UNITED STATES PATENT AND TRADEMARK OFFICE

_____

BEFORE THE PATENT TRIAL AND APPEAL BOARD

_____

SANDISK TECHNOLOGIES, INC. AND WESTERN DIGITAL TECHNOLOGIES, INC.,
Petitioner

U.S. Patent No. 8,554,968

_____

DECLARATION OF DR. PAUL MIN, UNDER 37 C.F.R. § 1.68 IN SUPPORT OF PETITION FOR *INTER PARTES* REVIEW

TABLE OF CONTENTS

I. Introduction....5

II. Qualifications and Professional Experience....7

III. Level of Ordinary Skill in the Art....12

IV. Relevant Legal Standards....13

V. Background....14

A. Memory Systems....14

B. Command and Response Queues....15

C. Circular Queues....15

D. Doorbell Registers....18

E. Interrupts....19

VI. Overview of the '968 Patent....20

A. Summary of the '968 patent....20

B. Prosecution History of the '968 Patent....30

VII. Claim Construction....33

VIII. Identification of how the Claims are Unpatentable....33

A. Ground 1: Claims 1-18 are obvious in view of Borchers....34

1. Summary of Borchers....34

2. The PCI Standard Represents Background Knowledge of a POSITA....40

3. Claim 1....40

4. Claim 2....73

5. Claim 3....74

6. Claim 4 75
7. Claim 5 77
8. Claim 6 81
9. Claim 7 84
10. Claim 8 86
11. Claim 9 99
12. Claim 10 103
13. Claim 11 105
14. Claim 12 105
15. Claim 13 106
16. Claim 14 106
17. Claim 15 108
18. Claim 16 109
19. Claim 17 111
20. Claim 18 111

B. Ground 2: Claims 1-18 are obvious in view of Borchers and the PCI standards. 112

1. Summary of the PCI Standards 113
1. Reasons to combine the PCI standard with Borchers .. 115
2. Claim Analysis for claims 1-18 118

C. Ground 3: Claims 1-18 are obvious in view of Borchers, the PCIe/PCI standards, and Serebin. 118

1. Summary of Serebrin 119
2. Serebrin is Entitled to its Provisional Filing Date. 122

3. Reasons to combine Serebrin with Borchers and the PCI standards 125

4. Claim Analysis for claims 1-18 129

IX. Conclusion 130

I, Paul Min, do hereby declare as follows:

I. INTRODUCTION

1. I am making this declaration at the request of Western Digital Technologies, Inc. in the matter of the *Inter Partes* Review of U.S. Patent No. 8,554,968 ("the '968 patent") to Onufryk *et al*.

2. I am being compensated for my work in this matter at my usual hourly rate for this work. I am also being reimbursed for reasonable and customary expenses associated with my work and testimony in this investigation. My compensation is not contingent on the outcome of this matter or the specifics of my testimony.

3. I have been asked to provide my opinions regarding whether claims 1-18 ("the Challenged Claims") of the '968 patent are unpatentable as they would have been obvious to a person having ordinary skill in the art ("POSITA") at the time of the alleged invention, in light of the prior art. It is my opinion that all of the limitations of the Challenged Claims would have been obvious to a POSITA.

4. In the preparation of this declaration, I have studied:

a. the '968 patent, Ex.1001;

b. the prosecution history of the '968 patent ("'968 File History"), Ex.1002;

c. U.S. Patent Publication No. 2010/0262979 to Borchers ("Borchers"), Ex.1005;

d. PCI Local Bus Specification 3.0 (February 3, 2004) ("PCI"), Ex.1006;

e. PCI Express 2.0 Base Specification Revision 0.9 (September 11, 2006), ("PCIe"), Ex.1007;

f. U.S. Patent Publication No. 2011/0197003 to Serebrin et al. ("Serebrin"), Ex.1008; and

g. U.S. Provisional Patent No. 61/301,937, ("Serebrin Provisional"), Ex.1009.

5. In forming the opinions expressed below, I have considered:

the documents listed above;

the relevant legal standards, including the standard for obviousness, and any additional authoritative documents as cited in the body of this declaration; and

my own knowledge and experience based upon my work in the field of optical communication as described below, as well as the following materials:

a. The Authoritative Dictionary of IEEE Standards Terms: Seventh Edition; ("IEEE"), Ex.1010;

b. An Internet Archive capture of the PCI Express standard page, Ex.1014;

c. An Internet Archive capture of the PCI standard page, Ex.1015;

d. U.S. Patent Publication No. 2007/0245061, Ex.1016;

e. U.S. Patent No. 7,924,914, Ex.1017; and

f. U.S. Patent Publication No. 2010/0185719, Ex.1018.

6. Unless otherwise noted, all emphasis in any quoted material has been added.

II. QUALIFICATIONS AND PROFESSIONAL EXPERIENCE

7. My qualifications and professional experience are described in my *Curriculum Vitae*, a copy of which can be found in Exhibit 1004. The following is a brief summary of my relevant qualifications and professional experience.

8. I received B.S., M.S., and Ph.D. degrees in Electrical Engineering from the University of Michigan in 1982, 1984, and 1987, respectively.

9. I am an internationally recognized expert in the field of high speed communication and computing, and the development of semiconductor devices for the related applications. I joined Washington University in St. Louis in 1990. I currently serve as a Senior Professor at the Preston M. Green Department of Electrical & Systems Engineering of the McKelvey School of Engineering.

10. Before joining Washington University, I spent three years at Bellcore

(now Telcordia Technologies, Inc., which is part of Ericsson), where I was a lead systems engineer for the Bell Operating Companies in transitioning away from their voice centric networks at the time of the AT&T divesture, into multi-services capable networks suited for the Internet era. I received an Outstanding Achievement Award for his contributions by Bellcore in 1989.

11. After coming to Washington University, I helped the development of a communication, computer, and electronics curricula in the School of Engineering and Applied Science. I regularly teach undergraduate and graduate courses, and supervise students to conduct research in these areas.

12. I designed one of the earliest nationwide, commercial-scale, code division multiple access, or CDMA, networks after winning a cellular communication license from the Korean Government through a highly competitive selection process. I was the leading author of the technical blueprint for this wireless cellular network. Since then, I have been continuously researching in 3G UMTS / 4G LTE / 5G NR IoT wireless networks as well as various generations of WiFi technology.

13. I have collaborated and consulted with numerous companies and organizations around the world, and served as an adviser for several national governments. I also founded and managed two start-up companies during 1997 - 2008. One of the commercial products I developed in my companies received a

best product award in 2002 from a major trade journal for the Internet industry. In the companies I founded as well as part of my research activities at Washington University, I have designed and implemented numerous high-speed, high-performance electronic circuitries.

14. I am a recipient of the Best Paper Award at MOBILITY 2011 in October 2011 in Barcelona, Spain, and a recipient of the Best Paper Award at 18th ISATA Award of Technical Excellence in 1988 in Florence, Italy. I received a Research Initiation Award in 1993 from the Defence Advance Research Project Agency (DARPA). I was a Rockwell Fellow in 1988 and 1989. In addition, I was the recipient of the Best Teaching Assistant Award the Best Graduate Student Award at the University of Michigan in Ann Arbor.

15. I was a member of the Presidential Business Advisors Committee (to President George W. Bush), and was named 2002 Businessman of the Year, for Missouri, by the Wall Street Journal for my entrepreneurial effort. I have served as an Ambassador for the McDonnell International Scholars Academy (2007-2013).

16. I have served as the Chair of the Saint Louis Section of the Institute for Electrical and Electronics Engineering (IEEE) (2014), and served as a member of the Executive Committee (2011-2015) and the Chair of the Communication Society for the Saint Louis Section of the IEEE for a number of years. I have been an organizer for several international symposiums, a guest editor of international

journals, and have given a number of invited talks. I hold 11 U.S. patents.

17. My research area includes switching, routing, performance control and security in the communication networks. For example, I and my students invented and implemented several methods of high performance switching, including multi-channel switches that alleviate the speed and performance constraints of electronic switches. The pattern matching engines that my research group developed are used for processing complex instructions embedded in communication packets at billions of packets per second. The content search engine that my research group developed has been incorporated into numerous cybersecurity systems in active networks and is being developed for machine learning and data mining for various applications including cybersecurity and the Internet of Things (IoT).

18. I have developed and implemented a number of semiconductor integrated circuit chips. These integrated circuit chips include switch fabrics, processors, high speed interfaces, protocol transposers, controllers, content search engine, pattern matching, and memory cells and arrays. Many of the integrated circuit chips I have developed incorporates high speed memory modules interacting with the memory controllers to perform various types of operations including pointer processing, pattern matching, tree-searching, and mathematical manipulation that require extremely fast read/write/bulk-transfer from/to the

memory contents. For example, the latest generations of the switching fabric that I developed incorporated what is known in the switching field as the virtual output queue based on shared memory. In the switch fabric based on the virtual output queue based on shared memory architecture, there is no space-division switching, and rather, the switching is performed from the shared memory that takes in all of the incoming packets, which are manipulated by the pointers that organized the packets in the memory.

19. In addition, I am actively involved in the development of virtualized cluster computing networks. I developed algorithms to control code downloading from the network servers to the end-user units. Based on these algorithms, applications running on the smartphones and portable devices can run much faster without consuming excessive battery powers. By quantifying the information related to the usage patterns, a large amount of user data can be stored, which can help improve the mobile application significantly.

20. I also focus on future generations of wireless technology. I have developed methods that enable coordinated transmission and reception of wireless signals among multiple base stations. Previously, radio signals from multiple base stations are considered as interference. Leveraging on these results, a cluster of base stations can work together as a single resource enhancing the quality of radio signals across extremely noisy wireless channels. This research is critical in

accommodating the next generation of the Internet and wireless networks, which integrates numerous Internet of Things (IoT) as part of the intelligent networks.

III. LEVEL OF ORDINARY SKILL IN THE ART

21. I understand there are multiple factors relevant to determining the level of ordinary skill in the pertinent art, including (1) the levels of education and experience of persons working in the field at the time of the invention; (2) the sophistication of the technology; (3) the types of problems encountered in the field; and (4) the prior art solutions to those problems.

22. A person of ordinary skill in the art ("POSITA") in the field of the '968 patent, as of its earliest possible filing date of August 16, 2010, would have been someone knowledgeable and familiar with memory systems. That person would have a bachelor's degree in electrical engineering, computer science, or equivalent training, and approximately two years of experience working in the field of memory systems. Lack of work experience can be remedied by additional education, and vice versa.

23. For purposes of this Declaration, in general, and unless otherwise noted, my statements and opinions, such as those regarding my experience and the understanding of a POSITA generally (and specifically related to the references I consulted herein), reflect the knowledge that existed in the field as of the priority date of the '968 patent. Unless otherwise stated, when I provide my understanding

and analysis below, it is consistent with the level of a POSITA prior to the priority date of the ’968 patent.

IV. RELEVANT LEGAL STANDARDS

24. I am not an attorney. In preparing and expressing my opinions and considering the subject matter of the ’968 patent, I am relying on certain basic legal principles that counsel have explained to me. These principles are discussed below.

25. I understand that prior art to the ’968 patent includes patents and printed publications in the relevant art that predate the priority date of the alleged invention recited in the ’968 patent. For purposes of this Declaration, I am applying August 16, 2010 as the earliest possible priority date of the ’346 patent.

26. I have been informed that a claimed invention is unpatentable under 35 U.S.C. § 103 if the differences between the invention and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which the subject matter pertains. I have also been informed by counsel that the obviousness analysis takes into account factual inquiries including the level of ordinary skill in the art, the scope and content of the prior art, and the differences between the prior art and the claimed subject matter.

27. I have been informed by counsel that the Supreme Court has

recognized several rationales for combining references or modifying a reference to show obviousness of claimed subject matter. Some of these rationales include the following: (a) combining prior art elements according to known methods to yield predictable results; (b) simple substitution of one known element for another to obtain predictable results; (c) use of a known technique to improve a similar device (method, or product) in the same way; (d) applying a known technique to a known device (method, or product) ready for improvement to yield predictable results; (e) choosing from a finite number of identified, predictable solutions, with a reasonable expectation of success; and (f) some teaching, suggestion, or motivation in the prior art that would have led one of ordinary skill to modify the prior art reference or to combine prior art reference teachings to arrive at the claimed invention.

V. BACKGROUND

A. Memory Systems

28. When a host system seeks to read from or write to memory storage device, it communicates with a memory controller associated with that memory device. For example, when a host computing system reads from or writes to a memory device, the host sends commands to the memory controller of that memory device. The memory controller of that memory storage device then processes those commands and reports back to the host. For example, when the

memory controller receives a write command, it then writes the appropriate data to memory and reports back to the host.

B. Command and Response Queues

29. To manage the exchange of information between the host and the memory controller, the host system relies on queues. To communicate commands to the memory controller, the host may use a "**command queue** that is populated with commands for retrieval by the data storage device." Borchers, [0004]. The host may also use a "**response queue** that is populated with responses by the data storage." Borchers, [0004]. These "pairs of queues enable the host to communicate with the data storage device using multiple threads or cores in an efficient manner." Borchers, [0017].

30. To send commands to the controller, "[t]he host 106 may write a command to the command buffer." Borchers, [0034]. The memory controller will then "read the command out of the command buffer." Borchers, [0034]. When the memory controller completes the command, it sends a response back to the host by writing that response in the response buffer. *See* Borchers, [0035]. When the host is then ready to process a completion message from the response queue, it will then "read the responses from the response buffer." Borchers, [0035].

C. Circular Queues.

31. Command and response queues may be implemented as circular

queues. A circular queue uses a designated section of memory to store items in the queue. A head pointer is used to identify the head of the queue (the item that has been in the queue the longest) within that designated section of memory. A tail pointer is used to identify the tail end of the queue (the most recently added item) within the designated section of memory. The diagram below illustrates how the tail pointer is updated when items are added to the queue.

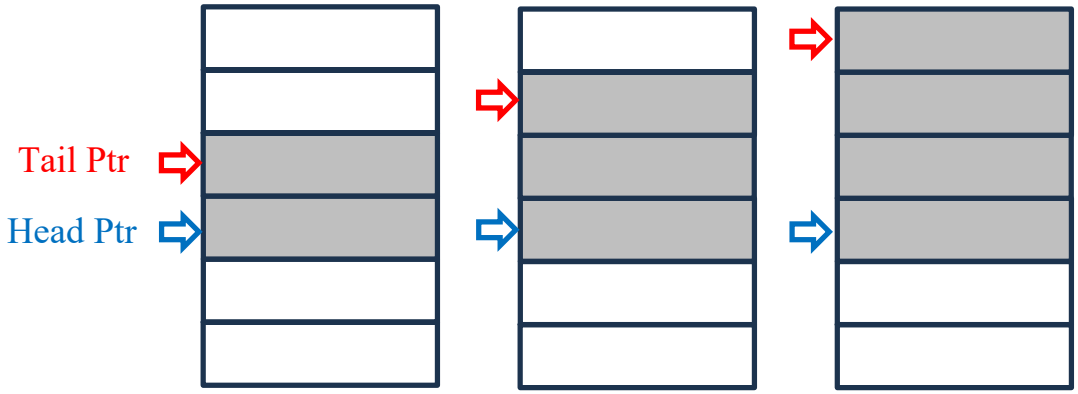


Fig. A

32. In other words, when items are *added* to the queue, the tail pointer is updated to identify where the most recently placed item is located. When items are *removed* from the queue, the head pointer is updated, as shown in the figure below.

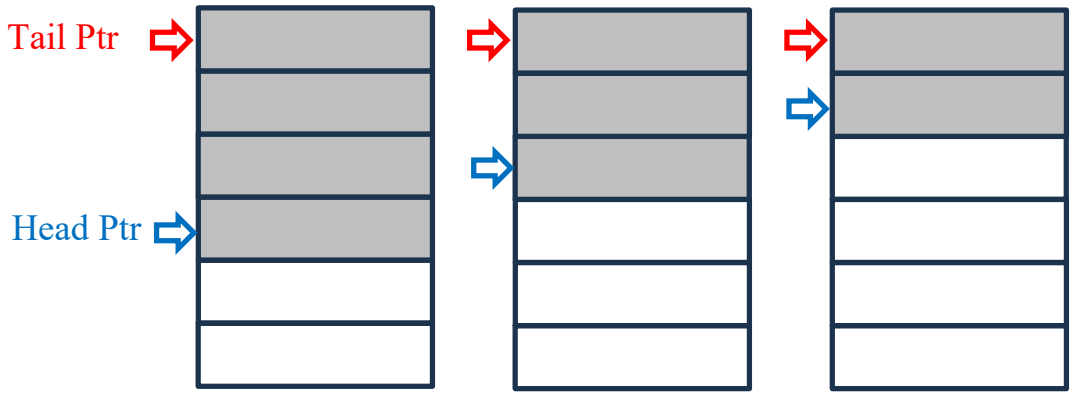


Fig. B

33. During operation of the queue, the tail pointer will reach the end of the designated section of memory. When at this end, and a new item is added to the queue, the tail pointer *circles* back to the other end of the designated section of memory, as shown in the diagram below.

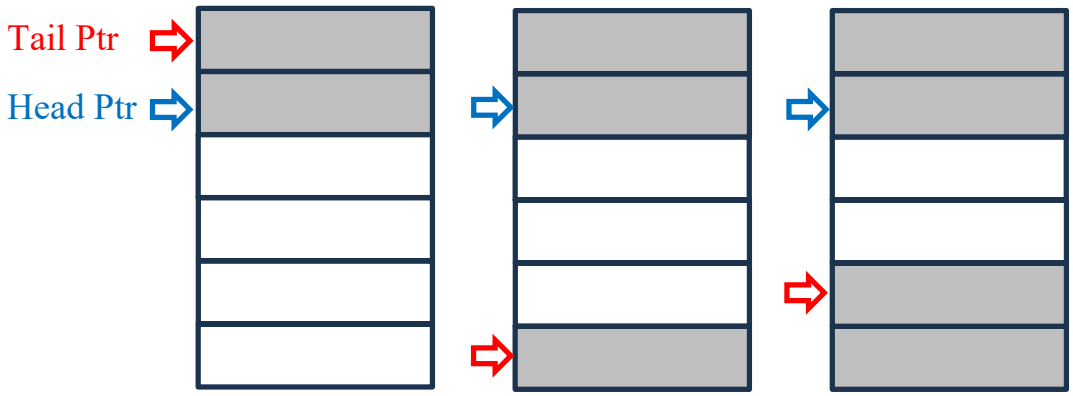


Fig. C

Thus, a circular queue does not have to reshuffle the items within the queue every time an item is added to or removed from the queue.

34. When the command and response queues described above are

implemented as circular queues, "the circular queues 159 of the command buffer 119 and the response buffer 123 include a head pointer and a tail pointer." Borchers, [0030]. "Commands and responses may be inserted into the circular queue 159 at the tail pointer and removed at the head pointer." Borchers, [0030].

35. Using two queues to exchange data in this manner is known as the producer-consumer model. "The host 106 may be the producer of the command buffer 119 and the consumer of the response buffer 123. The data storage device 100 may be the consumer of the command buffer 119 and the producer of the response buffer 123." Borchers, [0030]. In so doing, "The host 106 may write the command tail pointer and the response head pointer and may read the command head pointer and the response tail pointer. The data storage device 100 may write the command head pointer and the response tail pointer and may read the command tail pointer and the response head pointer." Borchers, [0030].

36. The PCI Standard also describes this producer-consumer model. As explained in the PCI specification: "In this model, one agent, the Producer, produces or creates the data and another agent, the Consumer, consumes or uses the data." PCI, 308. When using this model, the host and memory controller need to communicate both the head and tail pointers to each other.

D. Doorbell Registers

37. As explained in the PCI standard, "[t]he Producer and Consumer

communicate between each other via a flag and a status element. The Producer sets the flag when all the data has been written and then waits for a completion status code. The Consumer waits until it finds the flag set, then it resets the flag, consumes the data, and writes the completion status code." PCI, 308. "In practice, **the flag might be a doorbell register** in a device or it might be a main-memory pointer to data located somewhere else in memory." PCI, 308. "[T]he Consumer might signal the Producer using an interrupt or another doorbell register." PCI, 308.

38. The Borchers reference similarly uses a doorbell registers. Borchers' host system (the consumer of the response queue) uses a doorbell register in the memory controller to store the response queue head pointer. "The command tail pointer and **the response head pointer** may be **stored in registers of the data storage device 100** in memory mapped I/O space within the controller 110." Borchers, [0031]. "The host 106 is configured to read the responses from the response buffer 123 and update the head pointer in the MMIO space of the data storage device to indicate that the host has received the response." Borchers, [0035]. Accordingly, it was known to use doorbell registers for managing circular queues.

E. Interrupts

39. Interrupt signals are used to notify a processor that there is something

for that processor to do. For example, when the memory controller seeks to notify the processor that there are events in the response queue for processing, it can send an interrupt signal. Memory controllers may thus include an "interrupt processor [that] is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123." Borchers, [0035].

40. Interrupt signals may exist in a variety of forms. The PCI standard describes, for example, MSI and MSI-x interrupts. "Message Signaled Interrupts (MSI) is an optional feature that enables a device function to request service by writing a system-specified data value to a system-specified address (using a PCI DWORD memory write transaction)." PCI, 231.

41. Interrupt signals are often referred to as interrupt vectors. As explained in the PCI standard, "System software initializes the message address and message data (from here on referred to as the "**vector**") during device configuration, allocating one or more vectors to each MSI capable function." PCI, 231. Because an interrupt signal can include two components (address and data), it is referred to as a vector. An interrupt vector includes the starting address of the interrupt service routine.

VI. OVERVIEW OF THE '968 PATENT

A. Summary of the '968 patent

42. The '968 patent describes, and claims known concepts related to a

memory controller that triggers interrupt signals in a host system. Generally, when a memory controller completes a read or write command, it then sends a completion response to the host. That completion response is then placed into a completion queue for processing by the host. In some cases, the memory controller can send an interrupt signal to the host whenever it places a completion notification in the completion queue. However, in some cases, the memory controller may refrain from sending interrupts until certain conditions are met. One example of such a condition is that a certain number of completion notifications have been sent to the host. The process of refraining from sending interrupts until certain conditions are met is sometimes referred to as interrupt coalescing.

43. The ’968 patent relates to communication between a non-volatile memory controller 105 and a host processing unit 110, as shown in Fig. 1 below.

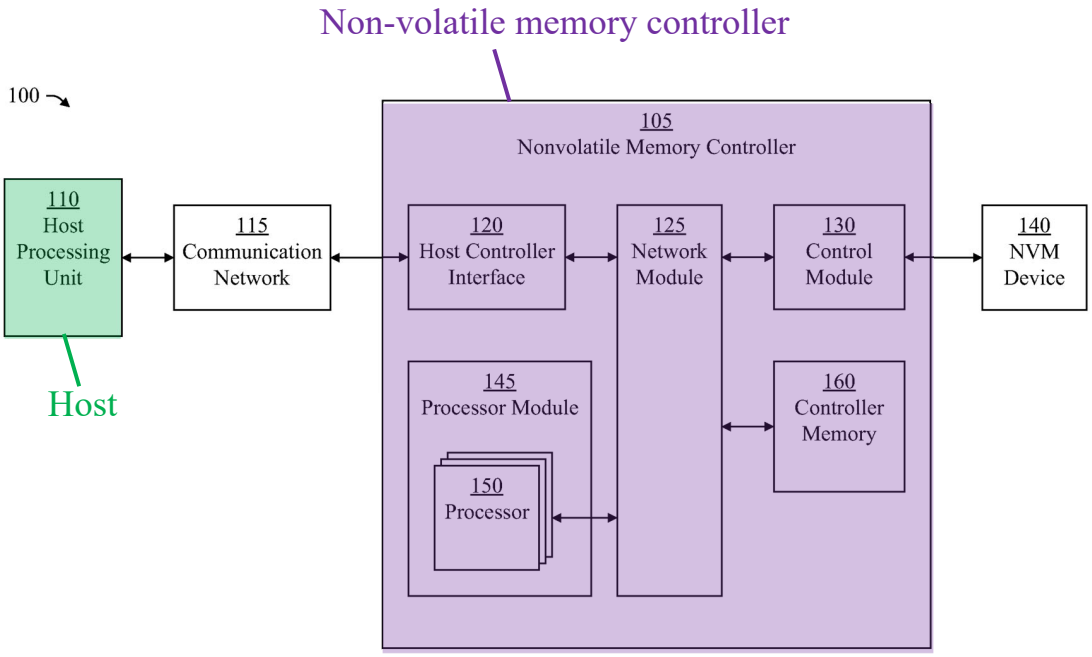


FIG. 1

'968 patent, Fig. 1 (annotated).

44. To process read or write commands from the host 110 to the controller 105, the host processing unit 110 "includes submission queues 520 and completion queues 525." '968 patent, 9:38-39. "The submission queues 520 store nonvolatile memory commands for controlling operation of the nonvolatile memory controller 105." '968 patent, 9:39-41. "The completion queues 525 store completion statuses corresponding to nonvolatile memory commands stored in the submission queues 520." '968 patent, 9:41-44. The completion and submission queues are shown in Fig. 5 below.

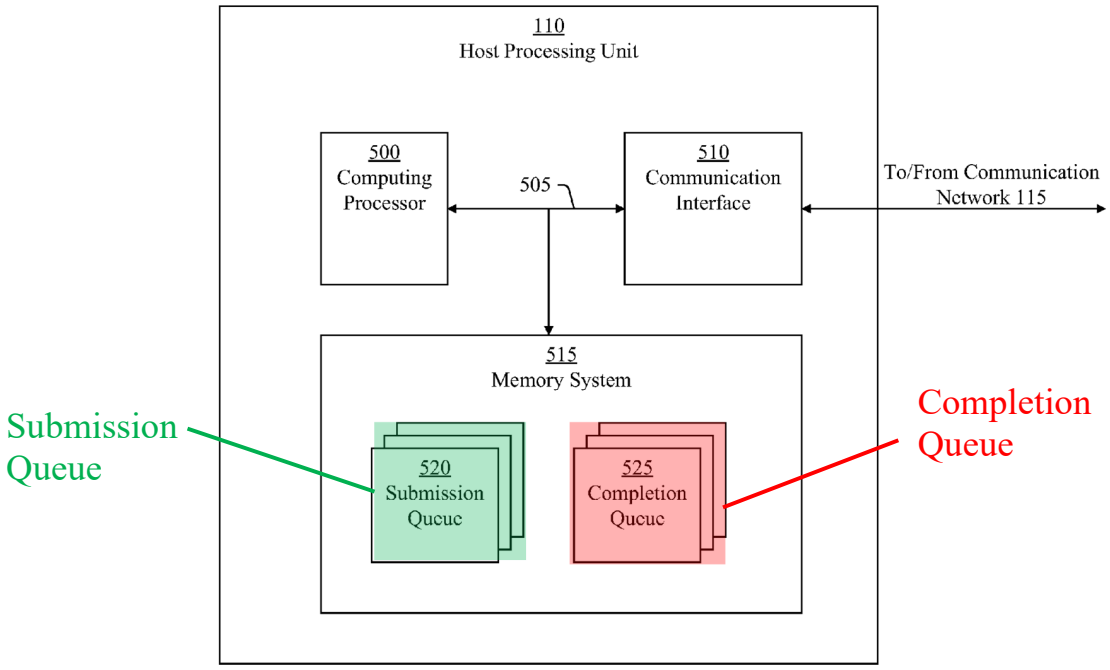


FIG. 5

'968 patent, Fig. 5 (annotated).

45. The submission queue 520 is used to provide read or write commands to the memory device. "[T]he computing processor 500 generates nonvolatile memory commands for controlling the nonvolatile memory controller 105 and writes the nonvolatile memory commands into the submission queues 520." '968 patent, 9:47-50. Then, when the memory controller is ready to process those commands, "[t]he nonvolatile memory controller 105 obtains the nonvolatile memory commands stored in the submission queues 520 and processes the nonvolatile memory commands to manage operation of the nonvolatile memory controller 105." '968 patent, 9:50-54; *see also* '968 patent, 9:47-10:35.

46. The completion queue is used to manage completion notifications

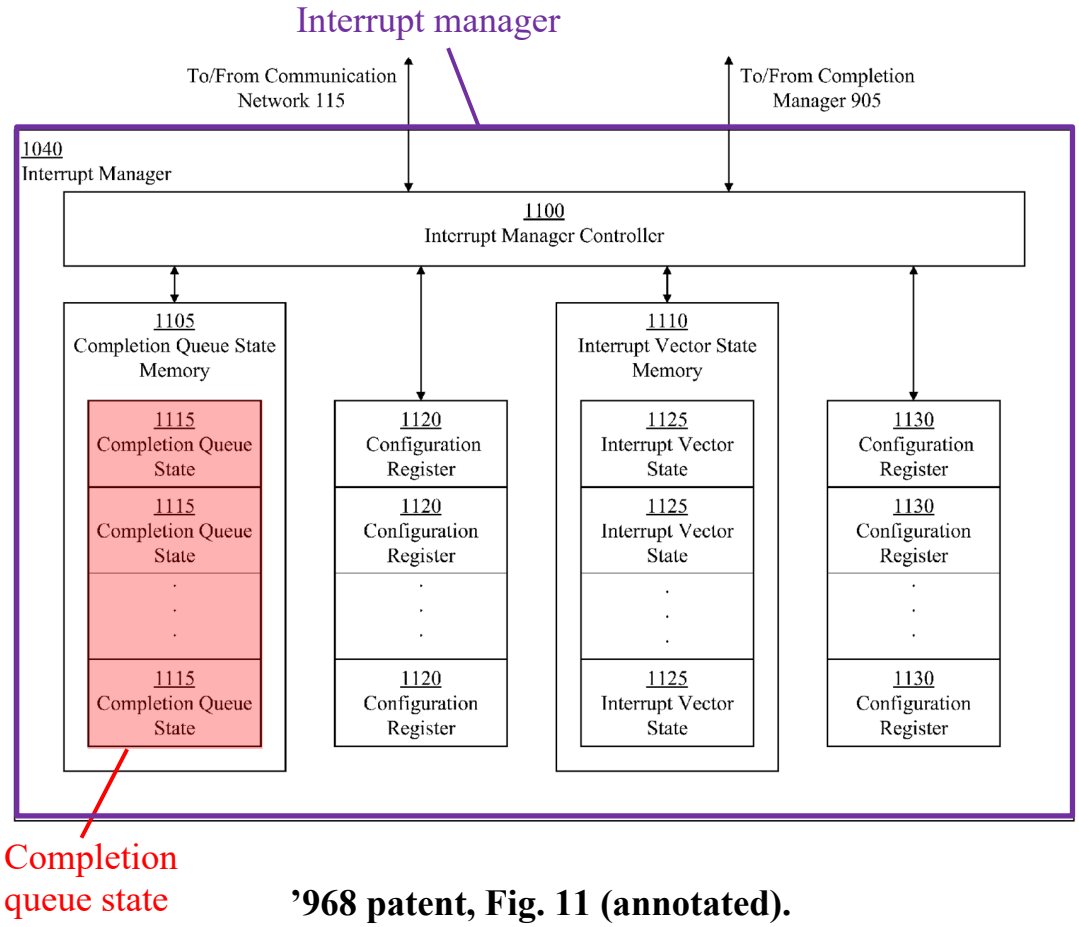
from the memory controller. After the memory controller completes a command, "the host processing unit 110 receives a request packet including a completion status from the host controller ... [and] writes the completion status into the storage location 700 at the tail 710 of the completion queue 525." '968 patent, 10:43-48. Then, when ready, "the host processing unit 110 reads completion statuses from storage locations 700 starting at the head 705 of the completion queue 525 to determine the status of the nonvolatile commands associated with the completion statuses that have been processed by the nonvolatile memory controller 105." '968 patent, 10:48-53.

47. To notify the host that there are items within the completion queue to be processed, the memory controller sends an interrupt signal to the host. The claims of the '968 patent are directed to this interrupt process. For example, claim 1 recites "*[a] nonvolatile memory controller for alerting a host processing unit to an unprocessed completion status contained in a completion queue of the host processing unit*." As explained above, the memory controller refrains from sending an interrupt until certain conditions are met.

48. To determine whether certain conditions are met, the memory controller receives status information for the completion queue. The '968 patent calls this information a "*completion queue state*." Claim 1 recites, for example, "*an interrupt manager configured to generate a completion queue state for indicating*

the occurrence of a completion queue event associated with the completion queue."

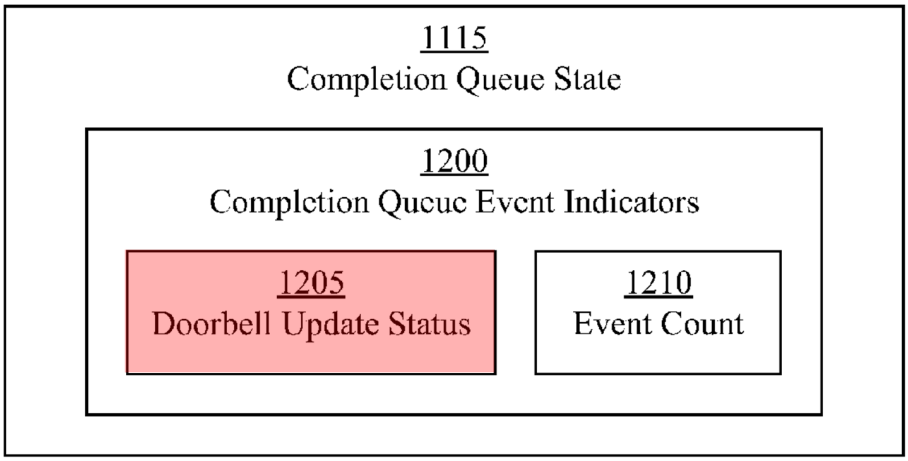
An interrupt manager 1040 that receives completion queue states 1115 is shown in Fig. 11 below.



'968 patent, Fig. 11 (annotated).

49. The completion queue states indicate the status of the completion queue. The "interrupt manager 1040 (e.g., the interrupt manager controller 1100) monitors completion queue events and generates completion queue states 1115." '968 patent, 15:13-15. Some examples of completion queue states are "completion queue event indicators 1200 [which] include a doorbell update status 1205 and an

event count 1210." '968 patent, 18:1-2. The doorbell update status 1205 is shown in Fig. 12 below.

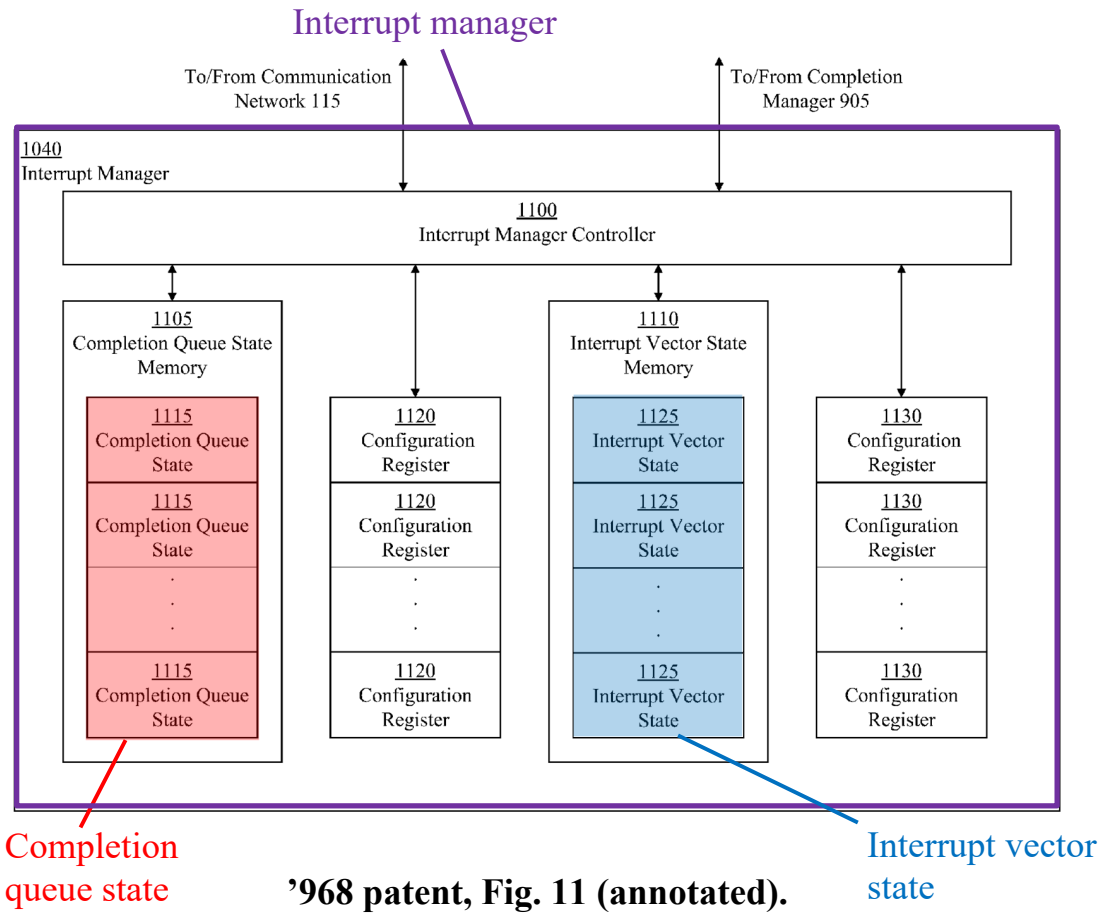


'968 patent, Fig. 12 (annotated).

50. The "doorbell update status 1205 indicates whether the host processing unit 110 has performed a doorbell update event since the last time the interrupt manager 1040 cleared the completion queue state 1115." '968 patent, 16:3-6. A doorbell update event is when the host updates the head pointer of the completion queue: "In the doorbell update event, the host processing unit 110 updates the completion queue head pointer 1025 of the completion queue 525 associated with the completion queue state 1115." '968 patent, 18:6-10. The doorbell update event concept is claimed as follows: "*and wherein the completion queue state includes a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller*

for the completion queue."

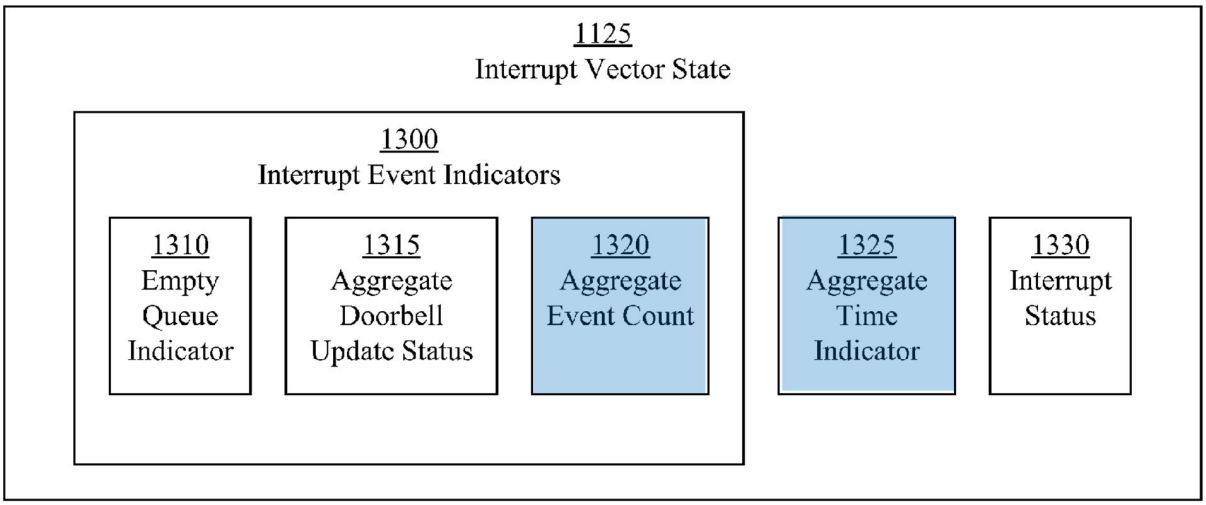
51. The status of the completion queue (i.e., completion queue state) is used to maintain the set of conditions for determining whether to generate an interrupt. Those set of conditions are referred to as the "interrupt vector state." '968 patent, 17:21-23. The claims of the '968 patent recite this concept as "*generate an interrupt vector state based on the completion queue state*." The interrupt vector state is also shown in Fig. 11.



'968 patent, Fig. 11 (annotated).

52. The '968 patent illustrates the interrupt vector state in further detail in

Fig. 13, shown below.



'968 patent, Fig. 13 (annotated).

53. As can be seen, the interrupt vector state includes items such as an aggregate event count 1320 or an aggregate time indicator 1325. The aggregate event count 1320 may be used to trigger an interrupt when the number of unprocessed completion packets sent reaches a threshold. *See* '968 patent 19:15-33. "In this way, the interrupt manager 1040 controls a transmission rate of interrupt message packets to the host processing unit 110 based on the aggregate event count threshold." '968 patent, 19:28-32. As another example, "[w]hen the aggregate time indicator 1325 is enabled, the interrupt manager 1040 controls a transmission rate of interrupt message packets to the host processing unit 110 based on the aggregate time threshold." '968 patent, 19:49-52.

54. Accordingly, the interrupt vector state is used to determine that there

are unprocessed completion events in the completion queue and that the conditions have been met for triggering an interrupt. "The interrupt manager 1040 determines based on the interrupt vector state 1125 whether any completion queue 525 associated with the interrupt vector state 1125 contains an unprocessed completion status." '968 patent, 16:3-6. "[T]he interrupt manager 1040 refrains from generating further interrupt message packets unless one or more conditions in the interrupt vector state 1125 are satisfied." '968 patent, 16:11-14. Claim 1 recites this concept as "*determine the completion queue of the host processing unit contains an unprocessed completion status based on the interrupt vector state*."

55. When the conditions are satisfied, "[t]he interrupt manager 1040 transmits the interrupt message packet to the host processing unit 110 through the communication network 115." '968 patent, 16:18-20. This concept is recited in claim 1 as "*generate an interrupt message packet for triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue*." Thus, by waiting until certain conditions have been met before sending more interrupt signals, the interrupt manager 1040 performs a technique known as interrupt coalescing.

56. However, the concept of interrupt coalescing, as well as the other concepts recited in the claims of the '968 patent, were already known as of the time of filing.

B. Prosecution History of the '968 Patent.

57. The '968 patent was filed on March 21, 2011 and claims priority to a provisional application filed August 16, 2010.

58. In a first Office Action, the USPTO rejected several claims based on 25 U.S.C. 112 as being indefinite. Ex.1002, 115-116. There were no prior art rejections. In response, the Applicant amended the claims to address the Office's 112 rejection. Ex.1002, 137-142.

59. While the Office indicated that some of the claims were allowed, the Office also stated that "the examiner is unable to find basis for the claimed subject matter in the provisional application. Applicant is invited to point out where the provisional application provides a basis for the claimed subject matter in order to receive the benefit of the filing date of the provisional." Ex.1002, 116. The Applicant declined the Office's invitation to point out where support was provided in the provisional application.[1] Ex.1002, 143-145.

60. The Office also issued a requirement for information under 37 CFR 1.105. Ex.1002, 116. The Office stated:

[1] For purposes of my declaration, I do not address whether there is support in the provisional application because the prior art I rely on predates the provisional application.

> It is noted that the NVM Express specification generally shows the subject matter claimed in at least the independent claims in section 7.5.1.1.1. However the examiner has only been able to locate Revision 1.0b of the specification which is dated July 12, 2011. This document does not indicate what changes were made between version 1.0 and 1.0b. Version 1.0 was published March 1, 2011 which does pre-date the application filing date.

Ex.1002, 116.

> In response to this requirement, please provide the title, citation and copy of each publication that any of the applicants relied upon to draft the claimed subject matter. For each publication, please provide a concise explanation of the reliance placed on that publication in distinguishing the claimed subject matter from the prior art.
>
> In particular, applicant's specification makes reference to the Nonvolatile Memory Express (NVMe) standard. It appears that the assignee is a "Promoter" of the NVM Express standard and a permanent member of the Promoters Group. The claimed subject matter is related to various elements of the NVM Express standard as noted above with respect to the NVM Express Revision 1.0b. It is presumed that applicant had access to the initial published version of the standard (NVM Express Revision 1.0) which predates applicant's filing date. Applicant should provide at least relevant portions of that document as well as any others relied upon in drafting the claims as indicated above.

Ex.1002, 117.

61. The Applicant responded to this request by stating that "the

Examiner's request for information is overly broad and that information regarding drafting of the NVM Express standard should be required **as it is not material to patentability**." Ex.1002, 144 (emphasis added). The Applicant nevertheless filed an IDS that included a copy of the NVM Express standard revision 1.0, published on March 1, 2011. Ex.1002, 144. The Applicant further stated that:

> Applicant is submitting separately, via USPS, drafts 056 (dated 6/3/10) and 086 (dated 11/4/10) of the NVM Express standard, and email correspondence relating to drafting of the NVM Express standard. This material is provided pursuant to MPEP 724.02 and constitutes proprietary information. Applicant asserts that the material provided pursuant to MPEP 724.02 is not material to patentability. Accordingly, Applicant requests that the material provided pursuant to MPEP 724.02 be maintained as proprietary information and not be included in the case file. Moreover, after this information has been reviewed by the Examiner, and after determination by the Examiner that the information is not material to patentability, Applicant requests that the material provided pursuant to MPEP 724.02 be returned to Applicant

Ex.1002, 144.

62. The amendments to the claims and the submission of information appears to have satisfied the Examiner. The Examiner then issued a notice of allowance on June 13, 2013. Ex.1002, 151. The stated reasons for allowance were that the claims include "limitations of the details of the operation of the interrupt manager's operation which are not shown in the prior art." Ex.1002, 157.

63. However, for the reasons explained in my analysis below, the prior art does indeed describe the interrupt coalescing techniques described in the ’968 patent.

VII. CLAIM CONSTRUCTION

64. It is my understanding that in order to properly evaluate the ’968 patent, the terms of the claims must first be interpreted. It is my understanding that for the purposes of this *inter partes* review, the claims are to be construed under the so-called *Phillips* standard, under which claim terms are given their ordinary and customary meaning as would be understood by one of ordinary skill in the art in light of the specification and prosecution history, unless the inventor has set forth a special meaning for a term. It is my opinion that for the purposes of this proceeding, no term requires express construction.

VIII. IDENTIFICATION OF HOW THE CLAIMS ARE UNPATENTABLE

65. I have been asked to provide my opinion as to whether the Challenged Claims of the ’968 patent would have been obvious in view of the prior art. The discussion below provides a detailed analysis of how the prior art references identified below teach the limitations of the Challenged Claims of the ’968 patent.

66. As part of my analysis, I have considered the scope and content of the prior art and any differences between the alleged invention and the prior art. I describe in detail below the scope and content of the prior art, as well as any

differences between the alleged invention and the prior art, on an element-by-element basis for each Challenged Claims of the ’968 patent.

67. As described in detail below, the alleged invention of the Challenged Claims would have been obvious in view of the teachings of the identified prior art references as well as the knowledge of a POSITA.

A. Ground 1: Claims 1-18 are obvious in view of Borchers.

1. Summary of Borchers.

68. Borchers, like the ’968 patent, describes techniques for interrupt coalescing. Borchers describes communication between a host 106 and a controller board 102 of a data storage device 100, as shown in Fig. 1 below.

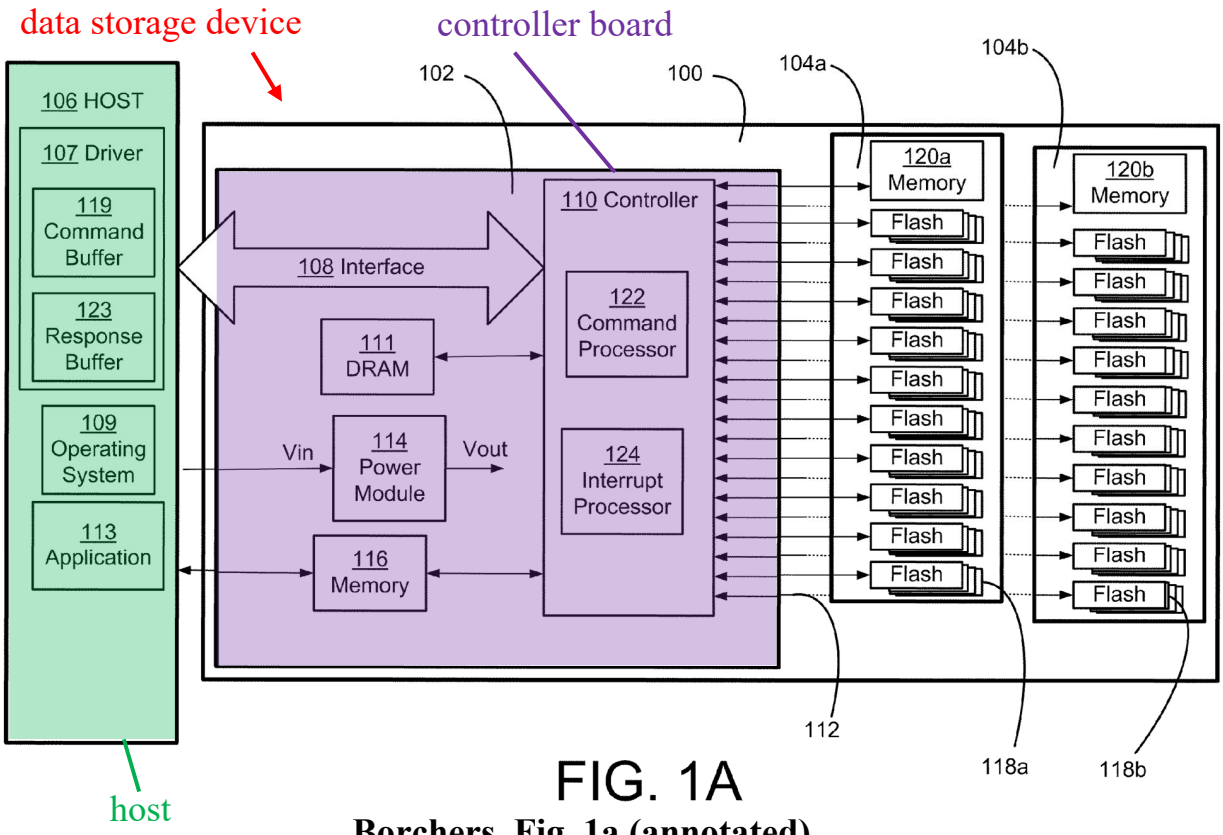
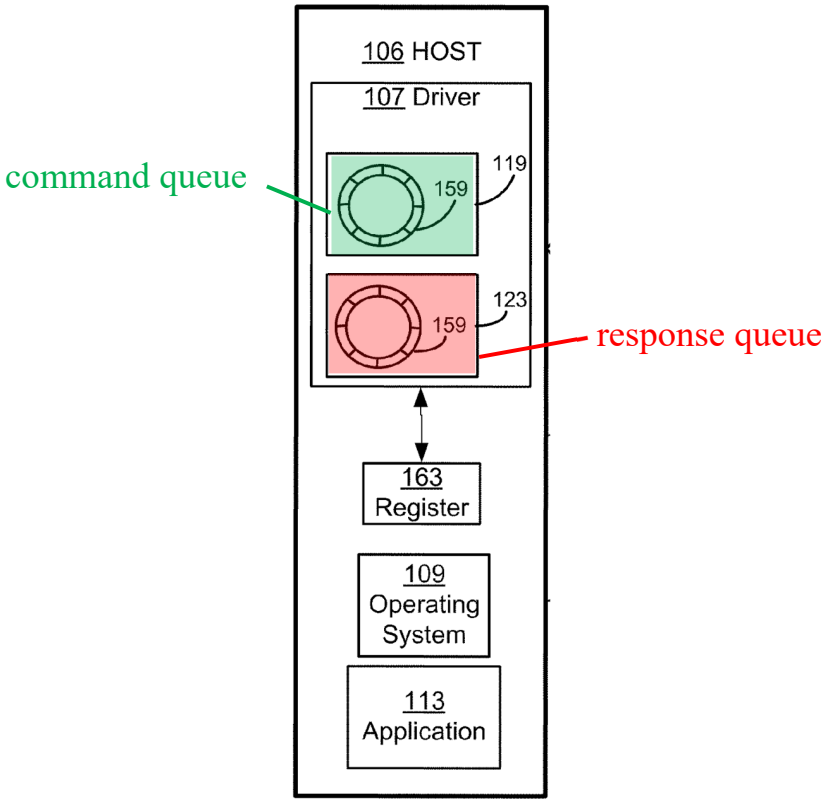


FIG. 1A

Borchers, Fig. 1a (annotated).

69. Like the '968 patent, the host includes two queues: a command queue 119 and a response queue 123. The command buffer 119 "is populated with commands for retrieval by the data storage device." Borchers, [0004]. The response buffer 123 "is populated with responses by the data storage device for retrieval by the host device." Borchers, [0004]. "[T]he command buffer 119 can be configured as a circular queue 159 that is used to communicate information from the host 106 and to the data storage device 100 of FIG. 1A. The response buffer 123 also can be configured as a circular queue." Borchers, [0030].



Borchers, Fig. 1c (partial, annotated).

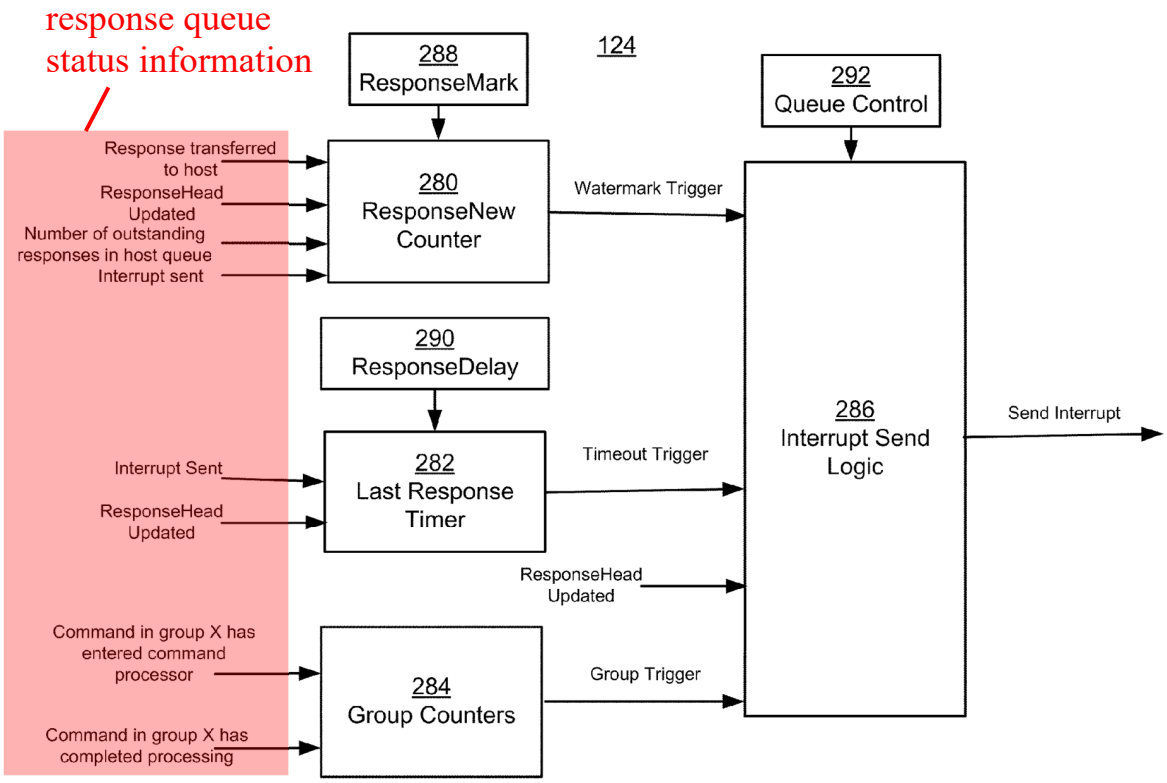
70. Borchers explains that "Each of the circular queues 159 of the command buffer 119 and the response buffer 123 include a head pointer and a tail pointer." Borchers, [0030]. "Commands and responses may be inserted into the circular queue 159 at the tail pointer and removed at the head pointer." For example, when the host seeks to read or write to memory, "[t]he host 106 may write a command to the command buffer 119 and update the command tail pointer." Borchers, [0034]. Then, when the memory controller is ready to process those commands, it will "read the command out of the command buffer 119 [and] update the head pointer ... to indicate to the host 106 that the command processor

122 has received the command." Borchers, [0034].

71. After the memory controller has processed a command, it sends a response command back to the host. It does so by writing "a response to the host" at the tail end of the response queue. Borchers, [0035]. The memory controller then "updates the response tail pointer ... to indicate that the command is finished." Borchers, [0035].

72. In some examples, "[t]he interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123." Borchers, [0035]. However, in some cases, the controller may refrain from sending an interrupt until certain conditions have been met. In particular, "the host 106, through its driver 107, may control when the interrupt processor 124 should generate interrupts." Borchers, [0036].

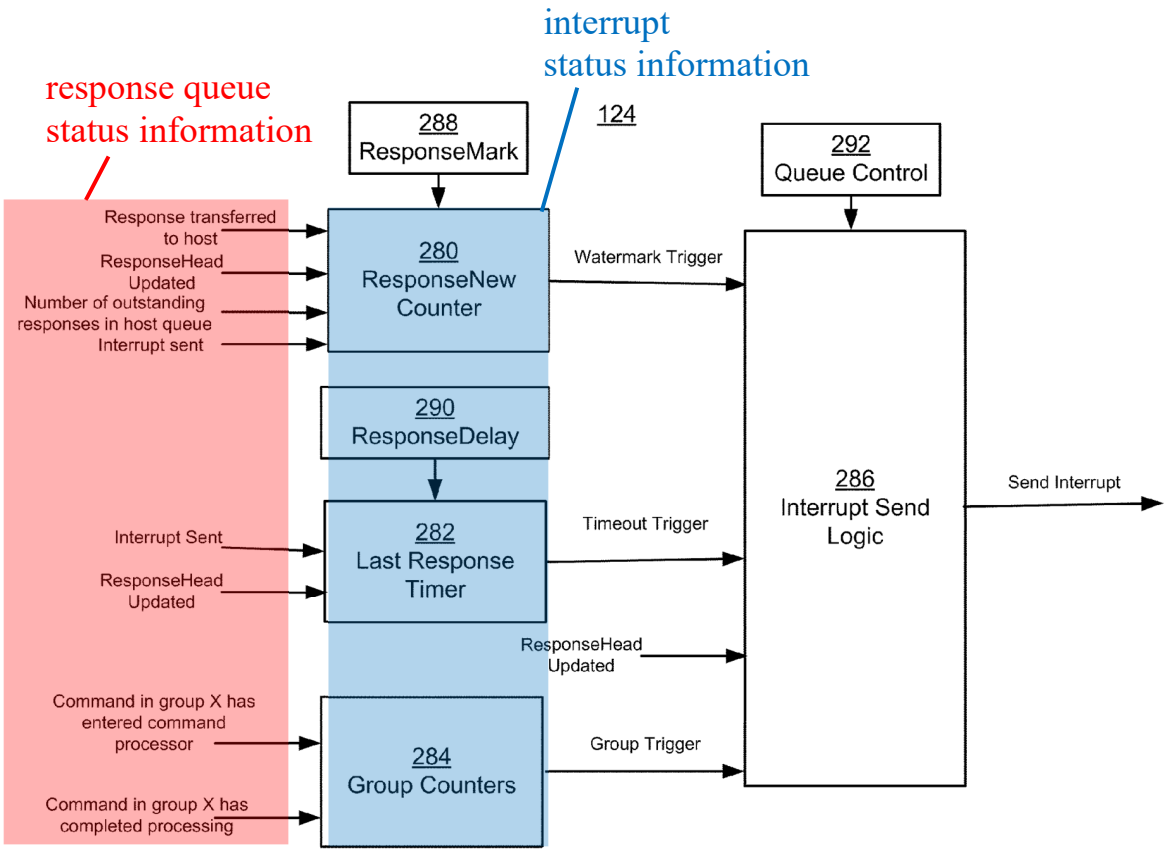
73. To determine whether to send interrupts, the memory controller receives information that indicates the status of the response queue. This response queue status information may include "when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent." Borchers, [0060]. Determining whether the ResponseHead (head pointer of the response queue) has been updated is sometimes referred to as a doorbell status update. *See e.g.,* PCI, 308.



Borchers, Fig. 2 (annotated).

74. The response queue status information can then be used to maintain the status of interrupt mechanisms, which keep track of the conditions used to trigger an interrupt. For example, the response queue status information can be used to update a ResponseNew Counter 280. "The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process." Borchers, [0060]. "When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286." Borchers, [0060]. Borchers describes other interrupt mechanisms such as a timer 282 or a

group counter 284. As illustrated in Fig. 2 below, these interrupt mechanisms maintain information as to the status of the conditions associated with these interrupt mechanisms.



Borchers, Fig. 2 (annotated).

75. The interrupt status information is then used to trigger an interrupt. For example, when the ResponseNew Counter 280 reaches a threshold or "watermark," this triggers an interrupt. "[W]hen the watermark is reached, then the interrupt send logic 286 generates and sends an interrupt to the host 106." Borchers, [0061]. In the case where the timer mechanism is used, the "last response timer 282 is configured to generate a timeout trigger when the amount of

time being tracked by the last response timer 282 is greater than the ResponseDelay parameter 290." Borchers, [0064].

76. Accordingly, Borchers describes the same interrupt coalescing technique described in the '968 patent. Borchers uses the state of the response queue ("*completion queue state*") to update interrupt status information ("*interrupt vector state*"). That interrupt status information keeps track of the conditions for triggering an interrupt signal. When those conditions are met (i.e., a watermark or expired timer), then the interrupt signal is sent.

2. The PCI Standard Represents Background Knowledge of a POSITA.

77. The PCIe Standard represents the background knowledge of a POSITA because its subject matter is critical for designing memory systems that are compliant with industry standards. Given that the PCIe standard is "an enhancement to the PCI architecture" which was "the most widely accepted and implemented expansion standard in the world" as of 2004 (PCIe, 21; PCI, 15), POSITAs in the field of memory systems would have been familiar with its teachings and would have sought to design their memory systems in accordance with such standards.

3. Claim 1

[1.0] ***A nonvolatile memory controller for alerting a host processing unit to an unprocessed completion status contained in a completion queue of the host processing unit, the nonvolatile memory controller comprising:***

78. The language that comes after the phrase "*A nonvolatile memory controller for* …" of the preamble [1.0] describes a purpose, or intended use, of the controller. To the extent that the preamble is deemed limiting, Borchers teaches the entire preamble

79. **First**, Borchers describes a controller board 102 ("*non-volatile memory controller*") for a data storage device 100 that stores data for a host 106 ("*host processing unit*").

> Referring to FIG. 1A, a block diagram of a system for processing and tracking commands in a group is illustrated. **FIG. 1A illustrates a block diagram of a data storage device 100 and a host 106**. **The data storage device 100 may include a controller board 102 and one or more memory boards 104 a and 104 b**. The data storage device 100 may communicate with the host 106 over an interface 108. The interface 108 may be between the host 106 and the controller board 102. **The controller board 102 may include a controller 110, a DRAM 111, multiple channels 112, a power module 114, and a memory module 116**. The controller 110 may include a command processor 122 and an interrupt processor 124, as well as other components, which are not shown. The memory boards 104 a and 104 b may include multiple flash memory chips 118 a and 118 b on each of the memory boards. The memory boards 104 a and 104 b also may include a memory device 120 a and 120 b, respectively.

Borchers, [0018]-[0019].

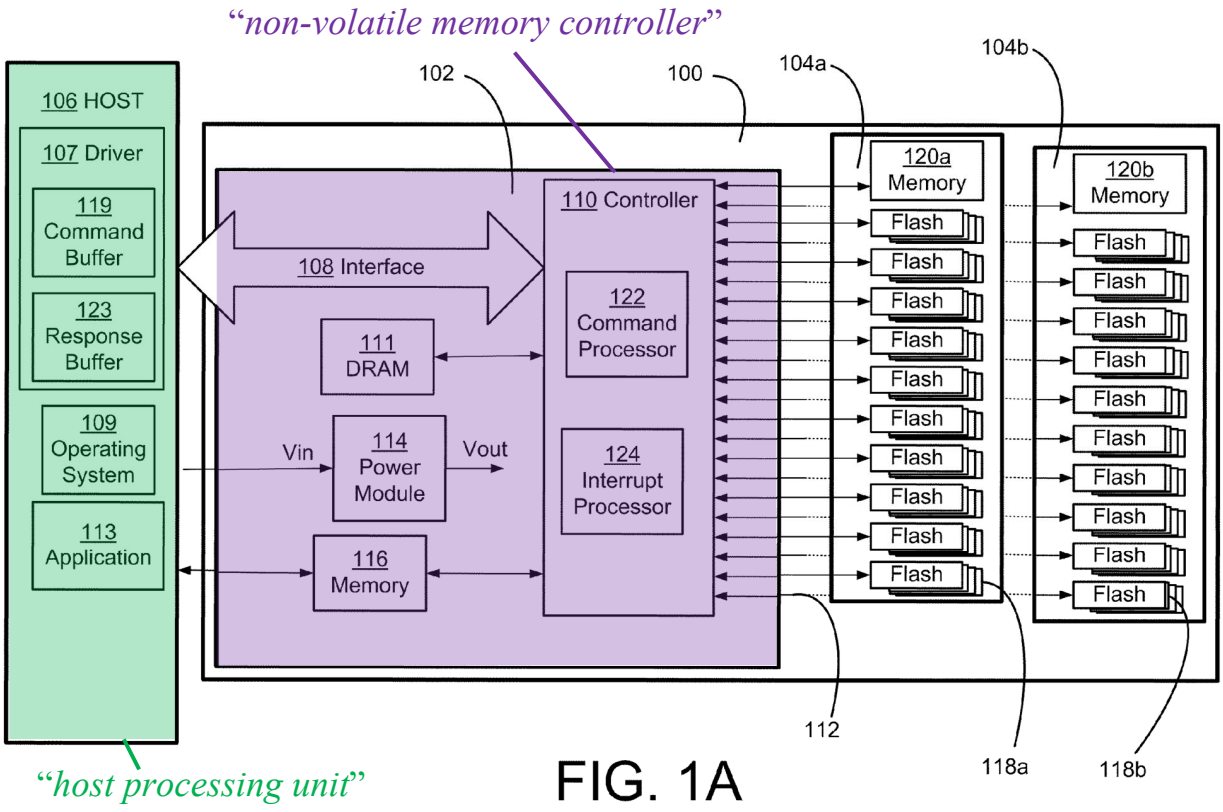


FIG. 1A

Borchers, Fig. 1a (annotated).

80. Borchers' controller board 102 is a non-volatile memory controller because it is responsible for controlling flash memory. Borchers, [0019]. Fig. 1A illustrates the memory devices 104a as flash memory and explains that "the memory boards 104a and 104b may include multiple flash memory chips 118a and 118b on each of the memory boards." Borchers, [0019]; *see also* Borchers, [0125]. Flash memory is described as a non-volatile type of memory. Borchers, [0128] ("**forms of non-volatile memory**, including by way of example semiconductor memory devices, e.g., EPROM, EEPROM, and **flash memory devices**."). Accordingly, because the controller board 102 serves as a controller for non-

volatile flash memory, it is a "*non-volatile memory controller*" as claimed.

81. Borchers' host 106 is a "*host processing unit*" because it processes instructions from an application.

> The host 106 may include a driver 107, an operating system 109 and one or more applications 113. In general, the host 106 may generate commands to be executed on the data storage device 100. For example, the application 113 may be configured to generate commands for execution on the data storage device 100. **The application 113 may be operably coupled to the operating system 109 and/or to the driver 107. The application 113 may generate the commands and communicate the commands to the operating system 109**. The operating system 109 may be operably coupled to the driver 107, where the driver 107 may act as an interface between the host 106 and the data storage device 100. In other exemplary implementations, the application 113 may communicate directly with the data storage device 100, as discussed below with respect to FIG. 1B.

Borchers, [0020].

82. **Second**, the host 106 ("*host processing unit*") includes a response buffer 123[2] ("*completion queue*").

> In one exemplary implementation, the host 106 using the driver 107 and the data storage device 100 may communicate commands and

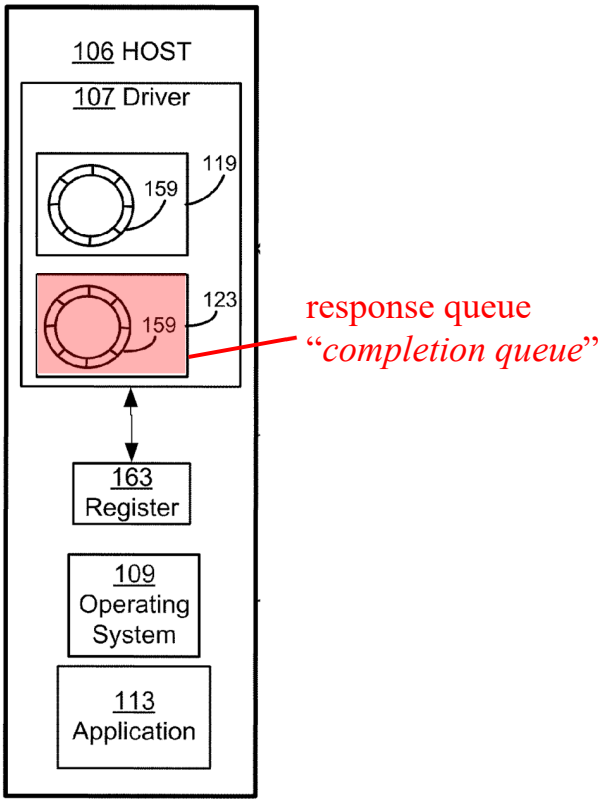
[2] In, Borchers, "the terms **buffer** and **queue** are used interchangeably." Borchers, [0027].

> responses using pairs of queues or buffers in host memory. Throughout this document, the terms buffer and queue are used interchangeably. **For example, a command buffer 119 may be used for commands and a response buffer 123 may be used for responses or results to the commands**. In one exemplary implementation, the commands and results may be relatively small, fixed size blocks. For instance, the commands may be 32 bytes and the results or responses may be 8 bytes. In other exemplary implementations, other sized blocks may be used including variable size blocks. Tags may be used to match the results to the commands. In this manner, the data storage device 100 may complete commands out of order.

Borchers, [0027].

83. The response queue 123 may be "configured as a circular queue."

Borchers, [0030]. Fig. 1C (partial) is shown below.



Borchers, Fig. 1c (partial, annotated).

84. The response queue 123, like the claimed completion queue, receives completion responses from the memory controller board 102. "For example, a command buffer 119 may be used for commands and **a response buffer 123 may be used for responses or results to the commands**." Borchers, [0027].

> When the command processor 122 **completes** the command, **the command processor 122 writes a response to the host using a DMA operation and updates the response tail pointer** with another DMA operation to indicate that the command is finished. The interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123.

Borchers, [0035].

> The response buffer 123 also can be configured as a circular queue. Each of the circular queues 159 of the command buffer 119 and the response buffer 123 include a head pointer and a tail pointer. Values of the head pointer of the circular queue 159 of the command buffer 119 can be stored in a register 163 on the host, and values of the tail pointer can be stored in a register 161 on the data storage device 100. Values of a tail pointer of a circular queue of the response buffer 123 can be stored in a register on the host, and values of the head pointer of the response buffer can be stored in a register on the data storage device 100. Commands and responses may be inserted into the circular queue 159 at the tail pointer and removed at the head pointer. **The host 106 may be** the producer of the command buffer 119 and **the consumer of the response buffer 123**. **The data storage device 100 may be** the consumer of the command buffer 119 and the **producer of the response buffer 123**. The host 106 may write the command tail pointer and the response head pointer and may read the command head pointer and the response tail pointer. **The data storage device 100 may write the** command head pointer and the **response tail pointer** and may read the command tail pointer and the response head pointer. In the data storage device 100, the controller 110 may perform the read and write actions. More specifically, the command processor 122 may be configured to perform the read and write actions for the data storage device 100. No other synchronization, other than the head and tail pointers, may be needed between the host 106 and the data storage device 100.

Borchers, [0030].

85. Accordingly, because the response queue stores the notifications of completed commands, it is a "*completion queue*" as claimed. Borchers' response queue performs a similar function as does the completion queue described in the '968 patent. *Compare* Borchers, [0030]-[0035] *with* '968 patent, 10:43-53.

86. **Third**, the non-volatile memory controller is configured to alert the host processing unit when there are new (i.e., unprocessed) events in the response queue ("*configured to alert a host processing unit to an unprocessed completion status contained in a completion queue*").

> When the command processor 122 completes the command, the command processor 122 writes a response to the host using a DMA operation and updates the response tail pointer with another DMA operation to indicate that the command is finished. **The interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123**. The host 106 is configured to read the responses from the response buffer 123 and update the head pointer in the MMIO space of the data storage device to indicate that the host has received the response. In one exemplary implementation, the interrupt processor 124 may not send another interrupt to the host 106 until the previous interrupt has been acknowledged by the host 106 writing to the response head pointer. Thus, receiving a response to the writing of a command can include just one write operation to memory mapped input/output space (i.e., the updating of the head pointer by the host) and two DMA events (i.e., the

> writing of the response by the command processor and the updating of the response tail pointer to indicate that the command is finished). Neither the writing of the command nor the reception of the response involves a MMIO read event, which can take a relatively long time compared to MMIO write events and DMA events, and in this manner the communication between the host and the device is accelerated.

Borchers, [0035].

87. Thus, because Borchers describes a non-volatile memory controller that alerts a host when there are new items in a response queue, Borchers renders obvious a *"nonvolatile memory controller for alerting a host processing unit to an unprocessed completion status contained in a completion queue of the host processing unit*" as claimed.

[1.1] ***an interrupt manager configured to generate a completion queue state for indicating the occurrence of a completion queue event associated with the completion queue,***

88. **First**, Borcher's memory controller board 102 includes an interrupt processor 124 ("*interrupt manager*").

> **The controller board 102 may include a controller 110**, a DRAM 111, multiple channels 112, a power module 114, and a memory module 116. **The controller 110 may include** a command processor 122 and **an interrupt processor 124**, as well as other components, which are not shown. The memory boards 104 a and 104 b may include multiple flash memory chips 118 a and 118 b on each of the memory

boards. The memory boards 104 a and 104 b also may include a memory device 120 a and 120 b, respectively.

Borchers, [0019].

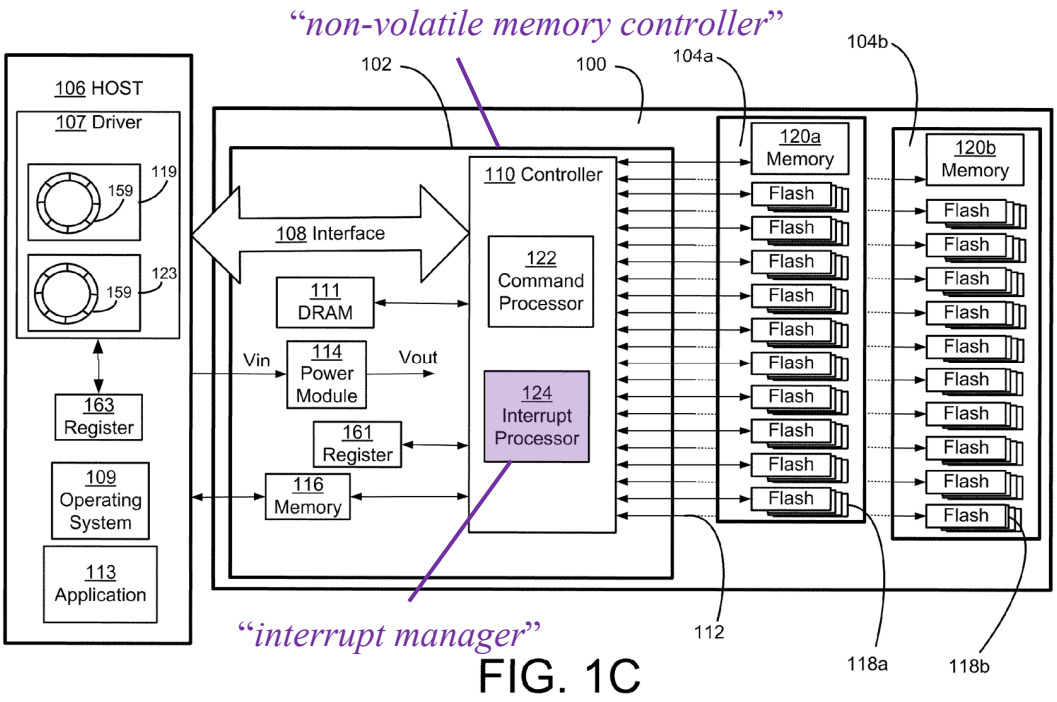
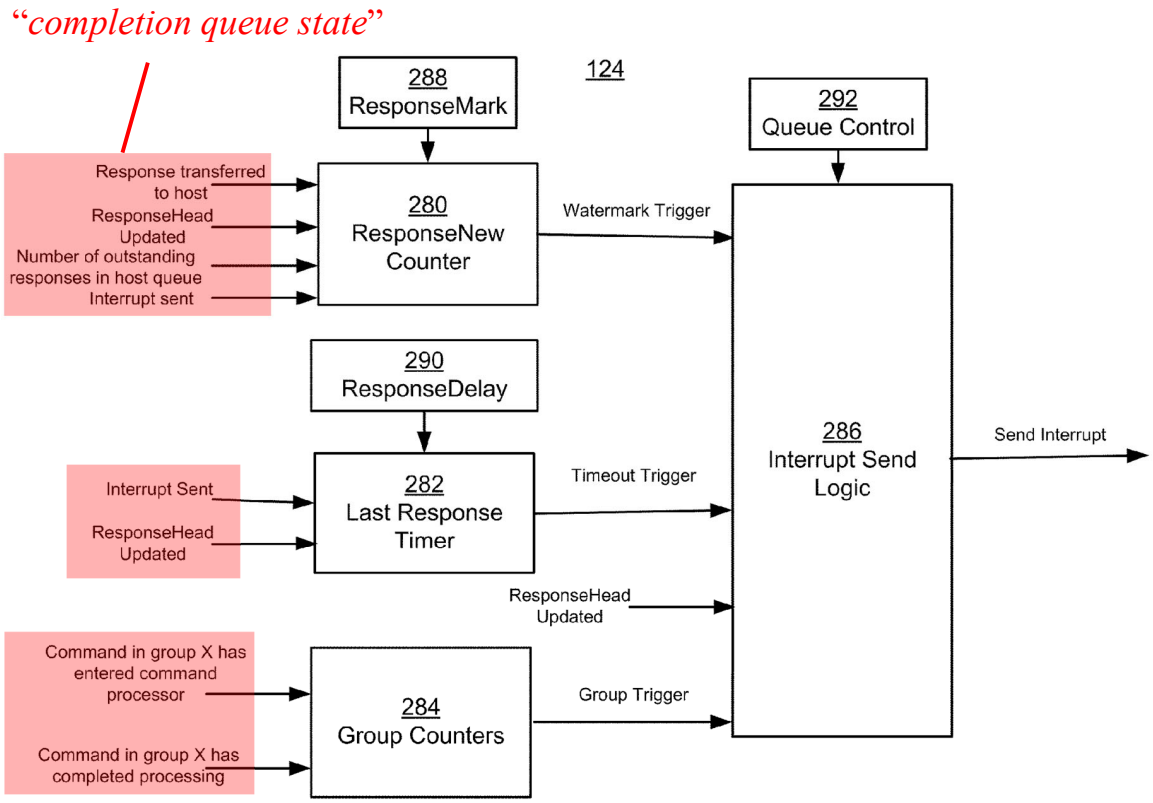


FIG. 1C

Borchers, Fig. 1C (annotated).

89. **Second**, Borchers' interrupt processor 124 (shown in detail in Fig. 2) receives information from the host system as to the status of the response queue ("*completion queue*"). To "*indicate[] the occurrence of a completion queue event associated with the completion queue,*" the response queue status information includes an updated response queue head pointer, and the number of outstanding items in the response queue.



Borchers, Fig. 2 (annotated).

The ResponseNew counter 280 may be enabled by the host 106 when the watermark interrupt mechanism is desired. The host 106 may set the ResponseMark 288, which is a parameter provided as input to the ReponseNew counter 280, as discussed above. **The ResponseNew counter 280 receives as inputs information including when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process**. Each time a response is transferred to the response buffer 123 the counter is incremented. When the counter 280 reaches or exceed the watermark

> level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].

90. Accordingly, because the updated head pointer and number of outstanding responses indicates the status of the response queue ("*completion queue*") that information is a "*completion queue state*" as claimed. In other words, the updated head pointer and number of outstanding responses "*indicat[e] the occurrence of a completion queue event associated with the completion queue*.

91. The status information received and relied upon by the interrupt controller 124 "*indicates the occurrence of a completion queue event associated with the completion queue*" as claimed. For example, an update to the head pointer of the response queue is a "*completion queue event associated with the completion queue*" because it is an event that directly affects and relates to the response queue ("*completion queue*").

92. Further, the information received by the interrupt processor (i.e., "when the ResponseHead is updated" or "the number of outstanding responses in the host response buffer") represents a "*completion queue* ***state***" because it

represents the *status* of the completion queue.

93. This status information is "*generate[d]*" by the interrupt processor when it is received and/or stored by Borchers' interrupt processor 124 as illustrated in Fig. 2. A POSITA would have found it obvious for Borchers' status information to be stored in some type of memory such as a buffer, register, flag, or other form of storage. In other words, a POSITA would have recognized that Borchers' response queue status information ("*completion queue state*") is not merely transitory signals. Rather, a POSITA would have recognized that the response queue status information would be stored somewhere by the interrupt processor until such information was needed or ready to be processed. Indeed, it was known for memory controllers and interrupt managers to use such types of storage. *See e.g.,* Serebrin, Figs. 2 & 5, [0057], [0090]-[0091]; Serebrin Provisional, Figs. 2 & 5, [0051], [0083], [0084]] (describing how an interrupt manager uses various forms of memory such as cache or registers to store state information). And a POSITA would have found it obvious for the response queue status information to be generated/stored by the interrupt processor given that the ResponseHead being updated is on the controller side: "The command tail pointer and the response head pointer may be stored in registers of the data storage device." Borchers, [0031]. Accordingly, because Borchers' interrupt processor receives and uses the response queue status information, the interrupt processor "*generate[s]*" a "*completion*

queue state" as claimed.

94. Thus, because Borchers' interrupt manager receives and uses information as to the status of the response queue, Borchers renders obvious "*an interrupt manager configured to generate a completion queue state for indicating the occurrence of a completion queue event associated with the completion queue*" as claimed.

[1.2] ***generate an interrupt vector state based on the completion queue state,***

95. **First**, Borchers' interrupt processor generates an "*interrupt vector state"* by maintaining a set of conditions that determine whether to send an interrupt signal. These set of conditions are maintained by various interrupt mechanisms including a ResponseNew counter 280, a last response timer 282, and group counters 284.

> Referring to FIG. 2, a block diagram of an exemplary interrupt processor 124 is illustrated. **The interrupt processor 124 may be configured to generate and send interrupts based on the interrupts mechanism** or mechanisms enabled by the host 106. **The interrupt processor 124 may include a ResponseNew counter 280, a last response timer 282, group counters 284 and interrupt send logic 286**.

Borchers, [0059].

96. The ResponseNew counter 280, for example, tracks the number of responses sent to the host that have yet to be processed. When that number exceeds

a threshold or "watermark level," an interrupt is sent. If the count is less than the threshold, an interrupt is not sent.

> **The ResponseNew counter 280** may be enabled by the host 106 when the watermark interrupt mechanism is desired. The host 106 may set the ResponseMark 288, which is a parameter provided as input to the ReponseNew counter 280, as discussed above. **The ResponseNew counter 280 receives as inputs information including when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process**. Each time a response is transferred to the response buffer 123 the counter is incremented. **When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286**. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].

97. The last response timer 282, for example, tracks the amount of time since the last interrupt. After a certain amount of time, an interrupt is sent.

> **The last response timer 282** may be enabled when the timer interrupt mechanism is desired. The last response timer 282 may be configured to keep track of time since the last interrupt. For instance, the last response timer 282 may track the amount of time since the last interrupt in microseconds. The host 106 may set the amount of time using a parameter, for example, a Response Delay parameter 290. **In one exemplary implementation, the Response Delay 290 timeout may be the number of microseconds since the last interrupt, or since the last time that the host 106 removed new responses from the response buffer 123, before an interrupt is generated**.

Borchers, [0062].

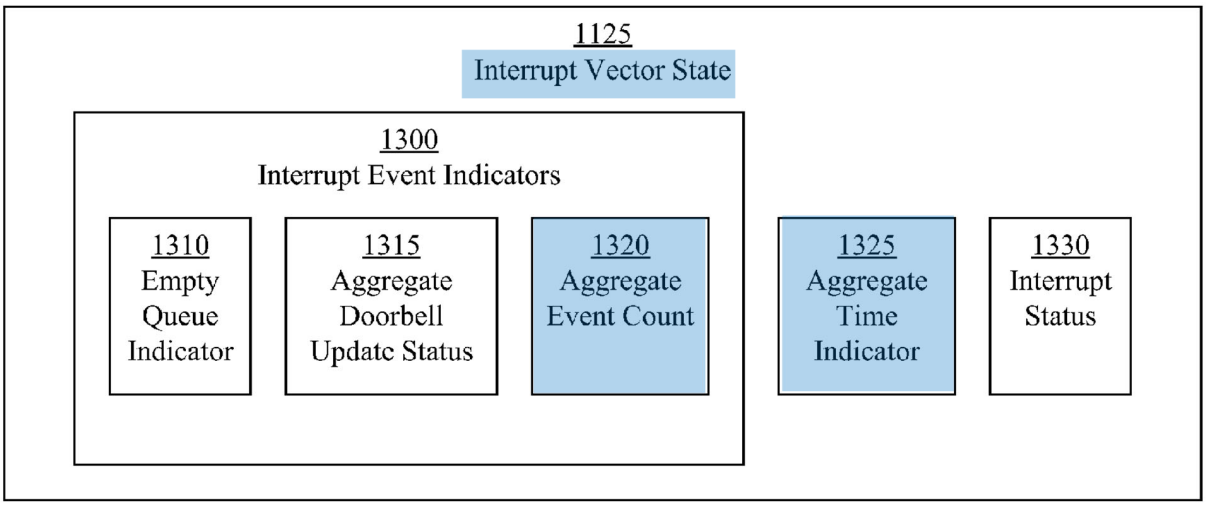
98. The group counters 284 track the number of unprocessed responses for a number of response queues so that an interrupt can be sent for the whole group.

> **The group counters 284** mechanism may be arranged and configured to track commands that are part of a group as designated by the driver 107. The group counters 284 may be enabled by the host 106 when the host 106 desires to track commands as part of a group such that a single interrupt is generated and sent back to the host 106 only when all of the commands in a group are processed. **In this manner, an interrupt is not generated for each of the individual commands but only for the group of commands**.

Borchers, [0066].

99. Accordingly, because the interrupt processor generates counters and

timers for determining whether to send an interrupt signal, Borchers' interrupt processor is configured to generate an "*interrupt vector state*" as claimed. Notably, the interrupt mechanisms of Borchers, which include timers and counters, are similar to how the '968 patent describes the interrupt vector states.



'968 patent, Fig. 13 (annotated).

100. As can be seen from Fig. 13 of the '968 patent, the interrupt vector state includes items such as an aggregate event count 1320 or an aggregate time indicator 1325. The aggregate event count 1320 may be used to trigger an interrupt when the number of unprocessed completion packets sent reaches a threshold. *See* '968 patent 19:15-33. "In this way, the interrupt manager 1040 controls a transmission rate of interrupt message packets to the host processing unit 110 based on the aggregate event count threshold." '968 patent, 19:28-32. As another example, "[w]hen the aggregate time indicator 1325 is enabled, the interrupt

manager 1040 controls a transmission rate of interrupt message packets to the host processing unit 110 based on the aggregate time threshold." '968 patent, 19:49-52.

101. Thus, Borchers' ResponseNew counter 280 and last response timer 282 are analogous to the aggregate event count 1320 and aggregate time indicator 1325 which are examples of interrupt vector states described in the '968 patent.

102. **Second**, Borchers' interrupt mechanisms (e.g., ResponseNew counter 280, last response timer 282, and group counters) teach an "*interrupt* ***vector*** *state*" because it would have been obvious for Borchers' interrupt signals to be implemented as interrupt vectors. Borchers explains that its memory interface "may include a PCIe interface." It was known that the PCI Express standard was required to support Message Signaled Interrupts (MSI), which use interrupt vectors. *See* PCIe.

> **PCI Express requires support of MSI or MSI-X or both**. The PCI Express MSI and MSI-X mechanisms are compatible with those defined 15 in the PCI Local Bus Specification, Revision 3.0.

PCIe, 341.

> **Message Signaled Interrupts (MSI)** is an optional feature that enables a device function to request service by writing a system-specified data value to a system-specified address (using a PCI DWORD memory write transaction). System software initializes **the message address and message data (from here on referred to as the "vector")** during device configuration, **allocating one or more vectors to each MSI capable function**.

PCI, 231.

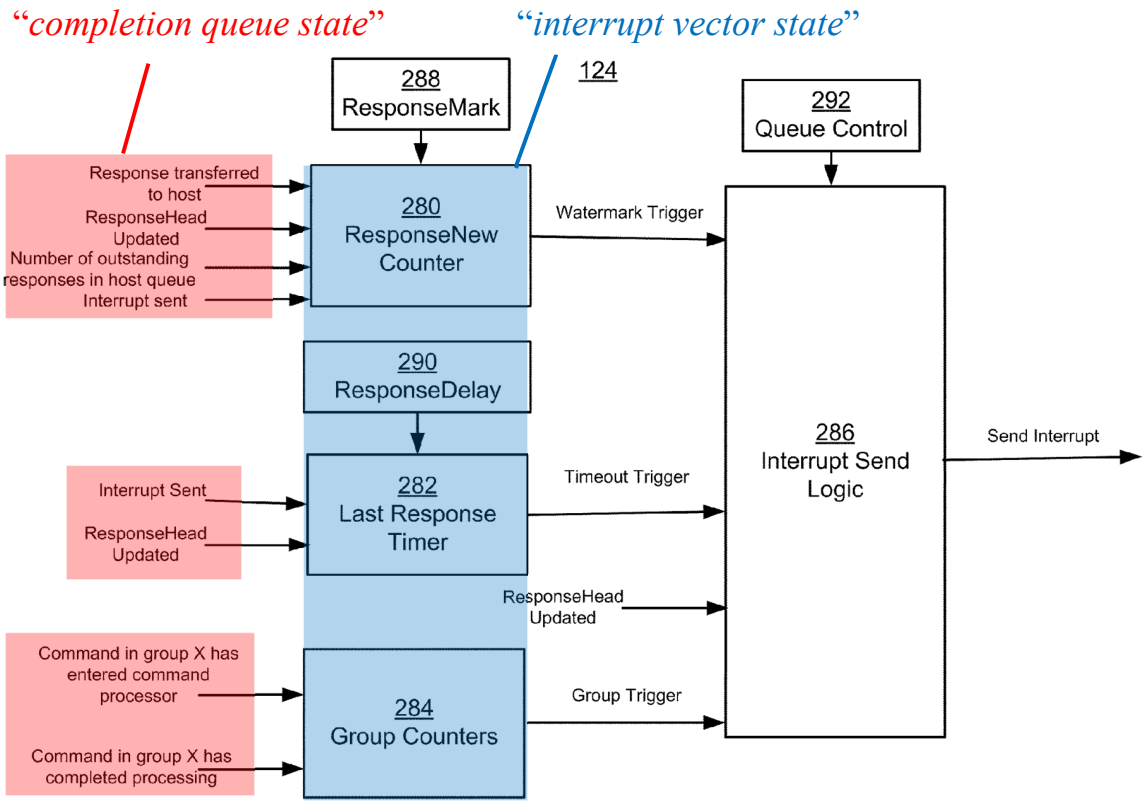
103. A POSITA would have thus recognized based on background knowledge that Borchers' interrupt signals—as part of a PCIe interface—would be MSI signals that use interrupt vectors. Accordingly, the interrupt mechanisms that maintain the set of conditions for determining whether to send an interrupt signal teach an "*interrupt vector state*" as claimed. Again, Borchers' interrupt mechanism are analogous to the interrupt vector state described in the '968 patent. *See* '968 patent 19:15-52.

104. Third, Borchers' interrupt mechanisms (timers and counters) ("*interrupt vector state*") are based on the response queue status information ("*completion queue state*").

> The ResponseNew counter 280 may be enabled by the host 106 when the watermark interrupt mechanism is desired. The host 106 may set the ResponseMark 288, which is a parameter provided as input to the ReponseNew counter 280, as discussed above. **The ResponseNew counter 280 receives as inputs information including when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process**. Each time a response is transferred to the response buffer 123 the counter is incremented. When the counter 280 reaches or exceed the watermark

> level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].



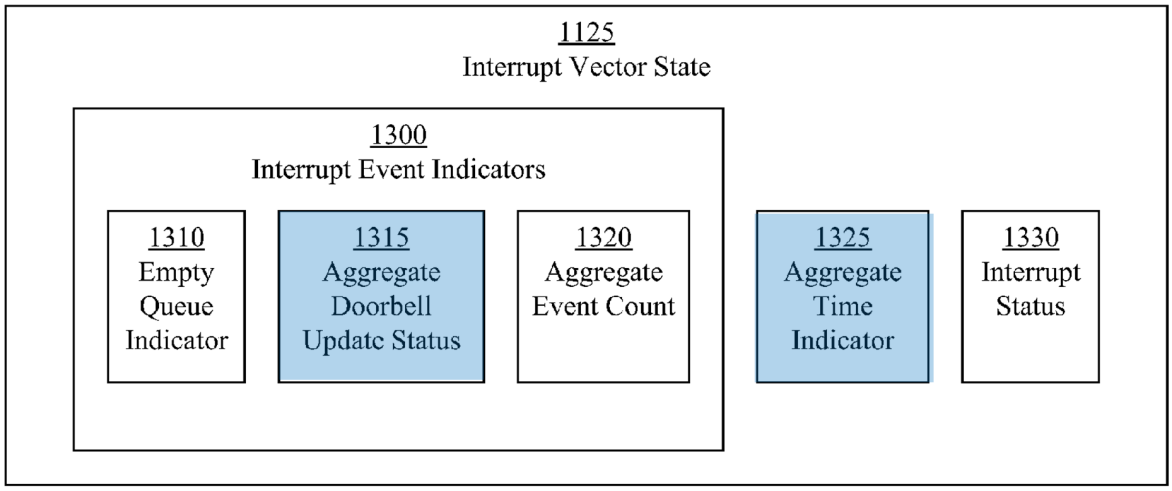
Borchers, Fig. 2 (annotated).

105. Accordingly, because the interrupt mechanisms ("*interrupt vector state*") receive information ("*completion queue state*") as to the status of the response queue ("*completion queue*"), Borchers' interrupt mechanisms are

generated "*based on*" that information. The interrupt mechanism information is *generated* when the timers or counters are updated. In some examples, the values of the timers or counters may be stored in a register. Indeed, a POSITA would have found it obvious for Borchers' interrupt mechanism information to be stored in some type of memory such as a buffer, register, flag, or other form of storage. In other words, a POSITA would have recognized that the timers and counters of the interrupt mechanisms ("*interrupt vector state*") are not merely transitory signals. Indeed, it was known for memory controllers and interrupt managers to use such registers or caches for storage. *See e.g.,* Serebrin, Figs. 2 & 5, [0057], [0090]-[0091]; Serebrin Provisional, Figs. 2 & 5, [0051], [0083], [0084]] (describing how an interrupt manager uses various forms of memory such as cache or registers to store state information). Accordingly, because Borchers' interrupt processor receives and uses the response queue status information to update the interrupt mechanism timers and counters, Borchers' interrupt processor "*generate[s]*" an "*interrupt vector state*" as claimed.

106. Again, Borchers' interrupt mechanisms are analogous to the interrupt vector states described in the '968 patent. For example, like Borchers' ResponseNew counter 280, the interrupt vector states of the '968 patent include an Aggregate Doorbell Update Status 1315. Similarly, like Borchers' Last Response Timer 282, the interrupt vector states in the '968 patent include an aggregate time

indicator 1325.



'968 patent, Fig. 13 (annotated).

107. Thus, because Borchers' interrupt processor uses counters and timers to maintain the set of conditions used to determine whether to send an interrupt, and the counters or timers are updated with information as to the status of the response queue, Borchers renders obvious "*generate an interrupt vector state based on the completion queue state*" as claimed.

[1.3] ***determine the completion queue of the host processing unit contains an unprocessed completion status based on the interrupt vector state, and***

108. **First**, as described above at [1.0], Borchers' host unit includes a response queue ("*completion queue of the host processing unit*").

109. **Second**, Borchers' ResponseNew counter 280 ("*interrupt vector state*") "is configured to track the number of responses transferred to the host 106 that the host has yet to process" ("*unprocessed completion status*"). Borchers,

[0060]. When the interrupt processor places an item in the response queue, the ResponseNew counter 280 is incremented.

> The ResponseNew counter 280 may be enabled by the host 106 when the watermark interrupt mechanism is desired. The host 106 may set the ResponseMark 288, which is a parameter provided as input to the ReponseNew counter 280, as discussed above. **The ResponseNew counter 280 receives as inputs information including** when a packet is transferred to the host 106, when the ResponseHead is updated, **the number of outstanding responses in the host response buffer 123** and when an interrupt has been sent. **The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process. Each time a response is transferred to the response buffer 123 the counter is incremented.** When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].

110. Accordingly, the ResponseNew counter 280 (part of the "interrupt vector state") determines there are unprocessed events in the response queue ("*completion queue*") when it receives that information from the host. Once the

number of unprocessed events reaches the threshold watermark, then an interrupt is sent to the host processor. The interrupt alerts the processor that there are unprocessed events in the response queue.

> In one exemplary implementation, the host 106 may set a ResponseMark parameter, which determines the water mark, and may set the ResponseDelay parameter, which determines the timeout. The host 106 may communicate these parameters to the interrupt processor 124. **If the count of new responses in the response buffer 123 is equal to or greater than the ResponseMark, then an interrupt is generated by the interrupt processor 124 and the count is zeroed**. If the time (e.g., time in microseconds) since the last interrupt is equal to or greater than the ResponseDelay and there are new responses in the response buffer 123, then the interrupt processor 124 generates an interrupt and the timeout is zeroed. If the host 106 removes the new response from the response buffer 123, the count of new responses is updated and the timeout is restarted. In this manner, the host 106 may poll ahead and avoid interrupts from the interrupt processor 124.

Borchers, [0037].

111. Accordingly, the interrupt processor determines there are unprocessed completion statuses in the response queue ("the completion queue of the host processing unit") "*based on*" the receipt of the number of outstanding responses in the response queue by the ResponseNew counter (part of the "*interrupt vector state*").

112. Thus, because the ResponseNew counter 280 (an "*interrupt vector state*") tracks the number of responses the host has yet to process, Borchers renders obvious "*determine the completion queue of the host processing unit contains an unprocessed completion status based on the interrupt vector state*" as claimed.

[1.4] *generate an interrupt message packet for triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue, and*

113. **First**, as explained above at [1.0], Borchers' system sends an interrupt to alert the host processing unit of the unprocessed completion status in the response queue ("*completion queue*"). "The interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123." Borchers, [0035]. The interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123." The interrupt signal received by the host then causes the host to process the interrupt, thereby "triggering the interrupt in the host processing unit." *See e.g.*, Borchers, [0050] ("an interrupt may be received by an interrupt handler 225a-225n").

114. **Second**, a POSITA would have found it obvious for Borchers' interrupt signal to be an "interrupt message packet" consistent with PCI/PCIe standards. Borchers explains that its memory interface "may include a PCIe interface." Borchers, [0110]. It was known that the PCI Express standard uses

Message Signaled Interrupts (MSI).

> **Message Signaled Interrupts (MSI)** is an optional feature that enables a device function to request service by writing a system-specified data value to a system-specified address (using a PCI DWORD memory write transaction). System software initializes the **message** address and message data (from here on referred to as the "vector") during device configuration, allocating one or more vectors to each MSI capable function.

PCI, 231.

> Once MSI or MSI-X is enabled (the appropriate bit in one of the Message Control registers is set), and one or more vectors is unmasked, the function is permitted to send messages. **To send a message**, a function does a DWORD memory write to the appropriate message address with the appropriate message data.
>
> For MSI, the DWORD that is written is made up of the value in the MSI Message Data register in the lower two bytes and zeroes in the upper two bytes.

PCI, 249.

115. Accordingly, it would have been obvious to a POSITA that when Borchers describes sending interrupt signals, those signals would be embodied as MSI or MSI-X interrupts consistent with the PCI/PCIe standards. Borchers' interrupt signals—embodied as MSI or MSI-x interrupts—are therefore "*interrupt message packets*." Such MSI or MSI-x interrupt signals are within the scope of the claims. Indeed, the '968 patent similarly describes the interrupt message packets as

being MSI or MSI-x interrupts: "In some embodiments, the hardware interrupt generated in the host processing unit 110 is a message signal interrupt (MSI) or an extended message signal interrupts (MSI-X) as defined in the PCI Specification 3.0." '968 patent, 16:61-64.

116. Thus, because Borchers' interrupt manager sends an interrupt signal to alert the host that there are unprocessed events in the response queue, Borchers renders obvious "*generate an interrupt message packet for triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue*" as claimed.

[1.5] ***wherein the completion queue state includes a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller for the completion queue.***

117. **First**, as mentioned above at [1.1], Borchers' interrupt processor receives information ("*completion queue state*") from the host as to the status of the response queue.

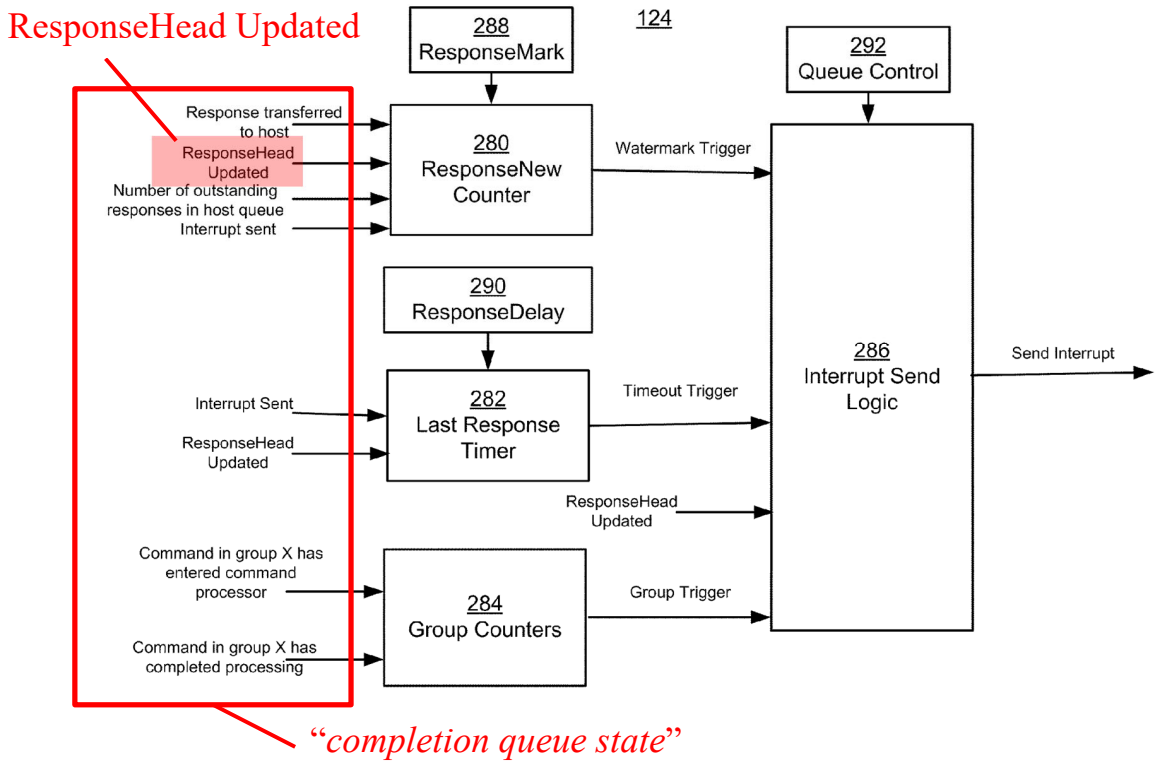
118. **Second**, Borchers' response queue status information ("*completion queue state*") indicates whether the host processing unit has updated the response head pointer ("*the host processing unit updates a head pointer*"). As explained above at [1.1], Borchers' interrupt processor 124 receives various pieces of information from the host system as to the status of the host's response queue. These pieces of information include an updated response queue head pointer.

> The ResponseNew counter 280 may be enabled by the host 106 when the watermark interrupt mechanism is desired. The host 106 may set the ResponseMark 288, which is a parameter provided as input to the ReponseNew counter 280, as discussed above. **The ResponseNew counter 280 receives as inputs information including when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process**. Each time a response is transferred to the response buffer 123 the counter is incremented. When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].

119. During operation, "[t]he host 106 is configured to read the responses from the response buffer 123 and **update the head pointer** in the MMIO space of the data storage device to indicate that the host has received the response."

Borchers, [0035].



120. Third, the indication that the host has updated the head pointer is a "*doorbell update status*" as claimed. Such a mechanism that indicates when the response head pointer has been updated is often referred to as a "doorbell" update status. As explained above at Section V, using a response queue that is produced by the memory controller and consumed by the host is referred to as the producer/consumer model. Indeed, Borchers explains that the "data storage device 100 may be the ... producer of the response buffer 123" and the "host 106 may be the ... consumer of the response buffer 123." Borchers, [0030].

121. The PCI standard explains that in this Producer-Consumer model, a notification of the head pointer being updated is a "doorbell" event.

The Producer - Consumer model for data movement between two masters is an example of a system that would require this kind of ordering. **In this model, one agent, the Producer, produces or creates the data and another agent, the Consumer, consumes or uses the data**. The Producer and Consumer communicate between each other via a flag and a status element. **The Producer sets the flag when all the data has been written and then waits for a completion status code**. The Consumer waits until it finds the flag set, then it resets the flag, consumes the data, and writes the completion status code. When the Producer finds the completion status code, it clears it and the sequence repeats. Obviously, the order in which the flag and data are written is important. If some of the Producer's data writes were posted, then without buffer-flushing rules it might be possible for the Consumer to see the flag set before the data writes had completed. The PCI ordering rules are written such that no matter which writes are posted, the Consumer can never see the flag set and read the data until the data writes are finished. This specification refers to this condition as "having a consistent view of data." Notice that if the Consumer were to pass information back to the Producer in addition to the status code, the order of writing this additional information and the status code becomes important, just as it was for the data and flag.

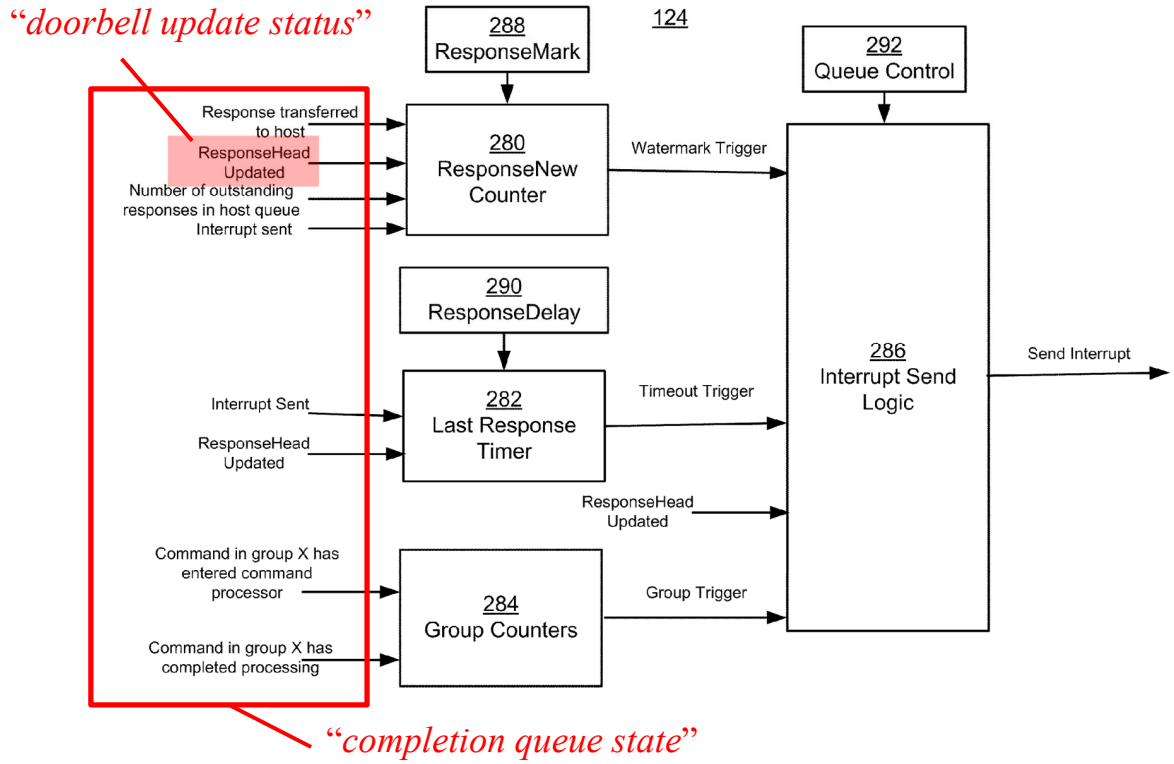
In practice, the flag might be a doorbell register in a device or it might be a main-memory pointer to data located somewhere else in memory. **And the Consumer might signal the Producer using an interrupt or another doorbell register**, rather than having the Producer poll the status element. But in all cases, the basic need remains

> the same; the Producer's writes to the data area must complete before the Consumer observes that the flag has been set and reads the data.
>
> This model allows the data, the flag, the status element, the Producer, and the Consumer to reside anywhere in the system. Each of these can reside on different buses and the ordering rules maintain a consistent view of the data. For example, in Figure E-1, the agent producing the data, the flag, and the status element reside on Bus 1, while the actual data and the Consumer of the data both reside on Bus 0. The Producer writes the last data and the PCI-to-PCI bridge between Bus 0 and 1 completes the access by posting the data. The Producer of the data then writes the flag changing its status to indicate that the data is now valid for the Consumer to use. In this case, the flag has been set before the final datum has actually been written (to the final destination). PCI ordering rules require that when the Consumer of the data reads the flag (to determine if the data is valid), the read will cause the PCI-to-PCI bridge to flush the posted write data to the final destination before completing the read. When the Consumer determines the data is valid by checking the flag, the data is actually at the final destination.

PCI, 308.

122. Accordingly, a POSITA would have recognized that the register that stores the head pointer is a doorbell update register. Accordingly, the notification that the head pointer of the response queue has been updated is a "doorbell update status" as claimed. Indeed, the claim language itself defines what a doorbell update status does—an indication "*in which the host processing unit updates a head*

pointer ... for the completion queue." Borchers does precisely that by sending a notification that the head pointer of the response queue ("*completion queue*") has been updated. Accordingly, Borchers' notification that the head pointer of the response queue is a "doorbell update status" as claimed.



Borchers, Fig. 2 (annotated).

123. Fourth, Borchers' head pointer is stored in a register on the storage device controller ("*head pointer stored in the nonvolatile memory controller for the completion queue*").

> Implementations can include one or more of the following features. For example, the circular command queue can include a command head pointer and a command tail pointer, and the circular response queue can

> include a response head pointer and a response tail pointer, and the host device can further include a first register configured to store command head pointer values, and a second register configured to store response tail pointer values. **The data storage device can include** a third register configured to store command tail pointer values, and **a fourth register configured to store responsehead pointer values**. The third register can exist in a memory mapped address space of the data storage device, and the driver can be configured to write updated command tail pointer values to the third register.

Borchers, [0005]; *see also* Borchers, [0030] ("[A]nd values of the head pointer of the response buffer can be stored in a register on the data storage device 100.")

124. Borchers further explains that the register that stores the head pointer may be within the controller: "The command tail pointer and **the response head pointer** may be **stored in registers of the data storage device 100** in memory mapped I/O space **within the controller 110**." Borchers, [0031].

125. Thus, because Borchers' interrupt processor maintains information about the response queue, including that the response head pointer has been updated, and the head pointer is stored in a doorbell register within the memory controller, Borchers renders obvious "*wherein the completion queue state includes a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller for the completion queue*" as claimed.

4. Claim 2

[2.1] ***The nonvolatile memory controller of claim 1, wherein the interrupt is a hardware interrupt.***

126. Borchers controller 110, which includes the interrupt processor 124, may be implemented in hardware. Thus, the interrupts sent by the interrupt processor are hardware interrupts.

> The controller 110 may include a microcontroller, a FPGA controller, other types of controllers, or combinations of these controllers. In one exemplary implementation**, the controller 110 is a microcontroller. The microcontroller may be implemented in hardware**, software, or a combination of hardware and software. For example, the microcontroller may be loaded with a computer program product from memory (e.g., memory module 116) including instructions that, when executed, may cause the microcontroller to perform in a certain manner. The microcontroller may be configured to receive commands from the host 106 using the interface 108 and to execute the commands. For instance, the commands may include commands to read, write, copy and erase blocks of data using the flash memory chips 118 a and 118 b, as well as other commands.

Borchers, [0113].

127. Additionally, the specification of the ’968 patent refers to MSI and MSI-x interrupts as hardware interrupts: “In some embodiments, **the hardware interrupt** generated in the host processing unit 110 **is a message signal interrupt (MSI) or an extended message signal interrupts (MSI-X)** as defined in the PCI

Specification 3.0." '968 patent, 16:61-64. For the reasons explained above at [1.4], a POSITA would have found it obvious for Borchers' PCIe interrupts to be MSI or MSI-x signals associated with the PCI express standard.

128. Thus, because Borchers' memory controller and interrupt processor may be implemented in hardware, and may be MSI or MSI-x interrupt signals, Borchers renders obvious "*wherein the interrupt is a hardware interrupt*" as claimed.

5. Claim 3

[3.1] ***The nonvolatile memory controller of claim 1, wherein the interrupt message packet includes an interrupt vector associated with the completion queue for triggering the interrupt in the host processing unit.***

129. **First**, as explained above at [1.4], a POSITA would have found it obvious—based on background knowledge—for Borchers' interrupt signals to be MSI or MSI-x interrupt signals. Such interrupt signals include both a message address and message data, which are referred to as an interrupt vector.

> Message Signaled Interrupts (MSI) is an optional feature that enables a device function to request service by writing a system-specified data value to a system-specified address (using a PCI DWORD memory write transaction). **System software initializes the message address and message data (from here on referred to as the "vector")** during device configuration, allocating one or more vectors to each MSI capable function.

PCI, 231.

130. Accordingly, Borchers' interrupt signals include an "interrupt vector" (i.e., a message address and message data).

131. **Second**, Borchers' interrupt signals trigger an interrupt in the host processor. "The interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123." Borchers, [0035]. The interrupt signal received by the host then causes the host to process the interrupt, thereby "*triggering the interrupt in the host processing unit*." *See e.g.*, Borchers, [0050] ("an interrupt may be received by an interrupt handler 225a-225n").

132. Thus, because it would have been obvious for Borchers' interrupt signals, which trigger interrupts in the host, to include vectors consistent with PCI standards, Borchers renders obvious "*wherein the interrupt message packet includes an interrupt vector associated with the completion queue for triggering the interrupt in the host processing unit*" as claimed.

6. Claim 4

[4.1] ***The nonvolatile memory controller of claim 3, wherein completion queue state includes an event count indicating a number of completion message packets received by the host controller interface for the completion queue associated with the completion queue state.***

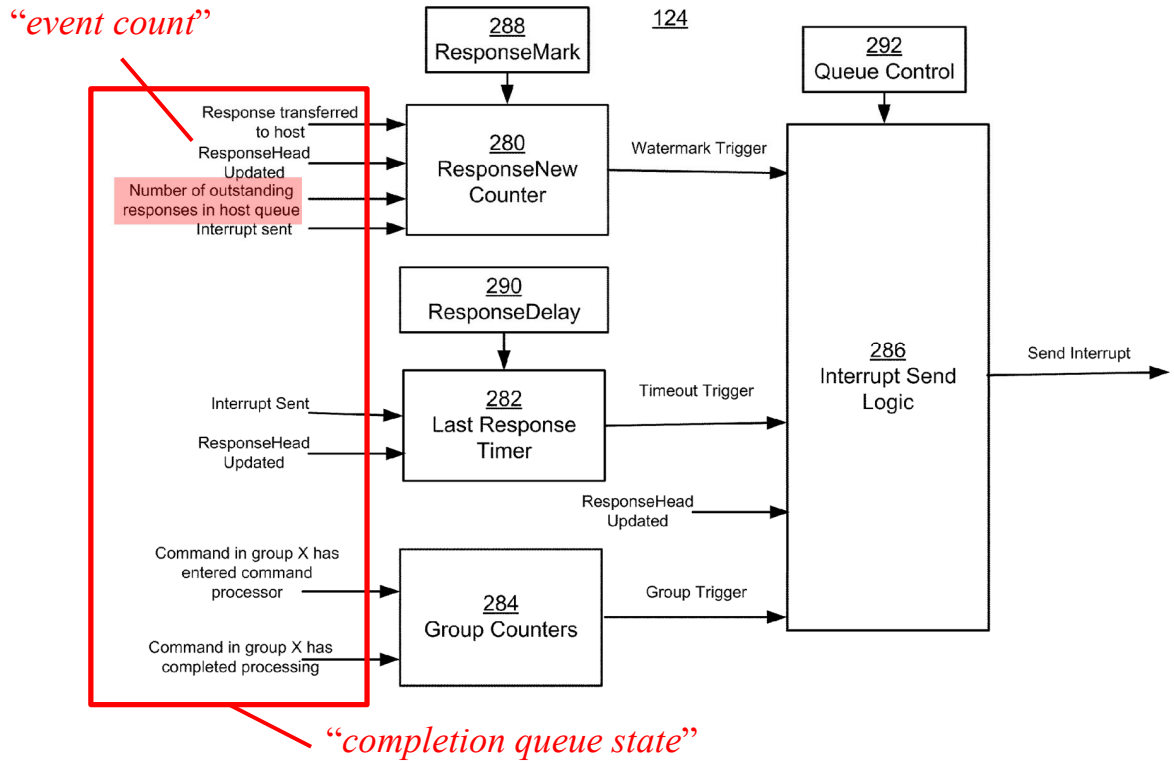
133. **First**, as explained above at [1.1], Borchers' interrupt processor generates information as to the status of the response queue.

134. **Second**, that information includes the number of outstanding

responses in the response queue ("*event count*").

> **The ResponseNew counter 280 receives as inputs information including when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process**. Each time a response is transferred to the response buffer 123 the counter is incremented. **When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286**. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].



Borchers, Fig. 2 (annotated).

135. Thus, because Borchers maintains status information including the number of outstanding responses in the host's response queue, Borchers renders obvious "*wherein completion queue state includes an event count indicating a number of completion message packets received by the host controller interface for the completion queue associated with the completion queue state*."

7. Claim 5

[5.1] ***The nonvolatile memory controller of claim 4, wherein the interrupt vector state includes an aggregate event count for aggregating event counts of the completion queue state over a plurality of update operations, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate event count is below an aggregate event count threshold.***

136. **First**, Borchers' interrupt mechanisms ("*interrupt vector state*")

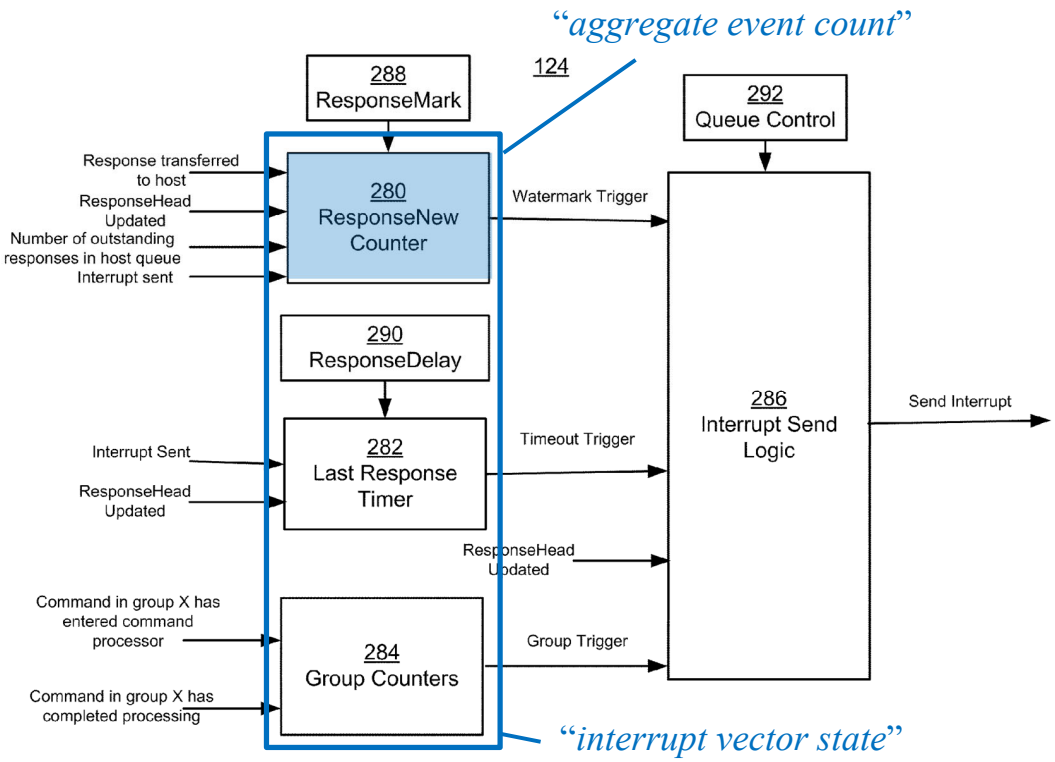
include a ResponseNew Counter 280 ("*aggregate event count*").

> **The ResponseNew counter 280 receives as inputs information including when a packet is transferred to the host 106, when the ResponseHead is updated, the number of outstanding responses in the host response buffer 123 and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process**. Each time a response is transferred to the response buffer 123 the counter is incremented. **When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286**. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level. When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].

137. The ResponseNew counter receives as an input various pieces of information as to the status of the response queue ("*completion queue state*"). One such piece of information is when an event is sent to the host. "The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process. Each time a response is transferred to the response buffer 123 the counter is incremented." Borchers, [0060]. Accordingly,

because the ResponseNew counter 280 is incremented over multiple events, the ResponseNew counter 280 is for "*aggregating event counts of the completion queue state over a plurality of update operations*."



Borchers, Fig. 2 (annotated).

138. **Second**, Borchers' interrupt processor 124 does not send an interrupt signal until the ResponseNew counter 280 ("*aggregate event count*") reaches a watermark level ("*the interrupt manager is inhibited from generating an interrupt message packet if the aggregate event count is below an aggregate event count threshold*").

In one exemplary implementation, the host 106, through its driver 107, may control when the interrupt processor 124 should generate

> **interrupts**. The host 106 may use one or more different interrupt mechanisms, including a combination of different interrupt mechanisms, to provide information to the interrupt processor 124 regarding interrupt processing. **For instance, the host 106 through the driver 107 may configure the interrupt processor 124 to use a water mark interrupt mechanism**, a timeout interrupt mechanism, a group interrupt mechanism, or a combination of these interrupt mechanisms. **In one exemplary implementation, the host 106 may set a ResponseMark parameter, which determines the watermark**, and may set the ResponseDelay parameter, which determines the timeout. The host 106 may communicate these parameters to the interrupt processor 124. **If the count of new responses in the response buffer 123 is equal to or greater than the ResponseMark, then an interrupt is generated by the interrupt processor 124 and the count is zeroed**. If the time (e.g., time in microseconds) since the last interrupt is equal to or greater than the ResponseDelay and there are new responses in the response buffer 123, then the interrupt processor 124 generates an interrupt and the timeout is zeroed. If the host 106 removes the new response from the response buffer 123, the count of new responses is updated and the timeout is restarted. In this manner, the host 106 may poll ahead and avoid interrupts from the interrupt processor 124.

Borchers, [0036]-[0037].

139. Thus, because the ResponseNew counter gets incremented when the interrupt processor places events in the response queue, and does not send an

interrupt until the watermark threshold is met, Borchers renders obvious "*wherein the interrupt vector state includes an aggregate event count for aggregating event counts of the completion queue state over a plurality of update operations, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate event count is below an aggregate event count threshold*" as claimed.

8. Claim 6

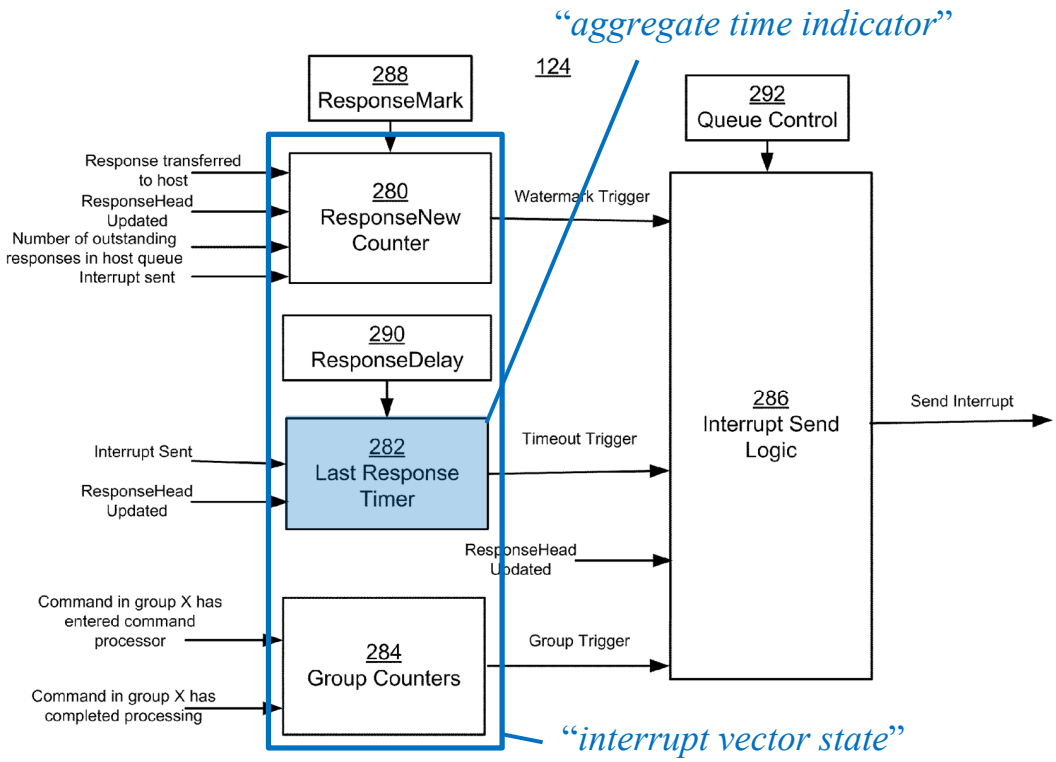
[6.1] ***The nonvolatile memory controller of claim 3, wherein the interrupt vector state includes an aggregate time indicator for indicating, a time period between the last time the interrupt manager generated an interrupt message packet for the completion queues associated with the interrupt vector state.***

140. Borchers' interrupt mechanisms ("*interrupt vector state*") include a Last Response timer 282 ("*aggregate time indicator*").

> The last response timer 282 may be enabled when the timer interrupt mechanism is desired. **The last response timer 282 may be configured to keep track of time since the last interrupt**. For instance, the last response timer 282 may track the amount of time since the last interrupt in microseconds. The host 106 may set the amount of time using a parameter, for example, a Response Delay parameter 290. In one exemplary implementation, the Response Delay 290 timeout may be the number of microseconds since the last interrupt, or since the last time that the host 106 removed new responses from the response buffer 123, before an interrupt is generated.

> The last response timer 282 receives as input a signal indicating when an interrupt is sent. **The last response timer 282 also may receive a signal when the ResponseHead is updated, which indicates that the host 106 has removed responses from the response buffer 123**. An interrupt may be generated only if the response buffer 123 contains outstanding responses.
>
> The last response timer 282 is configured to generate a timeout trigger when the amount of time being tracked by the last response timer 282 is greater than the Response Delay parameter 290. **When this occurs and the response buffer 123 contains new responses, then a timeout trigger signal is sent to the interrupt send logic 286**. If the last response timer 282 is the only interrupt mechanism enabled, then the interrupt send logic 286 generates and sends an interrupt to the host. If other interrupt mechanisms also are enabled, then the interrupt send logic 286 may take into account the other interrupt mechanisms as well.

Borchers, [0062]-[0064].



Borchers, Fig. 2 (annotated).

141. Accordingly, Borchers' last response timer 282 is for keeping track of the amount of time since the last interrupt was sent ("*for indicating, a time period between the last time the interrupt manager generated an interrupt message packet for the completion queues associated with the interrupt vector state*.")

142. Thus, because Borchers' Last Response Timer indicates the amount of time since the interrupt processor last sent an interrupt signal to the host, Borchers renders obvious "*wherein the interrupt vector state includes an aggregate time indicator for indicating, a time period between the last time the interrupt manager generated an interrupt message packet for the completion queues associated with*

the interrupt vector state" as claimed.

9. Claim 7

[7.1] ***The nonvolatile memory controller of claim 3, wherein the completion queue state includes an empty queue indicator for indicating whether the completion queue is empty, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the empty queue indicator indicates the completion queue is empty.***

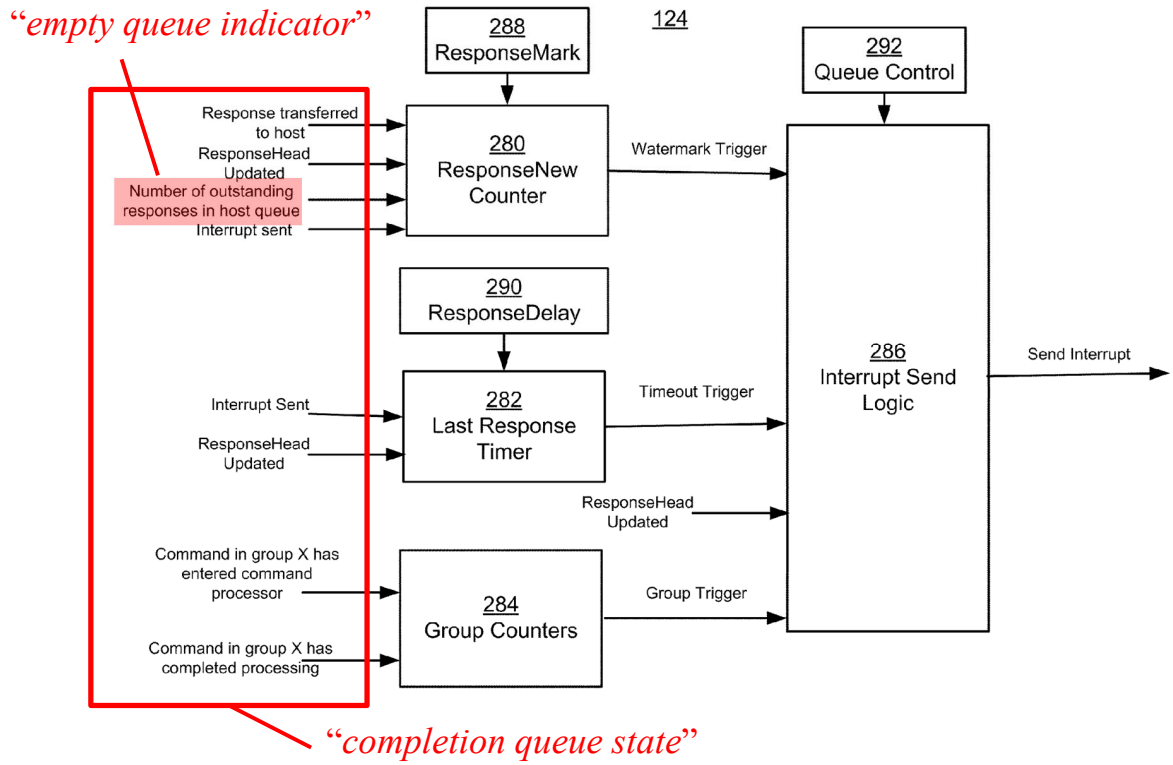
143. First, as explained above at [4.1], Borchers' response queue status information ("*completion queue state*") includes the number of outstanding responses in the response queue. When that number is zero, it is an "empty queue indicator" that indicates the response queue ("*completion queue*") is empty.

> **The ResponseNew counter 280 receives as inputs information including** when a packet is transferred to the host 106, when the ResponseHead is updated, **the number of outstanding responses in the host response buffer 123** and when an interrupt has been sent. The ResponseNew counter 280 is configured to track the number of responses transferred to the host 106 that the host has yet to process. Each time a response is transferred to the response buffer 123 the counter is incremented. **When the counter 280 reaches or exceed the watermark level set by the host 106, i.e., the ResponseMark 288, then a watermark trigger is generated and sent to the interrupt send logic 286**. The watermark level, i.e., the ResponseMark 288, is the number of new responses in the response buffer 123 needed to generate an interrupt. If the host 106 removes new responses from the response buffer 123, they do not count toward meeting the watermark level.

When an interrupt is generated, the count toward the ResponseMark is reset.

Borchers, [0060].

144. Accordingly, the information that indicates the number of outstanding responses in the response queue acts as an empty queue indicator when the number is zero.



Borchers, Fig. 2 (annotated).

145. **Second**, Borchers system does not send an interrupt if there are no outstanding responses in the response queue ("*wherein the interrupt manager is inhibited from generating an interrupt message packet if the empty queue indicator indicates the completion queue is empty*"). Rather, the interrupt send logic 286

generates interrupts only if the watermark level is reached, the timer is triggered, or the group counter is triggered. *See* Borchers, [0036]-[0037]. A POSITA would have further understood that there would be no reason to send an interrupt if there are no responses in the response queue for the host to process.

146. Thus, because the response queue status information includes the number of outstanding responses in the host queue, which may be zero, and the processor does not needlessly send interrupts when there are no responses in the response queue to process, Borchers renders obvious "*wherein the completion queue state includes an empty queue indicator for indicating whether the completion queue is empty, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the empty queue indicator indicates the completion queue is empty*" as claimed.

10. Claim 8

[8.0] ***A nonvolatile memory controller comprising:***

147. For the reasons explained at [1.0], Borchers teaches a "*non-volatile memory controller*" as claimed.

[8.1] ***a completion queue state memory configured to store a plurality of completion queue states corresponding to a plurality of completion queues in a host processing unit;***

148. **First**, as explained above at [1.1], Borchers receives status information for a response queue from the host. This status information teaches a

"*completion queue state*" as claimed.

149. Second, the memory location where Borchers' response queue status information is stored or buffered (e.g., RAM, a register, a flag), is "*a completion queue state memory*" as claimed. As one example, Borchers' memory controllers includes "register 161" that may be used to store or buffer such status information.

Given that the register stores items such as head and tail pointers, it would have been obvious for register 161 to buffer or temporarily store the response queue status information. Indeed, it was known to store such information in a buffer or register. *See e.g.,* Serebrin, Figs. 2 & 5, [0057], [0090]-[0091]; Serebrin Provisional, Figs. 2 & 5, [0051], [0083], [0084] (describing how an interrupt manager uses various forms of memory such as cache or registers to store state information). Accordingly, the storage location used by Borchers' interrupt processor to store the interrupt mechanism information is a "*a completion queue state memory configured to store a plurality of completion queue states*."

150. Third, Borchers describes an example in which there are a plurality of response queues in the host ("*a plurality of completion queues in a host processing unit*").

> **Although FIG. 1A illustrates one command buffer 119 and one response buffer 123, multiple pairs of buffers may be used, as illustrated in FIG. 1B** and discussed in more detail below. For example, up to and including 32 buffer pairs may be used. In one

> exemplary implementation, the data storage device 100 may service the multiple command buffers 119 in around robin fashion, where the data storage device 100 may retrieve a fixed number of commands at a time from each of the command buffers 119. The response buffer 123 may include its own interrupt and interrupt parameters.

Borchers, [0028]; *see also* [0046]-[0058].

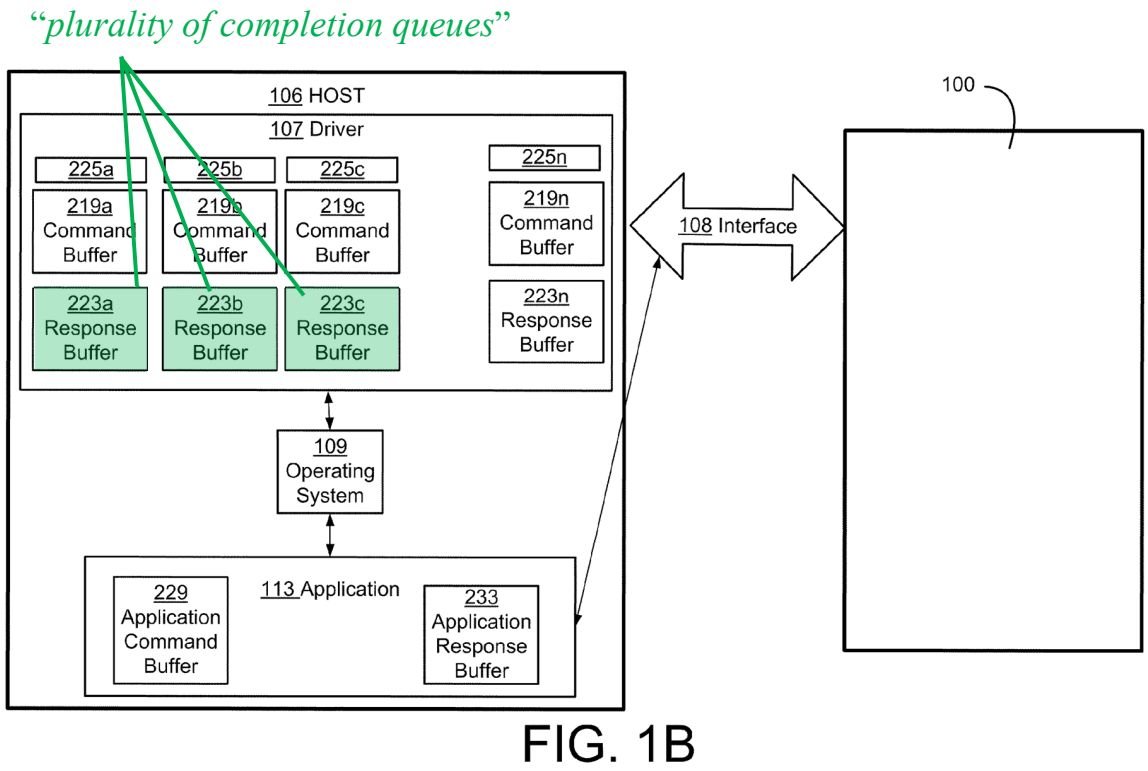


FIG. 1B

Borchers, Fig. 1B (annotated).

151. As also explained above, Borchers explains that the command and response queues of Fig. 1A may be implemented as circular queues: "In one exemplary implementation shown in FIG. 1C the command buffer 119 can be configured as a circular queue 159 that is used to communicate information from

the host 106 and to the data storage device 100 of FIG. 1A." Borchers, [0030]. Accordingly, it would have been obvious for Borchers' host to include multiple buffer pairs (as shown in Fig. 1B and described in accompanying text) which are implemented as circular queues (as shown in Fig. 1C and described in accompanying text). Indeed, when describing the buffer pairs of Fig. 1B, Borchers explains that "Each of the buffer pairs may include its own set of head and tail pointers. The use of the head and tail pointers may be the same as described above with respect to the command buffer 119 and the response buffer 123 of FIG. 1A." Borchers, [0049].

152. As explained above at [1.1], Borchers describes that the interrupt processor 124 receives status information as to the response queue. Accordingly, in the example where there are multiple response queues, it would have been obvious that the interrupt processor receives status information for each of the response queues. The status information for each of the response queues thus teaches "*a plurality of completion queue states corresponding to a plurality of completion queues*" as claimed.

153. Thus, because Borchers' interrupt controller receives and utilizes response queue status information for a plurality of buffers, Borchers renders obvious "*a completion queue state memory configured to store a plurality of completion queue states corresponding to a plurality of completion queues in a*

host processing unit" as claimed.

154. Additionally, Borchers teaches a "*plurality of completion queue states*" in that there are several pieces of status information, including that a Response has been transferred to the host, the response queue head pointer has been updated, and the number of outstanding responses in the host queue. *See* Borchers, Fig. 2; Borchers, [0060].

[8.2] ***an interrupt vector state memory configured to store a plurality of interrupt vector states corresponding to a plurality of interrupt vectors in the host processing unit; and***

155. **First**, as explained above at [1.2], Borchers teaches various interrupt mechanisms that teach an "*interrupt vector state*" as claimed. These interrupt mechanisms are used to determine whether to send an interrupt signal to alert the host that there are responses in the response queue that have yet to be processed. *See* Borchers, [0060].

156. **Second**, as explained above at [8.1], Borchers describes an example in which there are multiple response queues on the host. *See* Borchers, Fig. 1B, [0046]-[0048]. Accordingly, in such case, there would be multiple interrupt mechanisms—one for each response queue. For example, there would be a ResponseNew counter 280 for each response queue. There would be a last response timer 282 for each response queue. The multiple ResponseNew counters or last response timers thus teach "*a plurality of interrupt vector states*" as

claimed.

157. Additionally, Borchers teaches "*a plurality of interrupt vector states*" in that there are multiple types of interrupt vector states. For example, the ResponseNew counter 280 and last response timer 282 together teach "*a plurality of interrupt vector states*" as claimed.

158. **Third**, as with the completion queue state memory described above at [8.1], the memory used by the interrupt processor 124 to store the interrupt mechanism information (e.g., response count or time) teaches an "*interrupt vector state memory*" as claimed. For example, whether the interrupt mechanism information is stored in register 161 or some other register or location, such storage location teaches an "*interrupt vector state memory*" as claimed.

159. **Fourth,** for the reasons explained above at [1.2], it would have been obvious for Borchers' interrupt signals to be implemented as interrupt vectors consistent with the PCI standard. Accordingly, each of Borchers' interrupt mechanisms corresponds to a different interrupt vector ("*a plurality of interrupt vector states corresponding to a plurality of interrupt vectors*").

160. Thus, because Borchers' host includes a plurality of response buffers, and the memory controller utilizes interrupt mechanisms that determine when to send an interrupt signal for unprocessed responses in those buffers, Borchers renders obvious "*an interrupt vector state memory configured to store a plurality*

of interrupt vector states corresponding to a plurality of interrupt vectors in the host processing unit" as claimed.

161. Additionally, Borchers teaches a "*plurality of interrupt vector states*" in that there are multiple interrupt mechanisms. These multiple interrupt mechanisms include a ResponseNew counter 280, a last response timer 282, and a group counter 284. *See* Borchers, Fig. 2; Borchers, [0060].

[8.3] ***an interrupt manager controller coupled to the completion queue state memory and the interrupt vector state memory,***

162. **First**, as described above at [1.1], Borchers' memory controller includes an interrupt processor 124 ("*interrupt manager controller*").

163. **Second,** the interrupt manager controller utilizes the response queue status information ("*completion queue state*") and interrupt mechanisms ("*interrupt vector state*"). Accordingly, the interrupt processor is "coupled to" the storage locations of the status information and interrupt mechanism information. For example, the interrupt processor is coupled with register 161, which is one option where the response queue status information and interrupt mechanism information may be stored.

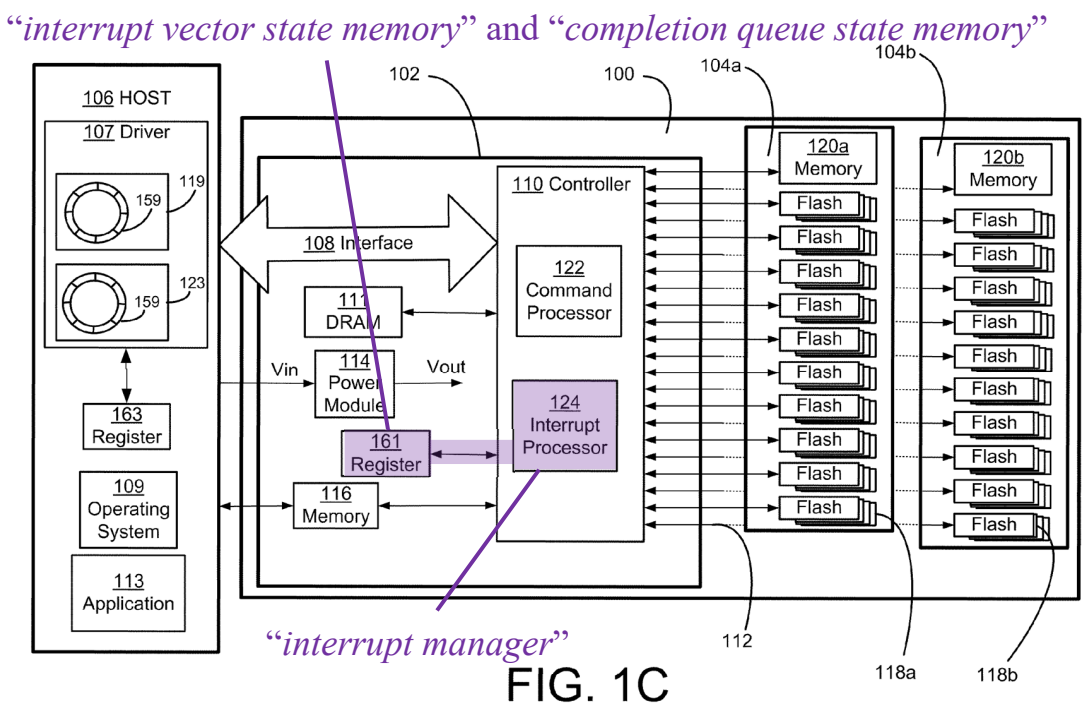


FIG. 1C

Borchers, Fig. 1C (annotated).

164. Thus, because Borchers describes an interrupt processor that utilizes response queue status information and interrupt mechanism information stored in a type of data storage, Borchers renders obvious "*an interrupt manager controller coupled to the completion queue state memory and the interrupt vector state memory*" as claimed.

[8.4] ***the interrupt manager controller configured to map at least one completion queue state of the plurality of completion queue states to an interrupt vector state of the plurality of interrupt vector states,***

165. Borchers explains that when there are multiple command/response queue pairs as shown in Fig. 1B, each buffer pair is processed and handled separately.

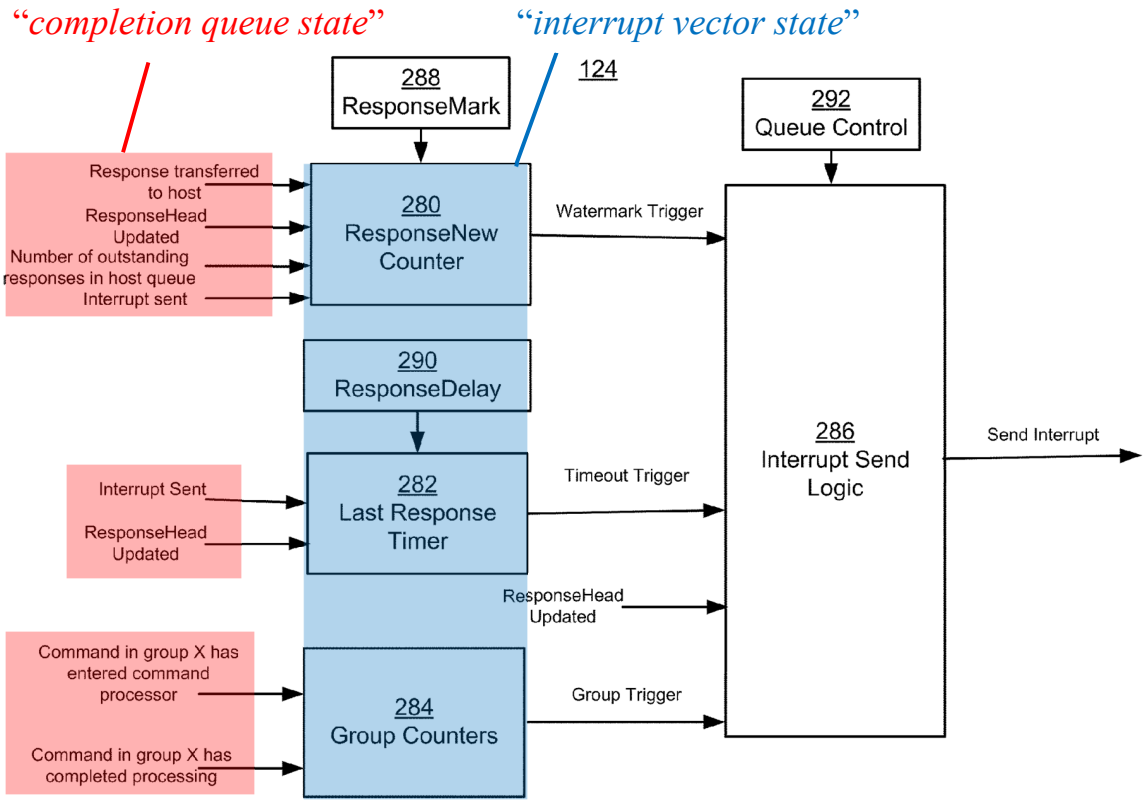
The command buffers 219a-219n and the response buffers 223a-223n may be configured to operate and function as described above with respect to the command buffer 119 and the response buffer 123 of FIG. 1A. Each of the buffer pairs may include its own set of head and tail pointers. The use of the head and tail pointers may be the same as described above with respect to the command buffer 119 and the response buffer 123 of FIG. 1A. **The multiple different head and tail pointers, each of which corresponds to a buffer pair, may be stored on the host 106, the data storage device 100 or a combination of the host 106 and the data storage device 100**.

Each of the response buffers 223a-223n may have an associated interrupt handler 225a-225m. In this manner, each response buffer 223a-223n may process the interrupts received from the data storage device 100 on an individual basis. **In some instances, an interrupt may be received by an interrupt handler 225a-225n when a related group of commands has been processed by the data storage device, as discussed in more detail below with respect to FIG. 2**.

Each of the buffer pairs may be granted access to any address mapping, which may be stored on the host 106 and/or on the data storage device 100. For example, each of the buffer pairs may be granted access to the logical to physical address mapping, which may be stored in DRAM 111 of FIG. 1A. In one exemplary implementation, any address mapping or tables such as, for example, the logical to physical address mapping may be shared Such that each pair of buffers may have access to the mapping.

Borchers, [0049]-[0051].

166. Accordingly, each buffer pair is handled separately in the manner described with Fig. 2. Fig. 2 describes the response queue status information ("*completion queue state*") and interrupt mechanisms ("*interrupt vector state*") as shown below. Thus, for each separate command/response buffer pair, the corresponding response queue status information is associated with the corresponding interrupt mechanisms for that pair ("*map at least one completion queue state of the plurality of completion queue states to an interrupt vector state of the plurality of interrupt vector states*"). Notably, the '968 patent describes mapping a completion queue state to an interrupt vector state with no more specificity.

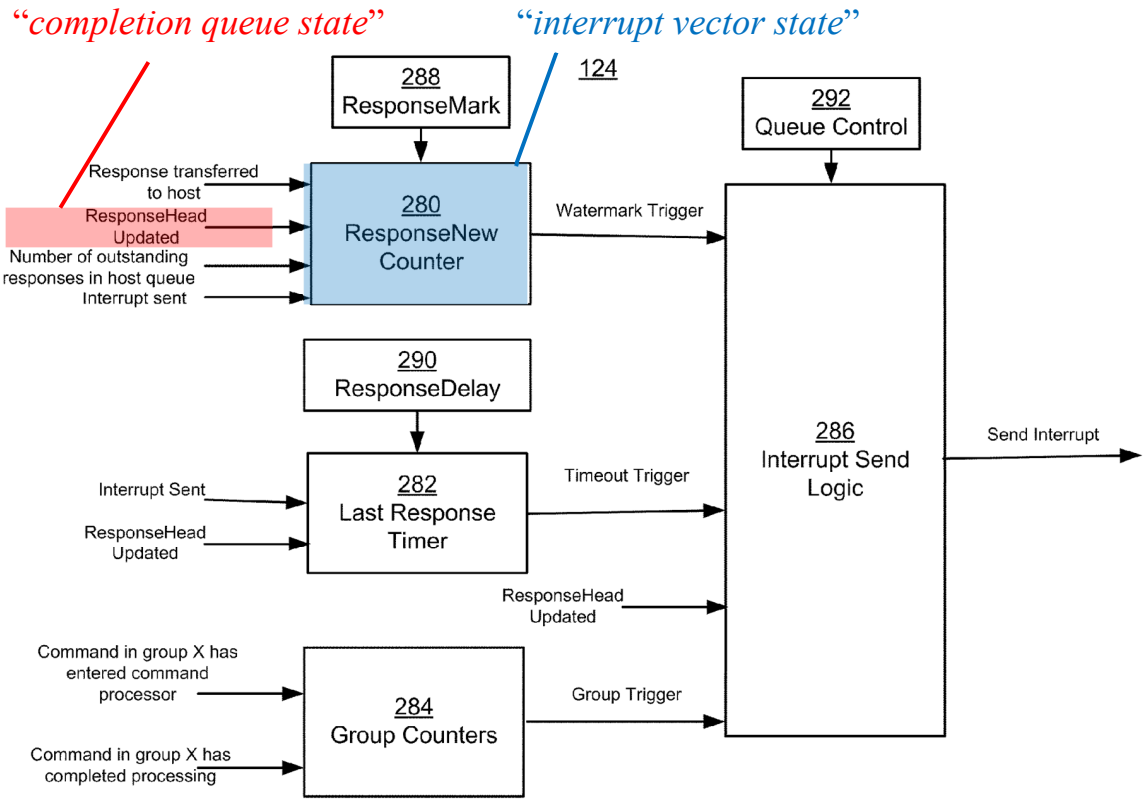


Borchers, Fig. 2 (annotated).

167. Thus, because Borchers' interrupt controller includes interrupt mechanisms for each separate command/response buffer pair, Borchers renders obvious "*the interrupt manager controller configured to map at least one completion queue state of the plurality of completion queue states to an interrupt vector state of the plurality of interrupt vector states*" as claimed.

168. Additionally, Borchers teaches a "plurality of completion queue states" in that there are several pieces of status information, including that a Response has been transferred to the host, the response queue head pointer has been updated, and the number of outstanding responses in the host queue. At least

one of these pieces of information is mapped to one of the interrupt mechanisms. For example, the ResponseHead updated piece of information is mapped to the ResponseNew counter 280 as shown below.



Borchers, Fig. 2 (annotated).

169. Thus, because Borchers maps at least one of the response queue status items (ResponseHead Updated) to one of the interrupt mechanisms (ResponseNew Counter 280), Borchers renders obvious "*the interrupt manager controller configured to map at least one completion queue state of the plurality of completion queue states to an interrupt vector state of the plurality of interrupt vector states*" as claimed.

[8.5] ***generate the interrupt vector state based on each completion queue state mapped to the interrupt vector state,***

170. **First**, for the reasons explained above at [1.2], Borchers generates interrupt mechanism information ("*interrupt vector state*") based on the status information of the response queue ("*each completion queue state mapped to the interrupt vector state*").

171. **Second,** in the case where there are multiple command/response buffer pairs as described in Fig. 1B (and described above at [8.1]), Borchers' system would include counters and timers ("*interrupt vector states*") for each command/response queue. Accordingly, each of the counters/timers are "*based on*" their respective response queue status information ("*completion queue state*").

[8.6] ***determine a completion queue corresponding to a completion queue state mapped to the interrupt vector state contains an unprocessed completion status based on the interrupt vector state, and***

172. For the reasons explained above at [1.3], Borchers' interrupt processor "*determine a completion queue corresponding to a completion queue state mapped to the interrupt vector state contains an unprocessed completion status based on the interrupt vector state.*"

[8.7] ***to generate an interrupt message packet based on the interrupt vector state for triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue.***

173. For the reasons explained above at [1.4], Borchers renders obvious "*to generate an interrupt message packet based on the interrupt vector state for*

triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue" as claimed.

11. Claim 9

[9.1] ***The nonvolatile memory controller of claim 8, wherein the interrupt manager controller is configured to map at least two completion queue states of the plurality of completion queue states to the interrupt vector state.***

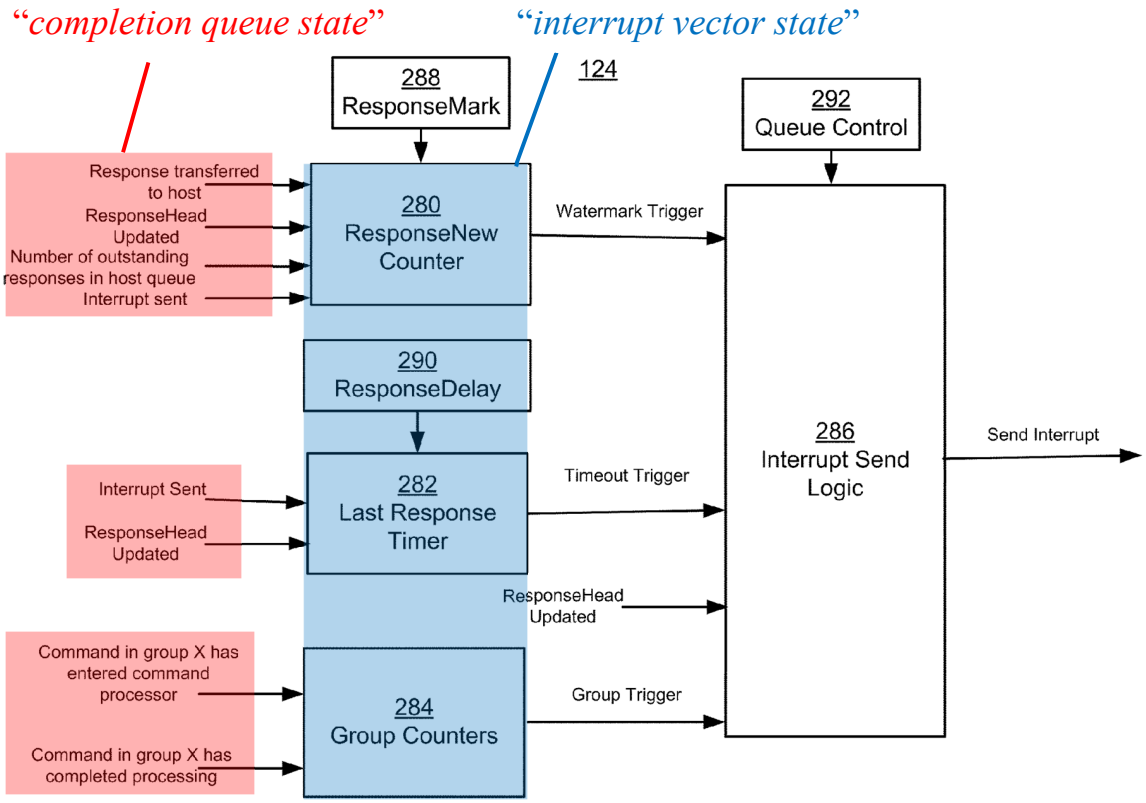
174. Borchers explains that when there are multiple command/response queue pairs as shown in Fig. 1B, each buffer pair is processed and handled separately.

> **The command buffers 219a-219n and the response buffers 223a-223n may be configured to operate and function as described above with respect to the command buffer 119 and the response buffer 123 of FIG. 1A**. Each of the buffer pairs may include its own set of head and tail pointers. The use of the head and tail pointers may be the same as described above with respect to the command buffer 119 and the response buffer 123 of FIG. 1A. **The multiple different head and tail pointers, each of which corresponds to a buffer pair, may be stored on the host 106, the data storage device 100 or a combination of the host 106 and the data storage device 100**.
>
> **Each of the response buffers 223a-223n may have an associated interrupt handler 225a-225m**. In this manner, each response buffer 223a-223n may process the interrupts received from the data storage device 100 on an individual basis. **In some instances, an interrupt may be received by an interrupt handler 225a-225n when a related**

> **group of commands has been processed by the data storage device, as discussed in more detail below with respect to FIG. 2**.
> **Each of the buffer pairs may be granted access to any address mapping, which may be stored on the host 106 and/or on the data storage device 100**. For example, each of the buffer pairs may be granted access to the logical to physical address mapping, which may be stored in DRAM 111 of FIG. 1A. In one exemplary implementation, any address mapping or tables such as, for example, the logical to physical address mapping may be shared Such that each pair of buffers may have access to the mapping.

Borchers, [0049]-[0051].

175. Accordingly, each buffer pair is handled separately in the manner described with Fig. 2. Fig. 2 describes the response queue status information ("*completion queue state*") and interrupt mechanisms ("*interrupt vector state*") as shown below. Thus, for each separate command/response buffer pair, the corresponding response queue status information is associated with the corresponding interrupt mechanisms for that pair ("*map at least two completion queue states of the plurality of completion queue states to the interrupt vector state*").

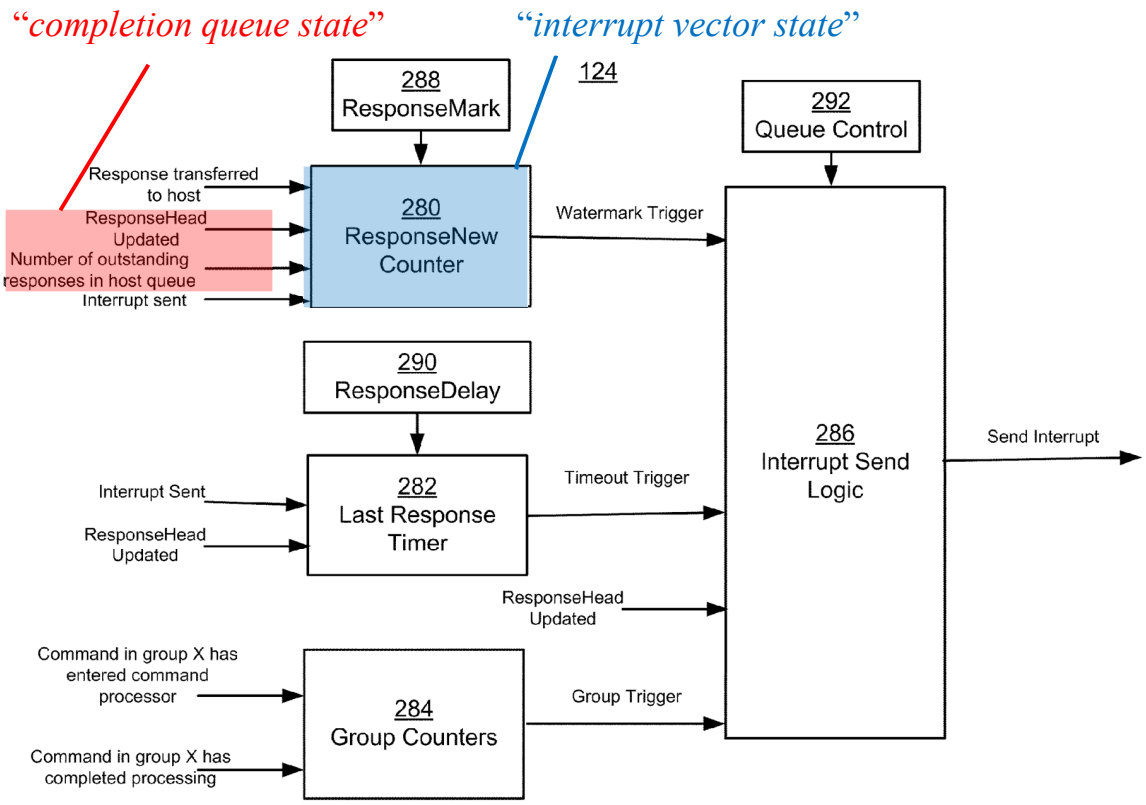


Borchers, Fig. 2 (annotated).

176. Thus, because Borchers' interrupt controller includes interrupt mechanisms for each separate command/response buffer pair, Borchers renders obvious "*wherein the interrupt manager controller is configured to map at least two completion queue states of the plurality of completion queue states to the interrupt vector state*" as claimed.

177. Additionally, Borchers teaches a "*plurality of completion queue states*" in that there are several pieces of status information, including that a Response has been transferred to the host, the response queue head pointer has been updated, and the number of outstanding responses in the host queue. At least

two of these pieces of information is mapped to one of the interrupt mechanisms. For example, the ResponseHead updated piece of information and number of outstanding response are mapped to the ResponseNew counter 280 as shown below.



Borchers, Fig. 2 (annotated).

178. Thus, because Borchers maps at least one of the response queue status items (ResponseHead Updated) to one of the interrupt mechanisms (ResponseNew Counter 280), Borchers renders obvious "*wherein the interrupt manager controller is configured to map at least two completion queue states of the plurality of completion queue states to the interrupt vector state*" as claimed.

12. Claim 10

[10.1] ***The nonvolatile memory controller of claim 8, wherein the interrupt manager controller is a sequential logic circuit.***

179. The specification of the ’968 patent describes a sequential logic circuit as a state machine: “In various embodiments, the interrupt manager 1040 is a hardware mechanism for generating interrupt message packets. For example, the interrupt manager controller 1100 in the interrupt manager 1040 may be a sequential logic circuit (e.g., a state machine).” ’968 patent, 17:55-59. A POSITA would have understood a state machine to be any discrete computing system such as a digital computer. *See e.g.,* IEEE Dictionary, 1102 (“**state machine** A model of a system in which all values are discrete, as in a digital computer.”). Borchers explains that the memory controller, of which the interrupt processor is a part, may be implemented in a variety of manners, all of which are some form of a state machine or “*sequential logic circuit*.”

> **The controller 110 may include a microcontroller, a FPGA controller, other types of controllers, or combinations of these controllers**. In one exemplary implementation, the controller 110 is a microcontroller. The microcontroller may be implemented in hardware, software, or a combination of hardware and software. For example, the microcontroller may be loaded with a computer program product from memory (e.g., memory module 116) including instructions that, when executed, may cause the microcontroller to perform in a certain manner. The microcontroller may be configured to receive commands from the

> host 106 using the interface 108 and to execute the commands. For instance, the commands may include commands to read, write, copy and erase blocks of data using the flash memory chips 118a and 118b, as well as other commands.
>
> **In another exemplary implementation, the controller 110 is a FPGA controller. The FPGA controller may be implemented in hardware, software, or a combination of hardware and software**. For example, the FPGA controller may be loaded with firmware from memory (e.g., memory module 116) including instructions that, when executed, may cause the FPGA controller to perform in a certain manner. The FPGA controller may be configured to receive commands from the host 106 using the interface 108 and to execute the commands. For instance, the commands may include commands to read, write, copy and erase blocks of data using the flash memory chips 118a and 118b, as well as other commands.

Borchers, [0113]-[0114].

> Method steps may be performed by one or more programmable processors executing a computer program to perform functions by operating on input data and generating output. **Method steps also may be performed by, and an apparatus may be implemented as, special purpose logic circuitry, e.g., a FPGA (field programmable gate array) or an ASIC (application-specific integrated circuit)**.

Borchers, [0127].

180. Thus, because Borchers' memory controller, and its associated interrupt processor may be implemented in a variety of manners, including FPGAs

or special logic circuitry, Borchers renders obvious "*wherein the interrupt manager controller is a sequential logic circuit*" as claimed.

13. Claim 11

[11.1] ***The nonvolatile memory controller of claim 8, wherein a completion queue state of the plurality of completion queue states includes a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller for the completion queue.***

181. For the reasons explained above at [1.5] and [8.1], Borchers renders obvious "*wherein a completion queue state of the plurality of completion queue states includes a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller for the completion queue*."

14. Claim 12

[12.1] ***The nonvolatile memory controller of claim 8, wherein a completion queue state of the plurality of completion queue states includes an event count indicating a number of completion message packets received by a host controller interface and including a completion status for the completion queue associated with the completion queue state.***

182. For the reasons explained above at [4.1], Borchers renders obvious "*wherein a completion queue state of the plurality of completion queue states includes an event count indicating a number of completion message packets received by a host controller interface*."

183. In addition, for the reasons explained above at [1.1] and [1.3], Borchers teaches "*a completion queue state of the plurality of completion queue states ... including a completion status for the completion queue associated with the completion queue state*."

15. Claim 13

[13.1] ***The nonvolatile memory controller of claim 12, wherein the interrupt vector state includes an aggregate event count for aggregating event counts of the completion queue state over a plurality of update operations, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate event count is below an aggregate event count threshold.***

184. For the reasons explained above at [5.1], Borchers renders obvious "*wherein the interrupt vector state includes an aggregate event count for aggregating event counts of the completion queue state over a plurality of update operations, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate event count is below an aggregate event count threshold*."

16. Claim 14

[14.1] ***The nonvolatile memory controller of claim 8, wherein the interrupt vector state includes an aggregate time indicator for indicating a time period between the last time the interrupt manager generated an interrupt message packet for the completion queues associated with the interrupt vector state, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate time indicator is below an aggregate time threshold.***

185. First, for the reasons explained above at [6.1], Borchers renders obvious "*wherein the interrupt vector state includes an aggregate time indicator*

for indicating a time period between the last time the interrupt manager generated an interrupt message packet for the completion queues associated with the interrupt vector state" as claimed.

186. Second, Borchers' interrupt processor 124 does not send an interrupt signal until the last response timer ("*aggregate time indicator*") reaches a time threshold ("*wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate time indicator is below an aggregate time threshold*").

> **In one exemplary implementation, the host 106, through its driver 107, may control when the interrupt processor 124 should generate interrupts**. The host 106 may use one or more different interrupt mechanisms, including a combination of different interrupt mechanisms, to provide information to the interrupt processor 124 regarding interrupt processing. **For instance, the host 106 through the driver 107 may configure the interrupt processor 124 to use** a water mark interrupt mechanism, **a timeout interrupt mechanism**, a group interrupt mechanism, or a combination of these interrupt mechanisms. In one exemplary implementation, the host 106 may set a ResponseMark parameter, which determines the watermark, and may set the ResponseDelay parameter, which determines the timeout. The host 106 may communicate these parameters to the interrupt processor 124. If the count of new responses in the response buffer 123 is equal to or greater than the ResponseMark, then an interrupt is generated by the interrupt processor 124 and the count is zeroed. **If the time (e.g.,**

> **time in microseconds) since the last interrupt is equal to or greater than the ResponseDelay and there are new responses in the response buffer 123, then the interrupt processor 124 generates an interrupt and the timeout is zeroed**. If the host 106 removes the new response from the response buffer 123, the count of new responses is updated and the timeout is restarted. In this manner, the host 106 may poll ahead and avoid interrupts from the interrupt processor 124.

Borchers, [0036]-[0037].

187. Thus, because Borchers uses a last response timer and does not send an interrupt signal until the timer reaches a Response Delay value, Borchers renders obvious "*wherein the interrupt manager is inhibited from generating an interrupt message packet if the aggregate time indicator is below an aggregate time threshold*" as claimed.

17. Claim 15

[15.1] ***The nonvolatile memory controller of claim 8, wherein a completion queue state of the plurality of completion queue states includes an empty queue indicator for indicating whether the completion queue is empty, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the empty queue indicator indicates the completion queue is empty.***

188. For the reasons explained above at [7.1], Borchers renders obvious "*wherein a completion queue state of the plurality of completion queue states includes an empty queue indicator for indicating whether the completion queue is empty, and wherein the interrupt manager is inhibited from generating an interrupt message packet if the empty queue indicator indicates the completion*

queue is empty."

18. Claim 16

[16.0] ***A method comprising:***

189. Borchers describes a method for using circular command queues for communication between a host and memory controller.

> In another general aspect, **a method for communicating commands between a host and a flash memory data storage device includes populating a circular command queue of a driver on the host with commands for retrieval by the data storage device**, transferring commands from the circular command queue to the data storage device via a device initiated direct memory access operation, populating, via a direct memory access operation initiated by the data storage device, a circular response queue of the host with responses by the data storage device for retrieval by the host device, where each response acknowledges the reception of a command from the host by the data storage device, and consuming responses from the circular response queue at the host.

Borchers, [0006].

190. Accordingly, Borchers teaches "*a method*" as claimed.

[16.1] ***generating a completion queue state by a nonvolatile memory controller for indicating the occurrence of a completion queue event associated with a completion queue in a host processing unit,***

191. For the reasons explained above at [1.1], Borchers renders obvious "generating a completion queue state by a nonvolatile memory controller for

indicating the occurrence of a completion queue event associated with a completion queue in a host processing unit" as claimed.

[16.2] ***the completion queue state including a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller for the completion queue;***

192. For the reasons explained above at [1.5], Borchers renders obvious "*the completion queue state including a doorbell update status indicating whether the host processing unit has performed a doorbell update event in which the host processing unit updates a head pointer stored in the nonvolatile memory controller for the completion queue*" as claimed.

[16.3] ***generating an interrupt vector state by the nonvolatile memory controller based on the completion queue state, the interrupt vector state associated with an interrupt vector in the host processing unit;***

193. First, for the reasons explained above at [1.2], Borchers renders obvious "*generating an interrupt vector state by the nonvolatile memory controller based on the completion queue state*."

194. Second, as explained above at [1.2], it would have been obvious for Borchers' interrupt signals to include interrupt vectors (i.e., a message address and message data). Accordingly, Borchers renders obvious "*the interrupt vector state associated with an interrupt vector in the host processing unit*" as claimed.

[16.4] ***determining by the nonvolatile memory controller based on the interrupt vector state that the completion queue contains an unprocessed completion status; and***

195. For the reasons explained above at [1.3], Borchers renders obvious "*determining by the nonvolatile memory controller based on the interrupt vector state that the completion queue contains an unprocessed completion status*."

[16.5] ***generating an interrupt message packet by a nonvolatile memory interface for triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue.***

196. For the reasons explained above at [1.4], Borchers renders obvious "*generating an interrupt message packet by a nonvolatile memory interface for triggering an interrupt in the host processing unit to alert the host processing unit of the unprocessed completion status in the completion queue*" as claimed.

19. Claim 17

[17.1] ***The method of claim 16, further comprising sending the interrupt message packet to the host processing unit.***

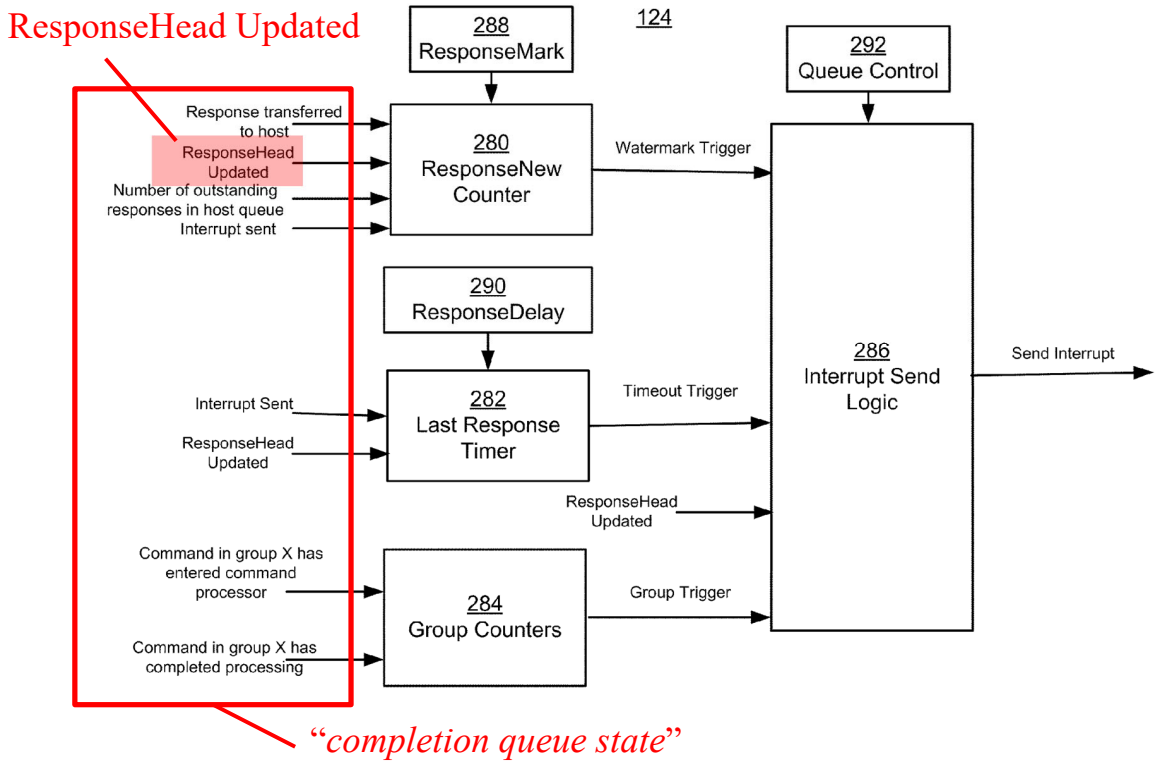
197. For the reasons explained above at [1.4], Borchers renders obvious "*sending the interrupt message packet to the host processing unit*" as claimed.

20. Claim 18

[18.1] ***The method of claim 16, wherein the completion queue event is the host processing unit updating a head pointer stored in the nonvolatile memory controller for the completion queue.***

198. First, as explained above at [1.1], Borchers receives status information for the response queue ("*completion queue*"). This status information includes that the host has updated the head pointer of the response queue. During operation, "[t]he host 106 is configured to read the responses from the response buffer 123

and **update the head pointer** in the MMIO space of the data storage device to indicate that the host has received the response." Borchers, [0035].



Borchers, Fig. 2 (annotated).

199. Accordingly, because the response queue status information ("*completion queue state*") includes an indication that the head pointer of the response queue has been updated, the status information includes "the host processing unit updating a head pointer stored in the nonvolatile memory controller for the completion queue."

B. Ground 2: Claims 1-18 are obvious in view of Borchers and the PCI standards.

200. Ground 1 shows how Borchers renders obvious each of the challenged

claim limitations. Ground 1 further relies on the PCIe/PCI standards as evidence of the background knowledge of a POSITA. To the extent Patent Owner argues that the PCIe/PCI standards would not have been within the background knowledge of a POSITA, Ground 2 is presented to show that combining the teachings of the PCIe/PCI standards with Borchers would have been obvious to a POSITA.

1. Summary of the PCI Standards

201. The PCI Express 2.0 Base Specification document "describes the PCI Express architecture, interconnect attributes, fabric management, and the programming interface required to design and build systems and peripherals that are compliant with the PCI Express specification." PCIe, 21. "The PCIe Standard states that "[t]he goal is to enable such devices from different vendors to inter-operate in an open architecture. The specification is intended as an enhancement to the PCI architecture spanning multiple market segments; Clients (Desktops and Mobile), Servers (Standard and Enterprise), and Embedded and Communication devices. The specification allows system OEMs and peripheral developers adequate room for product versatility and market differentiation without the burden of carrying obsolete interfaces or losing compatibility." PCIe, 21.

202. As an example, the PCIe Standard describes the structure of interrupt signals. PCIe describes a "Message Signaled Interrupt (MSI/MSI-X)" which are "Two similar but separate mechanisms that enable a device to request service by

writing a system-specified DWORD of data to a system-specified address using a Memory Write Request. Compared to MSI, MSI-X supports a larger maximum number of vectors and independent message address and data for each vector." PCIe, 26. *See also* PCIe, 458.

203. The PCIe express standards build upon and refer to the PCI base standard (Ex.1007). As stated in the PCI Express standard, "[a]ll PCI Express device functions that are capable of generating interrupts must implement MSI or MSI-X or both." PCIe, 453. "MSI and the MSI Capability structures are defined in the PCI Local Bus Specification, Revision 3.0." PCIe, 453.

204. POSITAs at the time the '968 patent was filed would have been familiar with various standards for memory systems, including the PCI and PCIe bus standards. Indeed, the background section of the '968 patent acknowledges that PCI and PCIe standards were known. *See* Ex.1001, 1:15-25. Moreover, the '968 patent explains that interrupt signals may comport with the "PCI Specification 3.0" and the "PCIe specification." '968 patent, 16:61-67. Other patents that predate the '968 patent refer to the PCI standards documents. US Patent Publication No. 2007/0245061, [0018] ("PCI Local Bus Specification Revision 3.0"), U.S. Patent No. 7,924,914 ("The PCI specification, revision 3.0, is available from the PCI special interest group"), U.S. Patent Publication No. 2010/0185719 ("PCIe version 2.0 was launched in 2007).

205. I have personal experience using and referring to the PCI standards throughout the 2000s. POSITAs, including myself, were very aware of these PCI standards documents because we designed our systems to be compatible with the commonly used PCI standards. I have reviewed the PCI documents in Ex.1006 and Ex.1007 and recognize them as the PCI standards documents that were available in the 2000s. POSITAs knew at the time that these documents could have been obtained on the PCI website (www.pcisig.com). For example, the Internet Archive shows that the PCI Express document (Ex.1007) was available on the PCI standards website (www.pcisig.com) as of at least February 8, 2007. *See* Ex.1014. The Internet Archive also shows that the PCI local bus specification document (Ex.1006) was available as of at least February 10, 2007. *See* Ex.1015.

206. I have also reviewed the metadata for Exhibits 1006 and 1007. The metadata for the PCI Express document (Ex.1007) indicates a creation date of September 9, 2006. The Metadata for the PCI local bus specification (Ex.1006) indicates a creation date of November 1, 2003. Accordingly, for all the reasons above, the PCI standards (Ex.1006 and Ex.1007) were publicly available years before the '968 patent was filed.

1. Reasons to combine the PCI standard with Borchers

207. A POSITA would have found it obvious to implement Borchers' techniques according to standardized practices such as those defined in the

PCIe/PCI standards.

208. Borchers describes a known interface device for communication between a host and memory controller. Borchers explains that "the high speed interface may include a PCIe interface." Borchers, [0110]. "The PCIe interface 108 may include a connector to the host 106 such as, for example, a PCIe connector cable assembly. Other high speed interfaces, connectors and connector assemblies also may be used." Borchers, [0110].

209. Accordingly, a POSITA would have looked to known techniques for implementing a PCI Express interface. It would have been obvious to look to the known standards documentation such as the PCI Express Base Standard and the PCI standard. Indeed, the PCI Express standard states that "describes the PCI Express architecture, interconnect attributes, fabric management, and the programming interface required to design and build systems and peripherals that are compliant with the PCI Express specification." PCIe, 21. Accordingly, it would have been natural for a POSITA to implement Borchers' memory interface 108 in accordance with industry standards.

210. For example, one aspect of the memory interface is how it implements interrupt signals. Borchers explains that "The interrupt processor 124 is configured to signal the host 106 with an interrupt when new responses are available in the response buffer 123." Borchers, [0035]. A POSITA would have found it obvious to

thus implement the interrupt signals according to the PCI Express standard. As stated in the PCI Express standard, "All PCI Express device Functions that are capable of generating interrupts must implement MSI or MSI-X or both." PCIe, 453. "MSI and the MSI Capability structures are defined in the PCI Local Bus Specification, Revision 3.0." PCIe, 453. PCIe describes a "Message Signaled Interrupt (MSI/MSI-X)" which are "Two similar but separate mechanisms that enable a device to request service by writing a system-specified DWORD of data to a system-specified address using a Memory Write Request. Compared to MSI, MSI-X supports a larger maximum number of vectors and independent message address and data for each vector." PCIe, 26. *See also* PCIe, 458

211. The PCI Local Bus Specification also explains that MSI and MSI-x interrupts use vectors. "System software initializes the message address and message data (from here on referred to as the '**vector**') during device configuration, allocating one or more vectors to each MSI capable function." PCI, 231. Accordingly, it would have been obvious that Borchers' interrupt signals, implemented according to the PCI Express standard, would have used interrupt vectors.

212. Accordingly, the combination of Borchers and the PCIe/PCI standards would have predictably resulted in interrupt signals implemented using interrupt vectors. A POSITA would have had a reasonable expectation of success because

the PCIe/PCI standards are tried-and-true, industry vetted implementations. The PCIe standard is "an enhancement to the PCI architecture" which was "the most widely accepted and implemented expansion standard in the world." PCIe, 21; PCI, 15.

213. Thus, the combination of Borchers and the PCIe/PCI standards are merely applying a known technique (interrupt signals according to PCIe/PCI standards) to a known device (Borchers' memory interface) ready for improvement to yield predictable results (interrupt signals with interrupt vectors).

2. Claim Analysis for claims 1-18

214. The claim analysis for Ground 2 is substantially the same as that of Ground 1. *See supra* VIII.A.

C. Ground 3: Claims 1-18 are obvious in view of Borchers, the PCIe/PCI standards, and Serebin.

215. Grounds 1 and 2 rely on the background knowledge of a POSITA that the timer values, counter values, and other data that represent response queue status information ("*completion queue state*") and interrupt mechanism information ("*interrupt vector state*") would be stored in some type of storage. It would have been obvious for this storage to be a register, flag, or other form of memory. To the extent Patent Owner argues that such is not background knowledge of a POSITA, Ground 3 shows that it would be obvious for an interrupt processor to include storage such as registers to temporarily store such status

information, as evidenced by Serebrin.

1. Summary of Serebrin

216. Like Borchers and the ’968 patent, Serebrin relates to “a device interrupt manager for a “memory management unit.” Serebrin, [0008]. Serebrin describes “a memory controller 42 and an input/output (I/O) memory management unit (IOMMU) 40 that further includes a device interrupt manager 38.” Serebrin, [0051]; Serebrin Provisional, 19. The interrupt manager is connected to memory 56 as shown in Fig. 2 below. The memory 56 stores gAPIC (guest Advanced Programmable Interrupt Controller) states. The “device interrupt manager 38 may be configured to record each guest interrupt received in the bridge 36 in the gAPIC state data structure 58 [and] update the gAPIC state in the gAPIC state data structure 58.” Serebrin, [0055], Serebrin Provisional, 21.

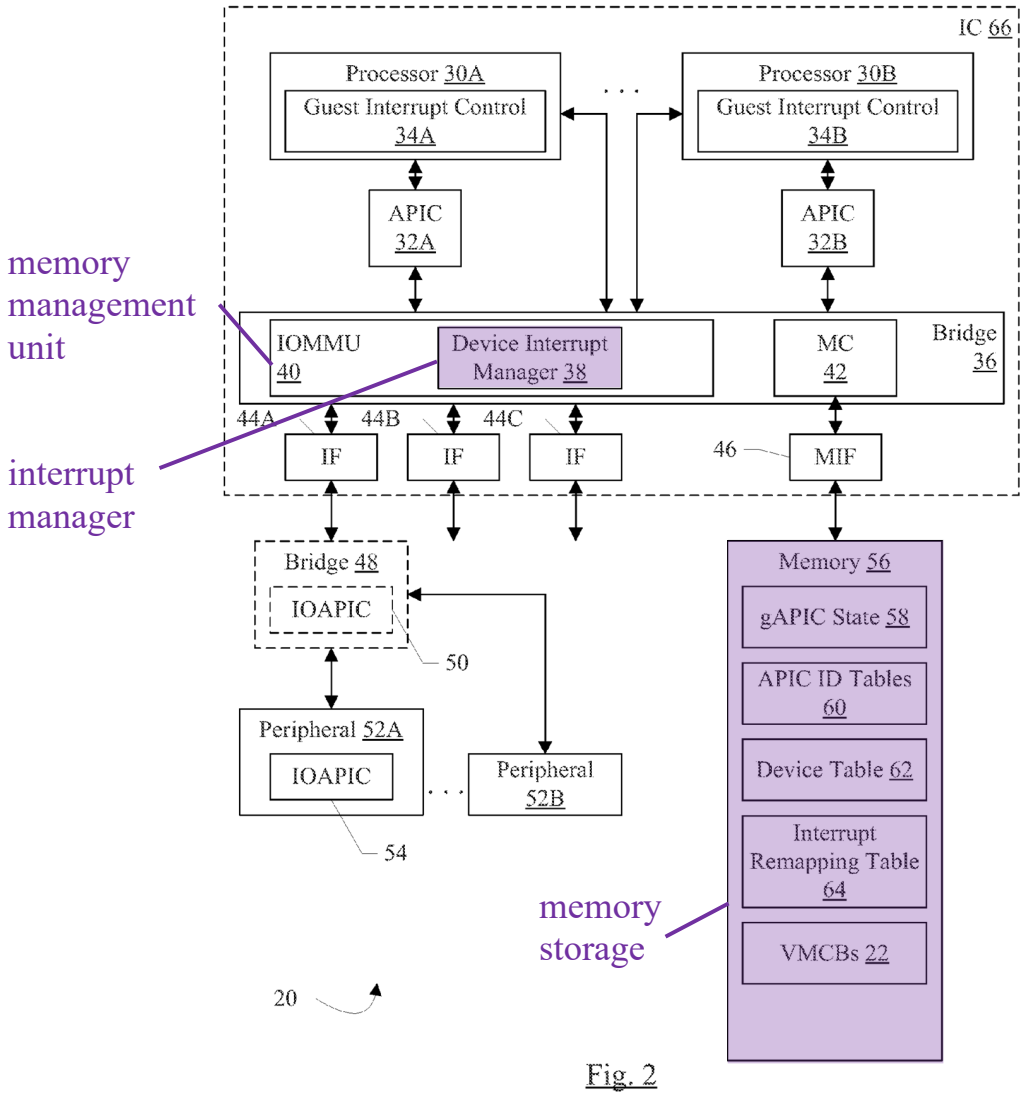


Fig. 2

Borchers, Fig. 1C (annotated).

217. Serebrin explains that a variety of forms of memory may be used to store the interrupt state information (i.e., the gAPIC state):

> While the gAPIC state data structure 58 is shown as stored in memory 56 in the illustrated embodiment, portions may be cached by the device interrupt manager 38 and/or a cache accessible to the bridge 36 in some

> embodiments. In addition or alternatively, a dedicated memory for one or more gAPIC state entries may be implemented in the bridge 36.

Serebrin, [0057]; Serebrin Provisional, 22.

218. Accordingly, Serebrin provides evidence that it was known for interrupt managers to use cache to store state information. The state information described by Serebrin is similar to the state information used by Borchers' interrupt processor.

> The gAPIC state entry 90 may include a complete copy of the APIC state. Not all state is illustrated in FIG. 5. However, the IRR, in-service register (ISR), the trigger mode register (TMR) and ICRH and ICRL registers, the task priority register (TPR) and the end of interrupt (EOI) registers are shown, along with other APIC state. The other APIC state may be interleaved within the illustrated State (e.g. according to the offsets for each register in the APIC address map). Furthermore, the order of the state as shown in FIG.5 may not be order of the state as stored in memory.
>
> The IRR register records the interrupt requests that have been transmitted to the gAPIC ... The other APIC state may include internally-generated interrupts, timers, the local vector table, etc.

Serebrin, [0090]-[0094]; Serebrin Provisional, 34-35.

219. Accordingly, Serebrin provides evidence that it was known for interrupt managers to use cache or registers to store state information.

2. Serebrin is Entitled to its Provisional Filing Date.

220. It has been explained to me that in order for the Serebrin reference to be entitled to its provisional filing date, the provisional filing date must support at least one claim. As one example, Claim 1 of the Serebrin publication recites:

> A device interrupt manager configured to receive an interrupt from a device that is assigned to a guest, wherein the device interrupt manager is configured to transmit an operation for recordal of the interrupt in a memory location, wherein the memory location is associated with a virtual processor within the guest, wherein the virtual processor is targeted by the interrupt.

Serebrin, claim 1.

221. The Serebrin Provisional is nearly identical to the Serebrin publication. As such, the Serebrin Provisional provides written description support for this claim. The following table illustrates this.

Serebrin Publication, Claim 1	Serebrin Provisional.
A device interrupt manager configured to receive an interrupt from a device that is assigned to a guest,	"A device interrupt manager configured to receive an interrupt from a device that is assigned to a guest." Serebrin Provisional, 48.
wherein the device interrupt manager is configured to transmit an operation for recordal of the interrupt in a	"The device manager may be configured to record the interrupt in a guest interrupt controller data structure." Serebrin Provisional, [0023], 10.

memory location,	
wherein the memory location is associated with a virtual processor within the guest,	In an embodiment, a computer system includes a VMM and one or more guests executing in virtual machines. A peripheral device may be assigned to a guest, and may generate device interrupts to be delivered to the guest. A device interrupt manager may be configured to detect that the device interrupt is targeted at a guest (as a opposed to a host interrupt targeted at the VMM or host software on which the VMM executes, if any). More particularly, the device interrupt manager may be configured to detect which vCPU within the guest is targeted by the interrupt. The device manager may be configured to record the interrupt in a guest interrupt controller data structure. If the targeted vCPU within the guest is currently executing on a hardware processor in the computer system, the hardware processor may detect the guest interrupt and may redirect the guest/vCPU to service the interrupt. Serebrin Provisional, [0023], 10.
wherein the virtual processor is targeted by the interrupt.	A processor or vCPU may be targeted by an interrupt if the interrupt is to be received by the

	interrupt controller associated with the processor/vCPU. The processor/vCPU may not necessarily process the interrupt, but may be a candidate for processing and may participate in determining which processor(s)/vCPU(s) will process the interrupt. An interrupt may explicitly specify its target(s) (e.g. with a physical or logical ID), or may be a broadcast interrupt that targets all processors/vCPUs. Generally, an interrupt may be referred to as accepted if the interrupt controller associated with a targeted processor has recorded the interrupt for delivery to the processor. That is, at some point in time after acceptance, the interrupt will be delivered to the processor. Interrupting the processor to service the interrupt may be referred to as delivering the interrupt. The vCPU and/or corresponding virtual interrupt controller and/or the pair may be referred to herein more succinctly as the destination of an interrupt within the guest. The destination may ultimately be the vCPU that is to service the interrupt, but the corresponding virtual interrupt controller may also be viewed as the destination since it is associated with the corresponding processor and records the interrupt.

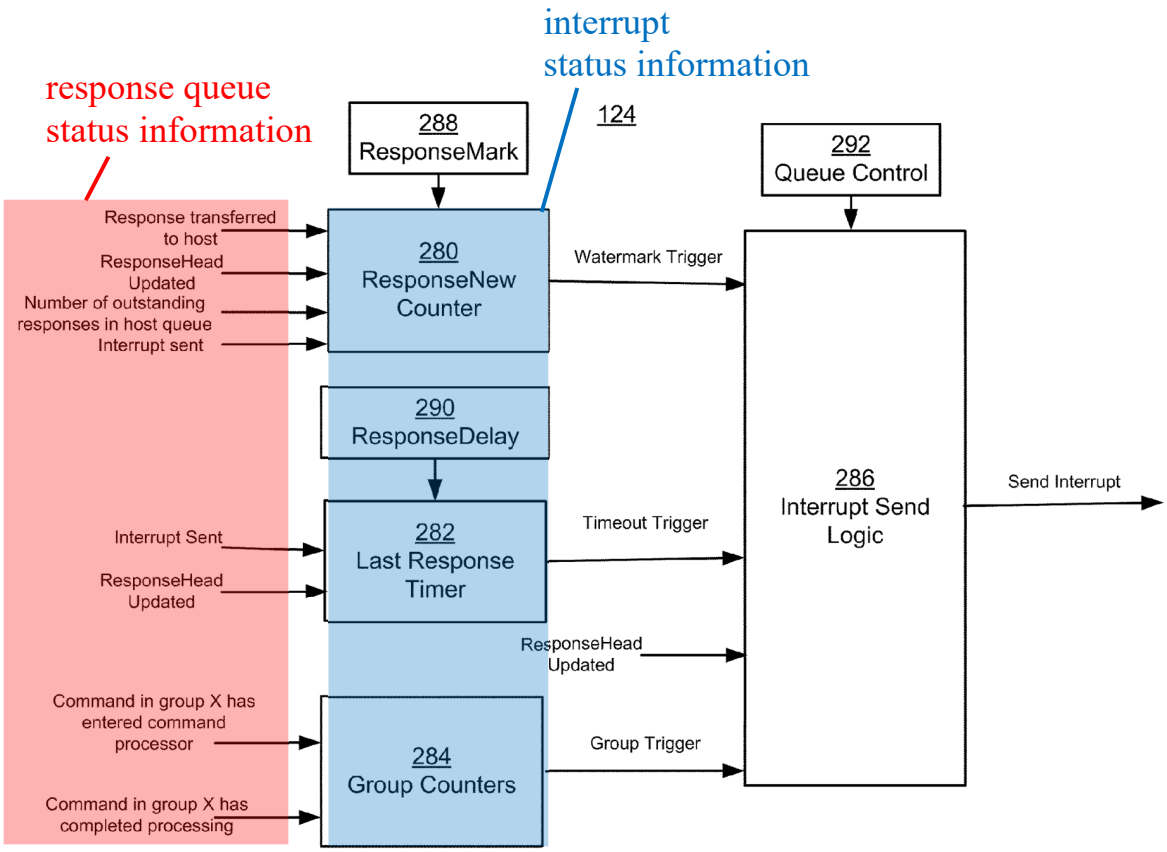
	Serebrin Provisional, [0027], 12.

222. Accordingly, because the Serebrin Provisional provides written description support for each element of the claims, it is entitled to the filing date of its provisional application. For the parts of Serebrin that my analysis relies upon, I have additionally cited to the Serebrin provisional.

3. Reasons to combine Serebrin with Borchers and the PCI standards

223. A POSITA would have found it obvious for Borchers' interrupt processor to utilize cache or registers to store state information, as evidenced by Serebrin.

224. As described above at VIII.A.1, Borchers' memory controller includes an interrupt processor 124. The interrupt processor 124 relies on status information from the host's response queue. The status information is then used by the interrupt processor 124 to maintain interrupt status information (e.g., timers and counters) for determining when to send an interrupt signal to the host. *See* Borchers, [0060].



Borchers, Fig. 2 (annotated).

225. A POSITA would have understood that both the response queue status information and the interrupt status information would be stored or buffered in some type of storage location, such as a cache or register. Indeed, it was known for interrupt managers to use caches or registers to store state information. Serebrin provides an example of using caches or registers to store state information.

226. Serebrin describes "a memory controller 42 and an input/output (I/O) memory management unit (IOMMU) 40 that further includes a device interrupt manager 38." Serebrin, [0051]; Serebrin, 19. The interrupt manager is connected to memory 56 as shown in Fig. 2 below. The memory 56 stores gAPIC (guest

Advanced Programmable Interrupt Controller) states. The "device interrupt manager 38 may be configured to record each guest interrupt received in the bridge 36 in the gAPIC state data structure 58 [and] update the gAPIC state in the gAPIC state data structure 58." Serebrin, [0055]; Serebrin Provisional, 21. Serebrin explains that a variety of forms of memory such as cache or registers may be used to store the interrupt state information. *See e.g.,* Serebrin, [0057]; Serebrin Provisional, 22.

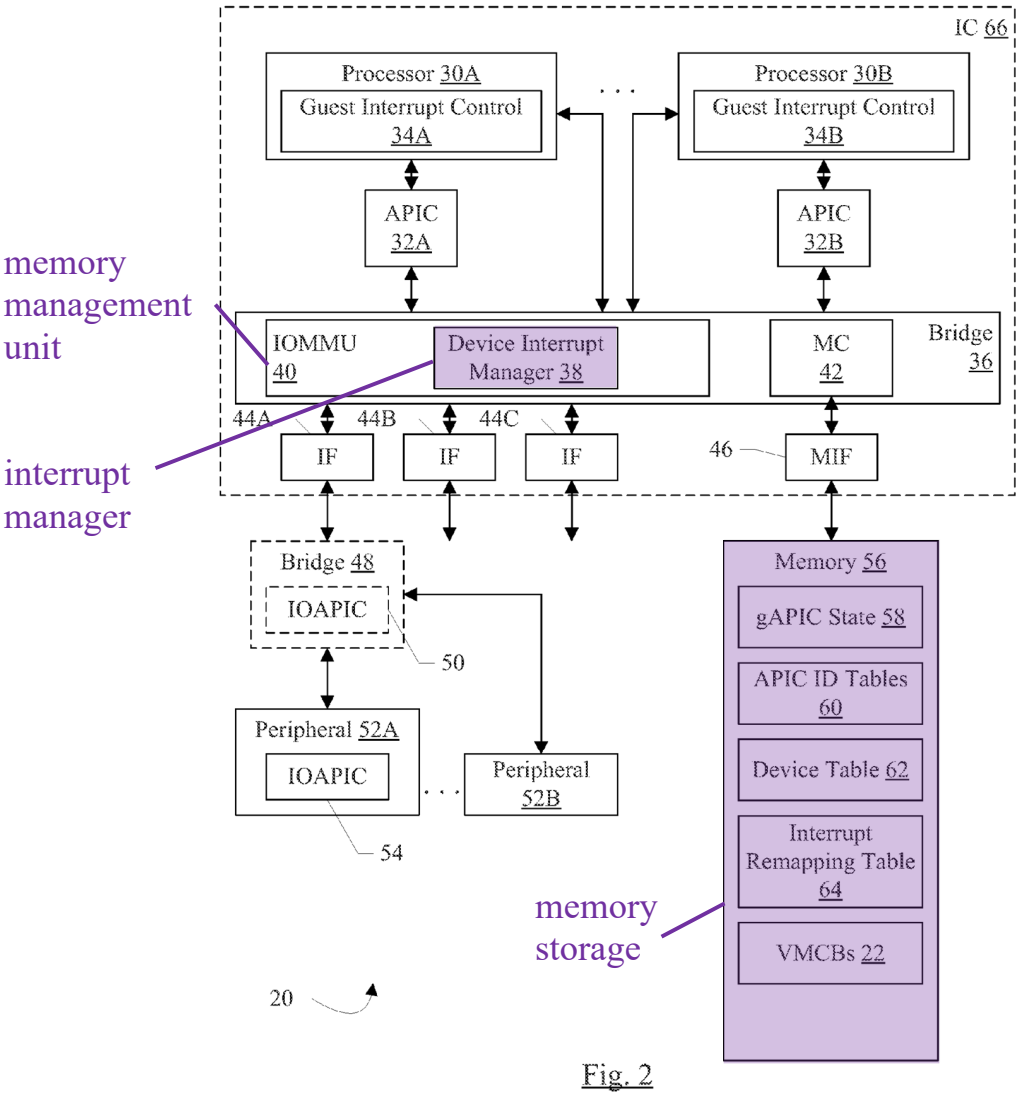


Fig. 2

Serebrin, Fig. 2 (annotated).

227. Accordingly, a POSITA would have found it obvious for Borchers' interrupt manager to similarly include cache/registers for storing state data. The combination of Serebrin and Borchers thus predictably results in Borchers' response queue status information and interrupt mechanism information being placed in storage such as cache or registers.

228. A POSITA would have had a reasonable expectation of success given that Serebrin stores similar types of state information as Borchers. In particular, Serebrin's interrupt manager stores state information such as "the interrupt requests that have been transmitted to the gAPIC" and "internally-generated interrupts, timers, the local vector table, etc." Serebrin, [0090]-[0094]; Serebrin Provisional, 34-35.

229. Accordingly the combination of Serebrin with the teachings of Borchers is merely applying a known technique (Serebrin's cache/registers) to a known device (Borchers' interrupt manager) ready for improvement to yield predictable results (storing state information in cache/registers).

4. Claim Analysis for claims 1-18

230. The claim analysis for Ground 2 is substantially the same as that of Ground 1. *See supra* VIII.A.

IX. CONCLUSION

231. This declaration and my opinions herein are made to the best of my knowledge and understanding, and based on the material available to me, at the time of signing this declaration. I declare that all statements made herein on my own knowledge are true and that all statements made on information and belief are believed to be true, and further, that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 or Title 18 of the United States Code.

Date: January 21, 2025

Paul Min