Developing SDIO Peripherals for Palm Handhelds

CONTRIBUTORS

Written by Greg Wilson
Production by <dot>**PS** document production services
Engineering contributions by William Pfutzenreuter, Geoff Richmond, Gary Stratton

Developing SDIO Peripherals for Palm Handhelds
February 1, 2002
PluggedIn program members (http://www.palm.com/developers/) can obtain the latest version of this document by logging into:
https://pluggedin.palm.com/

Palm, Inc.
5470 Great America Pkwy.
Santa Clara, CA 95052
USA
www.palm.com

Table of Contents

About This Document

This document is intended to assist you in writing Palm OS® applications that interact with SDIO hardware. Because there is a wide range of possible SDIO devices, it focuses solely on those aspects of program design that are specific to the Palm OS, Palm handhelds, and to the SDIO slot driver.

This document also contains a complete API reference for the SDIO slot driver.

Additional Resources

- Documentation

  Palm publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/tech/docs/

- Training

  Palm and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/tech/support/classes/

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions (FAQs), sample code, white papers, and the development documentation at

  http://www.palmos.com/dev/kb/

1

Developing SDIO Applications for Palm Handhelds

Much of an SDIO application is dictated by the hardware with which it interacts. However, because SDIO is a standard, and because these SDIO applications run on the Palm OS®, all such applications have a number of traits in common. This commonality is the subject of this chapter.

This chapter begins by ensuring that you have all of the software, hardware, and documentation that you'll need to create your application. It next talks about the various aspects of the Palm OS that you'll use when writing your application, and then provides some programming guidelines specific to SDIO applications. It ends with a few pointers relative to creating and debugging the SDIO card itself.

Useful Information and Tools

This document is by no means an exhaustive source of information with regard to creating SDIO applications. In addition to it, you'll want to have a copy of the SDIO Specification and an up-to-date copy of the *Palm OS Programmer's API Reference* and *Companion*.

If you are developing SDIO hardware, you will also want to know about Palm's HDK (Hardware Development Kit) and EDK (Expansion Development Kit). The HDK contains mechanical specifications, drawings, and documentation that assist with the design of peripherals. The EDK is a set of parts or items available for purchase at the Palm Expansion Parts Store. Information on all of these items can be found at the PluggedIn Program website at http://www.palmos.com/dev/pluggedin/.

SD, SDIO, and MMC Specifications

The SD Card Association (SDA) publishes the *SDIO Card Specification*, which is based on and refers to the SDA document titled *SD Memory Card Specifications, Part 1, PHYSICAL LAYER SPECIFICATION*. Both of these documents provide essential foundation material for the contents of this document. You should be familiar with the *SDIO Card Specification* and with those parts of the *SD Memory Card Specifications* that document card modes, card initialization, interrupts, registers, and card reading and writing. Depending on the SDIO hardware with which you are working, additional sections of the *SD Memory Card Specifications* document may be of interest.

The SD Card Association's website can be found at http://www.sdcard.org/. You'll need to be a member in order to obtain the specifications from the SD Card Association.

NOTE: Creating Palm OS applications that can use and exchange data from other products via SD Memory cards is outside the scope of this document. However, to make sure that data can be interchanged with present and future SD products, please refer to the appropriate SD Association specification depending on the type of application.

For developers working with MultiMediaCards (MMC), the MultiMediaCard Association's website can be found at http://www.mmca.org/. The MMC specifications are available from the MultiMediaCard Association to MMCA members.

The SDIO slot driver has been written to accommodate the following specifications:

- MMC memory cards, V1.4 to V3.0
- SD memory cards, Part 1, V1.0 (and the supplement to part 1)
- SDIO V1.0

Palm OS SDK

General Palm OS programming concepts are documented in the *Palm OS Programmer's Companion*. Reference documentation for the

APIs made public by the Palm OS can be found in the *Palm OS Programmer's API Reference*. Both of these documents are installed as part of the Palm OS Software Developer's Kit (SDK) and can also be found on Palm's developer website at http://www.palmos.com/dev/tech/docs/. Note that Palm's documentation is often updated after the SDK has shipped; always check the website for the latest, most up-to-date documentation. Be sure that you have the latest SDK and documentation; SDIO applications are not supported on versions of the Palm OS prior to 4.0.

Although you'll want to be familiar with a number of different aspects of Palm OS programming, pay particular attention to the portions of the *Companion* and *Reference* that cover the Expansion and VFS Managers; these chapters show you how to read and write expansion media, including SD memory cards.

In addition to the Palm OS SDK, you should also have the header files for the SDIO slot driver and copies of the SDIO sample applications provided by Palm. These are included with the Palm SDIO SDK. The header files included with the SDIO SDK are compatible with the Palm OS SDK and must be copied into a folder in your project's "include" path. Refer to the `ReadMe` file in the SDIO SDK for up-to-date installation instructions to ensure that the SDIO APIs can be used with your project.

Software Architecture of an SDIO Application

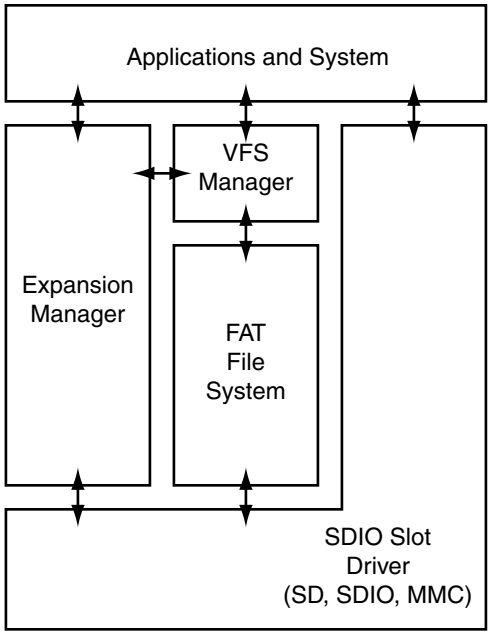
Palm OS applications that interact with SDIO cards make use of the functions provided by the Expansion Manager, the VFS Manager, and the SDIO slot driver. Before you can write such a Palm OS application, you should have an understanding of how your application will interact with these and other features of the Palm OS.

Figure 1.1 presents a simplified view of how the SDIO slot driver relates to your applications, the Expansion Manager, and the VFS Manager. Unlike other Expansion Manager slot drivers, the SDIO slot driver exposes its APIs to applications. Because it also lies beneath the Expansion and VFS managers, you access SDIO hardware through a combination of Expansion Manager, VFS Manager, and SDIO slot driver calls. Note that you use the VFS

Manager with a given SDIO card only if there is an SD or SDIO file system present on that card.

The VFS Manager APIs are used for all file system access on an expansion card. When inserted, SD memory and SDIO CSA memory is mounted as file system memory. Therefore, access to these memory areas is done using the VFS Manager APIs. Details of accessing data on file systems can be found in the standard Palm OS documentation on Expansion Manager and VFS APIs.

Figure 1.1 Relationship between SDIO application, SDIO slot driver, and other key OS components



Expansion Manager

The Expansion Manager is a software layer that manages slot drivers on Palm OS handhelds. The Expansion Manager is not solely responsible for support of expansion cards; rather, it provides an architecture and higher-level set of APIs that, with the help of low-level slot drivers and file system libraries, support various types of media.

The Expansion Manager:

- broadcasts notification of card insertion and removal
- plays sounds to signify card insertion and removal
- mounts and unmounts card-resident volumes

NOTE: Some of the functions provided by the Expansion Manager are designed to be used by slot drivers and file systems and are not generally used by third-party applications.

For a detailed explanation of the functions that make up the Expansion Manager, see the "Expansion Manager" chapter in the *Palm OS Programmer's API Reference*.

VFS Manager

The VFS (Virtual File System) Manager provides a unified API that gives applications access to many different file systems on many different media types, including SD media. The VFS Manager is used for all file system access on an expansion card. In the case of an SDIO card, the VFS Manager is typically used to access any function CSA memory. The data stored in CSA memory is structured as a FAT12/16 file system and is therefore ideally suited for access by the VFS Manager.

Combo cards may contain SD memory that is also accessed through the VFS Manager APIs.

For a detailed explanation of the functions that make up the VFS Manager, see the "Virtual File System Manager" chapter in the *Palm OS Programmer's API Reference*.

SDIO Slot Driver

To simplify the interaction with the SDIO hardware, Palm has created an SDIO slot driver. It replaces the Palm OS 4.0 SD/MMC slot driver, which isn't SDIO-aware, and consists of data structures and functions that allow you to easily manage power, interrupts, and data on the SDIO card.

The SDIO slot driver controls all media supported by an SD expansion slot, including SD media, MMC media, and SDIO media.

An examination of the functions provided by the SDIO slot driver shows that it implements most of the software functionality outlined in the SDIO Card Specification. It does not, however, support the following:

- SDIO Suspend/Resume Operation
- SDIO Read Wait Operation
- SDIO RW Extended Block Operation in "forever" mode

Notification Manager

The Palm OS Notification Manager allows applications to receive notification when certain system-level or application-level events occur. Although the Notification Manager has many uses, developers of SDIO applications should particularly take note of the fact that you use it to detect card removal.

Guidelines for SDIO Applications

All SDIO applications need to be aware of the power needs of the SDIO card. As well, they need to be able to handle interrupts generated by the card, and must be aware of when an SDIO card is inserted or removed from the handheld's SD slot. The following sections discuss these and other SDIO-application-specific topics.

Power Management

When the handheld awakes from sleep mode, it doesn't turn the card on. Only when there is a request to access the card does it turn the card on.

Turning on Card Functions

You can either turn on a given SDIO card function explicitly with SDIOSetPower, or you can turn it on implicitly by simply accessing the function. However you turn functions on, be aware that you as an application developer are responsible for managing card power.

You must ensure that the total of all function hardware that is active does not draw in excess of the SDIO-specified maximum of 200ma.

Perform the following steps to explicitly turn on an SDIO card function:

1. Disable SDIO interrupts with `SDIODisableHandheldInterrupt`—even if your application doesn't use interrupts.
2. Verify that there is sufficient current available to power the card function. To aid in the power management process, the SDIO slot driver provides three functions: `SDIOGetCurrentLimit`, `SDIOSetCurrentLimit`, and `SDIORemainingCurrentLimit`.

NOTE: These three functions do not detect or limit current draw, check the battery level, or reflect how much energy the battery has left.

   The current limit for each function can be obtained by calling `SDIOGetCurrentLimit` or changed by calling `SDIOSetCurrentLimit`. Prior to enabling power to a given function, call `SDIOGetCurrentLimit` to determine how much power it will draw, and compare it to the value returned from `SDIORemainingCurrentLimit`, which indicates how much current can be spared.
3. Turn the function on using `SDIOSetPower`.
4. Reenable interrupts by calling `SDIOEnableHandheldInterrupt`.

After turning off an SDIO card function (with `SDIOSetPower`), be sure to call `SDIOSetCurrentLimit` and set its current limit to zero.

When a card is removed, all of the in-memory current limits are automatically set to zero.

Auto Power Off

The `SDIOSetAutoPowerOff` function allows you to specify an amount of time after which the power and data signals to a given function on an SDIO card should be turned off. You specify this time

interval in system ticks; there are 100 system ticks per second. To disable the auto-power-off feature, simply call this function and supply a tick count of zero.

To obtain the current auto-power-off settings for a given SDIO card function, use `SDIOGetAutoPowerOff`.

Callbacks

The SDIO slot driver allows your application to register callback functions that will be invoked whenever the corresponding event occurs on the SDIO card. Several of these callbacks relate to power management.

Whenever the handheld is about to be put to sleep, the callback function corresponding to `sdioCallbackSelectSleep` is called. Just after the handheld wakes, the function corresponding to `sdioCallbackSelectAwake` is called. These callback functions can be called from either an interrupt routine or a non-interrupt routine; as a result interrupts may be disabled or enabled. In either case, they should always be as fast as possible.

Whenever SDIO card power is turned on or is about to be turned off, the callback function corresponding to `sdioCallbackSelectPowerOn` or `sdioCallbackSelectPowerOff`, respectively, is called. While processing these functions, never call `SDIOSetPower` in order to turn an SDIO card's power on or off. These functions can be called from within an interrupt handler, so they should be as fast as possible.

For more information on callback functions, see "Application-Defined Functions" on page 82.

Interrupt Handling

An SDIO card is capable of interrupting the host device into which it is inserted—in this case, the Palm handheld. The SDIO slot driver allows you to register a callback function that is called whenever the card interrupts the handheld.

Register for the interrupt callback by calling `SDIOSetCallback` and specifying that you are registering for `sdioCallbackSelectInterruptSdCard`. In your callback

function, be sure to reset the interrupt source to prevent the interrupt callback from being called again inadvertently. See "Application-Defined Functions" on page 82 for the parameters that are passed to your callback function when it is called.

Whether or not you have registered an interrupt callback function, you can enable or disable the SDIO interrupt on the handheld by calling `SDIOEnableHandheldInterrupt` or `SDIODisableHandheldInterrupt`. Note that these functions only affect interrupts on the handheld; they do not turn on or off interrupts on the SDIO card itself.

These functions are implemented as an incrementing counter, making them re-entrant. For instance, for every call to `SDIODisableHandheldInterrupt` there must be an equal number (or more) of calls to `SDIOEnableHandheldInterrupt` in order to re-enable interrupts.

By default, when the card is inserted interrupts on the handheld are enabled, but are disabled internally until an interrupt callback is set with `SDIOSetCallback`. Note that in order to receive the SDIO interrupt, power to the card must be on, even if the handheld is asleep.

Detecting Card Insertion and Removal

Applications that depend on the presence of the SDIO card in the slot should register for a `sysNotifyCardRemovedEvent`, which is broadcast when the user removes the card from the SD slot. The following code excerpt shows how you might do this for an executing SDIO application:

```
#include <PalmTypes.h>
#include <NotifyMgr.h>

typedef struct {
    UInt16 slotLibRefNum; // contains a valid slot driver
                          // library reference number
    UInt16 slotRefNum;    // contains a valid slot number
    // additional app-specific globals here
} MyGlobals;

MyGlobals myGlobals;      // Remember to lock this if you
                          // "App_Stop" and still need this!
```

```
Err MySysNotifyRegister(void){
    LocalID dbID;
    Err err;
    UInt16 cardNo;

    err = SysCurAppDatabase(&cardNo, &dbID);
    if (err == errNone)
        err = SysNotifyRegister(cardNo, dbID,
            sysNotifyCardRemovedEvent,
            &MyNotifyCardRemovedEvent,
            sysNotifyNormalPriority, &myGlobals);
    return(err);
}

Err MyNotifyCardRemovedEvent(SysNotifyParamType
*notifyParamsP){
    MyGlobals *myGlobalsP =
        (MyGlobals *)notifyParamsP->userDataP;
    UInt16 slotRefNum =
        (UInt16)notifyParamsP->notifyDetailsP;

    if (slotRefNum != myGlobalsP->slotRefNum)
        return(errNone);   // wrong slot driver
    // app-specific code here. this slot driver's card has
    // been removed
   return(errNone);
}
```

Be sure to unregister for the `sysNotifyCardRemovedEvent` notification and any SDIO callbacks when your application terminates.

For more information on registering and unregistering for notifications, see the Notification Manager chapter in the *Palm OS SDK Reference*. The "Expansion" chapter of the *Palm OS Programmer's Companion*, vol. I discusses, among other things, the various notifications that are issued when a card is inserted or removed, or when a volume is mounted or un-mounted.

Auto Run

When a card is inserted into the SD slot, after it has been initialized any file system memory present on the card is mounted by the Expansion Manager. This includes all SD memory, in the case of a

standard SD card or SDIO combo card, and all SDIO Function CSA memory for functions 0-7.

After mounting of the file systems, the SDIO slot driver broadcasts a series of Auto Run (`sysNotifyDriverSearch`) notifications. These notifications are sent in an attempt to locate function- or card-specific drivers, and allow those drivers that are already on the handheld to launch themselves.

The typical sequence of events after a card is inserted is as follows:

1. Power is applied to the card.
2. The card is initialized according to the SDIO, SD, or MMC specification, as appropriate.
3. Information about the card (tuples, clock speed, CSD, CID, etc.) is read.
4. Any recognized file systems are mounted.
5. `sysAppLaunchCmdCardLaunch` is sent to `start.prc` on each mounted file system.
6. The Auto Run notifications (`sysNotifyDriverSearch`) are sent.
7. `sysAppLaunchCmdNormalLaunch` is sent to `start.prc` on each mounted file system.

For SDIO cards, one Auto Run notification is broadcast for the SD memory portion of a combo card, and an additional notification is broadcast for each card function (up to 7). For SD memory and MMC memory cards, only one such notification is sent. The notifications are sent starting with SD memory, followed by function 7 (if there is one) and proceeding to function 1 as appropriate.

The `notifyDetailsP` field of the `SysNotifyParamType` structure that accompanies the Auto Run notification points to an `AutoRunInfoType` structure. Each driver that has registered for `sysNotifyDriverSearch` should examine the contents of the `AutoRunInfoType` structure to determine if it is the driver that should control the inserted card. If so, the driver should then check the `SysNotifyParamType` structure's `handled` field. If `handled` is set to `true`, another driver has received the broadcast and will

control the card. If `handled` is set to `false`, the driver should set it to `true` to indicate that it will control the device.

To see if an SDIO card is inserted and verify that it is the correct card, use the following sample code.

Listing 1.1 Checking for the correct SDIO card

```
/*************************************************************************
 *
 * FUNCTION:    CheckMyCardInfo
 *
 * DESCRIPTION: This routine is used to check if the card inserted is
 *              my card. This checks an "autorun" parameter block to see
 *              if this card is the correct card.
 *
 *              It can be used for the autorun event AND for detecting the
 *              identity of an already inserted card.
 *
 * PARAMETERS:  AutoRunInfoType *autoRunPtr
 *
 * RETURNED:    true  - success, correct card
 *              false - not my card.
 *
 *************************************************************************/
static Boolean CheckMyCardInfo( AutoRunInfoType* autoRunPtr ){
    //The SDIO cards Manufacturer & ID numbers
    #define MySdioCardOemManufacturer 0x00005672L    //Manufacturer ID
    #define MySdioCardOemID 0x00004673L              //OEM ID
    #define MySdioCardOemFunctionNum 1               //We only check function 1

    Boolean result = false;

    if ( autoRunPtr->media != autoRunMediaSDIO )
        goto Skip;

    // Check the AutoRun parameters to see if it is our card
    // First, check the manufacturer id
    if ( autoRunPtr->oemManufacturer != MySdioCardOemManufacturer )
        goto Skip;
    // Check the OEM id
    if ( autoRunPtr->oemID != MySdioCardOemID )
        goto Skip;
    // This card is an SDIO custom device
    if ( autoRunPtr->oemFunctionStandard != autoRunFunctionStandardSDIOCustom )
        goto Skip;
    //We are only checking function 1
```

```
    if ( autoRunPtr->oemFunctionNum != MySdioCardOemFunctionNum)
        goto Skip;
    if ( autoRunPtr->sourceStruct != autoRunSourceSlotDriverType )
        goto Skip;

    //
    // This is the correct SDIO card
    //
    result = true;

Skip:
    return( result );
}


/*************************************************************************
 *
 * FUNCTION:    CheckCardInserted
 *
 * DESCRIPTION: This routine is used to check if a card is inserted and
 *              that it is the correct card.
 *
 * PARAMETERS:  void
 *
 * RETURNED:    errNone - success, correct card
 *              expErrCardNotPresent - no expansion cards inserted.
 *              expErrEnumerationEmpty - no matching card found
 *
 *************************************************************************/
static Err CheckCardInserted( void ){
    Err err = errNone;
    UInt32 slotIterator;
    UInt16 slotRefNum;
    UInt16 slotLibRefNum;
    UInt32 mediaType;
    UInt16 count = 0;
    SDIOAutoRunInfoType autoRunInfo;

    // Check each slot
    slotIterator = expIteratorStart;
    while( slotIterator != expIteratorStop ){
        err =  ExpSlotEnumerate( &slotRefNum, &slotIterator );
        if ( err ){
            break;
        }

        // Find the slot driver for this slot
        err = ExpSlotLibFind(slotRefNum,  &slotLibRefNum);
```

```
        if ( !err ){
            err = SlotMediaType( slotLibRefNum, slotRefNum, &mediaType );
            if ( !err ){
                // Is this Slot Driver an SD slot driver?
                if ( mediaType == expMediaType_SecureDigital ){
                    // Is the card inserted?
                    err = ExpCardPresent( slotRefNum );
                    if ( !err ){
                        // Count the number of cards we have found
                        count++;

                        // Get the AutoRun Information from function 1 of
                        // the card. The autorun information contains fields
                        // that identify the card.
                        autoRunInfo.sdioSlotNum = sdioSlotFunc1;
                        err = SDIOGetAutoRun( slotLibRefNum, &autoRunInfo );
                        if ( err == errNone ){
                            if ( CheckMyCardInfo( &autoRunInfo.autoRun ) ){
                                // We found it!
                                goto Exit;
                            }
                        }
                    }
                }
            }
        }
    }
    if ( count == 0 )
        err = expErrCardNotPresent;
    else
        err = expErrEnumerationEmpty;
Exit:
    return( err );
}
```

Developing the SDIO Peripheral

An SDIO application is only as good as the hardware with which it interacts. The following sections provide some tips for the creation and debugging of an SDIO peripheral to be used with a Palm handheld.

EDK

Palm has made available the SDIO Developer Card #1, a sample SDIO design demonstrating an SDIO interface to a microcontroller. It is an Expansion Developer Kit (EDK) that allows hardware developers to experiment with SDIO hardware and software for prototyping and evaluation purposes. The card includes a flash-programmable PIC microcontroller and a CPLD for maximum flexibility in prototyping.

Palm's EDK is available for purchase at the Palm Expansion Parts Store. For more information, see Palm's PluggedIn Program website at http://www.palmos.com/dev/pluggedin/.

Specifications

When developing an SDIO peripheral, it is extremely important that you following the specifications identified in "SD, SDIO, and MMC Specifications" on page 4. Be sure to pay close attention to the power restrictions, as the Palm handheld isn't able to deliver more power to an SDIO peripheral than the specification maximum.

As discussed in "SDIO Slot Driver" on page 17, all SDIO cards must support SPI mode. For future compatibility, your SDIO card should also support SD 1-bit mode, as required in the SDIO specification. Future Palm handhelds will likely support the SD 1-bit or SD 4-bit modes.

SPI Mode

All Palm handhelds running Palm OS 4.0 are SPI mode hosts. Accordingly, SDIO cards must support SPI mode in order to be compatible with these handhelds. In addition, the SDIO specification indicates that all SDIO cards must support SPI mode and SD 1-bit mode to be compliant. It is important to be compliant with this specification, since future Palm handhelds will likely support the SD 1-bit or SD 4-bit modes.

SDIO Slot Driver

A Palm handheld running Palm OS 4.0 supports SD/MMC expansion cards. If the SDIO slot driver is installed, it will also

support SDIO expansion cards. In both cases, only one file system can be mounted for a given expansion card. Future versions of the Palm OS will likely lift this restriction, allowing up to seven file systems to be mounted for an SDIO expansion card.

In order to support SDIO peripherals, handhelds running Palm OS 4.0 must either be flash-upgraded to a version of the OS that supports SDIO, or must have the SDIO slot driver separately installed in RAM. The SDIO slot driver can be downloaded from the Palm website and installed as a PRC file in RAM on Palm OS 4.0 devices. After a soft reset, the slot driver in RAM is recognized and takes precedence over the SD/MMC slot driver in ROM.

You can verify whether a given slot driver is "SDIO-aware" by calling `SDIOAPIVersion`. This function returns `expErrUnimplemented` if the specified driver doesn't support SDIO, or `errNone` if it does. If the driver does support SDIO this function also returns the slot driver version number through the `versionP` parameter.

To remove the SDIO slot driver from RAM, you must perform a hard reset of the handheld. You cannot delete the SDIO slot driver using the Application Launcher's "Delete" function. Note that to avoid having the SDIO slot driver reinstalled on the handheld during the next HotSync operation, you must remove the slot driver PRC from the Backup directory of your desktop computer.

SDIO Card Initialization and Identification on Palm OS

The process of identifying and initializing an SDIO card is specified in the *SDIO Card Specification*. One of the first steps in developing an SDIO card is to have the card identify itself as an SDIO card to the host. While performing this task you'll likely want to make use of the command tracing functionality of the debug SDIO slot driver. By enabling tracing on the debug slot driver, you can follow the power-up/power-down sequence of the card, plus all commands sent to the card during the initialization and identification phase. See "Debugging Your SDIO Card" on page 20 for instructions on how to enable tracing.

Identification

Identification of a card is done only once, at the time the card is inserted in the handheld's SD slot. Information obtained from the card during the identification phase is retained in the handheld's memory until the card is removed. Among other things, this information includes:

- the type of card in the slot
- what the card contains
- the card's limits
- data read from tuples

By default SDIO cards power-off automatically after a certain amount of inactivity. This behavior can be modified with the `SDIOSetAutoPowerOff` function.

Initialization

A card is initialized every time it is turned on. The SDIO slot driver follows the appropriate initialization flowchart—SD mode or SPI mode—from the "SDIO Card Initialization" section of the *SDIO Card Specification* to initialize the card.

During the initialization phase, the handheld operates within the range of SD or SPI clock frequencies specified in the *SD Memory Card Specifications* (from zero to 400kHz). The actual clock frequency used depends upon the model of the Palm handheld.

The TPLFID_FUNCTION tuple, located immediately after the CISTPL_FUNCID tuple in the CIS for function 0, contains the TPLFE_MAX_TRAN_SPEED byte. This byte indicates "the maximum transfer rate per one data line during data transfer"; essentially, the maximum clock frequency that the card can support. As soon as this tuple is read, the SDIO slot driver increases the clock speed to the highest possible frequency that doesn't exceed the maximum specified in TPLFE_MAX_TRAN_SPEED.

CSA

In order for a an SDIO card's CSA (Code Storage Area) to be readable by the Palm OS, the CSA should be in FAT12/FAT16 format, and any drivers, data, or applications that the peripheral

would like to be automatically detected by the Palm handheld should reside in the `/Palm` and `/Palm/Launcher` directories. Once the CSA area is mounted, applications may access any data within the CSA irrespective of the directory in which that data resides.

Debugging Your SDIO Card

The SDIO slot driver includes the `SDIODebugOptions` function which, on a debug version of the SDIO slot driver, enables and disables command tracing. Command tracing is very useful for debugging the identification, initialization, and communications functions of an SDIO card. When tracing is enabled, all trace information is dumped in ASCII format to the handheld's USB or serial port.

In order to perform command tracing, a debug version of the SDIO slot driver must be resident on your handheld. A debug version of the SDIO slot driver is available through the Plugged In program. Install it as follows:

1. **If necessary, uninstall the existing RAM-resident "SlotDriver: SDIO-sdsd" slot driver from your Palm handheld by performing a hard reset. This step is only required if the slot driver is resident in RAM.**
2. **Install the debug version of the slot driver using the standard Palm Desktop Install Tool.**
3. **Perform a soft reset of the device to activate the newly-installed slot driver.**

   You can now enable command tracing by calling `SDIODebugOptions` directly from your application, or by using a helper application such as the `SDDbgTrace` sample application included with the SDIO SDK. To use the `SDDbgTrace` application to enable command tracing, perform the following steps:

1. **Install the `SDDbgTrace` PRC file to the handheld using the Palm Desktop Install Tool.**
2. **Start `SDDbgTrace` on the handheld.**
3. **Select the desired tracing option from the Trace Option pop-up trigger.**

NOTE: If the Trace Option shows "Not Supported", the debug version of the SDIO slot driver is not installed.

The selected tracing option remains active until you perform a soft reset or until the next time you run the `SDDbgTrace` application. Each time you run the `SDDbgTrace` application, tracing is initially set to "None".

The Palm Debugger is a convenient tool for viewing the trace output. Note that current versions of the Palm Debugger require that you connect to the handheld using the serial port—which means that you must have a serial cradle if you are working with a handheld such as the Palm m500 or m505. The following procedure shows you how to use the Palm Debugger to view trace output:

1. **Ensure that the HotSync Manager is not running on the desktop, and that a HotSync operation is not in progress.**
2. **Start the Palm Debugger, and set it to monitor the COM port on the desktop to which the Palm serial cradle is connected.**
3. **Insert the Palm handheld into the serial cradle.**
4. **Enable command tracing by having your application call the `SDIODebugOptions` function or by using the `SDDbgTrace` application.**

NOTE: If this option is activated before the device is in a serial cradle, all debug messages will be routed to the USB cradle (until a soft reset is generated) by default. However, since you are not connected to a USB cradle, the software will "lock" forever trying to open a non-existent USB port. To recover from this, either press reset or start a USB debugger on your desktop computer and then place the handheld in the USB cradle.

5. **Insert an SD, MMC, or SDIO card into the handheld's SD slot.**

   The specified tracing information is sent to the device serial port and displayed in the Debugger window on the desktop.

The Metrowerks debugger console window can also be used to monitor trace output, but note that the formatting of the output can be affected by display of CR/LF information.

2

SDIO Slot Driver

This chapter provides reference material for the SDIO Slot Driver API:

The header file `SDIO.h` declares the SDIO Slot Driver API. The AutoRun data structures and constants are declared in `AutoRun.h`.

AutoRun Data Structures

AutoRunInfoType

When a card is inserted into the SD slot, after it has been initialized the SDIO slot driver broadcasts a series of `sysNotifyDriverSearch` notifications (one for each function—up to 8—on an SDIO card; only one notification is broadcast for an SD or MMC memory card) in an attempt to locate function- or card-specific drivers. The `notifyDetailsP` field of the `SysNotifyParamType` structure that accompanies the notification points to an `AutoRunInfoType` structure. Each driver that has registered for `sysNotifyDriverSearch` should examine the contents of the `AutoRunInfoType` structure to determine if it is the driver that should control the inserted card. If so, the driver should then check the `SysNotifyParamType` structure's `handled` field. If `handled` is set to `true`, another driver has received the broadcast and will control the card. If `handled` is set to `false`, the driver should set it to `true` to indicate that it will control the device.

The `AutoRunInfoType` structure can also be obtained by calling `SDIOGetAutoRun`.

The `AutoRunInfoType` structure is declared as follows:

```
typedef struct {
    AutoRunMediaType media;
    AutoRunOemManufacturerType oemManufacturer;
    AutoRunOemProductIDType oemID;
    AutoRunFunctionNumType oemFunctionNum;
    AutoRunFunctionStandardType
oemFunctionStandard;
    AutoRunSourceType sourceStruct;
    union {
        AutoRunSlotDriverType slotDriver;
    } source;
} AutoRunInfoType

typedef AutoRunInfoType *AutoRunInfoP
```

Field Descriptions

`media`	Identifies the type of card in the SD slot. The contents of the `oem...` fields in the `AutoRunInfoType` structure depend on the value of this field and are obtained from the card. See "Media Types", below, for the defined values for this field and the corresponding values for the remaining `AutoRunInfoType` fields.
`oemManufacturer`	Device manufacturer number.
`oemID`	Device manufacturer's product number.
`oemFunctionNum`	Function number, for multi-function cards. Not used for single-function cards.

`oemFunctionStandard`	For multi-function cards, I/O device interface code for the function indicated by `oemFunctionNum`. Not used for single-function cards.
`sourceStruct`	Specifies which member of the `source` union to use, if any. This field is usually set to `autoRunSourceSlotDriverType`.
`source`	The members of this union provide additional information about the slot driver; which member to choose is determined by the value of the `sourceStruct` field. Currently this union has only one member: a structure that identifies the slot driver. See `AutoRunSlotDriverType` for a description of this structure.

AutoRunMediaType

```
typedef UInt32 AutoRunMediaType
```

SD card type.

AutoRunOemManufacturerType

```
typedef UInt32 AutoRunOemManufacturerType
```

Device manufacturer number.

AutoRunOemProductIDType

```
typedef UInt32 AutoRunOemProductIDType
```

Device manufacturer's product number.

AutoRunFunctionNumType

```
typedef UInt16 AutoRunFunctionNumType
```

Function number from a multi-function card (ranges in value from 1-7).

AutoRunFunctionStandardType

```
typedef UInt16 AutoRunFunctionStandardType
```

I/O device interface code.

AutoRunSourceType

```
typedef UInt16 AutoRunSourceType
```

Specifies which member of the `source` union to use, if any. The following values have been defined for this type:

Constant	Value	Description
`autoRunSourceNone`	`0`	`source` is not used.
`autoRunSourceSlotDriverType`	`1`	`source` is `AutoRunSlotDriverType`.

AutoRunSlotDriverType

Identifies the slot driver that issued the `sysNotifyDriverSearch` notification. This structure is a member of the `AutoRunInfoType` structure's `source` union.

```
typedef struct AutoRunSlotDriverType {
    UInt16 volRefNum;
    UInt16 slotLibRefNum;
    UInt16 slotRefNum;
} AutoRunSlotDriverType
```

Field Descriptions

`volRefNum` — The volume reference number for the mounted file system, if there is one, or `vfsInvalidVolRef` if there is no mounted file system.

`slotLibRefNum`	The slot library reference number for the slot driver that issued the `sysNotifyDriverSearch` notification.
`slotRefNum`	The slot reference number for the slot driver that issued the `sysNotifyDriverSearch` notification, or `expInvalidSlotRefNum` if there is no such slot.

AutoRun Constants

Media Types

The defined values for the `AutoRunInfoType` structure's `media` field and the corresponding values of the `oem...` `AutoRunInfoType` fields are listed in the following sections.

autoRunMediaMMCmem

`autoRunMediaMMCmem` is used for MMC memory cards. This constant is defined as follows:

```
#define autoRunMediaMMCmem
((AutoRunMediaType)'mcmm')
```

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaMMCmem`, the `oem...` fields are defined as shown here:

AutoRunInfoType Field	Value
`oemManufacturer`	MMC's CID register, MID (8-bit unsigned Manufacturer field)
`oemID`	MMC's CID Register, OID (16 bit unsigned OEM/Application ID)

AutoRunInfoType Field	Value
`oemFunctionNum`	Not used.
`oemFunctionStandard`	Not used.

autoRunMediaMMCrom

`autoRunMediaMMCrom` is used for MMC ROM cards. This constant is defined as follows:

```
#define autoRunMediaMMCrom
((AutoRunMediaType)'mcrm')
```

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaMMCrom`, the `oem...` fields are defined as shown here:

AutoRunInfoType Field	Value
`oemManufacturer`	MMC's CID register, MID (8-bit unsigned Manufacturer field)
`oemID`	MMC's CID Register, OID (16 bit unsigned OEM/Application ID)
`oemFunctionNum`	Not used.
`oemFunctionStandard`	Not used.

autoRunMediaSDmem

`autoRunMediaSDmem` is used for SD memory cards. This constant is defined as follows:

```
#define autoRunMediaSDmem
((AutoRunMediaType)'sdmm')
```

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaSDmem`, the `oem...` fields are defined as shown here:

AutoRunInfoType Field	Value
`oemManufacturer`	SD's CID register, MID (8-bit unsigned Manufacturer field)
`oemID`	SD's CID Register, OID (16 bit unsigned OEM/Application ID)
`oemFunctionNum`	Not used.
`oemFunctionStandard`	Not used.

autoRunMediaSDrom

`autoRunMediaSDrom` is used for SD ROM cards. This constant is defined as follows:

```
#define autoRunMediaSDrom
((AutoRunMediaType)'sdrm')
```

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaSDrom`, the `oem...` fields are defined as shown here:

AutoRunInfoType Field	Value
`oemManufacturer`	SD's CID register, MID (8-bit unsigned Manufacturer field)
`oemID`	SD's CID Register, OID (16 bit unsigned OEM/Application ID)
`oemFunctionNum`	Not used.
`oemFunctionStandard`	Not used.

autoRunMediaSDIO

`autoRunMediaSDIO` is used for SD I/O cards. This constant is defined as follows:

```
#define autoRunMediaSDIO
((AutoRunMediaType)'sdio')
```

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaSDIO`, the `oem...` fields are defined as shown here:

AutoRunInfoType Field	Value
`oemManufacturer`	TPLMID_MANF field inside the function's CID CISTPL_MANFID tuple (16-bit Manufacturer field)
`oemID`	TPLMID_CARD field inside the function's CID CISTPL_MANFID tuple (16 bit OEM/Application ID)
`oemFunctionNum`	Function number (1-7).
`oemFunctionStandard`	I/O device interface code field in the SD card's FBR. See "I/O Device Interface Codes" on page 31 for a list of constants that can be used with this field.

autoRunMediaPnps

`autoRunMediaPnps` is used for Plug and Play for a serial peripheral. This constant is defined as follows:

```
#define autoRunMediaPnps
((AutoRunMediaType)'pnps')
```

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaPnps`, the `oem...` fields are defined as shown here:

AutoRunInfoType Field	Value
`oemManufacturer`	Vendor ID from the Pnps Configuration Data Structure (16-bit unsigned field)
`oemID`	Device ID from the Pnps Configuration Data Structure (16-bit unsigned field)

AutoRunInfoType Field	Value
`oemFunctionNum`	Not used.
`oemFunctionStandard`	Not used.

I/O Device Interface Codes

When the `AutoRunInfoType` structure's `media` field is set to `autoRunMediaSDIO`, its `oemFunctionStandard` field can assume one of the following values:

Constant	Value	Description
`autoRunFunctionStandardSDIOCustom`	`0`	Driver for custom function.
`autoRunFunctionStandardSDIOUart`	`1`	Driver for SDIO UART.
`autoRunFunctionStandardSDIOBlueToothFat`	`2`	Driver for SDIO Bluetooth Fat.
`autoRunFunctionStandardSDIOBlueToothThin`	`3`	Driver for SDIO BlueTooth Thin.

sysNotifyDriverSearch

Constant	Value	Description
`sysNotifyDriverSearch`	`'arun'`	Sent after a card has been inserted and the card's information has been identified. It allows SDIO drivers already on the handheld to launch themselves. The parameter pointer that accompanies the notification points to an `AutoRunInfoType` structure.

SDIO Slot Driver Data Structures

SDIOAutoPowerOffType

Used with SDIOGetAutoPowerOff and SDIOSetAutoPowerOff to specify auto-power-off parameters for a specific SDIO card function.

```
typedef struct {
    SDIOSlotType sdioSlotNum;
    UInt16 ticksTillOff;
    SDIOCardPowerType sleepPower;
} SDIOAutoPowerOffType
```

Field Descriptions

`sdioSlotNum`	Identifies a specific SDIO card function's slot driver. See SDIOSlotType for a list of values that can be used here.
`ticksTillOff`	The amount of time, in system ticks, before power to the SDIO card function is automatically turned off. A value of zero disables the auto-power-off function.
`sleepPower`	Specifies whether the SDIO card function's power and data signals should be turned on or off, whether the SD Memory card portion of a combo card should be reset, or whether to wait for the SDIO portion of an SD card to be ready. See SDIOCardPowerType for a list of values that can be used here.

SDIOCallbackType

Used in conjunction with the SDIOGetCallback and SDIOSetCallback functions, this structure associates a C function with a particular SDIO function callback type.

```
typedef struct {
  SDIOSlotType sdioSlotNum;
  SDIOCallbackSelectType callbackSelect;
  SDIOCallbackPtr callBackP;
  MemPtr userDataP;
} SDIOCallbackType;
```

Field Descriptions

`sdioSlotNum`	Identifies the SDIO card function who's callback is needed or is to be set. See `SDIOSlotType` for the set of values that can be used with this field.
`callbackSelect`	Identifies the particular callback that is needed or is to be set. See `SDIOCallbackSelectType`, below, for the set of values that can be used with this field.
`callBackP`	Pointer to the callback function. See the `SDIOCallbackPtr` function description for the order and type of the callback function's parameters.
`userDataP`	Pointer to a block of user data that is passed to the callback function when the function is called.

SDIOCallbackSelectType

Values of type `SDIOCallbackSelectType` are used in an `SDIOCallbackType` structure to identify which of an SDIO card function's callbacks is needed or is to be set.

```
typedef UInt16 SDIOCallbackSelectType
```

The following constant values have been defined for this type:

Constant	Value	Description
`SDIOCallbackSelect InterruptSDCard`	0	Callback that occurs when an SDIO card interrupts the handheld. Note that this particular callback is made during the processing of an interrupt.
`SDIOCallbackSelect Sleep`	1	Callback that occurs when the handheld wants to go to sleep.
`SDIOCallbackSelect Awake`	2	Callback that occurs when the handheld wants to wake up.
`SDIOCallbackSelect PowerOn`	3	Callback that occurs when power is applied to the SDIO card.
`SDIOCallbackSelect PowerOff`	4	Callback that occurs when power is removed from the SDIO card.
`SDIOCallbackSelect Reset`	5	Callback that occurs when the SDIO section of the card is reset.
`SDIOCallbackSelect BitMode`	6	Callback that occurs when the bus width of the card is changed.

SDIOCardPowerType

Used with `SDIOPowerType` and `SDIOAutoPowerOffType` to specify whether the SDIO card's power and data signals should be turned on or off, whether the SD Memory section of a combo card should be reset, or whether to wait for the SDIO portion of an SD card to be ready.

```
typedef UInt16 SDIOCardPowerType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioCardPowerOff`	0	Turn off the card, put the data signals in a low power state.
`sdioCardPowerOn`	1	Power on and initialize the card.
`sdioCardResetSDMem`	2	Force the SD Memory section of an SD combo card to be software reset by a CMD0. The function returns after the card is initialized. This value cannot be used with `SDIOSetAutoPowerOff`.
`sdioCardWaitSDIO`	3	Wait for the I/O portion of an SDIO card to be ready (after IO_SEND_OP_COND—CMD5). Use this after resetting one or more functions. This value cannot be used with `SDIOSetAutoPowerOff`.

SDIOCurrentLimitType

Used with `SDIOGetCurrentLimit`, `SDIOSetCurrentLimit`, and `SDIORemainingCurrentLimit` to specify an SDIO card function and the maximum current that can be required by that function.

```
typedef struct {
    SDIOSlotType sdioSlotNum;
    UInt32 uaMaximum;
} SDIOCurrentLimitType
```

Field Descriptions

`sdioSlotNum` — Identifies a specific function slot driver within an SDIO card. See `SDIOSlotType` for a list of values that can be used here.

`uaMaximum` — The specified function's maximum peak current in micro-amps (when used with `SDIOGetCurrentLimit` or `SDIOSetCurrentLimit`), or the remaining maximum current for the entire card in micro-amps (when used with `SDIORemainingCurrentLimit`).

SDIOFuncType

Used with the `SDIORWDirect`, `SDIORWExtendedByte`, and `SDIORWExtendedBlock` functions to specify the number of the SDIO card function area to be read or written, or with the `SDIOTupleWalk` function to specify the number of the function to be searched.

```
typedef UInt16 SDIOFuncType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioFunc0`	0	SDIO function 0 area (CIA—Common I/O Area).
`sdioFunc1`	1	SDIO function 1 area.
`sdioFunc2`	2	SDIO function 2 area.
`sdioFunc3`	3	SDIO function 3 area.
`sdioFunc4`	4	SDIO function 4 area.
`sdioFunc5`	5	SDIO function 5 area.

Constant	Value	Description
sdioFunc6	6	SDIO function 6 area.
sdioFunc7	7	SDIO function 7 area.

SDIOPowerType

Used by SDIOGetPower and SDIOSetPower to get and set an SD card function's power setting.

```
typedef struct {
  SDIOSlotType sdioSlotNum;
  SDIOCardPowerType powerOnCard;
} SDIOPowerType
```

Field Descriptions

sdioSlotNum	Identifies a specific SDIO card function's slot driver. See SDIOSlotType for a list of values that can be used here.
powerOnCard	An SDIOCardPowerType that specifies whether power should be applied to or removed from the SDIO card function, or that indicates whether power is or is not currently being applied to the function.

SDIORWModeType

Specifies the particular operation to be performed when using SDIORWDirect, SDIORWExtendedBlock, and SDIORWExtendedByte.

```
typedef UInt16 SDIORWModeType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioRWModeWrite`	0x0001	Write data from the specified buffer to the card.
`sdioRWModeRead`	0x0002	Read data from the card and place it in the specified buffer.
`sdioRWModeWriteRead`	0x0003	Write data from the specified buffer to the card, then read the data from the card and place it back into the buffer.
`sdioRWModeFixedAddress`	0x0004	Use in combination with `sdioRWModeWrite`, `sdioRWModeRead`, or `sdioRWModeWriteRead` to perform a multi-byte read or write to a single register address. Useful when transferring data using a FIFO inside the I/O device.
`sdioRWModeForceBlockSize`	0x0008	Use in combination with `sdioRWModeWrite`, `sdioRWModeRead`, or `sdioRWModeWriteRead` to cause `SDIORWExtendedBlock` to always set the block size. This is useful if the driver resets an I/O only card or the I/O portion of a combo card, or if it alters the I/O block length in the FBR (Function Basic Registers).

SDIOSDBitModeType

Used in the `SDIOCardInfoType` and `SDIOSDBitModeRequestType` structures to indicate which SDIO bit mode is to be used when interacting with a particular SDIO card function.

```
typedef UInt16 SDIOSDBitModeType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioSD1BitMode`	1	SDIO 1-bit mode (SD or SPI mode)
`sdioSD4BitMode`	4	SDIO 4-bit mode (SD mode only)

SDIOSlotType

Used with a number of types and functions to identify a specific function slot driver within an SDIO card.

```
typedef UInt16 SDIOSlotType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioSlotSDMem`	0	SD Memory card slot (for regular memory cards or SD combo cards)
`sdioSlotFunc1`	1	SDIO function 1 slot for SDIO cards
`sdioSlotFunc2`	2	SDIO function 2 slot for SDIO cards
`sdioSlotFunc3`	3	SDIO function 3 slot for SDIO cards
`sdioSlotFunc4`	4	SDIO function 4 slot for SDIO cards
`sdioSlotFunc5`	5	SDIO function 5 slot for SDIO cards
`sdioSlotFunc6`	6	SDIO function 6 slot for SDIO cards
`sdioSlotFunc7`	7	SDIO function 7 slot for SDIO cards

SDIO Slot Driver Constants

sysFileApiCreatorSDIO

Constant	Value	Description
`sysFileApiCreatorSDIO`	`'sdio'`	Creator code for the SDIO slot driver.

Number of Entries

Constant	Value	Description
`sdioFuncEntries`	8	The number of possible SDIO card functions.
`sdioCallbackSelectEntries`	7	The number of possible callbacks for a given SDIO card function.

SDIO Slot Driver Functions

New SDIOAccessDelay

Purpose Change the SDIO card access timeout for reads and writes using IO_RW_DIRECT and IO_RW_EXTENDED from the 1 second default.

Prototype

```
Err SDIOAccessDelay (UInt16 slotLibRefNum,
SDIOAccessDelayType *delayMSP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`-> delayMSP`	Pointer to the desired timeout, in milliseconds. The minimum timeout that can be set with this function is 1,000 milliseconds, or 1 second. See `SDIOAccessDelayType` in the Comments section, below.

Result

`errNone`	No error.
`expErrCardNotPresent`	There isn't a card in the slot associated with the specified slot driver.
`expErrUnimplemented`	The specified slot driver does not support SDIO.
`sysErrParamErr`	`delayMSP` is `NULL`.

Comments

The default timeout for any SDIO card is 1 second. Use this function if your card needs a longer worst-case delay. Note that this timeout only affects reads and writes using IO_RW_DIRECT and IO_RW_EXTENDED; basic commands such as those that read the CCCRs, FBRs, and tuples **must** execute within 1 second. If a file system is available, it is mounted with this 1 second limitation.

NOTE: The SDIO slot driver initializes and identifies the card and mounts files before a custom slot driver gains access to the card. The SDIO slot driver assumes that the card functions properly with the 1-second specification time limit.

This function requires that an SDIO card be present in the slot.

This function can safely be called from within an interrupt service routine.

SDIOAccessDelayType

The maximum timeout, in milliseconds, for reads and writes using the IO_RW_DIRECT and IO_RW_EXTENDED commands.

```
typedef UInt16 SDIOAccessDelayType;
```

New SDIOAPIVersion

Purpose Determine if the specified slot driver is SDIO aware and, if so, return the slot driver version number.

Prototype

```
Err SDIOAPIVersion (UInt16 slotLibRefNum,
SDIOAPIVersionType *versionP)
```

Parameters

`-> slotLibRefNum` Slot driver library reference number.

`<- versionP` Pointer to the version number of this SDIO-aware slot driver. See the Comments section, below, for a description of `SDIOAPIVersionType`.

Result

`errNone` The specified slot driver is SDIO aware, and the version number was successfully set.

`expErrUnimplemented` (or any other error) The specified slot driver does not support SDIO.

`sysErrParamErr` `versionP` is invalid.

Comments This function can safely be called from within an interrupt service routine. It does not require a SDIO card in the slot to work

`SDIOAPIVersionType` is declared as follows:

```
typedef UInt32 SDIOAPIVersionType
```

New SDIODebugOptions

Purpose Enable or disable the sending of debug messages to the serial or USB port.

Prototype

```
Err SDIODebugOptions (UInt16 slotLibRefNum,
SDIODebugOptionType *debugOptionsP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> debugOptionsP`
Pointer to an `SDIODebugOptionType` structure which specifies which messages should be sent. See the Comments section, below, for a description of the `SDIODebugOptionType` structure.

Result

`errNone` No error.

`expErrUnsupportedOperation`
This is not a debug ROM or debug RAM patch. No debug features are available.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

Comments

If you use this function to enable debugging, be sure that the handheld is in a cradle and a debugger is running on your desktop computer. CodeWarrior can be used, but it adds extra carriage-return/linefeed pairs to the messages.

WARNING! If this option is activated, and the device is not in a cradle, all debug messages will be routed to the USB cradle by default. However, since the device is not in a USB cradle, the software will “lock”, forever trying to open a non-existent USB port. To recover, either press reset or start a USB debugger on your PC or Mac and then connect the handheld to the cradle.

To deactivate debugging, perform a soft reset on the handheld or call `SDIODebugOptions` and specify `sdioDebugOptionTraceNone`.

NOTE: This is not a real time trace: the serial port slows down the card's response. Use a logic analyzer for real time tracing.

This function can safely be called from within an interrupt service routine.

SDIODebugOptionType

This structure is used with `SDIODebugOptions` and identifies which debug messages, if any, are to be sent to the serial or USB port.

```
typedef UInt16 SDIODebugOptionType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioDebugOptionTraceCmds`	0x0001	Sends all commands that are issued to the card.
`sdioDebugOptionTraceRejection`	0x0002	Sends rejection reasons.
`sdioDebugOptionTraceCmdData`	0x0004	Sends the data from commands that have command/response/data, warning. Note that this is a **lot** of data.
`sdioDebugOptionTraceContents`	0x0008	Sends the contents of tuples and/or parts of the CSD (Card Specific Data register) when they are accessed just after card insertion.
`sdioDebugOptionTraceProgress`	0x0010	Sends the progress of the tests that are performed upon card insertion.
`sdioDebugOptionTraceISR`	0x0020	Allows debug messages to be sent from within interrupt handlers. Be sure to keep the stack small to avoid overflows.

Constant	Value	Description
`sdioDebugOptionTraceMost`	`sdioDebugOptionTraceCmds | sdioDebugOptionTraceRejection | sdioDebugOptionTraceContents | sdioDebugOptionTraceProgress | sdioDebugOptionTraceISR`	
`sdioDebugOptionTraceAll`	-1	Enable all options.
`sdioDebugOptionTraceNone`	0	Disable all options.

Compatibility For debug ROMs and debug RAM patches only.

New SDIODisableHandheldInterrupt

Purpose Disables the interrupt on the handheld. This function does not turn off interrupts on the card.

Prototype
```
Err SDIODisableHandheldInterrupt
(UInt16 slotLibRefNum)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`expErrUnsupportedOperation`
`SDIODisableHandheldInterrupt` has been called in excess of 65,535 times.

Comments This function is implemented as an incrementing counter, making it re-entrant. For every call to `SDIODisableHandheldInterrupt` there must be an equal number (or more) of calls to `SDIOEnableHandheldInterrupt` in order to re-enable interrupts.

This function requires that an SDIO card be present in the slot.

This function can safely be called from within an interrupt service routine.

New SDIOEnableHandheldInterrupt

Purpose Enables the interrupt on the handheld. This function does not affect interrupts on the card.

Prototype

```
Err SDIOEnableHandheldInterrupt
(UInt16 slotLibRefNum)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`expErrUnsupportedOperation`
Interrupts are already enabled.

Comments This function is implemented as a decrementing counter, making it re-entrant. For every call to `SDIODisableHandheldInterrupt` there must be an equal number (or more) of calls to `SDIOEnableHandheldInterrupt` in order to re-enable interrupts.

By default, when the card is inserted interrupts on the handheld are enabled by this function, but are disabled internally until an interrupt callback is set with `SDIOSetCallback`. Note that in order to receive the SDIO interrupt, power to the card must be on, even if the handheld is asleep.

This function requires that an SDIO card be present in the slot.

This function can safely be called from within an interrupt service routine.

New SDIOGetAutoPowerOff

Purpose

Get the current auto-power-off settings for the SD slot.

Prototype

```
Err SDIOGetAutoPowerOff (UInt16 slotLibRefNum,
SDIOAutoPowerOffType *autoP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> autoP`
Pointer to an `SDIOAutoPowerOffType` structure which indicates the current auto-power-off settings for the SD slot. Before calling `SDIOGetAutoPowerOff`, set this structure's `sdioSlotNum` field to indicate the current slot driver function number. Upon return, the `ticksTillOff` field indicates the number of system ticks until the slot is turned off. A `ticksTillOff` value of zero indicates that auto-off is disabled.

Result

`errNone`
No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented` (or any other error)
The specified slot driver does not support SDIO.

`sysErrParamErr` `autoP` is invalid.

Comments This function requires that an SDIO card be present in the slot, since these settings are erased when a card is removed. Note that every time a card is inserted into the SD slot, the auto-power-off time is set to 15 seconds.

When the handheld awakes from sleep mode, it doesn't turn the card on. Only when there is a request to access the card does it turn the card on.

This function only works with SDIO cards; it cannot be used when a memory card is in the slot.

This function can safely be called from within an interrupt service routine.

See Also `SDIOGetPower`, `SDIOSetAutoPowerOff`

New SDIOGetAutoRun

Purpose Provide a description of the SD/MMC memory card or SDIO card that is currently inserted.

Prototype
```
Err SDIOGetAutoRun (UInt16 slotLibRefNum,
SDIOAutoRunInfoType *autoRunP)
```

Parameters `-> slotLibRefNum`
Slot driver library reference number.

	`<-> autoRunP`	Pointer to an `SDIOAutoRunInfoType` structure, which describes the SD/MMC memory card or SDIO card that is currently inserted. Before calling `SDIOGetAutoRun` set this structure's `sdioSlotNum` field to the card function you are using (a value of 1-7 indicates one of the SDIO functions, while a value of 0 indicates the SD memory card slot driver). See the Comments section, below, for a description of the `SDIOAutoRunInfoType` structure.
Result	`errNone`	No error.
	`expErrCardNotPresent`	There isn't a card in the slot associated with the specified slot driver.
	`expErrUnimplemented`	The specified slot driver does not support SDIO.
	`expErrUnsupportedOperation`	The SDIO card does not support the specified SDIO card function.
	`sysErrParamErr`	`autoRunP` is invalid, or one of its fields is invalid.

Comments

Information provided by this function is only maintained while a card is in the SD slot, and is erased when the card is removed. Because of this, this function requires that an SDIO card be present in the slot.

This structure is provided to SDIO device drivers as part of the `sysNotifyDriverSearch` notification that is broadcast by the SDIO slot driver in an attempt to locate a device driver for an SDIO card.

This function can safely be called from within an interrupt service routine.

SDIOAutoRunInfoType

This structure is passed to the SDIOGetAutoRun function and is used to specify the function making the request and the SDIO bit mode that is to be set.

```
typedef struct {
    SDIOSlotType sdioSlotNum;
    AutoRunInfoType autoRun;
} SDIOAutoRunInfoType
```

Field Descriptions

`sdioSlotNum`	The ID of the function in the current slot driver about which slot library information is desired. Note that `sdioFunc0` is reserved for the SD Memory card slot driver and function 0. See the description of `SDIOSlotType` for the complete set of values that can be supplied here.
`autoRun`	Contains a description of the SD/MMC memory card or SDIO card that is currently inserted. See the description of the `AutoRunInfoType` structure for details.

New SDIOGetCallback

Purpose Obtain pointers to an SDIO card function's callback routine and associated data.

Prototype

```
Err SDIOGetCallback (UInt16 slotLibRefNum,
SDIOCallbackType *callBackP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> callBackP`	Pointer to an `SDIOCallbackType` structure. Before calling this function, set the `sdioSlotNum` and `callbackSelect` fields. Upon return, the `callBackP` and `userDataP` fields point to the callback function and any associated user data. Either or both of these pointers can be `NULL` to indicate that there is no associated callback function or that there is no user data block.

Result

`errNone`	No error.
`expErrCardNotPresent`	There isn't a card in the slot associated with the specified slot driver.
`expErrUnimplemented`	The specified slot driver does not support SDIO.
`sysErrParamErr`	`callBackP` is invalid, or one of its fields is invalid.

Comments

Each callback selector for each card function slot, as well as the SD Memory card slot, may have been assigned a separate callback function. The following list of callback selectors details those situations in which a particular callback function is called:

- `sdioCallbackSelectInterruptSdCard`: the corresponding callback function is executed whenever the SD card interrupts the handheld. The handheld enables the SDIO interrupt whenever a valid `sdioCallbackSelectInterruptSdCard` callback is set. The callback function needs to reset the interrupt source to prevent the interrupt callback from being called again.
- `sdioCallbackSelectSleep` and `sdioCallbackSelectAwake`: the corresponding callback functions are executed whenever the handheld is about to be put to sleep or just after it wakes. These callback functions are always called with interrupts disabled, and should be as fast as possible.

- `sdioCallbackSelectPowerOn` and `sdioCallbackSelectPowerOff`: the corresponding callback functions are executed when the SDIO card power is turned on or is about to be turned off. Never call `SDIOSetPower` while processing these functions in order to turn an SDIO card function on or off.
- `sdioCallbackSelectReset`: the corresponding callback function is executed whenever `SDIOSetPower` is called with the `powerP` structure's `powerOnCard` field set to `sdioCardWaitSdio`. `sdioCardWaitSdio` is typically used after the SDIO section has been reset by setting the RES (I/O Card Reset) bit in the CCCR (Card Common Control Registers).
- `sdioCallbackSelectBitMode`: the corresponding callback function is executed whenever the bus width is successfully changed with `SDIOSetBitMode`. Note that in version 1.0 of the SDIO slot driver, this callback is never executed because the bus is always one bit wide.

When a situation arises that causes one of the above callback functions to be called, the corresponding callback for the SD Memory card slot is generally the first one called, followed by the corresponding callback functions for SDIO functions 1 through 7. Because the SD Memory card slot and each SDIO card function slot can have a separate callback function for each callback selector, each callback function can limit itself to dealing with a single selector and a single SDIO card function.

Callback function information is automatically erased after a card is inserted or removed (before the card removal event). Because of this, `SDIOGetCallback` can only be used when a card is in the SD slot. To detect card removal, use the notification manager and register for `sysNotifyCardRemovedEvent`.

`SDIOGetCallback` can safely be called from within an interrupt service routine.

New SDIOGetCardInfo

Purpose

Return information about the SDIO card obtained with the SDIO IO_QUERY command (ACMD57).

Prototype

```
Err SDIOGetCardInfo (UInt16 slotLibRefNum,
SDIOCardInfoType *cardInfoP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<- cardInfoP` Pointer to an `SDIOCardInfoType` structure into which the SDIO card information is placed. See the Comments section, below, for a complete description of the `SDIOCardInfoType` structure.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`sysErrParamErr` `cardInfoP` is invalid.

Comments

Information about the SDIO card other than what is returned by this function can be obtained through the use of the normal SDIO read and write calls.

This function can safely be called from within an interrupt service routine.

This function requires that an SDIO card be present in the slot. Note, however, that information is cached in RAM.

SDIOCardInfoType

This structure is used in conjunction with the SDIOGetCardInfo function to return information about the SDIO card.

```
typedef struct {
  UInt16 numberOfFunctions;
  SDIOSDBitModeType bitMode;
  SDIOBitsOfFileSystemType bitsOfFileSystem;
  SDIOBitsOfStatusType bitsOfStatus;
} SDIOCardInfoType
```

Field Descriptions

numberOfFunctions	Number of SDIO functions on the card. This field's values range from 0 (no functions) to 7.
bitMode	The card's current SDIO bit mode. See the description of SDIOSDBitModeType for this field's values.
bitsOfFileSystem	Each bit in this field indicates whether the corresponding function has a standard SDIO file system. Note that just because a function has a file system, it does not mean that the file system is mounted. See the description of SDIOBitsOfFileSystemType for the precise meaning of the bits that make up this field.
bitsOfStatus	Various status bits, as defined under SDIOBitsOfStatusType.

SDIOBitsOfFileSystemType

Returned as part of an SDIOCardInfoType structure, each of the bits that makes up SDIOBitsOfFileSystemType indicates whether the corresponding function has a standard SDIO file system. Note that this file system may or may not be mounted.

```
typedef UInt16 SDIOBitsOfFileSystemType
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioBitsOfFileSystemMemory`	0x0001	This card has a standard SDIO file system in the SD Memory section.
`sdioBitsOfFileSystemFunction1`	0x0002	This card has a standard SDIO file system in function 1.
`sdioBitsOfFileSystemFunction2`	0x0004	This card has a standard SDIO file system in function 2.
`sdioBitsOfFileSystemFunction3`	0x0008	This card has a standard SDIO file system in function 3.
`sdioBitsOfFileSystemFunction4`	0x0010	This card has a standard SDIO file system in function 4.
`sdioBitsOfFileSystemFunction5`	0x0020	This card has a standard SDIO file system in function 5.
`sdioBitsOfFileSystemFunction6`	0x0040	This card has a standard SDIO file system in function 6.
`sdioBitsOfFileSystemFunction7`	0x0080	This card has a standard SDIO file system in function 7.

SDIOBitsOfStatusType

Returned as part of an `SDIOCardInfoType` structure, each of the bits that makes up `SDIOBitsOfStatusType` indicates various status information about the SDIO card.

```
typedef UInt16 SDIOBitsOfStatusType;
```

The following constant values have been defined for this type:

Constant	Value	Description
`sdioBitsOfStatusDriverHandledMemory`	0x0001	This card has an Auto Run function driver in the SD Memory section.
`sdioBitsOfStatusDriverHandledFunc1`	0x0002	This card has an Auto Run function driver in function 1.
`sdioBitsOfStatusDriverHandledFunc2`	0x0004	This card has an Auto Run function driver in function 2.
`sdioBitsOfStatusDriverHandledFunc3`	0x0008	This card has an Auto Run function driver in function 3.
`sdioBitsOfStatusDriverHandledFunc4`	0x0010	This card has an Auto Run function driver in function 4.
`sdioBitsOfStatusDriverHandledFunc5`	0x0020	This card has an Auto Run function driver in function 5.
`sdioBitsOfStatusDriverHandledFunc6`	0x0040	This card has an Auto Run function driver in function 6.
`sdioBitsOfStatusDriverHandledFunc7`	0x0080	This card has an Auto Run function driver in function 7.
`sdioBitsOfStatusWriteProtectTab`	0x0100	The write protect tab on the card indicates that this card is write protected

See Also SDIOSetBitMode, SDIOGetSlotInfo, SDIOAPIVersion

New SDIOGetCurrentLimit

Purpose

Get the maximum peak current allotted to one of the SDIO card's functions.

Prototype

```
Err SDIOGetCurrentLimit (UInt16 slotLibRefNum,
SDIOCurrentLimitType *currentLimitP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> currentLimitP`
Pointer to an SDIOCurrentLimitType structure. Before calling `SDIOGetCurrentLimit`, set this structure's `slotFuncNum` field to a valid slot driver function number. Upon return, the `uaMaximum` field contains the specified function's maximum peak current in micro-amps.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`sysErrParamErr` `currentLimitP` is invalid, or one of `currentLimitP`'s fields is invalid.

Comments

You use `SDIOGetCurrentLimit`, `SDIOSetCurrentLimit`, and `SDIORemainingCurrentLimit` to ensure that the total of all function hardware that is active never exceeds the SDIO specification maximum of 200ma. These three functions do not detect or limit current draw, check the battery level, or reflect how much energy the battery has left; you simply use them to manage the current limit values supplied using `SDIOSetCurrentLimit`. It

is up to the writer of the SDIO slot driver to both supply the proper current limit values and to use `SDIOGetCurrentLimit` and `SDIORemainingCurrentLimit` appropriately so that the total active SDIO card functions do not draw more current than the handheld's power source can provide.

When a card is removed, all allocations of current are set to zero. Because of this, in order to operate properly this function requires an SDIO card in the slot.

This function can safely be called from within an interrupt service routine.

New SDIOGetPower

Purpose Determine whether an SD card function is currently powered on or off.

Prototype

```
Err SDIOGetPower (UInt16 slotLibRefNum,
SDIOPowerType *powerP)
```

Parameters

`-> slotLibRefNum` Slot driver library reference number.

`<-> powerP` Pointer to an `SDIOPowerType` structure. Before calling `SDIOGetPower` set this structure's `sdioSlotNum` field to the SDIO card function, and upon return the value of this structure's `powerOnCard` field indicates whether or not the SD card function is turned on.

Result

`errNone` No error.

`expErrCardNotPresent` There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`sysErrParamErr` `powerP` is invalid.

Comments This function does **not** check the battery level, since turning on the SDIO card might lockout the handheld. It also does **not** check SDIO card function current limits. A card must be present in the SD slot in order to use this function, however.

This function can safely be called from within an interrupt service routine.

See Also `SDIOGetAutoPowerOff`, `SDIOGetCurrentLimit`, `SDIOGetCardInfo`, `SDIOSetPower`

New SDIOGetSlotInfo

Purpose Obtain the slot driver reference number, the slot driver library reference number, and the volume reference number for the associated file system, if any, for one of the SDIO functions or for the SD Memory card slot driver.

Prototype

```
Err SDIOGetSlotInfo (UInt16 slotLibRefNum,
SDIOSlotInfoType *slotInfoP)
```

Parameters

`->` `slotLibRefNum`
Slot driver library reference number.

	`<-> slotInfoP`	Pointer to an `SDIOSlotInfoType` structure. Before calling this function, set the `sdioSlotNum` field to indicate the function for which the slot driver information is needed. Upon return, the `slotLibRefNum`, `slotRefNum`, and `volRefNum` fields are set as described in the description of the `SDIOSlotInfoType` structure in the Comments section, below.
Result	`errNone`	No error.
	`expErrUnimplemented`	The specified slot driver does not support SDIO.
	`sysErrParamErr`	`slotInfoP` is invalid.

Comments This function does not require that an SDIO card be present in the slot.

This function can safely be called from within an interrupt service routine.

SDIOSlotInfoType

This structure is used with `SDIOGetSlotInfo` to obtain information about a specific SDIO function's slot driver.

```
typedef struct {
  SDIOSlotType sdioSlotNum;
  UInt16 volRefNum;
  UInt16 slotLibRefNum;
  UInt16 slotRefNum;
} SDIOSlotInfoType
```

Field Descriptions

`sdioSlotNum`	The ID of the SDIO card function in the current slot driver about which slot library information is desired. See the description of `SDIOSlotType` for the complete set of values that can be supplied here.
`volRefNum`	The volume reference number for the mounted file system, if there is one, or `vfsInvalidVolRef` if there is no mounted file system for the specified SDIO card function.
`slotLibRefNum`	The slot library reference number for the specified SDIO card function.
`slotRefNum`	The slot reference number for the specified SDIO card function, if there is one, or `expInvalidSlotRefNum` if there isn't.

See Also `SDIOGetCardInfo`, `SDIOAPIVersion`

New SDIORemainingCurrentLimit

Purpose Get the remaining current for the entire SDIO card.

Prototype

```
Err SDIORemainingCurrentLimit
(UInt16 slotLibRefNum,
SDIOCurrentLimitType *currentLimitP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<- currentLimitP`
: Pointer to an SDIOCurrentLimitType structure. Upon return, the `uaMaximum` field indicates how much current, in micro-amps, remains unallocated by the SDIO card's functions. Note that the `slotFuncNum` field isn't used when calling this function.

Result

`errNone`
: No error.

`expErrCardNotPresent`
: There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
: The specified slot driver does not support SDIO.

`sysErrParamErr`
: `currentLimitP` is invalid, or one of `currentLimitP`'s fields is invalid.

Comments

You use `SDIOGetCurrentLimit`, `SDIOSetCurrentLimit`, and `SDIORemainingCurrentLimit` to ensure that the total of all function hardware that is active never exceeds the SDIO specification maximum of 200ma. These three functions do not detect or limit current draw, check the battery level, or reflect how much energy the battery has left; you simply use them to manage the current limit values supplied using `SDIOSetCurrentLimit`. It is up to the writer of the SDIO slot driver to both supply the proper current limit values and to use `SDIOGetCurrentLimit` and `SDIORemainingCurrentLimit` appropriately so that the total active SDIO card functions do not draw more current than the handheld's power source can provide.

When a card is removed, all allocations of current are set to zero. Because of this, in order to operate properly this function requires an SDIO card in the slot.

This function can safely be called from within an interrupt service routine.

New SDIORWDirect

Purpose Read or write a single byte to any I/O function, including the common I/O area (CIA), at any address using CMD52 (IO_RW_DIRECT).

Prototype

```
Err SDIORWDirect (UInt16 slotLibRefNum,
SDIORWDirectType *directP)
```

Parameters

`-> slotLibRefNum` Slot driver library reference number.

`<-> directP` Pointer to an `SDIORWDirectType` structure which describes the read or write operation. See the Comments section, below, for a description of the `SDIORWDirectType` structure.

Result

`errNone` No error.

`expErrCardBadSector` The SDIO memory could not be read or written.

`expErrCardNotPresent` There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented` The specified slot driver does not support SDIO.

`expErrUnsupportedOperation` The SDIO card does not support the specified SDIO card function.

`sysErrParamErr` `directP` is invalid, or one of its fields is invalid.

Comments This function is commonly used to initialize registers or monitor status values for I/O functions. This function requires that an SDIO

card be present in the slot. The card will be turned on and accessed. See the SDIO specification for the SDIO registers that can be read or written.

NOTE: The write protect tab on the SD card is ignored by the SDIO slot driver. Issuing a write request with this function always causes the write command to be sent to the card.

This function can safely be called from within an interrupt service routine.

SDIORWDirectType

This structure is used with SDIORWDirect and encapsulates the read or write operation.

```
typedef struct {
    SDIOSlotType requestingFunc;
    SDIORWModeType mode;
    SDIOFuncType funcNum;
    UInt32 byteAddress;
    UInt8 byteData;
} SDIORWDirectType
```

Field Descriptions

`requestingFunc`	The number of the SDIO card function making the read or write request. This is the function that will be turned on.
`mode`	The operation to be performed: write, read, or write followed by read. See `SDIORWModeType` for a list of operations.
`funcNum`	The number of the function within the I/O card to be read or written. Function 0 selects the common I/O area (CIA).
`byteAddress`	The address of the byte inside of the selected SDIO card function's register space that will be read or written. There are 17 bits of address available, so the byte must be located within the first 128K addresses of that function.
`byteData`	For a direct write command, the byte that will be written. For a direct read command, the byte read is stored here.

See Also `SDIORWExtendedBlock`, `SDIORWExtendedByte`

New SDIORWExtendedBlock

Purpose Read or write multiple blocks of a specified size to any I/O function, including the the common I/O area (CIA), at any address using the optional block mode of CMD53 (IO_RW_EXTENDED).

Prototype

```
Err SDIORWExtendedBlock (UInt16 slotLibRefNum,
SDIORWExtendedBlockType *extendedBlockP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> extendedBlockP`
Pointer to an `SDIORWExtendedBlockType` structure which describes the read or write operation. See the Comments section, below, for a description of the `SDIORWExtendedBlockType` structure.

Result

`errNone` No error.

`expErrCardBadSector`
The SDIO memory could not be read or written.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`expErrUnsupportedOperation`
The SDIO card does not support the specified SDIO card function.

`sysErrParamErr` `extendedBlockP` is invalid, or one of its fields is invalid.

Comments

A given SDIO card may or may not support `SDIORWExtendedBlock`; the SDIO specification doesn't require it. Verify that the card supports block operations by checking the SMB (Card Supports MBIO) bit in the CCCR (Card Common Control Registers). This SDIO slot driver does not support the "infinite" mode (which is normally indicated by setting the block count to zero). See the SDIO specification for the SDIO registers that can be read or written.

This function is commonly used to initialize registers or monitor status values for I/O functions. This function requires that an SDIO card be present in the slot. The card will be turned on and accessed.

NOTE: The write protect tab on the SD card is ignored by the SDIO slot driver. Issuing a write request with this function always causes the write command to be sent to the card.

This function can safely be called from within an interrupt service routine.

SDIORWExtendedBlockType

This structure is used with SDIORWExtendedBlock and encapsulates the read or write operation.

```
typedef struct {
    SDIOSlotType requestingFunc;
    SDIORWModeType mode;
    SDIOFuncType funcNum;
    UInt32 byteAddress;
    MemPtr bufferP;
    UInt16 numBlocks;
    UInt16 ioBlockSize;
} SDIORWExtendedBlockType
```

Field Descriptions

`requestingFunc`	The number of the SDIO card function making the read or write request. This is the function that will be turned on.
`mode`	The operation to be performed. See `SDIORWModeType` for a list of operations.
`funcNum`	The number of the function within the I/O card to be read or written. Function 0 selects the common I/O area (CIA).
`byteAddress`	The address of the first byte inside of the selected SDIO card function's register space that will be read or written. There are 17 bits of address available, so the byte must be located within the first 128K addresses of that function.
`bufferP`	For an extended write command, a pointer to the data that will be written. For an extended read command, the data read is stored in the indicated buffer.
`numBlocks`	The number of blocks to transfer, up to 511. A value of 0 indicates that the block transfer should go on until explicitly stopped, but that mode is not supported by this SDIO slot driver.
`ioBlockSize`	The size of each block to be transferred. This value should range from 1 to 2048; all other values are illegal.

See Also `SDIORWDirect`, `SDIORWExtendedByte`

New SDIORWExtendedByte

Purpose Read or write multiple bytes to any I/O function, including the the common I/O area (CIA), at any address using the byte mode of CMD53 (IO_RW_EXTENDED).

Prototype

```
Err SDIORWExtendedByte (UInt16 slotLibRefNum,
SDIORWExtendedByteType *extendedByteP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> extendedByteP`
Pointer to an `SDIORWExtendedByteType` structure which describes the read or write operation. See the Comments section, below, for a description of the `SDIORWExtendedByteType` structure.

Result

`errNone` No error.

`expErrCardBadSector`
The SDIO memory could not be read or written.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`expErrUnsupportedOperation`
The SDIO card does not support the specified SDIO card function.

`sysErrParamErr` `extendedByteP` is invalid, or one of its fields is invalid.

Comments This function is commonly used to initialize registers or monitor status values for I/O functions. This function requires that an SDIO card be present in the slot. The card will be turned on and accessed. See the SDIO specification for the SDIO registers that can be read or written.

NOTE: The write protect tab on the SD card is ignored by the SDIO slot driver. Issuing a write request with this function always causes the write command to be sent to the card.

This function can safely be called from within an interrupt service routine.

SDIORWExtendedByteType

This structure is used with SDIORWExtendedByte and encapsulates the read or write operation.

```
typedef struct {
    SDIOSlotType requestingFunc;
    SDIORWModeType mode;
    SDIOFuncType funcNum;
    UInt32 byteAddress;
    MemPtr bufferP;
    UInt16 numBytes;
} SDIORWExtendedByteType
```

Field Descriptions

`requestingFunc`	The number of the SDIO card function making the read or write request. This is the function that will be turned on.
`mode`	The operation to be performed. See `SDIORWModeType` for a list of operations.
`funcNum`	The number of the function within the I/O card to be read or written. Function 0 selects the common I/O area (CIA).
`byteAddress`	The address of the first byte inside of the selected SDIO card function's register space that will be read or written. There are 17 bits of address available, so the byte must be located within the first 128K addresses of that function.
`bufferP`	For an extended write command, a pointer to the data that will be written. For an extended read command, the data read is stored in the indicated buffer.
`numBytes`	The number of bytes to transfer, up to 512. A `numBytes` value of either 512 or 0 indicates that 512 bytes are to be transferred.

See Also `SDIORWDirect`, `SDIORWExtendedBlock`

New SDIOSetAutoPowerOff

Purpose Alter the auto-power-off settings for the SD slot.

Prototype

```
Err SDIOSetAutoPowerOff (UInt16 slotLibRefNum,
SDIOAutoPowerOffType *autoP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`-> autoP`
Pointer to an `SDIOAutoPowerOffType` structure which specifies the auto-power-off settings for the SD slot. Before calling `SDIOSetAutoPowerOff`, set this structure's `sdioSlotNum` field to indicate the current slot driver function number, and set the `ticksTillOff` field to the desired number of system ticks until the slot is turned off (a value of zero disables auto-off). Set the `sleepPower` field to `sdioCardPowerOff` to turn the slot off after the specified period of time.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented` (or any other error)
The specified slot driver does not support SDIO.

`sysErrParamErr` `autoP` is invalid, or one of the fields of `autoP` is invalid.

Comments This function requires that an SDIO card be present in the slot, since these settings are erased when an SDIO card is removed. Note that every time a card is inserted into the SD slot, the auto-power-off time is set to 15 seconds.

When the handheld awakes from sleep mode, it doesn't turn the card on. Only when there is a request to access the card does it turn the card on.

This function only works with SDIO cards; it cannot be used when a memory card is in the slot.

This function can safely be called from within an interrupt service routine.

See Also SDIOSetPower, SDIOGetAutoPowerOff

New SDIOSetBitMode

Purpose Change the bus width.

Prototype

```
Err SDIOSetBitMode (UInt16 slotLibRefNum,
SDIOSDBitModeRequestType *bitModeRequestP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`<-> bitModeRequestP`
Pointer to an SDIOSDBitModeRequestType structure which indicates which function is making the request, and which bit mode to set. See the Comments section, below, for a complete description of the `SDIOSDBitModeRequestType` structure.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`expErrUnsupportedOperation`
The SDIO card does not support the requested bit mode.

`sysErrParamErr` `bitModeRequestP` is invalid.

Comments Set the bit mode after the card has been inserted but before setting any callbacks. Be sure to check the value returned from this function: due to hardware constraints, this command can be rejected, returning `expErrUnsupportedOperation`.

The current bit mode can be obtained with a call to `SDIOGetCardInfo`.

This function requires that an SDIO card be present in the slot, and it may turn on and access the card.

This function can safely be called from within an interrupt service routine.

SDIOSDBitModeRequestType

This structure is passed to the `SDIOSetBitMode` function and both specifies the function making the request and the SDIO bit mode that is to be set.

```
typedef struct {
  SDIOSlotType requestingFunc;
  SDIOSDBitModeType bitMode;
} SDIOSDBitModeRequestType;
```

Field Descriptions

`requestingFunc` The number of the function making this request. See `SDIOSlotType` for the set of values to which this can be set.

`bitMode` The requested SDIO bit mode. See `SDIOSDBitModeType` for the bit modes that can be requested.

See Also `SDIOGetCardInfo`

New SDIOSetCallback

Purpose

Set pointers to an SDIO card function's callback routine and associated data.

Prototype

```
Err SDIOSetCallback (UInt16 slotLibRefNum,
SDIOCallbackType *callBackP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`-> callBackP` Pointer to an `SDIOCallbackType` structure. Before calling this function, set each of this structure's fields.

Result

`errNone` No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`sysErrParamErr` `callBackP` is invalid, or one of its fields is invalid.

Comments

You can assign a separate callback function to each callback selector for each card function slot as well as the SD Memory card slot. The following list of callback selectors details those situations in which a particular callback function is called:

- `sdioCallbackSelectInterruptSdCard`: the corresponding callback function is executed whenever the SD card interrupts the handheld. The handheld enables the SDIO interrupt whenever a valid `sdioCallbackSelectInterruptSdCard` callback is set. The callback function needs to reset the interrupt source to prevent the interrupt callback from being called again.

- `sdioCallbackSelectSleep` and `sdioCallbackSelectAwake`: the corresponding callback functions are executed whenever the handheld is about to be put to sleep or just after it wakes. These callback functions are always called with interrupts disabled, and should be as fast as possible.
- `sdioCallbackSelectPowerOn` and `sdioCallbackSelectPowerOff`: the corresponding callback functions are executed when the SDIO card power is turned on or is about to be turned off. Never call `SDIOSetPower` while processing these functions in order to turn an SDIO card function on or off.
- `sdioCallbackSelectReset`: the corresponding callback function is executed whenever `SDIOSetPower` is called with the `powerP` structure's `powerOnCard` field set to `sdioCardWaitSdio`. `sdioCardWaitSdio` is typically used after the SDIO section has been reset by setting the RES (I/O Card Reset) bit in the CCCR (Card Common Control Registers).
- `sdioCallbackSelectBitMode`: the corresponding callback function is executed whenever the bus width is successfully changed with `SDIOSetBitMode`. Note that in version 1.0 of the SDIO slot driver, this callback is never executed because the bus is always one bit wide.

When a situation arises that causes one of the above callback functions to be called, the corresponding callback for the SD Memory card slot is generally the first one called, followed by the corresponding callback functions for SDIO functions 1 through 7. Because the SD Memory card slot and each SDIO card function slot can have a separate callback function for each callback selector, each callback function can limit itself to dealing with a single selector and a single SDIO card function.

If you use **any** of these callbacks, you may not have control of the user interface or access to your variables. Make any necessary data available to your callback function through the use of `userDataP`. Be sure to lock memory for your callback functions and variables.

Callback functions must be interrupt-safe: they should only call interrupt-safe functions. Your callback functions can be called from within an interrupt service routine, and interrupts can occur at **any**

time. The interrupt can even generate a wakeup event if the handheld is asleep and power to the card is still on. As with an interrupt service routine, your callback functions can only call a limited set of system functions and must execute quickly. Your callback functions do have access to any slot driver functions accessible through the `SlotCustomControl` function.

Callback function information is automatically erased after a card is inserted or removed (before the card removal event). Because of this, `SDIOSetCallback` can only be used when a card is in the SD slot. To detect card removal, use the notification manager and register for `sysNotifyCardRemovedEvent`.

`SDIOSetCallback` can safely be called from within an interrupt service routine.

New SDIOSetCurrentLimit

Purpose

Set the maximum peak current needed by one of the SDIO card's functions.

Prototype

```
Err SDIOSetCurrentLimit (UInt16 slotLibRefNum,
SDIOCurrentLimitType *currentLimitP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`-> currentLimitP`
Pointer to an SDIOCurrentLimitType structure. Before calling `SDIOSetCurrentLimit`, set this structure's `slotFuncNum` field to a valid slot driver function number, and set the `uaMaximum` field to the maximum peak current, in micro-amps, required by the function indicated by `slotFuncNum`.

Result

`errNone` — No error.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`sysErrParamErr` `currentLimitP` is invalid, or one of `currentLimitP`'s fields is invalid.

Comments

You use `SDIOGetCurrentLimit`, `SDIOSetCurrentLimit`, and `SDIORemainingCurrentLimit` to ensure that the total of all function hardware that is active never exceeds the SDIO specification maximum of 200ma. These three functions do not detect or limit current draw, check the battery level, or reflect how much energy the battery has left; you simply use them to manage the current limit values supplied using `SDIOSetCurrentLimit`. It is up to the writer of the SDIO slot driver to both supply the proper current limit values and to use `SDIOGetCurrentLimit` and `SDIORemainingCurrentLimit` appropriately so that the total active SDIO card functions do not draw more current than the handheld's power source can provide.

Note that this function doesn't write the supplied peak current value to the card; it only sets the value in RAM.

When a card is removed, all allocations of current are set to zero. Because of this, in order to operate properly this function requires an SDIO card in the slot.

This function can safely be called from within an interrupt service routine.

New SDIOSetPower

Purpose

Turns an SDIO card function on or off.

Prototype

```
Err SDIOSetPower (UInt16 slotLibRefNum,
SDIOPowerType *powerP)
```

Parameters

`-> slotLibRefNum`
Slot driver library reference number.

`-> powerP`
Pointer to an `SDIOPowerType` structure. Before calling `SDIOSetPower` set this structure's `sdioSlotNum` field to indicate the SDIO card function to be turned on or off, and set the `powerOnCard` field to one of the values defined for SDIOCardPowerType.

Result

`errNone` No error.

`expErrCardBadSector`
The card could not be initialized.

`expErrCardNotPresent`
There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`sysErrParamErr` `powerP` is invalid.

Comments

When used to turn an SDIO card function on, `SDIOSetPower` if necessary, sets the power and bus signals such that the entire slot is turned on: power is applied to the card and the data bus is ready to transmit or receive commands. When used to turn an SDIO card function off, power is removed from the card and the data bus is set to a low power state if no other functions are on. Note that when the card is turned off, SDIO interrupts cannot occur.

`SDIOSetPower` requires that an SDIO card be present in the slot. Turning on power causes the card to be accessed, for initialization.

This function can safely be called from within an interrupt service routine. However, you should not call `SDIOSetPower` from your `sdioCallbackSelectPowerOn` or `sdioCallbackSelectPowerOff` callback functions.

See Also `SDIOSetAutoPowerOff`, `SDIOSetCurrentLimit`, `SDIOGetPower`

New SDIOTupleWalk

Purpose Search an SDIO card function's Card Information Structure (CIS) for a particular data block (tuple) and return the contents of the data block.

Prototype

```
Err SDIOTupleWalk (UInt16 slotLibRefNum,
SDIOTupleType *tupleP)
```

Parameters

`-> slotLibRefNum` Slot driver library reference number.

`-> tupleP` Pointer to an `SDIOTupleType` structure which identifies the tuple to be located and indicates where the results should be placed. See the Comments section, below, for a description of the `SDIOTupleType` structure.

Result

`errNone` No error.

`expErrCardBadSector` The SDIO card's function memory could not be read or the requested tuple was not found

`expErrCardNotPresent` There isn't a card in the slot associated with the specified slot driver.

`expErrUnimplemented`
The specified slot driver does not support SDIO.

`expErrUnsupportedOperation`
The SDIO card does not support the specified SDIO card function.

`sysErrParamErr` `tupleP` is invalid, or one of its fields is invalid.

Comments This function requires that an SDIO card be present in the slot. The card will be turned on and accessed.

This function can safely be called from within an interrupt service routine.

SDIOTupleType

This structure is used with `SDIOTupleWalk` and both identifies the data block (tuple) to be located and indicates where the search results should be placed.

```
typedef struct {
    SDIOSlotType requestingFunc;
    SDIOFuncType funcNum;
    UInt8 tupleToFind;
    MemPtr bufferP;
    UInt16 bufferSizeOf;
} SDIOTupleType
```

Field Descriptions

`requestingFunc` The number of the SDIO card function making the read or write request. This is the function that will be turned on.

`funcNum` The number of the function within the I/O card to be searched. Function 0 selects the common I/O area (CIA).

`tupleToFind` The tuple code that identifies the desired data block.

`bufferP`	Pointer to a buffer into which the contents of the tuple are placed.
`bufferSizeOf`	The size of the supplied buffer. This buffer should be large enough to contain the entire tuple (including bytes for the tuple code and the tuple body size). According to the SDIO specification, a tuple is never larger 257 bytes.

Application-Defined Functions

New SDIOCallbackPtr

Purpose Perform driver-specific processing when one of the following occurs:

- the SDIO interrupt is received
- the handheld is going to sleep or has just awakened
- the SDIO card was just turned off or on
- the SDIO card was just reset
- the bus width changed

Which of the above events causes a given callback function to be called, if any, depends on what was passed to `SDIOSetCallback`.

Prototype

```
typedef Err (*SDIOCallbackPtr)
(SDIOSlotType sdioSlotNum, void *userDataP)
```

Parameters

`-> sdioSlotNum`

The ID of the SDIO card function in the current slot driver that generated the callback. See the description of `SDIOSlotType` for the complete set of values that can be supplied here.

`-> userDataP` Pointer to a block of user data specified when the callback was set up using `SDIOSetCallback`.

Result Return `errNone` if the callback function executed properly, or any other value if an error occurred during the processing of the callback.

Comments In your callback function you may not have control of the user interface or access to your variables. Any necessary data should be made available to your callback function through the use of `userDataP`. Be sure to keep memory for your callback functions and variables locked from the time you set up the callback function to the time when it can no longer be called.

Callback function information is automatically erased after a card is inserted or removed (before the card removal event). To detect card removal, use the notification manager and register for `sysNotifyCardRemovedEvent`.

Your callback functions can be called from within an interrupt service routine, and interrupts can occur at **any** time. The interrupt can even generate a wakeup event if the handheld is asleep and power to the card is still on. As with an interrupt service routine, your callback functions can only call a limited set of system functions (those that are interrupt-safe) and must execute quickly. Your callback functions do have access to any slot driver functions accessible through the `SlotCustomControl` function.

Index

T

V