Advanced SCSI Programming Interface
Frequently Asked Questions

ASPI for Microsoft®
Windows® 95 and Windows NT®
February 2, 1998

adaptec®

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.

General Administrative Questions

Q: What is ASPI for Windows 95 and Windows NT?

A: The Advanced SCSI Programming Interface (ASPI) for Windows 95 and Windows NT allows a variety of different applications, DLLs, or drivers to send commands to SCSI devices. ASPI is supported on a number of different operating systems including DOS, Windows 95, Windows NT, NetWare, and OS/2. There are also a number of different application programming interfaces (APIs) including ASPI for DOS, ASPI for Win16, ASPI for Win32, ASPI for NetWare, and ASPI for OS/2.

This document focuses on frequently asked questions about ASPI for Windows 95 and Windows NT, regardless of the API. Specifically, ASPI for Windows 95 supports the ASPI for DOS, ASPI for Win16, and ASPI for Win32 APIs, while Windows NT supports the ASPI for Win16 and ASPI for Win32 APIs. There is also a version of ASPI for Win32 for the Alpha under Windows NT, but it is not specifically covered by this document.

Unless otherwise noted in this document, the term ASPI shall mean ASPI for Windows 95 and Windows NT.

Q: What is the most current version of ASPI?

A: As of the date on this document, the most current ASPI for Windows 95 and Windows NT is version 4.57, build 1008. The latest version of ASPI (***not*** suitable for redistribution) can be downloaded from http://www.adaptec.com/support/files/upgrades.html#ezscsi, and the filename is ASPI32.EXE. Also available from this site is a small utility named ASPICHK.EXE that lists all the version numbers for the ASPI layer on a particular machine.

Q: Where can I get the latest ASPI specification and sample disk?

A: The latest ASPI for Win32 specification and sample disk may be obtained by sending an e-mail request to mailto:acap@corp.adaptec.com. The request should include your full name, company name, and address as well as a brief description of why you need the latest specification.

The latest specification for all other ASPI APIs can be obtained by purchasing the ASPI Software Development Kit. The latest is version 2.3, and it can be obtained directly from Adaptec. Call the Adaptec Software Hotline at +1.800.442.7274 (or +1.408.957.7274) and ask for the ASPI SDK, part number 521100D.

Q: Where can I get the latest SCSI specifications?

A: Neither the ASPI SDK nor the specifications include a tutorial on SCSI programming. They assume a solid knowledge of the SCSI specification, along with detailed knowledge of the device you are attempting to program. Here are sources for the latest SCSI specifications:

SCSI-1 and CCS:	American National Standards Institute 11 West 42nd Street, NY, NY 10036 Phone: (212) 642-4900 Fax: (212) 302-1286

SCSI-2: Global Engineering Documents
7730 Carondelet Ave, Ste 407
Clayton, MO 63105
Phone: (800) 854-7179
Phone: (314) 726-0444
Fax: (314) 726-6418

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.

Q: How does my company license ASPI for redistribution?

A: Although ASPI is delivered with Windows 95, it is not delivered with Windows NT. Furthermore, the version delivered with Windows 95 is older and does not have all of the latest features. If you want the latest version of ASPI to be delivered with your application for either Windows 95 or Windows NT, then you must license ASPI for redistribution from Adaptec.

Licensing ASPI is fairly easy, and assuming you qualify, it is free. First, you should contact Diane McGee in the Adaptec Legal Department. She can be reached via e-mail at mailto:dmcgee@corp.adaptec.com, and she will ask you to fill out an ASPI Distribution License Request form. The form requests your company name, complete address, phone, fax, name of contract representative, and a brief description of your development project. The form is either mailed or faxed to Diane for processing.

Following receipt of the form, it will take approximately 5 business days for approval, followed by up to 10 business days for the generation of the contracts and internal paperwork. Basically, you should plan on requesting the license roughly four to eight weeks before your project is to be completed.

Once licensed you will receive a single floppy which contains the ASPI installer for both Intel and Alpha (which ***must*** be used—it is no longer optional), the documentation for the installer, the latest ASPI for Win32 documentation, and the latest ASPI for Win32 sample. All of the documentation is distributed electronically as Adobe® Acrobat® files.

Please be aware that the ASPI layer is copyright Adaptec and should not be put on any on-line forums for free download. If your customers ever need an interim release of ASPI which you are not licensed to distribute, then you may direct them to the Adaptec web site to download the latest ASPI layer.

Q: How do I get technical support for ASPI development issues?

A: For the Intel platform, Adaptec provides technical support for the ASPI for Win32 layer. We do not explain Win32 programming, SCSI command sets, SCSI bus phases, etc. If you feel the ASPI layer is not performing properly, or if you have a question not answered in our specifications, please e-mail Daniel Polfer at mailto:polfer@moxietech.com. You should also include the output from ASPICHK.EXE if you are having any technical difficulties.

General Development Questions

Q: Which application programming interfaces does ASPI support?

A: ASPI for Windows 95 supports ASPI for DOS, ASPI for Win16, and ASPI for Win32.
ASPI for Windows NT (Intel) supports ASPI for Win16, and ASPI for Win32.
ASPI for Windows NT (Alpha) supports ASPI for Win32 only.

Q: Which SCSI features does ASPI support?

A: ASPI supports a fairly straightforward subset of the SCSI-2 protocol. In general, it is easier to state what ASPI does not support: tagged command queuing, linking, message phase control, disconnect/reconnect control, narrow/wide negotiation, synchronous negotiation. ASPI does support synchronous, wide, and disconnected transfers, it just does not allow you to control the setup of these SCSI features. Under Windows 95 and Windows NT, if the target, host adapter, BIOS, miniport, and user options agree, then ASPI will support the different features.

As an example of this, assume there is a SCSI target that supports synchronous transfers, but after checking your commands with a bus analyzer, you know that they are not being used. ASPI is ***not*** responsible for this behavior. Assuming the host adapter supports it, synchronous negotiation has either been turned off in the adapter BIOS, by the SCSIPORT because of an error, or even by the user through the Device Manager. Again, ASPI is ***not*** responsible, and you will not receive ASPI technical support for such an issue.

Q: Do I have to use data transfer flags in my SRBs?

A: Yes. The older real mode drivers under MSDOS would allow the use of SRB_DIR_SCSI to indicate that the direction of transfer is unknown. This flag still exists, but under Windows 95 and Windows NT it can cause strange behavior on certain miniport drivers (usually PIO based). For this reason it is extremely important that either SRB_DIR_IN or SRB_DIR_OUT be used on every command which transfers data. If there is no data transfer, do not use any data direction flags at all. The use of data direction flags is true under all API implementations for Windows 95 and Windows NT.

Q: How do I properly wait for SRB completion?

A: The proper way to wait for SRB completion depends on the completion method. For polling, simply check the SRB_Status code periodically for a value other than SS_PENDING. Once the value changes, the SRB is complete, and the SRB and data memories are safe for reuse or disposal.

For posted requests in ASPI for Win16 first check the ***return code*** from SendASPI32Command for SS_PENDING. Do ***not*** check the SRB_Status code. If you have a pending return code, then you know that at some point in the future your callback routine will be called with the completed routine, at which point it is safe to check status codes, reuse the SRB, etc. For ASPI for Win32 you will ***always*** receive a callback to your post routine regardless of how the SRB completes. In other words, if you have an immediate completion in ASPI for Win16 then the return code will be other than SS_PENDING and there will be no callback. In ASPI for Win32 the return code will be other than SS_PENDING and there will be a callback for that SRB.

Finally, for event notification in ASPI for Win32, you should first check the ***return code*** from SendASPI32Command for SS_PENDING. Do ***not*** check the SRB_Status code. If you have a pending return code, then you should do a WaitFor* Win32 API call on the handle of your event. When the event is signalled, it is then safe to check the SRB_Status field, reuse the SRB, etc. Do ***not*** check the status codes or reuse the SRB until after the event has been signalled. Remember to make sure that the event is a manual reset event (CreateEvent(NULL, TRUE, FALSE, NULL)), and that it has been reset to a not-signalled state before you call ASPI. All of these rules are fully documented in the latest ASPI for Win32 specification available from Robin Mizell at mailto:acap@corp.adaptec.com.

Q: How do I properly check status codes?

A: Once you have properly waited for completion of your SRB, you should check the SRB_Status field for SS_COMP. If you receive an SS_COMP then the SRB has completed successfully and you should ***not*** look at any of the other status codes (host adapter and target status). They may not be zero, but they do not mean anything significant unless you have received a result other than SS_COMP.

Q: Why can't I transfer more than 64KB at a time?

A: Because you have a bus-mastering SCSI host adapter, and there is a limit of 17 scatter/gather elements on your system. Again, understand that a polled I/O (PIO) adapter will not exhibit these limitations.

To understand the problem and the possible solutions, you must understand scatter/gather lists. A scatter/gather list is simply a list of elements which describe all the physical blocks of memory that make up a virtual block of memory. For example, a 64KB contiguous virtual buffer within a ring-3 application may consist of a partial 4KB physical page, 15 full 4KB physical pages, and another partial 4KB page for a total of 17 physical blocks of memory. The starting address and size of a physical block is stored within a single scatter/gather element, so the above 64KB buffer would require a full 17 scatter/gather elements. The scatter/gather list is passed to the DMA controller on the SCSI host adapter to help it walk through the transfer of data. This is where the 64KB transfer limitation is rooted—the 17 elements and the page size of 4KB.

Given the above, there are two ways to increase the transfer size beyond 64KB. The first is increase the number of scatter gather elements allowed, and the second is to decrease the fragmentation of the physical buffers underlying a given virtual buffer. Increasing the allowed number of scatter/gather elements is not allowed under Windows 95 or Windows NT 3.51, but in NT 4.0 there is a registry setting that allows the lists to be expanded to up to 255 elements. The registry entry is specified on a per-bus basis, and the REG_DWORD value within HKEY_LOCAL_MACHINE is:

System\CurrentControlSet\Services*DriverName*\Parameters\Device*n*\MaximumSGList

The *DriverName* is the name of the miniport driver, such as AIC78XX, and *n* in Device*n* is the bus number assigned at initialization. If a value is present in this subkey at device initialization, the SCSI port driver uses MaximumSGList as the initial maximum for scatter/gather list elements. The miniport can lower this value, and its can't be set any higher than 255. What this means is that the miniport must support the higher value, and if it does you will be able to do transfers of up to 1020KB within ASPI.

The second method for getting larger transfers involves a reduction in the number of physical memory blocks which make up a buffer. We have noticed that buffers are normally fairly fragmented, even down to the page level. A 512KB buffer would therefore typically have 128 4KB physical pages with each page starting at a discontiguous address from the previous one. The end result is that a 128 element scatter/gather element would be required to describe the buffer. However, let's say that we have a 512KB buffer made up of 16 32KB physical blocks. Each physical block consists of 4 back-to-back 4KB physical pages. Such a buffer would only require 16 scatter/gather elements, and a transfer could be performed using this buffer.

Both Windows 95 and Windows NT have ways that such largely contiguous buffers can be pieced together through either a VxD or a Kernel Mode Driver (KMD). Remember, though, that even with contiguous buffers there is a 512KB transfer limit in Windows 95, and a practical 64MB transfer limit in Windows NT. The Windows 95 limit cannot be exceeded, and it is debatable whether or not the 64MB barrier can be broken (through chained MDLs).

ASPI provides two functions – GetASPI32Buffer and FreeASPI32Buffer – which perform *dynamic* allocation of largely contiguous buffers. The ASPI functions rely on assistance from the ASPI ring-0 drivers. They work fairly well under Windows 95, but they make direct non-paged pool allocations on Windows NT. This greatly increases the chance of a memory allocation failure in NT, and forces us to look at an alternative method on that operating system. If you are interested in these functions, make sure you have the latest version of the ASPI for Win32 specification.

Under Windows NT large contiguous blocks of memory are really only available during while the system is initializing. During this time a KMD may use the MmAllocateContiguousMemory function to allocate a big block of memory which is purely contiguous (single scatter/gather element). This buffer can be mapped into the UserMode address space using a call to MmMapLockedPages. Of course, this call requires an MDL, and that is where the 64MB limit and the question about chained MDLs comes into play.

Please note that there are *no* source code samples for the above techniques, but that there is enough information for some versed in the Windows 95 or Windows NT DDK to implement a solution.

Q: Why don't SC_ABORT_SRB and SC_RESET_DEV SRBs work?

A: The ASPI SRBs to abort and reset are supported for legacy code which is being ported. The commands will succeed, but the lower operating system layers ignore the commands. In the future, these may start working, but for now the best alternative is to use the SC_GETSET_TIMEOUTS SRB documented in the latest ASPI for Win32 specification. This command is not supported in ASPI for DOS or ASPI for Win16, so your application must be ported to Win32 to support per-process/per-target timeouts.

Q: Why do my SC_EXEC_SCSI_CMD SRBs get aborted without reason?

A: In our experience, an unexplained abort or error is usually caused by a third party bus-reset. In other words, someone besides you (could be another driver, the operating system, or another peripheral on the SCSI bus) has reset the SCSI bus while you have I/O active.

One particularly common reason for this is lack of disconnects on devices which hold the SCSI bus for excessively long periods. If such a device holds the SCSI bus for too long, other I/O for a different target may time out (for example, page file I/O for a hard drive has a 3 second timeout). When such a timeout occurs the entire SCSI bus is reset and all the pending I/O is cancelled. In this case your operation will complete with an aborted status. The solution is to make sure that devices which have long operations (color calibrations, etc.) support disconnects.

Q: Can I access ATAPI devices from ASPI?

A: Not with all versions of ASPI. Neither ASPI nor the ATAPI drivers were originally meant for one another. There were bugs in ATAPI which prevented the full SCSI command set from working properly (the driver did not convert 6 byte CDBs to 12 byte CDBs correctly, etc.), and ASPI was not aware of the target device type enough to take corrective action. ASPI version 4.01 through 4.53 intentionally blocked access to any device controlled by the ATAPI drivers on Windows 95 and Windows NT. This was mainly to prevent crashes when running regular SCSI programs.

Since that time, ASPI has become more robust when it comes to non-SCSI bus types (Fibre Channel, ATAPI, parallel ports devices, etc.), and blocking has been disabled since version 4.53. Similarly, Microsoft's code has become much better with the release of Windows 95 OSR2, and with the NT 4.0 service packs.

If you are using a version of ASPI which is dated earlier than 4.53 then you should get an upgrade to ASPI from our web site before attempting access to ATAPI devices. You should also consider redistributing the latest version of ASPI with your application.

Q: Why does SC_HA_INQUIRY return a maximum SCSI target count different from what I expect when using non-SCSI buses?

A: ASPI does not return the exact same value that the miniport reports back to the SCSI applications. Some SCSI applications expect SCSI-like numbers to come back from the SC_HA_INQUIRY call, and they do not perform well with the strange values that come back from those miniports. For example, the ATAPI miniport may return a max target count of 2 to ASPI internally, but ASPI will convert this to an 8 before returning. ASPI then transparently blocks access to any target ID from 2 to 7. This allows the application to get good SCSI-like max target counts (8, 16, 32).

In general, ASPI will return power-of-two numbers, and as long as that number matches or exceeds the number of targets you can have on a real bus then you shouldn't worry. However, you may want to consider removing any fields from your dialog boxes which display the maximum target count for the user.

Q: Why isn't my device recognized by any ASPI APIs after boot?

A: This is usually related to a problem with how the device interacts with Windows 95 and Windows NT. Both operating systems have some real quirks when it comes to enumeration of the SCSI bus. If your

device does not behave just the way they expect, then it is simply dropped from the list of devices which are supported.

The biggest quirk is that SCSI devices must, in general, be powered on when the system boots. If the device is not powered on during the initial scan, then the device will not be visible until a rescan of the SCSI bus is forced. There is an ASPI command to accomplish this in NT and 95, but it is not until version 4.53 or later of ASPI that these commands were fully functional.

Another quirk is related to logical unit numbers in Windows 95. Windows 95 assumes that LUNs on a target are consecutively numbered, and it abandons a scan as soon as it sees an "empty" LUN. For example, a device at ID 4 with LUN 0, 1, 2, 3 would be seen just fine. A device with LUNS 0, 2, 3, 4, 5, 6, 7 would only show up to ASPI at LUN 0 because of the limitation in the operating system scan.

There are other issues as well: an error during scan drops a device, not supporting certain messages can cause a dropped device, etc. It is best to say that most problems related to missing devices do not have much to do with ASPI, but are usually related to differences of opinion on the SCSI specification between the device and the operating system. Understand, though, that a device which meets Microsoft's view of SCSI usually has pretty solid firmware.

Q: Why are certain CDBs being rejected by my device?
Why does my device hang when I send it this command?
Why isn't ASPI following the SCSI specification with our new device?

A: We really don't know, because these are not ASPI problems. ASPI is not directly involved in any of these failures. ASPI takes CDB blocks and, without looking at them, sends them to the operating system to be executed. What happens from there is highly dependent on the operating system, the miniport, the SCSI adapter hardware, and on the peripheral. Most of these problems require a SCSI bus analyzer to diagnose, and may require special debugging of the target peripheral firmware. Adaptec's ASPI technical support does not assist in these situations.

ASPI for Win32 Questions

Q: Why should I use event notification?

A: Because ASPI requests which use polling or posting force ASPI to start and maintain a background thread. This thread in turn issues the ASPI request using event notification and then waits for completion. This means that if you use event notification you cut down on resources and eliminate a layer of code between you and the kernel mode ASPI driver. Your code will run faster and be smaller.

Q: Why is ASPI for Win32 running so slowly?

A: You may not be using event notification as your completion method. Requests which use polling or posting for their completion method force ASPI to start and maintain a background thread. This thread in turn issues the ASPI request using event notification and waits for completion. When the event completes, the ASPI thread (which runs at slightly higher priority) must be scheduled so that it can complete the request and perform the callback. If you are polling, there may quite a delay before your own applications wakes up enough to realize that the request has been completed.

Q: Why doesn't my process ever complete in Windows NT?

A: A process is not really a unit of execution, so the real problem here is that one or more threads owned by your process have not completed. Windows NT keeps track of which thread has sent each I/O request, including those from ASPI. If a process is being terminated and any of these I/O requests are still outstanding (say the SCSI device hung and the timeout has not yet occurred), then NT will attempt to cancel this I/O. Unfortunately, SCSI requests which have hit an adapter are very hard to cancel, and

NT therefore will decide that it is not appropriate for the thread to terminate. The thread will remain alive until the I/O completes, which could be as long as 30 hours. The solution is to use shorter timeouts, and make sure that all of your I/O has completed before you exit the application.

Q: How do I build a LIB file for my compiler?

A: First, remember to use explicit dynamic linking (i.e. Loadlibrary/GetProcAddress) whenever possible. It gives you more control, and it is capable of handling different ASPI revisions that implicit dynamic linking (LIB file) cannot.

If you are still determined to use implicit dynamic linking, then you really have two choices: write a DEF file and link it directly to your program with an IMPORTS section, or build a LIB file. From the question, it is obvious that we want a LIB file.

Consider the Microsoft tools first. You will need to write a DEF file which contains a "LIBRARY WNASPI32" statement and then an EXPORTS statement with one or more of the ASPI function names. For example, the following would work:

```
LIBRARY WNASPI32
EXPORTS
        GetASPI32SupportInfo @1
        SendASPI32Command @2
```

Take this DEF file (call it WNASPI32.DEF) and run it through the LIB command as follows:

```
LIB /DEF:WNASPI32.DEF /MACHINE:IX86
```

That's it! There should now be a WNASPI32.LIB file in the same directory as the WNASPI32.DEF file, and the LIB can be linked with your application to get an implicit dynamic link at load.

The Borland compiler is another popular compiler. It has a special utility to automate all of the above steps. Just use IMPLIB on WNASPI32.DLL to form an appropriate LIB file.

Q: Can I use ASPI for Win32 from Microsoft Visual Basic® or Borland® Delphi™?

A: Yes. Anything which can call a function in a DLL using the 'C' calling convention should be able to use ASPI. There is no support for this, however, and you will be forced to write your own header files and function declarations using the SDK header files as a starting point.

ASPI for Win16 Questions

Q: Why does my ASPI for Win16 application "hang" while waiting for a posted completion in Windows NT?

A: Unlike Windows 95, the Windows NT implementation of ASPI for Win16 relies on "thunking" to get its work done. Thunking is a one way process, however, and commands from Win16 (the WINASPI.DLL) to Win32 (WNASPI32.DLL) cannot be directly returned. When ASPI for Win32 completes the ASPI for Win16 request, it must post a message to the message queue of a hidden Win16 application residing in the same NTVDM as the ASPI for Win16 application. The message causes WOWPOST.EXE to complete processing of the original Win16 SRB. The problem with this is that Win16 applications are cooperatively multi-tasked. WOWPOST.EXE cannot process the ASPI completion messages unless the ASPI for Win16 application is occasionally issuing GetMessage/PeekMessage calls to yield control. If the ASPI for Win16 application does not yield control ever (say it spins in a really tight loop), then the posting callback cannot ever occur.

Q: How do I properly declare a posting callback under the different compilers?

A: Check you compilers documentation for ways of implementing interrupt routines. The same issues apply for an interrupt routine as an ASPI callback function. Under Microsoft's Visual C++ 1.52, for example, a proper declaration might look something like this:

```
#pragma check_stack( off )
void __far _pascal _export __loadds ASPIPostProc( LPSRB DoneSRB ){}
#pragma check_stack( on )
```

Of particular importance is the "__loadds" keyword which takes care of the classic DS!=SS in 16-bit code. Note that the same flags should ***not*** be used when declaring a callback for ASPI for Win32. In particular, the callback does not occur at interrupt time as it does in ASPI for Win16, and it is also a 'C' style calling convention. This requires, at the very least, that the __loadds and _pascal keywords go away.

ASPI for DOS Questions

Q: Should I load DOS ASPI managers within CONFIG.SYS so that my application can use the ASPI for DOS API from within a Windows 95 DOS box?

A: No. Early versions of ASPI for Windows 95 would attempt to chain into ASPI for DOS drivers when there was no corresponding miniport. This caused several problems and the feature has been removed in the latest versions of ASPI. Since the ASPI for DOS API is supported on any host adapter with a valid miniport driver, and since there are now miniports for virtually ever adapter, there is no need for real-mode DOS ASPI drivers to be loaded in CONFIG.SYS. The only exception is when there is a DOS ASPI application that loads in CONFIG.SYS or AUTOEXEC.BAT before Windows 95 starts.

Q: Can my application use ASPI for DOS within a Windows NT DOS box?

A: No. ASPI for DOS is not a supported API under Windows NT.