PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 19/00 | A1

(11) International Publication Number: WO 97/27558

(43) International Publication Date: 31 July 1997 (31.07.97)

(21) International Application Number: PCT/US97/01104

(22) International Filing Date: 24 January 1997 (24.01.97)

(30) Priority Data:

60/010,549	25 January 1996 (25.01.96)	US
08/680,428	15 July 1996 (15.07.96)	US

(81) Designated States: CN, JP, KR, RU, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.

(71) Applicant: PATHLIGHT TECHNOLOGY INC. [US/US]; 767 Warren Road, Ithaca, NY 14850-1255 (US).

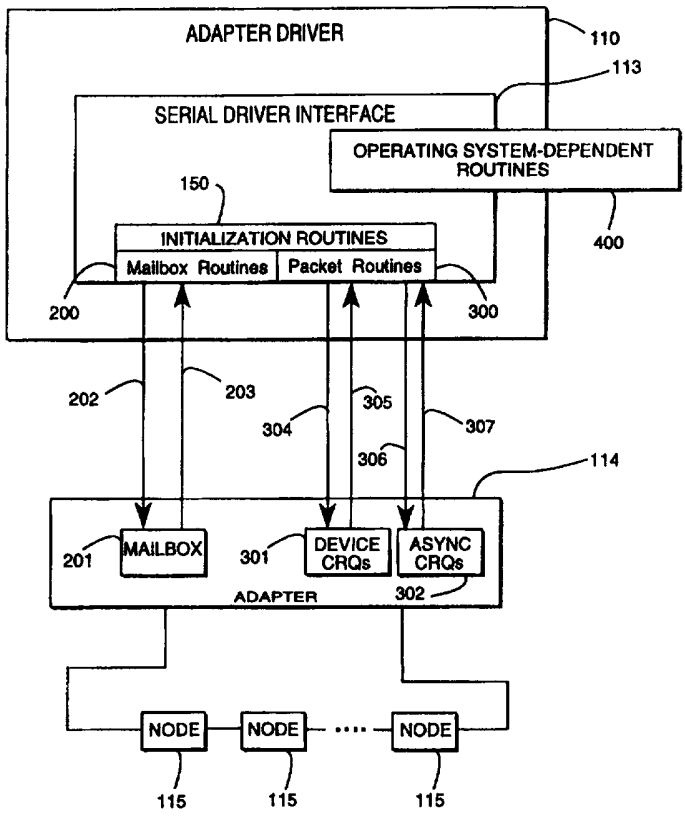
(72) Inventors: DRACUP, Andrew, D.; 304 Meadow Wood Terrace, Ithaca, NY 14850 (US). KELLEHER, Terence, M.; 10 Ringwood Court West, Ithaca, NY 14850 (US). SCAFFIDI, Salvatore, G., Jr.; 115 Creamery Road, P.O. Box 81, Harford, NY 13784 (US). KHEZRI, Said, Rahmani; 700 Warren Road 18-1F, Ithaca, NY 14850 (US).

(74) Agent: BURR, Stephen, P.; Parkhurst, Wendel & Burr, L.L.P., Suite 210, 1421 Prince Street, Alexandria, VA 22314 (US).

(54) Title: SERIAL DRIVER INTERFACE FOR MODULAR I/O SOFTWARE ARCHITECTURE

(57) Abstract

An operating system (100) and hardware-independent serial driver interface (113) receives device driver I/O requests in accordance with a first protocol, translates the request in accordance with a first protocol, translates the request into a second request according to a second protocol, assembles data structure, pointed to by a pointer, representative of the I/O requests, and sends the pointer to an adapter (110) thus enabling the adapter to pass I/O request, in the appropriate protocol, to a device connected to the adapter (110).



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	**GB**	United Kingdom	**MW**	Malawi
AT	Austria	**GE**	Georgia	**MX**	Mexico
AU	Australia	**GN**	Guinea	**NE**	Niger
BB	Barbados	**GR**	Greece	**NL**	Netherlands
BE	Belgium	**HU**	Hungary	**NO**	Norway
BF	Burkina Faso	**IE**	Ireland	**NZ**	New Zealand
BG	Bulgaria	**IT**	Italy	**PL**	Poland
BJ	Benin	**JP**	Japan	**PT**	Portugal
BR	Brazil	**KE**	Kenya	**RO**	Romania
BY	Belarus	**KG**	Kyrgystan	**RU**	Russian Federation
CA	Canada	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SG**	Singapore
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovakia
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LR**	Liberia	**SZ**	Swaziland
CS	Czechoslovakia	**LT**	Lithuania	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**LV**	Latvia	**TJ**	Tajikistan
DK	Denmark	**MC**	Monaco	**TT**	Trinidad and Tobago
EE	Estonia	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**UG**	Uganda
FI	Finland	**ML**	Mali	**US**	United States of America
FR	France	**MN**	Mongolia	**UZ**	Uzbekistan
GA	Gabon	**MR**	Mauritania	**VN**	Viet Nam

Serial Driver Interface for Modular I/O Software Architecture

Field of Invention

The present invention relates in general to data processing and computer systems and in particular to methods and apparatus for implementing an operating system and hardware independent software interface for serial data input/output (I/O).

Description of the Prior Art

As microprocessors and storage devices grow in speed and capacity, many computer users realize that current storage and networking capabilities and standards are failing to keep pace and are becoming critical performance bottlenecks.

For example, video and broadcast providers would like to avoid the so-called "sneakernet" wherein stored images on tape must be transferred by hand from one site to another. There is a demand, therefore, for a purely digital studio. However, for the digital studio to become a reality, disk storage systems must be capable of capturing data at full frame rates without data compression and, similarly, data distribution systems must be able to transfer video data between systems at comparable speeds.

Another industry requiring data transfer and storage beyond current capabilities is the digital pre-press industry. Digital pre-press systems are typically called

upon to move enormous image data files between editing stations, Raster Image Processing systems (RIPs), and in many cases, digital press controllers. The size of these image data files is often on the order of hundreds of megabytes per image. It is not uncommon, therefore, that at present data transfer rates the transfer of these images can take minutes apiece.

Furthermore, it is noteworthy that the speed of server systems is also affected by current storage subsystem input/output (I/O).

Accordingly, better storage and connectivity solutions are needed in order to address the problems set forth above.

One emerging solution to the above problem is the new Serial Storage Architecture (SSA) and connectivity technology. A full explanation and specification of the SSA is set out in SSA-Industry Association Reference Documents:

1. SSA-IA/95PH Serial Storage Architecture - 1995 Physical, October 12, 1995 (Rev. 3)

2. SSA-IA/95SP Serial Storage Architecture - 1995 SCSI -2 Protocol, October 11, 1995 (Rev. 3)

3. Serial Storage Architecture Physical Layer 1 (SSA-PH1) x3T10.1/1145D, March 29, 1996 (Rev. 9a)

4. Serial Storage Architecture Transport Layer 1 (SSA-TL1) x3T10.1/0989D, February 28, 1996 (Rev. 10)

5. Serial Storage Architecture SCSI-2 Protocol

(SSA-S2P) X3T10.1/1121D, February 28, 1996 (Rev. 7)

These above documents are all incorporated herein by reference. Generally, compared to aging technologies such as the Small Computer System Interface (SCSI), SSA offers six times the performance, eight times the device connectivity, fault tolerance, autoconfiguration, simplified cabling and connections, lower cost, and dramatically increased efficiency. SSA technology is therefore becoming a preferred architecture choice to realize the desired data transfer and storage speed requirements.

Similarly, serial fibre channel technology is presently being implemented to achieve improved data transfer and storage speeds. A full explanation of the fibre channel technology is set out in the following documents:

1. Fibre Channel Arbitrated Loop (FC-AL), March 26, 1996 (Rev. 5.1)

2. Fibre Channel Physical and Signaling Interface (FC-PH) x3.230-1994

3. Fibre Channel Single Byte Command Code Sets (SBCCS) Mapping Protocol (FC-SB), May 26, 1995 (Rev. 3.4)

4. Fibre Channel Private Loop Direct Attach (FC-PLDA), January 22, 1996 (Rev. 1.0)

5. SCSI-3 Fibre Channel Protocol for SCSI (FCP), May 30, 1995 (Rev. 12)

These documents are also incorporated herein by

reference.

However, in order to replace current I/O technology, e.g. SCSI, on a host/peripheral computer system with, for example, SSA or fibre channel, not only do the physical I/O adapter cards need to be replaced with serial adapter cards supportive of SSA or fibre channel, but also both software device drivers and adapter drivers need to be replaced with drivers in accordance with the SSA or fibre channel standard. The time and expense of redeveloping these drivers, however, can become prohibitive.

Therefore, a need exists for a method and apparatus whereby serial architecture-compatible adapter drivers can be developed inexpensively and quickly and which can operate with existing device drivers.

SUMMARY OF THE INVENTION

It is an objective of the present invention to provide a serial driver interface which is operable with existing device drivers and is portable across multiple operating systems.

It is a further object of the present invention to simplify development of new serial I/O adapter drivers.

Other objects of the present invention will be apparent from the remainder of the specification and claims.

The preferred embodiment of the present invention is a computer operating system and hardware-independent

process for implementing input/output (I/O) between protocol-incompatible device drivers and targets. The invention comprises the following steps.

An I/O request is received according to a first protocol from a device driver and is translated into a second I/O request according to a second protocol. A data structure, pointed to by a first pointer, is assembled and includes the second I/O request, a unique token value, and a second pointer to an operating system-dependent completion function. The first pointer is then sent to an adapter which is forced to send the second I/O request to the target. Upon receiving a response from the target, the process associates the response with the previously assembled data structure by way of the unique token value. Finally, the operating system-dependent completion function is called to process the response in accordance with the operating systems needs.

Thus, the present invention allows, for example, the interface of parallel SCSI device drivers and SSA or fibre channel compatible devices or nodes. Furthermore, the invention can also pass original SSA commands directly to SSA compatible devices by simply eliminating the translating step, or original fibre channel commands directly to fibre channel compatible devices.

Brief Description of the Drawings

The advantages and features of the present invention

will become more clearly apparent from the following description of an illustrative embodiment of the invention, given as a non-restrictive example only and represented in the accompanying drawings, in which:

FIG. 1 shows a typical parallel I/O software architecture and associated peripheral connectivity according to the prior art.

FIG. 2 shows a serial I/O software architecture and associated peripheral connectivity according to the prior art.

FIG. 3 depicts a serial I/O software and hardware architecture according to the present invention.

FIGs. 4a-c set forth the preferred data structures in accordance with the present invention.

FIGS. 5-8 set forth software routine prototypes in accordance with the present invention.

FIG. 9 shows defined registers for an interface controller in accordance with the present invention.

FIGs. 10-24 and 26-31 show flow diagrams for implementing the adapter driver use of serial driver interface software routines.

FIG. 25 shows a data structure comprising information representative of each adapter and device connected to the adapter.

FIGs. 32-39 show flow diagrams for implementing the adapter driver-to-adapter interface.

Detailed Description of the Preferred Embodiment

FIG. 1 shows a typical parallel input/output (I/O) software architecture according to the prior art. The operating system/application 100 communicates with any of nodes 104 via device driver 101, adapter driver 102, and adapter 103. Communication between the operating system/application 100 and adapter 103 is accomplished via common bus 105. Nodes 104 can be storage devices such as hard drives or tape drives or other similar devices, or alternatively, nodes 104 can be other host computer systems. The device driver 101, adapter driver 102, and adapter 103 work in conjunction to effect communication according to a standard parallel protocol such as the Small Computer System Interface (SCSI), SCSI-2, or any other standard parallel protocol known in the art.

FIG. 2 shows a typical serial I/O software architecture according to the prior art. Referring to Fig. 2, the operating system/application 100 communicates with any of serial I/O compatible nodes 115 via device driver 111, adapter driver 112, and adapter 116. Communication between the operating system/application 110 and the adapter 116 is accomplished via common bus 105. The architecture depicted in FIG. 2 is particularly suited to effect communication between the operating system/application 100 and nodes 115 using the Serial Storage Architecture (SSA). However, one skilled in the

art will recognize that to take advantage of SSA for example, device driver 111 and adapter driver 112 must be redeveloped in accordance with SSA protocol standards. This can lead to great expense and delay in implementing serial I/O.

The present invention, however, facilitates and reduces the cost of implementing serial I/O by keeping existing device driver 111, developed in accordance with, for instance, the SCSI-2 protocol, unmodified.

A preferred embodiment of the present invention is illustrated in FIG. 3. As shown in FIG. 3, the serial driver interface (SDI) 113 is incorporated within the adapter driver 110. Generally, the SDI 113 comprises a set of routines and data structures that provide a high-level interface between the adapter driver 110 and adapter 114. As will be further explained herein, the SDI 113 and adapter 114 provide all the necessary translation services, SSA protocol services, and transport and physical support required to transform, for example, SCSI-2 command/response exchanges into SSA-compatible command/response exchanges. One skilled in the art will recognize that any discussion in this specification referring to SSA alone is also equally applicable to fibre channel technology.

The SDI 113 provides routines for initializing itself, communicating with adapter 114, and interrupt handling. Several provided routines have an interface

but not an implementation. These routines are "operating system-dependent" as will be explained hereinafter.

More specifically, as shown in FIG. 3, the SDI 113 includes initialization routines 150, mailbox routines 200, packet routines 300, and operating system-dependent routines 400 to effect the aforementioned functions. In the preferred embodiment, the adapter 114 includes firmware and sufficient memory capacity for a mailbox 201 comprising random access memory (RAM), device command/response queues (CRQs) 301 and an asynchronous event command/response queue (Async CRQ) 302.

Initialization routines 150 are used to initialize the SDI 113. Mailbox routines 200 are used for obtaining information about the adapter 114 and the serial I/O-compatible nodes 115 attached to the adapter 114. Packet routines 300 are used for sending, for example, SCSI-2 SSA message structures (SMSs) to a node 115 through the adapter 114. Finally, the operating system-dependent routines 400 are used to implement the SDI 113 in different operating systems. Preferred data structures to be used with the SDI 113 and the various routines mentioned above will next be discussed in detail.

SDI Data Structures

With reference to FIGs. 4a-c, the following data structures are defined for use with the SDI 113. One skilled in the art will recognize the hierarchical

organization of the various structures as defined using the ANSI C programming language. In operation of the present invention, the defined data structures are located in host DMA memory, accessible to both the computer operating system and adapter 114. A discussion of the elements of each data structure is set forth hereinafter.

The data structure sdi_xt_packet is created by the SDI 113 and sent to a device CRQ, or Async CRQ. sdi_xt_packet is one of the following types: sdi_xt_ssa_packet or sdi_xt_async_packet, both of which are discussed hereinafter. Complete_callback is a pointer to a function called by adapter driver 110 when the packet has been completed. The adapter driver 110 must set the callback function in each packet. link_space is used for inserting and removing the packet from internal queues. phys_addr is the physical address of the packet. driver_space is a 16 byte space allocated for use by the adapter driver 110. For example, this is used by a Netware driver to link the packet to a corresponding Host Adapter Control Block (HACB).

The data structure sdi_xt_ssa_packet contains the request and response information necessary to issue an SSA packet to a target, that is a CRQ. The request section contains the SSA SCSI-2 SMS and data pointers that the adapter 114 requires in order to process the request. The response section contains completion status

information.

The data structure sdi_xt_ssa_request contains the command bytes (SMS) and data pointers that the adapter 114 requires in order to process the SMS to the node 115. crq_num is the CRQ associated with the destination node 115. Each node's CRQ is found by calling sdi_get_next_device discussed later in this specification. token is a 32-bit field that is used to obtain addressability to the packet during response processing. When the adapter 114 completes a request, it will return the token to the SDI 113. The SDI 113 casts token to a pointer to an sdi_xt_packet. The token field should be set to the appropriate value, typically the virtual address of the sdi_xt_packet. timeout is the timeout value in milliseconds for the packet. dma_address is the physical address of the data to be transferred. For requests that do not involve data transfer, this field should be set to zero. dma_size is the size in bytes of the data transfer to be transferred. For requests that do not involve data transfer, this field should be set to zero. dma_type is the type of the data transfer. For example, constants representative of normal or scatter/gather types can be used. sms is the sdi_xt_scsi2_sms discussed hereinafter.

The data structure sdi_xt_ssa_response contains the response information associated with a sdi_xt_ssa_request. token is a copy of the token in the

associated sdi_xt_ssa_request. The remaining elements in sdi_xt_ssa_response are protocol-dependent. Using SCSI-2 as an example, reason is the completion reason. state holds the status bits indicating the command progress at completion. If the command results in the device returning a SCSI-2 status SSA message, then scsi_status is set to the status field from that message. residual is set to the number of bytes not transferred if the command completes successfully, but with less data transferred than was requested.

The data structure sdi_xt_async_packet contains the request and response information necessary to queue an asynchronous packet to the adapter 114. Queue Async packets are returned to the SDI 113 when asynchronous events occur. Like the non-Async packet described above, sdi_xt_async_packet comprises a request section and a response section.

The data structure sdi_xt_async_request contains the CRQ number and a token. crq_num should be set to a constant representative of the async CRQ ID. token contains a value as explained above with regard to sdi_ssa_xt_request.

The data structure sdi_xt_async_response contains asynchronous event information returned from the adapter 114. token contains a value as described above with regard to sdi_ssa_xt_response. reason contains the type of async packet. Constants representative of the

following events can be defined: the adapter 114 has detected a new node 115 on the SSA web, the adapter 114 has detected that a node 115 was removed from the SSA web, an adapter 114 log event has occurred; the data in log_num is valid, or an adapter 114 or node 115 error event has occurred; or the data in error_num is valid. new_target contains information about a new target that was added to the web. del_target contains information about a target that was deleted from the web. log_num contains a log message representative of, for example, a web reconfiguration. err_num contains an error message representative of, for example, total reset, absolute reset, or internal error of a node 115.

The data structure sdi_xt_sg_element is an entry in a scatter/gather list that is sent to the adapter 114. The scatter/gather capability of adapter 114 can be found by calling sdi_get_scatter_gather_info described hereinafter. sg_buf_size is the length of the scatter/gather request in bytes and sg_buf_phys_addr is the physical address of the request buffer. The data structure sdi_xt_hba_info is used to return information about a host bus adapter (HBA), or adapter 114. The structure comprises firmware and hardware revision numbers and a unique ID assigned to the adapter 114.

The data structure sdi_xt_scsi2_sms is the last structure in sdi_xt_ssa_request. Each element of this

particular structure is disclosed by the SSA documentation cited earlier, incorporated herein by reference.

Initialization Routines

Initialization routines 150 are provided to initialize the SDI 113 and the adapter 114, query adapter 114 attributes, and reset the adapter 114. FIG. 5 shows ANSI C source code prototypes for each of the initialization routines 150 operable with the SDI 113. One skilled in the art will recognize the role of each of the following routines as it is further explained herein. Further, the use or sequence of use of the initialization routines 150 will be discussed hereinafter by way of flow charts.

The routine sdi_get_hba_object_size returns the size in bytes of an adapter data object. The size returned is used to allocate a block of memory that will be used by the SDI 113 to internally manage the adapter 114. The adapter's pointer is a parameter to most of the SDI 113 routines. As can be seen from the prototype in FIG. 5, the routine has no parameters.

The routine sdi_initialize detects the presence of an adapter 114 at a specified host bus adapter (HBA). If an adapter 114 is found, the SDI 113 initializes a data structure pointed to by a pointer. The parameter hba_number is the index of an HBA and adptr is a pointer

to an HBA object. The parameter caller_context is a pointer to a driver-defined object and is stored in the SDI 113 for each HBA. The pointer is passed to the operating system dependent routines 400 discussed hereinafter. The pointer should be set to a driver-defined object or to null if no object is required. The routine sdi_initialize will return a value of 0 if an HBA was detected and successfully initialized. Otherwise, the return value is a constant indicative of one of the following errors: device not found, bad vendor ID, unknown, or bad register.

The routine sdi_enable_hba is provided for certain device driver architectures, e.g. Netware, that require that adapter resources be registered before they can be used. In the case of Netware, the memory mapped run-time registers on the adapter 114 must be registered before they can be used. In the prototype of FIG. 5, parameter adptr is a pointer to an HBA object. The routine sdi_enable_hba will return a value of 0 if the adapter 114 was enabled. If the adapter 114 was not enabled, a non-zero constant will be returned.

The routine sdi_reset_hba resets the specified adapter 114. The parameter adptr is a ptr to an HBA object. This routine provides no return value.

The routine sdi_hba_count returns the number of adapters detected in the system. The routine has no parameters.

The routine sdi_hba_irq returns the interrupt request line (IRQ) associated with an adapter 114. The parameter adptr is a pointer to an HBA object.

The routine sdi_hba_rt_virtual_addr returns the logical, or virtual, address of the adapter 114 memory mapped registers. This routine is provided for those operating systems that require registering the memory mapped registers, e.g. Netware. The required parameter is adptr, a pointer to an HBA object.

The routine sdi_hba_rt_physical_addr returns the physical address of the adapter 114 memory mapped registers. This routine is provided for those operating systems that require registering the memory mapped registers, e.g. Netware. The required parameter is adptr, a pointer to an HBA object.

The routine sdi_hba_rt_size returns the size in bytes of the adapter memory mapped registers. This routine is provided for those operating systems that require registering the memory mapped registers, e.g. Netware. The required parameter is adptr, a pointer to an HBA object.

The routine sdi_hba_pci_bus_number returns the PCI bus number associated with the adapter. The required parameter is adptr, a pointer to an HBA object.

The routine sdi_hba_pci_device_number returns the PCI device number associated with the adapter 114. The required parameter is adptr, a pointer to an HBA object.

The routine sdi_hba_pci_device_number returns the PCI device number associated with the adapter.

Packet Routines

Packet routines 300 are used to allocate, send, free, and complete sdi_xt_packet structures discussed above. FIG. 6 shows ANSI C source code prototypes for each of the packet routines 300 operable with the SDI 113. With reference to FIG. 3, packet routines 300 send packets 304 or async packets 306 to the adapter 114. Upon completion of the packet, respective responses 305, 307 are sent from the adapter 114 to the SDI 113. One skilled in the art will recognize the role of each of the routines set forth in Fig. 6 as it is further explained herein. Further, the use or sequence of use of the packet routines 300 will be discussed hereinafter by way of flow charts.

The routine sdi_send_packet sends a specified packet to a specified adapter 114. Parameter adptr is a pointer to an HBA object and parameter pkt is a pointer to an sdi_xt_packet structure to be sent to the device at the CRQ set in the crq_num field of the packet request. This routine provides no return value.

The routine sdi_stock_packet_pool stocks a pool of free packets. The packets are carved from memory beginning at address vaddr. The size of the memory allocation depends on the number of the packets that are

to be created. This routine may be called more than once to increase the size of the pool. With reference to the prototype in FIG. 6, the parameter adptr is a pointer to an HBA object, vaddr is a pointer to a virtual base address of allocated space, paddr is a pointer to physical base address of allocated space, and size is the size of allocated space in bytes. The routine sdi_stock_packet_pool returns the number of packets actually stocked in the pool.

The routine sdi_get_packet_from_pool gets a packet pointer from the pool of created packets. Parameter adptr is a pointer to an HBA object. The routine returns the value 0 if no packets are available, or the value 1 if a packet was retrieved successfully.

The routine sdi_get_physical_base_addr sets mapping registers in the HBA to reference host memory beginning at base_addr. This function is used if the physical address range of the system does not begin at 0x00000000. Parameter adptr is a pointer to an HBA object and parameter base_addr is the base address in host memory. There is no return value associated with this routine.

The routine sdi_stock_sglist_pool stocks a pool of free scatter/gather lists. The lists are carved from memory beginning at vaddr. The size of the memory allocation depends on the number of lists that are to be created. This routine may be called more that once to increase the size of the pool. The parameter adptr is a

pointer to an HBA object, nelem is the number of scatter/gather elements in each scatter/gather list, vaddr is a pointer to a virtual base address of allocated space, and size is the size of allocated space in bytes. The routine returns the number of scatter/gather lists actually stocked in the pool.

The routine sdi_get_sglist_from_pool gets a scatter/gather list from the list pool. The parameter adptr is a pointer to an HBA object and psgl is a pointer to a pointer to a scatter/gather list. The routine returns a value of 0 if no scatter/gather list is available, or a value 1 if a scatter/gather list is available and pointed to by psgl.

The routine sdi_return_sglist_to_pool returns a scatter/gather list to the list pool. Parameter adptr is a pointer to an HBA object and psgl is a pointer to a scatter/gather list to be returned to the pool. There is no return value associated with this routine.

The routine sdi_interrupt is the SDI 113 time entry point. The interrupt service routine (ISR) of adapter driver 110 must call this routine for each adapter 114 supported by the driver 110. For those interrupts that indicate the completion of a packet a callback function associated with the packet is called. The routine sdi_interrupt executes in interrupt context. Parameter adptr is a pointer to an HBA object. The routine returns a value of 0 if the SDI 113 successfully processed the

interrupt, or a value of 1 if the adapter 114 did not have a pending interrupt, so that the SDI 113 could not process one.

The routine packet_complete_callback is an operating system dependent routine. The SDI 113 will call this routine for each packet that completes. The routine executes in interrupt context. Paramater pkt is a pointer to the sdi_xt_packet that the SDI 113 has completed. The routine provides no return value. For example, a command callback routine checks for the scsi_status upon packet completion.

The routine sdi_return_packet_to_pool is used to return a packet to the adapter's 114 pool of packets set up by sdi_stock_packet_pool.

Mailbox Routines

Mailbox routines 200 are used to obtain information about the adapter 114 and, for example, the SSA-compatible nodes 115 attached to the adapter 114. FIG. 7 shows ANSI C source code prototypes for each of the mailbox routines 200 operable with the SDI 113. With reference to FIG. 3, mailbox routines 200 send mailbox commands 202 expecting immediate responses 203. That is, the mailbox routines 200 will wait for the message response 203 before returning. One skilled in the art will recognize the role of each the following mailbox routines 200 as each is further explained herein.

Further, the use or sequence of use of the mailbox routines 200 will be discussed hereinafter by way of flow charts.

The return values of each of the mailbox routines 200, except for one exception described below, are constants representative of the status of the routine, namely, success, timeout, or internal error.

The routine sdi_get_hba_info is used to gather information about each adapter including firmware and hardware revisions, and the unique ID (UID) of the adapter. Parameter adptr is a pointer to an HBA object and rev is a pointer to a driver-declared variable of type sdi_xt_hba_info. Upon return, the detected firmware and hardware revisions, and the UID of the adapter 114 is stored in a data structure as shown in FIG. 9.

The routine sdi_get_next_device is used to gather information about devices that support a particular upper level protocol, e.g. SSA. Parameter adptr is a pointer to an HBA object, ulp is a constant representative of the upper level protocol, e.g. SSA or TCP. Parameter crq_num is a pointer to a driver-declared variable for the CRQ number of the target. To obtain the first target, this parameter should be set to a particular constant. On return, the crq_num will contain the CRQ number of the first target. To obtain information of subsequent targets, the previously returned crq_num is used. If there are no more targets, another constant is returned.

Parameter uid_hi and uid_low are, respectively, pointers to driver-declared variables for the high and low 32 bits of the target unique ID (UID).

The routine sdi_open_crq is used to instruct the adapter 114 to logically open the specified command/response queue (CRQ) with the specified upper level protocol. Each CRQ, with the exception of the Async CRQ, must be opened before it can be used. The parameter adptr is a pointer to an HBA object, ulp is the upper level protocol and crq_num is the CRQ number of the device or node 115.

The routine sdi_close_crq is used to instruct the adapter 114 to logically close the specified CRQ. Parameter adptr is a pointer to an HBA object and crq_num is the CRQ of the device or node 115.

The routine sdi_get_scatter_gather_info is used to determine the adapter's 114 scatter/gather capability. Paramater adptr is a pointer to an HBA object, capability is the capability of the adapter 114 to handle scatter/gather lists, max_entries is the maximum number of entries in the list, and max_xfer is the maximum data transfer per entry. If either of the latter two parameters are unlimited, it should be set to a value of -1.

The routine sdi_get_ssa_web_info obtains SSA web map information from the adapter 114. The web information will be transferred by the adapter into an assigned

buffer. Parameters include adptr, a pointer to an HBA object, web_buff_p, a physical address of buffer, web_buff_v, a virtual address of buffer, and buff_size, the size of the buffer in bytes. Web information is transferred by the adapter 114 into the buffer whose physical address is web_buff_p. The amount of information is limited to buffer_size.

Operating System Dependent Routines

Operating system dependent routines 400 enable the SDI 113 to be portable across multiple operating systems. FIG. 8 shows ANSI C source code prototypes for each of the operating system routines operable with the SDI 113. One skilled in the art will recognize the role of each of the following routines as it is further explained herein. Further, the use of the operating system dependent routines 400 will be discussed hereinafter by way of flow charts.

The routine sdi_pci_find_device is used to find the next PCI device. The parameter driver_context is a pointer to a driver-defined object associated with a particular HBA. The parameter is set by initialization routine 150 sdi_initialize. Parameter bus is the returned bus number, device is the returned PCI device number, function is the returned PCI function number, vendor_id is the PCI vendor ID, device_id is the PCI device ID, index is the number of matched adapters to

skip before beginning the search, and driver-context is a pointer to the driver defined object associated with an HBA. The routine sdi_pci_find_device returns a constant indicative of success or error including, no device found, bad vendor ID, or unknown.

The routine sdi_pci_read_config_register is used to read the adapter 114 configuration registers to determine its configuration information. The routine is also used to determine the adapter's 114 IRQ and the address of the adapter's 114 memory mapped registers. contents is the returned contents of the register, reg_num is the register number to read, bus is the PCI bus number, device is the PCI device number, function is the PCI function number, and driver context is a pointer to the driver defined object associated with an HBA. As noted above, driver-context is set by initialization routine 150 sdi_initialize. The routine sdi_pci_read_config_register returns a constant indicative of success or error including, bad register or unknown.

The routine sdi_map_physical_to_virtual is used to map a physical address memory area to a virtual address. The SDI 113 uses this routine to map the address of the memory mapped registers from physical to virtual. Parameter paddr is the physical address to map, size is the size of the memory area, and driver_context is a pointer to the driver-defined object associated with the

HBA. There is no return value associated with this routine.

The routine sdi_delay_task delays a current operating system task. It is used during SDI 113 mailbox routines 200. Parameter ticks is the number of ticks to delay, which is dependent upon the granularity of the timer. Parameter driver_context is the same as described above. The routine has no return value.

The routine sdi_ticks_per_second return the granularity of the operating system's timer and is used by the SDI 113 during mailbox routines 200. The parameter driver_context is the same as is explained above. The routine returns the number of ticks per second which can then be used by the routine sdi_delay_task.

Adapter Functionality

In order for the SDI to operate, many of the software routines described above must be able to communicate with the adapter 114. In the preferred embodiment of the present invention communication is made operable via defined registers for adapter 114 as shown in Fig. 9. A PCI 9060 interface controller is an example of an interface controller suitable for this purpose. It should be noted, however, that any adapter having similar functional capability of the 9060 controller can also be used. Moreover, the registers described could be

implemented via software on the host computer system.

All communication between the adapter driver 110 comprising the SDI 113 is initiated by the adapter driver 110. There are two types of messages: mail messages and packets. Mail messages, shown in Fig. 3 as 202, 203, are immediate messages and are placed in registers 5-8 in Fig. 9. Packets, shown in Fig. 3 as 304-307, are used to transfer SSA messages (commands) via registers 1, 2, to the adapter 114 and to receive responses in registers 3, 4 from the adapter 114. These message transfers can be accomplished across any standard bus architectures including GIO, PCI, Sbus or VME. Specifically, commands are sent to the adapter 114 by copying the physical address of the packet to register 1 or 2. Responses are received from the adapter 114 in registers 3 or 4. Register 9 of Fig. 9 is used to indicate to the adapter 114 that a command or mail message has been written to one of the registers, and register 10 is used to indicate to the SDI that a response has been written to one of the registers. In the present embodiment, the registers thus described are offset 0x40 from the address of the memory mapped registers.

Operation of the SDI

The operation of the adapter driver 110 comprising the SDI 113 in combination with the adapter 114 will now be explained.

Fig. 10 shows the highest level of operation of the SDI 113 and adapter 114. At step 2070, the adapter driver 110 is initialized. At step 2071, request and response queues are initialized on adapter 114 to support the sending and receiving of packets. Final initialization routines for adapter 114 are implemented at step 2072 including initializing a serial controller clock, on-board clock, and PCI interface control. Then an infinite loop is entered comprising steps 2073 and 2074 in which the response_wait_queue is checked and other functions including for example, serial controller, timer and clock, and LED control are implemented. Step 2074 is also the point at which internal process queues of the adapter are handled.

Initialization

The SDI 113 and adapter 114 are first initialized using the SDI routines described earlier. With reference to FIG. 11, sdi_get_hba_object_size is first called in step 1010 after which sdi_initialize is called in step 1011. From within the routine sdi_initialize, operating system dependent (OSD) routines sdi_pci_find_device and sdi_pci_read_config_register are called as shown in Fig. 12. That is, each host bus adapter (HBA), that is an adapter supporting the functions of adapter 114, will be found via steps 1020, 1021 and 1023. If no more devices are found, control is passed back to sdi_initialize at

step 1022. Again with reference to Fig. 11, sdi_initialize is called via step 1012 until all HBAs in the system have been found. At this point, sdi_enable_hba is called in step 1015 for each HBA. One of the functions of sdi_enable _hba is to call, OSD function sdi_map_physical_to_virtual as shown in step 1025 of Fig. 13. After this OSD function is called the control is returned in step 1026 to sdi_enable_hba. If there are no more HBAs to enable, the process is terminated in step 1014 of Fig. 11.

After each HBA has been enabled, any one, or combination, of functions A-J can next be called. Each of these routines is illustrated, respectively, in FIGs. 14-23. Which routines are called depend on the operating system in which the adapter driver 110 is located. For example, sdi_hba_rt_virtual_address need only be called for the Netware operating system, but not for the Windows NT operating system.

Fig. 14 depicts the calling of sdi_hba_irq in step 3000. The procedure returns at step 3001.

Fig. 15 depicts the calling of sdi_hba_rt_virtual_addr in step 3010 and returning in step 3011.

Fig. 16 shows sdi_hba_rt_physical_addr being called in step 3020 and then returning in step 3021.

Fig. 17 illustrates in step 3030 the calling of routine sdi_hba_rt_size and return in step 3031.

Fig. 18 shows sdi_hba_pci_bus_number being called in step 3040 and then returning in step 3041.

Fig. 19 shows in step 3050 a call to sdi_hba_pci_device and then return in step 3051.

Fig. 20 depicts a call to sdi_get_physical_base_addr in step 3060 and a return in step 3061.

Fig. 21 shows sdi_get_hba_info being called in step 3070 and a return in step 3071.

Fig. 22 shows a call to sdi_get_sg_info in step 3080. If at step 3081 the return value of sdi_get_sg_info indicates that the adapter 114 has scatter/gather capability then the device driver is informed that the adapter driver 110 can support scatter/gather. If the adapter 114 cannot support scatter/gather, then the device driver is informed likewise in step 3083. The procedure returns at step 3084.

Fig. 23 shows at step 3090 a call to sdi_get_web_info. The information returned is processed in step 3091. That is, the returned values can be stored in a data structure representative of the serial network or web. The procedure returns in step 3092.

Identifying Devices

Once the adapter driver 110 has found and initialized each HBA or adapter 114, each SSA device connected to each HBA must be identified and a data

structure representative of the device must be initialized. This process is explained with reference to Figs. 24 and 25. In step 1090, sdi_get_next_device is called to identify the first device on a web or network. This routine returns information about those SSA devices that support a predetermined upper level protocol. If a device is found in step 1091, the routine will return a unique command/request queue (CRQ) number for the upper level protocol-compatible device found. If no more devices are found in step 1091 then the procedure is exited.

For each device found a device structure is initialized in step 1093 and at step 1094 sdi_open_crq is called to logically open the device.

The result of finding all HBAs and devices connected to each HBA is a data structure such as that shown in Fig. 25. Each element 50 represents an HBA, or adapter 114, pointed to by pointer 51. Each HBA 51 is further associated with device structures 52 representing each identified device connected to that particular adapter or HBA.

Sending a Packet

A command from a device driver through the SDI will next be explained with reference to Fig. 26. Before any packets can be sent at all, a pool of packets must be pre-positioned by calling sdi_stock_packet_pool in step

1030. Upon receipt of an I/O request from the device driver in step 1031, sdi_get_packet_from_pool is called in step 1032. If no packets are available at step 1033, the process returns at step 1034. If a packet is available, the device driver I/O request, which may be in accordance with a SCSI protocol for example, is translated into its SSA equivalent in step 1035. Mapping of the SCSI-2 protocol to SSA is implemented in accordance with the SSA documentation noted in Description of the Prior Art section of this specification. Mapping a SCSI-2 protocol to fibre channel is likewire set out in the fibre channel documentation.

Once translation has been completed, the ssa_request structure is completed in step 1036 and then sdi_send_packet is called in step 1037. The sending of a packet is now complete and the process returns in step 1038.

If the adapter 114 has scatter/gather capability, as determined by an earlier call to sdi_get_scatter_gather_info, then sending a packet is slightly different than as just explained. With reference to Fig. 27, sdi_stock_sglist_pool is first called in step 1040 after which sdi_stock_packet_pool is called in step 1041. As in the earlier case, upon receiving an I/O request from the device driver in step 1042, sdi_get_packet_from_pool is called in step 1043 and

if none is available in step 1044, the process returns. If, however, a packet is available the process continues to step 1046 in which sdi_get_sglist_from_pool is called. If no scatter/gather list is available in step 1047, the packet previously retrieved is returned to the packet pool in step 1048 and the process returns.

Assuming a scatter/gather list is available in step 1047, the I/O request is translated to its SSA equivalent in step 1050, the ssa_request structure is completed in step 1051, the scatter/gather list is linked to the packet in step 1052 and then sdi_send_packet is called in 1053. Finally, the process returns at step 1054 after step 1053.

It was noted earlier that the adapter 114 includes an asynchronous event command/response queue (Async CRQ 302 in FIG. 3), and that there are special async data structures as shown in FIG. 4b. The handling of Async events represents a dramatic difference between handling of parallel SCSI and SSA. SSA is a network of device attachments, and the attachments may be dynamic with devices being removed and added while the network is in operation. The Async CRQ responds to these events (e.g., device removed, device attached, local port disconnected, local port connected) and passes the information to the adapter driver 110 which maintains a list of available nodes 115.

The Async Packet is a command whose function is to

wait for the next Asynchronous event and report it. The command may remain pending indefinitely. In the preferred embodiment, some number of Async Packets are posted to the Async CRQ to allow a series of event notifications to pass from the adapter 114 to the adapter driver 110. The Command/Response mechanism is used in this way to allow the Async Packet support to make use of the same interface as the other non-Async CRQ methods.

In other words, the adapter driver 110 may send one or more Async packets to the adapter 114 using sdi_sent_packet. The CRQ number for the Async packets should be set to a constant reserved for the Async CRQ. When an asynchronous event occurs, the adapter 114 returns the Async packet to the adapter driver 110. The packet's callback routine will then initiate the appropriate processing for the type of Async event.

Handling Completed Packets

Fig. 28 illustrates the process for completing packets. When a response is written to one of the response registers 3 or 4 of Fig. 9 and the local to PCI doorbell register 10 is set, as will be further explained below, the SDI is interrupted to process the response packet. In step 1060, the packet status is determined from the scsi_status field of the ssa_response. If the status is not OK, i.e. non-zero, in step 1061, an error code is set in step 1062. If the status is zero both the

packet and scatter/gather list are returned in steps 1063 and 1064 and then the status code is returned to the device driver in step 1065.

Other Functions

Fig. 29 depicts a process for closing CRQs and resetting HBAs if necessary or if desired. For instance, some operating systems, e.g. Netware, can dynamically unload drivers. The present invention supports such a capability through the procedure shown in Fig. 29. Beginning at step 1070, a counter is set to zero and then at step 1071 it is determined whether the CRQ number equivalent to that counter number is open. If the CRQ is open, the CRQ will then be closed at step 1072 by calling sdi_close_crq. The procedure proceeds to step 1073. If the CRQ is not open at step 1071, then the procedure advances directly to step 1073. At step 1073 it is determined whether the counter exceeds a predetermined maximum number of devices. If not, the counter is incremented at step 1074 and the process loops back to step 1071. If the counter is greater than or equal to the maximum number of devices at step 1073, then sdi_reset_hba is called at step 1075, thus resetting the HBA. If there are more HBAs at step 1076, the process loops back to step 1070, otherwise, the process returns at step 1077.

As was noted earlier, mail messages are immediate messages and the SDI expects responses from the adapter 114 before executing any other routines. Accordingly, the SDI 113 also includes OSD routines that will delay other processing until mail messages have completed. Fig. 30 illustrates the use of these routines which are called from within each mailbox routine 200. At step 1080, sdi_ticks_per_second is called. A multiple of the return value of that routine is then passed to sdi_delay_task at step 1081 in order to effectuate the delay. Step 1082 then returns control back to the mailbox routine 200 that was previously called.

Fig. 31 illustrates a typical interrupt handler operable with the SDI 113. Using the return value of sdi_hba_count from step 1100, sdi_interrupt is called in step 1103 after step 1101 for each adapter 114 in the system. If there are no more HBAs, the process returns in step 1102. As noted in the earlier description of the sdi_interrupt routine, the routine packet_complete_callback is called from within sdi_interrupt.

Flow of Commands/Responses Between the SDI and Adapter

The adapter driver 110 passes a packet to the adapter 114 by calling sdi_send_packet. As explained earlier, the packet field complete_callback points to an OSD function to be called when the packet completes,

while other packet fields describe the particular command.

With reference to Fig. 32, at step 2000, sdi_send_packet places the packet on a request_Q, a linked list of all packets pending transmission to the adapter 114. The routine sdi_send_packet also calls sdi_start_request_queue at 2005 which is shown in more detail in Fig. 33. After step 2005, the procedure returns. If routine sdi_start_request_queue is unable to service the queue due to no available command slots, Fig. 9 elements 1, 2, the routine will be called again from the sdi_interrupt function when a constant indicative of sdi_interrupt_slot_free is set in the status register 10 in Fig. 9.

The routine sdi_start_request_queue attempts to service the first packet of the request queue by writing the packet pointer to a free command slot register 1, 2. A command slot register 1, 2 is free when its contents is zero. According to the flow chart of Fig. 33, if there is no entry on the request queue the routine start_request_queue returns per step 2010. On the other hand, if there is an entry on the request queue at step 2010, a counter is reset at step 2011. If the counter is zero at step 2013 then the routine goes to step 2014 where it is determined whether command slot 0, register 1 in Fig. 9, is not in use. If command slot 0 is not in use the packet is removed from the request queue in step

2015 and the packet pointer (pkt) is written to command slot 0 in step 2016. Then, at step 2017, a constant sdi_interrupt_packet_command is set in the pci_to_local_doorbell register 9 of Fig. 9 and then the routine restarts.

If at step 2014 command slot 0 is in use, the counter is incremented at step 2018 and the routine flows back to step 2013. Thus if the counter is not 0 at step 2013, the counter is tested as to whether it is equal to 1. If no, the routine returns. If the counter is equal to 1, then the routine tests if command slot 1, register 2 in Fig. 9, is not in use in step 2021. If command slot 1 is not in use, then the packet is removed from the request queue in step 2022 and the packet pointer (pkt) is written to command slot 1 in step 2023. Thereafter, the constant sdi_interrupt_packet_command is set in the pci_to_local_doorbell register 9 of Fig. 9 and then the routine restarts as explained earlier. If at step 2021 command slot 1 was in use, the counter is incremented at step 2018 as shown.

If a command slot register 1, 2 has been written by the sdi_start_request_queue routine, the interrupt to the adapter 114 is set by writing an sdi_interrupt_packet_command bit to the pci_to_local_doorbell register 9 as explained above. In response to the interrupt, the adapter 114 enters a doorbell_interrupt routine as shown in Fig. 34. This

routine, at step 2030, reads the Interrupt and Control Status register 11 of Fig. 9 and determines if the sdi_interrupt_packet_command bit has been set. If it has not been set the routine, at step 2034, writes the constant sdi_interrupt_packet_command bit clear in the pci_to_local_doorbell register 9, thus clearing the interrupt, and then returns. If, on the other hand, the sdi_interrupt_packet_command bit is set at step 2031, then the doorbell_interrupt routine calls the check_cmd_list routine, shown in detail in Fig. 35.

Generally the check_cmd_list function gets the pointer to the packet from the valid command list entry and passes it to adapter 114 firmware for processing according to well known processes. The command list entry is cleared thus making it available for new packets. Then, the adapter 114 signals availability of a command slot register 1, 2 to the adapter driver 110 by writing the sdi_interrupt_slot_free bit in the local_to_pci_doorbell register 10 of Fig 9.

In particular, with reference to Fig. 35, a counter cmds_taken is reset to zero in step 2040. Thereafter, at step 2041, if command slot 0 is not zero the packet pointer is read from command slot 0 in step 2042, command slot 0 is reset in step 2043, the counter cmds_taken is incremented in step 2044, and finally, the read packet pointer is placed on an internal process queue of the adapter 114. If command slot 0 has a zero value at step

2041, then command slot 1 is tested for a zero value in step 2050. If command slot 1 is nonzero, the packet pointer is read from command slot 1 in step 2051, command slot 1 is reset in step 2052, the counter cmds_taken is incremented in step 2044, and finally, as with the command read from command slot 0, the read packet pointer from command slot 1 is placed on an internal process queue of the adapter 114.

If both command slots registers 1, 2 of Fig. 9 have zero values, step 2055 determines whether the counter cmds_taken is not zero. If cmds_taken is in fact not zero, then a constant sdi_interrupt_slot_free is set in the local_to_pci_doorbell in step 2056 and the function returns. If cmds_taken is zero the function immediately returns.

Upon command completion, that is the adapter 114 properly processed the command, the adapter 114 calls the resp_send routine shown in Fig. 36. The resp_send routine in step 2060 places the pointer to the packet in the resp_wait_queue. The resp_wait_queue is a linked list of packets pending for transmission to the adapter driver 110. In step 2061, resp_send calls the function check_resp_wait_queue and then returns. The routine check_resp_wait_queue attempts to service the resp_wait_queue's first entry by writing the packet pointer to a free response slot 3, 4 of Fig. 9. A slot is free if its contents is zero. If, however,

check_resp_wait_queue is unable to immediately service the queue due to nonzero response slot registers 3, 4, the routine will eventually be called again from the adapter driver 110 main loop, shown in Fig. 10.

Fig. 37 shows the check_resp_wait_queue function in detail. In step 2080, the flag resp_sent is reset to zero. Thereafter, at step 2081 it is determined whether there is an entry on the resp_wait_queue. If not, it is determined at step 2082 whether counter resp_sent is not 0. If the counter is zero the function returns. Otherwise, in step 2083, an sdi_interrupt_packet_response bit is set in the local_to_pci_doorbell register 10.

If there is an entry on the resp_wait_queue at step 2081, step 2085 determines whether response slot 0, i.e. register 3, is zero. If so, the packet is taken from the resp_wait_queue in step 2086 and the token is written to response slot 0 in step 2087. In step 2088 flag resp_sent is set to 1 and the function goes back to step 2081. If response slot 0 is not zero in step 2085, then response slot 1, i.e. register 4, is tested for its value. If zero, then the packet is taken from the resp_wait_queue in step 2091 and the token is written to response slot 1 in step 2092. If neither response slot has a zero value, then the function is routed back to step 2082 as shown in Fig. 38.

As explained earlier the token is used as a means for the adapter driver 110 to map a response back to the

original command. The token is set to the logical (or virtual) address of the packet. Upon reading the token from the response slot register 3, 4, the adapter driver 110 looks up pkt based on the token value and the completed packet status can then be processed as explained earlier with reference to FIG. 28.

As described above, if a response slot register 3, 4 has been written, the adapter driver 110 is signaled by setting the sdi_interrupt_packet_response bit in the local_to_pci_doorbell register 10. With the local_to_pci_doorbell set, the adapter driver's 110 sdi_interrupt function is entered. This function is illustrated in Fig. 38.

In step 2100 of Fig. 38, the cause is read from the local_to_pci_doorbell register 10 and then is written back to local_to_pci_doorbell register 10 to clear the interrupt cause. If the interrupt cause ANDED with a constant representative of sdi_interrupt_slot_free is yes in step 2102 then sdi_start_request_queue is called in step 2110. Otherwise, if the interrupt cause ANDED with a constant representative of sdi_interrupt_packet_response is yes in step 2103, sdi_process_response_list is called in step 2111. The cause is again evaluated in step 2104. If the cause is zero the function returns. Otherwise, the function loops back to step 2100.

The routine sdi_process_response_list, depicted in

Fig. 39, is used to check for entries in the response slot registers 3, 4. According to the figure, on start-up a counter is set to zero in step 2120. The counter's value is thereafter checked in step 2121. If the counter is zero, then response slot 0 is checked for a nonzero value in step 2123. If response slot 0 is not nonzero then the counter is incremented in step 2135 and the routine proceeds back to step 2121. If response slot 0 is nonzero then the packet is read therefrom in step 2123, the response slot 0 is reset to zero to make it available, the complete_callback function for the packet is called in step 2125, and then the counter is incremented.

If at step 2121, the counter is not equal to zero, and the counter equals 1 at step 2130, response slot 1 is checked for a nonzero value in step 2131. If response slot 1 has a nonzero value, the packet is read from response slot 1, response slot 1 is then reset to zero in step 2133 to make it available, the complete_callback function is called in step 2134 and the counter is incremented. If response slot 1 has a 0 value at step 2131 then the counter is simply incremented. However, if at step 2130, the counter is not equal to 1, then the function returns. In other words, there are no responses to be read.

The SDI 113 routines and data structures described herein enable economical development of adapter drivers

to permit protocol incompatible device drivers and targets to pass I/O requests and responses to and from one another. Moreover, the routines described herein are designed to operate independent of operating system environment.

While the present invention has been described with reference to particular routines and data structures, those skilled in the art will recognize that some of the routines may be combined and others may be broken down further and the data structures may be arranged differently. Furthermore, as noted previously, the SDI invention is operable in a plurality of operating systems including MIPS, Intel I960, Pentium, Power PC and SPARC based systems. Accordingly, not all disclosed routines are necessary for any particular implementation. Further still, those skilled in the art will appreciate that additional error trapping and handling routines, for example, may be included in any completed program code.

What is claimed is:

1. A computer operating system and hardware-independent process for implementing I/O between a device driver and a target, comprising the steps of:

receiving a first I/O request according to a first protocol from said device driver;

translating said first I/O request into a second I/O request according to a second protocol;

assembling a data structure pointed to by a first pointer including said second I/O request, a unique token value, and a second pointer to an operating system-dependent completion function;

sending said first pointer to an adapter;

signalling said adapter to send said second I/O request to said target;

receiving a response from said target;

associating said response with said data structure via said unique token value; and

calling said operating system-dependent completion function by way of said second pointer.

2. The process of claim 1 wherein said first protocol is parallel SCSI and said second protocol is SSA.

3. The process of claim 1 wherein said first

protocol is parallel SCSI and said second protocol is fibre channel.

4. The process of claim 1 wherein said first protocol is parallel SCSI-2 and said second protocol is SSA.

5. The process of claim 1 wherein said first protocol is parallel SCSI-2 and said second protocol is fibre channel.

6. The process of claim 1 wherein said first protocol is TCP and said second protocol is SSA.

7. The process of claim 1 wherein said first protocol is TCP and said second protocol is fibre channel.

8. The process of claim 1 wherein said first protocol is SSA and said second protocol is SSA.

9. The process of claim 1 wherein said first protocol is SSA and said second protocol is fibre channel.

10. The process of claim 1 wherein said first protocol is fibre channel and said second protocol is

SSA.

11. The process of claim 1 further comprising:

determining the number of adapter cards, supportive of a predetermined I/O protocol, in a computer system;

determining the number and type of target devices connected to each one of said previously identified adapter cards; and

building a data structure representative of each adapter card and associated targets such that each target can be logically enabled or disabled.

12. The process of claim 1 further comprising initializing a plurality of said data structures prior to said receiving a first I/O request.

13. The process of claim 1 wherein said response from said target represents an asynchronous event.

14. The process of claim 11 wherein processing in said computer system is delayed until completion of determining the number of adapter cards.

15. A computer operating system and hardware-independent process for implementing I/O between a standard device driver and an SSA compatible target, comprising the steps of:

receiving a first I/O request according to a standard protocol from said standard device driver;

translating said first I/O request into a second I/O request according to an SSA compatible protocol;

assembling a data structure pointed to by a first pointer including said second I/O request, a unique token value, and a second pointer to an operating system-dependent completion function;

sending said first pointer to an adapter;

signalling said adapter to send said second I/O request to said target;

receiving a response from said target;

associating said response with said data structure via said unique token value; and

calling said operating system-dependent completion function by way of said second pointer.

16. The process of claim 15 further comprising:

determining the number of adapter cards, supportive of said SSA compatible protocol, in a computer system;

determining the number and type of target devices connected to each one of said previously identified adapter cards; and

building a data structure representing each adapter card and associated targets such that each target can be logically enabled or disabled.

17. A computer operating system and hardware-independent process for implementing I/O between a standard device driver and a fibre channel compatible target, comprising the steps of:

receiving a first I/O request according to a standard protocol from said standard device driver;

translating said first I/O request into a second I/O request according to a fibre channel compatible protocol;

assembling a data structure pointed to by a first pointer including said second I/O request, a unique token value, and a second pointer to an operating system-dependent completion function;

sending said first pointer to an adapter;

signalling said adapter to send said second I/O request to said target;

receiving a response from said target;

associating said response with said data structure via said unique token value; and

calling said operating system-dependent completion function by way of said second pointer.

18. The process of claim 17 further comprising:

determining the number of adapter cards, supportive of said fibre channel compatible protocol, in a computer system;

determining the number and type of target devices connected to each one of said previously identified

adapter cards; and

building a data structure representing each adapter card and associated targets such that each target can be logically enabled or disabled.

19. An apparatus for implementing computer operating system and hardware-independent I/O between a protocol-incompatible device driver and target, comprising:

means for receiving a first I/O request according to a first protocol from a device driver;

means for translating said first I/O request into a second I/O request according to a second protocol;

means for assembling a data structure pointed to by a first pointer including said second I/O request, a unique token value, and a second pointer to an operating system-dependent completion function;

means for sending said first pointer to an adapter;

means for signalling said adapter to send said second I/O request to said target;

means for receiving a response from said target;

means for associating said response with said data structure via said unique token value; and

means for calling said operating system-dependent completion function.

20. The apparatus of claim 19 wherein said means

for sending said pointer and said means for receiving said response comprises a memory register on said adapter.

21. A computer system having a serial interface, comprising:

an adapter driver including a serial driver interface;

an adapter card having at least one port, including memory for a control register and device command/request queues; and

at least one device, connected to said at least one port of said adapter card.

22. The computer system according to claim 21 wherein said serial driver interface provides an interface between a device driver operable according to a first protocol and a device operable according to an SSA protocol.

23. The computer system according to claim 21 wherein said serial driver interface provides an interface between a device driver operable according to a first protocol and a device operable according to a fibre channel protocol.

24. The computer system according to claim 22

wherein said first protocol is parallel SCSI.

25. The computer system according to claim 23 wherein said first protocol is parallel SCSI.

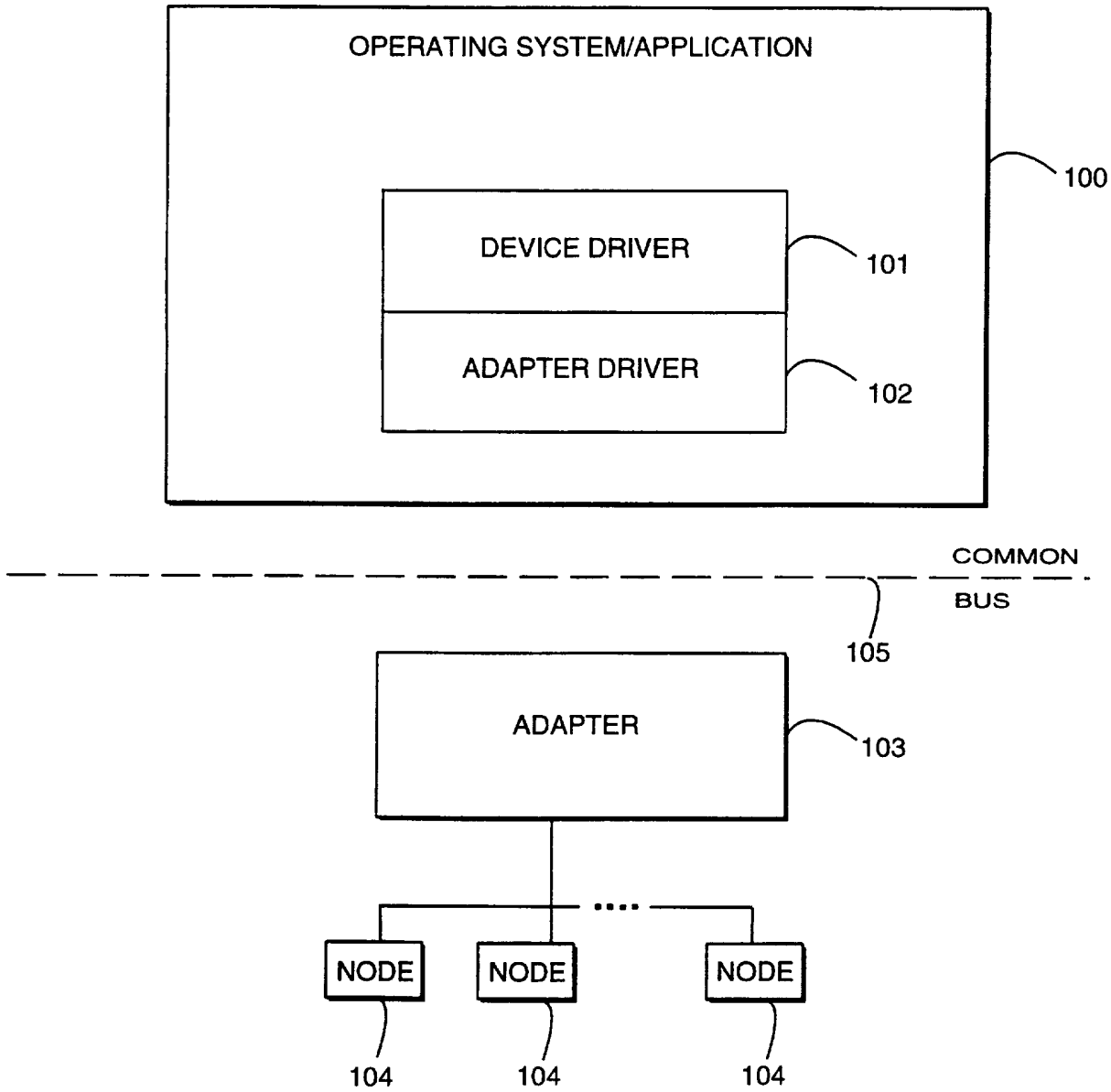


FIG. 1
PRIOR ART

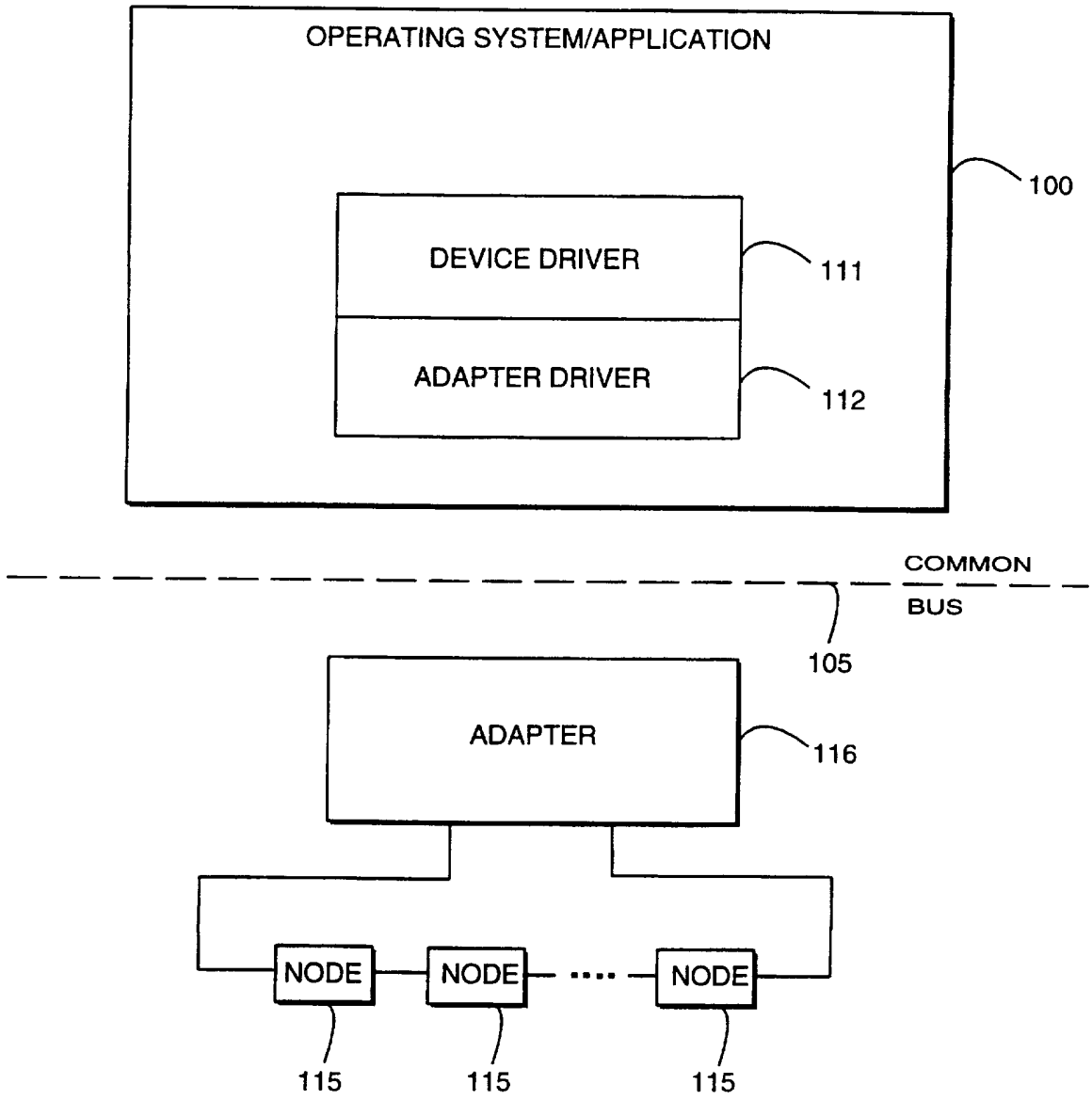


FIG. 2
PRIOR ART

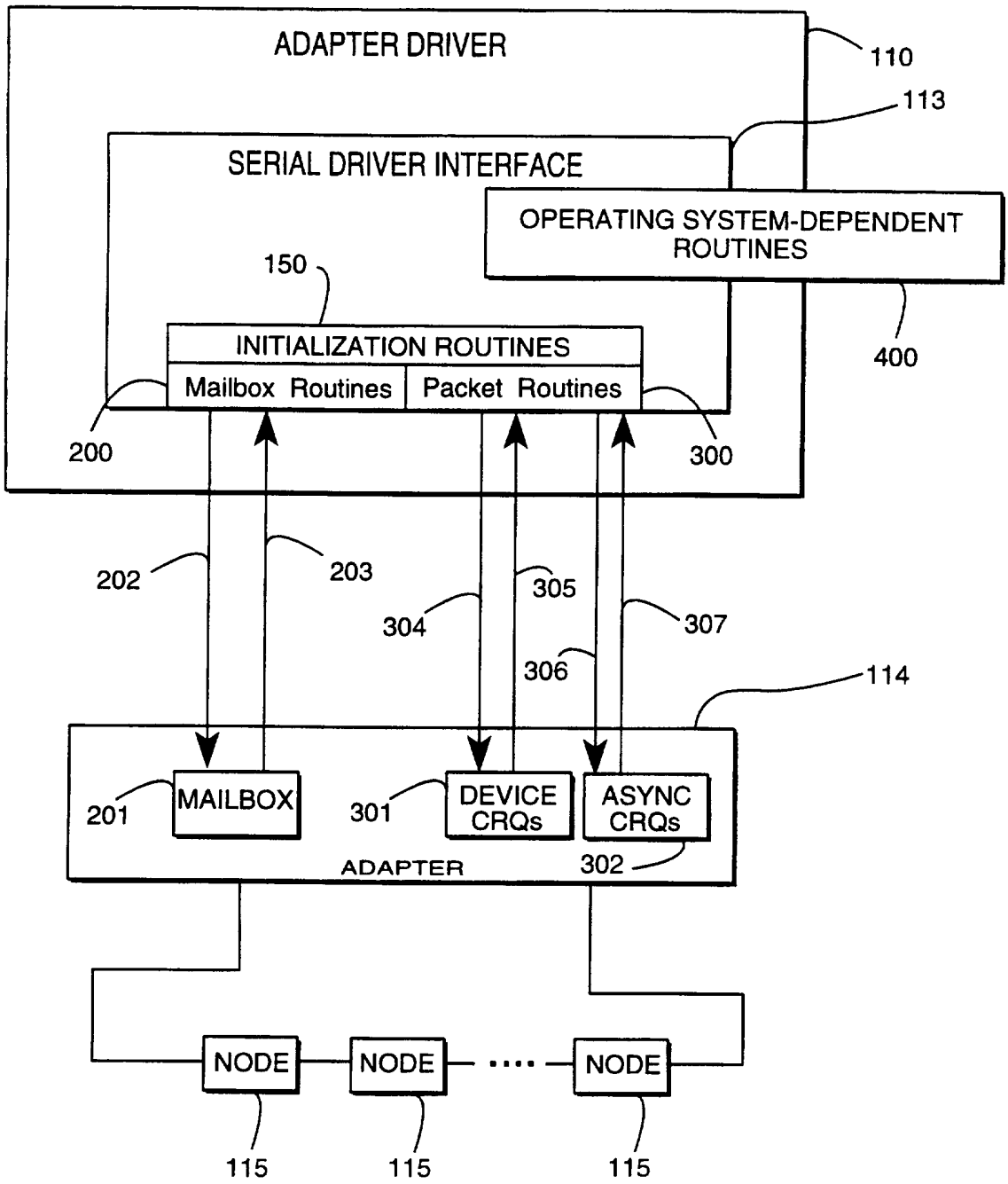

ADAPTER DRIVER
110
113
SERIAL DRIVER INTERFACE
OPERATING SYSTEM-DEPENDENT ROUTINES
150
INITIALIZATION ROUTINES
Mailbox Routines
Packet Routines
400
200
300
202
203
304
305
306
307
114
201
MAILBOX
301
DEVICE CRQs
ASYNC CRQs
ADAPTER
302
NODE
NODE
NODE
115
115
115

FIG. 3

STRUCTURE DEFINITION FOR sdi_xt_packet

```
typedef struct sdi_x_packet
{
     union
     {
        sdi_xt_ssa_packet ssa;
        sdi_xt_async_packet async;
     } type;
 #ifdef __DRIVER__

     void (*complete_callback)(struct sdi_x_packet *pkt);
     uint32 link_space[4];
     struct sdi_x_packet *phys_addr;
     uint32 driver_space[4];
#endif
} sdi_xt_packet;
```

STRUCTURE DEFINITION FOR sdi_xt_ssa_packet

```
typedef struct sdi_x_ssa_packet
{
     sdi_xt_ssa_request request;
     sdi_xt_ssa_response response;
} sdi_xt_ssa_packet;
```

STRUCTURE DEFINITION FOR sdi_xt_ssa_request

```
typedef struct sdi_x_ssa_request
{
     uint32 crq_num;
     uint32 token;
     uint32 timeout;
     uint32 dma_address;
     uint32 dma_size;
     uint32 dma_type;
     sdi_xt_scsi2_sms sms;
} sdi_xt_ssa_request;
```

STRUCTURE DEFINITION FOR sdi_xt_ssa_response

```
typedef struct sdi_x_ssa_response
{
     uint32 token;
     uint32 reason;
     uint32 state;
     uint32 scsi_status;
     uint32 residual;
} sdi_xt_ssa_response;
```

FIG. 4a

STRUCTURE DEFINITION FOR sdi_xt_async_packet

```
typedef struct sdi_x_async_packet
{
     sdi_xt_async_request request;
     sdi_xt_async_response response;
} sdi_xt_async_packet;
```

STRUCTURE DEFINITION FOR sdi_xt_async_request

```
typedef struct sdi_x_async_request
{
     uint32 crq_num;
     uint32 token;
}sdi_xt_async_request;
```

STRUCTURE DEFINITION FOR sdi_xt_async_response

```
typedef struct sdi_x_async_response
{
     uint32 token;
     uint32 reason;
     union
     {
        struct
        {
          uint32 crq_num;
          uint32 uid_hi;
          uint32 uid_lo;
        } new_target;

        struct
        {
          uint32 crq_num;
          uint32 uid_hi;
          uint32 uid_lo;
        } del_target;
        uint32 log_num;
        uint32 err_num;
     }type;
} sdi_xt_async_response;
```

STRUCTURE DEFINITION FOR sdi_xt_sg_element

```
typedef struct sdi_xt_sg_element
{
     uint32 sg_buf_size;
     uint32 sg_buf_phys_addr;
} sdi_xt_sg_element;
```

FIG. 4b

STRUCTURE DEFINITION FOR sdi_xt_hba_info

```
typedef struct sdi_x_hba_info
{
     byte fw_maj_rev;
     byte fw_min_rev;
     byte hw_maj_rev;
     byte hw_min_rev;
     uint32 uid_hi;
     uint32 uid_lo;
} sdi_xt_hba_info;
```

STRUCTURE DEFINITION FOR sdi_xt_scsi2_sms

```
typedef union sdi_x_scsi2_sms
{
     sdi_xt_scsi2_resposne     repsonse;
     sdi_xt_scsi2_command           command;
     sdi_xt_scsi2_status       status;
     sdi_xt_scsi2_abort_tag    abort_tag;
     sdi_xt_scsi2_abort        abort;
     sdi_xt_scsi2_clear_queue clear_queue;
     sdi_xt_scsi2_device_reset      device_reset;
     sdi_xt_scsi2-clear_aca    clear_aca;
} sdi_xt_scsi2_sms;
```

FIG. 4c

INITIALIZATION ROUTINE PROTOTYPES

```
sdi_get_hba_object_size
     uint32 sdi_get_hba_object_size();


sdi_initialize
     uint32 sdi_initialize_hba(uint8 hba_number,
                               void *adptr,
                               void *caller_context);

sdi_enable_hba
     uint32 sdi_enable_hba(void *adptr);

sdi_reset_hba
     void sdi_reset_hba(void *adptr);

sdi_hba_count
     uint32 sdi_hba_count();

sdi_hba_irq
     uint32 sdi_hba_irq(void *adptr);

sdi_hba_rt_virtual_addr
     uint32 sdi_hba_rt_virtual_addr(void *adptr);

sdi_hba_rt_physical_addr
     uint32 sdi_hba_rt_physical_addr(void *adptr);

sdi_hba_rt_size
     uint32 sdi_hba_rt_size(void *adptr);

sdi_hba_pci_bus_number
     uint8 sdi_hba_pci_bus_number(void *adptr);

sdi_hba_pci_device_number
     uint8 sdi_hba_pci_device_number(void *adptr);
```

FIG. 5

PACKET ROUTINE PROTOTYPES

```
sdi_send_packet
     void sdi_send_packet(void *adptr,
                  sdi_xt_packet *pkt);

sdi_stock_packet_pool
     uint32 sdi_stock_packet_pool(void *adptr,
                       void *vaddr,
                       void *paddr,
                       uint32 size);

sdi_get_packet_from_pool
     sdi_xt_packet *sdi_get_packet_from_pool(void *adptr,
                              sdi_xt_packet**pkt);


sdi_set_physical_base_addr
     void sdi_set_physical_base_addr (void *adptr
                             uint32 base_addr);

sdi_stock_sglist_pool
     uint32 sdi_stock_sglist_pool(void *adptr,
                       uint32 nelem,
                       void *vaddr,
                       uint32 size);

sdi_get_sglist_from_pool
     uint32 sdi_get_sglist_from_pool (void *adptr,
                             sdi_xt_sg_element *psgl);

sdi_return_sglist_to_pool
     void *sdi_return_sglist_to_pool (void *adptr,
                             sdi_xt_sg_element *psgl);

sdi_interrupt
     uint32 sdi_interrupt(void *adptr);

packet_complete_callback
     void packet_complete_callback(sdi_xt_packet *pkt);

sdi_return_packet_to_pool
     void *sdi_return_packet_to_pool_ (void *adptr,
                                  sdi_xt_packet *pkt);
```

FIG. 6

8/37

MAILBOX ROUTINE PROTOTYPES

```
sdi_get_hba_info
     uint32 sdi_get_hba_info(void *adptr,
                             sdi_xt_hba_info *pinfo);

sdi_get_next_device
     uint32 sdi_get_next_device(void *adptr,
                                uint16 ulp,
                                uint32 *crq_num,
                                uint32 *uid_hi,
                                uint32 *uid_lo);
sdi_open_crq
     uint32 sdi_open_crq (void *adptr,
                          uint16 ulp,
                          int32 crq_num);

sdi_close_crq
     uint32 sdi_close_crq(void *adptr,
                          uint16 crq_num);

sdi_get_scatter_gather_info
     uint32 sdi_get_scatter_gather_info(void *adptr,
                                  uint16 *capability,
                                  uint32 *max_entries,
                                  uint32 *max_xfer);

sdi_get_ssa_web_info
     int sdi_get_ssa_web_info(void *adptr,
                          void *web_buff_v
                          uint32 web_buff_p
                          uint32 buff_size);
```

FIG. 7

OPERATING SYSTEM DEPENDENT ROUTINE PROTOTYPES

```
sdi_pci_find_device
     uint32 sdi_pci_find_device(void *driver_context,
                          uint8 *bus,
                          uint8 *device,
                          uint8 *function,
                          uint16 vendor_id,
                          uint16 device_id,
                          uint16 index);

sdi_pci_read_config_register
     uint32 sdi_pci_read_config_register(void *driver_context,
                              uint16 *contents,
                              uint16 reg_num,
                              uint8 bus,
                              uint8 device,
                              uint8 function);

sdi_map_physical_to_virtual
     uint32 sdi_map_physical_to_virtual(void *driver_context
                             uint32 paddr,
                             uint32 size);

sdi_delay_task
     void sdi_delay_task(void *driver_context,
                  uint32 ticks);

sdi_ticks_per_second
     uint32 sdi_delay_task(void *driver_context);
```

FIG. 8

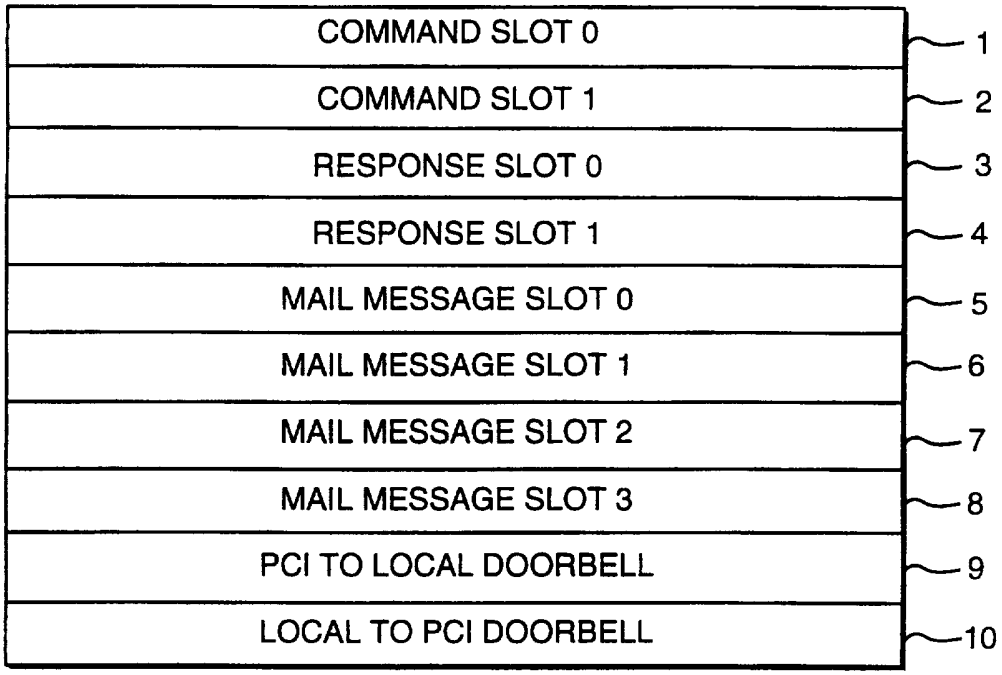

COMMAND SLOT 0
1
COMMAND SLOT 1
2
RESPONSE SLOT 0
3
RESPONSE SLOT 1
4
MAIL MESSAGE SLOT 0
5
MAIL MESSAGE SLOT 1
6
MAIL MESSAGE SLOT 2
7
MAIL MESSAGE SLOT 3
8
PCI TO LOCAL DOORBELL
9
LOCAL TO PCI DOORBELL
10

FIG. 9

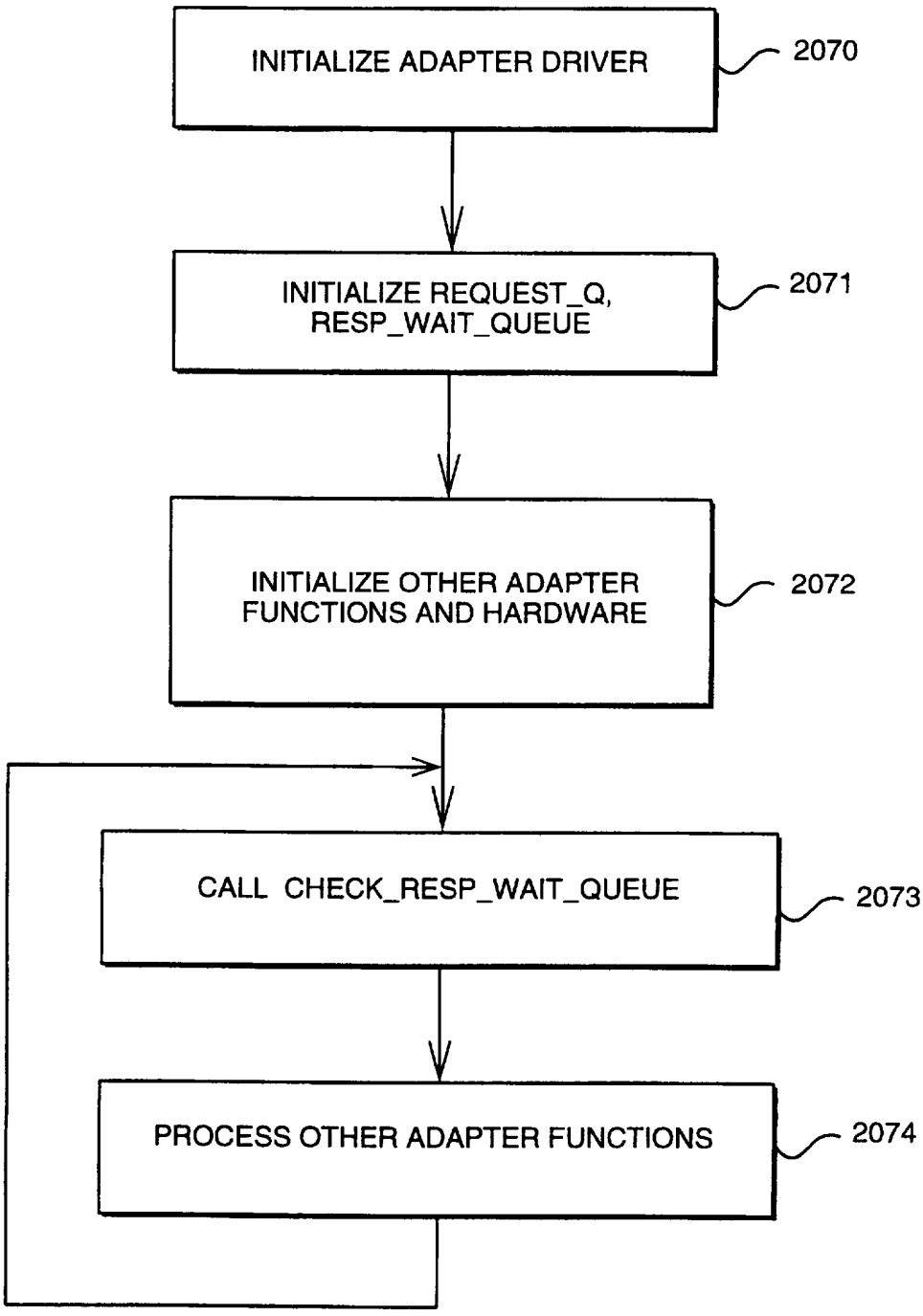

INITIALIZE ADAPTER DRIVER
2070
INITIALIZE REQUEST_Q, RESP_WAIT_QUEUE
2071
INITIALIZE OTHER ADAPTER FUNCTIONS AND HARDWARE
2072
CALL CHECK_RESP_WAIT_QUEUE
2073
PROCESS OTHER ADAPTER FUNCTIONS
2074

FIG. 10

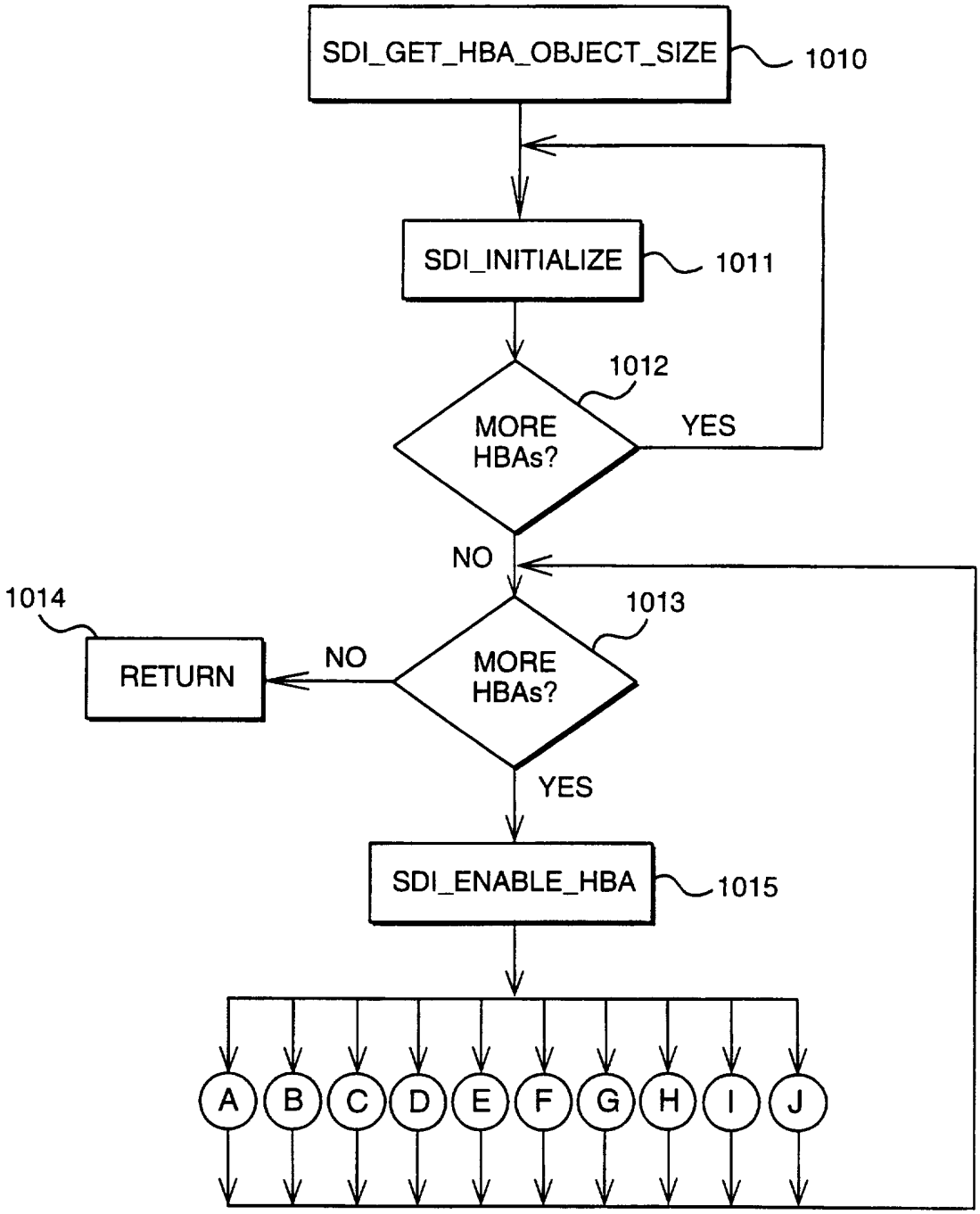

SDI_GET_HBA_OBJECT_SIZE
1010
SDI_INITIALIZE
1011
1012
MORE HBAs?
YES
NO
1014
1013
RETURN
NO
MORE HBAs?
YES
SDI_ENABLE_HBA
1015
A
B
C
D
E
F
G
H
I
J

FIG. 11

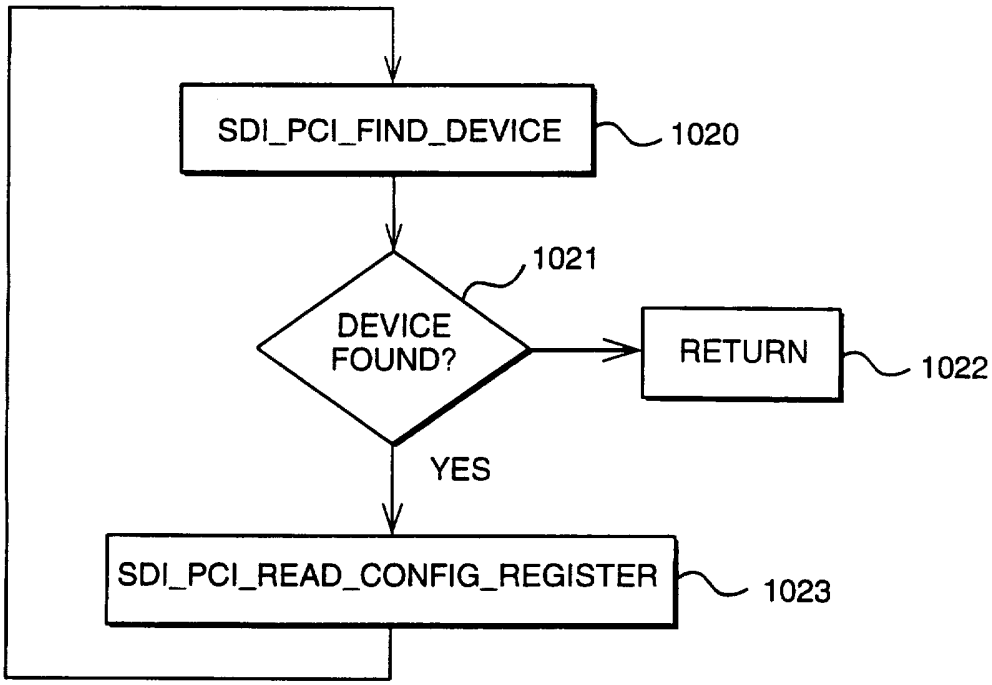

SDI_PCI_FIND_DEVICE
1020
1021
DEVICE FOUND?
RETURN
1022
YES
SDI_PCI_READ_CONFIG_REGISTER
1023

FIG. 12

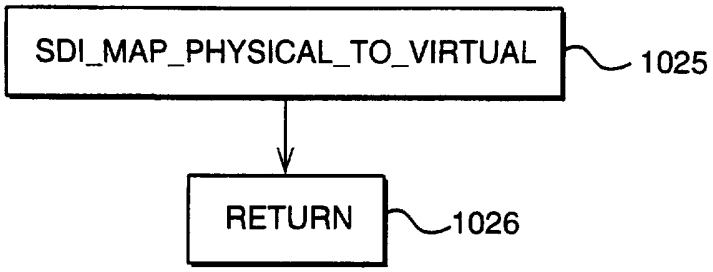

SDI_MAP_PHYSICAL_TO_VIRTUAL
1025
RETURN
1026

FIG. 13

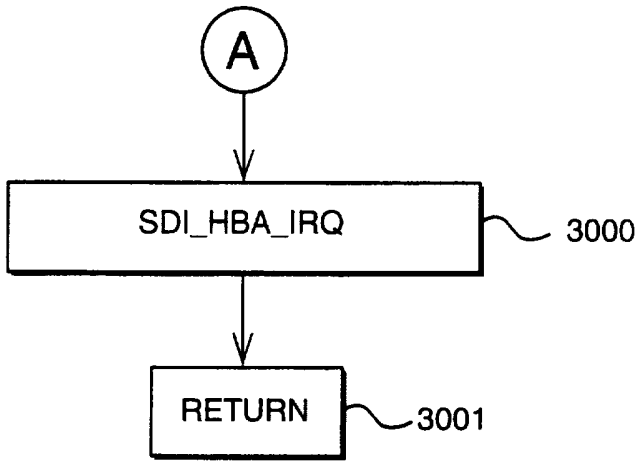


FIG. 14

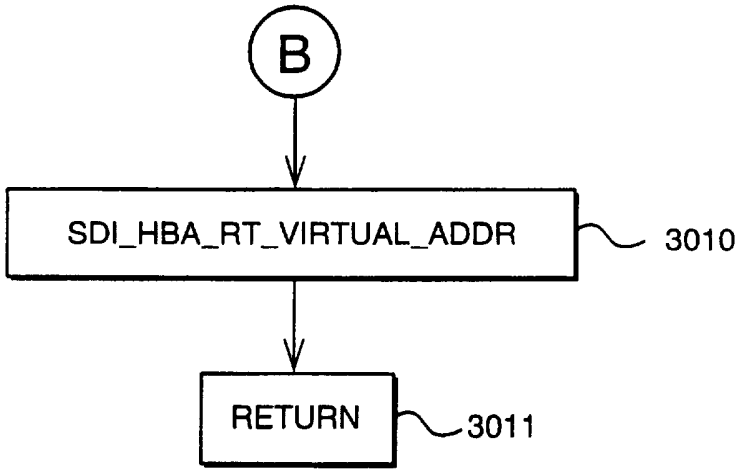

B
SDI_HBA_RT_VIRTUAL_ADDR
3010
RETURN
3011

FIG. 15

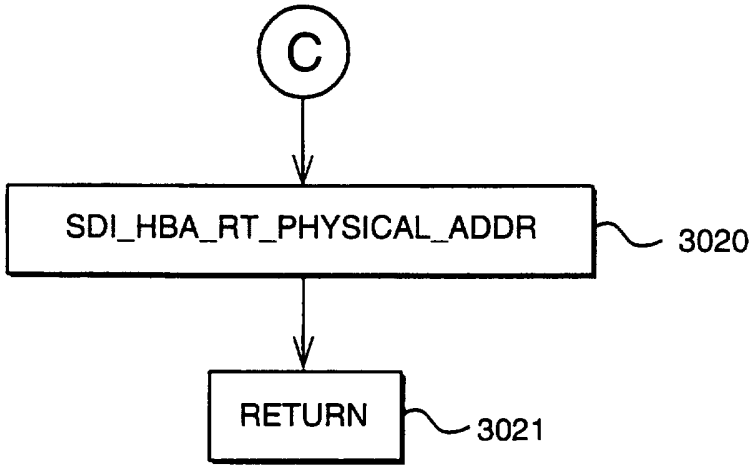

C
SDI_HBA_RT_PHYSICAL_ADDR
3020
RETURN
3021

FIG. 16

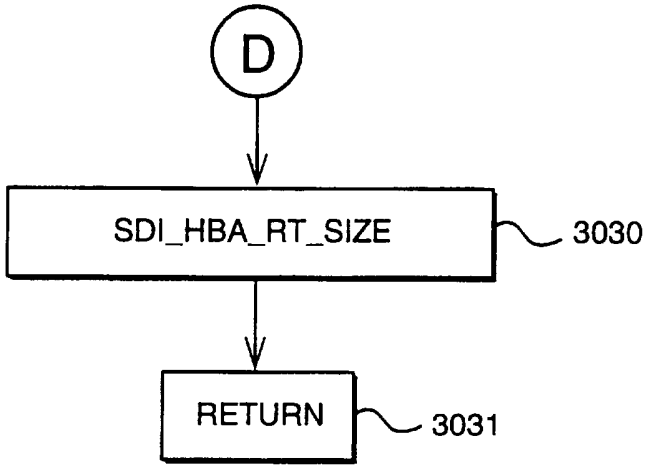

D
SDI_HBA_RT_SIZE
3030
RETURN
3031

FIG. 17

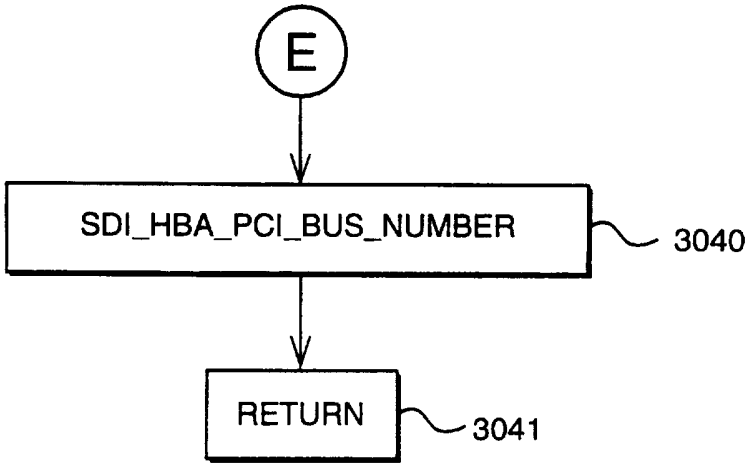

E
SDI_HBA_PCI_BUS_NUMBER
3040
RETURN
3041

FIG. 18

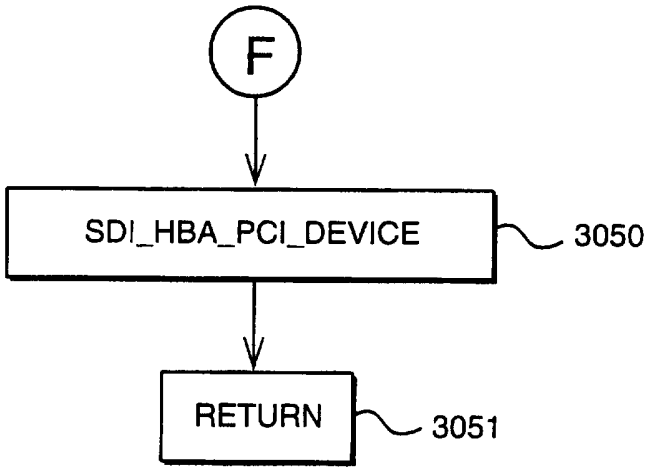

F
SDI_HBA_PCI_DEVICE
3050
RETURN
3051

FIG. 19

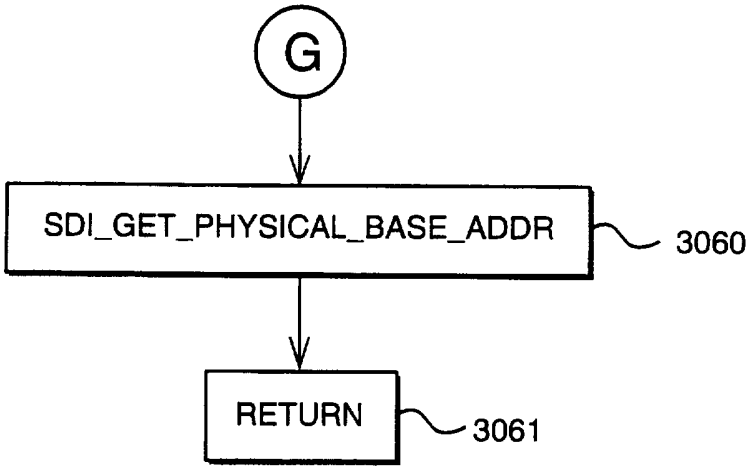

G
SDI_GET_PHYSICAL_BASE_ADDR
3060
RETURN
3061

FIG. 20

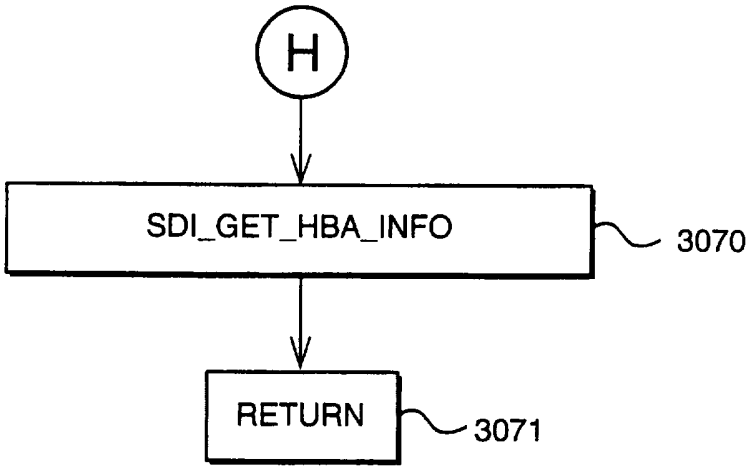


FIG. 21

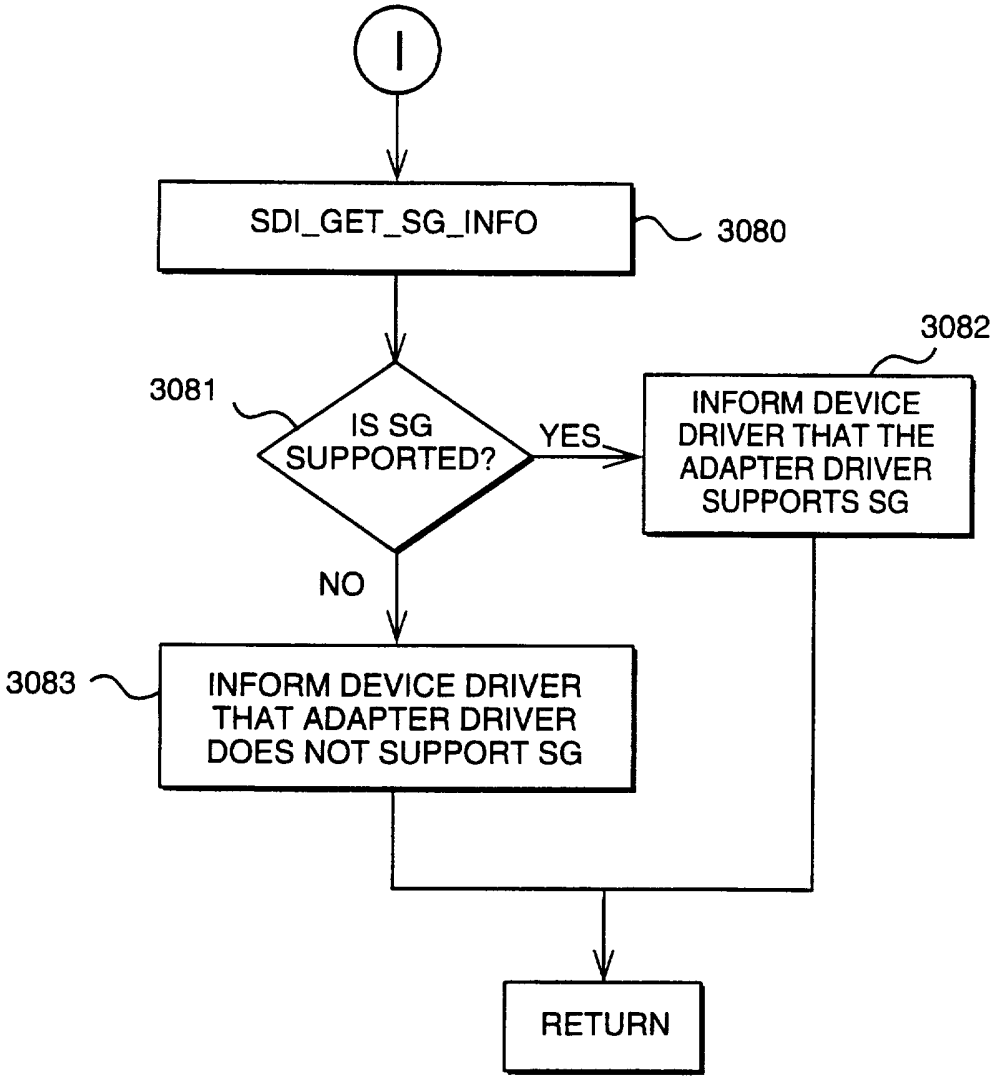

I
SDI_GET_SG_INFO
3080
3081
IS SG SUPPORTED?
YES
3082
INFORM DEVICE DRIVER THAT THE ADAPTER DRIVER SUPPORTS SG
NO
3083
INFORM DEVICE DRIVER THAT ADAPTER DRIVER DOES NOT SUPPORT SG
RETURN

FIG. 22

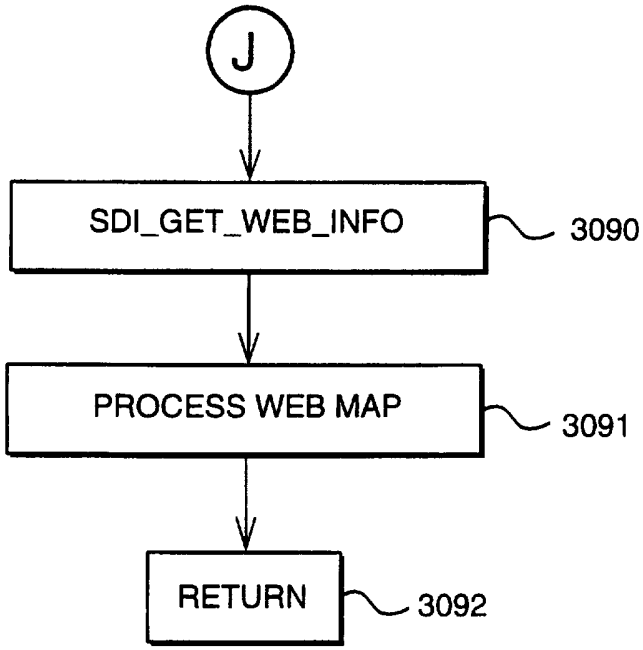


FIG. 23

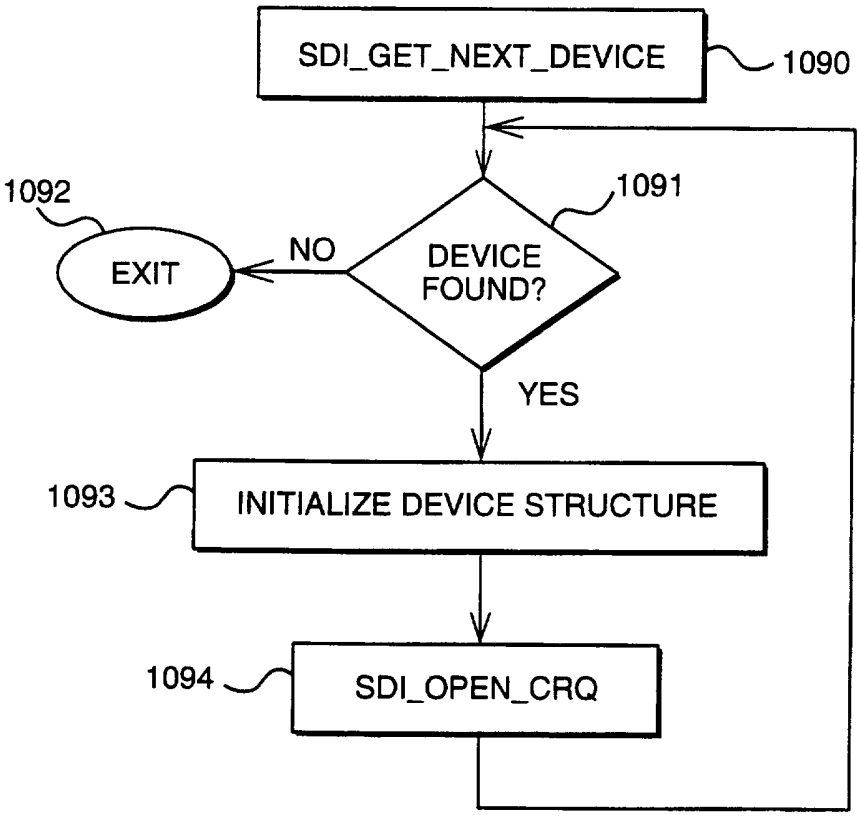

SDI_GET_NEXT_DEVICE
1090
1091
1092
EXIT
NO
DEVICE FOUND?
YES
1093
INITIALIZE DEVICE STRUCTURE
1094
SDI_OPEN_CRQ

FIG. 24

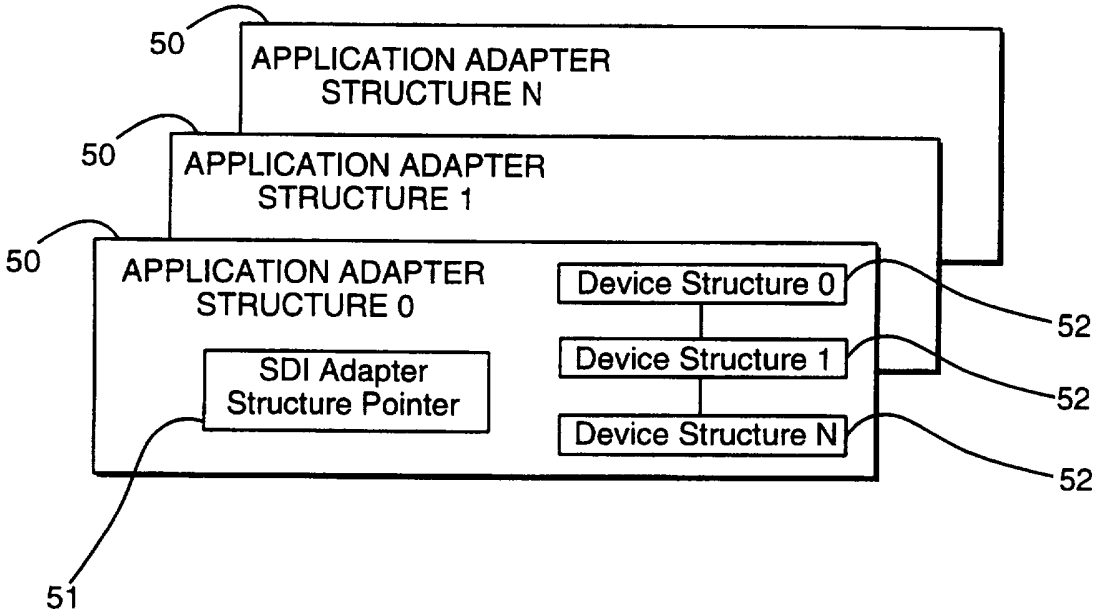

50
APPLICATION ADAPTER STRUCTURE N
50
APPLICATION ADAPTER STRUCTURE 1
50
APPLICATION ADAPTER STRUCTURE 0
SDI Adapter Structure Pointer
51
Device Structure 0
Device Structure 1
Device Structure N
52
52
52

FIG. 25

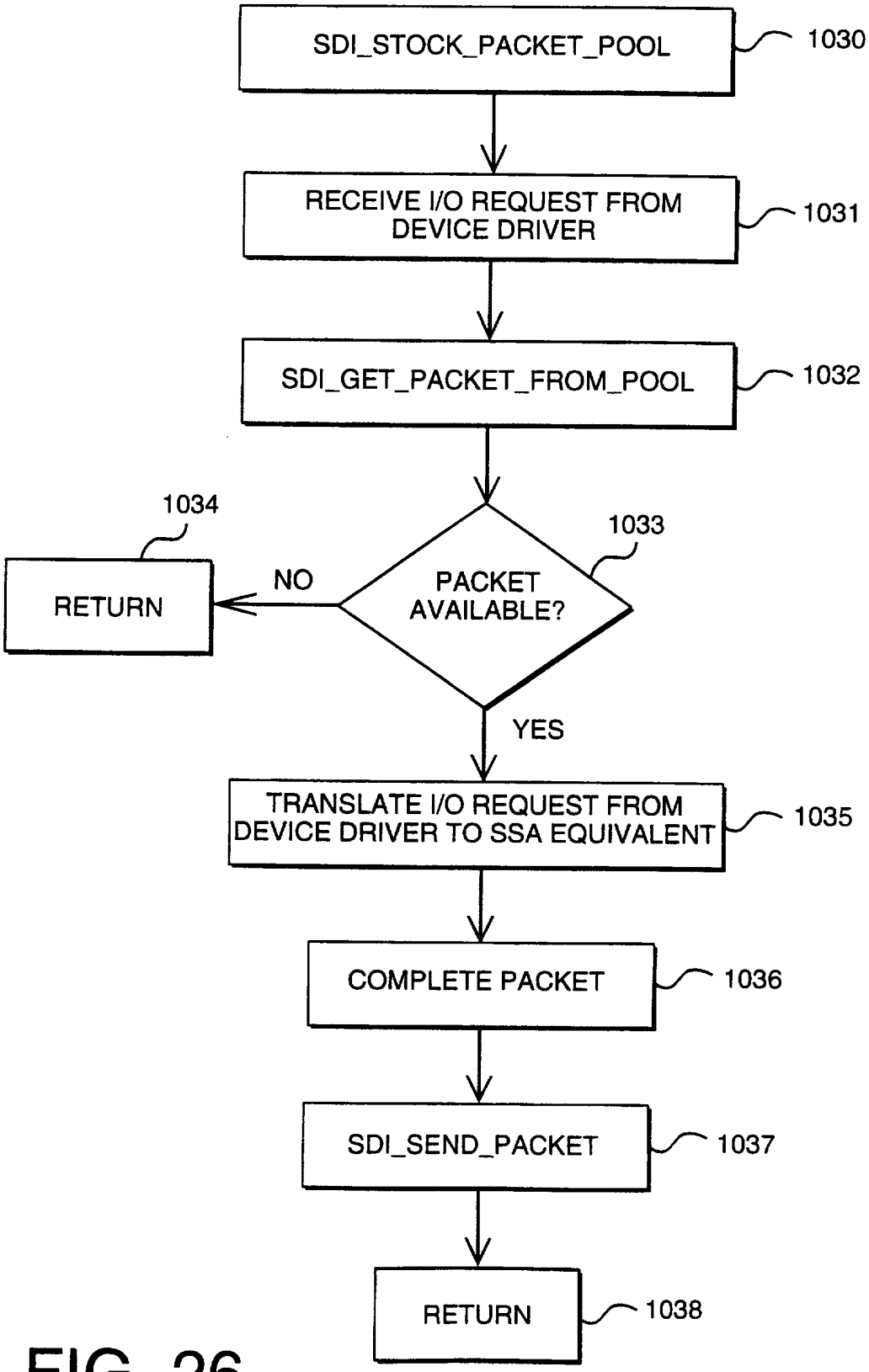

SDI_STOCK_PACKET_POOL
1030
RECEIVE I/O REQUEST FROM DEVICE DRIVER
1031
SDI_GET_PACKET_FROM_POOL
1032
1034
1033
RETURN
NO
PACKET AVAILABLE?
YES
TRANSLATE I/O REQUEST FROM DEVICE DRIVER TO SSA EQUIVALENT
1035
COMPLETE PACKET
1036
SDI_SEND_PACKET
1037
RETURN
1038

FIG. 26

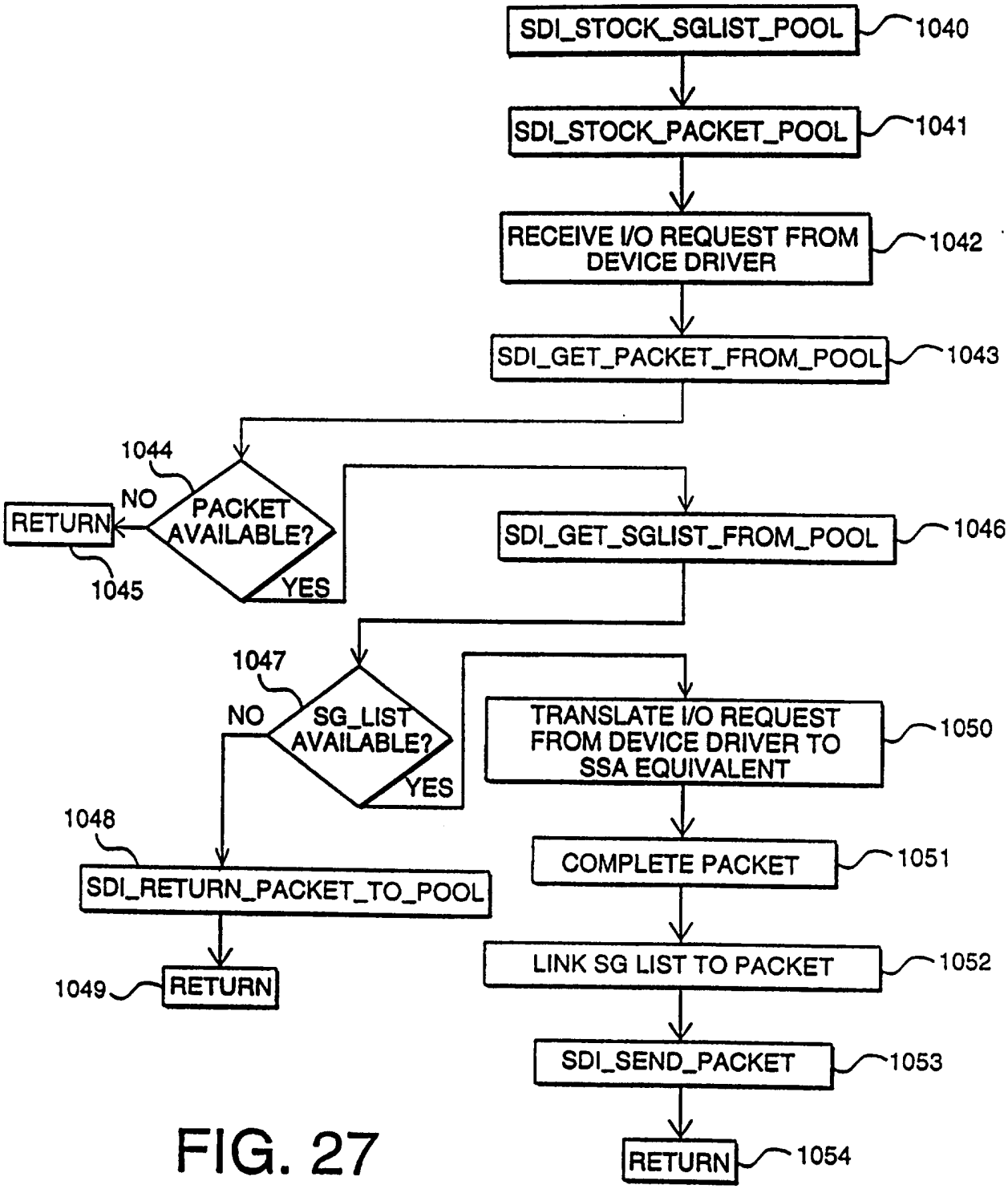

SDI_STOCK_SGLIST_POOL
1040
SDI_STOCK_PACKET_POOL
1041
RECEIVE I/O REQUEST FROM DEVICE DRIVER
1042
SDI_GET_PACKET_FROM_POOL
1043
1044
PACKET AVAILABLE?
NO
YES
RETURN
1045
SDI_GET_SGLIST_FROM_POOL
1046
1047
SG_LIST AVAILABLE?
NO
YES
TRANSLATE I/O REQUEST FROM DEVICE DRIVER TO SSA EQUIVALENT
1050
1048
SDI_RETURN_PACKET_TO_POOL
COMPLETE PACKET
1051
1049
RETURN
LINK SG LIST TO PACKET
1052
SDI_SEND_PACKET
1053
RETURN
1054

FIG. 27

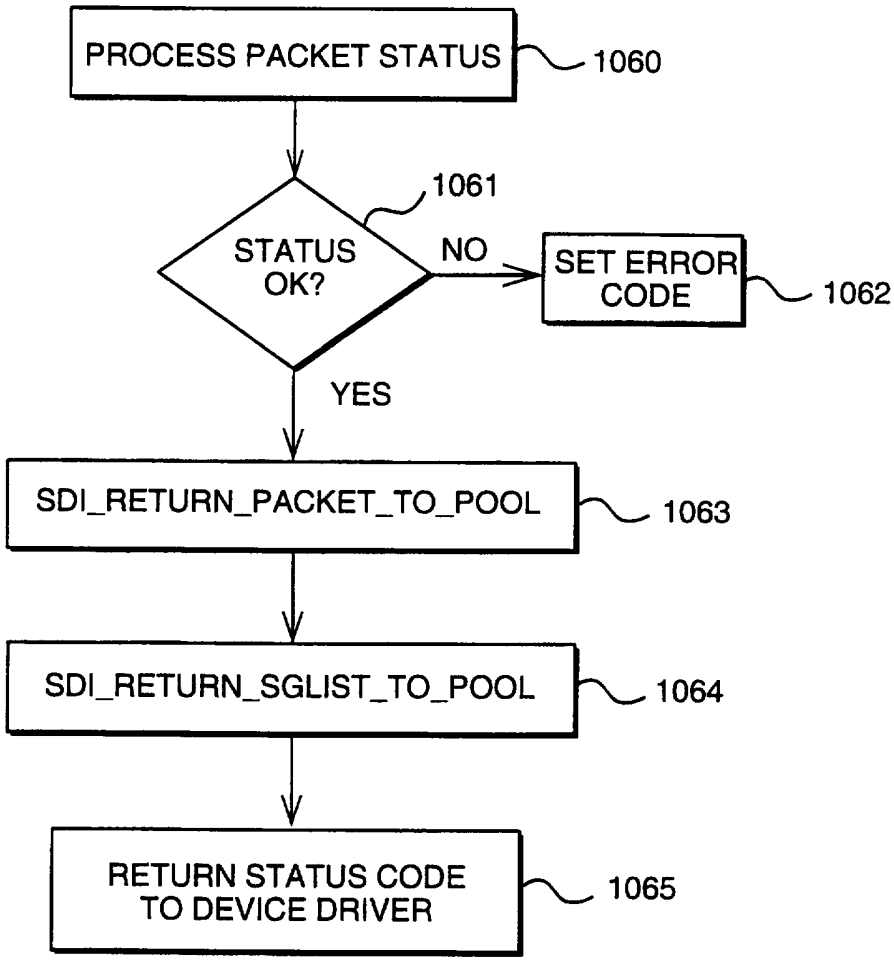

PROCESS PACKET STATUS
1060
1061
STATUS OK?
NO
SET ERROR CODE
1062
YES
SDI_RETURN_PACKET_TO_POOL
1063
SDI_RETURN_SGLIST_TO_POOL
1064
RETURN STATUS CODE TO DEVICE DRIVER
1065

FIG. 28

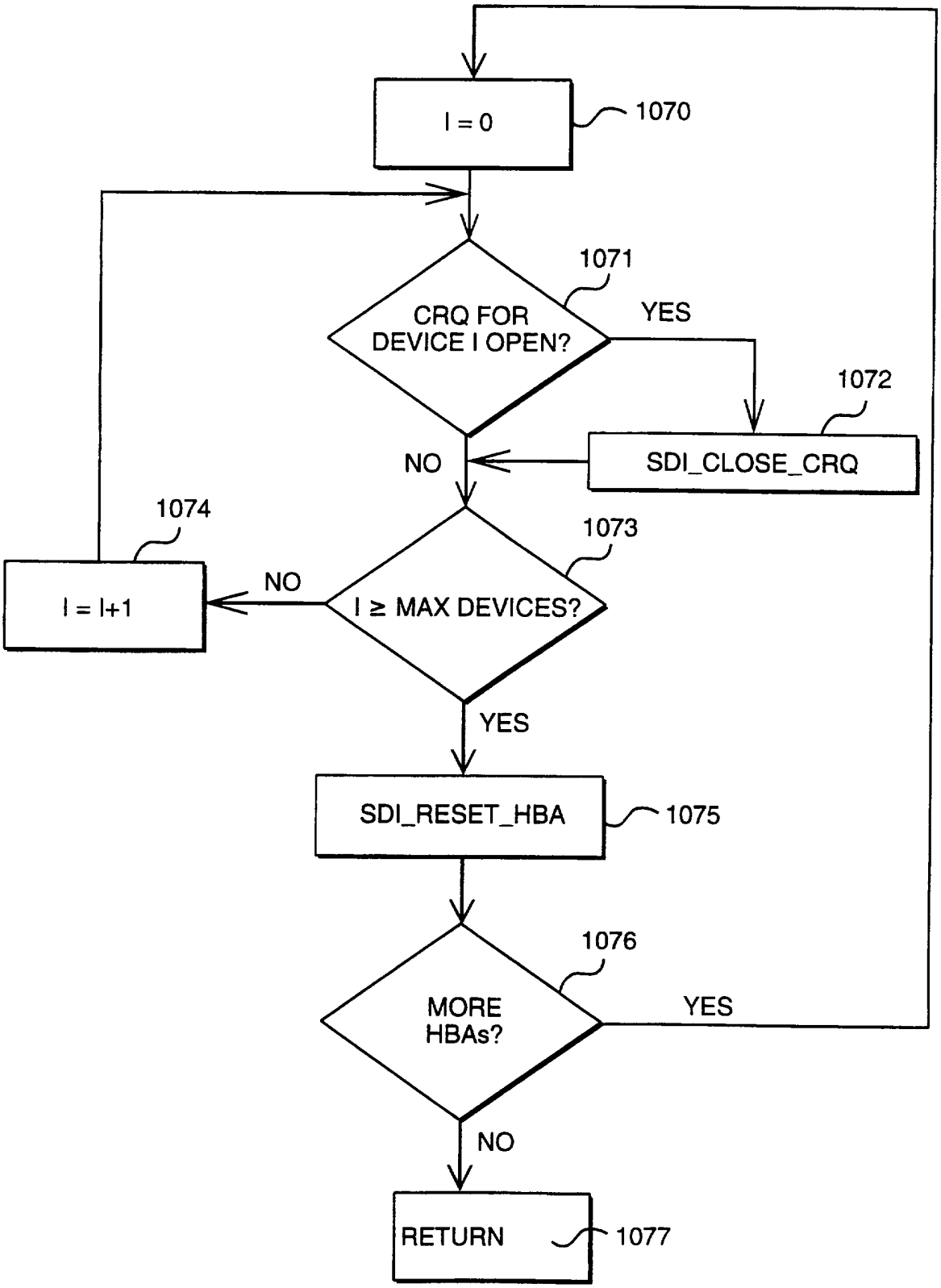

I = 0
1070
1071
CRQ FOR DEVICE I OPEN?
YES
1072
SDI_CLOSE_CRQ
NO
1074
1073
I = I+1
NO
I ≥ MAX DEVICES?
YES
SDI_RESET_HBA
1075
1076
MORE HBAs?
YES
NO
RETURN
1077

FIG. 29

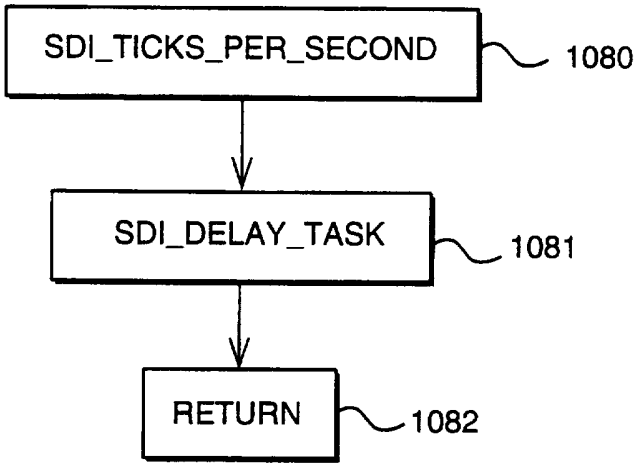


FIG. 30

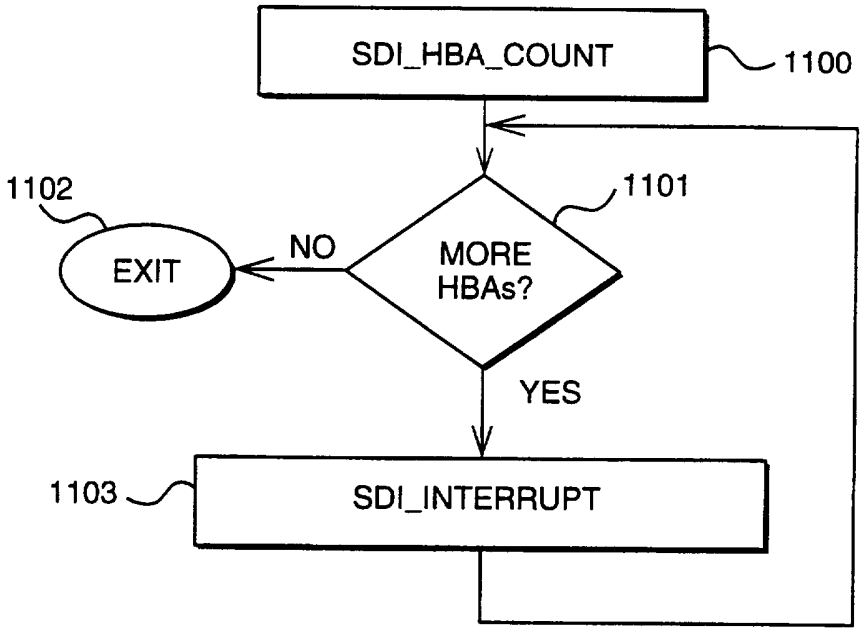

SDI_HBA_COUNT
1100
1101
1102
EXIT
NO
MORE
HBAs?
YES
1103
SDI_INTERRUPT

FIG. 31

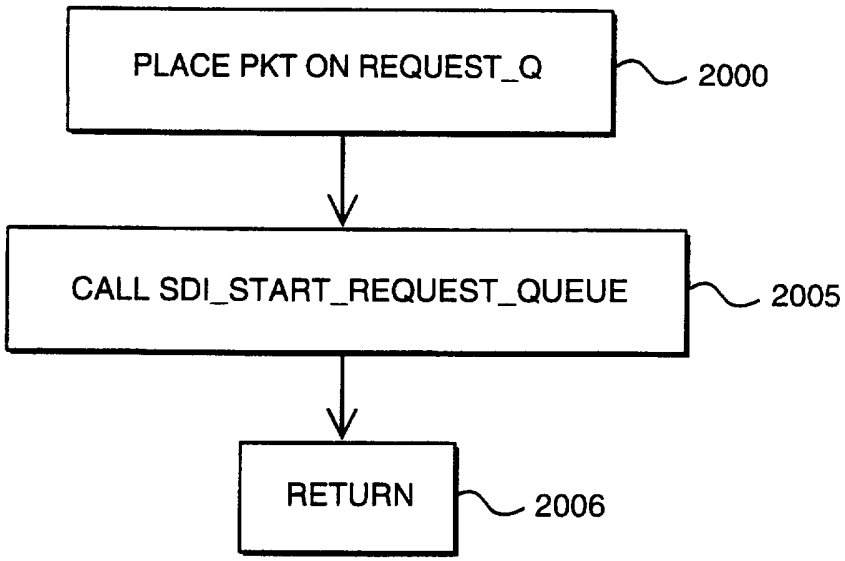

PLACE PKT ON REQUEST_Q
2000
CALL SDI_START_REQUEST_QUEUE
2005
RETURN
2006

FIG. 32

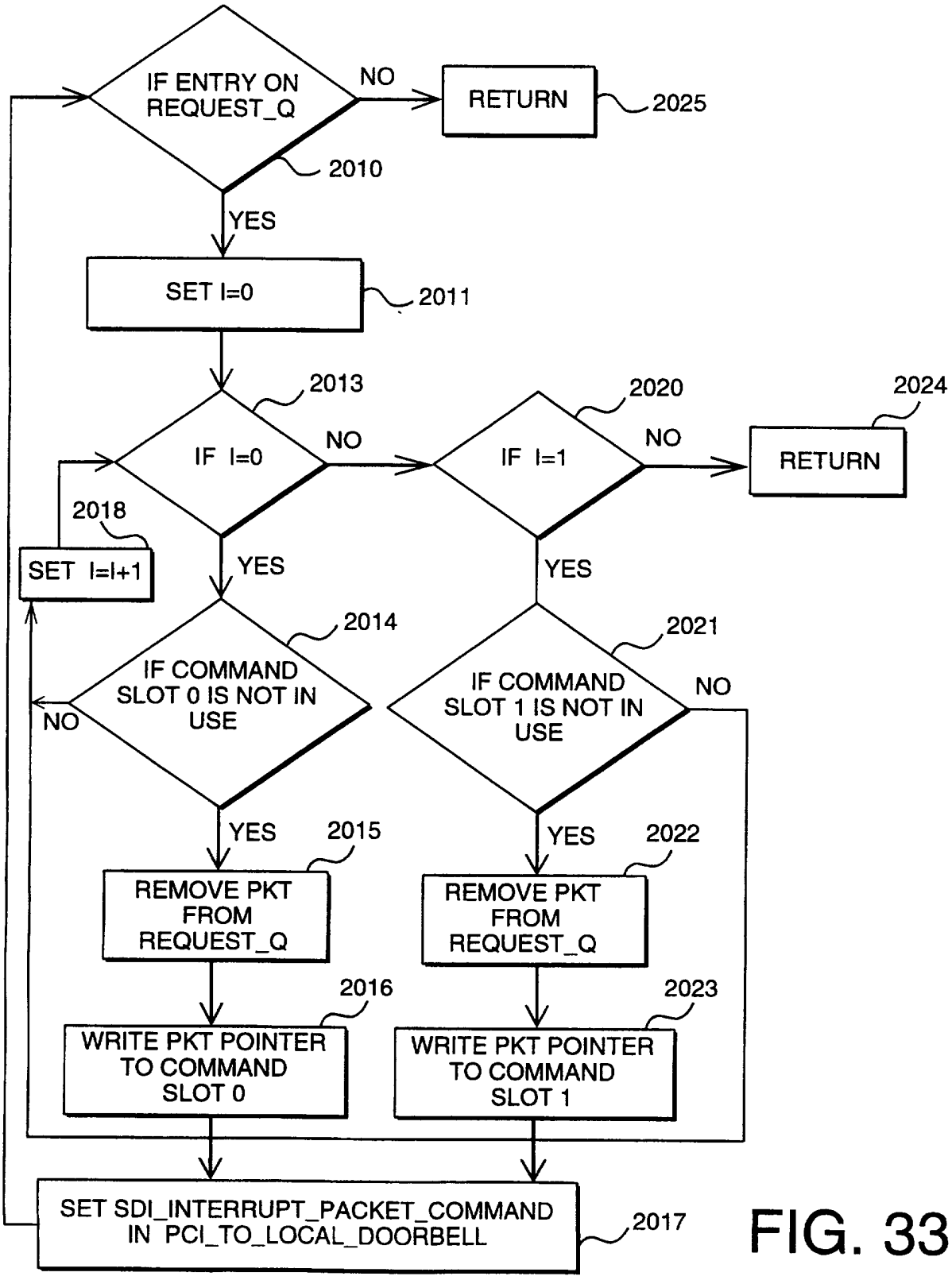

IF ENTRY ON REQUEST_Q
NO
RETURN
2025
2010
YES
SET I=0
2011
2013
IF I=0
NO
2020
IF I=1
NO
2024
RETURN
2018
SET I=I+1
YES
YES
2014
IF COMMAND SLOT 0 IS NOT IN USE
NO
2021
IF COMMAND SLOT 1 IS NOT IN USE
NO
YES
2015
REMOVE PKT FROM REQUEST_Q
YES
2022
REMOVE PKT FROM REQUEST_Q
2016
WRITE PKT POINTER TO COMMAND SLOT 0
2023
WRITE PKT POINTER TO COMMAND SLOT 1
SET SDI_INTERRUPT_PACKET_COMMAND IN PCI_TO_LOCAL_DOORBELL
2017

FIG. 33

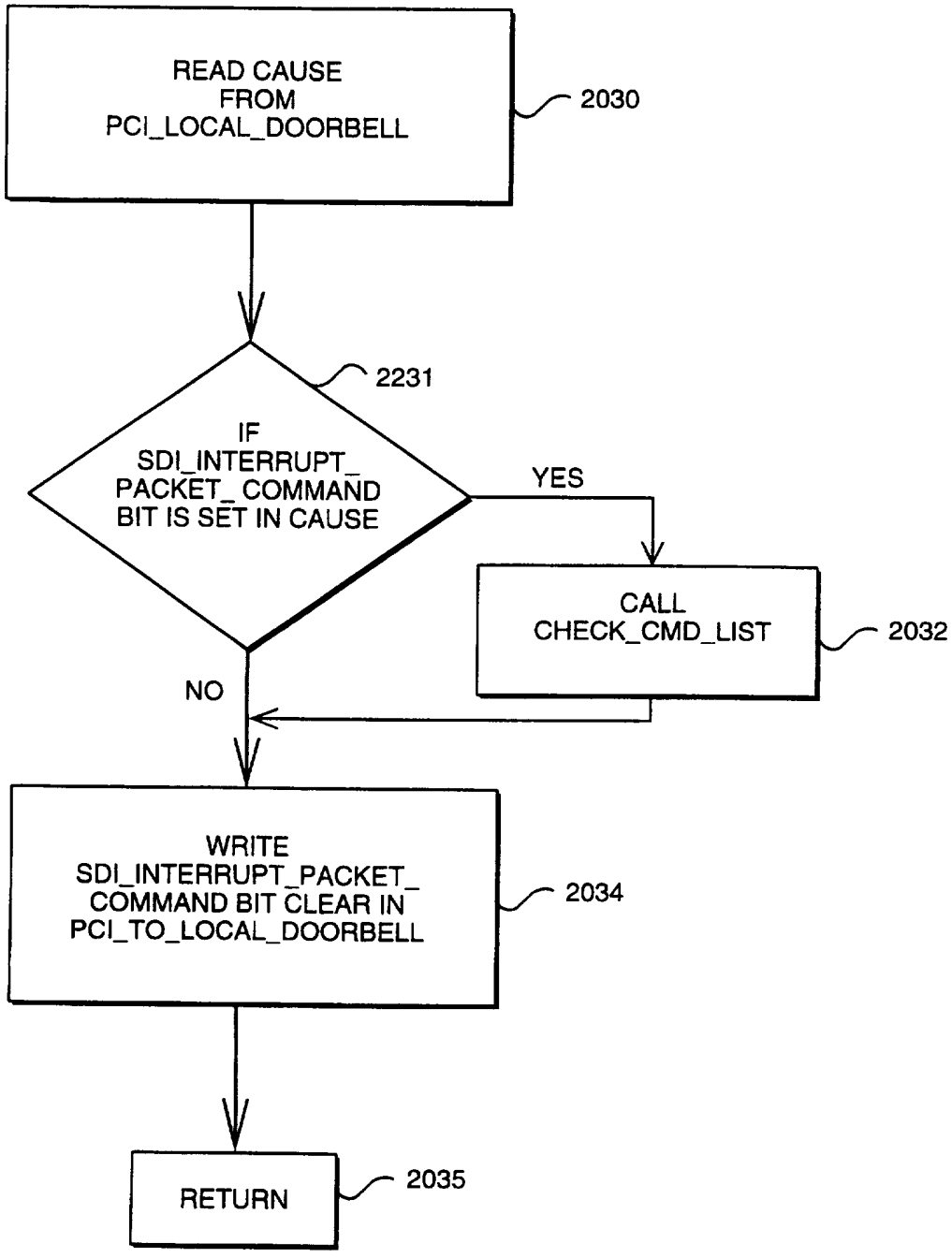

READ CAUSE FROM PCI_LOCAL_DOORBELL
2030
2231
IF SDI_INTERRUPT_ PACKET_ COMMAND BIT IS SET IN CAUSE
YES
CALL CHECK_CMD_LIST
2032
NO
WRITE SDI_INTERRUPT_PACKET_ COMMAND BIT CLEAR IN PCI_TO_LOCAL_DOORBELL
2034
RETURN
2035

FIG. 34

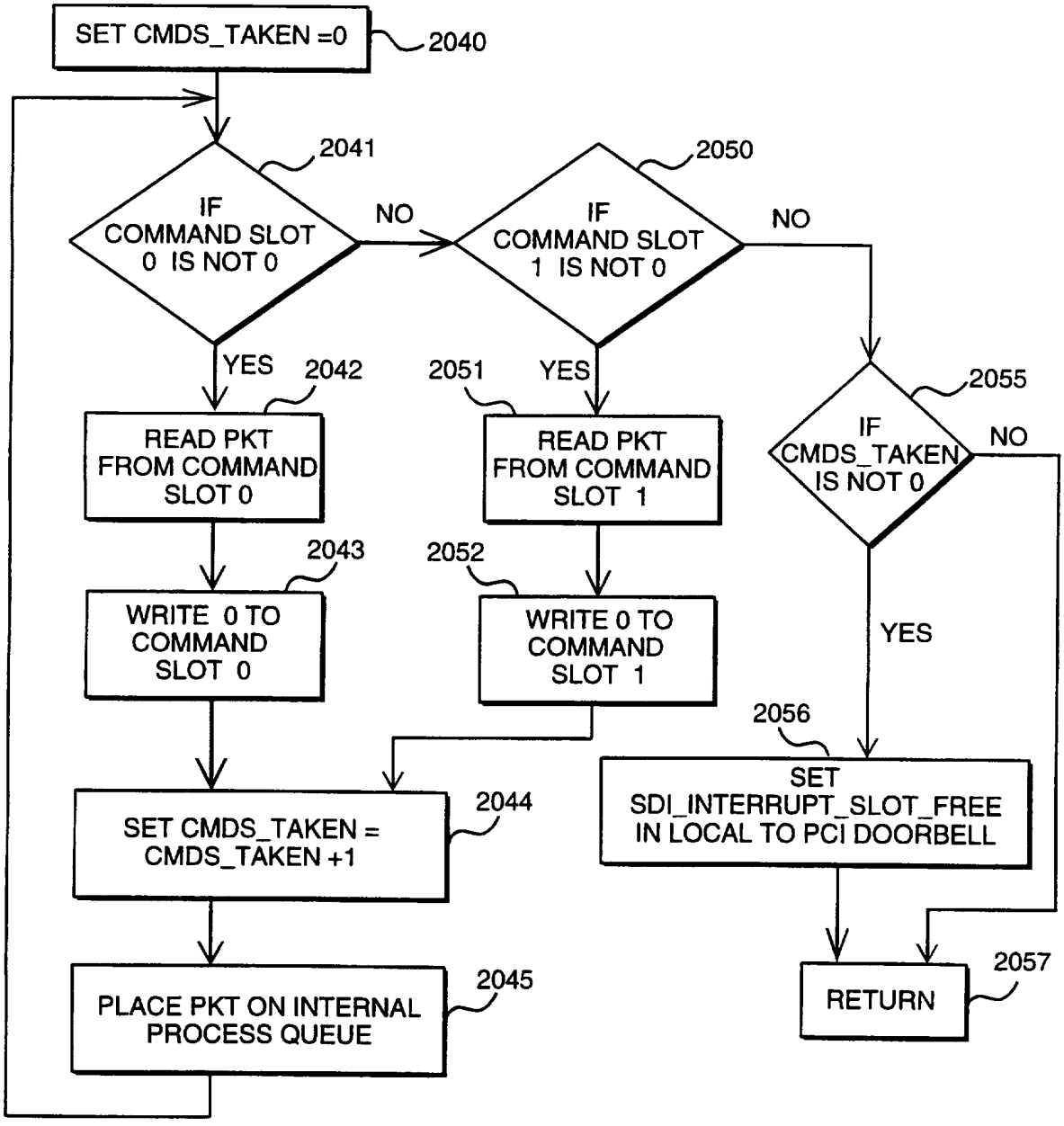

SET CMDS_TAKEN =0
2040
2041
IF COMMAND SLOT 0 IS NOT 0
NO
2050
IF COMMAND SLOT 1 IS NOT 0
NO
YES
2042
READ PKT FROM COMMAND SLOT 0
2043
WRITE 0 TO COMMAND SLOT 0
2051
YES
READ PKT FROM COMMAND SLOT 1
2052
WRITE 0 TO COMMAND SLOT 1
2055
IF CMDS_TAKEN IS NOT 0
NO
YES
2056
SET SDI_INTERRUPT_SLOT_FREE IN LOCAL TO PCI DOORBELL
SET CMDS_TAKEN = CMDS_TAKEN +1
2044
PLACE PKT ON INTERNAL PROCESS QUEUE
2045
RETURN
2057

FIG. 35

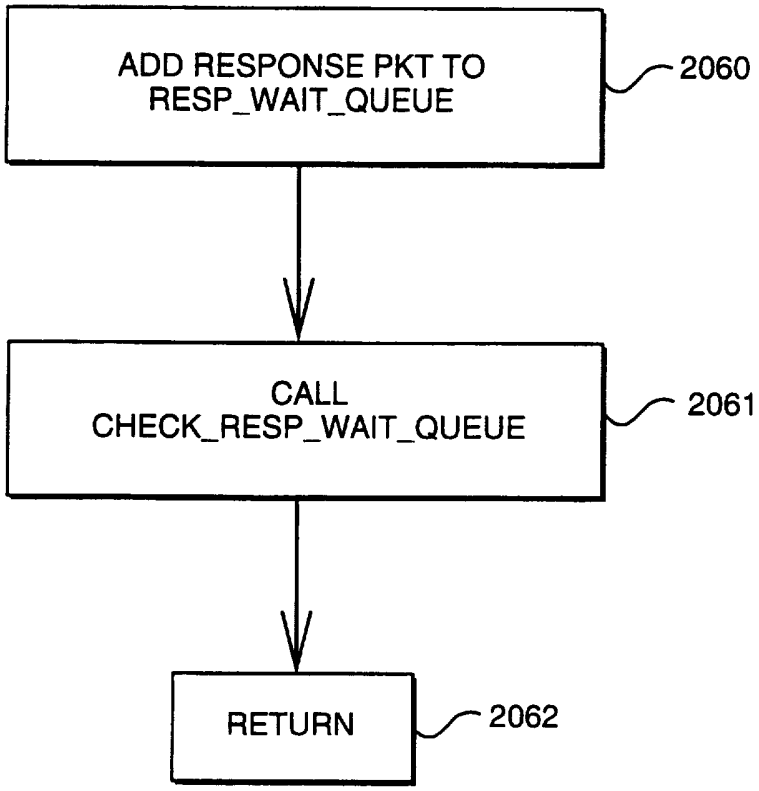

ADD RESPONSE PKT TO RESP_WAIT_QUEUE
2060
CALL CHECK_RESP_WAIT_QUEUE
2061
RETURN
2062

FIG. 36

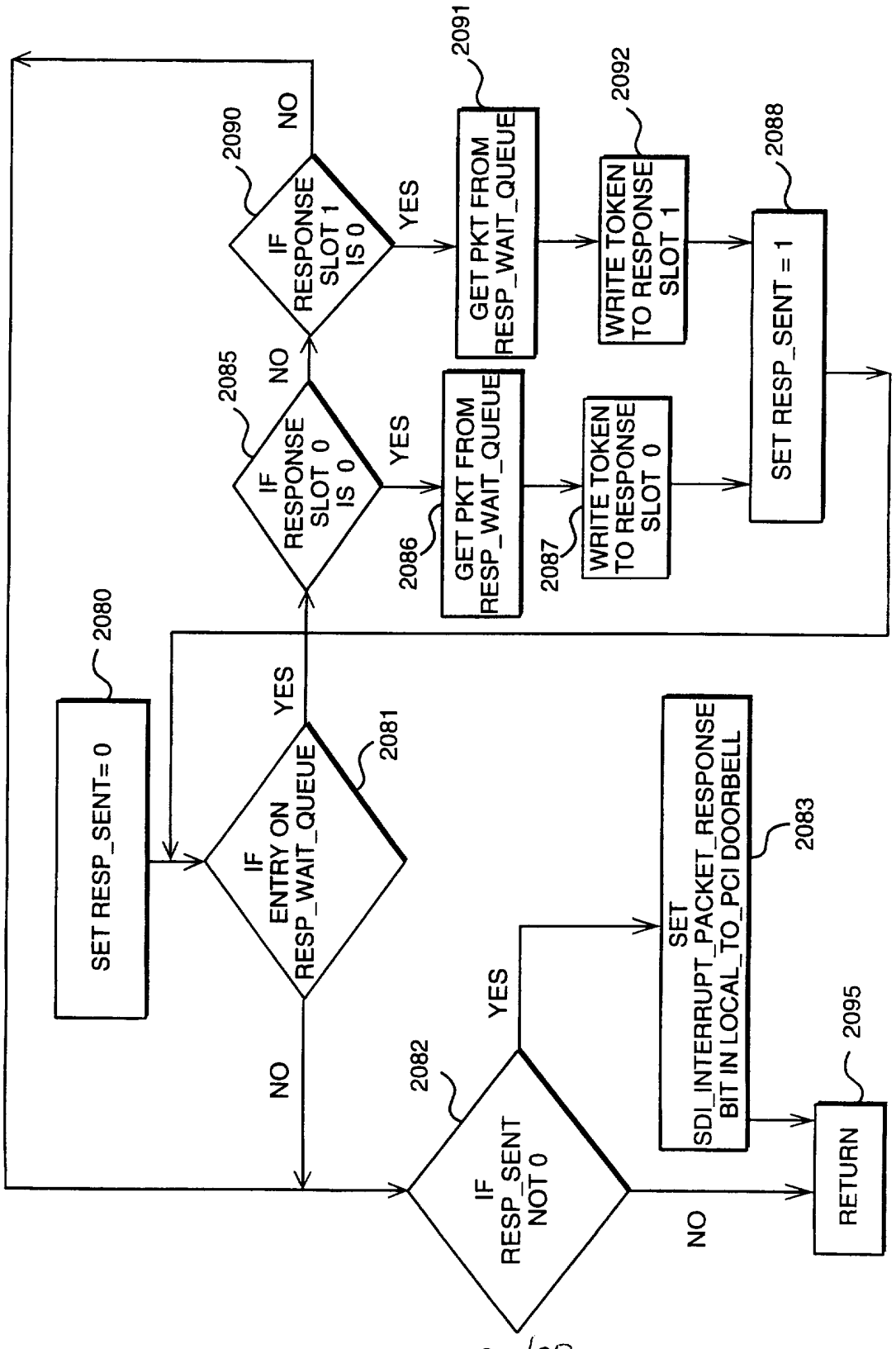

SET RESP_SENT= 0
2080
IF ENTRY ON RESP_WAIT_QUEUE
2081
YES
NO
IF RESP_SENT NOT 0
2082
YES
NO
SET SDI_INTERRUPT_PACKET_RESPONSE BIT IN LOCAL_TO_PCI DOORBELL
2083
RETURN
2095
IF RESPONSE SLOT 0 IS 0
2085
YES
NO
GET PKT FROM RESP_WAIT_QUEUE
2086
WRITE TOKEN TO RESPONSE SLOT 0
2087
IF RESPONSE SLOT 1 IS 0
2090
YES
NO
GET PKT FROM RESP_WAIT_QUEUE
2091
WRITE TOKEN TO RESPONSE SLOT 1
2092
SET RESP_SENT = 1
2088

FIG. 37

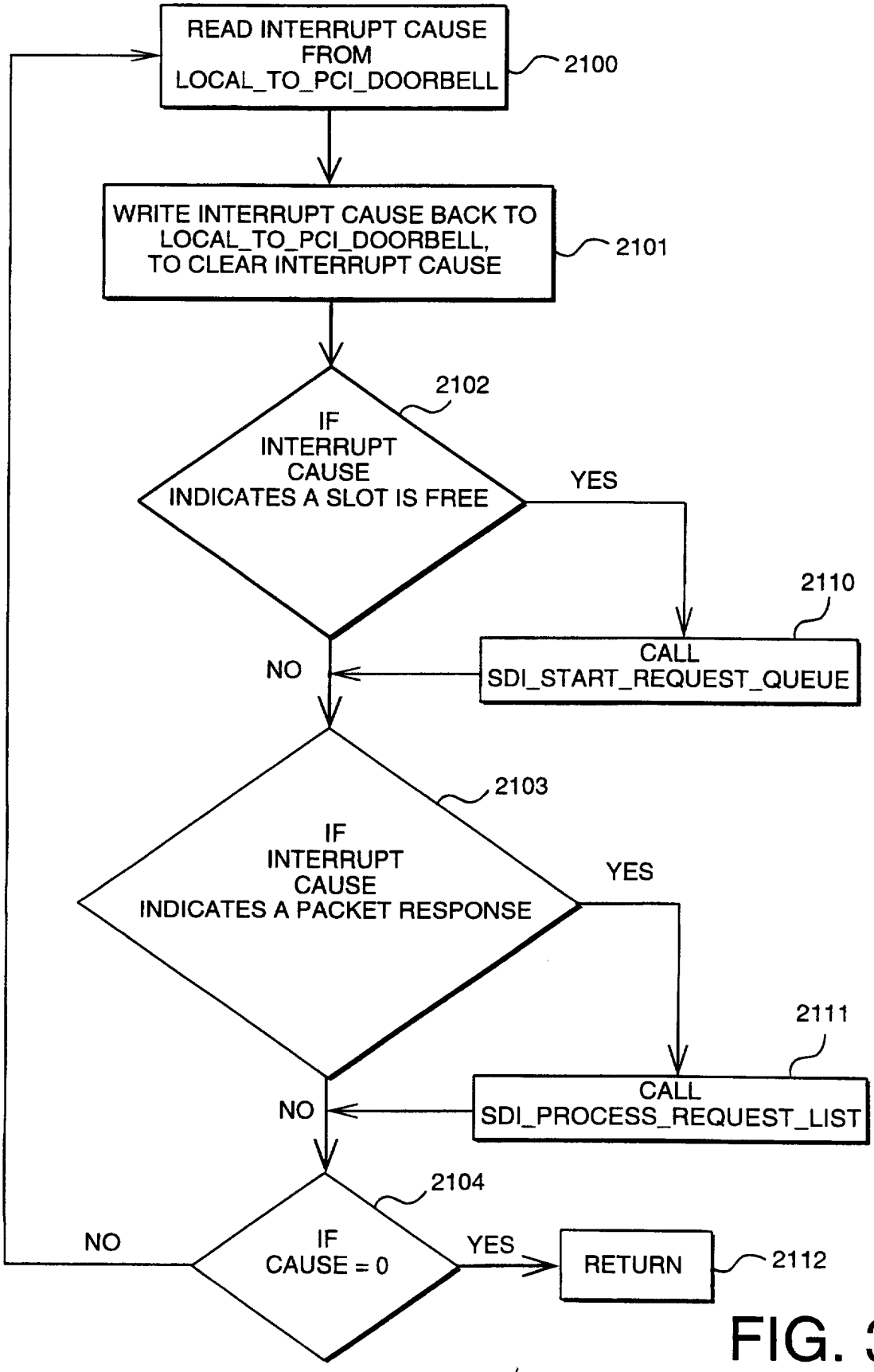

READ INTERRUPT CAUSE FROM LOCAL_TO_PCI_DOORBELL
2100
WRITE INTERRUPT CAUSE BACK TO LOCAL_TO_PCI_DOORBELL, TO CLEAR INTERRUPT CAUSE
2101
2102
IF INTERRUPT CAUSE INDICATES A SLOT IS FREE
YES
2110
CALL SDI_START_REQUEST_QUEUE
NO
2103
IF INTERRUPT CAUSE INDICATES A PACKET RESPONSE
YES
2111
CALL SDI_PROCESS_REQUEST_LIST
NO
2104
IF CAUSE = 0
NO
YES
RETURN
2112

FIG. 38

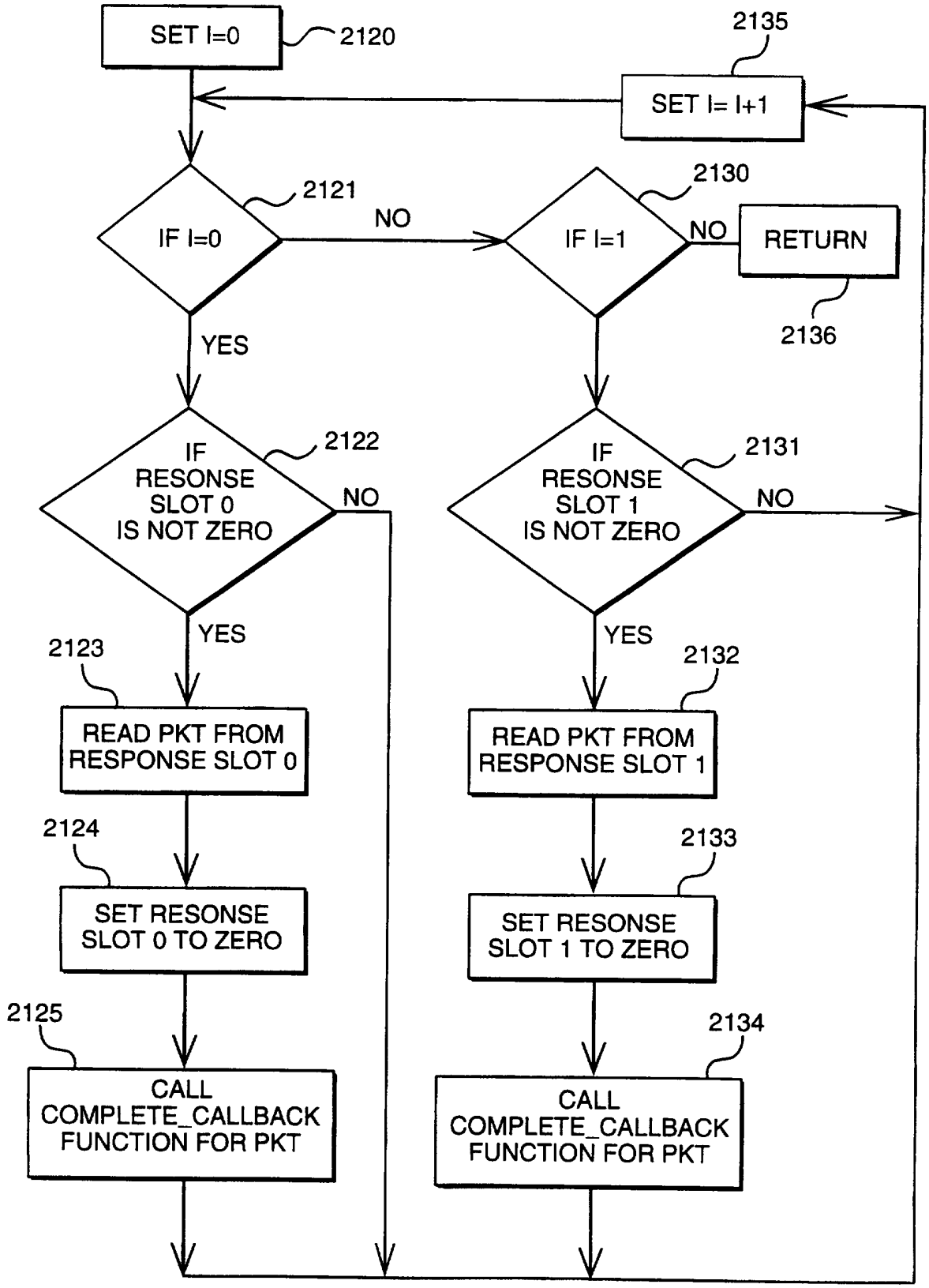

SET I=0
2120
2135
SET I= I+1
2121
IF I=0
NO
2130
IF I=1
NO
RETURN
2136
YES
2122
IF RESONSE SLOT 0 IS NOT ZERO
NO
2131
IF RESONSE SLOT 1 IS NOT ZERO
NO
YES
YES
2123
READ PKT FROM RESPONSE SLOT 0
2132
READ PKT FROM RESPONSE SLOT 1
2124
SET RESONSE SLOT 0 TO ZERO
2133
SET RESONSE SLOT 1 TO ZERO
2125
CALL COMPLETE_CALLBACK FUNCTION FOR PKT
2134
CALL COMPLETE_CALLBACK FUNCTION FOR PKT

FIG. 39

37/37

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US97/01104

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :G06F 19/00
US CL : 395/681, 285, 182.03, 182.05, 828, 825, 835
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/681, 285, 182.03, 182.05, 828, 825, 835

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 5,307,491 A (FERIOZI ET AL) 26 APRIL 1994 col. 5, lines 1-68, col. 7, lines 7-68, col. 8, lines 1-55	1-25
Y,P	US 5,564,061 A (DAVIES ET AL) 08 OCTOBER 1996 col. 7, lines 20-68	1-25
Y	US 5,530,897 A (MERITT) 25 JANUARY 1996 col. 7, lines 25-68, col. 8, lines 1-68, col. 9, lines 20-50	1-25

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 09 APRIL 1997	Date of mailing of the international search report 14 MAY 1997
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer PETER STECHER Telephone No. (703) 305-4005

Form PCT/ISA/210 (second sheet)(July 1992)*