US 20040221276A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2004/0221276 A1

Raj (43) Pub. Date: Nov. 4, 2004

(54) METHOD AND COMPUTER PROGRAM FOR DATA CONVERSION IN A HETEROGENEOUS COMMUNICATIONS NETWORK

(76) Inventor: **Ashok Raj**, Portland, OR (US)

Correspondence Address:
BLAKELY SOKOLOFF TAYLOR & ZAFMAN
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1030 (US)

(21) Appl. No.: **10/860,838**

(22) Filed: **Jun. 3, 2004**

Related U.S. Application Data

(63) Continuation of application No. 09/714,647, filed on Nov. 17, 2000, now Pat. No. 6,772,320.

Publication Classification

(51) **Int. Cl.**[7] **G06F 9/45**; G11C 5/00

(52) **U.S. Cl.** **717/136**; 719/320

(57) **ABSTRACT**

A method and computer program for data conversion in a heterogeneous communications network. This method and computer program converts data for computer systems having different data storage architectures so that these computer systems may simply and easily communicate over a network. This is most useful when converting data store in little endian and big endian format. This method relies on creating the data structure used to convert the data using embedded macros that are not executed at run time. These embedded macros are expanded by the compiler to generate the data structure and thereby saves substantial time during execution.

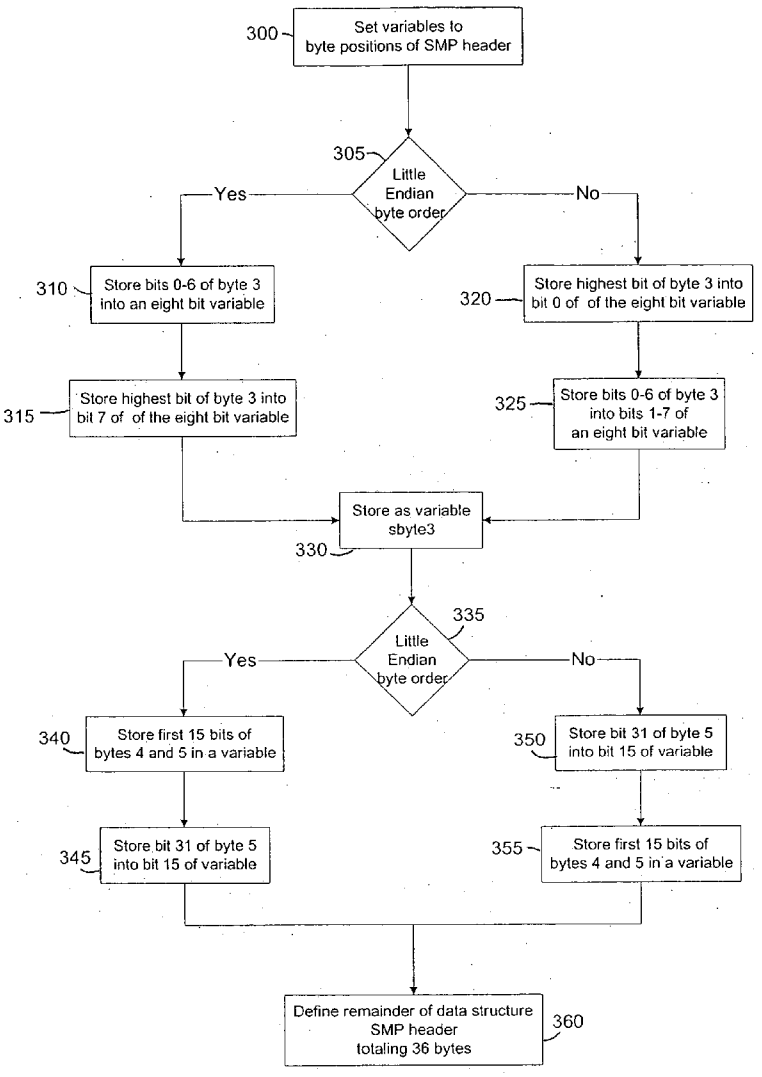


FIG. 1

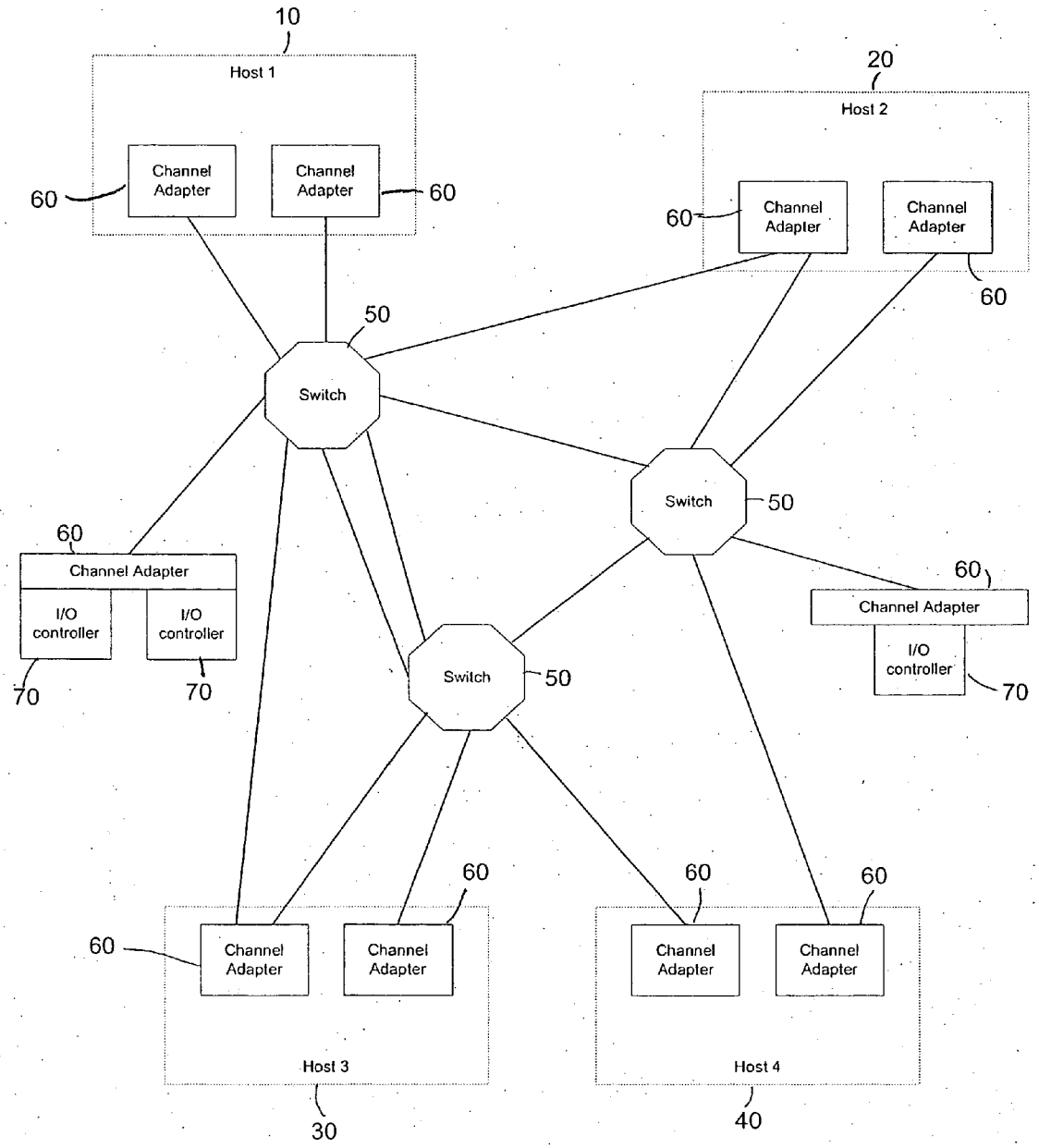
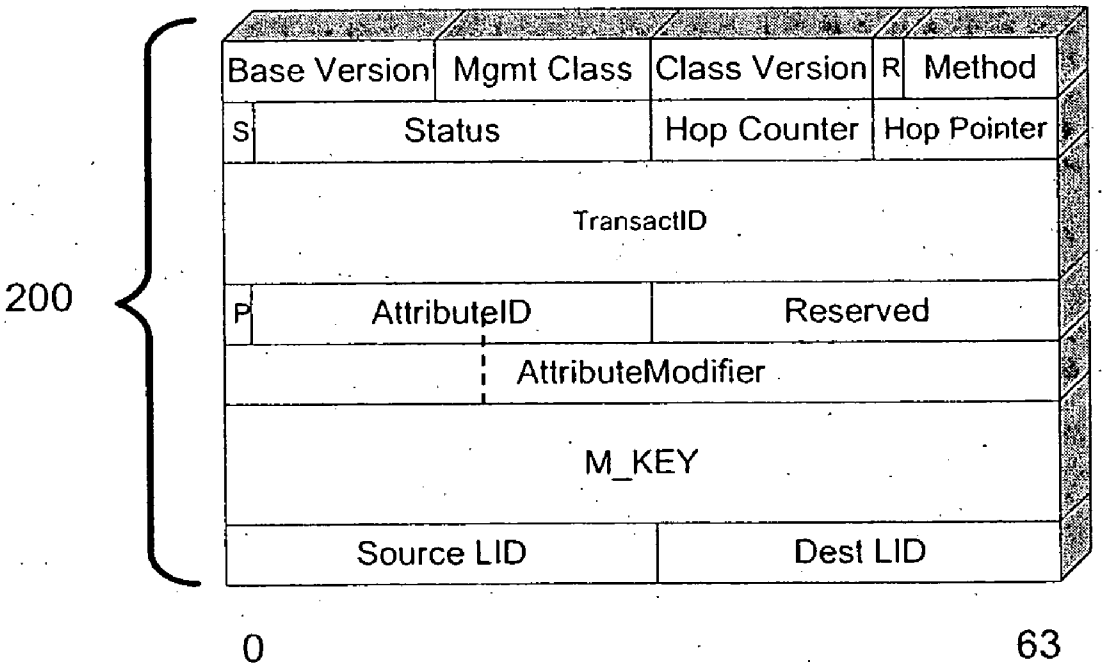

10
Host 1
20
Host 2
Channel Adapter
Channel Adapter
60
60
60
Channel Adapter
Channel Adapter
60
50
Switch
Switch
50
60
Channel Adapter
I/O controller
I/O controller
70
70
Switch
50
60
Channel Adapter
I/O controller
70
60
60
60
60
Channel Adapter
Channel Adapter
Channel Adapter
Channel Adapter
Host 3
Host 4
30
40

FIG. 2


Base Version
Mgmt Class
Class Version
R
Method
S
Status
Hop Counter
Hop Pointer
TransactID
P
AttributeID
Reserved
AttributeModifier
M_KEY
Source LID
Dest LID
200
0
63

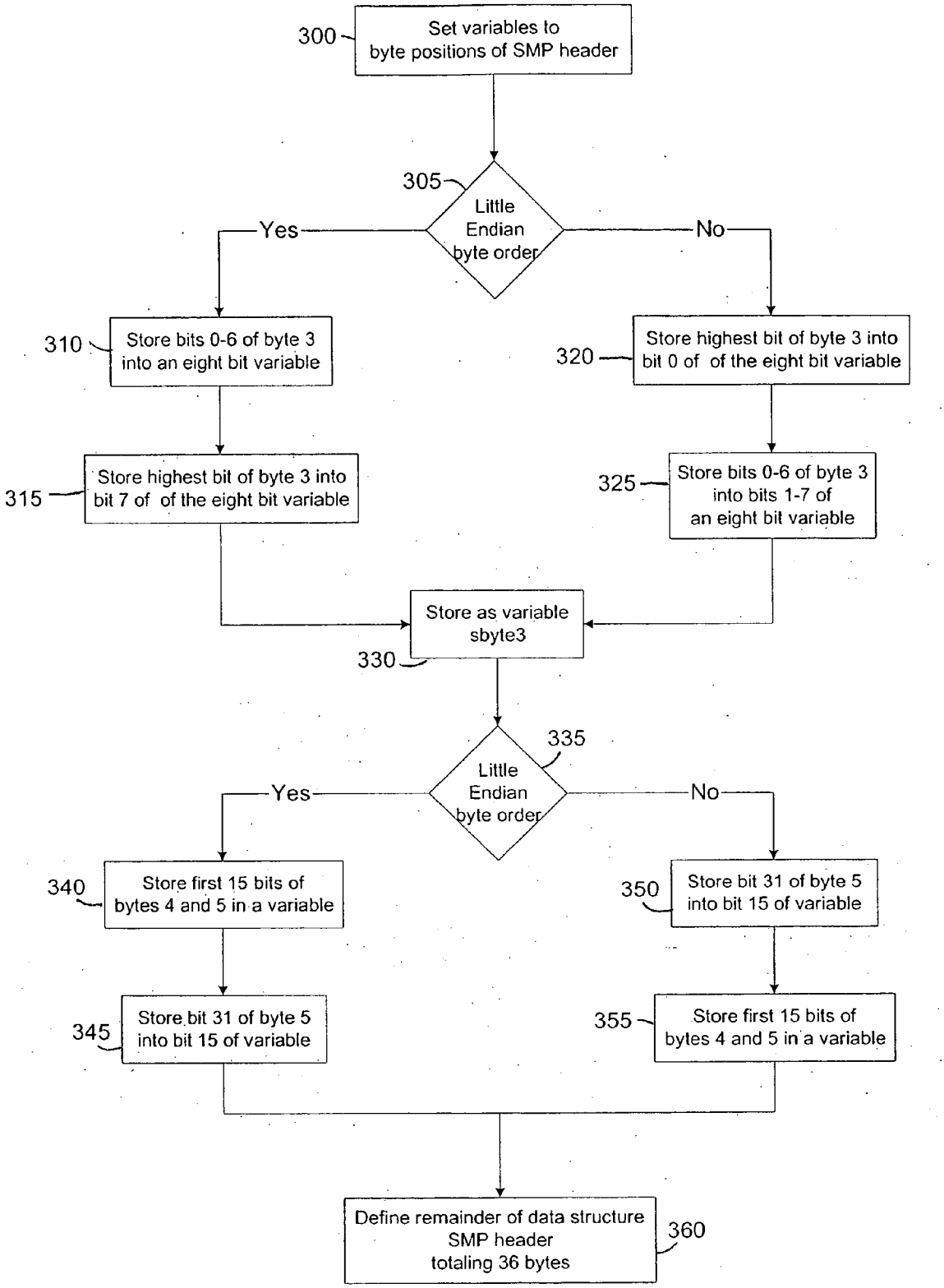


FIG. 3

Data Structure Definition

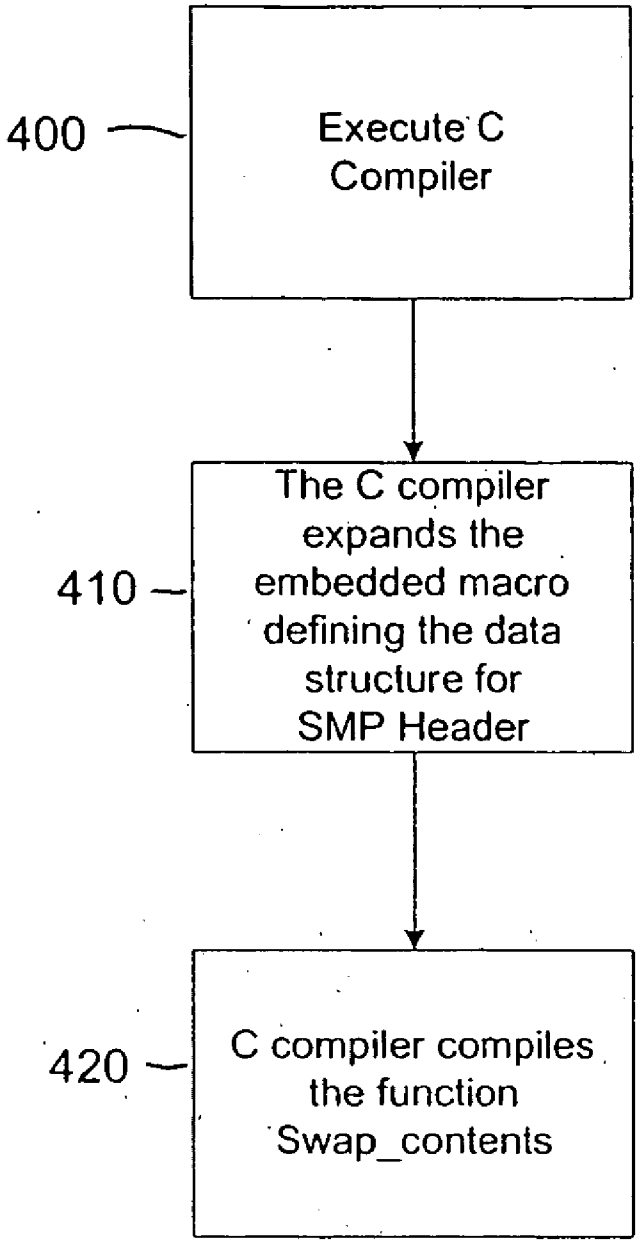


FIG. 4

Compiler expansion of embedded macros to create data structure

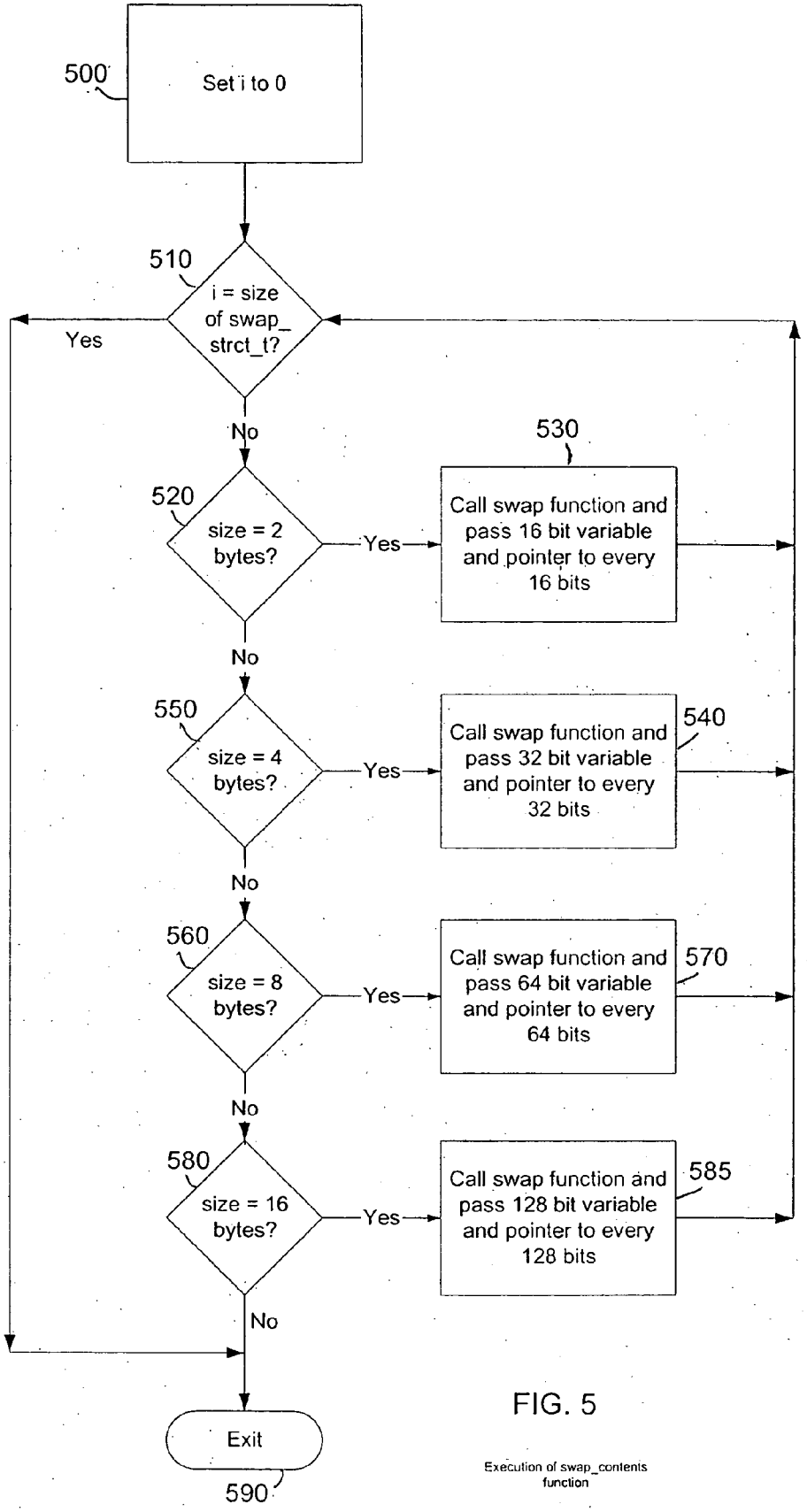


FIG. 5

Execution of swap_contents function

METHOD AND COMPUTER PROGRAM FOR DATA CONVERSION IN A HETEROGENEOUS COMMUNICATIONS NETWORK

FIELD

[0001] The invention relates to a method and computer program for data conversion in a heterogeneous communications network. More particularly, the present invention converts data for computer systems having different data storage architectures so that these computer systems may simply and easily communicate over a network.

BACKGROUND

[0002] In the rapid development of computers many advancements have been seen in the areas of processor speed, throughput, communications, and fault tolerance. Initially computer systems were standalone devices in which a processor, memory and peripheral devices all communicated through a single bus. Later, in order to improve performance, several processors and were interconnected to memory and peripherals using one or more buses. In addition, separate computer systems were linked together through different communications mechanisms such as, shared memory, serial and parallel ports, local area networks (LAN) and wide area networks (WAN). However, these mechanisms have proven to be relatively slow and subject to interruptions and failures when a critical communications component fails. Further, in the case where a high speed direct memory access communications method is used, all to often these communications systems are limited in the types of computers that may access and exchange information on the network.

[0003] Examples of differing processor and memory architectures may be seen in big endian versus little endian architectures. In a big endian architecture the most significant byte is placed in a lower memory address. This is typically the way humans deal with arithmetic functions and the method employed by a Motorola™ 680x0 system. There is also a little endian architecture in which the least significant byte is placed in a lower memory address. This is done since numbers are calculated by a processor starting with least significant digits and little endian data structures are already set up to facilitate this operation. This type of little endian architecture is employed by the Intel™x86 line of systems. In addition, a bi-endian machine has been developed, such as the PowerPC™, which can handle both types of byte ordering.

[0004] However, when a big endian machine and a little endian machine attempt to communicate through memory reads and writes to each other, the data must be re-formatted to be accessible by the other machine. In order to accomplish this programmer's have developed code that swaps bits and bytes of data within words being accessed. However, this development of code by different programmers is time-consuming and prone to error. Further, this code is often specific to each specific application generated. Therefore, one of the purposes of direct memory access for communications is defeated by the lengthy time requirements for this conversion process. Also, time involved in developing and debugging software for each application may be substantial and thus expensive.

[0005] Therefore, what is needed is a method and computer program which will create a data structure to enable the conversion from one memory storage format to another. This method and computer program must be generally applicable to all applications programs and all types of data. This method and computer program must also be efficient in the generation of the data structure and the re-formatting of the data. Thus, this method and computer program must be easily usable by all applications and programmers and must execute in run-time very quickly so as to facilitate the rapid exchange of information.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] The foregoing and a better understanding of the present invention will become apparent from the following detailed description of exemplary embodiments and the claims when read in connection with the accompanying drawings, all forming a part of the disclosure of this invention. While the foregoing and following written and illustrated disclosure focuses on disclosing example embodiments of the invention, it should be clearly understood that the same is by way of illustration and example only and the invention is not limited thereto. The spirit and scope of the present invention are limited only by the terms of the appended claims.

[0007] The following represents brief descriptions of the drawings, wherein:

[0008] **FIG. 1** is an example of an overall InfiniBand systems diagram;

[0009] **FIG. 2** is an example of a InfiniBand subnet management packet (SMP) header;

[0010] **FIG. 3** is a flowchart of the logic used to create an InfiniBand subnet management packet (SMP) header data structure in the example embodiments of the present invention;

[0011] **FIG. 4** is a flowchart of a compiler expansion of embedded macros used to create the SMP header data structure in the embodiments of the present invention; and

[0012] **FIG. 5** is a flowchart of the logic involved in converting data used in the example embodiments of the present invention.

DETAILED DESCRIPTION

[0013] Before beginning a detailed description of the subject invention, mention of the following is in order. When appropriate, like reference numerals and characters may be used to designate identical, corresponding or similar components in differing figure drawings. Further, in the detailed description to follow, exemplary sizes/models/values/ranges may be given, although the present invention is not limited to the same. As a final note, well-known components of computer networks may not be shown within the FIGS. for simplicity of illustration and discussion, and so as not to obscure the invention.

[0014] **FIG. 1** is an example of an overall InfiniBand systems diagram which may be used by the embodiments of the present invention. Using such an InfiniBand architecture it may be possible to link together a processor based system **10**, through switches **50** to several Input/Output (I/O) controllers **70**, and other processor based systems **20**, **30** and **40**. Each processor based system **10**, **20**, **30** and **40** may be composed of one or more central processing units (CPU)

(not shown), dynamic random access memory (DRAM) (not shown), memory controller (not shown) and a channel adapter **60**. A switch **50** may be used to interconnect serial ports to achieve transfer rates of more than one gigabit-per-second.

[**0015**] Referring to **FIG. 1**, the InfiniBand architecture defines interfaces that move data between two "memory" regions or nodes. Access to an I/O controller **70** and processor based system **10**, **20**, **30** and **40**, may be accomplished by send or receive operations, as well as, remote direct memory access (RDMA) read and RDMA write operations. Cluster or channel adapters provide the control and logic that allows nodes to communicate to each other over the InfiniBand network. A processor based system **10**, **20**, **30** or **40** may have one or more channel adapters **60** connected to it. Further, an I/O controller **70** may have one or more channel adapters **60** connected to it. Communications in an InfiniBand architecture may be accomplished through these cluster or channel adapters **60** directly or through switches **50**.

[**0016**] As can be seen in **FIG. 1**, the InfiniBand architecture enables redundant communications links between channel adapters **60** and switches **50**. Further, it may be possible to create a routing and distance table to identify the shortest paths between nodes in the network. In this case, distance is defined as being the shortest time between to points and not the physical distance. A node or cluster adapter may be a channel adapter **60**. Therefore, when data is sent to a memory location in a node it will take the shortest path available and arrive as fast as possible. However, if a failure occurs to a switch **50** then an alternate path may have to be configured and the distance table would have to be computed again.

[**0017**] **FIG. 2** is an example of an InfiniBand subnet management packet (SMP) header **200**. Each InfiniBand architecture is divided into subnets having at least one subnet manager (SM). Each SM may reside on a switch **50** and is responsible for configuring the subnet. Communications between the SM and other subnet management agents (SMA) is done through subnet management packet (SMP). Therefore, whenever a computer resides on the subnet it is necessary to be able to reformat data so that it meets the requirements of either big endian or little endian data format. This also applied to the SMP header **200** illustrated in **FIG. 2**.

[**0018**] The SMP header **200** illustrated in **FIG. 2** is composed of several 64-bit fields. The SMP header **200** defines the base version which is a one byte field indicating the version of management datagram based format. A one byte field for management class is provided which represents a subnet management class value. A one byte field for class version is provided which indicates the version field for the management class. A one bit field R is provided to indicate a request/response field. Further, a 7 bit field is provided for method representing the method of subset management being used. In the second 64-bit field a two byte status field is provided which encodes the status of the method. In addition, a hop pointer is provided to indicate the current byte of the initial/return path field. A hop counter is used to contain the number of valid bites in the initial/return that indicates how many direct route hops to take. In the third 64-bit field a transaction identification (ID) is provided indicative of a transaction specific identifier. In the fourth 64-bit field a 2 byte attribute identification (ID) is provided indicating the attributes for the SMP. Attributes for SMP's are typically data objects that are composite structures having registers in subnet nodes and these attributes may be read or written. The SMP method uses the attribute ID and attribute modifier to re-define or modify specific registers within a given node. Thereafter, two additional bytes are reserved for future use. In the fifth field an attribute modifier having four bytes is provided for an SMP to use as the index value to further specify data to be obtained or modified. In the sixth field a management key (M_KEY) having eight bytes is provided for SMP authentication. Thereafter, in the seventh field a source local identifier (SLID) and a destination local identifier (DLID) is provided for directed routing.

[**0019**] **FIGS. 3 through 5** are flowcharts representing the logic employed by the example embodiments of the present invention. The blocks illustrated in **FIGS. 3 through 5** represent modules, code, code segments, commands, firmware, hardware, instructions and data that are executable by a processor-based system(s) and may be written in a programming language, such as, but not limited, to C++.

[**0020**] **FIG. 3** is a flowchart of the logic used to create an InfiniBand subnet management packet (SMP) header **200**, shown in **FIG. 2**, data structure in the example embodiments of the present invention. The logic shown in **FIG. 3** may be implemented in the sample C++ program illustrated in Table 1 provided ahead. It should be noted that the C++ program illustrated in Table 1 is provided by way of example only and as one of ordinary skill in the art would appreciate this program may be implemented using any language and different commands.

TABLE 1

```
typedef struct_SMP_DIR_HDR {
   uint8   base_ver;   // byte 0
   uint8   mgmt_class;   // byte 1
   uint8 class_version;   // byte 2
   struct {      // byte 3
#ifdef LITTLE_ENDIAN
      uint8      method:7; // bits 0–7 of byte 3
      uint8      req_resp:1; // high bit of byte 3
#else
      uint8      req_resp:1; // high bit of byte 3
      uint8      method:7; // bits 0–7 of byte 3
#endif
   }sbyte3;
   struct {
#ifdef LITTLE_ENDIAN
      uint16 Status:15; // byte 4–5 15 bits
      uint16 direction:1; // bit 31 of byte 5
#else
      uint16 direction:1; // bit 31 of byte 5
      uint16 Status:15; // byte 4–5 15 bits
#endif
   }sbyte4
   uint8   hop_pointer;      // byte 6
   uint8   hop_count;         // byte 7
   uint64 transaction_id;    // bytes 8–15
   uint16 attribute_id;     // bytes 16–17
   uint16 resv1;            // bytes 18–19
   uint32 attribute_mod;      // bytes 20–23
   uint64 m_key;              // bytes 24–31
   uint16 dr_slid;            // bytes 32–33
   uint16 dr_dlid;            // bytes 34–35
} SMP_DIR_HDR;
```

[**0021**] Referring to **FIG. 3**, execution begins in operation **300** where the variables are set to byte positions of the

subnet management packet (SMP) header **200**, shown in **FIG. 2**. Thereafter, processing proceeds operation **305** where it is determined if the little endian byte order is desired. If little endian byte order is desired processing proceeds operation **310** where bits **0** through **6** of byte three are stored in an eight-bit variable. Thereafter, processing proceeds to operation **315** where the highest bit of byte three is stored into bit **7** of the of the eight-bit variable. However, if in operation **305** it is determined that little endian byte order is not desired then processing proceeds to operation **320** where the highest bit of byte three is stored into bit zero of the eight-bit variable. Processing then proceeds to operation **325** where bits **0** through **6** of byte **3** are stored into bits **1** through **7** of the eight-bit variable.

[**0022**] Regardless of whether the little endian byte order is selected in operation **305** processing proceeds to operation **330** where the eight-bit variable determined in operation **310** through **325** is stored as sbyte**3**. Thereafter, processing proceeds to operation **335** where again it is determined if little endian byte order is desired. If little endian byte order is desired then processing proceeds to operation **340** where the first 15 bits of bytes **4** and **5** are stored in a variable. Thereafter in operation **345** bit **31** of byte **5** is stored into bit **15** of the variable. However, if in operation **335** it is determined that little endian byte order is not desired then processing proceeds to operation **350**. In operation **350**, bit **31** of byte **5** is stored into bit **15** of a variable. In operation **355** the first 15 bits of bytes **4** and **5** are stored in the variable. Regardless of whether little endian byte order is desired in operation **335**, in operation **360**, the remaining portion of the SMP header **200** data structure is defined.

[**0023**] **FIG. 4** is a flowchart of a compiler expansion of embedded macros used to create the SMP header **200** data structure in the embodiments of the present invention. The processes shown in **FIG. 4** compile the embedded macros and code contained in Table II discusses further detail ahead. The operations shown in FIG. **4** begin in operation **400** by executing a standard C++ compiler. Thereafter, in operation **410**, the C++ compiler expands embedded macros shown in Table I which define the data structure for the SMP header **200** shown in Table I. These embedded macros takes the place of the logic shown and discussed in reference to **FIG. 3**. The embedded macros create the SMP header **200** data structure as shown and discussed in **FIG. 3**. The advantage of using the embedded macros shown in Table II and compiled in operation **410** is that the data structure is generated during compilation rather than during execution. The savings in terms of processor power is enormous when the data structure can be generated during compilation time which is in constant usage by application programs. Further, once tested these embedded macros may be incorporated in every application program that utilizes the SMP header **200**. Thereafter, in operation **420**, the C++ compiler compiles the function swap contents shown in Table III.

```
#define field_offset(struct_name, field) \
    ((uint16) ((char *)& ((struct_name *) 0)->field))
  #define DEFINE_STRUCT_START(name) \
    swap_struct_t swap_##name[] = {
  #define DEFINE_STRUCT_FIELD(struct_name, field) \
    {                                             \
      sizeof(((struct_name *) 0)->field), \
      field_offset(struct_name, field) \
    },
  #define DEFINE_STRUCT_END (struct_name) {0,0}};
  #define SWAP_STRUCT_NAME(struct_name) swap_##name
  // Now the definition that describes this structure
  // the same definition and function are used both while sending
  // and receiving a stucture that needs transformation.
  DEFINE_STRUCT_START (SMP_DIR_HDR)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, sbyte4)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, transaction_id)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, attribute_id)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, attribute_mod)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, m_key)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, dr_slid)
    DEFINE_STRUCT_FIELD(SMP_DIR_HDR, dr_dlid)
DEFINE_STRUCT_END
```

[**0024**] Table III is an example of the expansion of the embedded macros illustrated in Table II and executed in operation **410**. This expansion is only for the data structure referred to as "define _struct _start" and is only provided by way of example. As would be appreciated by one of ordinary skill in the art, the expansion of the macros performed by the compiler is three levels deep and further expansion in Table III is not required. It should be again noted that the expansion of the data structure is accomplished by the C++ compiler rather than during program execution as illustrated in **FIG. 3**.

TABLE III

```
// Expanded form after pre-compilation.
  swap_struct_t swap_SMP_DIR_HDR[] = // Defined by
  DEFINE_STRUCT_START
  {
    {2, 4}, // sbyte4: size 2, offset 4
    {8, 8}, // transaction id: size 8, offset 8
    {2, 16},// attribute_id: size 2, offset 16
    {4,20},// attribute mod: size 4, offset 20
    {8,24},// m_key: size 8, offset 24
    {2,32},// slid: size 2, offset 32
    {2,34},// dlid: size 2, offset 34
    {0,0}          // placed by DEFINE_STRUCT_END
};
```

[**0025**] **FIG. 5** is a flowchart of the logic involved in converting data used in the example embodiments of the present invention as performed by the swap_contents function module. The swap_contents function module begins execution in operation **500** where a variable I is set to zero. In operation **510**, it is determined whether I is equal to the size of the swap structure variable to be converted. If the value of I=the size of the swap structure variable then processing proceeds to operation **590** where processing terminates. However, if the value of I is not equal of the size of the swap structure variable, then processing proceeds to operation **520**. In operation **520**, it is determined if the size of the swap structure variable is equal to two bytes. If the swap structure variable has a size equal to two bytes then processing proceeds to operation **530**. In operation **530**, the swap function is called and passed a 16-bit variable and pointer to every 16 bits. Thereafter, processing loops back to operation **510** from operation **530**.

[**0026**] However, in operation **520** if the size of the swap structure variable is not two bytes then processing proceeds

to operation **550**. In operation **550**, it is determined whether the swap structure variable has a size equal to 4 bytes. If the swap structure variable has a size equal to 4 bytes then processing proceeds to operation **540**. In operation **540**, the swap function is called and passed a 32-bit variable and a pointer to every 32 bits. Thereafter, processing loops back from operation **540** to operation **510**.

[**0027**] However, in operation **550** if the size of the swap variable is not equal to four bytes in length then processing proceeds to operation **560**. In operation **560** it is determined if the size of the swap structure variable is equal to 8 bytes in length. If the size of the swap structure variable is equal to 8 bytes in length, then processing proceeds to operation **570**. In operation **570**, the swap function is called and passed a 64-bit variable and pointers to every 64 bits. Thereafter, processing loops back to operation **510** from operation **570**.

[**0028**] However, if in operation **560** it is determined that the swap structure variable is not equal 8 bytes in size, then processing proceeds to operation **580**. In operation **580** it is determined whether the size swap structure variable is equal to 16 bytes in length. If the swap structure variable is equal to 16 bytes in length then processing proceeds to operation **585**. In operation **585**, the swap function is called and passed a 128 bit variable with a pointer to every 128 bits. Thereafter, processing loops back to operation **510** from operation **585**. However, if the swap structure variable does not have a size of 16 bytes, then processing proceeds to operation **590** and processing terminates.

[**0029**] The benefit resulting from the present invention is that a simple, reliable, fast method and computer program is provided that will convert from little endian data format to big endian data format and back again. The data structure required to accomplish this is generated using embedded macros during compilation. Therefore, no runtime processing power is wasted creating the data structure for each application. This allows for extremely fast execution of the conversion.

[**0030**] While we have shown and described only a few examples herein, it is understood that numerous changes and modifications as known to those skilled in the art could be made to the example embodiment of the present invention. Therefore, we do not wish to be limited to the details shown and described herein, but intend to cover all such changes and modifications as are encompassed by the scope of the appended claims.

1. A method of converting the format of a data structure, comprising:

generating a plurality of embedded macros, wherein each embedded macro defines a portion of the data structure;

incorporating the plurality of embedded macros into an application program;

expanding the embedded macros to create the data structure by compiling the application program;

executing a swap function module to reformat the data structure to conform to a memory architecture specified; and

transmitting the data structure to a machine that has the ability to manipulate data contained in the data structure.

2. The method recited in claim 1, wherein the embedded macros and the application program are written in C++.

3. The method recited in claim 1, wherein the data structure is a subnet management packet.

4. The method recited in claim 3, wherein the memory architecture specified is either little endian or big endian.

5. The method recited in claim 4, wherein the swap function module modifies the data structure so as to conform to a little endian memory architecture.

6. The method recited in claim 4, wherein the swap function module modifies the data structure so as to conform to a big endian memory architecture.

7. A program embodied in a storage medium and executable by a machine, the program, when executed, resulting in performance of operations comprising:

generating a plurality of embedded macros, wherein each embedded macro defines a portion of the data structure;

incorporating the plurality of embedded macros into an application program;

expanding the embedded macros to create the data structure by compiling the application program;

executing a swap function module to reformat the data structure to conform to a memory architecture specified; and

transmitting the data structure to a machine that has the ability to manipulate data contained in the data structure.

8. The program recited in claim 7, wherein the embedded macros and the application program are written in C++.

9. The program recited in claim 7, wherein the data structure is a subnet management packet.

10. The program recited in claim 9, wherein the memory architecture specified is either little endian or big endian.

11. The program recited in claim 10, wherein the swap function module modifies the data structure so as to conform to a little endian memory architecture.

12. The program recited in claim 10, wherein the swap function module modifies the data structure so as to conform to a big endian memory architecture.

13-16. (Cancelled).

* * * * *