July 2003

Serial ATA Native Command Queuing

An Exciting New Performance Feature for Serial ATA

A JOINT WHITEPAPER BY:

Intel Corporation and Seagate Technology LLC

www.intel.com

www.seagate.com

Summary

Native Command Queuing (NCQ) is among the advanced and most anticipated features introduced in the Serial ATA II: Extensions to Serial ATA 1.0 Specification, download available at www.serialata.org. NCQ is a powerful interface/disc technology designed to increase performance and endurance by allowing the drive to internally optimize the execution order of workloads. Intelligent reordering of commands within the drive's internal command queue helps improve performance of queued workloads by minimizing mechanical positioning latencies on the drive. This paper will provide a basis for understanding how the features of NCQ apply to complete storage solutions and how software developers can enhance their applications to take advantage of Serial ATA NCQ thereby creating higher performance applications.

Introduction

Accessing media on mass storage devices, such as hard disc drives (HDD), can have a negative impact on overall system performance. Unlike other purely electrical components in a modern system, HDDs are still largely mechanical devices. Drives are hampered by the inertia of their mechanical components which effectively limits the speed of media access and retrieval of data. Mechanical performance can be physically improved only up to a certain point and these performance improvements usually come at an increased cost of the mechanical components. However, intelligent, internal management of the sequence of mechanical processes can greatly improve the efficiency of the entire workflow. The operative words are intelligent and internal, meaning that the drive itself has to assess the location of the target logical block addresses (LBAs) and then make the appropriate decisions on the order that commands should be executed in to achieve the highest performance.

Native Command Queuing is a command protocol in Serial ATA that allows multiple commands to be outstanding within a drive at the same time. Drives that support NCQ have an internal queue where outstanding commands can be dynamically rescheduled or re-ordered, along with the necessary tracking mechanisms for outstanding and completed portions of the workload. NCQ also has a mechanism that allows the host to issue additional commands to the drive while the drive is seeking for data for another command.

Operating systems such as Microsoft Windows* and Linux* are increasingly taking advantage of multi-threaded software or processor-based Hyper-Threading Technology. These features have a high potential to create workloads where multiple commands are outstanding to the drive at the same time. By utilizing NCQ, the potential disk performance is increased significantly for these workloads.

Drive Basics

Hard drives are electromechanical devices, and therefore hybrids of electronics and mechanical components. The mechanical portions of drives are subject to wear and tear and are also the critical limiting factor for performance. To understand the mechanical limitations, a short discussion of how data on is laid out on a drive may be helpful.

Data is written to the drive in concentric circles, called tracks, starting from the outer diameter of the bottom platter, disc 0, and the first read/write head, head 0. When one complete circle on one side of the disc, track 0 on head 0, is complete the drive starts writing to the next head on the other side of the disc, track 0 and head 1. When the track is complete on head 1 the drive starts writing to the next head, track 0 and head 2, on the second disc. This process continues until the last head on the last side of the final disc has completed the first track. The drive then will start writing the second track, track 1, with head 0 and continues with the same process as it did when writing track 0. This process results in a concentric circles where as writing continues the data moves closer and closer to the inner diameter of the discs. A particular track on all heads, or sides of the discs, is collectively called a cylinder. Thus, data is laid out across the discs sequentially in cylinders starting from the outer diameter of the drive.

One of the major mechanical challenges is that applications rarely request data in the order that it is written to the disc. Rather, applications tend to request data scattered throughout all portions of the drive. The mechanical movement required to position the appropriate read/write head to the right track in the right rotational position is non-trivial.

The mechanical overheads that affect drive performance most are seek latencies and rotational latencies. Both seek and rotational latencies need to be addressed in a cohesive optimization algorithm.

The best known algorithm to minimize both seek and rotational latencies is called Rotational Position Ordering. Rotational Position Ordering (or Sorting) allows the drive to select the order of command execution at the media in a manner that minimizes access time to maximize performance. Access time consists of both seek time to position the actuator and latency time to wait for the data to rotate under the head. Both seek time and rotational latency time can be several milliseconds in duration.

Earlier algorithms simply minimized seek distance to minimize seek time. However, a short seek may result in a longer overall access time if the target location requires a significant rotational latency period to wait for the location to rotate under the head. Rotational Position Ordering considers the rotational position of the disk as well as the seek distance when considering the order to execute commands. Commands are executed in an order that results in the shortest overall access time, the combined seek and rotational latency time, to increase performance.

Native Command Queuing allows a drive to take advantage of Rotational Position Ordering to optimally re-order commands to maximize performance.

Seek Latency Optimization

Seek latencies are caused by the time it takes the read/write head to position and settle over the correct track containing the target Logical Block Addressing (LBA). To satisfy several commands, the drive will need to access all target LBAs. Without queuing, the drive will have to access the target LBAs in the order that the commands are issued. However, if all of the commands are outstanding to the drive at the same time, the drive can satisfy the commands in the optimal order. The optimal order to reduce seek latencies would be the order that minimizes the amount of mechanical movement.

One rather simplistic analogy would be an elevator. If all stops were approached in the order in which the buttons were pressed, the elevator would operate in a very inefficient manner and waste an enormous amount of time going back and forth between the different target locations. As trivial as it may sound, most of today’s hard drives in the desktop environment still operate exactly in this fashion. Elevators have evolved to understand that re-ordering the targets will result in a more economic and, by extension, faster mode of operation. With Serial ATA, not only is re-ordering from a specific starting point possible but the re-ordering scheme is dynamic, meaning that at any given time, additional commands can be added to the queue. These new commands are either incorporated into an ongoing thread or postponed for the next series of command execution, depending on how well they fit into the outstanding workload.

To translate this into HDD technology, reducing mechanical overhead in a drive can be accomplished by accepting the queued commands (floor buttons pushed) and re-ordering them to efficiently deliver the data the host is asking for. While the drive is executing one command, a new command may enter the queue and be integrated in the outstanding workload. If the new command happens to be the most mechanically efficient to process, it will then be next in line to complete.

Keep in mind that re-ordering of pending commands based strictly on the ending location of the heads over the logical block address (LBA) of the last completed command is not the most efficient solution. Similar to an elevator that will not screech to a halt when a person pushes a button for a floor just being passed, HDDs will use complex algorithms to determine the best command to service next. Complexity involves possible head-switching, times to seek to different tracks and different modes of operation, for example, quiet seeks. Parameters taken into account encompass seek length, starting location and direction, acceleration profiles of actuators, rotational positioning (which includes differences between read and write settle times), read cache hits vs. misses, write cache enabled vs. disabled, I/O processes that address the same LBAs, as well as fairness algorithms to eliminate command starvation, to mention a few.

Rotational Latencies

Rotational latency is the amount of time it takes for the starting LBA to rotate under the head after the head is on the right track. In the worst-case scenario, this could mean that the drive will waste one full rotation before it can access the starting LBA and then continue to read from the remaining target LBAs. Rotational latencies depend on the spindle RPM, that is, a 7200-RPM drive will have a worst-case rotational latency of 8.3 msec, a 5400-RPM drive will need up to 11.1 msec, and a 10K-RPM drive will have up to 6 msec rotational latency. In a random distribution of starting LBAs relative to the angular position of the drive's head, the average rotational latency will be one half of the worst-case latency.

I/O delays in the order of milliseconds are quite dramatic compared to the overall performance of any modern system. This is particularly true in scenarios where modern operating systems are utilizing multi-threading or where Hyper-Threading Technology allows quasi-simultaneous execution of independent workloads, all of which need data from the same drive almost simultaneously.

Higher RPM spindles are one approach to reduce rotational latencies. However, increasing RPM spindle rates carries a substantial additional cost. Rotational latencies can also be minimized by two other approaches. The first is to re-order the commands outstanding to the drive in such a way that the rotational latency is minimized. This optimization is similar to the linear optimization to reduce seek latencies, but instead takes into account the rotational position of the drive head in determining the best command to service next. A second-order optimization is to use a feature called out-of-order data delivery. Out-of-order data delivery means that the head does not need to access the starting LBA first but can start reading the data at any position within the target LBAs. Instead of passing up the fraction of a rotation necessary to return to the first LBA of the requested data chunk, the drive starts reading the requested data as soon as it has settled on the correct track and adds the missing data at the end of the same rotation.

Using out-of-order data delivery, for the worst case, the entire transfer will be complete within exactly one rotation of the platter. Without out-of-order data delivery, the worst case time needed to complete the transfer will be one rotation plus the amount of time it takes to rotate over all target LBAs.

Benefits of Native Command Queuing

It is clear that there is a need for reordering outstanding commands in order to reduce mechanical overhead and consequently improve I/O latencies. It is also clear, however, that simply collecting commands in a queue is not worth the silicon they are stored on. Efficient reordering algorithms take both the linear and the angular position of the target data into account and will optimize for both in order to yield the minimal total service time. This process is referred to as "command re-ordering based on seek and rotational optimization" or tagged command queuing. A side effect of command queuing and the reduced mechanical workload will be less mechanical wear, providing the additional benefit of improved endurance. Serial ATA II provides an efficient protocol implementation of tagged command queuing called Native Command Queuing.

Native Command Queuing achieves high performance and efficiency through efficient command re-ordering. In addition, there are three new capabilities that are built into the Serial ATA protocol to enhance NCQ performance including race-free status return, interrupt aggregation, and First Party DMA.

- Race-Free Status Return Mechanism

This feature allows status to be communicated about any command at any time. There is no "handshake" required with the host for this status return to take place. The drive may issue command completions for multiple commands back-to-back or even at the same time.

- Interrupt Aggregation

Generally, the drive interrupts the host each time it completes a command. The more interrupts, the bigger the host processing burden. However, with NCQ, the average number of interrupts per command can be less than one. If the drive completes multiple commands in a short time span – a frequent occurrence with a highly queued workload – the individual interrupts may be aggregated. In that case, the host controller only has to process one interrupt for multiple commands.

- First Party DMA (FPDMA)

Native Command Queuing has a mechanism that lets the drive set up the Direct Memory Access (DMA) operation for a data transfer without host software intervention. This mechanism is called First Party DMA. The drive selects the DMA context by sending a DMA Setup FIS (Frame Information Structure) to the host controller. This FIS specifies the tag of the command for which the DMA is being set up. Based on the tag value, the host controller will load the PRD table pointer for that command into the DMA engine, and the transfer can proceed without any software intervention. This is the means by which the drive can effectively re-order commands since it can select the buffer to transfer on its own initiative.

Detailed Description of NCQ

There are three main components to Native Command Queuing:

1. Building a queue of commands in the drive
2. Transferring data for each command
3. Returning status for the commands that were completed

The following sections will detail how each mechanism works.

Building a Queue

The drive must know when it receives a particular command whether it should queue the command or whether it should execute that command immediately. In addition, the drive must understand the protocol to use for a received command; the command protocol could be NCQ, DMA, PIO, etc. The drive determines this information by the particular command opcode that is issued. Therefore in order to take advantage of NCQ, commands that are specifically for NCQ were defined. There are two NCQ commands that were added as part of the NCQ definition in Serial ATA II, Read FPDMA Queued and Write FPDMA Queued. The Read FPDMA Queued command inputs are shown in Figure 1; the inputs for Write FPDMA Queued are similar. The commands are extended LBA and sector count commands to accommodate the large capacities in today's drives.

The commands also contain a force unit access (FUA) bit for high availability applications. When the FUA bit is set for a Write FPDMA Queued command, the drive will commit the data to media before returning success for the command. By using the FUA bit as necessary on writes, the host can manage the amount of data that has not been committed to media within the drive's internal cache.

Register	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Features	Sector Count 7:0							
Features (exp)	Sector Count 15:8							
Sector Count	TAG					Reserved		
Sector Count (exp)	Reserved							
Sector Number	LBA 7:0							
Sector Number (exp)	LBA 31:24							
Cylinder Low	LBA 15:8							
Cylinder Low (exp)	LBA 39:32							
Cylinder High	LBA 23:16							
Cylinder High (exp)	LBA 47:40							
Device/Head	FUA	1	Res	0	Reserved			
Command	60h							

Figure 1 Read FPDMA Queued Command

One interesting field is the TAG field in the Sector Count register. Each queued command issued has a tag associated with it. The tag is a shorthand mechanism used between the host and the device to identify a particular outstanding command. Tag values can be between 0 and 31, although the drive can report support for a queue depth less than 32. In this case, tag values are limited to the maximum tag value the drive supports. Having tag values limited to be between 0 and 31 has some nice advantages, including that status for all commands can be reported in one 32-bit value. Each outstanding command must have a unique tag value.

The Read and Write FPDMA Queued commands are issued just like any other command would be, i.e. the taskfile is written with the particular register values and then the Command register is written with the command opcode. The difference between queued and non-queued commands is what happens after the command is issued. If a non-queued command was issued, the drive would transfer the data for that command and then clear the BSY bit in the Status register to tell the host that the command was completed. When a queued command is issued, the drive will clear BSY immediately, before any data is transferred to the host. In queuing, the BSY bit is not used to convey command completion. Instead, the BSY bit is used to convey whether the drive is ready to accept a new command. As soon as the BSY bit is cleared, the host can issue another queued command to the drive. In this way a queue of commands can be built within the drive.

Transferring Data

NCQ takes advantage of a feature called First Party DMA to transfer data between the drive and the host. First Party DMA allows the drive to have control over programming the DMA engine for a data transfer. This is an important enhancement since only the drive knows the current angular and rotational position of the drive head. The drive can then select the next data transfer to minimize both seek and rotational latencies. The First Party DMA mechanism is effectively what allows the drive to re-order commands in the most optimal way.

As an additional optimization, the drive can also return data out-of-order to further minimize the rotational latency. First Party DMA allows the drive to return partial data for a command, send partial data for another command, and then finish sending the data for the first command if this is the most efficient means for completing the data transfers.

To program the DMA engine for a data transfer, the drive issues a DMA Setup FIS to the host, shown in Figure 2. There are a few key fields in the DMA Setup FIS that are important for programming the DMA engine.

0	Reserved (0)	Reserved (0)	A	I	D	Reserved (0)	FIS Type (41h)
1	0						TAG
2	0						
3	Reserved (0)						
4	DMA Buffer Offset						
5	DMA Transfer Count						
6	Reserved (0)						

Figure 2 DMA Setup FIS

The TAG field identifies the tag of the command that the DMA transfer is for. For host memory protection from a rogue device, it is important to not allow the drive to indiscriminately specify physical addresses to transfer data to and from in host memory. The tag acts as a handle to the physical memory buffer in the host such that the drive does not need to have any knowledge of the actual physical memory addresses. Instead, the host uses the tag to identify which PRD table to use for the data transfer and programs the DMA engine accordingly.

The DMA Buffer Offset field is used to support out-of-order data delivery, also referred to as non-zero buffer offset within the specification. Non-zero buffer offset allows the drive to transfer data out-of-order or in-order but in multiple pieces.

The DMA Transfer Count field identifies the number of bytes to be transferred. The D bit specifies the direction of the transfer (whether it is a read or a write). The A bit is an optimization for writes called Auto-Activate, which can eliminate one FIS transfer during a write command.

One important note for HBA designers is that new commands cannot be issued between the DMA Setup FIS and the completion of the transfer of the data for that DMA Setup FIS. It is important that the drive is not interrupted while actively transferring data since taking a new command may cause a hiccup in the transfer of data. Thus this restriction was added explicitly in the NCQ definition. Analogously, drives cannot send a Set Device Bits FIS before the completion of the data transfer for that DMA Setup FIS. There is one exemption to this restriction; if an error is encountered before all of the data is transferred, a drive may send a Set Device Bits to terminate the transfer with error status.

After the DMA Setup FIS is issued by the drive, data is transferred using the same FISes that are used in a non-queued DMA data transfer operation.

Status Return

Command status return is race-free and allows interrupts for multiple commands to be aggregated. The host and the drive work in concert to achieve race-free status return without handshakes between the host and drive. Communication between the host and drive about which commands are outstanding is handled through a 32-bit register in the host called SActive. The SActive register has one bit allocated to each possible tag, i.e. bit *x* shows the status of the command with tag *x*. If a bit in the SActive register is set, it means that a command with that tag is outstanding in the drive (or a command with that tag is about to be issued to the drive). If a bit in the SActive register is cleared, it means that a command with that tag is not outstanding in the drive. The host and drive work together to make sure that the SActive register is accurate at all times.

The host can set bits in the SActive register, while the device can clear bits in the SActive register. This ensures that updates to the SActive register require no synchronization between the host and the drive. Before issuing a command, the host sets the bit corresponding to the tag of the command it is about to issue. When the drive successfully completes a command, it will clear the bit corresponding to the tag of the command it just finished.

The drive clears bits in the SActive register using the Set Device Bits FIS, shown in Figure 3. The SActive field of the Set Device Bits FIS is used to convey successful status to the host. When a bit is set in the SActive field of the FIS, it means that the command with the corresponding tag has completed successfully. The host controller will clear bits in the SActive register corresponding to bits that are set to one in the SActive field of a received Set Device Bits FIS.

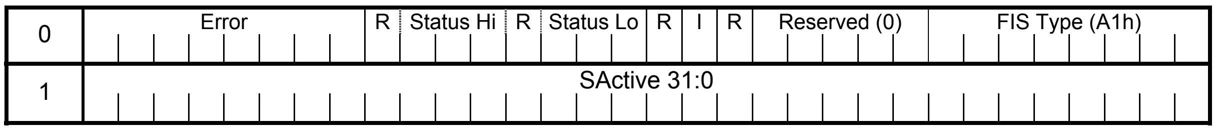

Figure 3 Set Device Bits FIS

Another key feature is that the Set Device Bits FIS can convey that multiple commands have completed at the same time. This ensures that the host will only receive one interrupt for multiple command completions. For example, if the drive completes the command with tag 3 and the command with tag 7 very close together in time, the drive may elect to send one Set Device Bits FIS that has both bit 3 and bit 7 set to one. This will complete both commands successfully and is guaranteed to generate only one interrupt.

Since the drive can return a Set Device Bits FIS without a host handshake, it is possible to receive two Set Device Bits FISes very close together in time. If the second Set Device Bits FIS arrives before host software has serviced the interrupt for the first, then the interrupts are automatically aggregated. This means that the host effectively only services one interrupt rather than two thus reducing overhead.

How Applications Take Advantage of Queuing

The advantages of queuing are only realized if a queue of requests is built to the drive. One major issue in current desktop workloads is that many applications ask for one piece of data at a time, and often only ask for the next piece of data once the previous piece of data has been received. In this type of scenario, the drive is only receiving one outstanding command at a time. When only one command is outstanding at a time, the drive can perform no re-ordering and all the benefit of queuing is lost.

Note that with the advent of Hyper-Threading Technology, it is possible to build a queue even if applications issue one request at a time. Hyper-Threading Technology allows significantly higher amounts of multi-threading to occur such that multiple applications are more likely to have I/O requests pending at the same time. However, the best performance improvement can only be achieved if applications are slightly modified to take advantage of queuing.

The modifications to take advantage of queuing are actually fairly minor. Today most applications are written to use synchronous I/O, also called blocking I/O. In synchronous I/O, the function call to read from or write to a file does not return until the actual read or write is complete. In the future, applications should be written to use asynchronous I/O. Asynchronous I/O is non-blocking, meaning that the function call to read from or write to a file will actually return before the request is complete. The application determines whether the I/O has completed by checking for an event to be signaled or by receiving a callback. Since the call returns immediately, the application can continue to do useful work, including issuing more read or write file functions.

The preferred method for writing an application that needs to make several different file accesses is to issue all of the file accesses using non-blocking I/O calls. Then the application can use events or callbacks to determine when individual calls have completed. If there are a large number of I/Os, on the order of four to eight, by issuing all of the I/Os at the same time the total time to retrieve all of the data can be cut in half.

Using Asynchronous I/O in Windows*

In Windows* applications, there are two main functions used for accessing files called ReadFile and WriteFile. In a typical application, these functions are used in a blocking, or synchronous, manner. An example is shown below of reading 1KB from a file using ReadFile:

```
bStatus = ReadFile(
             hFile,              // Handle to file to read from
             pBuffer,            // Pointer to buffer to place data in
             1024,               // Want to read 1024 bytes from the file
             &numBytesRead,      // Number of bytes read from the file
             NULL);              // Synchronous so overlapped parameter is NULL

//
// Code for checking the status value and ensuring there were not any errors.
//
...
```

When this call is made, it does not return until all of the data is read from the file. During this time, the application cannot do any useful work nor can it issue any more I/O calls within this thread.

A preferred method for reading data from a file is to use ReadFile and WriteFile in an asynchronous manner by opening the file using the flag FILE_FLAG_OVERLAPPED. The same example is shown below using the asynchronous mechanism.

```
//
// Fill in the OVERLAPPED structure used for asynchronous IO.
//
overlap.offset = 0;                 // Offset in the file to start reading from
overlap.offsetHigh = 0;             // Upper 32-bits of offset
overlap.hEvent = hEvent;           // Event to trigger when complete, initialized
                                    // using CreateEvent()

//
// Make the call to start reading the file.
//
bStatus = ReadFile(
             hFile,              // Handle to file to read from
             pBuffer,            // Pointer to buffer to place data in
             1024,               // Want to read 1024 bytes from the file
             &numBytesRead,      // Number of bytes read from the file
```

*Other names and brands may be claimed as the property of others.

```
             &overlap);          // Contains event and offset for asynchronous
                                 // operation

//
// Make sure there was not an error.
//
if ((ERROR_SUCCESS != bStatus) || (ERROR_IO_PENDING != bStatus))
{
       fprintf(stderr, "ReadFile call failed.\n");
       ExitProcess();
}

//
// Make more IO calls or do other useful work.
//
...

//
// Check that the IO has been completed.  Note that multiple completions can be
// checked for at the same time.
//
dwResult = WaitForMultipleObjects(
             n,                  // Number of IOs issued
             handleArray,        // Array of handles to the events in the
                                 // overlapped structure for each IO
             TRUE,               // Wait for all IOs to be completed, could
                                 // wait for first to complete and then act on
                                 // that data, then check for more
             50);                // Wait up to 50 milliseconds for completion,
                                 // could be infinity.

//
// Check the value of dwResult and also call GetOverlappedResult to ensure
// that each IO that completed was with good status.
//
...
```

As can be seen from this example, asynchronous I/O involves more code and can seem a bit cumbersome. However the performance potential of combining queuing with asynchronous IO is worth the extra lines of code.

Conclusion

Native Command Queuing has the potential to offer significant performance advantages. The benefits of NCQ are realized when a queue of commands is built up in the drive such that the drive can optimally re-order the commands to reduce both seek and rotational latency. NCQ delivers an efficient solution through features in the Serial ATA protocol including race-free status return, interrupt aggregation, and First Party DMA.

The NCQ performance advantage can only be realized when a queue is built in the drive. Therefore it is imperative that applications and operating systems use asynchronous I/O where possible and keep the drive queue active with multiple commands at a time. Independent Software Vendors (ISVs) and operating system providers are encouraged to start utilizing asynchronous I/O now in order to best take advantage of the benefits of NCQ when as it is widely deployed in late 2003 and early 2004.

Authors

Amber Huffman, Staff Architect, Intel Corporation

Amber Huffman is a staff architect in a research and development group at Intel where her responsibilities include storage performance and architecture. Amber's current projects concentrate on definition and development of Serial ATA II advanced features and leading the Advanced Host Controller Interface (AHCI) definition. Amber holds a BSE in Computer Engineering from the University of Michigan and has been with Intel for 6 years.

Joni Clark, Product Marketing Manager, Seagate Technology

Joni Clark is a product marketing manager for desktop interfaces at Seagate Technology, focusing on promoting Serial ATA technology to customers from system builders to end users. Her current responsibilities include sales and marketing training, Serial ATA adoption and serial interface technology positioning. Joni is also chairperson for the industry's Serial ATA Working Group marketing team to lead the promotion of the Serial ATA interface.

Disclaimer