Advanced SCSI Programming Interface

ASPI for Win32
Technical Reference
November 6, 2001

å

Copyright

Trademarks

Adaptec, the Adaptec logo, and AHA are trademarks of Adaptec, Inc. which may be registered in some jurisdictions.

All other trademarks are owned by their respective owners.

Changes

The material in this document is for information only and is subject to change without notice. While reasonable efforts have been made in the preparation of this document to assure its accuracy, Adaptec, Inc. assumes no liability resulting from errors or omissions in this document, or from the use of the information contained herein.

Adaptec reserves the right to make changes in the product design without reservation and without notification to its users.

Adaptec Warranties, Technical Support and Services

THE ADAPTEC SOFTWARE IS PROVIDED "AS IS". THERE ARE NO WARRANTIES AND ADAPTEC EXPRESSLY DISCLAIMS ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR PARTICULAR PURPOSE. Adaptec has no obligation to provide any modifications, improvements, updates, training or support relating to the Adaptec Software. Any such matters, if applicable, shall be subject to mutual written agreement between the parties.

For licensing information, please contact Adaptec's Diane McGee at (408) 957-4836 or diane_mcgee@adaptec.com.

ASPI for Win32 Overview

The architecture of SCSI makes it possible to access a wide variety of devices using a single bus linked to a host computer with a SCSI host adapter. Support for peripheral devices in Windows (98, ME, NT, 2000 and Windows XP (32-bit)) is normally achieved through device specific drivers layered on top of the operating systems' native SCSI support.

Because of the tremendous diversity of SCSI devices, no driver can support all SCSI peripherals. Instead, separate drivers are needed for each major class of installed SCSI device. These drivers share the host adapter hardware through the operating systems' native SCSI support. The native SCSI layers are different between Windows 98/ME and Windows NT/2000/XP (32-bit). In addition, development and debugging of VxDs or kernel mode drivers can be very difficult. The need for a standard SCSI programming interface to simplify SCSI application development and ease the porting of SCSI applications from one Win32 platform to another brought ASPI for Win32 into existence.

The Advanced SCSI Programming Interface (ASPI) for Win32 was designed to increase compatibility and simplify the connection of SCSI peripheral devices like tape, CD-ROM, WORM, magneto-optical, scanners, and other devices. It defines a protocol for SCSI applications (called ASPI modules) to submit I/O requests to a single operating system driver (called the ASPI manager). Access to the operating system driver is made through a Dynamic Link Library named WNASPI32.DLL.

Before Beginning

Before you begin your ASPI for Win32 development effort, be sure that you have a solid understanding of the SCSI specifications. Much of your success in developing an ASPI module is dependent on your understanding of these specifications. Here are sources for the specifications:

SCSI-1 and CCS:	American National Standards Institute 25 West 43rd Street, 4th Fl. NY, NY 10036 Phone: (212) 642-4900 Fax: (212) 398-0023
SCSI-2 and SCSI-3:	Global Engineering Documents World Headquarters 15 Iverness Way East Englewood, CO 80112-5776 Phone: (800) 854-7179 Phone: (303) 397-7956 Fax: (314) 726-6418 http://global.ihs.com/
SCSI BBS:	(719) 574-0424

In addition, it is highly recommended that you acquire the technical reference manuals for any SCSI hardware which your ASPI module intends to support. These manuals can be obtained from the hardware manufacturer, and they provide detailed information on which SCSI commands are supported and how they are implemented.

Programming Conventions

This specification contains function prototypes and structure definitions with the following data types:

Type	Size (Bytes)	Description
VOID	N/A	Indicates lack of a return value or lack of function arguments.
BYTE	1	Unsigned 8-bit value.
WORD	2	Unsigned 16-bit value.
DWORD	4	Unsigned 32-bit value.
LPVOID	4	Generic pointer. Used in SRB fields which require either a pointer to a function or a Win32 handle (for example, SRB_PostProc).
LPBYTE	4	Pointer to an array of BYTEs. Mainly used as a buffer pointer.
LPSRB	4	Generic pointer to one of the SRB_* structures defined below.

Unless otherwise noted, all multibyte fields follow Intel's byte order of low byte first and end with the high byte. For example, if there is a 2-byte offset field, the first byte is the low byte of the offset while the second byte is the high byte of the offset.

All structure fields marked reserved *must be set to zero*, and structures *must be packed!* Packed means that byte alignment is used on all structure definitions. Microsoft compilers allow byte packing to be set through the use of "#pragma pack(1)" while Borland compilers allow packing to be set with "#pragma option -a1". See your compiler documentation for more information. Failure to pack structures and zero reserved fields can cause system instability, including crashes.

All ASPI for Win32 functions are exported from WNASPI32.DLL using the 'C' calling convention (specifically, __cdecl as implemented by Microsoft's compilers). With the 'C' calling convention the caller pushes the last function argument on the stack first (the first argument has the lowest memory address), and the caller is responsible for popping arguments from the stack.

Programming Guidelines

The following are some general guidelines to keep in mind while reading this specification and while writing ASPI for Win32 applications:

- If you are using explicit dynamic linking, remember that the ASPI for Win32 DLL is named WNASPI32.DLL and not WINASPI.DLL. Make sure to call LoadLibrary appropriately.
- ASPI for Win32 is fully re-entrant and permits overlapped, asynchronous I/O. ASPI modules can send additional ASPI requests while others are pending completion. Be sure to use separate SRBs for each ASPI request.
- For requests requiring data transfers, the direction bits in the SRB_Flags field must be set correctly. Direction bits are no longer optional for data transfers. This means that SRB_DIR_SCSI is no longer a valid setting. For requests not requiring data transfers, the direction bits are ignored.
- Be sure that buffers are aligned according to the buffer alignment mask returned by the SC_HA_INQUIRY command.
- ASPI SCSI Request Blocks (SRBs) and data buffers do not need to be in page-locked memory. The ASPI manager takes care of locking buffers and SRBs.
- If an error SS_BUFFER_TOO_BIG is returned by the SendASPI32Command routine, you should break the transfer down into multiple 64KByte transfers or less. Another alternative is to use the GetASPI32Buffer/FreeASPI32Buffer calls to allocate large transfer buffers.
- If you send an ASPI request with posting (callbacks) enabled, the post procedure will *always* be called.

- The CDB area has been fixed in length at 16. Therefore, the sense data area no longer shifts location depending on command length as in ASPI for Win16. If you are developing an application targeted only at Win32, you no longer need to account for the "floating" sense buffer.
- When scanning for devices, the `SendASPI32Command` may also return the status `SS_NO_DEVICE` in the `SRB_Status` field. Check for this exception in addition to the host adapter status `HASTAT_SEL_TO`.
- Error codes are located at the end of this technical reference.

Calling ASPI Functions

Applications which utilize ASPI for Win32 are known as ASPI modules. ASPI modules interact with ASPI through WNASPI32.DLL which is a dynamic-link library with five entry points:

Entry Point	Description
`GetASPI32SupportInfo`	Initializes ASPI and returns basic configuration information.
`SendASPI32Command`	Submits SCSI Request Blocks (SRBs) for execution by ASPI.
`GetASPI32Buffer`	Allocates buffers which meet Win98/WinNT large transfer requirements.
`FreeASPI32Buffer`	Releases buffers previously allocated with `GetASPI32Buffer`.
`TranslateASPI32Address`	Translates ASPI HA/ID/LUN address triples to/from Windows 98 DEVNODEs.

In order to access these five functions, they must be resident in memory. Dynamic linking is the process by which Windows loads dynamic-link libraries (DLLs) into memory and then resolves application references to functions within those DLLs. There are two ways in which this load/resolve sequence is handled: explicitly or implicitly.

Explicit Dynamic Linking

Explicit dynamic linking occurs when applications or other DLLs *explicitly* load a DLL using LoadLibrary and then manually resolve references to individual DLL functions through calls to GetProcAddress. This is the preferred method for loading and calling ASPI for Win32. Explicit dynamic linking allows complete control over when ASPI is loaded and how load errors are handled. It also is the only way to detect if the three newer ASPI functions are available for use in an application.

The following block of code is all that is required to load ASPI:

```
HINSTANCE hinstWNASPI32;

hinstWNASPI32 = LoadLibrary( "WNASPI32" );
if( !hinstWNASPI32 )
{
    // Handle ASPI load error here.  Usually this involves the display of an
    // informative message based on the results of a call to GetLastError().
}
```

Once a valid instance handle for ASPI is obtained, GetProcAddress is used to obtain addresses for each of the ASPI for Win32 entry points:

```
DWORD  (*pfnGetASPI32SupportInfo)( void );
DWORD  (*pfnSendASPI32Command)( LPSRB );
BOOL   (*pfnGetASPI32Buffer)( PASPI32BUFF );
BOOL   (*pfnFreeASPI32Buffer)( PASPI32BUFF );
BOOL   (*pfnTranslateASPI32Address)( PDWORD, PDWORD );

pfnGetASPI32SupportInfo   = GetProcAddress( hinstWNASPI32, "GetASPI32SupportInfo"
);
pfnSendASPI32Command      = GetProcAddress( hinstWNASPI32, "SendASPI32Command" );
pfnGetASPI32Buffer        = GetProcAddress( hinstWNASPI32, "GetASPI32Buffer" );
pfnFreeASPI32Buffer       = GetProcAddress( hinstWNASPI32, "FreeASPI32Buffer" );
pfnTranslateASPI32Address = GetProcAddress( hinstWNASPI32,"TranslateASPI32Address"
);
```

At this point there should be a valid address for each of the five functions. If you have an old version of ASPI then the last three function addresses will be NULL. This case should be handled by disabling all use of new features in your ASPI module. It is also good practice to check pfnGetASPI32SupportInfo and pfnSendASPI32Command for NULL as well. These variables will be NULL if there is an error accessing the DLL. If either of these two functions have NULL addresses your application should cease its use of ASPI and unload WNASPI32.DLL with a call to FreeLibrary.

Using the addresses returned from GetProcAddress is very simple. Just use the variable name wherever you would normally use a function name. For example,

```
DWORD dwASPIStatus = pfnGetASPI32SupportInfo();
```

will call the `GetASPI32SupportInfo` and place the result in dwASPIStatus. Of course, if one of these function pointers is NULL and you make a call to it, your application will crash.

Implicit Dynamic Linking

Implicit dynamic linking occurs when a dependent DLL is loaded as a result of loading another module. This dependency can be established either by listing exported functions from the DLL in the IMPORTS section of a ".DEF" file linked with the application.

Implicit dynamic linking is not recommended for three reasons:

- You cannot control when ASPI is loaded. Like anything else, ASPI consumes system resources. When you use implicit dynamic linking those resources are allocated as soon as the application starts, and they remain allocated until the application shuts down. With explicit dynamic linking the application controls when (and if) ASPI is loaded.
- You have no control over how load errors are reported to users. If ASPI is not found during an implicit load a fairly ugly error message (sometimes two) is displayed by the operating system. If you use explicit loading in conjunction with a call to `SetErrorMode( SEM_NOOPENFILEERRORBOX )` then your application can fully handle any load errors on its own.
- Your application cannot recover if it relies on new ASPI features and it is run with an older version of ASPI. If your application relies on `GetASPI32Buffer, FreeASPI32Buffer,` or `TranslateASPI32Address`, and then that function is not found in the loaded version of WNASPI32.DLL, then the load fails. By using explicit dynamic linking the application can alter its behavior so that the functions are not used. For example, an application which "relies" on TranslateASPI32Address could simply disable Plug and Play support if the function is not found in the DLL.

GetASPI32SupportInfo

The GetASPI32SupportInfo function returns the number of host adapters installed and ensures that the ASPI manager is initialized properly. This function must be called once at initialization time, before SendASPI32Command is accessed.

```
DWORD GetASPI32SupportInfo( VOID );
```

Parameters

None.

Return Values

The DWORD return value is split into three pieces. The high order WORD is reserved and shall be set to 0. The two low order bytes represent a status code (bits 15-8) and a host adapter count (bits 7-0).

If the call to GetASPI32SupportInfo is successful, then the status byte is set to either SS_COMP or SS_NO_ADAPTERS. If set to SS_COMP then the host adapter status will be non-zero. An error code of SS_NO_ADAPTERS indicates that ASPI initialized successfully, but that it could not find any SCSI host adapters to manage.

If the function fails the status byte will be set to one of SS_ILLEGAL_MODE, SS_NO_ASPI, SS_MISMATCHED_COMPONENTS, SS_INSUFFICIENT_RESOURCES, SS_FAILED_INIT. See the table of ASPI errors at the end of this manual for more information on each of the errors.

Remarks

The number of host adapters returned represents the logical bus count, not the true physical adapter count. For host adapters with a single bus, the host adapter count and logical bus count are identical.

Example

This example returns the current status of ASPI for Win32.

```
BYTE   byHaCount;
BYTE   byASPIStatus;
DWORD  dwSupportInfo;

dwSupportInfo = GetASPI32SupportInfo();
byASPIStatus = HIBYTE(LOWORD(dwSupportInfo));
byHaCount = LOBYTE(LOWORD(dwSupportInfo));

if( byASPIStatus != SS_COMP && byASPIStatus != SS_NO_ADAPTERS )
{
    // Handle ASPI error here.  Usually this involves the display
    // of a dialog box with an informative message.
}
```

SendASPI32Command

The SendASPI32Command function handles all SCSI I/O requests. Each SCSI I/O request is handled through a SCSI Request Block (SRB) which defines the exact ASPI operation to be performed.

```
DWORD SendASPI32Command( LPSRB psrb );
```

Parameters

psrb

All SRBs have a standard header, and the header contains a command code which defines the exact type of SCSI I/O being requested.

```
typedef struct
{
    BYTE   SRB_Cmd;        // ASPI command code
    BYTE   SRB_Status;     // ASPI command status byte
    BYTE   SRB_HaId;       // ASPI host adapter number
    BYTE   SRB_Flags;            // ASPI request flags
    DWORD  SRB_Hdr_Rsvd;   // Reserved, MUST = 0
}
SRB_Header;
```

The SRB_Cmd field contains the command code for the desired SCSI I/O operation. This field can be set to one of the following values.

Symbol	Value	Description
SC_HA_INQUIRY	0x00	Queries ASPI for information on specific host adapters.
SC_GET_DEV_TYPE	0x01	Requests the SCSI device type for a specific SCSI target.
SC_EXEC_SCSI_CMD	0x02	Sends a SCSI command (arbitrary CDB) to a SCSI target.
SC_ABORT_SRB	0x03	Requests that ASPI cancel a previously submitted request.
SC_RESET_DEV	0x04	Sends a BUS DEVICE RESET message to a SCSI target.
SC_GET_DISK_INFO	0x06	Returns BIOS information for a SCSI target (Win98 only).
SC_RESCAN_SCSI_BUS	0x07	Requests a rescan of a host adapter's SCSI bus.
SC_GETSET_TIMEOUTS	0x08	Sets SRB timeouts for specific SCSI targets.

The use of the remaining header fields varies according to the command type. Each of the commands along with their associated SRBs are described in detail in the following sections.

Return Values

The above ASPI commands may be broken into two categories: synchronous and asynchronous. All of the SRBs are synchronous except for SC_EXEC_SCSI_CMD and SC_RESET_DEV which are asynchronous.

Calls to SendASPI32Command with synchronous SRBs will not return until execution of that SRB is complete. Upon return the SRB_Status field will be set to the same value which is returned from SendASPI32Command.

Calls to SendASPI32Command with asynchronous SRBs may return control to the caller before the submitted SRB has completed execution. In this case the return value from this function is SS_PENDING, and the caller will have to use polling, posting, or event notification to wait for SRB completion. Once completed, the SRB_Status field contains the true completion status. Remember that while waiting for SRB completion, it is always safe to submit additional SRBs to ASPI for execution.

See the “Waiting for Completion” and “ASPI for Win32 Errors” sections for more information on synchronous/asynchronous SRBs and the various error codes which can be returned either from this function or within an `SRB_Status` field.

SC_HA_INQUIRY

The SendASPI32Command function with command code SC_HA_INQUIRY is used to get information on the installed host adapter hardware, including the number of host adapters installed.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_HA_INQUIRY
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // Reserved, MUST = 0
    DWORD   SRB_Hdr_Rsvd;           // Reserved, MUST = 0
    BYTE    HA_Count;               // Number of host adapters present
    BYTE    HA_SCSI_ID;             // SCSI ID of host adapter
    BYTE    HA_ManagerId[16];       // String describing the manager
    BYTE    HA_Identifier[16];      // String describing the host adapter
    BYTE    HA_Unique[16];          // Host Adapter Unique parameters
    WORD    HA_Rsvd1;               // Reserved, MUST = 0
}
SRB_HAInquiry, *PSRB_HAInquiry;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_HA_INQUIRY (0x00).

SRB_Status (Output)

SC_HA_INQUIRY is a synchronous SRB. On return, this field is the same as the SendASPI32Command return value and is set to either SS_COMP or SS_INVALID_HA.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for. Host adapter numbers are always assigned by the ASPI manager, beginning with zero. To determine the total number of host adapters in the system set this field to 0 and then check the HA_Count value on return. GetASPI32SupportInfo can also be used.

HA_Count (Output)

The number of host adapters detected by ASPI. For example, a return value of 2 indicates that host adapters #0 and #1 are valid. The number of host adapters returned represents the logical bus count instead of the true physical adapter count. For host adapters that support single bus only, the host adapter count and logical bus count are identical. For host adapters that support multiple buses, the host adapter count represents the total logical bus count.

HA_SCSI_ID (Output)

The SCSI ID of the host adapter on the SCSI bus. SCSI adapters usually use ID 7 as their SCSI ID.

HA_ManagerId (Output)

The ASCII string “ASPI for Win32”. The string is padded with spaces to the full width of the buffer, and it is *not* null terminated.

HA_Identifier (Output)

An ASCII string describing the host adapter. The string is padded with spaces to the full width of the buffer, and it is *not* null terminated.

HA_Unique (Output)

Host adapter unique parameters as follows.

Size	Offset	Description
WORD	0	Buffer alignment mask. The host adapter requires data buffer alignment specified by this 16-bit value. A value of 0x0000 indicates no boundary requirements (e.g. byte alignment), 0x0001 indicates word alignment, 0x0003 indicates double-word, 0x0007 indicates 8-byte alignment, etc. The 16-bit value allows data buffer alignments of up to 65536-byte boundaries. Alignment of buffers can be tested by logical ANDing ('&' in 'C') this mask with the buffer address. If the result is 0 the buffer is properly aligned.
BYTE	2	Residual byte count. Set to 0x01 if residual byte counting is supported, 0x00 if not. See "Remarks" below for more information.
BYTE	3	Maximum SCSI targets. Indicates the maximum number of targets (SCSI IDs) the adapter supports. If this value is not set to 8 or 16, then it should be assumed by the application that the maximum target count is 8.
DWORD	4	Maximum transfer length. DWORD count indicating the maximum transfer size the host adapter supports. If this number is less than 64KB then the application should assume a maximum transfer count of 64KB.

Remarks

Residual byte length is the number of bytes not transferred to, or received from, the target SCSI device. For example, if the ASPI buffer length for a SCSI INQUIRY command is set for 100 bytes, but the target only returns 36 bytes; the residual length is 64 bytes. If the ASPI buffer length for a SCSI WRITE command is set for 514 bytes but the target only takes 512 bytes, the residual length is 2 bytes. ASPI modules can determine if the ASPI manager supports residual byte length by checking byte 1 of the `HA_Unique` field. See `SC_EXEC_SCSI_CMD` for more information on enabling residual byte counting.

Example

This example sends an `SC_HA_INQUIRY` to host adapter #1, and, if successful, records the maximum transfer length supported by the host adapter.

```
DWORD          dwMaxTransferBytes;
SRB_HAInquiry  srbHAInquiry;

memset( &srbHAInquiry, 0, sizeof(SRB_HAInquiry) );
srbHAInquiry.SRB_Cmd = SC_HA_INQUIRY;
srbHAInquiry.SRB_HaId = 1;

SendASPI32Command( (LPSRB)&srbHAInquiry );
if( srbHAInquiry.SRB_Status != SS_COMP )
{
    // Error in HAInquiry.  Most likely SS_INVALID_HA.
    Return FALSE;
}

dwMaxTransferBytes = *(DWORD *)(srbHAInquiry.HA_Unique + 4);
```

SC_GET_DEV_TYPE

The SendASPI32Command function with command code SC_GET_DEV_TYPE enables you to identify the devices available on the SCSI bus. A Win32 tape backup package, for example, can scan each target/LUN on each installed host adapter looking for a device type corresponding to sequential access devices. This eliminates the need for each Win32 application to duplicate the effort of scanning the SCSI bus for devices.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_GET_DEV_TYPE
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // Reserved, MUST = 0
    DWORD   SRB_Hdr_Rsvd;           // Reserved, MUST = 0
    BYTE    SRB_Target;             // Target's SCSI ID
    BYTE    SRB_Lun;                // Target's LUN number
    BYTE    SRB_DeviceType;         // Target's peripheral device type
    BYTE    SRB_Rsvd1;              // Reserved, MUST = 0
}
SRB_GDEVBlock, *PSRB_GDEVBlock;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_GET_DEV_TYPE (0x01).

SRB_Status (Output)

SC_GET_DEV_TYPE is a synchronous SRB. On return, this field is the same as the SendASPI32Command return value and is set to SS_COMP, SS_INVALID_HA, or SS_NO_DEVICE.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for.

SRB_Target (Input)

SCSI ID of target device.

SRB_Lun (Input)

Logical Unit Number (LUN) of target device.

SRB_DeviceType (Output)

The peripheral device type. The value is one of the codes defined by the SCSI specification.

Symbol	Value	Description
DTYPE_DASD	0x00	Direct-access device (e.g. magnetic disk)
DTYPE_SEQD	0x01	Sequential-access device (e.g. magnetic tape)
DTYPE_PRNT	0x02	Printer device
DTYPE_PROC	0x03	Processor device
DTYPE_WORM	0x04	Write-once device (e.g. some optical disks)
DTYPE_CDROM	0x05	CD-ROM device
DTYPE_SCAN	0x06	Scanner device
DTYPE_OPTI	0x07	Optical memory device (e.g. some optical disks)
DTYPE_JUKE	0x08	Medium changer device (e.g. jukeboxes)
DTYPE_COMM	0x09	Communication device
N/A	0x0A-0x0B	Defined by ASC IT8 (Graphic arts pre-press devices)
N/A	0x0C-0x1E	Reserved
DTYPE_UNKNOWN	0x1F	Unknown or no device type

Example

This example scans the system for all CD-ROM drives (all targets must be at LUN #0). Please note that `MAX_HA_ID` and `MAX_TARGET_ID` should be replaced with a host adapter count returned by `GetASPI32SupportInfo` and a target count retrieved from a `SC_HA_INQUIRY SRB` performed within the host adapter loop.

```
BYTE            byHaId;
BYTE            byTarget;
SRB_GDEVBlock   srbGDEVBlock;

for( byHaId = 0; byHaId < MAX_HA_ID; byHaId++ )
{
    for( byTarget = 0; byTarget < MAX_TARGET_ID; byTarget++ )
    {
        memset( &srbGDEVBlock, 0, sizeof(SRB_GDEVBlock) );
        srbGDEVBlock.SRB_Cmd = SC_GET_DEV_TYPE;
        srbGDEVBlock.SRB_HaId = byHaId;
        srbGDEVBlock.SRB_Target = byTarget;

        SendASPI32Command( (LPSRB)&srbGDEVBlock );
        if( srbGDEVBlock.SRB_Status != SS_COMP ) continue;

        if( srbGDEVBlock.SRB_DeviceType == DTYPE_CDROM )
        {
            // A CD-ROM exists at HA/ID/LUN = byHaId/byTarget/0.
            // Do whatever you want with it from here!
        }
    }
}
```

SC_EXEC_SCSI_CMD

The SendASPI32Command function with command code SC_EXEC_SCSI_CMD is used to execute a SCSI I/O command. Once an ASPI client has initialized, virtually all I/O is performed with this command.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_EXEC_SCSI_CMD
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // ASPI request flags
    DWORD   SRB_Hdr_Rsvd;           // Reserved, MUST = 0
    BYTE    SRB_Target;             // Target's SCSI ID
    BYTE    SRB_Lun;                // Target's LUN number
    WORD    SRB_Rsvd1;              // Reserved for Alignment
    DWORD   SRB_BufLen;             // Data Allocation Length
    LPBYTE  SRB_BufPointer;         // Data Buffer Pointer
    BYTE    SRB_SenseLen;           // Sense Allocation Length
    BYTE    SRB_CDBLen;             // CDB Length
    BYTE    SRB_HaStat;             // Host Adapter Status
    BYTE    SRB_TargStat;           // Target Status
    LPVOID  SRB_PostProc;           // Post routine
    BYTE    SRB_Rsvd2[20];          // Reserved, MUST = 0
    BYTE    CDBByte[16];            // SCSI CDB
    BYTE    SenseArea[SENSE_LEN+2]; // Request Sense buffer
}
SRB_ExecSCSICmd, *PSRB_ExecSCSICmd;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_EXEC_SCSI_CMD (0x02).

SRB_Status (Output)

SC_EXEC_SCSI_CMD is an asynchronous SRB. This field should not be examined until *after* the caller has waited for proper completion of the SRB (see “Waiting for Completion”). Once completed, this field may be set to a number of different values. The most common values are SS_COMP or SS_ERR. SS_COMP indicates successful completion while SS_ERR indicates the caller should examine the SRB_HaStat and SRB_TargStat fields for more information. See “ASPI for Win32 Error” for a complete description of possible error codes.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for. Host adapter numbers are always assigned by the SCSI manager layer beginning with zero.

SRB_Flags (Input)

One or more of the following flags (note restrictions where they apply):

Symbol	Value	Description
SRB_POSTING	0x01	Enable posting. See "Waiting for Completion" for more information. This flag and SRB_EVENT_NOTIFY are mutually exclusive.
SRB_ENABLE_RESIDUAL_COUNT	0x04	Enables residual byte counting assuming it is supported. Whenever a data underrun occurs the SRB_BufLen field is updated to reflect the remaining bytes to transfer.
SRB_DIR_IN	0x08	Data transfer is from SCSI target to host. Mutually exclusive with SRB_DIR_OUT.
SRB_DIR_OUT	0x10	Data transfer is from host to SCSI target. Mutually exclusive with SRB_DIR_IN.
SRB_EVENT_NOTIFY	0x40	Enable event notification. See "Waiting for Completion" for more infomration. This flag and SRB_POSTING are mutually exclusive.

SRB_Target (Input)

SCSI ID of target device.

SRB_Lun (Input)

Logical Unit Number (LUN) of target device.

SRB_BufLen (Input)

This field indicates the number of bytes to be transferred. If the SCSI command to be executed does not transfer data (e.g., Test Unit Ready, Rewind, etc.), this field *must* be set to zero. If residual byte length is supported (see "SC_HA_INQUIRY") and selected (see SRB_Flags above), this field is returned with the residual number of bytes (usually 0).

SRB_BufPointer (Input)

This field is a pointer to the data buffer. If there is no data to be transfered this field should be NULL.

SRB_SenseLen (Input)

This field indicates the number of bytes allocated at the end of the SRB for sense data. A request sense is automatically generated if a check condition is presented at the end of a SCSI command. Please note that under Windows NT it is not possible to reliably request more than 18 bytes of sense data.

SRB_CDBLen (Input)

This field establishes the length, in bytes, of the SCSI Command Descriptor Block (CDB). This value is typically 6, 10, or 12. See the SCSI specification for more information on valid CDBs.

SRB_HaStat (Output)

Upon completion of the SCSI command, this field is set to the host adapter status. Do not examine this status byte if `SRB_Status` is set to `SS_COMP`. It is only to be considered valid if there is unsuccessful completion of the SRB.

Symbol	Value	Description
HASTAT_OK	0x00	Host adapter did not detect an error.
HASTAT_TIMEOUT	0x09	The time allocated for a bus transaction ran out.
HASTAT_COMMAND_TIMEOUT	0x0B	SRB expired while waiting to be processed.
HASTAT_MESSAGE_REJECT	0x0D	MESSAGE REJECT received while processing SRB.
HASTAT_BUS_RESET	0x0E	A bus reset was detected.
HASTAT_PARITY_ERROR	0x0F	A parity error was detected.
HASTAT_REQUEST_SENSE_FAILED	0x10	The adapter failed in issuing a Request Sense after a check condition was reported by the target device.
HASTAT_SEL_TO	0x11	Selection of target timed out.
HASTAT_DO_DU	0x12	Data overrun.
HASTAT_BUS_FREE	0x13	Unexpected Bus Free.
HASTAT_PHASE_ERR	0x14	Target Bus phase sequence failure.

SRB_TargStat (Output)

Upon completion of the SCSI command, this field is set to the final SCSI target status. Do not examine this status byte if `SRB_Status` is set to `SS_COMP`. It is only to be considered valid if there is unsuccessful completion of the SRB. Note that the table below only covers the most common result codes. Check the SCSI specification for more information on these and other status byte codes.

Symbol	Value	Description
STATUS_GOOD	0x00	No target status.
STATUS_CHKCOND	0x02	Check status (sense data is in SenseArea).
STATUS_BUSY	0x08	Specified Target/LUN is busy.
STATUS_RESCONF	0x18	Reservation conflict.

SRB_PostProc (Input)

If posting is enabled (`SRB_POSTING`) this field contains a pointer to a function. The ASPI manager calls this function upon completion of the SRB. If event notification is enabled (`SRB_EVENT_NOTIFY`) this field contains a handle to an event. The ASPI manager signals this event upon completion of the SRB. See "Waiting for Completion" for more information.

CDBByte (Input)

This field contains the CDB as defined by the target's SCSI command set. The length of the SCSI CDB is specified in the `SRB_CDBLen` field.

SenseArea (Output)

The SenseArea is filled with the sense data after a check condition (`SRB_Status == SS_ERR` and `SRB_TargStat == STATUS_CHKCOND`). The maximum length of this field is specified in the `SRB_SenseLen` field.

Example

This example sends a SCSI INQUIRY command to host adapter #0, target #5, LUN #0. When examining the code, please note the following:

- Manual-reset events are used. The ResetEvent is not needed in this particular sample because we just created the event, but it is good practice to put the reset immediately before every `SendASPI32Command` call to make sure you don't enter the routine with an event signalled.
- Because this is an asynchronous SRB, we fully wait for completion before checking the `SRB_Status` byte. Also, we use `dwASPIStatus` instead of `SRB_Status` to check for a `SS_PENDING` return for the same reason.
- There is an `INFINITE` timeout on the `WaitForSingleObject` because SRB timeouts are *not* the same as event timeouts. Use `SC_GETSET_TIMEOUT` to associate a timeout with an SRB.

```
BYTE             byInquiry[32];
DWORD            dwASPIStatus;
HANDLE           heventSRB;
SRB_ExecSCSICmd  srbExec;

heventSRB = CreateEvent( NULL, TRUE, FALSE, NULL );
if( !heventSRB )
{
    // Couldn't get manual reset event, put error handling code here!
}

memset( &srbExec, 0, sizeof(SRB_ExecSCSICmd) );
srbExec.SRB_Cmd = SC_EXEC_SCSI_CMD;
srbExec.SRB_Flags = SRB_DIR_IN | SRB_EVENT_NOTIFY;
srbExec.SRB_Target = 5;
srbExec.SRB_BufLen = 32;
srbExec.SRB_BufPointer = byInquiry;
srbExec.SRB_SenseLen = SENSE_LEN;
srbExec.SRB_CDBLen = 6;
srbExec.SRB_PostProc = (LPVOID)heventSRB;
srbExec.CDBByte[0] = SCSI_INQUIRY;
srbExec.CDBByte[4] = 32;

ResetEvent( hevenSRB );
dwASPIStatus = SendASPI32Command( (LPSRB)&srbExec );
if( dwASPIStatus == SS_PENDING )
{
    WaitForSingleObject( heventSRB, INFINITE );
}

if( srbExec.SRB_Status != SS_COMP )
{
    // Error processing the SRB, put error handling code here.
}
```

SC_ABORT_SRB

The SendASPI32Command function with command code SC_ABORT_SRB is used to request that a pending SRB be aborted. It should be issued on any I/O request that has not completed if the application wishes to halt execution of that request. Success of the abort command is *never* assured.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_ABORT_SRB
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // Reserved, MUST = 0
    DWORD   SRB_Hdr_Rsvd;           // Reserved, MUST = 0
    LPSRB   SRB_ToAbort;            // Pointer to SRB to abort
}
SRB_Abort, *PSRB_Abort;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_ABORT_SRB (0x03).

SRB_Status (Output)

SC_ABORT_SRB is a synchronous SRB. On return, this field is the same as the SendASPI32Command return value and is set to SS_COMP, SS_INVALID_HA, or SS_INVALID_SRB. Remember that a return of SS_COMP does *not* indicate that the SRB to be aborted has been halted. Instead, it indicates that *an attempt* was made at aborting that SRB. If the SRB to be aborted completes with SS_ABORTED then there is positive indication that the original SC_ABORT_SRB worked.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for. Host adapter numbers are always assigned by the ASPI manager layer beginning with zero.

SRB_ToAbort (Input)

This field contains a pointer to the SRB which is to be aborted. The actual failure or success of the abort operation is indicated by the status eventually returned in this SRB.

Remarks

As stated above, the success of an SC_ABORT_SRB command is *never* guaranteed. As a matter of fact, the situations in which ASPI is capable of aborting an SRB already sent to the system are few and far between.

The original use for SC_ABORT_SRB was to terminate I/O which had timed out under ASPI for DOS and ASPI for Win16. The nature of SC_ABORT_SRB under Win32 greatly reduces its usefulness. It is recommended that the SC_GETSET_TIMEOUTS SRB be used to manage SRB timeouts in all new ASPI mo dules.

SC_RESET_DEV

The SendASPI32Command function with command code SC_RESET_DEV is used to send a SCSI Bus Device reset to the specified target.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_RESET_DEV
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // Reserved, MUST = 0
    DWORD   SRB_Hdr_Rsvd;           // Reserved, MUST = 0
    BYTE    SRB_Target;             // Target's SCSI ID
    BYTE    SRB_Lun;                // Target's LUN number
    BYTE    SRB_Rsvd1[12];          // Reserved, MUST = 0
    BYTE    SRB_HaStat;             // Host Adapter Status
    BYTE    SRB_TargStat;           // Target Status
    LPVOID  SRB_PostProc;           // Post routine
    BYTE    SRB_Rsvd2[36];          // Reserved, MUST = 0
}
SRB_BusDeviceReset, *PSRB_BusDeviceReset;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_RESET_DEV (0x04).

SRB_Status (Output)

SC_RESET_DEV is an asynchronous SRB. This field should not be examined until *after* the caller has waited for proper completion of the SRB (see "Waiting for Completion"). Once completed, this field may be set to a number of different values. The most common values are SS_COMP or SS_ERR. SS_COMP indicates successful completion while SS_ERR indicates the caller should examine the SRB_HaStat and SRB_TargStat fields for more information. See "ASPI for Win32 Error" for a complete description of possible error codes.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for. Host adapter numbers are always assigned by the SCSI manager layer beginning with zero.

SRB_Target (Input)

SCSI ID of target device.

SRB_Lun (Input)

Logical Unit Number (LUN) of target device. This field is ignored by ASPI for Win32, since SCSI BUS DEVICE RESET is done on a per-target basis only.

SRB_HaStat (Output)

Upon completion of the SCSI command, this field is set to the host adapter status. Do not examine this status byte if SRB_Status is set to SS_COMP. It is only to be considered valid if there is unsuccessful completion of the SRB.

Symbol	Value	Description
HASTAT_OK	0x00	Host adapter did not detect an error.
HASTAT_TIMEOUT	0x09	The time allocated for a bus transaction ran out.
HASTAT_COMMAND_TIMEOUT	0x0B	SRB expired while waiting to be processed.
HASTAT_MESSAGE_REJECT	0x0D	MESSAGE REJECT received while processing SRB.
HASTAT_BUS_RESET	0x0E	A bus reset was detected.
HASTAT_PARITY_ERROR	0x0F	A parity error was detected.
HASTAT_REQUEST_SENSE_FAILED	0x10	The adapter failed in issuing a Request Sense after a check condition was reported by the target device.
HASTAT_SEL_TO	0x11	Selection of target timed out.
HASTAT_DO_DU	0x12	Data overrun/underrun.
HASTAT_BUS_FREE	0x13	Unexpected Bus Free.
HASTAT_PHASE_ERR	0x14	Target Bus phase sequence failure.

SRB_TargStat (Output)

Upon completion of the SCSI command, this field is set to the final SCSI target status. Do not examine this status byte if SRB_Status is set to SS_COMP. It is only to be considered valid if there is unsuccessful completion of the SRB. Note that the table below only covers the most common result codes. Check the SCSI specification for more information on these and other status byte codes.

Symbol	Value	Description
STATUS_GOOD	0x00	No target status.
STATUS_CHKCOND	0x02	Check status (sense data is in SenseArea).
STATUS_BUSY	0x08	Specified Target/LUN is busy.
STATUS_RESCONF	0x18	Reservation conflict.

SRB_PostProc (Input)

If posting is enabled (SRB_POSTING) this field contains a pointer to a function. The ASPI manager calls this function upon completion of the SRB. If event notification is enabled (SRB_EVENT_NOTIFY) this field contains a handle to an event. The ASPI manager signals this event upon completion of the SRB. See "Waiting for Completion" for more information.

Remarks

The Windows (98, ME, NT, 2000, XP (32-bit)) operating systems do not handle BUS DEVICE RESET properly at the current time. For this reason, SC_RESET_DEV calls are not guaranteed to function properly. The command is present mainly to keep older code ported from Win16 from failing.

SC_GET_DISK_INFO

The SendASPI32Command function with command code SC_GET_DISK_INFO is used to obtain information about a disk type SCSI device. The information returned includes BIOS Int 13h control and accessibility of the device, the drive's Int 13h physical drive number, and the geometry used by the Int 13h services for the drive.

Note: This command is not valid for Windows NT/2000/XP (32-bit), which does not use the Int 13 interface.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_GET_DISK_INFO
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // Reserved
    DWORD   SRB_Hdr_Rsvd;           // Reserved
    BYTE    SRB_Target;             // Target's SCSI ID
    BYTE    SRB_Lun;                // Target's LUN number
    BYTE    SRB_DriveFlags;         // Driver flags
    BYTE    SRB_Int13HDriveInfo;    // Host Adapter Status
    BYTE    SRB_Heads;              // Preferred number of heads translation
    BYTE    SRB_Sectors;            // Preferred number of sectors translation
    BYTE    SRB_Rsvd1[10];          // Reserved
}
SRB_GetDiskInfo, *PSRB_GetDiskInfo;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_GET_DISK_INFO (0x06).

SRB_Status (Output)

SC_GET_DISK_INFO is a synchronous SRB. On return, this field is the same as the SendASPI32Command return value and is set to SS_COMP, SS_INVALID_HA, or SS_NO_DEVICE, or SS_INVALID_SRB.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for. Host adapter numbers are always assigned by the ASPI manager layer beginning with zero.

SRB_Target (Input)

SCSI ID of target device.

SRB_Lun (Input)

Logical Unit Number (LUN) of target device.

SRB_DriveFlags (Output)

Upon completion of the SCSI command this field is set as follows:

Symbol	Value	Description
DISK_NOT_INT13	0x00	Device is not controlled by Int 13h services
DISK_INT13_AND_DOS	0x01	Device is under Int 13h control and is claimed by DOS
DISK_INT13	0x02	Device is under Int 13h control but not claimed by DOS

SRB_Int13DriveInfo (Output)

Upon completion of the SCSI command, the ASPI manager sets this field with the physical drive number that Int 13h services assigned to the device. The valid drive numbers are 0x00 to 0xFF. This field is only valid if `SRB_DriveFlags` is set to `DISK_INT13_AND_DOS` or `DISK_INT13`.

SRB_Heads (Output)

Upon completion of the SCSI command, the ASPI manager sets this field to the number of heads the Int 13h services is using for this device's geometry. The valid drive numbers are 0x00 to 0xFF. This field is only valid if `SRB_DriveFlags` is set to `DISK_INT13_AND_DOS` or `DISK_INT13`.

SRB_Sectors (Output)

Upon completion of the SCSI command, the ASPI manager sets this field to the number of sectors the Int 13h services is using for this device's geometry. The valid drive numbers are 0x00 to 0xFF. This field is only valid if `SRB_DriveFlags` is set to `DISK_INT13_AND_DOS` or `DISK_INT13`.

Example

This example obtains disk information from device LUN 0, SCSI ID 2, attached to host adapter 0.

```
SRB_GetDiskInfo  srbGetDiskInfo;

memset( &srbGetDiskInfo, 0, sizeof(SRB_GetDiskInfo) );
srbGetDiskInfo.SRB_Header.SRB_Cmd = SC_GET_DISK_INFO;
srbGetDiskInfo.SRB_Target = 2;

SendASPI32Command( (LPSRB)&srbGetDiskInfo );
if( srbGetDiskInfo.SRB_Status != SS_COMP )
{
    // Error handling GetDiskInfo SRB.  Error handling code goes here!
}
```

SC_RESCAN_SCSI_BUS

The SendASPI32Command function with command code SC_RESCAN_SCSI_BUS is used to rescan the SCSI bus specified by the host adapter number in the SRB. It will instruct the I/O subsystem to rescan the SCSI bus and update both the system device map and the ASPI manager device tables.

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_RESCAN_SCSI_BUS
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // Reserved, MUST = 0
    DWORD   SRB_Hdr_Rsvd;           // Reserved, MUST = 0
}
SRB_RescanPort, *PSRB_RescanPort;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_RESCAN_SCSI_BUS (0x07).

SRB_Status (Output)

SC_RESCAN_SCSI_BUS is a synchronous SRB. On return, this field is the same as the SendASPI32Command return value and is set to SS_COMP, or SS_INVALID_HA.

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for. Host adapter numbers are always assigned by the ASPI manager layer beginning with zero.

Remarks

Under Windows NT/2000/XP (32-bit), the I/O subsystem does not rescan devices/IDs it already knows about. The impact of this is that it will detect new devices but will not detect removal of devices or exchanging of devices.

Under Windows 98/ME, there can be a substantial delay between the time a rescan is initiated with this command and the time at which new devices are added or old devices are removed from the device map. The best way to deal with this is to rely on the Plug and Play messages in conjunction with TranslateASPI32Address, or to simply perform your own refresh five or ten seconds after the rescan command is issued.

There is no way to force a rescan of the entire system. It is up to the operating system to detect the arrival of new host adapters (for example, PCMCIA) through Plug and Play, if it is available.

Example

The following example forces a rescan of the SCSI bus attached to host adapter #0:

```
SRB_RescanPort  srbRescanPort;
memset( &srbRescanPort, 0, sizeof(SRB_RescanPort) );
srbRescanPort.SRB_Cmd = SC_RESCAN_SCSI_BUS;

SendASPI32Command( (LPSRB)&srbRescanPort );
if( srbRescanPort.SRB_Status != SS_COMP )
{
    // Error issuing port rescan.  Error handling code goes here.
}
```

SC_GETSET_TIMEOUTS

The SendASPI32Command function with command code SC_GETSET_TIMEOUTS enables you to set target specific timeouts in 1/2 second increments. Once set, a timeout applies to all SCSI commands sent through the SC_EXEC_SCSI_CMD command. Timeouts are process specific, so two different applications may set different timeouts for the same target. The SRB_HaId, SRB_Target, and SRB_Lun fields may be set to a wildcard value to ease the setting of timeouts on multiple targets. Note that by default, all target timeouts are set to 30 hours (the maximum allowed).

```
typedef struct
{
    BYTE    SRB_Cmd;                // ASPI command code = SC_GETSET_TIMEOUTS
    BYTE    SRB_Status;             // ASPI command status byte
    BYTE    SRB_HaId;               // ASPI host adapter number
    BYTE    SRB_Flags;              // ASPI request flags
    DWORD   SRB_Hdr_Rsvd;           // Reserved
    BYTE    SRB_Target;             // Target's SCSI ID
    BYTE    SRB_Lun;                // Target's LUN number
    DWORD   SRB_Timeout;            // Timeout in half seconds
}
SRB_GetSetTimeouts, *PSRB_GetSetTimeouts;
```

SRB Fields

SRB_Cmd (Input)

This field must contain SC_GETSET_TIMEOUTS (0x08).

SRB_Status (Output)

SC_GETSET_TIMEOUTS is a synchronous SRB. On return, this field is the same as the SendASPI32Command return value and is set to SS_COMP, SS_INVALID_HA, SS_NO_DEVICE, or SS_INVALID_SRB (bad flags, invalid timeout, etc.).

SRB_HaId (Input)

This field specifies which installed host adapter the request is intended for If SRB_DIR_OUT is set in SRB_Flags then this value may be a wildcard (0xFF) indicating that the SRB_Target/SRB_Lun combination on ALL host adapters should get new a timeout.

SRB_Flags (Input)

May be set to one and only one of the following two constants:

Symbol	Value	Description
SRB_DIR_IN	0x08	SRB is being used to retrieve current timeout setting. Wildcards are not allowed in the ASPI address fields
SRB_DIR_OUT	0x10	SRB is being used to change the current timeout setting. Wildcards are valid in the ASPI address fields.

SRB_Target (Input)

This field indicates the SCSI ID of the target device. If SRB_DIR_OUT is set in SRB_Flags then this value may be a wildcard (0xFF) indicating that ALL SCSI IDs of the passed SRB_HaId/SRB_Lun combination should get a new timeout.

SRB_Lun (Input)

This field indicates the Logical Unit Number (LUN) of the device. If SRB_DIR_OUT is set in SRB_Flags then this value may be a wildcard (0xFF) indicating that ALL LUNs of the passed SRB_HaId/SRB_Target combination should get a new timeout.

SRB_Timeout (Input)

Target's timout in half seconds. If SRB_DIR_OUT then this value holds the new timeout for the specified target(s). If SRB_DIR_IN then the value is set by ASPI to the current timeout for the specified target. The timeout can be from 0-216000 (30 hours) with 0 being an easier way of saying "max timeout" (again, 30 hours).

Remarks

Once a timeout is set for a target, that timeout will be used on all SRBs passed to SendASPI32Command with SC_EXEC_SCSI_CMD. If one of these SRBs actually times out, then the SCSI bus will be reset (this is NOT a bus device reset, but a full SCSI bus reset). This causes all of the SRBs executing on the bus to be cancelled, and the miniport will set error codes in the SRBs as appropriate. It is up to the code which originally submitted these SRBs to retry the commands as necessary (for example, if an ASPI request times out and the bus is reset, a file system command to another target could be cancelled, and it is up to the file system to retry the command). In addition, the result placed in the SRB which times out depends on the error codes which the miniport places in the SRB. In the case of Adaptec controllers, the result code is SS_ABORT. In other miniports, the result may be SS_ERR with a host adapter status set to HASTAT_TIMEOUT or HASTAT_COMMAND_TIMEOUT, or it may be some new error result not yet encountered. Suffice it to say that the SRB which times out should return with an error, and it is up to the higher level applications to perform retries of the SRB and any other SRB which may have been affected by the associated bus reset.

When using event notification with timeouts, it is important to remember that the HEVENT used in the SRB_PostProc field has an ENTIRELY SEPERATE timeout associated with it. In other words, the timeout associated with an event is seperate from the timeout associated with an SRB. If you set a timeout on an SRB and then set an infinite timeout in WaitForSingleObject on the SRB event, then the SRB will STILL TIMEOUT and signal completion of the SRB. Conversely, if you set a 30 hour timeout on the SRB and a 5 second timeout on the event, the event will always go signaled before the SRB completes, and no cleanup of the SRB on the bus will take place.

Examples

The first example illustrates how wildcards work with set timeout. The main point here is that the wildcards are specific. In other words, setting the HaId to 0xFF does not make `SRB_Target/SRB_Lun` "don't cares".

HA	ID	LN	Device Affected
00	01	FF	All of target 1's luns on host adapter 0.
FF	00	FF	All luns on targets with ID 0 on any host adapter.
FF	FF	00	Lun 0 of all targets on any host adapter.
FF	FF	FF	All targets on any host adapter with any lun number (everything).

Next is an example in which all LUNs on target 5, host adapter 0 are set to 10 seconds:

```
SRB_GetSetTimeouts  srbGetSetTimeouts;

memset( &srbGetSetTimeouts, 0, sizeof(SRB_GetSetTimeouts) );
srbGetSetTimeouts.SRB_Cmd = SC_GETSET_TIMEOUTS
srbGetSetTimeouts.SRB_Flags = SRB_DIR_OUT;
srbGetSetTimeouts.SRB_Target = 0x05;
srbGetSetTimeouts.SRB_Lun = 0xFF;
srbGetSetTimeouts.SRB_Timeout = 10*2;

SendASPI32Command( (LPSRB)&srbGetSetTimeouts );
if( srbGetSetTimeouts.SRB_Status != SS_COMP )
{
    // Error setting timeouts.  Put error handling code here.
}
```

GetASPI32Buffer

`GetASPI32Buffer` allocates blocks of memory (up to 512KB) which are "safe" for use in ASPI modules. Under normal circumstances memory buffers from the stack or allocated with VirtualAlloc will be too physically fragmented to allow a transfer greater than 64KB on bus-mastering host adapters. For those rare instances where a large transfer is required, `GetASPI32Buffer` allows a buffer to be allocated which will pass all operating system requirements for physical continuity.

```
BOOL GetASPI32Buffer( PASPI32BUFF pab );
```

Parameters

pab

Pointer to a filled out ASPI32BUFF structure.

```
typedef struct
{
    LPBYTE  AB_BufPointer;          // Pointer to the ASPI allocated buffer
    DWORD   AB_BufLen;              // Length in bytes of the buffer
    DWORD   AB_ZeroFill;            // Flag set to 1 if buffer should be
zeroed
    DWORD   AB_Reserved;            // Reserved, MUST = 0
}
ASPI32BUFF, *PASPI32BUFF;
```

AB_BufPointer (Output)

After a successful call (return value TRUE) this field contains the address of the large transfer buffer which has been allocated for the application.

AB_BufLen (Input)

Set to the size, in bytes, desired for the transfer buffer. This must be less than or equal to 512KB and should be greater than 64KB (although there are no requirements on the low end).

AB_ZeroFill (Input)

Set this flag to 1 if ASPI should clear the transfer buffer after allocation but before returning to the caller. Leave the flag set to 0 if the memory can remain uninitialized.

Remarks

If you have experienced a failure to allocate a buffer, you may need to do one or all of the following:

1. Increase the non-paged pool size on your PC (allocate more RAM to non-paged pool),
2. Physically add more RAM to your PC.
3. Decrease the buffer size requested.

The first thing to do is to increase the non-paged pool size.

1) How to increase the non-paged pool size?

i. Run regedt32.exe (click Start ➢Run and type regedt32),

ii. Edit `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ ↵ SessionManager\MemoryManagement\NonPagedPoolSize`

This parameter is a REG_DWORD (32 bits).

NOTE: If it is set to 0x00, the default minimum portion of RAM is allocated to non-paged pool.

iii. Set this parameter to 0xffffffff (the maximum portion of RAM is allocated to non-paged pool.) Reboot the PC and try the application again.

NOTE: If GetASPI32Buffer still fails; you may not have enough physical RAM in the PC, and you may need to physically install additional RAM.

If GetASPI32Buffer now works, then gradually decrease the non-paged pool size (by specifying the amount of RAM allocated to non-paged pool) to optimize.

Tips

Set the NonPagedPoolSize to 0x00100000 for 1 MegaByte; 0x00200000 for 2 MegaBytes; etc.

See the Microsoft Developer Network (MSDN) article **Q126402** (www.microsoft.com) for more information regarding setting the non-paged pool size.

Return Values

This function returns TRUE if it successfully allocates a large transfer buffer, and FALSE otherwise. The caller should assume that this call can fail, and should allow the code to work with smaller transfer buffers allocated from VirtualAlloc (if at all possible).

Example

The following example allocates a 128KB buffer for use with ASPI.

```
ASPI32BUFF  ab;

memset( &ab, 0, sizeof(ASPI32BUFF) );
ab.AB_BufLen = 131072lu;
ab.AB_ZeroFill = 1;

if( !GetASPI32Buffer( &ab ) )
{
    // Unable to allocate buffer.  Error handling code goes here!
}
```

FreeASPI32Buffer

`FreeASPI32Buffer` releases memory previously allocated by a successful call go **`GetASPI32Buffer`**.

```
BOOL FreeASPI32Buffer( PASPI32BUFF pab );
```

Parameters

pab

Pointer to a filled out ASPI32BUFF structure.

```
typedef struct
{
    LPBYTE  AB_BufPointer;          // Pointer to the ASPI allocated buffer
    DWORD   AB_BufLen;              // Length in bytes of the buffer
    DWORD   AB_ZeroFill;            // Reserved, MUST = 0
    DWORD   AB_Reserved;            // Reserved, MUST = 0
}
ASPI32BUFF, *PASPI32BUFF;
```

AB_BufPointer (Input)

Pointer to the buffer previously returned from a successful call to `GetASPI32Buffer`. The address must match exactly for the free to occur.

AB_BufLen (Input)

Set to the original size, in bytes, of the buffer allocated by a call to `GetASPI32Buffer`. The size must match exactly for the free to occur.

Return Values

This function returns TRUE if the memory allocated to the buffer has been released. FALSE is returned if there is an error freeing the memory or if the passed in AB_BufPointer/AB_BufLen fields don't match a those of a previously allocated buffer.

TranslateASPI32Address

`TranslateASPI32Address` provides translation between Windows 98 DEVNODEs and ASPI HA/ID/LUN triples (or vice versa). Because DEVNODEs are associated with WM_DEVICECHANGE messages, it is possible to use this function to associate ASPI target addresses with Plug and Play events.

```
BOOL TranslateASPI32Address( PDWORD pdwPath, PDWORD pdwDEVNODE );
```

Parameters

pdwPath (Input/Output)

Pointer to a ASPI address "path." The path is simply a packed version of an ASPI address triple. Every target address in ASPI consists of a host adapter identifier, a SCSI ID, and a SCSI LUN. Each of these values consists of a BYTE, so an ASPI address "path" is a DWORD encoded as 0x00HHIILL where HH is the host adapter identifier, II is the SCSI ID, and LL is the SCSI LUN. Note that if II and LL are both 0xFF then the path represents a host adapter. This is necessary because host adapters have their own DEVNODEs in the Plug and Play subsystem.

pdwDEVNODE (Input/Output)

Pointer to a DWORD which contains a Windows 98 DEVNODE ID. This parameter controls the direction of translation. If the DWORD contains a 0 (note that this does *not* mean that pdwDEVNODE is NULL) then translation is from the ASPI triple to the DEVNODE. If the DEVNODE is non-zero then translation is from the DEVNODE to an ASPI triple.

Return Values

TRUE if there is a successful translation. FALSE is returned if the parameters are invalid or if there is no translation between ASPI path and Windows 98 DEVNODE.

Remarks

In order for this scheme to work properly, applications should pay attention to WM_DEVICECHANGE messages which utilize DBT_DEVTYP_DEVNODE device change data. The device change data type can be detected by checking the dcbh_devicetype field in the DEV_BROADCAST_HEADER associated with device change events. Review the Plug and Play documentation in Win32 for more information.

Example

The function below checks broadcast data from a WM_DEVICECHANGE message to see if the device change message is related to an ASPI target (but not host adapter).

```
BOOL CheckForASPITargetBroadcast( PDEV_BROADCAST_HDR pHeader )
{
    BOOL                    bStatus;
    DWORD                   dwTargetPath;
    DWORD                   dwDEVNODE;
    PDEV_BROADCAST_DEVNODE  pDevnodeData

    if( pHeader->dbch_devicetype != DBT_DEVTYP_DEVNODE )
    {
        return FALSE;
    }

    pDevnodeData = (PDEV_BROADCAST_DEVNODE)pHeader;
    dwDEVNODE = pDevnodeData->dbcd_devnode;

    bStatus = TranslateASPI32Address( &dwTargetPath, &dwDEVNODE );
    if( !bStatus || ((dwTargetPath & 0xFFFFlu) == 0xFFFFlu) )
    {
        return FALSE;
    }

    return TRUE;
}
```

Waiting for Completion

There are two types of SRBs sent to `SendASPI32Command`: synchronous and asynchronous. Synchronous SRBs are always complete when the call to `SendASPI32Command` returns. Asynchronous SRBs, however, may or may not be complete upon return from the `SendASPI32Command` call.

When called with an asynchronous SRB, the status return from `SendASPI32Command` should be checked for a value of `SS_PENDING`. If the status code *is not* `SS_PENDING` then the SRB is complete and it is safe to look at its status codes, etc. If `SS_PENDING` *is* returned then the SRB is still under the control of ASPI, and the caller needs to wait for the SRB to complete before doing anything else with that SRB.

There are three ways of being notified that an asynchronous SRB has completed. The first and recommended method uses event notification. The second method uses posting (a callback), and the third method uses polling. All three completion methods are illustrated below using a simple INQUIRY command to host adapter #0, SCSI ID #5, LUN #0.

Event Notification

Event notification is an ideal mechanism for notifying ASPI clients of the completion of an ASPI request. ASPI clients may efficiently block on this event until completion. Upon completion of a request, the ASPI for Win32 manager will set the event to the signaled state. The ASPI client is responsible for making sure that the event is a manual-reset style event which is not in a signaled state when an ASPI request is submitted.

```
BYTE             byInquiry[32];
DWORD            dwASPIStatus;
HANDLE           heventSRB;
SRB_ExecSCSICmd  srbExec;

heventSRB = CreateEvent( NULL, TRUE, FALSE, NULL );
if( !heventSRB )
{
    // Couldn't get manual reset event, put error handling code here!
}

memset( &srbExec, 0, sizeof(SRB_ExecSCSICmd) );
srbExec.SRB_Cmd = SC_EXEC_SCSI_CMD;
srbExec.SRB_Flags = SRB_DIR_IN | SRB_EVENT_NOTIFY;
srbExec.SRB_Target = 5;
srbExec.SRB_BufLen = 32;
srbExec.SRB_BufPointer = byInquiry;
srbExec.SRB_SenseLen = SENSE_LEN;
srbExec.SRB_CDBLen = 6;
srbExec.SRB_PostProc = (LPVOID)heventSRB;
srbExec.CDBByte[0] = SCSI_INQUIRY;
srbExec.CDBByte[4] = 32;

ResetEvent( hevenSRB );
dwASPIStatus = SendASPI32Command( (LPSRB)&srbExec );
if( dwASPIStatus == SS_PENDING )
{
    WaitForSingleObject( heventSRB, INFINITE );
}

if( srbExec.SRB_Status != SS_COMP )
{
    // Error processing the SRB, put error handling code here.
}
```

Posting

Posting (or callbacks) may be used to receive notification that a SCSI request has completed. When posting is used, ASPI for Win32 posts completion by passing control to a callback function. If you send an ASPI request with posting enabled, the callback procedure will always be called. The post or callback routine is called as a standard C function. The caller (in this case, the ASPI manager) cleans up the stack. The prototype for the callback is below in the sample.

```
BYTE             byInquiry[32];
SRB_ExecSCSICmd  srbExec;

memset( &srbExec, 0, sizeof(SRB_ExecSCSICmd) );
srbExec.SRB_Cmd = SC_EXEC_SCSI_CMD;
srbExec.SRB_Flags = SRB_DIR_IN | SRB_POSTING;
srbExec.SRB_Target = 5;
srbExec.SRB_BufLen = 32;
srbExec.SRB_BufPointer = byInquiry;
srbExec.SRB_SenseLen = SENSE_LEN;
srbExec.SRB_CDBLen = 6;
srbExec.SRB_PostProc = ASPIInquiryCallback;
srbExec.CDBByte[0] = SCSI_INQUIRY;
srbExec.CDBByte[4] = 32;

SendASPI32Command( (LPSRB)&srbExec );

. . .

/**
*** The code above is a separate thread of execution from
*** the code below which handles the inquiry callback.  Note that
*** the callback usually signals the main thread of execution that
*** the an SRB it submitted has completed.  In this case we aren't
*** doing anything but checking for errors.
**/

VOID ASPIInquiryCallback( SRB_ExecSCSICmd psrbExec )
{
    if( psrbExec->SRB_Status != SS_COMP )
    {
        // Error processing the SRB, put error handling code here.
    }
}
```

Polling

Polling is another method of determining SCSI request completion. This method is not recommended because of the large number of CPU cycles consumed while checking the status byte. After the command is sent and ASPI for Win32 returns control back to the calling application, you can then poll the status byte waiting for the command to complete. Note that this completion method is the only one to "break" the rule of not touching an SRBs data until after completion. With polling you must look at the `SRB_Status` byte in order to tell when the SRB is complete. You are still prohibited from accessing any other fields of the SRB.

```
BYTE             byInquiry[32];
SRB_ExecSCSICmd  srbExec;

memset( &srbExec, 0, sizeof(SRB_ExecSCSICmd) );
srbExec.SRB_Cmd = SC_EXEC_SCSI_CMD;
srbExec.SRB_Flags = SRB_DIR_IN;
srbExec.SRB_Target = 5;
srbExec.SRB_BufLen = 32;
srbExec.SRB_BufPointer = byInquiry;
srbExec.SRB_SenseLen = SENSE_LEN;
srbExec.SRB_CDBLen = 6;
srbExec.CDBByte[0] = SCSI_INQUIRY;
srbExec.CDBByte[4] = 32;

SendASPI32Command( (LPSRB)&srbExec );
while( srbExec.SRB_Status == SS_PENDING );

if( srbExec.SRB_Status != SS_COMP )
{
    // Error processing the SRB, put error handling code here.
}
```

ASPI for Win32 Errors

Each of these errors can be returned by ASPI for Win32 on either Windows 98/ME or Windows NT/2000/XP (32-bit). The ASPI header files that were included with the ASPI SDK may have codes defined which cannot be returned by an actual ASPI implementation. These codes are in the header file to serve as placeholders for other ASPI managers. They are not documented in this table.

Symbol	Value	Description
SS_PENDING	0x00	Returned from SendASPI32Command on SC_EXEC_SCSI_CMD and SC_RESET_DEV SRBs to indicate that the command is in progress. Use polling, posting, or event-notification (preferred) to wait for completion.
SS_COMP	0x01	Either returned from SendASPI32Command, or set in the SRB_Status field of the SRB header. This value indicates successful completion of an SRB.
SS_ABORTED	0x02	The current SRB was aborted either by the operating system directly (for example, a third party does a hard reset of the SCSI bus) or through a SC_ABORT_SRB.
SS_ERR	0x04	Returned on SC_EXEC_SCSI_CMD calls if there is a host adapter, SCSI bus, or SCSI target error. It indicates that the caller should examine SRB_TargStat and SRB_HaStat for additional information.
SS_INVALID_CMD	0x80	The SRB_Cmd passed in an SRB is invalid.
SS_INVALID_HA	0x81	The SRB_HaId passed in an SRB is invalid. Call GetASPI32SupportInfo to determine the valid range of host adapters identifiers.
SS_NO_DEVICE	0x82	Returned from calls to SendASPI32Command, or set in the SRB_Status field of the SRB header. This value indicates that there is no target present at the SCSI address indicated in the SRB. Note that this is not a selection timeout. The operating system keeps a table of known devices and does not permit commands to "non-existent" devices. This code could be returned if an operating system rescan of the SCSI bus is required to detect a newly powered on device.
SS_INVALID_SRB	0xE0	An SRB sent to ASPI had a valid address and a valid command byte, but it was somehow faulty in another way. The exact cause of the failure is dependent on the SRB type. For example, an SC_EXEC_SCSI_CMD SRB may fail if an invalid flag is set in the SRB_Flags word, if a buffer length is specified but there is a NULL buffer pointer, or if ASPI detects an SRB has been reused. In any case, the code creating the SRB is faulty and needs to be analyzed.
SS_BUFFER_ALIGN	0xE1	SRB data buffers must meet alignment requirements as returned by SC_HA_INQUIRY SRBs. If a transfer buffer does not meet those requirements, this error is returned.

SS_ILLEGAL_MODE	0xE2	An attempt was made to start ASPI for Win32 from Win32s. ASPI for Win32 is a pure Win32 component and cannot be run under the Windows 3.1x Win32 subsystem.
SS_NO_ASPI	0xE3	WNASPI32.DLL is present on the system, but it could not find it's helper driver. Under Windows 98/ME APIX.VXD is the helper driver, and under Windows NT/2000/XP (32-bit) ASPI32.SYS is the helper driver. Either the ASPI installation is invalid, or there are resource conflicts preventing ASPI from starting.
SS_FAILED_INIT	0xE4	A general internal failure has occurred within ASPI. This can occur during initialization or at run-time. This error should only occur if basic Windows operating services begin to fail, in which case the whole system is unstable.
SS_ASPI_IS_BUSY	0xE5	Returned either from `SendASPI32Command`, or set in the `SRB_Status` field of the SRB header. This code indicates that ASPI did not have enough resources to complete the requested SRB at the present time. This is different from `SS_INSUFFICIENT_RESOURCES` in that it is usually a temporal condition, and the failed SRB may be retried at a later time.
SS_BUFFER_TOO_BIG	0xE6	Returned in the `SRB_Status` field of a failing SRB. The code indicates that the buffer associated with the SRB did not meet internal operating system constraints for a valid transfer buffer. For example, a buffer >64KB on a bus-mastering controller will usually fail with this error because it is not physically contiguous enough to be described by a scatter/gather list.
SS_MISMATCHED_COMPONENTS	0xE7	ASPI for Win32 consists of three components under Windows 98: WNASPI32.DLL, APIX.VXD, and ASPIENUM.VXD. It consists of two components under Windows NT: WNASPI32.DLL, ASPI32.SYS. Each of these components has a version number, and all the version numbers on a particular platform must agree for ASPI to function. This error will only occur if the installation has been corrupted, and components with different version numbers have been installed on the system. The only fix for this is to remove all of the ASPI components for that operating system, and then reinstall a full, consistent set of ASPI drivers.
SS_NO_ADAPTERS	0xE8	Returned from `GetASPI32SupportInfo` if ASPI has initialized successfully, but there are no host adapters on the system. It is still possible that an adapter may become active through Plug and Play, so a lack of manageable host adapter is no longer considered an error as it was in previous versions of ASPI.
SS_INSUFFICIENT_RESOURCES	0xE9	The error occurs only during initialization if there are not enough system resources (memory, event handles, critical sections, etc.) to fully initialize ASPI. If this error occurs it is likely that the system is critically low on memory.