

ENABLE/DISABLE ADAPTERS OF A COMPUTING ENVIRONMENT

BACKGROUND

5 This invention relates, in general, to input/output processing of a computing environment, and in particular, to enabling/disabling adapters of the computing environment.

Today, computing environments have various configurations and use various types of input/output (I/O) devices. In order to use an I/O device, it is enabled, and then once its use
10 is complete, it is disabled. The manner in which an I/O device is enabled/disabled is device-dependent.

In the z/Architecture® and its predecessors offered by International Business Machines Corporation, the enablement and disablement of I/O devices have traditionally been
15 performed on a channel path, control unit and subchannel basis. The various functions of a Channel Subsystem Call instruction provide interfaces by which operating systems can manipulate the various I/O resources.

Other types of I/O devices, however, may be used that do not include channels and
20 subchannels. For instance, peripheral component interconnect (PCI) adapters use attachment and communication paradigms that are different than that of traditional I/O devices. The PCI specification is available from the World Wide Web at www.pcisig.com/home.

US Publication No. 2004/0117534 A1, published June 17, 2004, Parry et al., "Apparatus and
25 Method for Dynamically Enabling and Disabling Interrupt Coalescing in Data Processing System," describes an apparatus and method for dynamically enabling and disabling interrupt coalescing in a data processing system. The present invention involves consistently monitoring IO load on an IOP of an IO adapter. The firmware on the IO adapter may have a global variable that stores counters for PCI function registers. Each counter tracks the
30 number of outstanding IOs of a corresponding PCI function register. The counter is incremented whenever a new IO is received and is decremented upon posting the completed message back to the OS. A timer interrupt is generated periodically so that an ISR may be

periodically performed. In the ISR, the maximum value stored of each counter seen since last timer interrupt is analyzed. When the maximum value stored is greater than a predetermined threshold value, the interrupt coalescing is enabled.

5 US Publication No. 2010/0005234 A1, published January 7, 2010, Ganga et al., "Enabling Functional Dependency in a Multi-Function Device," describes in one embodiment, the present invention includes a method for reading configuration information from a multi-function device (MFD), building a dependency tree of a functional dependency of functions performed by the MFD based on the configuration information, which indicates that the
10 MFD is capable of performing at least one function dependent upon another function, and loading software associated with the functions in order based at least in part on the indicated functional dependency. Other embodiments are described and claimed.

15 US Publication No. 2004/0199700 A1, published October 7, 2004, Shawn Adam Clayton, "Virtual Peripheral Component Interconnect Multiple-Function Device," describes a peripheral component interconnect (PCI) device comprising a bus interface coupled to a component interconnect bus, a plurality of configuration space register sets, and virtual multiple-function logic. Each set of configuration space registers is associated with a function. The virtual multiple-function logic is coupled to the bus interface and the
20 configuration space register sets. The virtual multiple-function logic provides access to a plurality of configuration space registers for a plurality of functions. The virtual multiple-function logic also enables a plurality of functions to share the bus interface and other internal logic.

25 BRIEF SUMMARY

In accordance with an aspect of the present invention, a capability is provided for enabling/disabling adapters, such as PCI adapters. In one example, the capability, as it appears to the operating system, is common across the adapters, and therefore, is considered
30 device-independent.

The shortcomings of the prior art are overcome and advantages are provided through the provision of a computer program product for enabling adapters in a computing environment. The computer program product includes a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method. The method includes, for instance, responsive to executing a Call Logical Processor (CLP) instruction for enabling an adapter, the CLP instruction comprising a function handle identifying the adapter and having an adapter not enabled indicator, the CLP instruction requesting a number of DMA address spaces, the execution enabling one or more DMA address spaces comprising a) and b): a) enabling the adapter, wherein the enabling includes enabling registration for address translation and interruptions for supporting direct memory accesses and message signaled interruptions for the adapter; and b) returning the function handle having an adapter enabled indicator.

Methods and systems relating to one or more aspects of the present invention are also described and claimed herein.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

A preferred embodiment of the present invention will now be described, by way of example only, with reference to the accompanying drawings, in which:

FIG. 1A depicts one embodiment of a computing environment to incorporate and use one or more aspects of the present invention;

FIG. 1B depicts another embodiment of a computing environment to incorporate and use one or more aspects of the present invention;

FIG. 2 depicts one embodiment of further details of the system memory and I/O hub of FIGs. 1A and 1B, in accordance with an aspect of the present invention;

FIG. 3A depicts one example of a function table entry used in accordance with an aspect of the present invention;

FIG. 3B depicts one embodiment of a function handle used in accordance with an aspect of the present invention;

FIG. 4A depicts one embodiment of a Call Logical Processor instruction used in accordance with an aspect of the present invention;

FIG. 4B depicts one embodiment of a request block used by the Call Logical Processor instruction of FIG. 4A, in accordance with an aspect of the present invention;

FIG. 4C depicts one embodiment of a response block provided by the Call Logical Processor instruction of FIG. 4A, in accordance with an aspect of the present invention;

FIG. 5 depicts one embodiment of the logic to enable a PCI function, in accordance with an aspect of the present invention;

FIG. 6 depicts one embodiment of the logic to disable a PCI function, in accordance with an aspect of the present invention;

FIG. 7 depicts one embodiment of a computer program product incorporating one or more aspects of the present invention;

FIG. 8 depicts one embodiment of a host computer system to incorporate and use one or more aspects of the present invention;

FIG. 9 depicts a further example of a computer system to incorporate and use one or more aspects of the present invention;

FIG. 10 depicts another example of a computer system comprising a computer network to incorporate and use one or more aspects of the present invention;

FIG. 11 depicts one embodiment of various elements of a computer system to incorporate and use one or more aspects of the present invention;

FIG. 12A depicts one embodiment of the execution unit of the computer system of FIG. 11 to incorporate and use one or more aspects of the present invention;

FIG. 12B depicts one embodiment of the branch unit of the computer system of FIG. 11 to incorporate and use one or more aspects of the present invention;

FIG. 12C depicts one embodiment of the load/store unit of the computer system of FIG. 11 to incorporate and use one or more aspects of the present invention; and

FIG. 13 depicts one embodiment of an emulated host computer system to incorporate and use one or more aspects of the present invention.

DETAILED DESCRIPTION

In accordance with an aspect of the present invention, a capability is provided for enabling/disabling adapters of a computing environment. The capability is device-independent from the standpoint of an operating system. That is, the operating system performs the same logic regardless of the type of adapter.

As used herein, firmware includes, e.g., the microcode, millicode, and macrocode of the processor. It includes, for instance, the hardware-level instructions and/or data structures used in implementation of higher level machine code. In one embodiment, it includes, for instance, proprietary code that is typically delivered as microcode that includes trusted software or microcode specific to the underlying hardware and controls operating system access to the system hardware.

Further, the term adapter includes any type of adapter (e.g., storage adapter, processing adapter, PCI adapter, other type of input/output adapters, etc.). Moreover, in the examples presented herein, adapter is used interchangeably with adapter function (e.g., PCI function). In one embodiment, an adapter includes one adapter function. However, in other
5 embodiments, an adapter may include a plurality of adapter functions. One or more aspects of the present invention are applicable whether an adapter includes one adapter function or a plurality of adapter functions. In one embodiment, if an adapter includes a plurality of adapter functions, then each function may be enabled/disabled in accordance with an aspect of the present invention.

10 One embodiment of a computing environment to incorporate and use one or more aspects of the present invention is described with reference to FIG. 1A. In one example, a computing environment 100 is a System z® server offered by International Business Machines Corporation. System z® is based on the z/Architecture® offered by International Business
15 Machines Corporation. Details regarding the z/Architecture® are described in an IBM® publication entitled, "z/Architecture Principles of Operation," IBM Publication No. SA22-7832-07, February 2009. IBM®, System z® and z/Architecture® are registered trademarks of International Business Machines Corporation, Armonk, New York. Other names used herein may be registered trademarks, trademarks or product names of International Business
20 Machines Corporation or other companies.

In one example, computing environment 100 includes one or more central processing units (CPUs) 102 coupled to a system memory 104 (a.k.a., main memory) via a memory controller 106. To access system memory 104, a central processing unit 102 issues a read or write
25 request that includes an address used to access system memory. The address included in the request is typically not directly usable to access system memory, and therefore, it is translated to an address that is directly usable in accessing system memory. The address is translated via a translation mechanism (XLATE) 108. For example, the address is translated from a virtual address to a real or absolute address using, for instance, dynamic address
30 translation (DAT).

The request, including the address (translated, if necessary), is received by memory controller 106. In one example, memory controller 106 is comprised of hardware and is used to arbitrate for access to the system memory and to maintain the memory's consistency. This arbitration is performed for requests received from CPUs 102, as well as for requests
5 received from one or more adapters 110. Like the central processing units, the adapters issue requests to system memory 104 to gain access to the system memory.

In one example, adapter 110 is a Peripheral Component Interconnect (PCI) or PCI express (PCIe) adapter that includes one or more PCI functions. A PCI function issues a request that
10 is routed to an input/output hub 112 (e.g., a PCI hub) via one or more switches (e.g., PCIe switches) 114. In one example, the input/output hub is comprised of hardware, including one or more state machines, and is coupled to memory controller 106 via an I/O-to-memory bus 120.

The input/output hub includes, for instance, a root complex 116 that receives the request from a switch. The request includes an input/output address that is provided to an address translation and protection unit 118 which accesses information used for the request. As
15 examples, the request may include an input/output address used to perform a direct memory access (DMA) operation or to request a message signaled interruption (MSI). Address translation and protection unit 118 accesses information used for the DMA or MSI request.
20 As a particular example, for a DMA operation, information may be obtained to translate the address. The translated address is then forwarded to the memory controller to access system memory.

In a further embodiment of a computing environment, in addition to or instead of one or more CPUs 102, a central processing complex is coupled to memory controller 106 as shown in FIG. 1B. In this example, a central processing complex 150 includes, for instance, one or more partitions or zones 152 (e.g., logical partitions LP1-LPn), one or more central
25 processors (e.g., CP1-CPm) 154, and a hypervisor 156 (e.g., a logical partition manager),
30 each of which is described below.

Each logical partition 152 is capable of functioning as a separate system. That is, each logical partition can be independently reset, initially loaded with an operating system or a hypervisor (such as z/VM® offered by International Business Machines Corporation, Armonk, New York), if desired, and operate with different programs. An operating system, a hypervisor, or an application program running in a logical partition appears to have access to a full and complete system, but only a portion of it is available. A combination of hardware and Licensed Internal Code (also referred to as microcode or millicode) keeps a program in a logical partition from interfering with the program in a different logical partition. This allows several different logical partitions to operate on a single or multiple physical processor in a time slice manner. In this particular example, each logical partition has a resident operating system 158, which may differ for one or more logical partitions. In one embodiment, operating system 158 is a z/OS® or zLinux operating system, offered by International Business Machines Corporation, Armonk, New York. z/OS® and z/VM® are registered trademarks of International Business Machines Corporation, Armonk, New York.

Central processors 154 are physical processor resources that are allocated to the logical partitions. For instance, a logical partition 152 includes one or more logical processors, each of which represents all or a share of the physical processor resource 154 allocated to the partition. The underlying processor resource may either be dedicated to that partition or shared with another partition.

Logical partitions 152 are managed by hypervisor 156 implemented by firmware running on processors 154. Logical partitions 152 and hypervisor 156 each comprise one or more programs residing in respective portions of central storage associated with the central processors. One example of hypervisor 156 is the Processor Resource/Systems Manager (PR/SM), offered by International Business Machines Corporation, Armonk, New York.

Although, in this example, a central processing complex having logical partitions is described, one or more aspects of the present invention may be incorporated in and used by other processing units, including single or multi-processor processing units that are not partitioned, among others. The central processing complex described herein is only one example.

Further details regarding system memory and the input/output hub are described with reference to FIG. 2. In this example, the memory controller is not shown, but may be used. The I/O hub may be coupled to system memory 104 and/or processor 204 directly or via a memory controller.

5

Referring to FIG. 2, in one example, system memory 104 includes one or more address spaces 200. An address space is a particular portion of system memory that has been assigned to a particular component of the computing environment, such as a particular adapter. In one example, the address space is accessible by direct memory access (DMA) initiated by the adapter, and therefore, an address space is referred to in the examples herein as a DMA address space. However, in other examples, direct memory access is not used to access the address space.

10

15

In one example, there is an operating system 202 executing within a processor 204 (e.g., CPU 102 or a CP 154 assigned to an LP 152) that assigns a DMA address space to a particular adapter. This assignment is performed via a registration process, which causes an initialization (via, e.g., trusted software) of a device table entry 210 for that adapter. There is one device table entry per assigned address space and this device table entry is associated with a single adapter. The device table entry is located in a device table 212 located in I/O hub 112. For example, device table 212 is located within the address translation and protection unit of the I/O hub.

20

25

In one example, device table entry 210 includes information usable in providing various services for the adapter. For example, the device table entry includes an enable indicator 214 that indicates whether the device table entry is enabled for a particular adapter. The device table entry may include more, less or different information for the enable/disable operations, as well as for other provided services, such as address translation, interruption handling, etc.

30

In one embodiment, the device table entry to be used by a particular adapter that issues a request is located using a requestor identifier (RID) (and/or a portion of the address) located in a request issued by a PCI function 220 associated with an adapter. The requestor id (e.g.,

a 16-bit value specifying, for instance, a bus number, device number and function number) is included in the request as well as an I/O address to be used. The request, including the RID and the I/O address, are provided to, e.g., a contents addressable memory (CAM) 230 via, e.g., a switch 114, which is used to provide an index value. For instance, the CAM includes multiple entries, with each entry corresponding to an index into the device table. Each CAM entry includes the value of a RID. If, for instance, the received RID matches the value contained in an entry in the CAM, a corresponding device table index is used to locate the device table entry. That is, the output of the CAM is used to index into device table 212 to locate device table entry 210. If there is no match, the received packet is discarded. (In other embodiments, a CAM or other lookup is not needed and the RID is used as the index.)

In addition to a device table entry, another data structure is also associated with an adapter, which includes information regarding the adapter. In the particular examples described herein, the adapter is a PCI function, and therefore, the data structure is referred to as a function table entry (FTE). Although the examples herein refer to PCI functions, in other embodiments, other adapter functions or adapters may be enabled/disabled, in accordance with an aspect of the present invention.

As shown in FIG. 3A, in one example, a function table entry 300 is an entry in a function table 302 stored, for instance, in secure memory. Each function table entry 300 includes information to be used in processing associated with its adapter. In one example, function table entry 300 includes an instance number 308 indicating a particular instance of the adapter function associated with the function table entry; one or more device table entry indices 310, each of which is used as an index into the device table to locate its corresponding device table entry (a PCI function may have a plurality of address spaces assigned thereto, and therefore, a plurality of DTEs); a busy indicator 312 that indicates whether the PCI function is busy; a permanent error state indicator 314 that indicates whether the function is in a permanent error state; a recovery initiated indicator 316 that indicates whether recovery has been initiated for the function; a permission indicator 318 that indicates whether the operating system trying to enable the PCI function has authority to do so; and an enable indicator 320 indicating whether the function is enabled (e.g., 1=enabled, 0=disabled).

In one example, the busy indicator, permanent error state indicator, and recovery initiated indicator are set based on monitoring performed by the firmware. Further, the permission indicator is set, for instance, based on policy. In other embodiments, the function table entry may include more, less or different information.

In one embodiment, to locate a function table entry in a function table that includes one or more entries, a function identifier, such as a function handle, is used. For instance, one or more bits of the function handle are used as an index into the function table to locate a particular function table entry.

Referring to FIG. 3B, additional details regarding a function handle are described. In one example, a function handle 350 includes an enable indicator 352 that indicates whether the PCI function handle is enabled; a PCI function number 354 that identifies the function (this is a static identifier) and, in one embodiment, is an index into the function table; and an instance number 356 which indicates the particular instance of this function handle. For instance, each time the function is enabled, the instance number is incremented to provide a new instance number.

In order to use a PCI function, it is to be enabled. For instance, the operating system that would like to use a PCI function performs a query to determine the one or more functions that it is eligible to use (based on I/O configuration), and selects one of those functions to be enabled. In one example, the function is enabled using a set PCI function command of a Call Logical Processor instruction. One embodiment of this instruction is depicted in FIG. 4A. As shown, in one example, a Call Logical Processor instruction 400 includes an operation code 402 indicating that it is the Call Logical Processor instruction; and an indication for a command 404. In one example, this indication is an address of a request block that describes the command to be performed. One embodiment of such a request block is depicted in FIG. 4B.

As shown in FIG. 4B, in one example, a request block 420 includes a number of parameters, such as, for instance, a length field 422 indicating the length of the request block; a command field 424 indicating the set PCI function command; a PCI function handle 426,

which is the handle to be provided to either the enable or disable function; an operation code 428, which is used to designate either an enable or disable operation; and a number of DMA address spaces (DMAAS) 430, which indicates the requested number of address spaces to be associated with the particular PCI function. More, less or different information may be included in other embodiments.

For instance, in a virtual environment in which the instruction is issued by a host of a pageable storage mode guest, a guest identity is provided. Other variations are also possible. In one example, in the z/Architecture®, a pageable guest is interpretively executed via the Start Interpretive Execution (SIE) instruction, at level 2 of interpretation. For instance, the logical partition (LPAR) hypervisor executes the SIE instruction to begin the logical partition in physical, fixed memory. If z/VM® is the operating system in that logical partition, it issues the SIE instruction to execute its guests (virtual) machines in its V=V (virtual) storage. Therefore, the LPAR hypervisor uses level-1 SIE, and the z/VM® hypervisor uses level-2 SIE.

Responsive to issuing and processing the Call Logical Processor instruction, a response block is returned and the information included in the response block is dependent on the operation to be performed. One embodiment of the response block is depicted in FIG. 4C. In one example, response block 450 includes a length field 452 indicating the length of the response block; a response code 454 indicating a status of the command; and a PCI function handle 456 that identifies the PCI function. Responsive to the enable command, the PCI function handle is an enabled handle of the PCI function. Further, upon completion of the disable operation, the PCI function handle is a general handle that can be enabled by an enable function in the future.

One embodiment of the logic to enable a PCI function is described with reference to FIG. 5. In one example, this logic is initiated responsive to issuing a Call Logical Processor instruction in which the command is set to the set PCI function command and the operation code is set to the enable function. This logic is performed by, for instance, a processor responsive to the operating system or a device driver of the operating system authorized to

perform this logic issuing the instruction. In other embodiments, the logic may be performed without the use of the Call Logical Processor instruction.

Referring to FIG. 5, initially, a determination is made as to whether a handle provided in the request block of the Call Logical Processor instruction is a valid handle, INQUIRY 500. That is, does the handle point to a valid entry in the function table or is it outside the range of valid entries (e.g., does function number portion of handle designate an installed function). If the handle is not known, then a corresponding response code is provided indicating that the handle is not recognized. However, if the handle is known, then a further inquiry is made as to whether the handle is enabled, INQUIRY 504. This determination is made by checking the enable indicator in the PCI function handle. If the indication is set indicating the handle is enabled, then a response code is returned indicating such, STEP 506.

However, if the handle is known and not enabled (i.e., valid for enablement), then a further determination is made as to whether the requested number of address spaces to be assigned to the PCI function is greater than a maximum value, INQUIRY 508. To make this determination, the number of DMA address spaces as specified in the request block is compared against a maximum value (provided based on policy, in one example). If the number of address spaces is greater than the maximum value, then a response code is provided indicating an invalid value for DMA address spaces, STEP 510. Otherwise, a determination is made as to whether the number of requested address spaces is available, INQUIRY 512. This determination is made by checking whether there are device table entries available for the requested number of address spaces. If the number of requested address spaces is not available, then a response code is returned indicating that there are insufficient resources, STEP 514. Otherwise, processing continues to enable the PCI function.

The provided handle is used to locate a function table entry, STEP 516. For instance, one or more designated bits of the handle are used as an index into the function table to locate a particular function table entry. Responsive to locating the appropriate function table entry, a determination is made as to whether the function is enabled, INQUIRY 518. This determination is made by checking the enable indicator in the function table entry. If the

function is already enabled (i.e., the indicator is set to one), then a response code is returned indicating that the PCI function is already in the requested state, STEP 520.

5 If the function is not already enabled, then processing continues with determining whether the function is in a permanent error state, INQUIRY 522. If the permanent error state indicator in the function table entry indicates it is in a permanent error state, then a response code is returned indicating such, STEP 524. However, if the function is not in a permanent error state, a further determination is made as to whether error recovery has been initiated for the function, INQUIRY 526. If the recovery initiated indicator in the function table entry is
10 set, then a response code indicating recovery has been initiated is provided, STEP 528.

Otherwise, a further inquiry is made as to whether the PCI function is busy, INQUIRY 530. Again, if a check of the busy indicator in the function table entry indicates the PCI function is busy, then such an indication is provided, STEP 532. However, if the PCI function is not in the permanent error state, recovery is not initiated and it is not busy, then a further inquiry
15 is made as to whether the operating system is permitted to enable this PCI function, STEP 534. If it is not permitted based on the permission indicator of the function table entry, then a response code indicating an unauthorized action is provided, STEP 536. However, if all the tests are successfully passed, then a further determination is made as to whether there are any DTEs available for this PCI function, INQUIRY 538. As examples, the determination
20 of DTEs being available can be based on the DTEs that are not currently enabled in the I/O hub. Additionally, policy could be applied to further limit the number of DTEs available to a given operating system or logical partition. Any available DTE that is accessible to the adapter may be assigned. If there are no available DTEs, then a response code is returned indicating that one or more of the requested DTEs are unavailable, STEP 540.

25 If the DTEs are available, then a number of DTEs corresponding to the requested number of address spaces are assigned and enabled, STEP 542. In one example, the enabling includes setting the enable indicator in each DTE to be enabled. Further, the enabling includes, in this example, setting up the CAM to provide an index to each DTE. For instance, for each
30 DTE, an entry in the CAM is loaded with the index.

Further, the DTEs are associated with the function table entry, STEP 544. This includes, for instance, including each DTE index in the function table entry. The function is then marked as enabled by setting the enable indicator in the function table entry, STEP 546. Moreover, the enable bit in the handle is set, and the instance number is updated, STEP 548. This
5 enabled handle is then returned, STEP 550, allowing use of the PCI adapter. For instance, responsive to enabling the function, registration for address translations and interruptions may be performed, DMA operations may be performed by the PCI function; interruptions may be requested by the function; and/or load, store, store block and/or modify function controls instructions (e.g., PCI Load, PCI Store, PCI Store Block, Modify PCI Function
10 Controls) may be issued to the function.

One embodiment of the logic to disable a PCI function is described with reference to FIG. 6. In this example, the set PCI function command is requested via a Call Logical Processor instruction in which the operation code is set to disable; however, in other embodiments,
15 such an instruction is not used. In one example, it is the operating system or a device driver of the operating system that performs this logic.

Referring to FIG. 6, initially, a determination is made as to whether the handle provided in the request block for the Call Logical Processor instruction is a known handle, INQUIRY
20 600. For instance, a check is made as to whether the handle points to a valid entry in the function table. If the handle points to a valid entry, then the handle is a known handle. If not, then a response code indicating an unknown handle is provided, STEP 602. However, if the handle is known, then a further determination is made as to whether the handle is already disabled, INQUIRY 604. If the enable indicator in the handle indicates that handle is
25 already disabled, then a response code indicating such is provided. Otherwise, if the handle is known and enabled, the handle is valid for a disable operation and is used to locate the function table entry, STEP 608.

Responsive to obtaining the function table entry, a determination is made as to whether the
30 function is already disabled as indicated by the enable indicator in the function table entry, STEP 610. If the indicator is not set (i.e., the enable indicator = 0), then a response code is provided indicating that the function is already disabled, STEP 612.

If the indicator is set (e.g., enable = 1), then a determination is made as to whether the function is in the permanent error state, INQUIRY 614. If it is in a permanent error state, then a response code indicating error is provided, STEP 616. Otherwise, a determination is made as to whether error recovery is initiated, INQUIRY 618. If error recovery is initiated, then a response code indicating such is provided, STEP 620. If error recovery is not initiated, then a determination is made as to whether the PCI function is busy, INQUIRY 622. If it is busy, then a response code providing this is provided, STEP 624. Otherwise, a determination is made as to whether the operating system is authorized to issue this disable command, STEP 626. This determination is made by, for instance, checking the permission indicator in the function table entry, as well as comparing the instance number in the handle with the instance number in the function table entry. If they are unequal, then a request is being made to disable a different instance of the function that was enabled. If the permission indicator indicates unpermitted or the instance numbers are unequal, the operating system is not authorized and a response code indicating unauthorization is provided, STEP 628. However, if the permission indicator specifies permitted and the instance numbers are equal, the operating system is authorized.

If all the checks are successful, then the function is disabled, STEP 630. In one example, this includes setting the enable indicator in the function table entry to zero (or otherwise to an off state). Thereafter, the registration parameters in the DTEs associated with this PCI function are cleared, STEP 632, and those DTEs are released to be used by other PCI functions, STEP 634. For instance, the enable bit in the DTE is cleared and the CAM entry associated with the DTE is removed. Further, the enable indicator in the handle is reset to zero (or some other value indicating disabled or off), STEP 636, and the disabled handle is returned, STEP 638.

In a further embodiment, if one or more of the tests at INQUIRIES 614, 618 and 622 fail, then the disable still continues and a response code indicating such may be provided.

Described in detail above is a capability for enabling/disabling a PCI function. This capability is device-independent from the standpoint of the operating system, and provides a fine granularity of control, in which an operating system is able to enable and disable a PCI

function. Responsive to disabling the function, another operating system may enable the function. This allows multiple operating systems (e.g., in a logically partitioned environment) to share adapter functions.

5 In the embodiments described herein, the adapters are PCI adapters. PCI, as used herein, refers to any adapters implemented according to a PCI-based specification as defined by the Peripheral Component Interconnect Special Interest Group (PCI-SIG), including but not limited to, PCI or PCIe. In one particular example, the Peripheral Component Interconnect Express (PCIe) is a component level interconnect standard that defines a bi-directional
10 communication protocol for transactions between I/O adapters and host systems. PCIe communications are encapsulated in packets according to the PCIe standard for transmission on a PCIe bus. Transactions originating at I/O adapters and ending at host systems are referred to as upbound transactions. Transactions originating at host systems and terminating at I/O adapters are referred to as downbound transactions. The PCIe topology is
15 based on point-to-point unidirectional links that are paired (e.g., one upbound link, one downbound link) to form the PCIe bus. The PCIe standard is maintained and published by the PCI-SIG, as noted above in the Background section.

As will be appreciated by one skilled in the art, aspects of the present invention may be
20 embodied as a system, method or computer program product. Accordingly, aspects of the present invention may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module" or "system". Furthermore, aspects of the present invention
25 may take the form of a computer program product embodied in one or more computer readable medium(s) having computer readable program code embodied thereon.

Any combination of one or more computer readable medium(s) may be utilized. The computer readable medium may be a computer readable storage medium. A computer
30 readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, electromagnetic, infrared or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the

computer readable storage medium include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), an optical fiber, a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain or store a program for use by or in connection with an instruction execution system, apparatus, or device.

Referring now to FIG. 7, in one example, a computer program product 700 includes, for instance, one or more computer readable storage media 702 to store computer readable program code means or logic 704 thereon to provide and facilitate one or more aspects of the present invention.

Program code embodied on a computer readable medium may be transmitted using an appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

Computer program code for carrying out operations for aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language, such as Java, Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language, assembler or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products

according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

These computer program instructions may also be stored in a computer readable medium that can direct a computer, other programmable data processing apparatus, or other devices to function in a particular manner, such that the instructions stored in the computer readable medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

The flowchart and block diagrams in the figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function(s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and

combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

5 In addition to the above, one or more aspects of the present invention may be provided, offered, deployed, managed, serviced, etc. by a service provider who offers management of customer environments. For instance, the service provider can create, maintain, support, etc. computer code and/or a computer infrastructure that performs one or more aspects of the present invention for one or more customers. In return, the service provider may receive
10 payment from the customer under a subscription and/or fee agreement, as examples. Additionally or alternatively, the service provider may receive payment from the sale of advertising content to one or more third parties.

In one aspect of the present invention, an application may be deployed for performing one or
15 more aspects of the present invention. As one example, the deploying of an application comprises providing computer infrastructure operable to perform one or more aspects of the present invention.

As a further aspect of the present invention, a computing infrastructure may be deployed
20 comprising integrating computer readable code into a computing system, in which the code in combination with the computing system is capable of performing one or more aspects of the present invention.

As yet a further aspect of the present invention, a process for integrating computing
25 infrastructure comprising integrating computer readable code into a computer system may be provided. The computer system comprises a computer readable medium, in which the computer medium comprises one or more aspects of the present invention. The code in combination with the computer system is capable of performing one or more aspects of the present invention.

30 Although various embodiments are described above, these are only examples. For example, computing environments of other architectures can incorporate and use one or more aspects

of the present invention. As examples, servers other than System z® servers, such as Power Systems servers or other servers offered by International Business Machines Corporation, or servers of other companies can include, use and/or benefit from one or more aspects of the present invention. Further, although in the example herein, the adapters and PCI hub are considered a part of the server, in other embodiments, they do not have to necessarily be considered a part of the server, but can simply be considered as being coupled to system memory and/or other components of a computing environment. The computing environment need not be a server. Further, although tables are described, any data structure can be used and the term table is to include all such data structures. Yet further, although the adapters are PCI based, one or more aspects of the present invention are usable with other adapters or other I/O components. Adapter and PCI adapter are just examples. Moreover, the FTE or the parameters of the FTE can be located and maintained in other than secure memory, including, for instance, in hardware (e.g., PCI function hardware). The DTE, FTE and/or handle may include more, less or different information, as well as the request and/or response block. Additionally, the Call Logical Processor instruction may include more, less or different fields. Many other variations are possible.

Further, other types of computing environments can benefit from one or more aspects of the present invention. As an example, a data processing system suitable for storing and/or executing program code is usable that includes at least two processors coupled directly or indirectly to memory elements through a system bus. The memory elements include, for instance, local memory employed during actual execution of the program code, bulk storage, and cache memory which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

Input/Output or I/O devices (including, but not limited to, keyboards, displays, pointing devices, DASD, tape, CDs, DVDs, thumb drives and other memory media, etc.) can be coupled to the system either directly or through intervening I/O controllers. Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the available types of network adapters.

Referring to FIG. 8, representative components of a Host Computer system 5000 to implement one or more aspects of the present invention are portrayed. The representative host computer 5000 comprises one or more CPUs 5001 in communication with computer memory (i.e., central storage) 5002, as well as I/O interfaces to storage media devices 5011 and networks 5010 for communicating with other computers or SANs and the like. The CPU 5001 is compliant with an architecture having an architected instruction set and architected functionality. The CPU 5001 may have dynamic address translation (DAT) 5003 for transforming program addresses (virtual addresses) into real addresses of memory. A DAT typically includes a translation lookaside buffer (TLB) 5007 for caching translations so that later accesses to the block of computer memory 5002 do not require the delay of address translation. Typically, a cache 5009 is employed between computer memory 5002 and the processor 5001. The cache 5009 may be hierarchical having a large cache available to more than one CPU and smaller, faster (lower level) caches between the large cache and each CPU. In some implementations, the lower level caches are split to provide separate low level caches for instruction fetching and data accesses. In one embodiment, an instruction is fetched from memory 5002 by an instruction fetch unit 5004 via a cache 5009. The instruction is decoded in an instruction decode unit 5006 and dispatched (with other instructions in some embodiments) to instruction execution unit or units 5008. Typically several execution units 5008 are employed, for example an arithmetic execution unit, a floating point execution unit and a branch instruction execution unit. The instruction is executed by the execution unit, accessing operands from instruction specified registers or memory as needed. If an operand is to be accessed (loaded or stored) from memory 5002, a load/store unit 5005 typically handles the access under control of the instruction being executed. Instructions may be executed in hardware circuits or in internal microcode (firmware) or by a combination of both.

As noted, a computer system includes information in local (or main) storage, as well as addressing, protection, and reference and change recording. Some aspects of addressing include the format of addresses, the concept of address spaces, the various types of addresses, and the manner in which one type of address is translated to another type of address. Some of main storage includes permanently assigned storage locations. Main storage provides the system with directly addressable fast-access storage of data. Both data

and programs are to be loaded into main storage (from input devices) before they can be processed.

5 Main storage may include one or more smaller, faster-access buffer storages, sometimes called caches. A cache is typically physically associated with a CPU or an I/O processor. The effects, except on performance, of the physical construction and use of distinct storage media are generally not observable by the program.

10 Separate caches may be maintained for instructions and for data operands. Information within a cache is maintained in contiguous bytes on an integral boundary called a cache block or cache line (or line, for short). A model may provide an EXTRACT CACHE ATTRIBUTE instruction which returns the size of a cache line in bytes. A model may also provide PREFETCH DATA and PREFETCH DATA RELATIVE LONG instructions which effects the prefetching of storage into the data or instruction cache or the releasing of data
15 from the cache.

Storage is viewed as a long horizontal string of bits. For most operations, accesses to storage proceed in a left-to-right sequence. The string of bits is subdivided into units of eight bits. An eight-bit unit is called a byte, which is the basic building block of all information
20 formats. Each byte location in storage is identified by a unique nonnegative integer, which is the address of that byte location or, simply, the byte address. Adjacent byte locations have consecutive addresses, starting with 0 on the left and proceeding in a left-to-right sequence. Addresses are unsigned binary integers and are 24, 31, or 64 bits.

25 Information is transmitted between storage and a CPU or a channel subsystem one byte, or a group of bytes, at a time. Unless otherwise specified, in, for instance, the z/Architecture®, a group of bytes in storage is addressed by the leftmost byte of the group. The number of bytes in the group is either implied or explicitly specified by the operation to be performed. When used in a CPU operation, a group of bytes is called a field. Within each group of
30 bytes, in, for instance, the z/Architecture®, bits are numbered in a left-to-right sequence. In the z/Architecture®, the leftmost bits are sometimes referred to as the “high-order” bits and the rightmost bits as the “low-order” bits. Bit numbers are not storage addresses, however.

Only bytes can be addressed. To operate on individual bits of a byte in storage, the entire byte is accessed. The bits in a byte are numbered 0 through 7, from left to right (in, e.g., the z/Architecture®). The bits in an address may be numbered 8-31 or 40-63 for 24-bit addresses, or 1-31 or 33-63 for 31-bit addresses; they are numbered 0-63 for 64-bit addresses. Within any other fixed-length format of multiple bytes, the bits making up the format are consecutively numbered starting from 0. For purposes of error detection, and in preferably for correction, one or more check bits may be transmitted with each byte or with a group of bytes. Such check bits are generated automatically by the machine and cannot be directly controlled by the program. Storage capacities are expressed in number of bytes.

When the length of a storage-operand field is implied by the operation code of an instruction, the field is said to have a fixed length, which can be one, two, four, eight, or sixteen bytes. Larger fields may be implied for some instructions. When the length of a storage-operand field is not implied but is stated explicitly, the field is said to have a variable length. Variable-length operands can vary in length by increments of one byte (or with some instructions, in multiples of two bytes or other multiples). When information is placed in storage, the contents of only those byte locations are replaced that are included in the designated field, even though the width of the physical path to storage may be greater than the length of the field being stored.

Certain units of information are to be on an integral boundary in storage. A boundary is called integral for a unit of information when its storage address is a multiple of the length of the unit in bytes. Special names are given to fields of 2, 4, 8, and 16 bytes on an integral boundary. A halfword is a group of two consecutive bytes on a two-byte boundary and is the basic building block of instructions. A word is a group of four consecutive bytes on a four-byte boundary. A doubleword is a group of eight consecutive bytes on an eight-byte boundary. A quadword is a group of 16 consecutive bytes on a 16-byte boundary. When storage addresses designate halfwords, words, doublewords, and quadwords, the binary representation of the address contains one, two, three, or four rightmost zero bits, respectively. Instructions are to be on two-byte integral boundaries. The storage operands of most instructions do not have boundary-alignment requirements.

On devices that implement separate caches for instructions and data operands, a significant delay may be experienced if the program stores into a cache line from which instructions are subsequently fetched, regardless of whether the store alters the instructions that are subsequently fetched.

5

In one embodiment, the invention may be practiced by software (sometimes referred to licensed internal code, firmware, micro-code, milli-code, pico-code and the like, any of which would be consistent with the present invention). Referring to FIG. 8, software program code which embodies the present invention is typically accessed by processor 5001
10 of the host system 5000 from long-term storage media devices 5011, such as a CD-ROM drive, tape drive or hard drive. The software program code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users from computer memory 5002 or storage of one computer system over a network 5010 to
15 other computer systems for use by users of such other systems.

The software program code includes an operating system which controls the function and interaction of the various computer components and one or more application programs. Program code is normally paged from storage media device 5011 to the relatively higher-
20 speed computer storage 5002 where it is available for processing by processor 5001. The techniques and methods for embodying software program code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs),
25 DVDs, Magnetic Tape and the like is often referred to as a "computer program product". The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

FIG. 9 illustrates a representative workstation or server hardware system in which the
30 present invention may be practiced. The system 5020 of FIG. 9 comprises a representative base computer system 5021, such as a personal computer, a workstation or a server, including optional peripheral devices. The base computer system 5021 includes one or more

processors 5026 and a bus employed to connect and enable communication between the processor(s) 5026 and the other components of the system 5021 in accordance with known techniques. The bus connects the processor 5026 to memory 5025 and long-term storage 5027 which can include a hard drive (including any of magnetic media, CD, DVD and Flash Memory for example) or a tape drive for example. The system 5021 might also include a user interface adapter, which connects the microprocessor 5026 via the bus to one or more interface devices, such as a keyboard 5024, a mouse 5023, a printer/scanner 5030 and/or other interface devices, which can be any user interface device, such as a touch sensitive screen, digitized entry pad, etc. The bus also connects a display device 5022, such as an LCD screen or monitor, to the microprocessor 5026 via a display adapter.

The system 5021 may communicate with other computers or networks of computers by way of a network adapter capable of communicating 5028 with a network 5029. Example network adapters are communications channels, token ring, Ethernet or modems.

Alternatively, the system 5021 may communicate using a wireless interface, such as a CDPD (cellular digital packet data) card. The system 5021 may be associated with such other computers in a Local Area Network (LAN) or a Wide Area Network (WAN), or the system 5021 can be a client in a client/server arrangement with another computer, etc. All of these configurations, as well as the appropriate communications hardware and software, are known in the art.

FIG. 10 illustrates a data processing network 5040 in which the present invention may be practiced. The data processing network 5040 may include a plurality of individual networks, such as a wireless network and a wired network, each of which may include a plurality of individual workstations 5041, 5042, 5043, 5044. Additionally, as those skilled in the art will appreciate, one or more LANs may be included, where a LAN may comprise a plurality of intelligent workstations coupled to a host processor.

Still referring to FIG. 10, the networks may also include mainframe computers or servers, such as a gateway computer (client server 5046) or application server (remote server 5048 which may access a data repository and may also be accessed directly from a workstation 5045). A gateway computer 5046 serves as a point of entry into each individual network. A

gateway is needed when connecting one networking protocol to another. The gateway 5046 may be preferably coupled to another network (the Internet 5047 for example) by means of a communications link. The gateway 5046 may also be directly coupled to one or more workstations 5041, 5042, 5043, 5044 using a communications link. The gateway computer
5 may be implemented utilizing an IBM eServer™ System z® server available from International Business Machines Corporation.

Referring concurrently to FIG. 9 and FIG. 10, software programming code which may embody the present invention may be accessed by the processor 5026 of the system 5020
10 from long-term storage media 5027, such as a CD-ROM drive or hard drive. The software programming code may be embodied on any of a variety of known media for use with a data processing system, such as a diskette, hard drive, or CD-ROM. The code may be distributed on such media, or may be distributed to users 5050, 5051 from the memory or storage of one computer system over a network to other computer systems for use by users of such other
15 systems.

Alternatively, the programming code may be embodied in the memory 5025, and accessed by the processor 5026 using the processor bus. Such programming code includes an operating system which controls the function and interaction of the various computer
20 components and one or more application programs 5032. Program code is normally paged from storage media 5027 to high-speed memory 5025 where it is available for processing by the processor 5026. The techniques and methods for embodying software programming code in memory, on physical media, and/or distributing software code via networks are well known and will not be further discussed herein. Program code, when created and stored on a
25 tangible medium (including but not limited to electronic memory modules (RAM), flash memory, Compact Discs (CDs), DVDs, Magnetic Tape and the like is often referred to as a “computer program product”. The computer program product medium is typically readable by a processing circuit preferably in a computer system for execution by the processing circuit.

30 The cache that is most readily available to the processor (normally faster and smaller than other caches of the processor) is the lowest (L1 or level one) cache and main store (main

memory) is the highest level cache (L3 if there are 3 levels). The lowest level cache is often divided into an instruction cache (I-Cache) holding machine instructions to be executed and a data cache (D-Cache) holding data operands.

5 Referring to FIG. 11, an exemplary processor embodiment is depicted for processor 5026. Typically one or more levels of cache 5053 are employed to buffer memory blocks in order to improve processor performance. The cache 5053 is a high speed buffer holding cache lines of memory data that are likely to be used. Typical cache lines are 64, 128 or 256 bytes of memory data. Separate caches are often employed for caching instructions than for
10 caching data. Cache coherence (synchronization of copies of lines in memory and the caches) is often provided by various “snoop” algorithms well known in the art. Main memory storage 5025 of a processor system is often referred to as a cache. In a processor system having 4 levels of cache 5053, main storage 5025 is sometimes referred to as the level 5 (L5) cache since it is typically faster and only holds a portion of the non-volatile
15 storage (DASD, tape etc) that is available to a computer system. Main storage 5025 “caches” pages of data paged in and out of the main storage 5025 by the operating system.

A program counter (instruction counter) 5061 keeps track of the address of the current instruction to be executed. A program counter in a z/Architecture® processor is 64 bits and
20 can be truncated to 31 or 24 bits to support prior addressing limits. A program counter is typically embodied in a PSW (program status word) of a computer such that it persists during context switching. Thus, a program in progress, having a program counter value, may be interrupted by, for example, the operating system (context switch from the program environment to the operating system environment). The PSW of the program maintains the
25 program counter value while the program is not active, and the program counter (in the PSW) of the operating system is used while the operating system is executing. Typically, the program counter is incremented by an amount equal to the number of bytes of the current instruction. RISC (Reduced Instruction Set Computing) instructions are typically fixed length while CISC (Complex Instruction Set Computing) instructions are typically variable
30 length. Instructions of the IBM z/Architecture® are CISC instructions having a length of 2, 4 or 6 bytes. The Program counter 5061 is modified by either a context switch operation or a branch taken operation of a branch instruction for example. In a context switch operation,

the current program counter value is saved in the program status word along with other state information about the program being executed (such as condition codes), and a new program counter value is loaded pointing to an instruction of a new program module to be executed. A branch taken operation is performed in order to permit the program to make decisions or loop within the program by loading the result of the branch instruction into the program counter 5061.

Typically an instruction fetch unit 5055 is employed to fetch instructions on behalf of the processor 5026. The fetch unit either fetches “next sequential instructions”, target instructions of branch taken instructions, or first instructions of a program following a context switch. Modern Instruction fetch units often employ prefetch techniques to speculatively prefetch instructions based on the likelihood that the prefetched instructions might be used. For example, a fetch unit may fetch 16 bytes of instruction that includes the next sequential instruction and additional bytes of further sequential instructions.

The fetched instructions are then executed by the processor 5026. In an embodiment, the fetched instruction(s) are passed to a dispatch unit 5056 of the fetch unit. The dispatch unit decodes the instruction(s) and forwards information about the decoded instruction(s) to appropriate units 5057, 5058, 5060. An execution unit 5057 will typically receive information about decoded arithmetic instructions from the instruction fetch unit 5055 and will perform arithmetic operations on operands according to the opcode of the instruction. Operands are provided to the execution unit 5057 preferably either from memory 5025, architected registers 5059 or from an immediate field of the instruction being executed. Results of the execution, when stored, are stored either in memory 5025, registers 5059 or in other machine hardware (such as control registers, PSW registers and the like).

A processor 5026 typically has one or more units 5057, 5058, 5060 for executing the function of the instruction. Referring to FIG. 12A, an execution unit 5057 may communicate with architected general registers 5059, a decode/dispatch unit 5056, a load store unit 5060, and other 5065 processor units by way of interfacing logic 5071. An execution unit 5057 may employ several register circuits 5067, 5068, 5069 to hold information that the arithmetic logic unit (ALU) 5066 will operate on. The ALU performs

arithmetic operations such as add, subtract, multiply and divide as well as logical function such as and, or and exclusive-or (XOR), rotate and shift. Preferably the ALU supports specialized operations that are design dependent. Other circuits may provide other architected facilities 5072 including condition codes and recovery support logic for example. Typically the result of an ALU operation is held in an output register circuit 5070 which can forward the result to a variety of other processing functions. There are many arrangements of processor units, the present description is only intended to provide a representative understanding of one embodiment.

An ADD instruction for example would be executed in an execution unit 5057 having arithmetic and logical functionality while a floating point instruction for example would be executed in a floating point execution having specialized floating point capability. Preferably, an execution unit operates on operands identified by an instruction by performing an opcode defined function on the operands. For example, an ADD instruction may be executed by an execution unit 5057 on operands found in two registers 5059 identified by register fields of the instruction.

The execution unit 5057 performs the arithmetic addition on two operands and stores the result in a third operand where the third operand may be a third register or one of the two source registers. The execution unit preferably utilizes an Arithmetic Logic Unit (ALU) 5066 that is capable of performing a variety of logical functions such as Shift, Rotate, And, Or and XOR as well as a variety of algebraic functions including any of add, subtract, multiply, divide. Some ALUs 5066 are designed for scalar operations and some for floating point. Data may be Big Endian (where the least significant byte is at the highest byte address) or Little Endian (where the least significant byte is at the lowest byte address) depending on architecture. The IBM z/Architecture® is Big Endian. Signed fields may be sign and magnitude, 1's complement or 2's complement depending on architecture. A 2's complement number is advantageous in that the ALU does not need to design a subtract capability since either a negative value or a positive value in 2's complement requires only an addition within the ALU. Numbers are commonly described in shorthand, where a 12 bit field defines an address of a 4,096 byte block and is commonly described as a 4 Kbyte (Kilo-byte) block, for example.

Referring to FIG. 12B, branch instruction information for executing a branch instruction is typically sent to a branch unit 5058 which often employs a branch prediction algorithm such as a branch history table 5082 to predict the outcome of the branch before other conditional operations are complete. The target of the current branch instruction will be fetched and speculatively executed before the conditional operations are complete. When the conditional operations are completed the speculatively executed branch instructions are either completed or discarded based on the conditions of the conditional operation and the speculated outcome. A typical branch instruction may test condition codes and branch to a target address if the condition codes meet the branch requirement of the branch instruction, a target address may be calculated based on several numbers including ones found in register fields or an immediate field of the instruction for example. The branch unit 5058 may employ an ALU 5074 having a plurality of input register circuits 5075, 5076, 5077 and an output register circuit 5080. The branch unit 5058 may communicate with general registers 5059, decode dispatch unit 5056 or other circuits 5073, for example.

The execution of a group of instructions can be interrupted for a variety of reasons including a context switch initiated by an operating system, a program exception or error causing a context switch, an I/O interruption signal causing a context switch or multi-threading activity of a plurality of programs (in a multi-threaded environment), for example. Preferably a context switch action saves state information about a currently executing program and then loads state information about another program being invoked. State information may be saved in hardware registers or in memory for example. State information preferably comprises a program counter value pointing to a next instruction to be executed, condition codes, memory translation information and architected register content. A context switch activity can be exercised by hardware circuits, application programs, operating system programs or firmware code (microcode, pico-code or licensed internal code (LIC)) alone or in combination.

A processor accesses operands according to instruction defined methods. The instruction may provide an immediate operand using the value of a portion of the instruction, may provide one or more register fields explicitly pointing to either general purpose registers or special purpose registers (floating point registers for example). The instruction may utilize

implied registers identified by an opcode field as operands. The instruction may utilize memory locations for operands. A memory location of an operand may be provided by a register, an immediate field, or a combination of registers and immediate field as exemplified by the z/Architecture® long displacement facility wherein the instruction
5 defines a base register, an index register and an immediate field (displacement field) that are added together to provide the address of the operand in memory for example. Location herein typically implies a location in main memory (main storage) unless otherwise indicated.

10 Referring to FIG. 12C, a processor accesses storage using a load/store unit 5060. The load/store unit 5060 may perform a load operation by obtaining the address of the target operand in memory 5053 and loading the operand in a register 5059 or another memory 5053 location, or may perform a store operation by obtaining the address of the target
15 operand in memory 5053 and storing data obtained from a register 5059 or another memory 5053 location in the target operand location in memory 5053. The load/store unit 5060 may be speculative and may access memory in a sequence that is out-of-order relative to instruction sequence, however the load/store unit 5060 is to maintain the appearance to programs that instructions were executed in order. A load/store unit 5060 may communicate
20 with general registers 5059, decode/dispatch unit 5056, cache/memory interface 5053 or other elements 5083 and comprises various register circuits, ALUs 5085 and control logic 5090 to calculate storage addresses and to provide pipeline sequencing to keep operations in order. Some operations may be out of order but the load/store unit provides functionality to make the out of order operations to appear to the program as having been performed in
25 order, as is well known in the art.

Preferably addresses that an application program “sees” are often referred to as virtual addresses. Virtual addresses are sometimes referred to as “logical addresses” and “effective
30 addresses”. These virtual addresses are virtual in that they are redirected to physical memory location by one of a variety of dynamic address translation (DAT) technologies including, but not limited to, simply prefixing a virtual address with an offset value, translating the virtual address via one or more translation tables, the translation tables preferably comprising at least a segment table and a page table alone or in combination,

preferably, the segment table having an entry pointing to the page table. In the z/Architecture®, a hierarchy of translation is provided including a region first table, a region second table, a region third table, a segment table and an optional page table. The performance of the address translation is often improved by utilizing a translation lookaside buffer (TLB) which comprises entries mapping a virtual address to an associated physical memory location. The entries are created when the DAT translates a virtual address using the translation tables. Subsequent use of the virtual address can then utilize the entry of the fast TLB rather than the slow sequential translation table accesses. TLB content may be managed by a variety of replacement algorithms including LRU (Least Recently used).

In the case where the processor is a processor of a multi-processor system, each processor has responsibility to keep shared resources, such as I/O, caches, TLBs and memory, interlocked for coherency. Typically, “snoop” technologies will be utilized in maintaining cache coherency. In a snoop environment, each cache line may be marked as being in any one of a shared state, an exclusive state, a changed state, an invalid state and the like in order to facilitate sharing.

I/O units 5054 (FIG. 11) provide the processor with means for attaching to peripheral devices including tape, disc, printers, displays, and networks for example. I/O units are often presented to the computer program by software drivers. In mainframes, such as the System z® from IBM®, channel adapters and open system adapters are I/O units of the mainframe that provide the communications between the operating system and peripheral devices.

Further, other types of computing environments can benefit from one or more aspects of the present invention. As an example, an environment may include an emulator (e.g., software or other emulation mechanisms), in which a particular architecture (including, for instance, instruction execution, architected functions, such as address translation, and architected registers) or a subset thereof is emulated (e.g., on a native computer system having a processor and memory). In such an environment, one or more emulation functions of the emulator can implement one or more aspects of the present invention, even though a computer executing the emulator may have a different architecture than the capabilities being emulated. As one example, in emulation mode, the specific instruction or operation

being emulated is decoded, and an appropriate emulation function is built to implement the individual instruction or operation.

5 In an emulation environment, a host computer includes, for instance, a memory to store instructions and data; an instruction fetch unit to fetch instructions from memory and to optionally, provide local buffering for the fetched instruction; an instruction decode unit to receive the fetched instructions and to determine the type of instructions that have been fetched; and an instruction execution unit to execute the instructions. Execution may include loading data into a register from memory; storing data back to memory from a register; or
10 performing some type of arithmetic or logical operation, as determined by the decode unit. In one example, each unit is implemented in software. For instance, the operations being performed by the units are implemented as one or more subroutines within emulator software.

15 More particularly, in a mainframe, architected machine instructions are used by programmers, usually today “C” programmers, often by way of a compiler application. These instructions stored in the storage medium may be executed natively in a z/Architecture® IBM® Server, or alternatively in machines executing other architectures. They can be emulated in the existing and in future IBM® mainframe servers and on other
20 machines of IBM® (e.g., Power Systems servers and System x® Servers). They can be executed in machines running Linux on a wide variety of machines using hardware manufactured by IBM®, Intel®, AMD™, and others. Besides execution on that hardware under a z/Architecture®, Linux can be used as well as machines which use emulation by Hercules or FSI (Fundamental Software, Inc), where generally execution is in an emulation
25 mode. In emulation mode, emulation software is executed by a native processor to emulate the architecture of an emulated processor. Information about the above-referenced emulator products is available on the World Wide Web at, respectively, www.hercules-390.org and www.funsoft.com.

30 The native processor typically executes emulation software comprising either firmware or a native operating system to perform emulation of the emulated processor. The emulation software is responsible for fetching and executing instructions of the emulated processor

architecture. The emulation software maintains an emulated program counter to keep track of instruction boundaries. The emulation software may fetch one or more emulated machine instructions at a time and convert the one or more emulated machine instructions to a corresponding group of native machine instructions for execution by the native processor.

5 These converted instructions may be cached such that a faster conversion can be accomplished. Notwithstanding, the emulation software is to maintain the architecture rules of the emulated processor architecture so as to assure operating systems and applications written for the emulated processor operate correctly. Furthermore, the emulation software is to provide resources identified by the emulated processor architecture including, but not
10 limited to, control registers, general purpose registers, floating point registers, dynamic address translation function including segment tables and page tables for example, interrupt mechanisms, context switch mechanisms, Time of Day (TOD) clocks and architected interfaces to I/O subsystems such that an operating system or an application program designed to run on the emulated processor, can be run on the native processor having the
15 emulation software.

A specific instruction being emulated is decoded, and a subroutine is called to perform the function of the individual instruction. An emulation software function emulating a function of an emulated processor is implemented, for example, in a "C" subroutine or driver, or
20 some other method of providing a driver for the specific hardware as will be within the skill of those in the art after understanding the description of the preferred embodiment. Various software and hardware emulation patents including, but not limited to U.S. Letters Patent No. 5,551,013, entitled "Multiprocessor for Hardware Emulation", by Beausoleil et al.; and U.S. Letters Patent No. 6,009,261, entitled "Preprocessing of Stored Target Routines for
25 Emulating Incompatible Instructions on a Target Processor", by Scalzi et al; and U.S. Letters Patent No. 5,574,873, entitled "Decoding Guest Instruction to Directly Access Emulation Routines that Emulate the Guest Instructions", by Davidian et al; and U.S. Letters Patent No. 6,308,255, entitled "Symmetrical Multiprocessing Bus and Chipset Used for Coprocessor Support Allowing Non-Native Code to Run in a System", by Gorishek et al; and U.S. Letters
30 Patent No. 6,463,582, entitled "Dynamic Optimizing Object Code Translator for Architecture Emulation and Dynamic Optimizing Object Code Translation Method", by Lethin et al; and U.S. Letters Patent No. 5,790,825, entitled "Method for Emulating Guest

Instructions on a Host Computer Through Dynamic Recompilation of Host Instructions”, by Eric Traut; and many others, illustrate a variety of known ways to achieve emulation of an instruction format architected for a different machine for a target machine available to those skilled in the art.

5

10

15

20

25

30

In FIG. 13, an example of an emulated host computer system 5092 is provided that emulates a host computer system 5000' of a host architecture. In the emulated host computer system 5092, the host processor (CPU) 5091 is an emulated host processor (or virtual host processor) and comprises an emulation processor 5093 having a different native instruction set architecture than that of the processor 5091 of the host computer 5000'. The emulated host computer system 5092 has memory 5094 accessible to the emulation processor 5093. In the example embodiment, the memory 5094 is partitioned into a host computer memory 5096 portion and an emulation routines 5097 portion. The host computer memory 5096 is available to programs of the emulated host computer 5092 according to host computer architecture. The emulation processor 5093 executes native instructions of an architected instruction set of an architecture other than that of the emulated processor 5091, the native instructions obtained from emulation routines memory 5097, and may access a host instruction for execution from a program in host computer memory 5096 by employing one or more instruction(s) obtained in a sequence & access/decode routine which may decode the host instruction(s) accessed to determine a native instruction execution routine for emulating the function of the host instruction accessed. Other facilities that are defined for the host computer system 5000' architecture may be emulated by architected facilities routines, including such facilities as general purpose registers, control registers, dynamic address translation and I/O subsystem support and processor cache, for example. The emulation routines may also take advantage of functions available in the emulation processor 5093 (such as general registers and dynamic translation of virtual addresses) to improve performance of the emulation routines. Special hardware and off-load engines may also be provided to assist the processor 5093 in emulating the function of the host computer 5000'.

The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the invention. As used herein, the singular forms “a”, “an” and “the” are intended to include the plural forms as well, unless the context clearly

indicates otherwise. It will be further understood that the terms “comprises” and/or “comprising”, when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components and/or groups thereof.

The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below, if any, are intended to include any structure, material, or act for performing the function in combination with other claimed elements as specifically claimed. The description of the present invention has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope of the invention. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiment with various modifications as are suited to the particular use contemplated.

CLAIMS

1. A method for enabling adapters in a computing environment, comprising the steps of:
responsive to executing a Call Logical Processor (CLP) instruction for enabling an
5 adapter, the CLP instruction comprising a function handle identifying the adapter and having
an adapter not enabled indicator, the CLP instruction requesting a number of direct memory
access (DMA) address spaces, the execution enabling one or more DMA address spaces
comprising a) and b):

10 a) enabling the adapter, wherein the enabling comprises enabling registration for
address translation and interruptions for supporting direct memory accesses and message
signaled interruptions for the adapter; and

b) returning the function handle having an adapter enabled indicator.

15 2. The method of claim 1, wherein the enabling comprises enabling one or more
instructions to be issued to the adapter.

3. The method of claim 1, wherein the number of address spaces to be assigned is
indicated in a request block of the CLP instruction.

20 4. The method of claim 1, wherein the enabling further comprises:
using the function handle of the adapter to locate a function table entry associated
with the adapter; and
employing information in the function table entry to determine whether the adapter is
to be enabled, and wherein the enabling assigns one or more device table entries to the
25 adapter responsive to determining the adapter is to be enabled.

30 5. The method of claim 4, wherein the function handle is associated with the function
table entry and including a function number and an instance number, and wherein the
method further comprises determining the validity of the handle, the determining
comprising:

checking that the not enabled indicator indicates not enabled; and

checking that the function number designates an installed function, wherein the using is performed responsive to determining a valid handle.

5 6. The method of claim 4, wherein the employing comprises checking at least one of the not enabled indicator, a permanent error state indicator, an error recovery initiated indicator, a busy indicator or a permission indicator in the function table entry to determine whether the adapter is to be enabled.

10 7. The method of claim 1, wherein the enabling further comprises associating one or more device table entries with a function table entry associated with the adapter, the function table entry providing information regarding the adapter.

15 8. The method of claim 7, wherein the function table entry is associated with the function handle, and wherein the enabling further comprises performing at least one of:

 setting a function enable indicator in the function table entry to indicate enabled;
 setting one or more device enable indicators in the one or more device table entries to indicate enabled;

20 including in a contents addressable memory one or more indices to one or more device table entries, the contents addressable memory usable in locating a device table entry responsive to a request from the adapter;

 setting the adapter enabled indicator in the function handle to indicate enabled; and
 updating an instance number of the function handle.

25 9. The method of claim 1, wherein the method further comprises disabling the adapter.

10. The method of claim 9, wherein the disabling comprises:

 using the function handle to locate a function table entry associated with the adapter;
 and

30 employing information in the function table entry to determine whether the adapter is to be disabled, and proceeding with disabling responsive to determining the adapter is to be disabled.

11. The method of claim 10, wherein the proceeding with disabling comprises at least one of:

setting a function enable indicator in the function table entry to disabled;
clearing and releasing one or more device table entries associated with the adapter;

5 and

setting the not enabled indicator of the function handle to indicate disabled.

12. The method of claim 10, wherein the disabling further comprises determining the validity of the function handle, the determining comprising:

10 checking that the adapter enabled indicator is set to enabled; and

checking that the handle points to a valid entry in the function table, wherein the using is performed responsive to determining the function handle is valid.

13. The method of claim 12, wherein the employing comprises comparing an instance number in the function handle with an instance number in the function table entry, wherein the proceeding with disabling occurs responsive to the comparing indicating equality.

14. The method of claim 1, wherein the adapter comprises a Peripheral Component Interconnect (PCI) function.

15. A computer program comprising computer program code stored on a computer readable medium to, when loaded into a computer system and executed thereon, cause said computer system to perform all the steps of a method according to any of claims 1 to 14.

16. A computer system for enabling adapters in a computing environment, said computer system comprising:

a memory; and

a processor in communications with the memory;

a requestor element responsive to executing a Call Logical Processor (CLP)

instruction for enabling an adapter, the CLP instruction comprising a function handle identifying the adapter and having an adapter not enabled indicator, the CLP instruction

requesting a number of direct memory access (DMA) address spaces, the execution enabling one or more DMA address spaces comprising a) and b):

- a) enabling the adapter, wherein the enabling comprises enabling registration for address translation and interruptions for supporting direct memory accesses and message signaled interruptions for the adapter; and
- b) returning the function handle having an adapter enabled indicator.

17. A computer system for enabling adapters in a computing environment, said computer system comprising:

- a memory; and
- a processor in communications with the memory, wherein the computer system is configured to perform a method, said method comprising:

responsive to executing a Call Logical Processor (CLP) instruction for enabling an adapter, the CLP instruction comprising a function handle identifying the adapter and having an adapter not enabled indicator, the CLP instruction requesting a number of direct memory access (DMA) address spaces, the execution enabling one or more DMA address spaces comprising a) and b):

- a) enabling the adapter, wherein the enabling comprises enabling registration for address translation and interruptions for supporting direct memory accesses and message signaled interruptions for the adapter; and
- b) returning the function handle having an adapter enabled indicator.

18. The computer system of claim 17, wherein the enabling comprises enabling one or more instructions to be issued to the adapter.

19. The computer system of claim 17, wherein the enabling further comprises:

using the function handle of the adapter to locate a function table entry associated with the adapter; and

employing information in the function table entry to determine whether the adapter is to be enabled, and wherein the enabling assigns one or more device table entries to the adapter responsive to determining the adapter is to be enabled.

20. The computer system of claim 17, wherein the enabling further comprises associating one or more device table entries with a function table entry associated with the adapter, the function table entry providing information regarding the adapter.

5 21. The computer system of claim 20, wherein the function table entry is associated with the function handle, and wherein the enabling further comprises performing at least one of:
setting a function enable indicator in the function table entry to indicate enabled;
setting one or more device enable indicators in the one or more device table entries to indicate enabled;

10 including in a contents addressable memory one or more indices to one or more device table entries, the contents addressable memory usable in locating a device table entry responsive to a request from the adapter;

setting the adapter enabled indicator in the function handle to indicate enabled; and
updating an instance number of the function handle.

15 22. The computer system of claim 17, wherein the method further comprises disabling the adapter, and wherein the disabling comprises:

using the function handle to locate a function table entry associated with the adapter;
and

20 employing information in the function table entry to determine whether the adapter is to be disabled, and proceeding with disabling responsive to determining the adapter is to be disabled.

23. The computer system of claim 22, wherein the proceeding with disabling comprises
25 at least one of:

setting a function enable indicator in the function table entry to disabled;
clearing and releasing one or more device table entries associated with the adapter;

and

setting the not enabled indicator of the function handle to indicate disabled.

30 24. A computer program product for enabling adapters in a computing environment, said computer program product comprising:

a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing the steps of the method of any of claims 1 to 14.