt service routine. (OEM specific platform.)
CRITERR.C	Critical error handler. (OEM specific platform.)
REPORT.C	Error reporting routine. (OEM specific platform.)
TIMER.C	Timer supported routines. (OEM specific platform.)
RDWR.C	Block read/write routines. (OEM specific platform.)
UTIL.C	String manipulation and byte order conversion routines.
PLX9054.H	PLX 9054 PCI controller definitions.
CRC.C	CRC generation/validation for SPI/MultiMediaCard mode.
MMCOEM.C	OEM specific low level routines.
SPIOEM.C	OEM specific low level routines.

The **FAT File System** directory contains the following files:

PCDISK.H	File System data structures and equates
INTRFACE.H	Peripheral Bus software layer for File System
APIUTIL.C	Utility routines to support the File System
BLOCK.C	Directory block buffering routines
CHKMEDIA.C	Device checking and Configuration
DEVIO.C	File I/O software layer to access storage devices
DROBJ.C	Object management. (Internal use only.)
ERRCODE.C	Converts critical errors to internal error codes
FILESRVC.C	File structure source code
FLAPI.C	File System API
FLCONST.C	File System constant data variables and structures
FLUTIL.C	File System utilities
FORMAT.C	High level format service
FSAPI.C	User API level source code
LONGFN.C	Long file name support routines
LOWL.C	Low level file allocation table management
PCKERNEL.C	Supported routines for multitasking environment
PC_MEMRY.C	Memory service routines

The **Interface** directory contains the following files:

DRIVE.H	Peripheral bus interface driver's structures and data
IOCONST.C	Contains all constant variables
IOUTIL.C	Shared routines between modules
SD/MultiMediaCard and SPI Files-*	
SDMMC.C	Low level driver for SD/MultiMediaCard and SPI buses shared between the those interfaces
SDMMC.H	Data structures and equates for SD/MultiMediaCard and SPI buses
SPIDRV.C	SPI driver interface and hardware related routines
SPI.C	Data structures for SPI driver
MMCDRV.C	SD/MultiMediaCard driver for SD/MultiMediaCard bus
MMC.C	Data structures for SD/MultiMediaCard driver

Note: The File System is optional and can be removed from the build. If the File System is removed, there will be only one file, **UTIL.C**, remaining from the File System to support the peripheral buses.
The description above shows all Peripheral Buses currently supported by the SDDK-05, but not all of these buses will be included in the SDDK-05 development floppy.

The **Tstsampl** directory includes sample files for different peripheral bus interfaces and File Systems. They are described below:

SD/MultiMediaCard and SPI Files-	
HDTKMMC.C	MultiMediaCard interface demonstration without File System
HDTKSPI.C	SPI interface demonstration without File System
Common Files-	
CPFRSD.C	File copy from device to host utility
CPSDTOSD.C	File copy between devices supported by File System
CPTOSD.C	File Copy from host to device utility
DOTEST.C	A simple test to create a file on the target
REGRESS.C	Disk exercise utility
SDCAT.C	Display file content utility
SDLS.C	Get hierarchical Directory utility
SDMKD.C	Make directory utility
SDREN.C	Rename file name utility
SDRM.C	Delete file utility
SDRMD.C	Remove directory utility
TSTECC.C	ECC test utility
TSTEXT.C	File extend utility
TSTSH.C	Command shell file utility

The **OEM** directory contains build instructions (makefiles) and other related information to create object files and libraries.

4.0 Porting

This section describes the SDDK-05 porting and configuration for either SPI or SD/MultiMediaCard mode. It describes how to best configure the SDDK-05 for the user's environment and how to port the system-specific portions to user's environment. There are several files that need to be ported to the target platform, most of which are in the platform module:

- Configuration **SDCONFIG.H** - contains configuration options.
- Interrupt Management Functions **INTERUPT.C** - interrupt service routine for the target platform.
- Timer Management Functions **TIMER.C -** contains timer routines to support the run-time driver.
- Critical Error Handler **CRITERR.C** - critical error handler.
- Error Reporting Functions **REPORT.C** - error reporting routine.
- System Dependent I/O Access **RDWR.C** - block move data routines
- SPI Hardware Access **SPIOEM.C** - SPI-specific hardware driver routines.
- SD/ MultiMediaCard Controller Access **MMCOEM.C**, SD/MultiMediaCard-specific hardware driver routines.

The block diagram below describes the flow of the porting process.

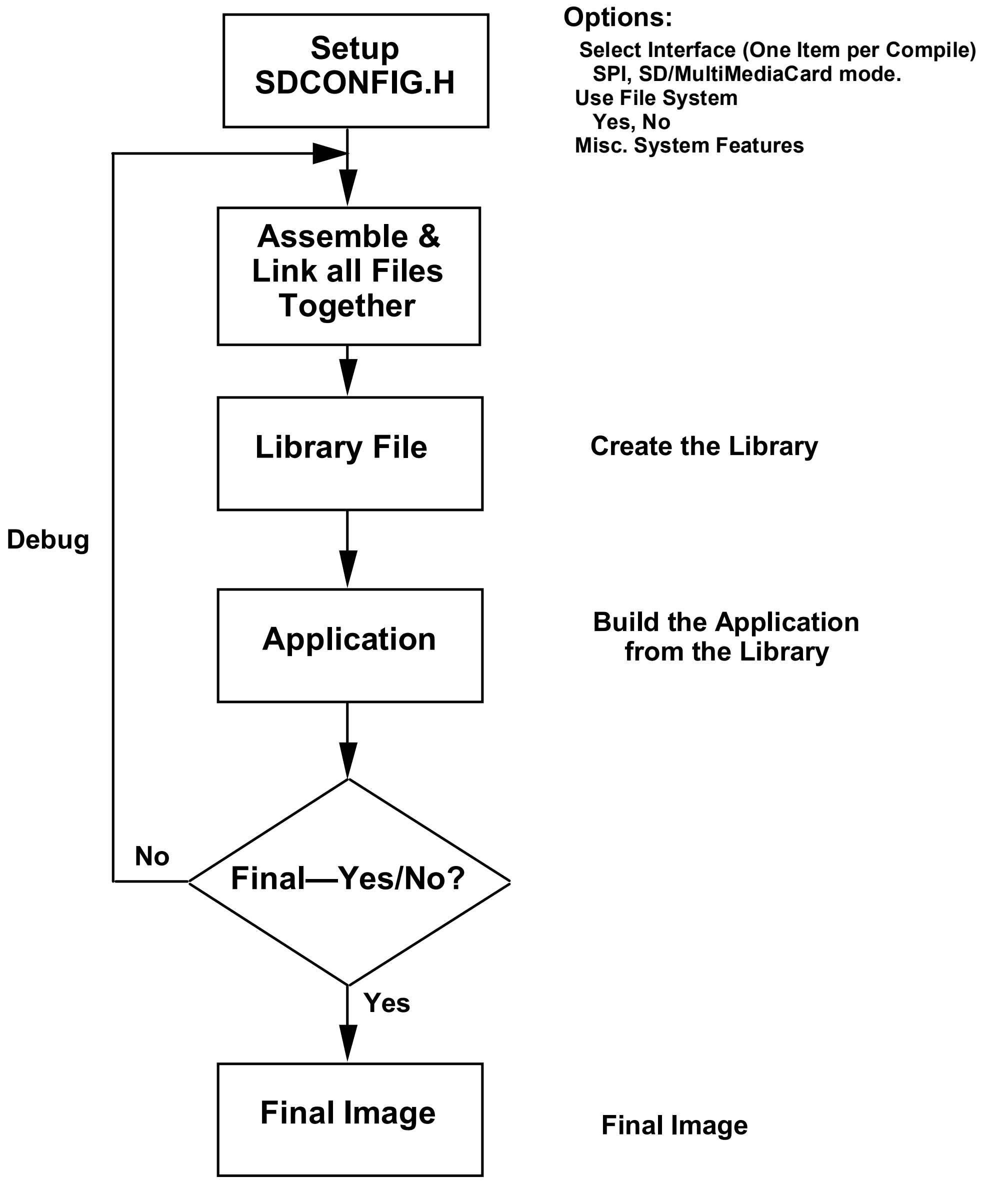

Figure 4-1 Sample Flow of Porting Process

4.1 SDCONFIG.H - Configuration Options

The file **SDCONFIG.H** contains compilation configuration constants that may be changed to tune memory utilization and to enable/disable subsections. By modifying constants in this file, the user can select either one of the peripheral bus interfaces (SPI or SD/MultiMediaCard), memory or I/O address configuration, the number of controllers and drives to support (1 or 2), and various other options.

Other specialized configuration options include enabling the pre-erase feature. By enabling this feature, the user directs the File System to pre-erase sectors so that subsequent write operations will be faster. Because flash memory must be erased before it is written, the performance of normal write operations includes this erasure overhead. If sectors are pre-erased, the subsequent write operations can take place with a significant performance improvement.

Pre-erasing is useful when write operations must take place at the highest possible performance. However, the actual pre-erase operations require just about as much time as normal writes. Thus, the user should only use pre-erasure in areas of system processing where this additional time is not prohibitive. Pre-erase can be enabled within several areas of the file system, including file deletion, the allocation of contiguous extensions to files, and also during disk formatting. A detailed applications note on Pre-erase is available from SanDisk.

Other configuration options allow a developer to tailor performance and memory usage. These include selecting the amount of memory to use for internal buffering and omitting sections of code to reduce the ROM footprint. After changing any of the values in **SDCONFIG.H,** the user must recompile the whole library.

The configuration options are divided as follows:

- Software Configuration Group
  - File System
  - Peripheral Bus Interface
- Hardware Configuration Group
  - SPI
  - SD/MultiMediaCard
- System Specific and Compilation Group

The following sections describe the configuration groups.

4.1.1 Software Configuration Group

The File System and Peripheral Bus Interface options can be specified independently of each other. When the user is configuring the File System, he/she does not need to know how the low level interface works. Similarly, when configuring the peripheral bus interface, it can be done without knowledge of whether or not the File System is enabled. Options are divided into the following groups:

- File System Group
- Peripheral Bus Interface Group

4.1.1.1 File System Group

The File System group is enabled or disabled via the USE_FILE_SYSTEM option. When this option is set (#define USE_FILE_SYSTEM 1), the File System is enabled and included in the build. When it is zero, the File system is disabled and excluded from the executable image.

The following File System group configuration options may be modified:

- USE_FILE_SYSTEM
- RTFS_SHARE
- RTFS_SUBDIRS
- RTFS_WRITE
- NUM_USERS
- NBLKBUFFS
- NUSERFILES
- FAT_BUFFER_SIZE
- EMAXPATH

Each configuration option of the File System group is discussed below in detail.

USE_FILE_SYSTEM	This option determines if the File System is to be included or excluded from the build. Setting this option to one will enable the File System. Setting this option to zero will remove the File System from the build. If this option is zero, all related optional files can be ignored.
RTFS_SHARE	Set this option to zero to disable checking file sharing options such as open exclusive, open exclusive write, etc. The default setting is zero (disabled). Disabling RTFS_SHARE saves a small amount of ROM space.
RTFS_SUBDIRS	Set this option to zero to disable sub-directory support. The default is one (enabled). Setting RTFS_SUBDIRS to zero saves a small amount of ROM space but eliminates sub-directory support.
RTFS_WRITE	Set this option to zero to disable writing support. The default is one (enabled). Setting RTFS_WRITE to zero saves a small amount of ROM space but eliminates all write support including file writes, formatting and sub-directory creation.

NUM_USERS	This option determines the number of user contexts provided. Each user context contains a current working directory and a current default drive. The default value is one. If this value is increased, the multitasking support macros and routines in **PCKERNEL.H** and **PCKERNEL.C** must be implemented.
NBLKBUFFS	This option determines the number of block buffers for sub-directory traversal. There must be at least one buffer per drive. Increasing the number of buffers can increase performance, but since flash ATA read performance is relatively high, it is not necessary to use a large value for this constant. Each block buffer requires approximately 530 bytes of RAM.
NUSERFILES	This option determines the maximum number of simultaneous files that may be opened. The default value is 10. Each file requires approximately 100 bytes of RAM. Reducing this value uses less RAM, increasing it uses more.
FAT_BUFFER_SIZE	This option defines the number of memory blocks reserved per drive to buffer the drive’s file allocation table. Each buffer requires a block of 512 bytes of RAM. The minimum value is two blocks. The default value for FAT_BUFFER_SIZE is two. Increasing this value will improve performance but require more RAM.
EMAXPATH	This option defines the maximum path size for path names passed to API calls. The default value is 128. If the user has a controlled, embedded system and does not require such large paths, this value may be reduced. This will reduce stack requirements a bit. Note that the API calls do not check the lengths of the string arguments that are passed to them.

4.1.1.2 Peripheral Bus Interface Group

The Peripheral Bus Interface Group consists of several different options. Each option, listed below, selects a particular Peripheral Bus.

USE_SD_CARD

USE_SPI

USE_SD_MMC

USE_SPI_EMULATION

USE_MMC_EMULATION

USE_SECURITY

Only one Peripheral Bus interface is selected and enabled at a time. Other bus interfaces should be disabled. When a bus interface is selected, the referred low level driver is enabled and included into the build. Other options offer more features that can be added to the low level driver.

USE_SD_CARD	Set this option to one to use SD Card. Otherwise, the MultiMediaCard will be used.
USE_SPI	Set this option to one to use True SPI Mode to access SD/MultiMediaCard devices. True SPI mode can be found on Motorola processors such as 68HC11, 68328, PowerPC 821, 860, many TI and non-TI, 370C Intel family, etc.
USE_SD_MMC	Set this option to one to use True SD/MultiMediaCard Mode to access SD/MultiMediaCard devices. True SD/MultiMediaCard mode can be found on platforms that support the SD/MultiMediaCard specification.
USE_SPI_EMULATION	Set this option to one to use SPI Emulation hardware to access SD/MultiMediaCard devices. An SPI Emulation hardware device emulates SPI signals to access SD/MultiMediaCard devices such as Parallel to SPI, Serial to SPI, etc., to be used on non-SPI systems.
USE_MMC_EMULATION	Set this option to one to use SD/MultiMediaCard Emulation hardware to access SD/MultiMediaCard devices. An SD/MultiMediaCard Emulation hardware device emulates the SD/MultiMediaCard signals to be used on non-MultiMediaCard systems.
USE_SECURITY	Set this option to one to enable access to the SD Card Secure Area.

Additional Options, related to the Peripheral Bus interface, are also provided:

USE_MEMMODE
USE_CONTIG_IO
USE_INTERRUPTS
USE_ONLY_LBA

USE_MULTI
USE_SET_FEATURE
WORD_ACCESS_ONLY
PREERASE_ON_ALLOC
PREERASE_ON_DELETE
PREERASE_ON_FORMAT
USE_PWR_MGMT
USE_HW_OPTION

USE_MEMMODE	Set this option to one to configure the Peripheral Bus Interface in Memory Mode. Set it to zero to allow the bus interface to operate in I/O Mode. I/O mode follows the Intel class processor interface for peripherals. For ATA devices, the register set will appear in the common memory space window.
USE_CONTIG_IO	Set this option to one to access the peripheral bus registers in a contiguous I/O fashion. Set this option to zero to access the peripheral bus registers at different locations. For IDE devices, if this option is set to one, the IDE register set is defined as 16 contiguous I/O locations. If set to zero, the alternate status register and drive address register are offset from the IDE register bank by 0x206 and 0x207 respectively. The latter configuration is standard for IDE in an IBM-AT class machine but the contiguous configuration is superior for most embedded systems. For other Peripheral Bus interfaces, the USE_CONTIG_IO option may not be available.
USE_INTERRUPTS	Set this option to zero to allow the bus interface to run only in polled mode. Setting this option to one enables the interrupt service, where the system runs in interrupt mode. When this value is set to zero, the different constants related to interrupt service are automatically set to -1 and the interrupt management code in **INTERUPT.C** is automatically stubbed out. This is the simplest port to do because an easier debug environment is provided. It is advisable to disable interrupts when first porting the code to the target system.
USE_ONLY_LBA	Set this option to one to force the device driver to access the device using LBA (logical block address) mode only. This reduces code size somewhat and speeds execution. If this constant is zero, the driver determines at run time whether to use CHS or LBA mode. LBA mode is always selected if the device supports it. Because SanDisk products always support LBA mode, it has been made a compile time option.

USE_MULTI	Set this option to one to instruct the driver to perform multi-sector transfers per interrupt. If the drive supports this mode, it can reduce the number of interrupts required to complete a transfer. Turning this option off at compile time reduces code size a bit. The default is off (0) since SanDisk products all support MULTI opcodes, but transfer one block per interrupt.
USE_SET_FEATURES	For SPI or SD/MultiMediaCard devices, set this option to one to force the CRC feature to be sent to the SD/MultiMediaCard device after each power up/reset. When this feature is enabled, data will be checked and command CRC information is calculated for every requested command.
WORD_ACCESS_ONLY	Set this value to one to enable WORD accessing to Task File Registers. Setting this value to zero will enable BYTE accessing to Task File Registers.
PREERASE_ON_ALLOC	Set this value to one to cause all sectors allocated when extending a file (via the po_extend_file function) to be pre-erased. This will cause additional processing time (performance is comparable to that of writing all of the associated sectors), but subsequent write performance is increased.
PREERASE_ON_DELETE	Set this value to one to cause all sectors occupied by deleted files to be pre-erased. This will cause additional processing time for file deletion (performance is comparable to that of writing all of the associated sectors), but subsequent write performance is increased.
PREERASE_ON_FORMAT	Set this value to one to cause all sectors on the volume to be pre-erased during the format operation. Time to pre-format the format is increased (performance is comparable to that of writing all of the associated sectors), but subsequent write performance is increased.
USE_PWR_MGMT	The default value is set to zero. The device will enter sleep mode after 5 msec of inactivity. Setting this option to a non-zero value will cause the device to stay in idle for a multiple of 5 msec before going to sleep mode if there is no disk access activity.
USE_HW_OPTION	In Big Endian (Motorola), the 16-bit data bus should be swapped for the ATA environment. Thus, set this option to one. In the SPI and SD/MultiMediaCard environments, the SDDK-05 will force this option to be set to one for internal code selection.

4.1.2 Hardware Configuration

The **SDCONFIG.H** file also includes hardware-related information based upon which peripheral bus interface is selected:

- SPI
- SD/MultiMediaCard

4.1.2.1 SPI Interface

The SPI bus supports both I/O and Memory Mapped Mode.

I/O Mode

In I/O mode, the constants SPI_PRIMARY_IO_ADDRESS and SPI_SECONDARY_IO_ADDRESS are assumed to be unsigned integers that contain the I/O addresses of the SPI devices. They are placed in the array io_mapped_addresses[ ] in **IOCONST.C** and are used by the peripheral bus driver to map the I/O space in and by **SPIDRV.C** to initialize the controller structure's I/O address pointer. This feature is associated with setting USE_MEMMODE to zero.

SPI_PRIMARY_IO_ADDRESS	Defines the primary SPI I/O address. The variable io_mapped_addresses[0] in **IOCONST.C** is initialized to this value. The default constant value should be set for the target environment.
SPI_SECONDARY_IO_ADDRESS	Defines the secondary SPI I/O address. The variable io_mapped_addresses[1] in **IOCONST.C** is initialized to this value. The default value should be selected for the target platform.

Memory Mode

In Memory Mode, the constants SPI_PRIMARY_MEM_ADDRESS and SPI_SECONDARY_MEM_ADDRESS are assumed to be unsigned char pointers that contain the memory addresses of the SPI devices. They are placed in the array mem_mapped_addresses_pointer[] in **IOCONST.C** and are used by the peripheral bus driver to map the memory space in and by **SPIDRV.C** to initialize the controller structure's memory address pointer. This feature is associated with USE_MEMMODE set to one.

SPI_PRIMARY_MEM_ADDRESS	Defines the primary SPI memory address for the user's environment. The variable mem_mapped_addresses[0] in **IOCONST.C** is initialized to this value. The default value must be set for this constant.
SPI_SECONDARY_MEM_ADDRESS	Defines the secondary SPI memory address for the user's environment. The variable mem_mapped_addresses[1] in **IOCONST.C** is initialized to this value. The default value must be set for this constant.

The interrupts are supported by setting SPI_PRIMARY_INTERRUPTS and SPI_SECONDARY_INTERRUPT. They are placed in the array dev_interrupts[ ] in **IOCONST.C** and are used by the peripheral bus driver **SPIDRV.C** to initialize the controller structure's information and to setup the interrupt service routines.

SPI_PRIMARY_INTERRUPT	Defines the primary SPI interrupt for the user's environment. The variable dev_interrupts[0] in **IOCONST.C** is initialized to this value. The default value is seven.
SPI_SECONDARY_INTERRUPT	Defines the secondary SPI interrupt for the user's environment. The variable dev_interrupts[1] in **IOCONST.C** is initialized to this value. The default value is five.

Note: Set SPI_PRIMARY_INTERRUPT to -1 to run the primary interface in non-interrupt mode. Set SPI_SECONDARY_INTERRUPT to -1 to run the secondary interface in non-interrupt mode. The interrupt service could also be shared among SPI devices. See also USE_INTERRUPTS.

4.1.2.2 SD/MultiMediaCard Interface

The SD/MultiMediaCard bus supports both I/O and Memory Mapped Mode.

I/O Mode

In I/O Mode, the constants MMC_PRIMARY_IO_ADDRESS and MMC_SECONDARY_IO_ADDRESS are assumed to be unsigned integers that contain the I/O addresses of the SD/MultiMediaCard devices. They are placed in the array io_mapped_addresses[ ] in **IOCONST.C** and are used by the peripheral bus driver to map the I/O space in and by **MMCDRV.C** to initialize the controller structure's I/O address pointer. This feature is associated with setting USE_MEMMODE to zero.

MMC_PRIMARY_IO_ADDRESS	Defines the primary SD/MultiMediaCard I/O address. The variable io_mapped_addresses[0] in **IOCONST.C** is initialized to this value. The default constant value should be set for the target environment.
MMC_SECONDARY_IO_ADDRESS	Defines the secondary SD/MultiMediaCard I/O address. The variable io_mapped_addresses[1] in **IOCONST.C** is initialized to this value. The default value should be selected for the target platform.

Memory Mode

In Memory Mode, the constants MMC_PRIMARY_MEM_ADDRESS and MMC_SECONDARY_MEM_ADDRESS are assumed to be unsigned char pointers that contain the memory addresses of the SD/MultiMediaCard devices. They are placed in the array mem_mapped_addresses_pointer[] in **IOCONST.C** and are used by the peripheral bus driver to map the memory space in and by **MMCDRV.C** to initialize the controller structure's memory address pointer. This feature is associated with USE_MEMMODE set to one.

MMC_PRIMARY_MEM_ADDRESS	Defines the primary SD/MultiMediaCard memory address for the user's environment. The variable mem_mapped_addresses[0] in **IOCONST.C** is initialized to this value. The default value must be set for this constant.
MMC_SECONDARY_MEM_ADDRESS	Defines the secondary SD/MultiMediaCard memory address for the user's environment. The variable mem_mapped_addresses[1] in **IOCONST.C** is initialized to this value. The default value must be set for this constant.

The interrupts are supported by setting MMC_PRIMARY_INTERRUPTS and MMC_SECONDARY_INTERRUPT. They are placed in the array dev_interrupts[] in **IOCONST.C** and are used by the peripheral bus driver **MMCDRV.C** to initialize the controller structure's information and to setup the interrupt service routines.

MMC_PRIMARY_INTERRUPT	Defines the primary SD/MultiMediaCard interrupt for the user's environment. The variable dev_interrupts[0] in **IOCONST.C** is initialized to this value. The default value is seven.
MMC_ SECONDARY _INTERRUPT	Defines the secondary SD/MultiMediaCard interrupt for the user's environment. The variable dev_interrupts[1] in **IOCONST.C** is initialized to this value. The default value is five.

Note: Set MMC_PRIMARY_INTERRUPT to -1 to run the primary interface in non-interrupt mode. Set MMC_SECONDARY_INTERRUPT to -1 to run the secondary interface in non-interrupt mode. The interrupt service could also be shared among SD/MultiMediaCard devices. See also USE_INTERRUPTS.

4.1.3 System Specific and Compilation Options

All other options for the Peripheral Bus or compiler are system specific. They are:

N_CONTROLLERS
DRIVES_PER_CONTROLLER1
DRIVES_PER_CONTROLLER2
LITTLE_ENDIAN
FAR

N_CONTROLLERS	This option defines the number of controllers supported. The maximum value is two. The default is one. For a system that has more than two peripheral controllers, a few data structures in **IOCONST.C** need to be modified to accommodate the requirements.
DRIVES_PER_CONTROLLER1	This option defines the number of drives supported on the first controller. The maximum number of flash devices varies depending on the peripheral controller. The default is one.
DRIVES_PER_CONTROLLER2	This option defines the number of drives supported on the second controller. The maximum number of flash devices varies depending on the peripheral controller. The default is zero.
LITTLE_ENDIAN	The user may set this to one if he/she is running in an Intel little_endian environment. Doing this results in slightly reduced code size and slightly increased performance. If little_endian is zero, the code will execute in a big_endian environment but data will be converted to little_endian in the appropriate places.
FAR	Define this as far or _far in segmented Intel environments, otherwise define it as nothing (for example, #define FAR far or #define FAR).

4.1.4 *Examples*

This section includes several examples that illustrate the use of options in **SDCONFIG.H**.

4.1.4.1 *SPI Emulation Configuration Options*

When the SPI Interface is selected with the File System, the options in the **SDCONFIG.H** are set as follows:

```
#define USE_FILE_SYSTEM          1     /* FAT File System is enabled */
#define RTFS_SHARE               0     /* File Sharing is disabled */
#define RTFS_SUBDIRS             1     /* Sub-directory is allowed */
#define RTFS_WRITE               1     /* Writing to the device is allowed */
#define NUM_USERS                1     /* Number of users or tasks */
#define NBLKBUFFS                2     /* Number of block buffers */
#define NUSERFILES               10    /* Maximum number of open files */
#define FAT_BUFFER_SIZE          2     /* Size of the FAT buffer in 512 bytes per block */
#define EMAXPATH                 128   /* Maximum path length */

#define USE_SD_CARD              1     /* SD Card is used */
#define USE_SECURITY             1     /* Secure access to SD Card is enabled */
#define USE_SPI                  0     /* SPI interface is disabled */
#define USE_SD_MMC               0     /* SD/MultiMediaCard interface is disabled */
#define USE_SPI_EMULATION        1     /* SPI Emulation mode is enabled */
#define USE_MMC_EMULATION        0     /* SD/MultiMediaCard Emulation mode is disabled */
#define WORD_ACCESS_ONLY         0     /* 16-bit Data Bus is disabled */
#define USE_MEMODE               1     /* Memory Mapped mode is enabled */
#define USE_INTERRUPT            0     /* No interrupt service.  Use polling technique */
#define USE_LBA_ONLY             1     /* Use Logical Block Address */
#define USE_SET_FEATURES         1     /* Error Correction Code is enabled */
#define USE_CONTIG_IO            1     /* Memory or I/O Address range is contiguous */
#define USE_MULTI                1     /* Multiple sectors transfer is enabled */
#define PREERASE_ON_ALLOC        0     /* Erase when files are extended */
#define PREERASE_ON_DELETE       0     /* Erase when files are deleted */
#define PREERASE_ON_FORMAT       0     /* Erase when the volume is formatted */
#define LITTLE_ENDIAN            1     /* Use Intel data format type */

#if (USE_INTERRUPTS)
#define SPI_PRIMARY_INTERRUPT        0x7   /* Primary I/O interrupt */
#define SPI_SECONDARY_INTERRUPT      0x5   /* Secondary I/O interrupt */
#else
#define SPI_PRIMARY_INTERRUPT        -1    /* No Primary I/O interrupt */
#define SPI_SECONDARY_INTERRUPT      -1    /* No Secondary I/O interrupt */
#endif
```

4.1.4.2 SD/MultiMediaCard Configuration Options

The SD/MultiMediaCard Interface is configured with the following options:

```
#define USE_FILE_SYSTEM          1      /* FAT File System is enabled */
#define RTFS_SHARE               0      /* File Sharing is disabled */
#define RTFS_SUBDIRS             1      /* Sub-directory is allowed */
#define RTFS_WRITE               1      /* Writing to the device is allowed */
#define NUM_USERS                1      /* Number of users or tasks */
#define NBLKBUFFS                2      /* Number of block buffers */
#define NUSERFILES               10     /* Maximum number of open files */
#define FAT_BUFFER_SIZE          2      /* Size of the FAT buffer in 512 bytes per block */
#define EMAXPATH                 128    /* Maximum path length */

#define USE_SD_CARD              1      /* SD Card is used */
#define USE_SECURITY             1      /* Secure access to SD Card is enabled */
#define USE_SPI                  0      /* SPI interface is disabled */
#define USE_SD_MMC               1      /* SD/MultiMediaCard interface is enabled */
#define USE_SPI_EMULATION        0      /* SPI Emulation mode is disabled */
#define USE_MMC_EMULATION        0      /* SD/MultiMediaCard Emulation mode is disabled */
#define WORD_ACCESS_ONLY         0      /* 8-bit Data Bus is enabled */
#define USE_MEMODE               1      /* Memory Mapped mode is enabled */
#define USE_INTERRUPT            0      /* No interrupt service.  Use polling technique */
#define USE_LBA_ONLY             1      /* Use Logical Block Address */
#define USE_SET_FEATURES         1      /* ECC (Error Correction Code) is enabled */
#define USE_CONTIG_IO            0      /* Memory or I/O address region  is random */
#define USE_MULTI                1      /* Multiple sectors transfer is enabled */
#define PREERASE_ON_ALLOC        0      /* Erase when files are extended */
#define PREERASE_ON_DELETE       0      /* Erase when files are deleted */
#define PREERASE_ON_FORMAT       0      /* Erase when the volume is formatted */
```

4.2 INTERUPT.C — Interrupt Management Functions

INTERUPT.C provides interrupt services. In many embedded environments, real time interrupts are needed at all times. The SDDK-05 interrupt model has been designed to meet the real time computing requirements. The interrupt model is based on the semaphore mechanism for signaling and exchanging information. When a semaphore operation is performed, it is waiting at the semaphore for a condition change before proceeding. This semaphore is private to the SDDK-05, but can easily be modified to adapt to different system environments.

The SDDK-05 can be programmed to use no interrupts. In this case, there are no integration issues surrounding interrupts, but the user loses the ability to react to card removal and re-insertion events. He/she also loses the ability to allow other tasks to execute while waiting for a data transfer to complete. Since debugging a non-interrupt system is typically easier, it is recommended that the user starts without interrupts and turn them on once he/she has basic functionality.

4.2.1 Porting Requirements

The following functions define the interrupt service for the SDDK-05.

These routines are invoked by the Peripheral Bus Interface through the use of macros and are defined as follows:

SDVOID platform_controller_init(INT16 controller_no)
SDVOID platform_controller_close(INT16 controller_no)
SDVOID platform_set_signal(INT16 driveno)
SDVOID platform_clear_signal(INT16 driveno)
SDBOOL platform_wait_for_action(INT16 driveno, COUNT wait_ticks)

SDVOID platform_controller_init(INT16 controller_no)

Sets up system hardware, enables hardware interrupts and installs interrupt service routines.

This routine must be implemented if the interrupt service is preferred. It is given a controller structure which contains the interrupt number to use and the controller number. This routine must make sure that the interrupt is hooked so that when an interrupt occurs, the interrupt service will provide a signaling mechanism so that platform_wait_for_action() can block at the user level and platform_set_signal() can be called from the interrupt layer to wake the blocked thread with a proper signal. In most real time kernel environments, a simple counting semaphore initialized to zero will suffice.

SDVOID platform_controller_close(INT16 controller_no)

Removes interrupt service and restores system information.

This routine must be implemented if the interrupt service is selected. The interrupt signal is reset to avoid further false alarm requests and interrupt data structures are clear. System information will be restored to the original condition.

SDBOOL platform_set_signal(INT16 driveno)

Signals a device interrupt.

This routine must be implemented if interrupts are enabled. It is called from the interrupt service routines and must create a condition such that platform_wait_for_action() returns YES. A typical implementation would increment a counting semaphore.

SDVOID platform_clear_signal(INT16 driveno)

Clears device interrupt signal.

This routine must be implemented if the interrupt service is enabled. This routine is invoked at the user level. It is called before a command is issued and must establish a state such that platform_wait_for_action() will block until platform_set_signal() is called by the interrupt service routines.

SDBOOL platform_wait_for_action(INT16 driveno, COUNT wait_ticks)

Waits for an interrupt signal in a specific time.

This routine must be implemented if the interrupt service is enabled. It must wait for platform_set_signal() to signal that an interrupt has occurred. If no signal arrives in wait_ticks, the routine returns NO to indicate that there is a communication problem with the media, otherwise it returns YES.

Note: Platform_wait_for_action() must be implemented such that it returns YES if the signal occurs before the routine is called. A typical implementation would call a counting semaphore timed wait function.

These routines are invoked by the Peripheral Bus Interface through the use of the following macros:

OS_CONTROLLER_INIT	platform_controller_init
OS_CONTROLLER_CLOSE	platform_controller_close
OS_SET_SIGINAL	platform_set_signal
OS_CLEAR_SIGNAL	platform_clear_signal
OS_WAIT_FOR_ACTION	platform_wait_for_action

4.3 *TIMER.C — Timer Management Functions*

Most systems have at least one timer function that is called at a fixed interval. The SDDK-05 makes use of several system clock timer functions. These functions rely on a system timer being available and the system hardware architecture. The SDDK-05 does not provide this access in portable C code. Instead, several function prototypes are defined by the SDDK-05 to form a simple timer service.

4.3.1 Porting Requirements

Those routines are invoked by the File System or the Peripheral Bus Interface through the use of macros and are defined as follows:

```
ULONG platform_ticks_p_second(SDVOID);
ULONG platform_get_ticks(SDVOID);
platform_delayms(COUNT milliseconds);
oem_getsysdate(UINT16 *date, UINT16 *time);
```

ULONG platform_ticks_p_second()	Returns the current system clock period ticks per second. This routine must be implemented. It must return the current system tick period in ticks per second. This returned value is a scaled down value of the system timer.
ULONG platform_get_ticks()	Returns the ever-increasing timer tick count. This routine must be implemented. It must return the current system tick count. The period of the system tick is unimportant but the function platform_ticks_p_second() in **TIMER.C** must be changed to match the particular system. The peripheral driver uses this function to set up a certain time it can remain in a particular process and generates a time out event to avoid a dead loop when the time has expired.
platform_delayms()	Returns to the caller after a specific time expired. This routine must be implemented. Given a certain time, the main purpose of this routine is to make sure the device comes up in a stable state during the power down and up cycles or reset.
oem_getsysdate()	Returns the current system date and time. This routine is only for use with the File System. The returned values by this routine are translated into DOS format. Then, the translation is written as the date/time values in the directory entries. The Date field is combined into a 16-bit Date field and is encoded with the following format: Bit 0-4: Day of month (1-31) Bit 5-8: Month (1-12) Bit 9-15: Year relative to 1980

The Time field is combined into a 16-bit Time field and encoded with the following format:

Bit 0-4:	Second multiply by 2 (0-29)
Bit 5-10:	Minutes (0-59)
Bit 11-15:	Hours (0-23)

Please note that it is not necessary to implement this routine if the peripheral bus is the only target.

4.4 CRITERR.C — Critical Error Handler

4.4.1 Introduction

For cases where a disk access fails, host platforms need to be notified of the error condition. Many of these errors are caused by inappropriate end user actions such as prematurely removing or swapping devices. In such instances, a function, critical_error_handler(), is called with parameters indicating the drive number and error code. Routines that the user supplies within the critical error handler can interact with the user in a platform-specific way, requesting that the device be reinserted. It can then return a code to the File System, directing it to retry the failed operation. Most platforms have at least some means of reporting an error to the end user, even if it's just beeping or flashing an LED. However, if the user's system is totally non-interactive, he/she can hard code return values, directing the File System to always retry or abort failed operations.

The device I/O layer implemented in **DEVIO.C**, **CHKMEDIA.C** and the Peripheral Bus driver calls the critical error handler when it requires feedback to determine which route to take in an error recovery process. A sample implementation (**CRITERR.C**) is provided which prints the error and some vital statistics to the console and then queries the console for the recovery route to proceed with. The sample implementation also demonstrates how to extract vital statistics from the File System such as the current mounted volume and whether or not buffers have been flushed.

4.4.2 Unusual Error Conditions

In addition to common errors such as "drive not ready" (i.e., ATA device is not properly inserted), it's possible (although not likely) that an "ID not found" or ECC error could occur. SanDisk products have extensive built-in defect management ECC and spare sectors. Thus, unlike rotating disks, any problems can be dynamically handled by the controller. However, if flash devices are misused under the most abusive circumstances, it is possible to encounter an error condition.

For example, during a write operation, if the device is prematurely ejected, writing to a particular sector may not be completed. The critical error handler will be notified of the condition, and if the card is successfully replaced and the critical error handler directs the File System to retry the operation, the write can continue successfully. However, if the user does not replace the device, or the system is powered off at that moment, the sector can be left partially written.

Subsequent reads of the sector may result in either an "ID not found" or an ECC error. The data is lost, but compounding the problem is the fact that this sector must be rewritten to be made useable again (the device itself is still useable, it's only the individual sector that's out of commission). Although this error scenario is uncommon, the file system error recovery scheme provides a solution. In such cases, during the failed read operation, the critical error handler will be notified of the error (and the logical block number of the associated sector). The critical error handler must notify the application so that it can take appropriate steps to recover; it can then pass a return code to the file system indicating that the sector should be written with a null record so that it can at least be made useable.

Again, this error condition is rare, but the user should add code to the critical error handler to process it if he/she anticipates any of the scenarios that could cause this problem.

4.4.3 *Porting Requirements*

The platform_critical_handler has the following syntax:

INT16 platform_critical_handler (INT16 driveno, INT16 media_status, ULONG sector)

driveno	The drive number.
media_status	The error to handle.
sector	Sector number where error occurred.

The Error may be set with one of the following error numbers:

CRERR_BAD_FORMAT	A valid device is in the slot but it does not have a recognizable MS-DOS partition on it. If critical_error_handler() returns CRITICAL_ERROR_FORMAT, the recovery code will attempt to format the device and mount it. If it returns CRITICAL_ERROR_RETRY, the mount operation will be retried. If it returns CRITICAL_ERROR_ABORT, the operation will fail. (The API call will fail.)
CRERR_NO_CARD	The device is expected but there is no device in the slot. The error handler should prompt the user to insert a device and then return CRITICAL_ERROR_RETRY or it should return CRITICAL_ERROR_ABORT to force the operation to fail. (The API call will fail.)
CRERR_CHANGED_CARD	The device is installed but its serial number does not match the serial number of the volume that is currently mounted. The error handler should prompt the user to re-insert the proper device and then return CRITICAL_ERROR_RETRY or it should return CRITICAL_ERROR_ABORT to force the operation to accept the new device, flush all buffers from the old mount and proceed with the new device.

Note: The File System does a check media call when it enters all API calls.

	If the CRITICAL_ERROR_ABORT condition (remount and proceed) is detected by an API call such as po_open, pc_mkdir, pc_rmdir, pc_gfirst etc., the call will proceed on the new volume without returning a failure to the application. If the application is executing an API call that can't proceed without another API call first executing, such as po_read, po_lseek, po_write, pc_gnext etc., the API call will fail.
CRERR_BAD_CARD	The device is expected but an unrecognized device type is in the slot. This is logically equivalent to CRERR_NO_CARD. The error handler should prompt the user to insert a device and then return CRITICAL_ERROR_RETRY or it should return CRITICAL_ERROR_ABORT to force the operation to fail. (The API call will fail.)
CRERR_CARD_FAILURE	A disk I/O error occurred. The device is in a slot and the I/O layer has become unable to talk to it. If the error handler returns CRITICAL_ERROR_RETRY, the I/O layer will attempt to reset the device and retry the operation. If it returns CRITICAL_ERROR_ABORT, the I/O layer will force the operation to fail. (The API call will fail.)

CRERR_ID_ERROR	The read operation has encountered an “ID not found” error.
CRERR_ECC_ERROR	The read operation has encountered an ECC error.

The critical_error_handler return codes are as follows:

CRITICAL_ERROR_ABORT	Always forces a failure return to the upper layer.
CRITICAL_ERROR_RETRY	Always forces the I/O layer to retry the current operation.
CRITICAL_ERROR_FORMAT	Always forces the I/O layer to format the disk. This return code can only be used in response to the CRERR_BAD_FORMAT error.
CRITICAL_ERROR_CLEARECC	Forces the File System to write a null record to a sector that caused an “ID not found” or ECC error during a read. This is not often used.

4.4.4 Error Recovery Strategies

The above paragraphs describe how the critical error handler should react to I/O errors. It is important to note that the applications layer must also be able to recover. To recover from a CRITICAL_ERROR_ABORT, the application should either be able to back up to the previous operation or it should be prepared to discard data. The application might also wish to communicate with the error handler to change the error recovery logic depending on where in the application thread it is.

For example, a digital camera has an image stored in memory. It wants to open a file and dump the image to a disk. To do this, it opens a file, writes to it several times and then closes it. If the card was removed after the open but before the close, it must be re-inserted or the data will be lost. The strategy here can be to either instruct the error handler not to allow the abort, or if the abort is accepted, the application layer must be willing to either discard the image or to back up and open the file again (on a new device) and then write the data and close.

4.5 *REPORT.C — Error Reporting Functions*

Most of the time, the flash devices and the host system communicate without user knowledge. The user only gets involved when the SDDK-05 cannot determine whether to handle an error event such as writing to the non-existing flash device, data corrupted, or Invalid File system, etc.

The user will be asked to inform the SDDK-05 what it should do next. This error report process should be coupled tightly with the Critical Error Handler so that all error events are monitored within the handler.

4.5.1 Porting Requirements

The syntax of the error reporting function is defined as shown below:

SDVOID platform_report_error (INT16 error_number) User interaction.

4.6 *RDWR.C — System Dependent I/O Access*

In many environments, it is possible to perform hardware access through macros. The SDDK-05 offers several macros, SDREAD_DATA08, SDREAD_DATA16, SDREAD_DATA32, SDWRITE_DATA08, SDWRITE_DATA16 and SDWRITE_DATA32, to perform basic read/write access to system hardware or device registers. The implementation of these macros is shown below:

In an I/O Mapped environment, the following macros are provided:

```
        #define SDREAD_DATA08(X) (inpbyte((UINT16) (X) ))
        #define SDWRITE_ DATA08(X,Y)  outpbyte((UINT16) (X),(UCHAR) (Y))

#if (WORD_ACCESS_ONLY) /* 16-bit interface */

        #define SDREAD_DATA16(X) (inpword ((UINT16) (X) ))
        #define SDWRITE_DATA16(X,Y) outpword ((UINT16) (X), (UINT16) (Y))
#endif
```

In a Memory Mapped environment, the following macros are provided:

```
/* Read/Write register access */
#define SDREAD_DATA32(X)            *((FPTR32) (X))
#define SDREAD_DATA16(X)            *((FPTR16) (X))
#define SDREAD_DATA08(X)            *((FPTR08) (X))

#define SDWRITE_DATA32(X, Y)                (*((FPTR32) (X)) = (UINT32) (Y))
#define SDWRITE_DATA16(X, Y)                (*((FPTR16) (X)) = (UINT16) (Y))
#define SDWRITE_DATA08(X, Y)                (*((FPTR08) (X)) = (UINT08) (Y))
```

Note: The value of X (the address) comes from the register_file_address field in the device_control structure (see **ATADRV.H**). These value are loaded from the array io_mapped_addresses[] or mem_mapped_addresses[]. (See **IOCONST.C** and the Configuring section of this user's guide.)

4.6.1 Porting Requirements

In an I/O Mapped Mode, the following routines must be ported to match the user's platform.

```
UINT16 inpword(UINT16 address);
SDVOID outpword(UINT16 address, UINT16 data);
UCHAR inpbyte(UINT16 address);
SDVOID outpbyte(UINT16 address, UCHAR data);
```

The block data moves improve system performance. They are:

```
#if (WORD_ACCESS_ONLY)
SDVOID oem_in_words (FPTR16 p, UCOUNT words, FPTR16 dreg);
SDVOID oem_out_words (FPTR16 p, UCOUNT words, FPTR16 dreg);
#else
SDVOID oem_in_words (FPTR p, UCOUNT words, FPTR dreg);
SDVOID oem_out_words (FPTR p, UCOUNT words, FPTR dreg);
#endif
```

Where:

FPTR16 p: unsigned short SDFAR pointer to data buffer
FPTR p: unsigned char SDFAR pointer to data buffer
UCOUNT words: number of words to transfer
FPTR16 dreg,
FPTR dreg: This pointer has two different meanings:
- For I/O Mapped Mode, this refers to the controller structure
- For Memory Mapped Mode, this refers to the location offset 400h from the base address of the common memory if the A10 address line is mapped. Otherwise, it is a pointer to the base address of the common memory.

These routines are defined in the file **RDWR.C**.

4.7 SPIOEM.C

4.7.1 Overview

The **SPIOEM.C** file provides the functions to support OEM-specific hardware. It is related to the OEM hardware portion in the register read/write level. In order to port the whole SDDK-05, the user has to make this portion of the driver work first.

For example, for a microprocessor with SPI peripheral support, it has a set of registers, such as data registers, status registers, and control registers to be programmed, in order to realize reliable transfer of data IN/OUT of the microprocessor. Once this layer of the driver works correctly, the microprocessor can send a command in SPI format to the SanDisk SD card and MultiMediaCard from the upper layer. Therefore, this portion is similar to the "physical layer" of the network protocol stack. It is the base of all upper layer porting, and it should be stable and able to detect bit/byte transfer errors, such as the overrun error.

The **SPIOEM.C** functions primarily deal with reset of the (SPI) controller, transmission/reception of data, sending command, and checking status (ready, error, etc), all of which are in the register level and are hardware dependent. The related data structures are also defined here.

The OEM customer must modify the **SPIOEM.C** functions to work with their hardware system. Since SDDK-05 is designed to be portable and generic, the **SPIOEM.C** file has generic components also, as explained below.

4.7.2 SPI Modes Supported

SPIOEM.C consists of four modes:

- Native SPI
- PCI-SPI (for SanDisk Host Adapter board)
- SPI Emulation
- SPI Emulation using Parallel port

From the #if statements and comments below, the reader can identify these four modes.

```
#if (USE_SPI) /* Native SPI mode */

        #if (USE_MEMMODE) /*Memory mapped mode */

        /* This is for the Native SPI mode, where the Motorola DragonBall microprocessor is used as the
        template */

        #else /* I/O mode */

        /* This is for PCI-SPI mode with SanDisk FPGA host adapter */

        #endif

#elif (USE_SPI_EMULATION) /* This is for SPI Emulation mode */
```

```
        #if (USE_MEMMODE) /* Memory mapped mode */

        /* This is for the SPI Emulation mode using the serial port, where the Mitsubishi M16C
        microcontroller is used as  the template */

        #else /* I/O mode */

        /* This is for SPI Emulation using the PC's Parallel Port */

        #endif

#endif
```

4.7.3 Porting Requirements

Before making changes to these functions, the user should make their own low level device driver portion work standalone. Then fit the working portion into the **SPIOEM.C** file.

Based on the functional description above, the following is done to the four modes: Note: the USE_SPI, USE_SPI_EMULATION, USE_MEMMODE constants are taken from **SDCONFIG.H.** Please refer to section 4.1 above for details on these definitions.

- **PCI-SPI Mode** - USE_SPI and NOT USE_MEMMODE are used for the SanDisk PCI-SPI mode. It is fixed to the SanDisk PCI_SD Host Adapter, and should not be changed. If there is a need, please contact SanDisk.
- **SPI Emulation using Parallel Port Mode** - USE_SPI_EMULATION and NOT USE_MEMMODE are used for SanDisk Parallel Port Emulation of SPI mode. It is fixed to the SanDisk Parallel Port Card reader and should not be changed. If there is a need, please contact SanDisk.
- **Native SPI Mode** - Normally, the user will come to USE_SPI or USE_SPI_EMULATION mode. Both are in memory-mapped mode (USE_MMODE). The Motorola DragonBall microprocessor is used as a template for Native SPI mode (memory mapped),
- **SPI Emulation Mode** - The Mitsubishi M16C microcontroller SIO3/UART0 is used as a template for USE_SPI_EMULATION Mode (memory mapped).

As mentioned above, in many cases customers only need to match base addresses in **SDCONFIG.H** to use Native SPI or SPI Emulation mode. To do this, the user should go to **SDCONFIG.H** where he/she will see similar #if statement for USE_SPI or USE_SPI_EMULATION, USE_MEMMODE, etc. The user will also see definitions for different register sets, etc. defined there. The code itself is self-explanatory.

In spite of the generic aspect of the **SPIOEM.C** file, each system (microprocessor/microcontroller) has unique properties. When the user needs to make changes, he/she should consider that most functions in this file will be called from **SPIDRV.C.** Care should be taken not to change the prototype of functions like:

- mmcresetcontroller()
- mmcSendCommand()
- getMMCResponseInfo()
- MMCTransmit()
- MMCReceive

- MMCPrepareAndSetup()
- CheckCardBusy()
- isCRCErrOnWrite()
- startMMC80Clocks()
- spi_cs_disable()
- spi_cs_enable()
- spi_hw_init()
- MMCSelectController()
- ……..

If prototypes of these functions are changed, a compilation/linking error will occur.

The user can make changes inside these functions based on different platforms. He/she can add special functions to be called within these functions, if needed.

4.7.4 Conclusion

To use PCI-SD host adapter and Parallel port SPI emulation mode, normally the user should not change anything.

If Native SPI or SPI Emulation(Not using Parallel port) modes are used, the following describes how to make the required changes:

1. First, the user should make their own low level device driver portion, with register read/write, work standalone.
2. Then, locate the portion in **SPIOEM.C** which choose USE_SPI or USE_SPI_EMULATION, and study the template
3. Go to **SDCONFIG.H** to change the base address, etc., and
4. Go to **SPIOEM.C** to define new data structures, if needed. Carefully make changes inside the existing functions based on the specific microprocessor/micro controller system.

4.8 MMCOEM.C

4.8.1 Overview

The **MMCOEM.C** file provides the functions to support OEM-specific hardware. It is related to the OEM hardware portion in the register read/write level. In order to port the whole SDDK-05, the user has to make this portion of the driver work first.

For example, for a microprocessor with SD/MultiMediaCard peripheral support, it has a set of registers, such as data registers, status registers, and control registers to be programmed, in order to realize reliable transfer of data IN/OUT of the microprocessor. Once this layer of the driver works correctly, the microprocessor can send a command in SD/MMC protocol format to the SanDisk SD card and MultiMediaCard from the upper layer. Therefore, this portion is similar to the "physical layer" of the network protocol stack. It is the base of all upper layer porting, and it should be stable and able to detect bit/byte transfer errors, such as the overrun error.

The **MMCOEM.C** functions primarily deal with reset of the (SD/MultiMediaCard) controller, transmission/reception of data, sending command, and checking status (ready, error, etc), all of which are in the register level and are hardware dependent. The related data structures are also defined here.

The OEM customer must modify the **MMCOEM.C** functions to work with their hardware system. Since SDDK-05 is designed to be portable and generic, the **MMCOEM.C** file has generic components also, as explained below.

4.8.2 SD/MMC Modes Supported

MMCOEM.C consists of 3 modes:

- Native SD/MMC mode
- PCI-SD/MMC mode (for SanDisk Host Adapter board)
- SD/MMC Emulation using Parallel port

From the #if statements and comments below, the reader can identify these 3 modes.

```
#if (USE_SD_MMC)

        #if (USE_MEMMODE) /*Memory mapped mode */

        /* This is for the Native SD/MMC mode, user need to fill their own code to make it work */

        #else /* I/O mode */

        /* This is for PCI-SD/MMC mode with SanDisk FPGA host adapter */

        #endif

#elif (USE_MMC_EMULATION)

        /* This is for SD/MMC emulation using the PC's Parallel Port */

#endif
```

4.8.3 Porting Requirements

Before making changes to these functions, the user should make their own low level device driver portion work standalone, and then fit the working portion into the **MMCOEM.C** file.

Based on the functional description above, the following is done to the 3 modes:
Note: the USE_SD_MMC, USE_MMC_EMULATION, USE_MEMMODE constants are taken from **SDCONFIG.H.** Please refer to section 4.1 above for details on these definitions.

- **PCI-SD/MMC Mode** - USE_SD_MMC and NOT USE_MEMMODE are used for the SanDisk PCI-SD/MMC mode. It is fixed to the SanDisk PCI_SD Host Adapter, and should not be changed. If there is a need, please contact SanDisk.
- **Native SD/MMC Mode** - Normally, the user will use native SD/MMC mode if their hardware comes with built-in SD/MMC port. Currently this mode is chosen under USE_SD_MMC and USE_MEMMODE. User need to fill their own code into this mode.
- **SD/MMC Emulation Mode** - USE_MMC_EMULATION is used for SanDisk Parallel Port Emulation of SD/MMC mode. It is fixed to the SanDisk Parallel Port Card reader and should not be changed. If there is a need, please contact SanDisk.

As mentioned above, in many cases customers only need to match base addresses in **SDCONFIG.H** to use Native SD/MMC or SD/MMC Emulation mode. To do this, the user should go to **SDCONFIG.H** where he/she will see a similar #if statement for USE_SD_MMC or USE_MMC_EMULATION, USE_MEMMODE, etc. The user will also see definitions for different register sets, etc. defined there. The code itself is self-explanatory.

In spite of the generic aspect of the **MMCOEM.C** file, each system (microprocessor/microcontroller) has unique properties. When the user needs to make changes, he/she should consider that most functions in this file will be called from **MMCDRV.C.** Care should be taken not to change the prototype of functions provided in **MMCOEM.C.** For example,

- mmcresetcontroller()
- mmcSendCommand()
- getMMCResponseInfo()
- MMCTransmit()
- MMCReceive()
- MMCPrepareAndSetup()
- CheckCardBusy()
- isCRCErrOnWrite()
- startMMC80Clocks()
- MMCSelectController()
- ……..

If prototypes of these functions are changed, a compilation/linking error will occur.

The user can make changes inside these functions based on different platforms. He/she can add special functions to be called within these functions, if needed.

4.8.4 Conclusion

To use PCI-SD host adapter and Parallel port SD/MMC emulation mode, normally the user should not change anything.

If Native SD/MMC or SD/MMC Emulation (Not using Parallel port) modes are used, the following describes how to make the required changes:

1. First, the user should make their own low level device driver portion, with register read/write, work standalone.
2. Then, locate the portion in **MMCOEM.C** which choose USE_SD_MMC or USE_MMC_EMULATION, and study the template
3. Go to **SDCONFIG.H** to change the base address, etc., and
4. Go to **MMCOEM.C** to define new data structures, if needed. Carefully make changes inside the existing functions based on the specific microprocessor/micro controller system.

5.0 Peripheral Bus Device Driver

5.1 Introduction

The Peripheral Bus Device Driver is implemented as a storage device driver to support the SanDisk SD Card and MultiMediaCard. The device driver has been designed to be very portable and should be easily adapted to any environment that supports either Memory or I/O Mapped peripheral access, interrupt driven or polled mode.

5.2 Configuring the Peripheral Bus Device Driver

When I/O Mapped Mode is preferred, USE_MEMMODE is set to zero. Two arrays are provided in **IOCONST.C**. These arrays, io_mapped_addresses[] and dev_interrupts[], house the I/O addresses and interrupt numbers used by the low level driver software to control the peripheral controllers. The description of these arrays follows:

const UINT16 io_mapped_addresses[n]	I/O base address of register access region
const INT16 dev_interrupts[n]	Device interrupt numbers

When Memory Mapped Mode is selected, USE_MEMMODE is set to one. Three arrays are used in **IOCONST.C**. These arrays are initialized from constant definitions in **SDCONFIG.H**. The description of these arrays is shown below:

const ULONG mem_mapped_addresses[n]	Linear base address of register access region.
const UTINY FAR *mem_mapped_addresses_pointer[n]	Pointer in the target memory map to the linear system address space.
const INT16 dev_interrupts[n]	Device interrupt numbers.

Note: Set dev_interrupt[n] to -1 to run that interface in polled mode. Define the constant USE_INTERRUPTS to zero in **SDCONFIG.H** to globally disable device interrupts and eliminate all porting issues related to signaling and interrupts.

For an example, look at the SD/MultiMediaCard device driver in I/O Mapped Mode. The default configuration is I/O address 0x278, interrupt 7 for the primary device and I/O address 0x378, interrupt 5 for the secondary. They are shown below:

```
io_mapped_addresses[0] = MMC_PRIMARY_IO_ADDRESS
io_mapped_addresses[1] = MMC_SECONDARY_IO_ADDRESS
dev_interrupts[0] = MMC_PRIMARY_INTERRUPT
dev_interrupts[1] = MMC_SECONDARY_INTERRUPT
```

When the SPI Emulation Device Driver is configured in Memory Mapped Mode, the arrays are initialized with the constants defined in **SDCONFIG.H**. They are shown below.

```
mem_mapped_addresses[0] = SPI_BASE_ADDR
mem_mapped_addresses[1] = SPI_BASE_ADDR
mem_mapped_addresses_pointer[0] = SPI_BASE_ADDR
mem_mapped_addresses_pointer[1] = SPI_BASE_ADDR
dev_interrupts[0] = SPI_PRIMARY_INTERRUPT
dev_interrupts[1] = SPI_SECONDARY_INTERRUPT
```

5.3 Peripheral Bus Device Driver Public Subroutines

The Peripheral Bus Device Driver in SDDK-05 supports different peripheral buses such as SPI and SD/MultiMediaCard. This driver has a uniform interface across these buses. Each peripheral bus interface differs by bus name only. The first three letters of the bus make the bus name. Each peripheral bus name is defined as follows:

SPI	For SPI devices
MMC	For SD/MultiMediaCard devices

There are eight functions for each peripheral bus. They are:

xxx_init
xxx_drive_open
xxx_drive_close
xxx_read
xxx_write
xxx_erase
xxx_read_serial
where xxx is the peripheral bus name.

The function prototypes of each peripheral bus are described below.

xxx_init	Hardware initialization and data structure configuration.
xxx_drive_open	Drive open routine. This routine is called by devio_open() and by the error recovery code in **DEVIO.C** and **CHKMEDIA.C**. It calls xxx_controller_open() to make sure the controller is initialized and then resets the drive, performs diagnostics and retrieves the drive geometry. This routine can be called either on behalf of a drive mount operation or on behalf of recovery logic that senses a drive re-insertion or power down cycle.
xxx_drive_close	Closes down a drive. This routine is only called by the PC based demo programs when they exit. It restores the interrupt management scheme to what it was before xxx_controller_init() was called. Note that it is not necessary to implement this function in an embedded system.
xxx_read	Reads blocks of data from the device.

xxx_write	Writes blocks of data to the device. This routine performs block I/O to and from the device. It is called by devio_read() and devio_write() in **DEVIO.C**. The controller structure contains the flag ‘enable_mapping.’ If this flag is set a partition base is added to the block number. This is the normal case. In special cases such as reading or writing the partition table, the mapping flag is turned off.
xxx_erase	Pre-erases blocks on the device. This routine performs a block pre-erase on the device. It is called by devio_erase() in **DEVIO.C**. The controller structure contains a the flag ‘enable_mapping.’ If this flag is set, a partition base is added to the block number. This function is called by the File System when it erases a file.
xxx_read_serial	Gets the drive’s unique serial number. This routine performs a procedure to read the serial number from the drive. It is used by the check_media() functions (**CHKMEDIA.C**) to detect if a device has been swapped. Beside this serial number, the geometry of the device is also returned.

For example, if the SD/MultiMediaCard interface is selected, the bus interface will have the following supported routines:

mmc_init
mmc _drive_open
mmc _drive_close
mmc _read
mmc _write
mmc _erase
mmc _read_serial

If the SPI interface is selected, the bus interface will have the following supported routines:

spi_init
spi _drive_open
spi _drive_close
spi _read
spi _write
spi _erase
spi _read_serial

6.0 System Internals

The FAT File System is widely accepted in the industry and will continue to be supported for many years to come. To take advantage of this, the SDDK-05 offers a compatible FAT File System that allows the media to be exchanged among many PC systems. There are many books that describe in detail how the FAT file system works. Ray Duncan's *Advanced MS-DOS* programming is particularly good. A brief description follows, but a good DOS book is worth reviewing.

The file system consists of a parameter block (block zero) followed by one or more identical File Allocation Tables (FATs), the fixed size root directory and the data area. The data area is logically broken up into clusters. A cluster is simply one or more contiguous sectors.

The FAT works as follows: Each entry in the FAT maps to a cluster in the heap. If the entry is zero, that cluster is free. If the entry is a special magic number and is unlinked, it marks a bad cluster. Otherwise, the entry contains the index number of the next cluster in the chain. A special terminator value ends the chain.

A file or sub-directory consists of a directory entry plus the FAT index number of the beginning of the chain of clusters for that object.

To access the contents of a file, the user simply gets the first cluster from the directory entry and reads its content from the heap area, then looks in the FAT for the next entry in the chain. If the entry contains a number less than the end of the file marker, read its contents from the heap and get the next cluster from the FAT, and so on. Directory entries contain a file name, creation date and time, an attribute byte, a size and a pointer to the first cluster (if there is one) reserved by the entry.

A sub-directory is a special case of a directory entry. The sub-directory attribute is set in the attribute byte and its clusters contain more directory entries. Sub-directories can grow by claiming more clusters from the heap. Every sub-directory contains two special entries '**.**' and '**..**'. '**..**' points to the first cluster in the chain of the directory's parent and '**.**' points to the beginning of the sub-directory. The root directory contains room for a fixed number of directory entries. The number is determined by the format program and is stored in block zero.

6.1 *Important Data Structures for the FAT File System*

DDRIVE	This structure is initialized when a drive is mounted. It is shared by all tasks. It contains the location of the FAT, the location of the root sector, the cluster size, the disk size and several other drive geometry values. DDRIVE structure also stores running information about the drive, including FAT swapping information and internal hints about where to put new file blocks and how much free space is left on the drive. DDRIVE structure is accessed often by low level routines.
DOSINODE	This structure is the exact image of a DOS directory entry. It is used as a template while scanning directory blocks and as a destination when creating directory entries. A directory entry can be converted to a FINODE structure and worked within that form.
FINODE	This is the central structure of the SDDK-05. It contains the DOSINODE information stored in host byte order plus information about its own block and block offset (where it resides on the disk). It also contains several elements that are used to control shared access to the directory entry. This includes a LOCKOBJ structure, an opencount and a sharing mode flag. All directory and file access routines eventually access the FINODE structure. FINODE structures are shared by all tasks. Each in-use directory entry has one FINODE structure in the shared FINODE pool, no matter how many times it is being accessed by directory scans, or how many times it is open as a file. The directory entry that a FINODE structure represents is uniquely determined by a combination of drive structure pointer, block number and block offset.
DROBJ	This is an abstract structure which tasks use to manage access to directories. Unlike FINODE structures which are shared by all tasks, the DROBJ structure is private to the task that allocated it and is not shared in any way. It contains a pointer to the DDRIVE structure for the mounted volume, a pointer to the FINODE structure of the directory, a pointer to a BLKBUFF structure in the shared buffer pool, and a BLKINFO structure which is used to track the current directory location. During directory searches, BLKBUFF structures attach to the DROBJ as the directory at the FINODE is scanned and then the BLKINFO structure scans the block. This makes the DROBJ a fully self-sufficient data structure for scanning operations. Each open file structure points to a DROBJ structure which represents the file's directory entry. The file accesses the entries FINODE through this linkage and it uses the DROBJ when updating the directory entry on disk.
BLKBUFF	This structure controls directory block buffering. It contains a 512-byte block buffer and control fields. These include the block number, the DDRIVE structure pointer, a lock flag for disallowing swaps while a buffer is being scanned and a few flags for monitoring I/O status. All directory accesses are done through block buffers. File I/O does not use them. BLKBUFFs are shared by all tasks in the system.
PC_FILE	This is the structure that controls file I/O. It contains open flags, a file pointer and some additional pointers for optimizing file I/O. It also contains a pointer to a DROBJ which represents the file's directory entry. A PC_FILE structure is assigned to each file descriptor in the system.

DSTAT	This is the SDDK-05 stat structure. It is loaded by calling pc_gfirst and pc_gnext. These are equivalent to the DosFindFirst and DosFindNext functions of DOS. The DSTAT structure contains the found directory entry’s name, size, datestamp and attributes. The numeric values are in host byte order.
FILE_SYSTEM_USER	This structure contains the current working directory, default drive and current errno for the current task. In single task environments or in environments where these properties may be kept global, there is only one of these structures, otherwise there is one such structure per task. See the Porting and Configuration chapter for more information.

6.2 *System Internals Implementation*

The system internals describe the above data structures and discuss how the SDDK-05 Files System is implemented. The data structures are intended for use with the FAT File System. Please ignore them if the File System is not included. There are several important implementations that need to be addressed. They are as follows:

1. FAT Management Code
2 Directory Block Management Code
3. Directory Object Management Code

The underlying algorithms and data structures associated with those implementations are described below.

6.2.1 *FAT Management Code*

There are a dozen or so functions that manage the FAT. Most are involved in building up and releasing chains but a few are specially optimized routines for allocating and finding contiguous blocks. These routines enhance the performance of the file I/O package. Ultimately, all of these routines call either **pc_faxx** to read values from the FAT or **pc_pfaxx** to write. These two routines then call either **pc_pfgword** or **pc_pfpword** which provide these services through the algorithm.

The FAT swapping/buffer algorithm works as follows: Each time a FAT read or write is requested, the block offset is calculated where that entry resides and **pc_pfswap** is called. **Pc_pfswap** returns a pointer to a memory block where the disk block is cached, if writing, it also sets the blocks dirty bit so it will be flushed when **pc_flushfat** is called. **Pc_pfswap** uses the FATSWAP structure in the DDRIVE structure to manage swapping blocks in and out. First, it looks in **data_array**; if **data_array**[block_offset] is non zero, it uses this as an index into the in-memory array of blocks and returns a pointer to that location. Otherwise, there is a “page fault” condition and it must read the block in. Assuming steady state, the cache area is already filled, so a block must be replaced. To do this, all the dirty blocks are flushed and a block is replaced using a simple round robin algorithm. The replaced block’s entry in **data_array** is set to zero and the new block is read from disk into the freed memory block. This memory block’s offset is now placed in **data_array**[block_offset] and its address is returned. If it is a write request, the block’s dirty bit is set. The routines that manipulate the FAT must be protected from re-entry for all this to work. This is done throughout the SDDK-05. All FAT manipulation routines are in the file **LOWL.C**.

6.2.2 Directory Block Management Code

The SDDK-05 uses a directory block buffer pool for all disk operations concerning directories. This method eliminates excess disk traffic by providing a cache for the most frequently accessed disk blocks. It also helps in multi-tasking implementations by allowing some API calls to complete even though there may be an I/O backlog at the device driver. The buffering algorithm is straight forward. Writes use the write through method. On reads, the buffer pool is searched for the requested block. If the block is not found, a block is selected to be replaced based on an LRU (least recently used) algorithm and the claim that the buffer for the block will be read. When available, the driver initiates the read. There is also a block “alloc” call, this is similar to read but it does not perform the disk I/O. This call is used when initializing a block during a directory create or extend. The block returned by a read or alloc call is in a “locked” state, meaning it is excluded as a candidate for swapping until it is unlocked. This allows scanning and modifying the buffers’ contents directly. There is also code to manage simultaneous read access to blocks. This code handles both I/O waits and I/O errors. Simultaneous write access does not occur. All block buffering routines are in the file **BLOCK.C**.

6.2.3 Directory Object Management Code

This code manages directory scanning, insertion and expansion. There are five basic entry points to this code: **pc_fndnode**, **pc_get_inode**, **pc_mknode**, **pc_rmnode** and **pc_update_inode**. The first two are concerned with finding directory entries and the latter three are used to create, delete and modify them. These routines are used heavily by the API calls.

The scanning code works as follows: **pc_fndnode,** with the help of the string processing routines in **UTILS.C**, parses a path specifier into its drive and path components. It selects the drive to search and the top of the search tree (either root or cwd) based on these values. This results in a search root DROBJ. With this in hand it "nibbles" the path string from left to right, each time getting a directory entry name to search for. It then calls **pc_get_inode** to find the entry in the directory at DROBJ. Each time an entry is found, the entry's DROBJ becomes the new search root. This continues until the path string is exhausted. The calls to **pc_get_inode** are bracketed with non-exclusive semaphores to defend against simultaneous writing to the directory being searched.

The real work is done in **pc_get_inode** and its subordinate **pc_findin**. Keep in mind that **pc_get_inode** is called often by the API as well. The API calls it when a directory entry is to be modified. To do this, the directory it resides in (its parent) must be locked. So **pc_fndnode** is called to find its parent. The parent's FINODE structure is then locked and **pc_findin** is called to find the entry to be changed. After the change is made, **pc_update_inode** flushes it to disk and the parent is released. Then, **Pc_get_inode** grabs a new DROBJ structure and initializes its BLKINFO portion to point to the first block in the subdirectory. Special code makes the root directory and subdirectories appear the same even though their structures are quite different. The DROBJ is now handed off to **pc_findin** which uses it to search for the entry. **Pc_findin** calls the block buffer code to read each block in the directory until it runs out of blocks or it finds the entry being searched. Each time it reads an entry, it updates the entry index counter in the DROBJ's BLKINFO structure. When it hits a block boundary, it clears this counter, increments the block counter and releases the buffer it was scanning. It then calls the buffer code for a new block. Special code detects when a cluster boundary has been hit and adjusts the block counters as needed. If the entry is not found, the routine returns empty handed; otherwise it must make sure that a FINODE structure exists for this directory entry. First it calls **pc_scani** to see if the FINODE already exists. If so, the FINODE's open count is increased and the DROBJ's FINODE pointer is linked to it. Otherwise, the directory entry information is converted from the disk resident form to a similar internal form and copied to a new FINODE structure. This structure is then put on the shared FINODE list by **pc_marki**, so that other instances of **pc_findin** use this shared copy. The DROBJ now fully represents the directory entry and may be used for further scanning and directory manipulation. If the entry is a file, it may be used as the underlying DROBJ for a PC_FILE structure.

When the API functions **po_open** and **pc_mkdir** need to create a directory entry, they find the parent with **pc_fndnode**, lock it and call **pc_mknode** to create the new entry. **Pc_mknode** allocates a DROBJ and FINODE structure and initializes them. If creating a directory entry, it does some manipulations to create the '.' and '..' entries. It then calls **pc_insert_inode** which scans the directory for a deleted or unused entry (see **pc_findin**). If an entry is found it calls **pc_update_inode** to replace it with the new one. If none was found, it calls the FAT management code to extend the directory chain and copies the FINODE into the new cluster. (Root directories of course can not be extended.) Finally, the new FINODE is placed in the shared FINODE pool with **pc_marki**.

Pc_rmnode is used by **pc_rmdir** and **pc_unlink** to remove directories and files respectively. These routines first find the entry using **pc_fndnode** and **pc_findin,** then lock the parent FINODE before making the call. **Pc_rmnode** checks the FINODE open count to be sure no one else is accessing it. Then it calls the FAT management code to free the entry's cluster chain and marks the entry deleted and calls **pc_update_inode** to flush it to disk.

Pc_update_inode uses the DROBJ’s BLKINFO structure and FINODE structure to write a directory entry to disk. It first reads the appropriate block and then merges the FINODE data into the block, converting it from host bytes order to Intel byte order. Finally, it writes the block back out. All directory object code may be found in **DROBJ.C**.

7.0 API Introduction

The SDDK-05 provides a comprehensive Applications Programmer's Interface (API) for accessing and manipulating data on storage devices. The API provides two methods to achieve this goal:

- FAT File System: The combination of the File System and low level driver to manage data at high level.
- Peripheral Bus Interface: The low level driver to directly access to the storage device.

7.1 File System

The File System Initialization and Close are two routines that must be called before entering and exiting the application respectively. The initialization process must be done first before the File System is used. The routine pc_system_init has to be called first to initialize internal memory buffers for use by the rest of the routines. The closing process must be done after the File System is used. The routine pc_system_close has to be called at the end to release internal memory buffers used by the File System. The API is re-entrant so multiple tasks may access the File System simultaneously.

Programmers with experience on DOS, Unix, Posix or any other "normal" operating system should have very little trouble programming to the API since it is similar to those APIs. The SDDK-05 purposely does not match the Posix/Unix API. This is because in many cases the SDDK-05 co-resides with another File System and we did not want to have symbol/constant clashes. If the user is doing a large port that would benefit from full compatibility, call SanDisk Applications Engineering at 408-542-0405.

Note: The programs in the samples directory provide a very good framework for using the API. The user should use them as a resource for resolving any questions he/she might have.

7.1.1 pc_cluster_size

Name:

pc_cluster_size - Return a drive's cluster size.

Summary:

```
#include "sdapi.h"
UINT16 pc_cluster_size(INT16 driveno)
```

Description:

This function will return the cluster size of the mounted device named in the argument.

Returns:

The cluster size or zero if the device is not mounted.

See Also:

po_extend_file

Example:

Given a byte count, calculate by rounding up how many clusters to extend a file by and then extend the file.

```
#include "sdapi.h"
UINT16      cluster_size;
UINT16      n_clusters;
cluster_size = pc_cluster_size(0);
n_clusters = (n_to_write + cluster_size - 1) / cluster_size;
po_extend_file (fd, n_clusters, PC_FIRST_FIT, preerase_region);
```

7.1.2 pc_diskabort

Name:

pc_diskabort - Abort operations on a disk.

Summary:

```
#include “sdapi.h”
SDVOID pc_diskabort(INT16 driveno)
```

Description:

If an application senses that there are problems with a disk, it should call **pc_diskabort**(“D:”). This will cause all resources associated with that drive to be freed, but no disk writes will occur. All file descriptors associated with the drive become invalid.

Returns:

Nothing

Example:

```
#include “sdapi.h”
if (ask_driver_if_there_is_a_problem(0))
{
        pc_diskabort(0);
        if (drive_clear_error(0))
                pc_dskopen(0);
}
```

Note: It should not be necessary to call this routine. The card management software in **CHKMEDIA.C** handles this function.

7.1.3 pc_dskclose

Name:

pc_dskclose - Flush buffers and free core.

Summary:

#include “sdapi.h”

SDBOOL **pc_dskclose**(INT16 driveno)

Description:

Given a path name containing a valid drive letter, flush the file allocation table and purge any buffers or objects associated with the drive. Also, make sure all changed files are flushed to disk.

Returns:

Returns YES if successful.

Example:

#include “sdapi.h”

pc_dskclose(0);

Note: This function is useful when the user knows that he/she will be removing the drive. In practice it is not absolutely necessary to call this function since the card management code in **CHKMEDIA.C** will free the resources for the user. See **PC_DSKFLUSH.C**.

7.1.4 pc_diskflush

Name:

pc_diskflush - Flush the FAT and all files on a disk.

Summary:

#include "sdapi.h"
SDBOOL pc_diskflush(INT16 driveno)

Description:

Given a path containing a valid drive letter flush the file allocation table and all changed files to the disk. After this call returns, the disk image is synchronized with the SDDK-05 internal view of the volume.

Returns:

YES if the disk flush succeeded, otherwise NO

Example:

```
 #include "sdapi.h"
if (!pc_diskflush(0))
     printf("Flush operation failed \n");
```

7.1.5 pc_format

Name:

pc_format - Format the device.

Summary:

```
#include “sdapi.h”
SDBOOL pc_format (INT16 driveno)
```

Description:

Given a string containing a valid drive letter, place a standard partition and volume structure on the drive.

Returns:

Returns YES if the format succeeded, otherwise NO.

Example:

```
#include “sdapi.h”
if (!pc_format (0))
     printf (“could not format A: \n”);
```

7.1.6 pc_free

Name:

pc_free - Return count of free bytes on a disk.

Summary:

#include “sdapi.h”
ULONG **pc_free**(INT16 driveno)

Description:

Given a path containing a valid drive letter, count the number of free bytes on the drive.

Note: The first time this routine is called after pc_dskinit it must scan the whole file allocation table to calculate the number of free clusters, this takes some time. Subsequent calls return immediately with a valid value.

Returns:

The number of bytes available on the disk.

Example:

```
#include “sdapi.h”
    printf (“%lu Bytes Free on A:”, pc_free(0));
```

7.1.7 pc_fstat

Name:

pc_fstat - Obtain statistics on an open file.

Summary:

```
#include "sdapi.h"
INT16 pc_fstat(PCFD file_descriptor, STAT *pstat)
```

Description:

This routine uses the file descriptor in the first argument and fills in the stat structure as described here.

st_dev	-	the entry's drive number
st_mode	-	Type of File Supported
S_IFMT		type of file mask
S_IFCHR		character special (unused)
S_IFDIR		directory
S_IFBLK		block special (unused)
S_IFREG		regular (a "file")
S_IWRITE		Write permitted
S_IREAD		Read permitted.
st_rdev	-	the entry's drive number
st_size	-	file size
st_atime	-	creation date in DATESTR format
st_mtime	-	creation date in DATESTR format
st_ctime	-	creation date in DATESTR format
t_blksize	-	optimal blocksize for I/O (cluster size)
t_blocks	-	blocks allocated for file
fattributes	-	the DOS attributes. This is non-standard but supplied if the user wants to look at them.

Returns:

Returns zero if all went well otherwise it returns -1 and fs_user->p_errno is set to this value:

PENBADF - Invalid file descriptor

Example:

```
#include "sdapi.h"
struct stat st;
PCFD fd;
 fd = po_open("A:\\MYFILE.TXT", (PO_BINARY|PO_RDONLY), 0);
 if (pc_fstat(fd, &st)==0)
 {
    printf("DRIVENO: %02d SIZE: %7ld DATE:%02d-%02d-%02d  TIME:%02d:%02d\n",
    st.st_dev,
    st.st_size,                               /* Size in bytes */
    (st.st_atime.date >> 5 ) & 0xF,       /* Month */
    (st.st_atime.date & 0x1F),            /* Day */
    80 +(st.st_atime.date >> 9) & 0xFF,           /* Year */
    (st.st_atime.time >> 11) & 0x1F,    /* Hour */
    (st.st_atime.time >> 5) & 0x3F);    /* Minute */
    printf("OPTIMAL BLOCK SIZE: %7ld FILE size (BLOCKS): %7ld\n",
          st.st_blksize, st.st_blocks);
    printf(“MODE BITS :”);
    if (st.st_mode&S_IFDIR)
       printf(“S_IFDIR”);
    if (st.st_mode&S_IFREG)
       printf(“ | S_IFREG”);
    if (st.st_mode&S_IWRITE)
       printf(“ | S_IWRITE”);
    if (st.st_mode&S_IREAD)
       printf(“ | S_IREAD\n”);
    printf(“\n”);
}
```

7.1.8 pc_gdone

Name:

pc_gdone - Free pc_gnext and pc_gfirst resources.

Summary:

#include “sdapi.h”
SDVOID **pc_gdone**(DSTAT *statobj)

Description:

Given a pointer to a DSTAT structure that was set up by a call to **pc_gfirst** free internal elements used by the statobj.

Note: The user MUST call this function after having finished calling pc_gfirst and pc_gnext.

Returns:

Nothing.

Example:

See pc_gnext

7.1.9 pc_get_attributes

Name:

pc_get_attributes - Get file attributes**.**

Summary:

#include "sdapi.h"
SDBOOL pc_get_attributes**(TEXT *path, UINT16 *p_return);**

Description:

Given a file or directory name, returns the directory entry attributes associated with the entry.
One or more of the following values will be or'ed together:

BIT	Nemonic
0	ARDONLY
1	AHIDDEN
2	ASYSTEM
3	AVOLUME
4	ADIRENT
5	ARCHIVE

Returns:

Returns YES if successful otherwise it returns NO and fs_user->p_errno is set to this value:
PENOENT

Example:

```
    #include "sdapi.h"
UTINY attribs;
    if (pc_get_attributes("A:\\COMMAND.COM", &attribs)
    {
        if (attribs & ARDONLY)
                printf("File is %s\n", "ARDONLY");
        if (attribs & AHIDDEN)
                printf("File is %s\n", "AHIDDEN");
        if (attribs & ASYSTEM)
                printf("File is %s\n", "ASYSTEM");
        if (attribs & AVOLUME)
                printf("File is %s\n", "AVOLUME");
        if (attribs & ADIRENT)
                printf("File is %s\n", "ADIRENT");
        if (attribs & ARCHIVE)
                printf("File is %s\n", "ARCHIVE");
    }
```

7.1.10 pc_gfirst

Name:

pc_gfirst - Return the first entry in a directory.

Summary:

#include "sdapi.h"
SDBOOL pc_gfirst(DSTAT *statobj, TEXT *pattern)

Description:

Given a pattern which contains both a path and a search pattern, fills in the structure at statobj with information about the file and sets up internal parts of statobj to supply appropriate information for calls to pc_gnext.

Examples of patterns are:

"D:\USR\RELEASE\NETWORK\*.C"
"BIN\UU*.*"
"MEMO_?.*"
"*.*"

Note: The user must call pc_gdone to free internal resources if pc_gfirst succeeds.

Returns:

YES if a match was found, otherwise returns NO.

See Also:

pc_gnext, **pc_gdone**, and **pcls.c** in the samples directory.

Example*:*

See PC_GNEXT

7.1.11 pc_gnext

Name:

pc_gnext - Return next entry in a directory.

Summary:

```
#include “sdapi.h”
SDBOOL pc_gnext(DSTAT *statobj)
```

Description:

Continues with the directory scan started by a call to **pc_gfirst**.

Returns:

YES if a match was found, otherwise returns NO.

See Also:

pc_gnext and **pc_gdone** in the samples directory.

Example:

```
#include “sdapi.h”

if (pc_gfirst(&statobj,”A:\\dev\\*.c”))
{
    do
    {
            /* print file name, extension and size */
            printf(“%-8s.%-3s %7ld \n”,statobj.fname, statobj.fext, statobj.fsize);
    } while (pc_gnext(&statobj));

    /* Call gdone to free up internal resources */
    pc_gdone(&statobj);
}
```

7.1.12 pc_isdir

Name:

pc_isdir - Test if a path is a directory.

Summary:

```
#include “sdapi.h”
SDBOOL pc_isdir(TEXT *path)
```

Description:

This is a simple routine that opens a path and checks if it is a directory, then closes the path. The program cp2pc in the samples directory uses it to test if a destination is a directory. The same functionality can be gotten by calling **pc_gfirst** and testing the DSTAT structure.

Returns:

YES if path points to a valid existing directory, otherwise returns NO.

Example:

```
#include “sdapi.h”

if (pc_isdir(path))
{
    printf(“ This %s is a directory. \n”, path);
}
```

7.1.13 pc_mfile

Name:

pc_mfile - Build a complete path from file name and extension.

Summary:

#include “sdapi.h”

TEXT *pc_mfile(TEXT *to, TEXT *filename, TEXT *ext)

Description:

Builds a file from a file name and extension. The file name is stored into the text string.

Returns:

The file name.

Example:

#include “sdapi.h”

TEXT to[128];
pc_mfile(to, filename, ext);

7.1.14 pc_mpath

Name

pc_mpath - Build a specific path name.

Summary:

#include “sdapi.h”

TEXT *pc_mpath(TEXT *to, TEXT *path, TEXT *filename)

Description:

Builds a specific path from a file and path name. The resulting name is stored into the text string.

Returns:

The file name.

Example:

#include “sdapi.h”

TEXT to[128];
pc_mfile(to, filename, ext);

7.1.15 pc_mkdir

Name:

pc_mkdir - Create a subdirectory.

Summary:

#include "sdapi.h"
SDBOOL **pc_mkdir**(TEXT *path)

Description:

Creates a sub-directory in the path specified by path. Fails if a file or directory of the same name already exists or if the directory component (if there is one) of path is not found.

Returns:

Returns YES if the subdirectory was created.
If NO is returned, **fs_user->p_errno** will be set to one of these values:

PENOENT	- Directory not found
PEEXIST	- File or directory already exists
PENOSPC	- Write failed

Example:

#include "sdapi.h"
pc_mkdir("\\USR\\LIB\\HEADER\\SYS");

7.1.16 pc_mv

Name:

pc_mv - Rename a file or directory.

Summary:

```
#include “sdapi.h”
SDBOOL pc_mv(TEXT *oldpath, TEXT *newname)
```

Description:

Renames the file oldpath to newname. Fails if newname is invalid, already exists or oldpath is not found. **Pc_mv** does not test if oldpath is a simple file. This makes it possible to rename directories and volume labels.

Returns:

YES if the file was renamed. Or returns no if oldpath was not found.

If NO is returned, **fs_user->p_errno** will be set to one of these values:

PENOENT	- oldpath not found
PEEXIST	- newname already exists
PENOSPC	- Write failed

Example:

```
#include “sdapi.h”
if (!pc_mv(“\\USR\\TXT\\LETTER.TXT”, “LETTER.OLD”))
    printf(“Can’t rename LETTER.TXT\n”);
```

7.1.17 pc_system_init

Name:

pc_system_init - Initialize internal memory buffers and mount a drive.

Summary:

#include “sdapi.h”
SDBOOL pc_system_init(INT16 driveno);

Description:

Initializes the internal memory, searches for the device, checks for a valid drive to mount the device.

Returns:

YES if successful.
NO if failure.

Example:

```
#include “sdapi.h”
if (!pc_system_init(0))
{
    printf(“File System initialization failed. \n”);
    return(NO);
}
```

7.1.18 pc_system_close

Name:

pc_system_close - Release internal memory buffers and dismount a drive.

Summary:

```
#include “sdapi.h”
SDBOOL pc_system_close(INT16 driveno);
```

Description:

Clears all File System data structures and unmounts a selected drive.

Returns:

YES if closing is successful.
NO if closing failed.

Example:

```
#include “sdapi.h”
if (!pc_system_close(0)
{
    printf(“Unable to release all internal buffers. \n”);
    return(NO);
}
```

7.1.19 pc_pwd

Name:

pc_pwd - Return the current working directory.

Summary:

```
#include “sdapi.h”
SDBOOL pc_pwd(TEXT *drive, TEXT *return_buffer)
```

Description:

Fills return_buffer with the full path name of the current working directory for the drive specified in drive. If the drive points to an empty string **(“”)** or an invalid drive letter, the current working directory for the default drive is returned.

Note: Return buffer must contain enough space to hold the full path.

Returns:

YES if a valid path was returned in **return_buffer**.
Returns NO if the current working directory could not be found. The failure mode would be due to either the fact that the drive is not mounted, or an I/O error occurred.

Example:

```
#include “sdapi.h”
TEXT  pwd[EMAXPATH];
if (pc_pwd(“A:”, pwd))
    printf (“Working dir is %s\n”,  pwd);
else
    printf (“Can’t find working directory. \n”);
```

7.1.20 pc_rmdir

Name:

pc_rmdir - Delete a directory

Summary:

```
#include “sdapi.h”
SDBOOL pc_rmdir(TEXT *path)
```

Description:

Deletes the directory specified in the path. Fails if path is not a directory, is read only or is not empty.

Returns:

YES if the directory was successfully removed otherwise returns NO.

If NO is returned **fs_user->p_errno** will be set to one of these values:

PENOENT	- Directory not found
PEACCES	- Not a directory, not empty or in use
PENOSPC	- Write failed

Example:

```
#include “sdapi.h”
if (!pc_rmdir(“D:\\USR\\TEMP”)
    printf(“Can’t delete directory. \n”);
```

7.1.21 pc_set_attributes

Name:

pc_set_attributes - Set File Attributes

Summary:

#include "sdapi.h"

SDBOOL pc_set_attributes**(TEXT *path, UINT16 attributes);**

Description:

Given a file or directory name, sets the directory entry attributes associated with the entry.

One or more of the following values may be or'ed together

BIT	Nemonic
0	ARDONLY
1	AHIDDEN
2	ASYSTEM
5	ARCHIVE

Returns:

Returns YES if successful, otherwise returns NO and fs_user->p_errno is set to one of these values:

PENOENT	-	Couldn't find the entry
PENOSPC	-	Write failed

Example:

```
 #include "sdapi.h"
 UTINY attribs;
    if (pc_get_attributes("A:\\COMMAND.COM", &attribs)
    {
            attribs |= ARDONLY|AHIDDEN
            pc_set_attributes("A:\\COMMAND.COM", attribs);
    }
```

7.1.22 pc_set_cwd

Name:

pc_set_cwd - Set current working directory

Summary:

```
#include “sdapi.h”
SDBOOL pc_set_cwd(TEXT *path)
```

Description:

Makes the path the current working directory. If the path contains a drive component, the current working directory is changed for that drive. Otherwise, the current working directory is changed for the default drive.

Returns:

Returns YES if the current working directory was changed otherwise returns NO.
If NO is returned fs_user->p_errno will be set to this value:
PENOENT - Directory not found

Example:

```
#include “sdapi.h”
if(!pc_set_cwd(“D:\\USR\\DATA\\FINANCE”))
    printf(“Can’t change working directory. \n”);
```

7.1.23 pc_set_default_drive

Name:

pc_set_default_drive - Set the default drive

Summary:

```
#include “sdapi.h”
SDBOOL pc_set_default_drive(INT16 driveno)
```

Description:

Use this function to set the current default drive that will be used when a path does not contain a drive letter.

Note: Before this function is called, the default is 0.

Returns:

NO if the drive is invalid or not mounted.

Example:

```
#include “sdapi.h”
if(!pc_set_default_drive(0))
    printf(“Can’t change working drive\n”);
```

7.1.24 pc_stat

Name:

pc_stat - Obtain statistics on a path

Summary:

#include "sdapi.h"
INT16 pc_stat(TEXT *path, STAT *pstat)

Description:

This routine searches for the file or directory provided in the first argument.
If found it fills in the stat structure as described here:

st_dev	-	The entry's drive number
st_mode	-	File types
S_IFMT		type of file mask
S_IFCHR		character special (unused)
S_IFDIR		directory
S_IFBLK		block special (unused)
S_IFREG		regular (a "file")
S_IWRITE		Write permitted
S_IREAD		Read permitted.
st_rdev	-	The entry's drive number
st_size	-	file size
st_atime	-	creation date in DATESTR format
st_mtime	-	creation date in DATESTR format
st_ctime	-	creation date in DATESTR format
t_blksize	-	optimal blocksize for I/O (cluster size)
t_blocks	-	blocks allocated for file
fattributes	-	The DOS attributes. This is non-standard but supplied if the user wants to look at them

Returns:

Returns zero if successful, otherwise -1 and fs_user->p_errno is set to one of these values:
PENOENT

Example:

```
#include "sdapi.h"
struct stat st;
if (pc_stat("A:\\MYFILE.TXT", &st)==0)
{
    printf("DRIVENO: %02d SIZE: %7ld DATE:%02d-%02d-%02d  TIME:%02d:%02d\n",
  st.st_dev,
  st.st_size,                                   /* Size in bytes */
  (st.st_atime.date >> 5 ) & 0xF,       /* Month */
  (st.st_atime.date & 0x1F),                  /* Day */
  80 +(st.st_atime.date >> 9) & 0xFF,         /* Year */
  (st.st_atime.time >> 11) & 0x1F,   /* Hour */
  (st.st_atime.time >> 5) & 0x3F);   /* Minute */
  printf("OPTIMAL BLOCK SIZE: %7ld FILE size (BLOCKS): %7ld\n",
       st.st_blksize, st.st_blocks);
  printf("MODE BITS :");
  if (st.st_mode & S_IFDIR)
```

```
        printf(“S_IFDIR”);
    if (st.st_mode & S_IFREG)
        printf(“ | S_IFREG”);
    if (st.st_mode & S_IWRITE)
        printf(“ | S_IWRITE”);
    if (st.st_mode & S_IREAD)
        printf(“ | S_IREAD \n”);
    printf(“\n”);
}
```

7.1.25 pc_unlink

Name:

pc_unlink - Delete a file

Summary:

#include “sdapi.h”
SDBOOL **pc_unlink**(TEXT *path)

Description:

Deletes the file in name. Fails if not a simple file, if it is open, does not exist or is read only.

Returns:

YES if it successfully deleted the file**.**
If NO is returned **fs_user->p_errno** will be set to one of these values:

PENOENT	-	File not found
PEACCES	-	Is a directory or an open file
PENOSPC	-	Write failed

Example:

```
if (!pc_unlink(“B:\\USR\\TEMP\\TMP001.PRN”) )
    printf(“Cant delete file \n”)
```

7.1.26 po_close

Name:

po_close - Close a file

Summary:

#include “sdapi.h”
INT16 **po_close**(PCFD fd)

Description:

Closes the file and updates the disk by flushing the directory entry and file allocation table, then frees all core associated with FD.

Returns:

Zero if all went well otherwise it returns -1.
If -1 is returned, **fs_user->p_errno** will be set to one of these values:

PENBADF	-	Invalid file descriptor
PENOSPC	-	I/O error occurred

See Also:

po_flush

Example:

```
#include “sdapi.h”
if (po_close(fd) < 0)
    printf(“Error closing file:%i\n”,p_errno);
```

7.1.27 po_extend_file

Name:

po_extend_file - Contiguous File Extend

Summary:

```
#include "sdapi.h"
UINT16 po_extend_file(PCFD fd,
                      UINT16 n_clusters,
                      INT16 method,
                      SDBOOL preerase_region)
```

Description:

Given a file descriptor, n_clusters clusters and method, extends the file and updates the file size. If n_clusters free contiguous clusters are not available, then the file is not extended and the size in clusters of the largest contiguous block of free clusters is returned. If the pre-erase region is set, all allocated sectors are pre-erased.

Note: The file pointer is unchanged.

Method may be one of the following:

PC_FIRST_FIT	-	The first chain >= n_clusters
PC_BEST_FIT	-	The smallest chain >= n_clusters
PC_WORST_FIT	-	The largest chain >= n_clusters

Note: PC_FIRST_FIT is significantly faster than the others.

Returns:

Returns n_clusters if the file was extended. Otherwise it returns the largest free chain available. If n_clusters is not returned, the file was not extended. 0xFFFF is returned if an error occurred. If the return value is 0xFFFF, **n_clusters fs_user**->p_errno will be set with one of the following:

PENBADF	-	File descriptor invalid or open read only
PENOSPC	-	I/O failure

Example:

Allocate a 100 Kbyte contiguous file, perform a data collect and write it out.

```
ULONG ltemp= 102400L;
UINT16 n_clusters;
UINT16 cluster_size;
INT16 i;
UTINY buffer[10240];

cluster_size = pc_cluster_size("C:);
ltemp += cluster_size - 1;
n_clusters = ltemp / cluster_size;
if(po_extend_file(fd, n_clusters, PC_FIRST_FIT, 0) == n_clusters)
{
    for (i = 0; i < 10; i++)
    {
            collect_10k(buffer);
            po_write(fd, buffer, 10240);
    }
}
```

7.1.28 po_flush

Name:

po_flush - Flush a file to disk

Summary:

#include “sdapi.h”
SDBOOL **po_flush**(PCFD fd)

Description:

Writes the file’s directory entry to disk and flushes the FAT. After this call completes, the on disk view of the file is completely consistent with the in memory view. It is a good idea to call this function periodically if a file is being extended. If a file is not flushed or closed and a power down occurs, the file size will be wrong on disk and the FAT chains will be lost.

Returns:

Returns YES if flush succeeded.
If NO is returned, **fs_user->p_errno** will be set to one of these values:

PENBADF	-	Invalid file descriptor
PENOSPC	-	I/O error occurred

Example:

```
#include “sdapi.h”
if (po_flush(fd) < 0)
    printf(“Error flushing file:%i\n”,p_errno);
```

See Also:

pc_dskflush()

7.1.29 po_lseek

Name:

po_lseek - Move file pointer

Summary:

#include “sdapi.h”
ULONG **po_lseek**(PCFD fd, INT32 offset, INT16 origin, INT16 *err_flag)

Description:

Moves the file pointer by offset bytes described by origin.
<u>**method**</u> may have the following values:

PSEEK_SET	**-**	Seek from beginning of file
PSEEK_CUR	**-**	Seek from the current file pointer
PSEEK_END	**-**	Seek from end of file

Attempting to seek beyond end of file puts the file pointer one byte past end of file.

Returns:

The new offset or -1 on error.
If -1 is returned, **fs_user->p_errno** will be set to one of these values**:**

PENBADF	-	File descriptor invalid
PEINVAL	-	Seek to negative file pointer
err_flag	-	0 no error; 1 file not exist; -1 illegal file offset pointer

Example:

```
#include “sdapi.h”
record = rec_number * rec_size;
if (po_lseek (fd, record , PSEEK_SET, & err_flag) != record)
    printf(“Cant find record %ld\n”, record);
```

7.1.30 po_open

Name:

po_open - Open a file

Summary:

#include "sdapi.h"
PCFD **po_open**(TEXT *path, UINT16 flag, UINT16 mode)

Description:

Opens the file for access as specified in a flag. If creating, use mode to set the access permissions.

<u>Flag values are</u>:

PO_BINARY	-	Ignored
PO_TEXT	-	Ignored
PO_RDONLY	-	Open for read only
PO_RDWR	-	Read/write access allowed.
PO_WRONLY	-	Open for write only
PO_CREAT	-	Create the file if it does not exist.
PO_EXCL	-	If flag has (PO_CREAT|PO_EXCL) and the file already exists, fail and set **fs_user->p_errno** to EEXIST.
PO_TRUNC	-	Truncate the file if it already exists
PO_NOSHAREANY	-	Fail if already open, fail if another open is tried.
PO_NOSHAREWRITE	-	Fail if already open for write. And fail if another open for write is tried.

<u>Mode values are:</u>

PS_IWRITE	-	Write permitted
PS_IREAD	-	Read permitted. (Always true anyway)

Returns:

Returns a non-negative integer to be used as a file descriptor for calling **po_read**, **po_write**, **po_seek**, **po_flush**, **po_truncate**, and **po_close** otherwise it returns -1 and **fs_user->p_errno** is set to:

PENOENT	-	File not found or path to file not found
PEMFILE	-	No file descriptors available (too many files open)
PEEXIST	-	Exclusive access requested but file already exists.
PEACCESS	-	Attempt to open a read only file or a special (directory)file.
PENOSPC	-	Create failed
PESHARE	-	Sharing error. Already open exclusive or attempting to open exclusive and the file is already open.

Example:

```
#include "sdapi.h"
PCFD fd;
if ( fd = po_open("\\USR\\MYFILE",
                  (PO_CREAT|PO_EXCL|PO_WRONLY),
                  PS_IWRITE) < 0 )
    printf("Cant create file error:%i\n", fs_user->p_errno)
```

7.1.31 po_read

Name:

po_read - Read from a file

Summary:

```
#include “sdapi.h”
UCOUNT po_read(PCFD fd,  UCHAR *buf, UCOUNT count)
```

Description:

Attempts to read **count** bytes from the file at **fd** and place the data in **buf**. The file pointer is updated.

Returns:

Returns the actual number of bytes read or 0xFFFF on error. If the return value is 0xFFFF **fs_user->p_errno** will be set to one of the following:

ENBADF	-	File descriptor invalid
PENOSPC	-	File I/O error

Example:

```
PCFD fd;
PCFD fd2;
fd = po_open(“FROM.FIL”, PO_RDONLY, 0);
fd2 =po_open(“TO.FIL”, PO_CREAT|PO_WRONLY, PS_IWRITE)
if (fd >= 0 && fd2 >= 0)
{
    while (po_read(fd, buff, 512) ==512)
        po_write(fd2, buff, 512);
}
```

7.1.32 po_truncate

Name:

**po_truncate - ** Truncate a file

Summary:

#include “sdapi.h”
SDBOOL **po_truncate**(PCFD fd, LONG newsize)

Description:

Truncates the open file at fd to newsize. Any file blocks beyond newsize are freed and the file size is adjusted. The file pointer is left at the new end of the file.

Returns:

Returns YES if po_truncate succeeded.
If NO is returned, **fs_user->p_errno** will be set to one of these values:

PENBADF	-	Invalid file descriptor or open read only
PENOSPC	-	I/O error occurred
PEINVAL	-	Newsize is invalid
PESHARE	-	The file is open with another handle. Truncate is not permitted of the file is open more then one.

Example:

PCFD fd;
fd = **po_open**(“DATA.FIL”, **PO_RDWR**, 0);
if (fd > 0)
    po_truncate(fd, 1024L);

7.1.33 po_write

Name:

po_write **-** Write to a file

Summary:

```
#include “sdapi.h”
UCOUNT po_write(PCFD fd, UCHAR *buf, UCOUNT bytes_to_write)
```

Description:

Attempts to write bytes_to_write from buf to the current file pointer of file at fd. The file pointer is updated.

Returns:

Returns the number of bytes written or 0xFFFF on error. If the returned value is 0xFFFF, **fs_user->p_errno** will be set with one of the following:

PENBADF - File descriptor invalid or open read only

PENOSPC - Write failed because of no space or an I/O error.

Example:

```
PCFD fd;
PCFD fd2;
fd = po_open(“FROM.FIL”, PO_RDONLY, 0);
fd2 =po_open(“TO.FIL”, PO_CREAT|PO_WRONLY, PS_IWRITE)
if (fd >= 0 && fd2 >= 0)
{
    while (po_read(fd, buff, 512) == 512)
        po_write(fd2, buff, 512);
}
```

7.2 *Peripheral Bus Interface*

The Peripheral Bus Interface is a generic interface that supports the SanDisk SD Card and MultiMediaCard. The bus interface is very simple and very easy to port to different storage products. The SDDK-05 currently supports two different bus interfaces: SPI and SD/MultiMediaCard. The bus protocol always begins with the first three letter of the bus type. They are describes as follows.

xxx_init
xxx_drive_open
xxx_drive_close
xxx_read
xxx_write
xxx_erase
xxx_read_serial
where xxx is the bus type.

The following table is the summary of the bus interfaces.

Bus Type	xxx
SPI	spi
SD/MultiMediaCard	mmc

The bus protocol is described in detail in the following sections.

7.2.1 xxx_init

Name:

xxx_init - Hardware configuration and set up internal information.

Summary:

```
#include “sdapi.h”
SDBOOL xxx_init(VOID);
```

Description: Hardware initialization process for a selected bus protocol.

Returns:

YES if successful
NO if failure

Example:

```
#include “sdapi.h”
if (!xxx_init())
{
     printf(“Initialization process failed. \n”);
     return(NO);
}
```

7.2.2 xxx_drive_open

Name:

xxx_drive_open - Initialize a device

Summary:

```
#include “sdapi.h”
SDBOOL xxx_drive_open(INT16 driveno);
```

Description: Initializes the device and sets up internal data structure.

Returns:

YES if successful
NO if failure

Example:

```
#include “sdapi.h”
INT16 driveno;
driveno = 0;
if (!xxx_drive_open(driveno))
{
    printf(“ Failed to initialize the device %d. \n”, driveno);
    return(NO);
}
```

7.2.3 xxx_drive_close

Name:

xxx_drive_close - Release internal buffer for a selected device

Summary:

```
#include “sdapi.h”
SDBOOL xxx_drive_close(INT16 driveno);
```

Description: Closes and releases internal structure on the selected device.

Returns:

YES if successful
NO if failure

Example:

```
#include “sdapi.h”
INT16 driveno;

driveno = 0;
if (!xxx_drive_close(driveno))
{
    printf(“Unable to release the device %d \n”, driveno);
    return(NO);
}
```

7.2.4 *xxx_read*

Name:
xxx_read **-** Read data from a select device

Summary:
```
#include "sdapi.h"
SDBOOL xxx_read(INT16 driveno, ULONG lba, UCHAR *buffer, UCOUNT no_blocks);
```

Description: Reads no_blocks data from a selected device beginning at sector LBA and stores information into buffer.

Returns:
YES if successful
NO if failure

Example:
```
#include "sdapi.h"
INT16 driveno;
ULONG lba;
UCOUNT number_of_blocks;
UCHAR buffer[1024];
driveno = 0;
lba = 0L;
number_of_blocks = 2;
if (!xxx_read(driveno, lba, buffer, number_of_blocks))
{
    printf(" Failed to read from sector %ld \n", lba);
    return(NO);
}
```

7.2.5 xxx_write

Name:

xxx_write - Write data to a selected device

Summary:

```
#include "sdapi.h"
SDBOOL xxx_write(INT16 driveno, ULONG lba, UCHAR *buffer, UCOUNT no_blocks);
```

Description: Writes no_blocks of data to a selected device beginning at the sector LBA from the buffer.

Returns:

YES if successful
NO if failure

Example:

```
#include "sdapi.h"
INT16 driveno;
ULONG lba;
UCOUNT number_of_blocks;
UCHAR buffer[2048];
driveno = 0;
lba = 1L;
number_of_blocks = 4;
if (!xxx_write(driveno, lba, buffer, number_of_blocks))
{
    printf(" Failed to write to sector %ld \n", lba);
    return(NO);
}
```

7.2.6 xxx_erase

Name:

xxx_erase - Erase data to a selected device

Summary:

```
#include "sdapi.h"
SDBOOL xxx_erase(INT16 driveno, ULONG lba, UCOUNT no_blocks);
```

Description: Erases no_blocks of data from a selected device beginning at the sector LBA.

Returns:

Example:

```
#include "sdapi.h"
INT16 driveno;
ULONG lba;
UCOUNT number_of_blocks;
driveno = 0;
lba = 0L;
number_of_blocks = 2;
if (!xxx_erase(driveno, lba, number_of_blocks))
{
    printf(" Failed to erase data beginning at sector %ld \n", lba);
    return(NO);
}
```

7.2.7 *xxx_read_serial*

Name:

**xxx_read_serial - ** Get device geometry

Summary:

```
#include "sdapi.h"
SDBOOL xxx_read_serial(INT16 driveno, PDRV_GEOMETRY_DESC drv_geometry);
```

Description: Given a selected device, the device geometry is returned. If the request is not granted, the drv_geometry fields are set to zero.

Returns:

YES if successful
NO if failure

Example:

```
#include "sdapi.h"
INT16 driveno;
DRV_GEOMETRY_DESC drv_geometry;
driveno = 0;
if (!xxx_read_serial(driveno, &drv_geometry))
{
    printf(" Failed to get the device %d geometry \n", driveno);
    return(NO);
}
```

8.0 Sample Utility Programs

8.1 Introduction

In this section, the user is provided some useful programs that he/she may use right out of the box or modify for his/her purposes. Even if the user doesn't use these programs, he/she should definitely study them before using the SDDK-05. Every API call is used in these programs and they all work, so the user is encouraged to take advantage of them.

Along with the sample programs, a few useful test programs and a tool to aid conversion of ANSI 'C' programs to K&R 'C' are provided. In this section, some liberties were taken that were not taken in the library, namely some ANSI string handling functions are called, without providing portable versions of all of those functions. There aren't too many and it is hoped it will not be too much of an inconvenience. Most of these tools require **printf** to link. The mini **printf,** in the source directory, is adequate for these programs. The program **REGRESS.C** is designed to work without **printf**. This should be used to test the user's port on deeply embedded systems. If a console of some sort is available to the user, the program **TSTSH.C** is a very powerful tool for debugging and testing his/her port.

8.2 CPTOSD

Name:

CPTOSD - Copy host files to the device with specific directory

Summary:

cptosd [-b] file file file ... destpath
-b = binary, don’t convert \n to \n\r

Description:

Copies the file(s) from the host to a PC destination path or filename. If the -b flag is asserted, files are copied verbatim, otherwise MS-DOS style \n\r combinations are created from \n. If destpath is a PC directory then the file(s) will be copied to that directory.
If MS-DOS is defined during compilation, binary copy mode is always used.
During compilation the SDDK-05 tests for the MS-DOS predefined macro. If there, it builds for DOS. Otherwise it builds for UNIX. The UNIX flavor is NeXT mach. Look for #ifdef MS-DOS to find any portability issues.

Examples:

<u>**DOS**</u>

```
cptosd \usr\*.c A:\usr\pvo\sources
cptosd *.exe D:\usr\ebs\bin
cptosd memo.txt C:memo.txt
```

<u>**UNIX**</u>

```
cptosd /usr/*.c A:\usr\sources
cptosd -b *.dat D:\datafile
cptosd memo.txt C:memo.txt
```

8.3 CPFRSD

Name:

CPFRSD - Copy DOS files to a host directory

Summary:

cpfrsd [-b] [-d] file destpath
-b = binary, don’t convert \n\r to \n
-d = the destination path is a directory, not
a full file spec. In this case a path specification will be created. This is not needed if file contains wildcard characters.

Description:

Copies the file or wildcard expression to the destination path or filename. If the -b flag is asserted files are copied verbatim, otherwise MS-DOS style \n\r combinations are converted to \n.
If MS-DOS is defined during compilation binary copy mode is always used.
During compilation the SDDK-05 tests for the MS-DOS predefined macro. If there, it builds for DOS. Otherwise we build for UNIX. The UNIX flavor is NeXT mach. Look for #ifdef MS-DOS to find any portability issues.

Examples:

<u>DOS</u>

```
cpfrsd A:\source\*.c C:\source
cpfrsd  A:\source\main.c  C:\source\main.c
cpfrsd -d A:\source\main.c C:\source
cpfrsd A:\source\ma?n.c C:\source
```

<u>UNIX</u>

```
cpfrsd “A:\source\*.c” C:/source
cpfrsd  A:\source\main.c  C:/source/main.c
cpfrsd -d A:\source\main.c C:/source
cpfrsd  “A:\source\ma?n.c” C:/source
```

Known Bugs:

The program should take multiple input file specifications, which it does not. The -d flag does not work. Test on the host to see if the destination is a directory instead of relying on the -d flag.**<u>Name.</u>**

8.4 SDLS

Name:

SDLS - Display directory information on the device

Summary:

sdls [path]
Default path is A:*.*

Description:

This program is similar to the DOS **DIR** command and the UNIX **ls** commands. It is simpler in that it does not provide multiple sort and display options. **SDLS** calculates the free space on the device and displays it after the display is complete.
This is a simple portable program. A lot of porters start with this program to test their port. The feedback is very visual and it does not write to the disk.

Note: This program does not always follow the exact conventions that DOS and UNIX follow. It can sometimes be confusing. If the user is testing an initial port, he/she should first issue the **SDLS** command on a known file. Leave wildcard checks until later. Also note that UNIX users should protect wildcards from shell expansion.

Examples:

```
SDLS C:*.BAT
AUTOEXEC.BAT   452     12-22-92  17:06
    1    File(s)  2611200  Bytes free
SDLS C:\MOUSE\*.*
.    .                                   0 <DIR>      06-30-92      13:21
..   .                                   0 <DIR>      06-30-92      13:21
MOUSE          .COM          34295       07-26-91     08:10
MOUSE          .SYS          34499       07-26-91     08:10
MWINST         .EXE          60856       07-26-91     08:10
MWINST         .SCR          9731        07-26-91     08:10
MWINST         .CFG          1082        07-26-91     08:11
MTUTOR         .EXE          39736       07-26-91     08:11
MTUTOR         .SCR          18930       07-26-91     08:11
COMCHECK       .EXE          16300       07-26-91     08:11
READ           .ME           626     07-26-91      08:11
INSTALL        .BAT          1400        07-26-91     08:11
                    12    File(s)   2611200  Bytes free
```

8.5 SDMKD

Name:
SDMKD - Create a directory on the device

Summary:
sdmkd <path>

Description:
This program is similar to the DOS and UNIX **MKDIR** commands. It creates a directory.

Examples:
```
sdmkd a:\usr
sdmkd a:\usr\devt
sdmkd a:\usr\devt\source
```

8.6 SDRM

Name:

SDRM - Delete file(s) from a DOS directory

Summary:

sdrm <path>

Description:

This program is similar to the DOS **DELETE** and UNIX **RM** commands. It removes file(s) from a directory.

Examples:

```
sdrm a:\usr\oldfile.c
sdrm a:\usr\temp*.c
sdrm a:\usr\*.tmp
```

8.7 SDRMD

Name:

SDRMD - Remove a DOS sub-directory

Summary:

sdrmd <path>

Description:

This program is similar to the DOS and UNIX **RMDIR** commands. It removes a subdirectory if the subdirectory is empty.

Examples:

```
sdrmd a:\usr\subdir
sdrmd a:\usr\subdir\subdir1
sdrmd a:\subdir
```

8.8 SDCAT

Name:

SDCAT - Displays a file's contents

Summary:

sdcat <path>

Description:

This program is similar to the UNIX **CAT** command. It displays the contents of the specified file.

Example:

sdcat a:\usr\datafile

8.9 REGRESS

Name:

REGRESS - Stress Test SDDK-05

Summary:

regress

Description:

This program performs two functions. It calls virtually all of the API routines plus it stress tests the system for driver bugs and memory leaks. It works by repeatedly opening a disk and then entering an inner loop which creates a directory and then creates N sub-directories below that. Finally, the inner loop creates NUSERFILES files, writes to them, reads from them, seeks, truncates, closes, renames and deletes them. Along the way it checks set current working directory and get working directory. Finally the inner loop deletes all of the sub-directories it created and compares the current disk free space to the free space before it started. These should be the same. After the inner loop completes, the outer loop closes the drive and then reopens it to continue the test.

There are a few functions that do not get tested, they are:

pc_gfirst
pc_gnext
pc_gdone

Not all modes of **po_open** and **po_lseek** are tested and the user's port is not tested in multitasking mode. The user may modify this program and run it in multiple threads if needed.

The following parameters may be changed:

USEPRINTF	-	Set this to zero to run completely quietly. If this is done the user should set a break point in **regress_error** to catch errors.
test_drive[]	-	The drive where the test will occur.
test_dir[]	-	The directory where the test will occur
INNERLOOP	-	The Number of times we run the inner loop
OUTERLOOP	-	The Number of times we run the outer loop
SUBDIRDEPTH	-	The depth of the tested subdirectories.
NSUBDIRS	-	The number of subdirectories below **test_dir**. Must be less then 26. Each of these directories will have SUBDIRDEPTH subdirectories below it.

8.10 TSTSH

Name:

TSTSH - Interactive Test Shell

Summary:

tstsh

Description:

This program provides an interactive shell for accessing SDDK-05 functions. It provides a handy and relaxed method for testing the user's port. The test shell works only with the device (drive letters for other system devices are not recognized). All commands are summarized below:

Command Descriptions:

CAT - Display contents of a file
This command displays the contents of a file to the console.

Example:

Cat A:\use\ASCII\budget.txt

CD - Set or display working directory
This command sets the default directory if an argument is supplied, otherwise it displays the current working directory.

Example:

CD	- Display working directory
CD \usr\data	- Change working directory

CLOSE - Close a random access file
This command closes a random access file that was opened with **RNDOP**. See **RNDOP** for a discussion of random access files.

Example:

close 1 - Close random access file 1

Note: The disk must already be open. See DSKOPEN.

COPY - Copy a file to another.
This command copies the source file to the destination.

Example:

COPY A:FILE.DAT A:FILE2.DAT

DELETE - Delete a file.
This command will delete a file.

Example:

DELETE A:\FILE001.CHK

DIFF - Compare two files.
This command compares two files and prints whether or not they are the same.

Example:

DIFF A:FILE1.DAT   A:FILE2.DAT

DIR - Print a directory listing.

Example:
Dir *.c

DSKSEL - Set default drive
This command set the default drive so that subsequent commands that do not explicitly contain a drive letter will refer to this drive.

Example:
DSKSEL D:

FILLFILE - Create a file and fill it with a pattern.
This command creates a file and repeatedly fills it with a pattern. It is especially useful when the user wishes to create some test files for copying, deleting, catting, etc. (i.e., when the user first brings up a ramdisk version).

Example:
Create and fill the file file.dat with the pattern "THIS IS A TEST" 1000 times.
FILLFILE FILE.DAT "THIS IS A TEST" 1000

FORMAT - Format a device.
This command formats the device.

Example:
FORMAT A:

GETATTR - Print a file's attributes.
This command calls the pc_get_attributes library routine and prints the results.

Example:
GETATTR FILE.DAT

HELP - Display all commands

Example:
HELP

LSTOPEN - Display all open random access files
This command lists all open random access files along with their file handles. This is especially useful since after the initial OPEN all accesses are done via the handle, and it is easy to forget which handle goes with which file.

Example:
LSTOPEN

MKDIR - Create a directory
This command creates a directory.

Example:
MKDIR \USR\NEWDIR

QUIT - Exit the command shell
This command exits the command shell. The user should first issue a close on all files and open disks.

Example:
```
QUIT
```

READ - Read and display a random access record
This command reads data from the random access file and prints its value to the console. (See **WRITE** for how to write data to the file, **SEEK** for how to seek to a record in the file, LSTOPEN to list all random access files by handle and, RNDOP for how to open a random access file.)

Example:
```
RNDOP \TEST\FILE 100- open(returns handle=0)
SEEK 0 0
WRITE 0 “This is record zero”
SEEK 0 1
WRITE 0 This is record one”
SEEK 0 0
READ 0 - This will print
        “This is record zero”
CLOSE 0
```

RENAME - Rename a file
This command will rename a FILE or directory.

Example:
```
RENAME C:\TES\JOSUF.TXT JOSEPH.TXT
```

RMDIR **-** Remove directory
This command will remove an empty subdirectory.

Example:
```
RMDIR \USR\THEDIR
```

RNDOP **-** Open a random access file
This routine will open or reopen a file for use by random access file I/O test commands READ, WRITE and SEEK. It must be given the file name and the record size for the file. The record size is stored internally and is used to pad write operations to the correct width. (Record size should not exceed 512). Use CLOSE to close a file that was opened with RNDOP and LSTOPEN to display all open files. RNDOP does not return the file handle so use LSTOPEN. Note that the file handles are always returned 0, 1, 2, 3, etc. The user should use this knowledge if he/she wants to use random access files in a script.

Example:
```
RNDOP TESTFIL 200
```

SEEK - Seek to a record in a random access file.
This command seeks to a record number in a random access file. It takes a file handle and a record number as an argument.

Example:

See READ for an example

SETATTR - Change a file's attributes.
This command calls the pc_set_attributes library routine to change a file's attributes

Example:

SETATTR FILE:DAT RDONLY*

The following values are valid for the attribute:

RDONLY
HIDDEN
SYSTEM
ANORMAL

STAT - Stat a file and print results.
This command calls the stat library routine and prints the results.

Example:

STAT A:FILE.DAT

**WRITE - ** Write data to a random access file
This command writes data to the current record of a random access file. The data is filled to the correct width (with spaces) internally. Multi word strings should be quoted.

Example:

See READ.

9.0 Evaluating the Tool Kit in a PC Environment

Since no configuration or porting is necessary to use SDDK-05 in the typical DOS environment, by using a SanDisk-supplied PCI_SD board, the user can readily experiment with the system before starting the port to his/her target hardware.

To run sample utility programs, the user should go to the directory **Sdbuild/tstsampl** where he/she can execute any *.exe.

This reference design was developed using Microsoft C. The user can easily experiment with the APIs, using this compiler.