EXHIBIT 7

US005969705A

United States Patent [19]

Fisher et al.

[11] **Patent Number: 5,969,705**

[45] **Date of Patent: Oct. 19, 1999**

[54] **MESSAGE PROTOCOL FOR CONTROLLING A USER INTERFACE FROM AN INACTIVE APPLICATION PROGRAM**

[75] Inventors: **Stephen Fisher**, Menlo Park; **Eric Mathew Trehus**, Milpitas, both of Calif.

[73] Assignee: **Apple Computer, Inc.**, Cupertino, Calif.

[21] Appl. No.: **08/816,492**

[22] Filed: **Mar. 13, 1997**

Related U.S. Application Data

[63] Continuation of application No. 08/312,437, Sep. 26, 1994, abandoned, which is a continuation of application No. 08/084,288, Jun. 28, 1993, abandoned.

[51] **Int. Cl.**^{6} **G09G 5/00**

[52] **U.S. Cl.** **345/114**; 345/345

[58] **Field of Search** 345/119, 120, 345/118, 113, 114, 115, 343, 344, 345, 346, 347, 348; 395/155, 156, 157, 158

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,313,113	1/1982	Thornburg .
4,484,302	11/1984	Cason et al. .
4,555,775	11/1985	Pike .
4,688,167	8/1987	Agarwal .
4,698,624	10/1987	Barker et al. .

(List continued on next page.)

OTHER PUBLICATIONS

"Notebook Tabs as Target Location for Drag/Drop Operations",IB, vol. 35, No. 7, Dec. 1992.

Microsoft Corporation, "Microsoft Windows Paint User's Guide," Version 2.0, 1987, pp. 8–10, 44–45.

Microsoft Corporation, "Microsoft Windows Write User's Guide," Version 2.0, 1987, pp. 60–65.

Microsoft Corporation, "Microsoft Word: Using Microsoft Word", Version 5.0, 1989, pp. 69, 88–93.

Screen Dumps from Microsoft Windows V 3.1, Microsoft Corporation 1985–1992 (14 pages).

WordPerfect for Windows V 5.1, WordPerfect Corporation, 1991 (16 pages).

Jeffrey M. Richter, "Implementing Drag–and–Drop," *Windows 3.1: A Developer's Guide*, 2nd Edition, M&T Books, A Division of M&T Publishing, Inc. (1992), pp. 541–577 (Chapter 9).

Charles Petzold, "Windows™ 3.1—Hello to TrueType™, OLE, and Easier DDE; Farewell to Real Mode," *Microsoft Systems Journal*, vol. 6, No. 5 Sep. 1991, pp. 17–26.

Jeffrey Richter, "Drop Everything: How to Make Your Application Accept and Source Drag–and–Drop Files," *Microsoft Systems Journal*, vol. 7, No. 3, May/Jun. 1992, pp. 19–30.

Future Enterprises Inc., A Microcomputer Education Course for: U.S. Department of Commerce "Studen Workbook for Quattro Pro 3.0—Concepts and Basic Uses," 1991 (3 pages).

Inside Macintosh, vol. VI, 1991, pp. 5–1 to 5–117.

Microsoft Windows 3.1, Step by Step, 1991, pp. 168–170.

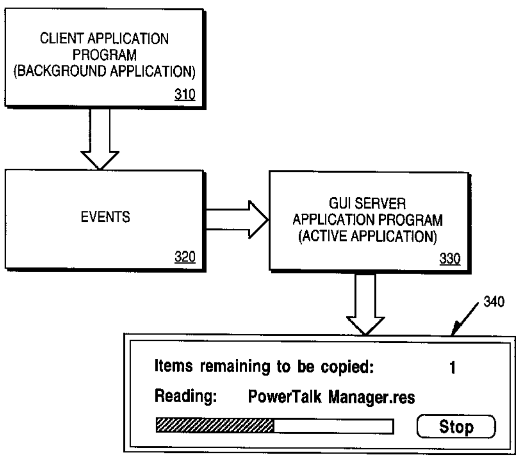
Apple Computer, Inc., *Inside Macintosh, vol. VI* Table of Contents, 5–1 through 6–117 (1991).

Primary Examiner—Chanh Nguyen
Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman

[57] **ABSTRACT**

Method and apparatus for a first process operative in a computer system controlling a user interface on a computer system display under control of a second process operative in the computer system. An event handler is installed for the second process, the event handler servicing events generated for controlling the user interface display under control of the second process. The first process may then perform a first set of functions in the computer system. The first process generates events for controlling the user interface display, the events related to the functions performed by the first process. The event handler receives the events generated by the first process and updates the user interface on the computer system display according to the events generated by the first process and received by the event handler.

1 Claim, 7 Drawing Sheets


CLIENT APPLICATION PROGRAM (BACKGROUND APPLICATION)
310
EVENTS
320
GUI SERVER APPLICATION PROGRAM (ACTIVE APPLICATION)
330
340
Items remaining to be copied: 1
Reading: PowerTalk Manager.res
Stop

U.S. PATENT DOCUMENTS

4,698,625	10/1987	McCaskill .	
4,720,703	1/1988	Schnarel, Jr. et al. .	
4,780,883	10/1988	O'Connor et al. .	
4,831,556	5/1989	Oono .	
4,862,376	8/1989	Ferriter et al. .	
4,868,765	9/1989	Diefendorff .	
4,905,185	2/1990	Sakai .	
4,922,414	5/1990	Holloway et al. .	
4,954,967	9/1990	Takahashi .	
5,047,930	9/1991	Martens et al. .	
5,079,695	1/1992	Dysart et al. .	
5,140,677	8/1992	Fleming et al. .	
5,157,763	10/1992	Peters et al. .	
5,196,838	3/1993	Meier et al. .	
5,202,828	4/1993	Vertelney et al. .	
5,214,756	5/1993	Franklin et al. .	
5,226,117	7/1993	Miklos .	
5,226,163	7/1993	Karsh et al. .	
5,228,123	7/1993	Heckel .	
5,260,697	11/1993	Barrett et al.	345/173
5,287,448	2/1994	Nicol et al. .	
5,301,268	4/1994	Takeda .	
5,305,435	4/1994	Bronson .	
5,333,256	7/1994	Green et al. .	
5,339,392	8/1994	Risberg et al. .	
5,341,293	8/1994	Vertelney et al. .	
5,371,844	12/1994	Andrew et al. .	
5,371,851	12/1994	Pieper et al. .	
5,400,057	3/1995	Yin .	
5,422,993	6/1995	Fleming .	
5,442,742	8/1995	Greyson et al. .	

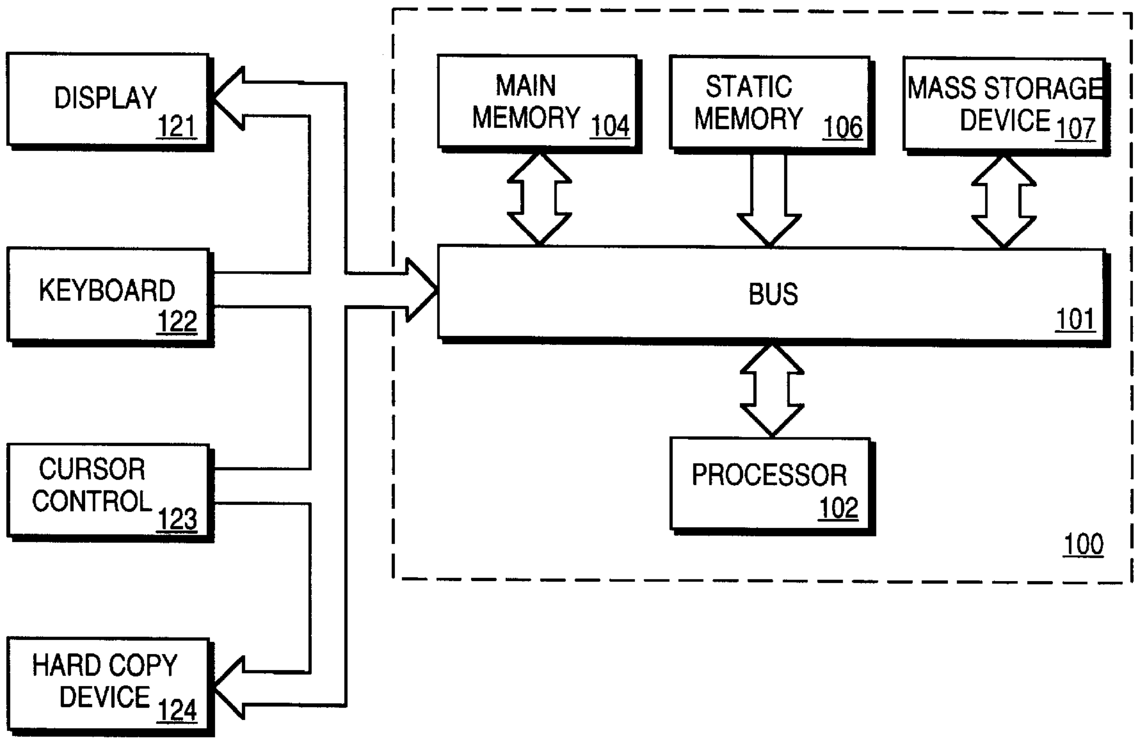
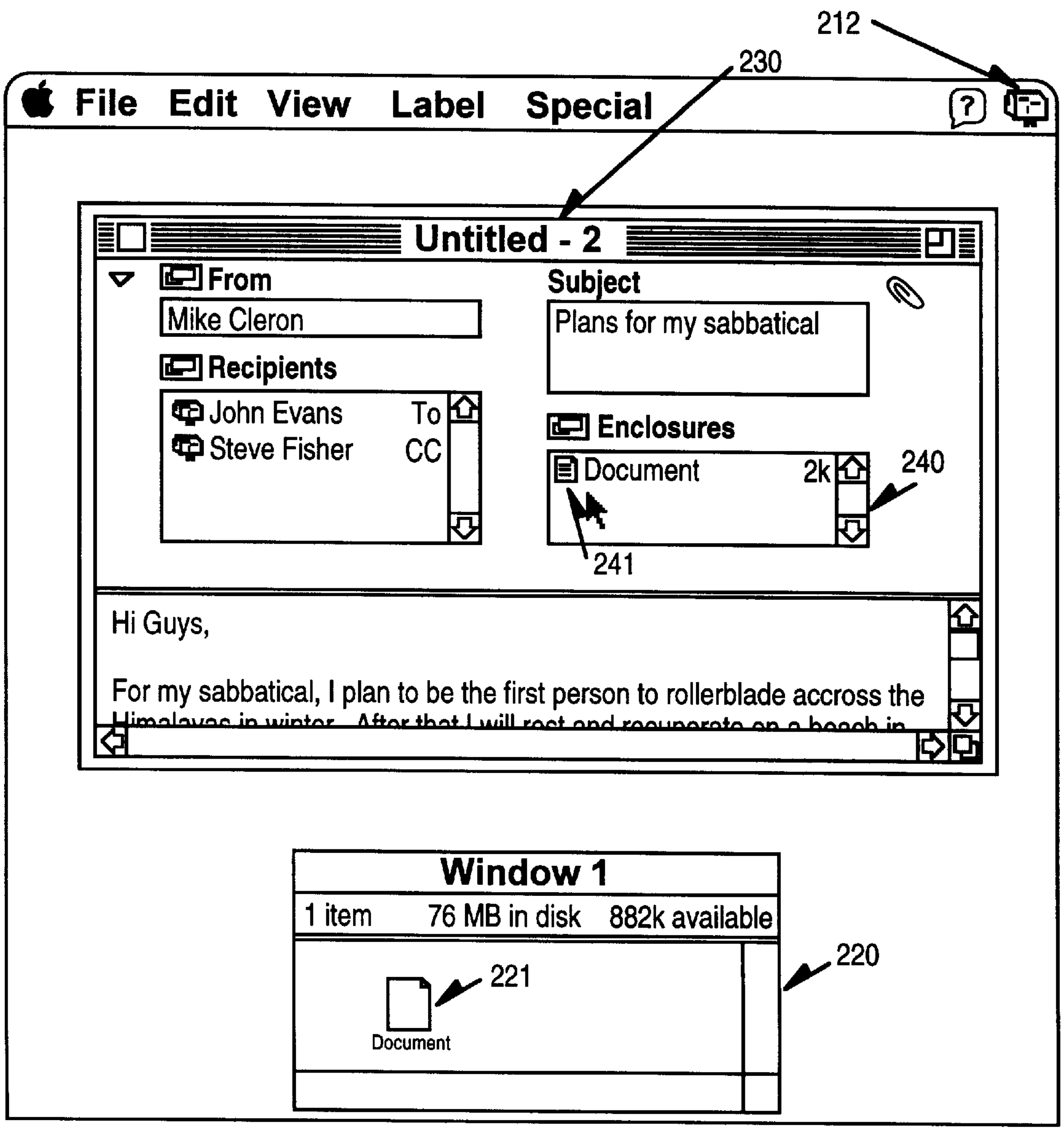

DISPLAY 121
KEYBOARD 122
CURSOR CONTROL 123
HARD COPY DEVICE 124
MAIN MEMORY 104
STATIC MEMORY 106
MASS STORAGE DEVICE 107
BUS 101
PROCESSOR 102
100

FIG. 1



Mike Cleron

Plans for my sabbatical

John Evans To
Steve Fisher CC

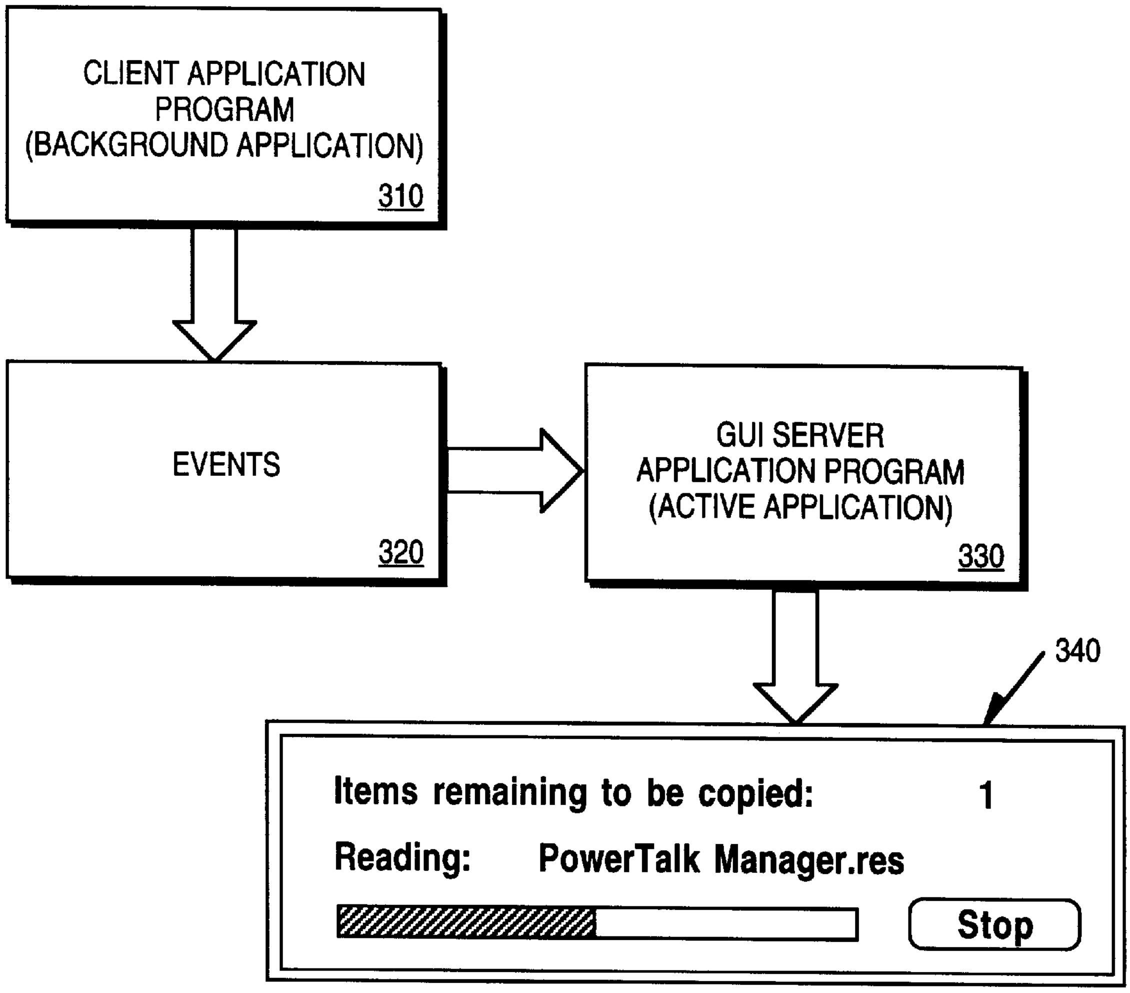
Document 2k

Hi Guys,

For my sabbatical, I plan to be the first person to rollerblade accross the

1 item 76 MB in disk 882k available

FIG. 2
(PRIOR ART)


CLIENT APPLICATION PROGRAM (BACKGROUND APPLICATION)
310
EVENTS
320
GUI SERVER APPLICATION PROGRAM (ACTIVE APPLICATION)
330
340
Stop

Items remaining to be copied: 1

Reading: PowerTalk Manager.res

FIG. 3

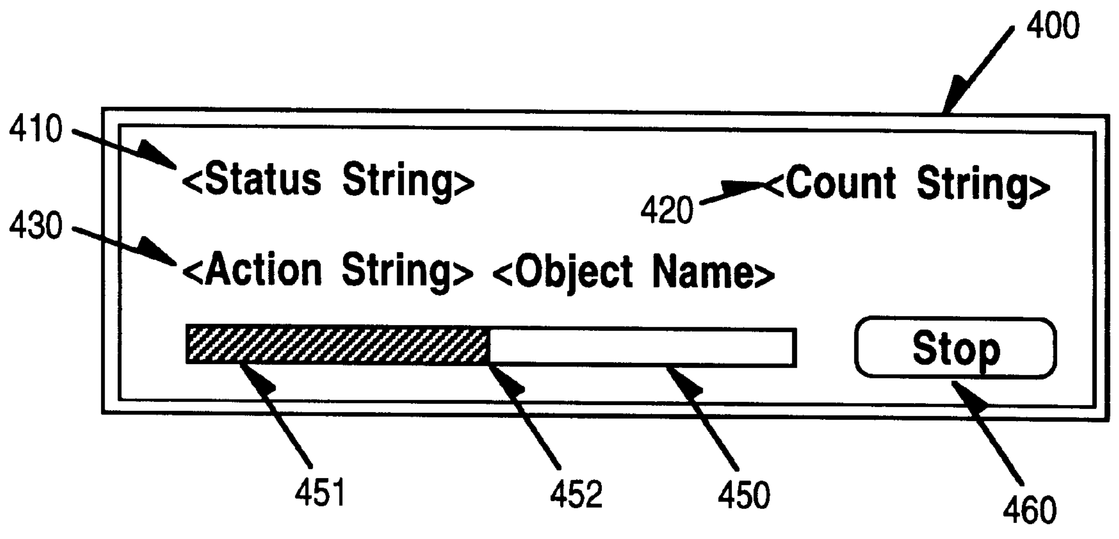
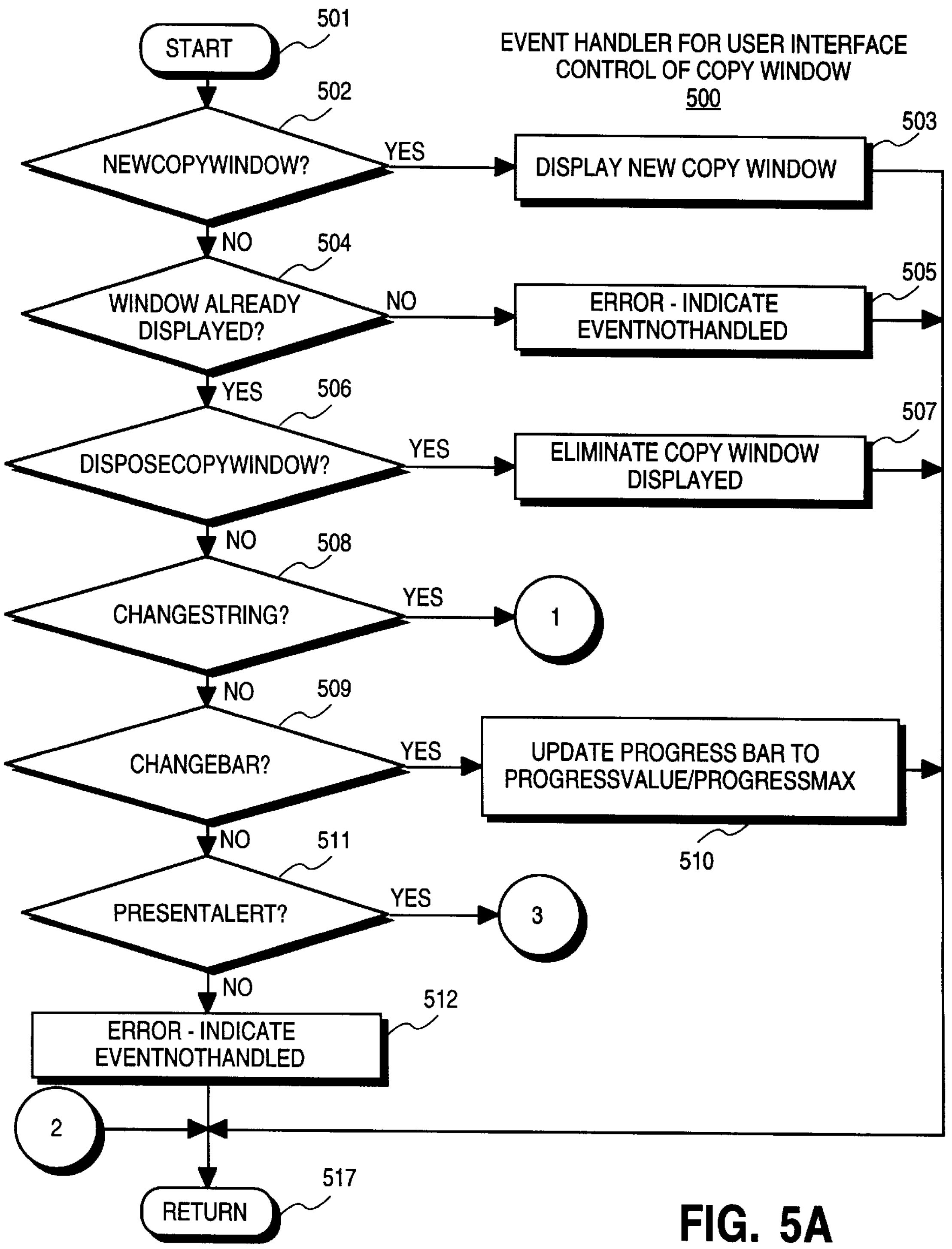


FIG. 4
(PRIOR ART)



501
START
EVENT HANDLER FOR USER INTERFACE CONTROL OF COPY WINDOW
500
502
NEWCOPYWINDOW?
YES
503
DISPLAY NEW COPY WINDOW
NO
504
WINDOW ALREADY DISPLAYED?
NO
505
ERROR - INDICATE EVENTNOTHANDLED
YES
506
DISPOSECOPYWINDOW?
YES
507
ELIMINATE COPY WINDOW DISPLAYED
NO
508
CHANGESTRING?
YES
1
NO
509
CHANGEBAR?
YES
UPDATE PROGRESS BAR TO PROGRESSVALUE/PROGRESSMAX
510
NO
511
PRESENTALERT?
YES
3
NO
512
ERROR - INDICATE EVENTNOTHANDLED
2
517
RETURN

FIG. 5A

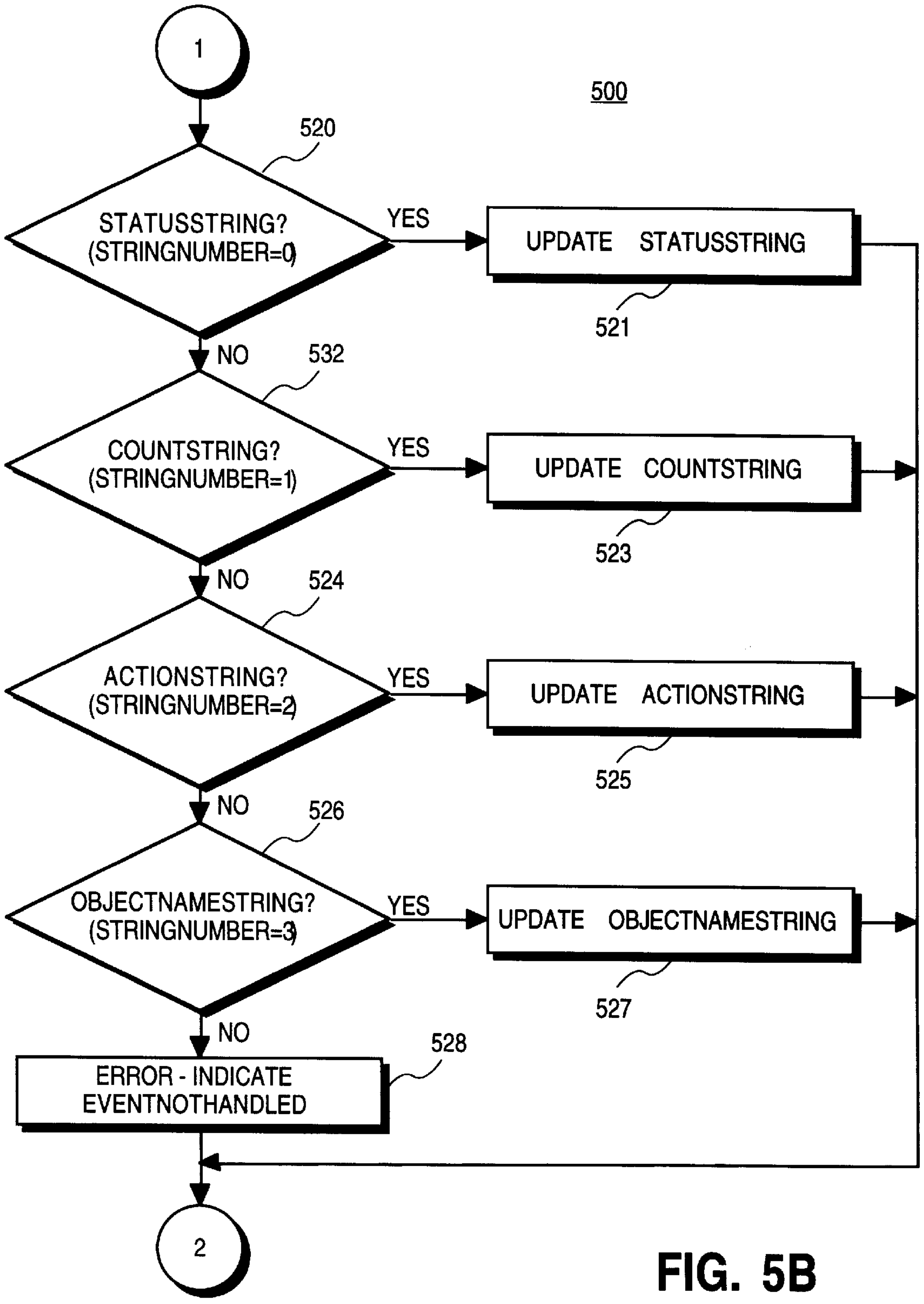

1
500
520
STATUSSTRING?
(STRINGNUMBER=0)
YES
UPDATE STATUSSTRING
521
NO
532
COUNTSTRING?
(STRINGNUMBER=1)
YES
UPDATE COUNTSTRING
523
NO
524
ACTIONSTRING?
(STRINGNUMBER=2)
YES
UPDATE ACTIONSTRING
525
NO
526
OBJECTNAMESTRING?
(STRINGNUMBER=3)
YES
UPDATE OBJECTNAMESTRING
527
NO
528
ERROR - INDICATE EVENTNOTHANDLED
2

FIG. 5B

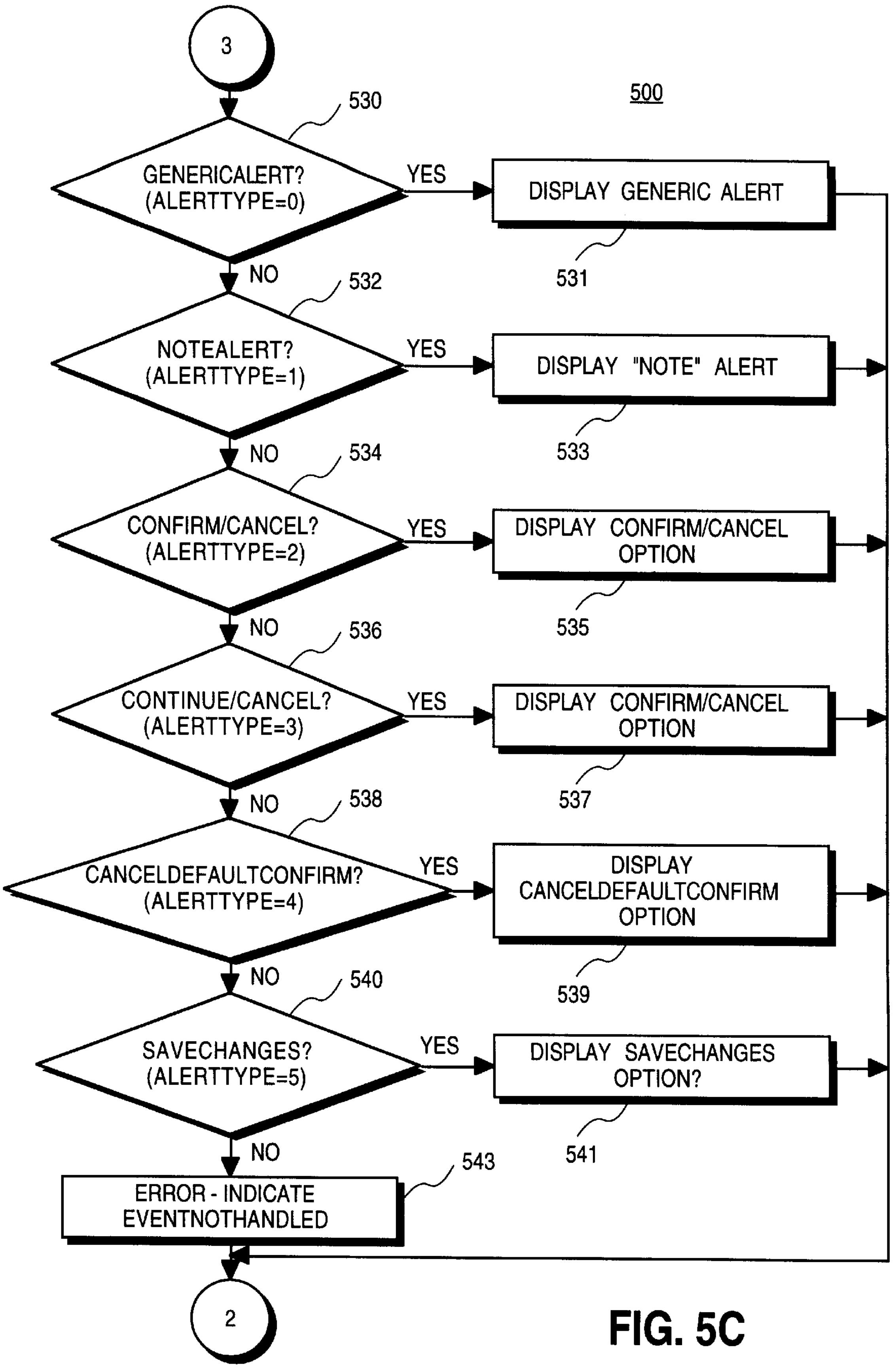

3
500
530
GENERICALERT? (ALERTTYPE=0)
YES
DISPLAY GENERIC ALERT
531
NO
532
NOTEALERT? (ALERTTYPE=1)
YES
DISPLAY "NOTE" ALERT
533
NO
534
CONFIRM/CANCEL? (ALERTTYPE=2)
YES
DISPLAY CONFIRM/CANCEL OPTION
535
NO
536
CONTINUE/CANCEL? (ALERTTYPE=3)
YES
DISPLAY CONFIRM/CANCEL OPTION
537
NO
538
CANCELDEFAULTCONFIRM? (ALERTTYPE=4)
YES
DISPLAY CANCELDEFAULTCONFIRM OPTION
539
NO
540
SAVECHANGES? (ALERTTYPE=5)
YES
DISPLAY SAVECHANGES OPTION?
541
NO
543
ERROR - INDICATE EVENTNOTHANDLED
2

FIG. 5C

MESSAGE PROTOCOL FOR CONTROLLING A USER INTERFACE FROM AN INACTIVE APPLICATION PROGRAM

This is a continuation of application Ser. No. 08/312,437, filed Sep. 26, 1994, now abandoned, which is a continuation of application Ser. No. 08/084,288, filed Jun. 28, 1993 status: abandoned.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to user interface control in a computer system. More specifically, the present invention relates to a messaging protocol which allows one application program to specify the appearance of an interface of a computer system while under control of a second application program.

2. Background Information

In multitasking operating systems, such as the Macintosh brand System 7.0 operating system available from Apple Computer, Inc. of Cupertino, Calif., typically, only one application program is given complete control of the user interface in order to prevent conflicts. There are circumstances, however, in which an application program which does not currently have control of the user interface will require that some information be presented to a user of the computer system. Typically, in the prior art, in these situations, the "inactive" application program must be "brought to the front" or made the active application (one in control of the user interface) in order for user interface control to become available to the application program. If events or other activities occur within the process that does not have control of the user interface, then the user may not be informed of the activity until after the activity has taken place, when the user brings the background application to the front. There are some circumstances in which the delay between the occurrence of the action within the background process, and the failure to provide feedback upon the computer system display may pose a substantial problem. For example, data may be overwritten, the user may wish to abort the task being performed, or he may wish to take corrective measures to otherwise address the activity occurring in the background task. There thus has arisen a need for background process to control the user interface which is currently under control of an "active" or foreground process within a computer system.

Another situation which frequently occurs is when one application program requires a complex service such as a file copying mechanism, but yet does not possess the necessary code in order to perform these tasks. An active application can use the services of the inactive application's processes without possessing the necessary code, and the inactive application program may drive the user interface of the "active" application program in order to provide feedback that the complex operation is taking place. Unfortunately, prior art techniques have no mechanism for allowing this to take place.

SUMMARY AND OBJECTS OF THE PRESENT INVENTION

One of the objects of the present invention is to allow a background application to provide user interface feedback when it is not the currently active application program.

Another of the objects of the present invention is to provide a protocol wherein a background application program may direct a foreground application program to control its user interface in a specified way.

Another of the objects of the present invention is to allow a background application program to communicate with a foreground application program for controlling the user interface of a computer system display.

Another object of the present invention is to allow a foreground application program controlling a user interface to take advantage of the services of a background application program.

Method and apparatus for a first process operative in a computer system controlling a user interface on a computer system display under control of a second process operative in the computer system. An event handler is installed for the second process, the event handler servicing events generated for controlling the user interface display under control of the second process. The first process may then perform a first set of functions in the computer system, in one embodiment, such as file management functions (e.g. copying and/or moving of files in the file system). The first process generates a first set of events for controlling the user interface display, the first set of events related to the first set of functions performed by the first process. For example, in various embodiments, feedback may be given about the progress of the file management functions (such as copying/moving specific files, reading from a source, and copying to a destination). The event handler receives the first set of events generated by the first process and updates the user interface on the computer system display according to the events generated by the first process and received by the event handler. This may include, showing the progress of the file management operation, and alerting the user of any abnormal conditions.

Other features, objects, and advantages of the present will become apparent from viewing the figures and the description below.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the figures of the accompanying in which like references indicate like elements and in which:

FIG. **1** shows an example of a computer system architecture upon which one embodiment of the present invention may be implemented.

FIG. **2** shows a situation in which a file system manipulation may be performed when the filed management task is not the "active" application program (the program controlling the user interface).

FIG. **3** shows an event-driven architecture which is used in one embodiment of the present invention for allowing a background task to control a user interface of a foreground application program.

FIG. **4** shows an example of user interface display which may be controlled directly by a foreground application program or "server" process, but which may be directed by and whose functionality may be provided for by a background process (or "client").

FIGS. **5***a*–**5***c* show process flow diagrams of an event handler which is registered for use by a server application program to service events generated by a client application program for user interface control.

DETAILED DESCRIPTION

The present invention relates to a messaging protocol between processes and a computer system wherein a first process (e.g., a client process) sends messages to a second process (e.g., a server process) so that the client process can direct the appearance of the user interface under control of the server process. In this manner, the client process performs certain functions, and the server process controls all user interface functions such as the display of feedback for those functions. For the remainder of this application, various process steps, apparatus, data structures, message formats, events, parameters, and other information will be discussed in detail, however, these are merely for illustrative purposes and are not intended to limit the present invention. It can be appreciated by one skilled in the art that many departures and modifications may be made from these specific embodiments without departing from the overall spirit and scope of the present invention.

Referring to FIG. **1**, a system upon which one embodiment of the present invention is implemented is shown as **100**. **100** comprises a bus or other communication means **101** for communicating information, and a processing means **102** coupled with bus **101** for processing information. System **100** further comprises a random access memory (RAM) or other volatile storage device **104** (referred to as main memory), coupled to bus **101** for storing information and instructions to be executed by processor **102**. Main memory **104** also may be used for storing temporary variables or other intermediate information during execution of instructions by processor **102**. Computer system **100** also comprises a read only memory (ROM) and/or other static storage device **106** coupled to bus **101** for storing static information and instructions for processor **102**, and a data storage device **107** such as a magnetic disk or optical disk and its corresponding disk drive. Data storage device **107** is coupled to bus **101** for storing information and instructions. Computer system **100** may further be coupled to a display device **121**, such as a cathode ray tube (CRT) or liquid crystal display (LCD) coupled to bus **101** for displaying information to a computer user. An alphanumeric input device **122**, including alphanumeric and other keys, may also be coupled to bus **101** for communicating information and command selections to processor **102**. An additional user input device is cursor control **123**, such as a mouse, a trackball, stylus, or cursor direction keys, coupled to bus **101** for communicating direction information and command selections to processor **102**, and for controlling cursor movement on display **121**. Another device which may be coupled to bus **101** is hard copy device **124** which may be used for printing instructions, data, or other information on a medium such as paper, film, or similar types of media. Note, also, that any or all of the components of system **100** and associated hardware may be used in various embodiments, however, it can be appreciated that any configuration of the system may be used for various purposes as the user requires.

In one embodiment, system **100** is one of the Macintosh® family of personal computers such as the Macintosh® Quadra™ or Macintosh® Performa™ brand personal computers manufactured by Apple® Computer, Inc. of Cupertino, Calif. (Apple, Macintosh, Quadra, and Performa are trademarks of Apple Computer, Inc.). Processor **102** may be one of the 68000 family of microprocessors, such as the 68030 or 68040 manufactured by Motorola, Inc. of Schaumburg, Ill.

Note that the following discussion of various embodiments discussed herein will refer specifically to a series of routines which are generated in a high-level programming language (e.g., the C++ language available from Symantec of Cupertino, Calif.) and compiled, linked, and then run as object code in system **100** during run time. It can be appreciated by one skilled in the art, however, that the following methods and apparatus may be implemented in special purpose hardware devices, such as discrete logic devices, large scale integrated circuits (LSI's), application-specific integrated circuits (ASIC's), or other specialized hardware. The description here has equal application to apparatus having similar function.

Graphical User Interface

Before discussing the preferred embodiment in detail, a brief overview of the user interface used in this system is required. A "windowing" or graphical user interface (GUI) operating environment is used wherein selections are performed using a cursor control device such as **123** shown in FIG. **1**. Typically, an item is "selected" on a computer system display such as **121** using cursor control device **123** by positioning a cursor, or other indicator, on the screen over (or in proximity to) an object on the screen and by depressing a "selection" button which is typically mounted on or near the cursor control device. The object on the screen is often an icon which has an associated file or operation which the user desires to use in some manner. In order to launch a user application program, in some circumstances, the user merely selects an area on a computer display represented as an icon by "double clicking" the area on the screen. A "double click" selection is an operation comprising, while positioning the cursor over the desired object (e.g., an icon), two rapid activations of the selection device by the user. "Pull-down" or "pop-up" menus are also used in the preferred embodiment. A pull-down or pop-up menu is a selection which is accessible by depressing the selection button when the cursor is pointing at a location on a screen such as a menu bar (typically at the top of the display), and "dragging" (moving cursor control device **123** while the selection button is depressed) until the selection the user wishes to access is reached on the pull-down menu. An item is indicated as being "selected" on a pull-down menu when the item is highlighted or displayed in "reverse video" (white text on a black background). The selection is performed by the user releasing the selection device when the selection he wishes to make is highlighted. Also, in some GUI's, as is described in the background above, the "selection" and "dragging" of items is provided to move files about in the file system or perform other system functions. These techniques include "dragging and dropping" which comprises making a "selection" of an icon at a first location, "dragging" that item across the display to a second location, and "dropping" (e.g., releasing the selection device) the item at the second location. This may cause the movement of a file to a subdirectory represented by the second location.

Note also that GUI's may incorporate other selection devices, such as a stylus or "pen" which may be interactive with a display. Thus, a user may "select" regions (e.g., an icon) of the GUI on the display by touching the stylus against the display. In this instance, such displays may be touch or light-sensitive to detect where and when the selection occurs. Such devices may thus detect screen position and the selection as a single operation instead of the "point (i.e., position) and click (e.g., depress button)," as in a system incorporating a mouse or trackball. Such a system may also lack a keyboard such as **122** wherein the input of text is provided via the stylus as a writing instrument (like a pen) and the user handwritten text is interpreted using handwriting recognition techniques. These types of systems may also benefit from the improved manipulation and user feedback described herein.

One problem solved by the present invention is illustrated with reference to FIG. **2**. Window **200** of FIG. **2** illustrates a typical Macintosh user interface display while one application program, such as an electronic mail program, controls the user interface display. This is illustrated by the icon present in the upper right-hand portion **212** of the display **200**. The operating system only allows a single application program to control the user interface at any given time. However, other application programs such as, those performing file and/or program management functions (e.g., the "Finder" within the Macintosh brand operating system), allow the launching of applications, programs, and the movement of files within the file system. In one embodiment of the present invention, a user may decide to "enclose" a file represented by icon **221** in the file system with the mail message represented on window **230**. The application program controlling window **230** does not possess file transfer or file transfer feedback capabilities, whereas the File System Manager (known as the "Finder" in the Macintosh) does possess these capabilities. Therefore, it is desired that the application program controlling window **230** have certain functions and user interface capabilities of the Finder. In a typical prior art systems, feedback is provided by the file management function to show that the file movement or copy operation is taking place. This typically takes the form of a progress bar on a typical prior art user interface to show the progression of the file transfer operation as it takes place in the file system. For example, if a plurality of files are moved in the file system, or copied from one media device to another, file names showing each transfer of each file, and a darkened representation is shown in the progress bar to show overall completion of the copying of the files is represented on the progress bar. This will be illustrated in more detail below.

Event-Driven Architecture

An operation such as copying or moving files from one location to another in the file system is a nontrivial task. In this embodiment, the application program controlling the user interface defers to the background task the "Finder" so that it may perform the file management functions for transfer and/or copy of the file(s). For example, several tasks need to be performed by the copy/movement process prior to copying or moving the files. For example, the destination subdirectory or other file location needs to be scanned to determine whether any of the transferred file names are equivalent to those already in the subdirectory. If so, the user needs to be alerted in order to determine whether he wishes to overwrite the existing files at that location or, perhaps, use a different name. In another instance, there may not be sufficient space at the destination to which the files are being moved for writing the files. In this instance, the user is alerted that there was not sufficient space on the storage medium to store the files, is informed that the operation was not successful, and any file(s) already written or other intermediate file information can be deleted. However, in a situation such as that illustrated in FIG. **2**, the underlying file management process cannot present this user feedback or alert information to the user because it does not presently have control of the user interface. The process having control of window **230**, an electronic mail application program, is currently in control of the user interface. Thus, an improved means for allowing the background process (e.g., the Finder) to control the user interface is used in various embodiments of this invention to present feedback to the user regarding the underlying functions that are taking place. This is performed via interprocess communication, in the Macintosh brand operating system, using Events and the accompanying operating system Event Manager and Apple

Event Manager available from Apple Computer of Cupertino, Calif.

Interprocess communication is an important aspect of modern computer system design. For example, such inter-application communication has provided in modern computer systems, such as the Macintosh brand computer's operating System 7.0, available from Apple Computer of Cupertino Calif., through a mechanism known as the Event Manager. The Event Manager and the Apple Event Manager are used for handling a wide variety of functions within application programs such as detecting user actions—key clicks, selections using a cursor control device, the detection of the insertion of disks into a disk drive on the computer system, opening files, closing files, or other actions within the computer system. Typically, processes running within a Macintosh brand computer system comprise a main program loop known as an "event" loop which detect the occurrence of these events in the system. Then, the application typically branches to portions of the program to allow the event to be serviced. Such an event driven architecture forms the core of many application programs within many different types of computers and operating systems in present use but, in this embodiment, resides in a system such as **100** described above.

Various embodiments of the present invention use the Event Manager and the Apple Event Manager supplied by Apple Computer for interprocess communication between applications programs which are operative within computer system **100** during run time to implement the features described herein. The event driven architecture for message passing between a first application program (e.g., a client application program which is operative in the background), and a second application program (e.g., a server application program which is the active application controlling the user interface), is illustrated with reference to FIG. **3**. For example, using the event driven architecture specified in *Inside Macintosh, Volume* 6, pages 5–1 through 6–118, "client" application program **310** communicates with the Graphical User Interface (GUI) "server" application program or active application program **330** via events **320**. Each of these events are generated by the client application program **310** and are detected by the Apple Event Manager. GUI server application **330** registers a process with the Apple Event Manager known as an Event Handler so that whenever defined events are detected, the Apple Event Manager forwards the event(s) to the registered handler(s) and cause the handler(s) to be invoked and service the events. At that point, the handler may determine the appropriate action to take place. In various embodiments of the present invention described herein, the registered event handler for GUI server **330** will cause the activation and modification of a user interface display, in this case, copy window **340** illustrated in FIG. **3**. Upon the launching of the GUI server application program **330**, or any application program which may become an active application program during specified user actions (e.g., the copying of file(s), the server application program will register with the Apple Event Manager the handler(s) which are used to service the events, including those generated by any potential client application programs (e.g., **310** of FIG. **3**). In the situation where a defined event is not serviced by any handler which are registered by the application program, then, a default handler supplied by the operating system is instead used for servicing the events.

For the remainder of this application, in the embodiment discussed herein, it will be assumed that the "server" (e.g., **330** of FIG. **3**) has registered a handler which will service user interface events. It will also be assumed that a client program (e.g., **310** of FIG. **3**) provides the underlying functionality for performing the actions represented by the user interface (e.g., copying files), which occurs upon the inactive program detecting that a file should be copied (or moved) from one directory to another, such as a directory for "Enclosures" within an electronic mail application program. This function may also be requested by server process **330** sending an event to a handler registered for client process **310**, such as "CopyFile" 'File 1' to 'Enclosures.'" The mechanics of this operation, however, will not be described in detail because they are beyond the scope of the present invention.

Client to Server Events for Controlling the User Interface

The following events are defined in this new protocol for communicating from client **310** to user interface server **330**:

1. NewCopyWindow;
2. DisposeCopyWindow;
3. ChangeString;
4. ChangeBar; and
5. PresentAlert.

As client application program **310**'s file copy operations progress, certain of these events are issued by client process **310** to server process **330**. Moreover, communication is provided via another set of events from server **330** to client **310** to indicate the success of the action indicated by the events, and user interface feedback in response to information presented on the user interface by server **330**. A description of each of the specific events and the parameters used in each of these events will now be discussed with reference to FIG. **4**.

400 of FIG. **4** illustrates a typical copy window which is displayed during a file copying operation well known in the prior art. However, each of the informative portions of the window are displayed with corresponding parameter name for the event in angled brackets (e.g., <status string> **410**, <action string> **430**, <count string> **420**, and <object name> **440**), which is replaced by the strings specified within parameters associated with the event(s) issued by client process **310**. The events and parameters associated with the events will now be discussed.

NewCopyWindow

The NewCopyWindow event is signaled by client application program **310** to indicate that a new copy window (e.g., **400** in FIG. **4**) should be displayed. Using typical prior art user interface commands, the server application program's handler creates upon the display screen a copy window **400** as is illustrated in FIG. **4** with the appropriate strings specified in the event parameters. Each of the event parameters for the NewCopyWindow are specified in the following order:

1. actionString (e.g., "reading"/"writing"/"verifying")
2. objectName (name of object being copied)
3. statusString (e.g., "Preparing To Copy," "Items remaining," etc.)
4. countString (how many items are being copied)
5. progressValue (an initial value, usually 0)
6. progressMax (a maximum value)

actionString is used for specifying the operation being performed. In typical prior art copy operations, the value thus is one of the following three strings: "Reading"; "Writing"; or "Verifying", for specifying the operation being performed. The specified action string is placed into region

430 of the copy window **400** upon detection of the NewCopyWindow event.

objectName is used for specifying the string which will appear at region **440** on copy window **400**. It is used for specifying the name of the object or file being copied. Feedback can thus provide to the user which object is currently in the process of being read, written, or verified.

statusString **410** is used to specify the intermediate status of the copy operation taking place. For example, in certain prior art systems, this string may read "Preparing to Copy," "Items Remaining," etc. The status string indicates to the user the current status of the copy operation taking place.

countString is used for specifying the value which is shown at region **420** of display **400**, such as the number of items (e.g., files or bytes) which are being copied. Thus, in one situation, this string may contain "120 bytes" when a file or file(s) of 120 bytes in length are being copied.

progressvalue and progressMax are used for specifying the status of progress bar **450**. For example, the progressMax parameter is used for specifying some maximum value at which the progress bar will be completely darken. In an instance where a total of 120 bytes are being copied, progressMax may be equal to an integer value, such as 120. The progressValue parameter will thus be used to specify an intermediate value from some initial value (in one embodiment, the integer 0) to the progressMax value. Thus, on the display, progress bar **450** may be filled with a darkened region **451** up to an intermediate position **452** based upon the fraction progressValue/progressMax. For example, if progressValue equals 60 and progressMax equals 120, then progress bar **450** will have a representation such as that shown in FIG. **4** wherein $^{60}/_{120}$ or ½ of the progress bar has been darkened. Feedback is thus provided to the user to illustrate the current completion of the copy operation. In typical situations, the progressvalue will have an initial value such as 0, and the progressMax value will be some nonzero value, for example, equivalent to an integer representing the maximum number of bytes to be copied.

DisposeCopyWindow

The DisposeCopyWindow event is used for indicating the termination of a copy operation. This event causes the server's handler to remove window **400** from the display using well known prior art interface techniques. The event has no parameters because it merely removes from the display the currently displayed progress bar window.

ChangeString

The ChangeString event has the following parameters:

```
stringNumber
        statusString = 0
        countString = 1
        actionString = 2
        objectNameString = 3
stringValue
```

stringNumber—For each of the above specified values of stringNumber, the identified string modified in copy window **400** using the ChangeString event according to the string contained in stringValue. statusString **410**, countString **420**, actionString **430**, or objectNameString **440** may be modified within, copy window **400** illustrated in FIG. **4**. The client application may thus cause updates to be performed within copy window **400** so that the user is informed of the current status of the copy operation (such as a current file being copied)

stringValue is a string which will replace the string specified by the integer value contained in stringNumber.

ChangeBar

The ChangeBar event has the following parameters:

progressValue (a current value)

progressMax (a maximum value) Each of these parameters are integer values specifying the current progression and the maximum progression of progress bar **450** of copy window **400**, is discussed with reference to the NewCopyWindow event above. The progress bar is thus adjusted to have a filled in representation, such as that shown as **451** according to the fraction progressValue/progressMax. The progress bar is update by the server's handler using standard prior art user interface commands.

PresentAlert

The following parameters are defined for the PresentAlert event:

```
alertType
    GenericAlert = 0
    NoteAlert = 1
    Confirm/Cancel = 2
    Continue/Cancel = 3
    CancelDefaultConfirm = 4
    SaveChanges = 5
alertString
```

alertType is an integer value which is used for specifying the type of alert displayed which is displayed to the user. Note that these alerts are all similar to those which are displayed in typical prior art copy operations upon detection of certain conditions, abnormal or otherwise. Some of these specified alerts require that the user respond. For example, the Confirm/Cancel alert displays a window which requests that the user "confirm" or "cancel" an ongoing operation. For example, the Confirm/Cancel alert may be used when the same file name is detected at a destination directory for a file which is being copied. Any of the standard alert windows which may be used in certain prior art copy operations may be specified using the proper alertType integer value. User responses to alerts will be provided with events from user interface server **330** to client **310** via another set of events discussed below.

alertString is a string value indicating the associated message to be associated with the alert. For example, the client application program may determine that a file name having an equivalent file name to a file being copied already resides at the destination application. In this event, the alertString may contain a message such as "File 'My File' already exists. Replace?" In any event, using the foregoing alert parameters, the PresentAlert event may specify appropriate alerts to the user.

Server to Client Events

All of the above events are shown for illustrative purposes only and are for the client application process **310** (e.g., the "Finder" performing the copy operation) alerting server process **330** (e.g., an electronic mail application program) to change the user interface display in a specified manner. However, other events may also be defined to specify other changes to the user interface display and for other operations, especially in instances where user interface response(s) to alerts are required. Because the background process (e.g., client process **310**) cannot detect user interface actions (such as selections on the display), the handler for server process **330** must detect these actions and transmit response event messages to client process **310**. In this case, client **310** will have its own event handler registered for servicing events issued by server **330** to client **310**. For example, if the user wishes to "cancel" an operation, an event entitled "CopyCancel" may be issued to the client process **310** in one embodiment of the present invention so that any ongoing operation(s) may be aborted. This is detected by a user selecting "Stop" button **460** at any time during the operation. The cancel operation may be detected by server **330** by the detection of a "mouseDown" event at a specific location, such as "Stop" button **460**, or its keyboard equivalent (e.g., a command period combination in the Macintosh). In this case, client application **310**'s handler may be alerted that an abort was indicated and take appropriate action.

In another embodiment, responses to alerts such as file overwrite messages may be sent from server **330** to client **310**. In this instance, an integer may be passed as a parameter wherein one value of the integer (e.g., 0) causes the operation to be aborted or a second value (e.g., 1) causes the file overwrite to be confirmed. Responses to alerts are performed using other defined events from GUI server **330** to client **310**. The user may be presented with the option of either confirming replacement of the file or canceling the copy operation. In one embodiment, when an alert is presented, client **310** remains idle until a response is made by a user on the user interface display. Then a corresponding response event (e.g., AlertReply) with a response parameter (e.g., an integer value Result containing an integer 0 indicating confirmation of the operation or integer 1 indicating canceling of the operation) is generated by server **330** to client **310** to either confirm or cancel the operation being performed by client **310**.

Other user responses to queries, such as alerts, errors, or other conditions, may also be responded to in this manner, as detected by GUI server **330** and sent to client **310** via a registered handler.

Event Handling by Server Application Program

FIGS. **5***a*–**5***g* show a process flow diagram of a typical event handler which may be registered by server application program **330** and service events generated by a background process for controlling the user interface using the messaging protocol of one embodiment of the present invention. For example, such an event handler may be registered using the Apple Event Manager described in *Inside Macintosh, Volume VI*, Chapter 6, wherein all of the above-described events are handled by this single handler **500**. Handler **500** will typically have a process entry point, such as **501** illustrated in FIG. **5***a*, and comprise an IF or CASE programming statement in a typical high level programming language or other condition checking loop, which is illustrated in the remainder of the figures. Then, each of the events may be checked for, and upon detection of a specific event, the user interface display specified by the event and associated parameters may be displayed upon system display **121**. For example, it will be determined at step **502** whether a NewCopyWindow event has been detected. If so, then a new copy window (e.g., **400**) is displayed at step **503** with the specified parameters, such as statusString **410**, countString **420**, actionString **430**, objectName **440** , and having the progress bar **450** with an appropriate representation as defined by the progressValue/progressMax parameters passed within the event. Upon display of the new copy window **400** at step **503**, process **500** continues and returns at step **517**.

If, however, a NewCopyWindow event was not detected at step **502**, and a window is not currently displayed as detected at step **504**, then an error condition may be indicated at step **505**, such as by issuing an event from server **330** to client **310** to specify an error (e.g., "EventNotHandled") to specify that the event was not serviced. Then, event handler **500** may exit at step **517**. If, however, a window has already been displayed, then the condition for the various remaining events defined in the messaging protocol may be checked for using a suitable programming construct such as a CASE statement or other similar condition-checking statement(s). For example, it is determined at step **506** whether the DisposeCopyWindow event has been detected. If so, then the copy window displayed upon computer system display **121** is eliminated at step **507** using well-known prior art user interface operations, and handler **500** returns at step **517**.

Upon the detection of a ChangeString event, as detected at step **508**, then process **500** proceeds to a more detailed sequence of steps to determine the value passed within the string number, as reflected on Figure **5***b*. As is illustrated in FIG. **5***b*, it is determined using a condition checking loop, such as a CASE statement or other programming construct, what the value of stringNumber is. For example, at step **520**, it is determined whether stringNumber indicates that the statusString should be modified. If so, then statusString **410** on display **400** is updated at step **521**, and the handler returns at step **517**. If, however, the countString is specified at step **522** (when stringNumber=1), then the count string (e.g., **420**) is updated at step **523**, and the handler returns at step **517**. If, however, the actionString should be modified (stringNumber=2), as detected at step **524**, then it is updated on the copy window. If, however, stringNumber=3 indicating that objectNameString **440** is sought to be updated, as detected at step **526**, then objectNameString **440** is updated at step **527**, and handler **500** returns at step **517** of FIG **5***a* . Any other string number results in an error being generated at step **528**, and a return from the handler at step **517** with an appropriate error event message to client **310**, such as "EventNotHandled," indicating that the handler did not service the event.

Process **500** of FIG. **5***a* proceeds to step **509** if a ChangeString event was not detected. Step **509** determines whether the ChangeBar event has been detected. If so, then handler **500** proceeds to step **510** which updates progress bar **450** using the progressvalue and progressMax parameters passed in the event. If the value(s) passed are invalid, an error may be indicated via a response event and the update to the progress bar abort.

If, however, the ChangeBar event is not detected at step **509**, then the handler proceeds to determine whether a PresentAlert event has been detected at step **511**. Various types of alerts can then be checked for, as illustrated in FIG **5***c*, by checking the alertType parameter. For example, each of the steps illustrated at steps **530**–**540** on FIG. **5***c* may be conformed using a typical high-level programming construct such as a CASE statement. Then, upon detection of the corresponding value in the alertType parameter, the associated alert is displayed. For instance, for alertType=0, as detected at step **530**, the generic alert is tested for and then displayed at step **531**. At step **532**, the NoteAlert is tested for (with alertType=1) and displayed at step **533**. At step **534**, the Confirm/Cancel alert is tested for (with the alertType=2), and it is displayed if detected at step **535**. At step **536**, if the Continue/Cancel alert type is detected (alertType=3), then the Confirm/Cancel option window is displayed at step **537**. At step **538**, the CancelDef aultConfirm option is tested for (alertType=4), and the corresponding option is displayed at step **539**. Finally, the SaveChanges parameter is tested for (alertType=5) at step **540** and then displayed at step **541**. If any other alertType value is detected, an error is indicated at step **543**, and the process returns at step **517** of FIG. **5***a*. Otherwise, upon completion of detection of any of the above alertType values, then the option previously displayed is saved for use when the next event is detected in the event handler at step **542**, and process **500** returns at step **517** of FIG. **5***b*.

At any rate, upon detection of the all the previous events, if the events are serviced, then an event message from server **330** to client **310** such as EventHandled is issued, and handler **500** returns at step **517**. Otherwise, any events detected which do not fall into one of the categories tested for or other events which are not serviced may issue a suitable error event message, such as EventNotHandled at step **517** to indicate to client **310** that the event was not serviced. Then, the client may take appropriate actions via its own event handler.

Thus, an invention for a background application controlling the user interface of a foreground application has been described. Although the present invention has been described particularly with reference to specific embodiments as illustrated in FIGS. **1**–**5***c*, it may be appreciated by one skilled in the art that many departures and modifications may be made by one of ordinary skill in the art without departing from the general spirit and scope of the present invention.

What is claimed is:

1. In a computer system comprising a processor, a display, a memory, a user input device, a first process operative in the computer system, a second process operative in the computer system as a foreground process and a user interface on said computer system display under the control of the second process, a method for the first process to perform operations for the second process and control a content of the user interface on said computer system display, said content under control of the foreground second process operative in said computer system, said first process controlling the content to display information regarding the operations performed by the first process for the second process, said method comprising the following steps:

a. installing an event handling process as part of said second process, said event handling process when said second process is operative in said computer system, servicing events generated by the first process for controlling said user interface display under control of said second process;

b. said second process initiating said first process to perform operations for said second process, said second process operative in the foreground and said first process operative in the background;

d. said first process generating events for controlling said user interface display while the second process remains as a foreground process and the first process is a background process, said events providing information regarding the operations performed by said first process for the second process; and

e. said event handling process receiving events generated by said first process, said event handling process updating said user interface on said computer system display according to said events generated by said first process, while said first process remains in the background, and received by said event handling process.

* * * * *

EXHIBIT 8

US006275983B1

(12) **United States Patent**
Orton et al.

(10) **Patent No.: US 6,275,983 B1**
(45) **Date of Patent: *Aug. 14, 2001**

(54) OBJECT-ORIENTED OPERATING SYSTEM

(75) Inventors: **Debra Lyn Orton**, San Jose; **Eugenie Lee Bolton**, Sunnyvale; **Daniel F. Chernikoff**, Palo Alto; **David Brook Goldsmith**, Los Gatos; **Christopher P. Moeller**, Los Altos, all of CA (US)

(73) Assignee: **Object Technology Licensing Corp.**, Cupertino, CA (US)

(21) Appl. No.: **09/140,523**

(22) Filed: **Aug. 26, 1998**

Related U.S. Application Data

(63) Continuation of application No. 08/521,085, filed on Aug. 29, 1995, now abandoned.

(51) **Int. Cl.**^{7} **G06F 9/45**
(52) **U.S. Cl.** **717/5**
(58) **Field of Search** 395/705; 709/304; 717/5

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,456,954	6/1984	Bullions et al.	364/200
4,530,052	7/1985	King et al.	364/200
4,722,048	1/1988	Hirsch et al.	395/650
4,821,220	4/1989	Duisberg	364/578
4,885,717	12/1989	Beck et al.	364/900

(List continued on next page.)

OTHER PUBLICATIONS

Schmidt, D., System Programming with C++ Wrappers, C++ Report, Sep./Oct. 1992, 1–7.*
Julin, D.P. et al., "Generalized Emulation Services for Mach 3,0—Overview, Experiences and Current Status," USENIX Association—Mach Symposium Proceedings, Nov. 1991, pp. 13–26.
Malan, G. et al., "DDS as a Mach 3.0 Application," USENIX Assoc.—Mach Symposium Proceedings, Nov. 1991, pp. 27–40.

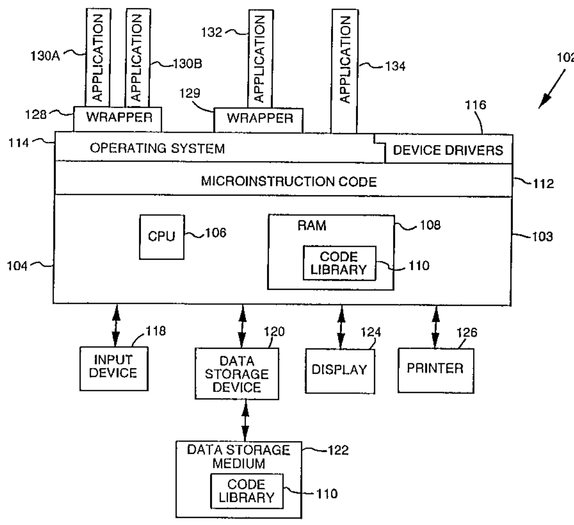
(List continued on next page.)

Primary Examiner—Kakali Chaki
Assistant Examiner—John Q. Chavis
(74) *Attorney, Agent, or Firm*—Morgan & Finnegan LLP

(57) **ABSTRACT**

An apparatus for enabling an object-oriented application to access in an object-oriented manner a procedural operating system having a native procedural interface is disclosed. The apparatus includes a computer and a memory component in the computer. A code library is stored in the memory component. The code library includes computer program logic implementing an object-oriented class library. The object-oriented class library comprises related object-oriented classes for enabling the application to access in an object-oriented manner services provided by the operating system. The object-oriented classes include methods for accessing the operating system services using procedural function calls compatible with the native procedural interface of the operating system. The computer processes object-oriented statements contained in the application and defined by the class library by executing methods from the class library corresponding to the object-oriented statements.

22 Claims, 17 Drawing Sheets



U.S. PATENT DOCUMENTS

4,891,630		1/1990	Friedman et al.	340/706
4,926,322		5/1990	Stimac et al.	364/200
4,953,080		8/1990	Dysart et al.	364/200
4,974,159		11/1990	Hargrove et al.	364/200
5,041,992		8/1991	Cunningham et al.	364/518
5,050,090		9/1991	Golub et al.	364/478
5,060,276		10/1991	Morris et al.	382/8
5,075,848		12/1991	Lai et al.	395/425
5,093,914		3/1992	Coplien et al.	395/700
5,119,475		6/1992	Smith et al.	395/156
5,125,091		6/1992	Staas, Jr. et al.	395/650
5,133,075		7/1992	Risch	395/800
5,136,705		8/1992	Stubbs et al.	395/575
5,136,711		8/1992	Huggard et al.	395/700
5,151,987		9/1992	Abraham et al.	395/575
5,179,703		1/1993	Evans	395/700
5,181,162		1/1993	Smith et al.	364/419
5,237,669		8/1993	Spear et al.	395/400
5,247,681	*	9/1993	Janis et al.	709/305
5,274,821		12/1993	Rouquie	395/700
5,280,610		1/1994	Travis, Jr. et al.	395/600
5,287,507		2/1994	Hamilton et al.	395/650
5,293,385		3/1994	Hary	371/19
5,297,284		3/1994	Jones et al.	395/700
5,313,636		5/1994	Noble et al.	395/700
5,315,703		5/1994	Matheny et al.	395/164
5,315,709		5/1994	Alston, Jr. et al.	395/600
5,317,741		5/1994	Schwanke	395/700
5,321,841		6/1994	East et al.	395/725
5,325,481		6/1994	Hunt	395/159
5,325,522		6/1994	Vaughn	395/600
5,325,524		6/1994	Black	395/600
5,325,533		6/1994	McInerney et al.	395/700
5,327,562		7/1994	Adcock	395/600
5,339,422		8/1994	Brender et al.	395/700
5,339,430	*	8/1994	Lyndin et al.	709/305
5,339,438		8/1994	Conner et al.	395/700
5,341,478		8/1994	Travis, Jr. et al.	395/200
5,361,350		11/1994	Conner et al.	395/600
5,361,358		11/1994	Cox et al.	395/700
5,369,766	*	11/1994	Nakano et al.	717/5
5,379,432	*	1/1995	Orton et al.	709/303
5,404,529		4/1995	Chernikoff et al.	395/700
5,446,902	*	8/1995	Islam	395/703
5,475,845		12/1995	Orton et al.	395/682
5,555,418	*	9/1996	Nilsson et al.	709/305
5,752,034	*	5/1998	Srivastava et al.	395/704
5,901,313	*	5/1999	Wolf et al.	345/335

OTHER PUBLICATIONS

Rashid, R., "A Catalyst for Open Systems," *Datamation,* May 15, 1988, p. 32–33.

Guedes, Paulo, "O–O Interfaces in the Mach 3.0 Multi–Server System," IEEE 1991.

Rashid, R. "Mach: A Foundation for Open Systems," Carnegie Mellon University, IEEE 1989.

Foley, M.J., "Taligent, IBM draw closer," PC Week, Feb. 21, 1994, vol. 11, No. 7, p. 8.

Franz, M., "Emulating an Operating System on Top of Another," 1993 by John Wiley & Sons. Ltd.

IBM C/C++Tools: 'User Interface Class Library Reference' May 1993, IBM Part No.: 61G1179, Denmark. See p. 464, lines 1–11.

Petzold, Charles, "Intro to OS/2 Function Calls," *PC Magazine,* vol. 6, No. 18, Oct. 27, 1987, NY, US, pp. 375–380.

Breisacher, Lee, "Smalltalk/V Presentation Mgr.," OS/2 Notebook: *The Best of IBM Personal Systems Developer,* (Dick Conklin, General Editor), 1990, Microsoft Press, Redmond, US, pp. 226–232.

IBM C/C++Tools, "Programming Guide," Mar. 1993, IBM Part No.: 61G1181, Denmark. See p. 35, line 1; see p. 438, lines 1–6.

Bernabeau–Auban et al., "Clouds—A Distributed Object–Based Operating System Architecture and Kernel Implementation," *New Directions for Unix. Proc. Autumn 1988 EUUG Conf.* Oct. 3, 1988, Cascais, Porugal, pp. 25–37.

Daponte et al., "Object–Orineted Design of Measurement Systems," *Conference Record of IEEE Instrumentation and Measurement Technology Conference,* May 12, 1992, New York, pp. 243–248.

Dohlberg, S, "Galaxy from Visix," Open Information Systems, vol. 7, No. 10, Oct. 1992, pp. 1–16.

Hruschka, "Towards an Object Oriented Method for System Architecture Design," Proceeding of the 1990 IEEE International Conference on Computer Systems and Software Engineering, COMPEURO '90, May 8, 1990, Tel Aviv, pp. 12–17.

McCormack et al., "Using the Toolkit or How to Write a Widget," *Proc. Of the Summer* 1988 USENIX Conf., Jun. 20, 1988, San Francisco, pp. 1–13.

Musser, John, "Extending streambufs: class logstrub" C++ *Report,* 4(3), Mar. 1992, pp. 51–55.

Schmidt, Doug, "Systems Programming with C++ Wrappers: Encapsulating IPC Services," C++ *Report,* 4(8), Oct. 1992, pp. 50–54.

Schmidt, Doug, "An Object–Orients Interface to IPC Services," C++ Report,4(9), Nov. 1992, pp. 48–54.

Schmidt, Doug, "Encapsulating Operating Systems IPCs: An Object–Oriented Interface for Event–Driven UNIX I/O Multiplexing," C++ *Report,* 5(2), Feb. 1993, pp. 43–50.

* cited by examiner

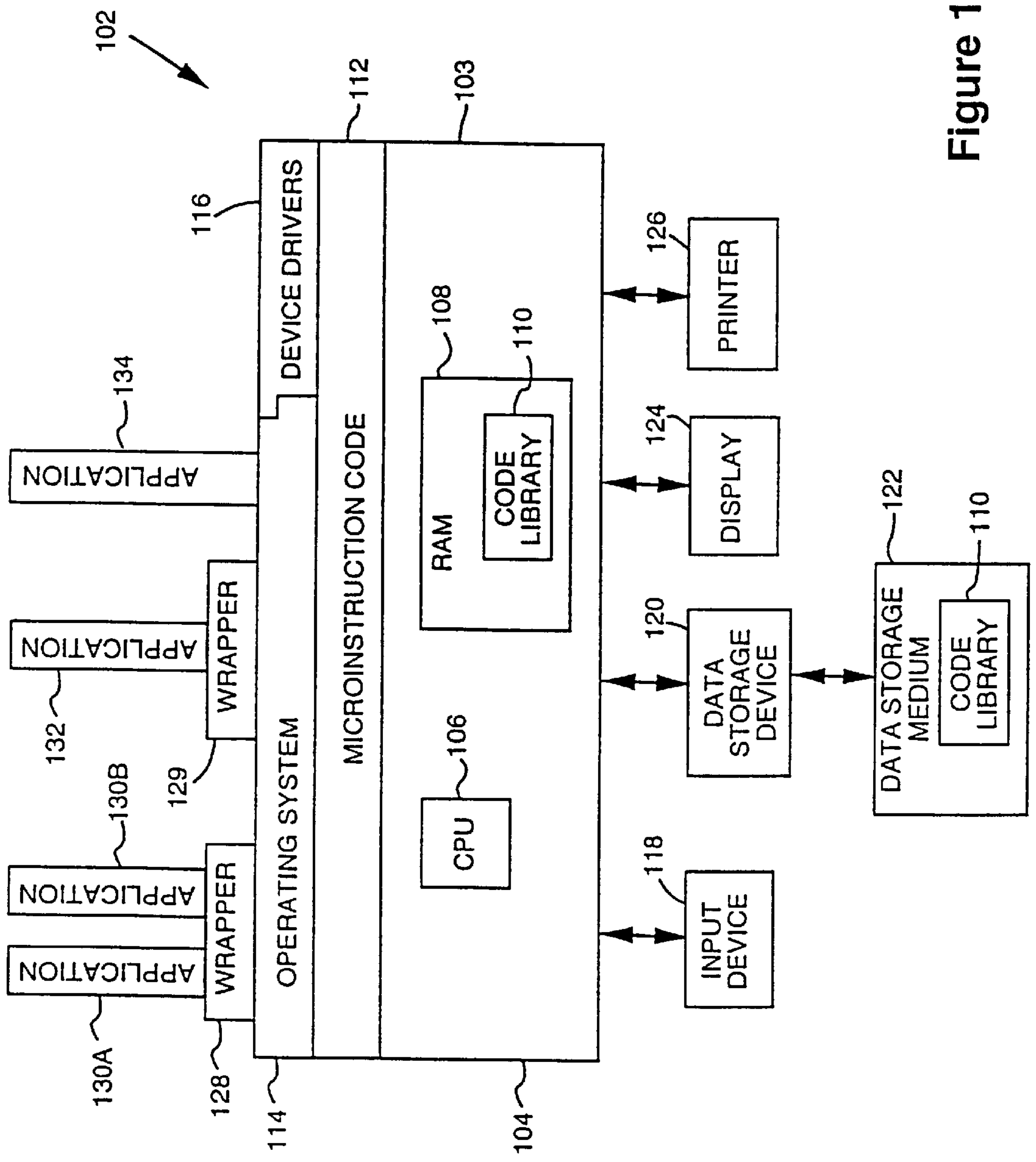

102
112
103
116
DEVICE DRIVERS
134
APPLICATION
108
110
RAM
CODE LIBRARY
MICROINSTRUCTION CODE
126
PRINTER
124
DISPLAY
WRAPPER
APPLICATION
132
129
OPERATING SYSTEM
122
110
120
DATA STORAGE DEVICE
DATA STORAGE MEDIUM
CODE LIBRARY
106
CPU
130B
APPLICATION
WRAPPER
APPLICATION
130A
128
114
104
118
INPUT DEVICE

Figure 1

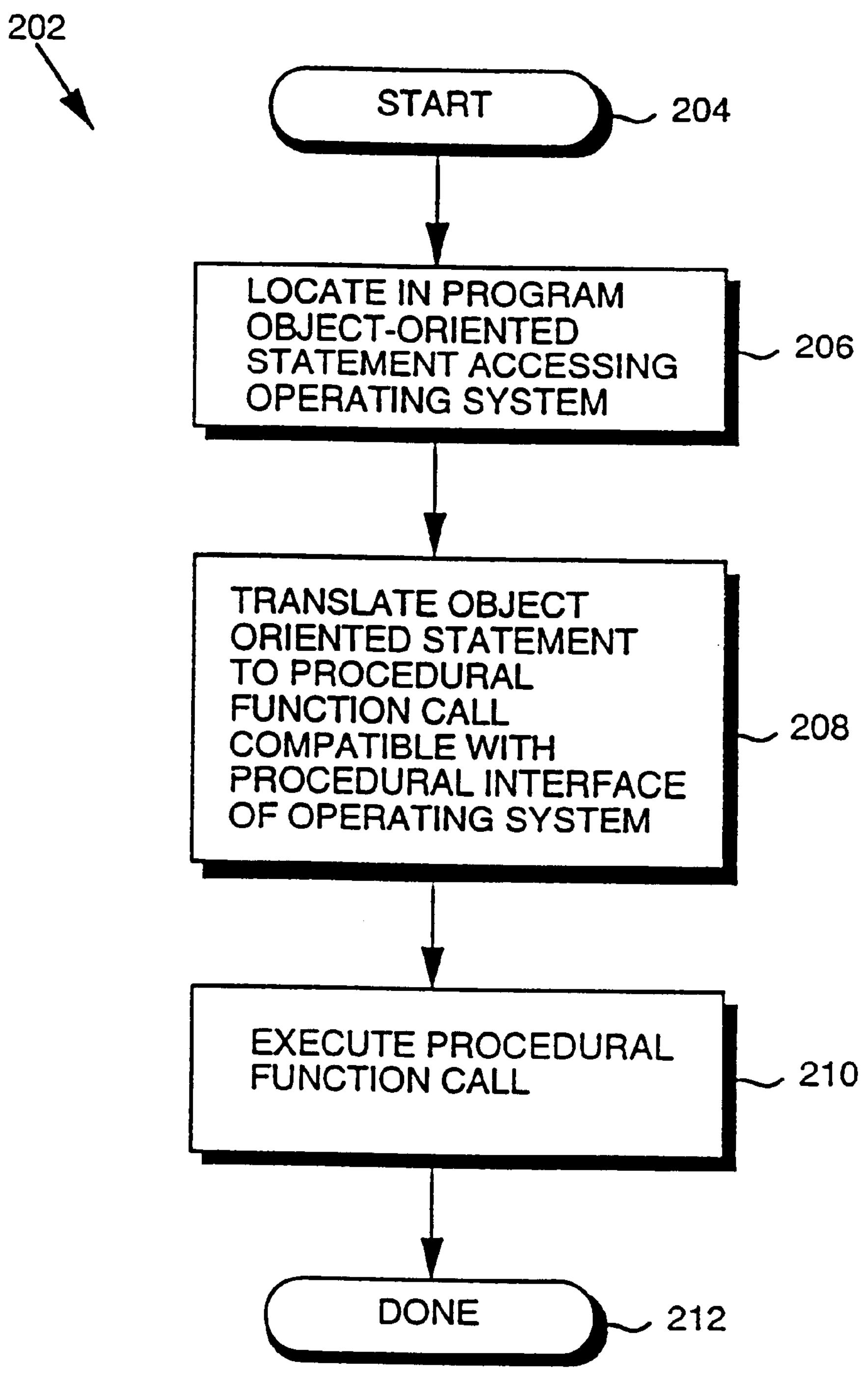

202
START
204
LOCATE IN PROGRAM OBJECT-ORIENTED STATEMENT ACCESSING OPERATING SYSTEM
206
TRANSLATE OBJECT ORIENTED STATEMENT TO PROCEDURAL FUNCTION CALL COMPATIBLE WITH PROCEDURAL INTERFACE OF OPERATING SYSTEM
208
EXECUTE PROCEDURAL FUNCTION CALL
210
DONE
212

Figure 2

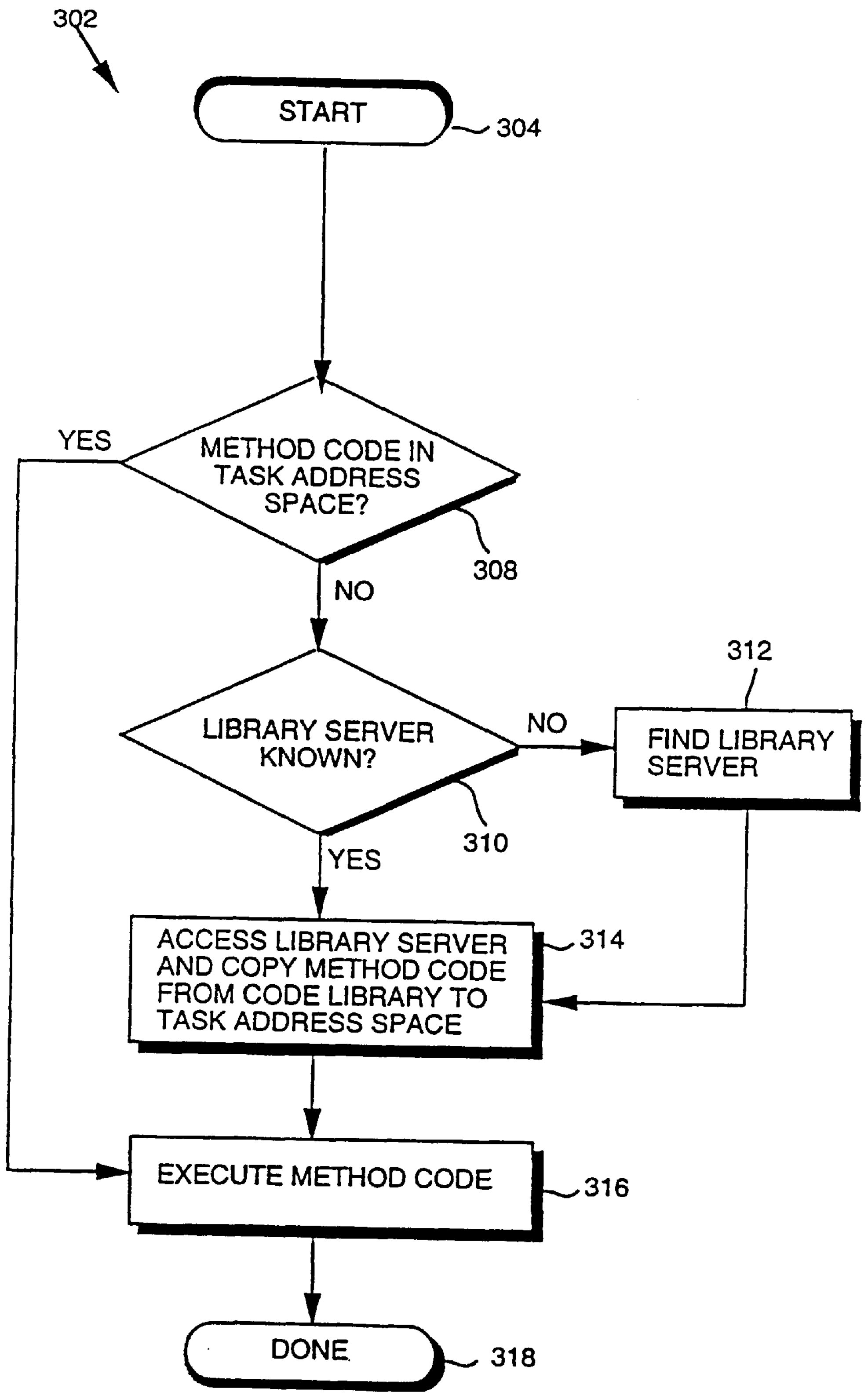

302
START
304
METHOD CODE IN TASK ADDRESS SPACE?
308
YES
NO
LIBRARY SERVER KNOWN?
310
NO
YES
FIND LIBRARY SERVER
312
ACCESS LIBRARY SERVER AND COPY METHOD CODE FROM CODE LIBRARY TO TASK ADDRESS SPACE
314
EXECUTE METHOD CODE
316
DONE
318

Figure 3

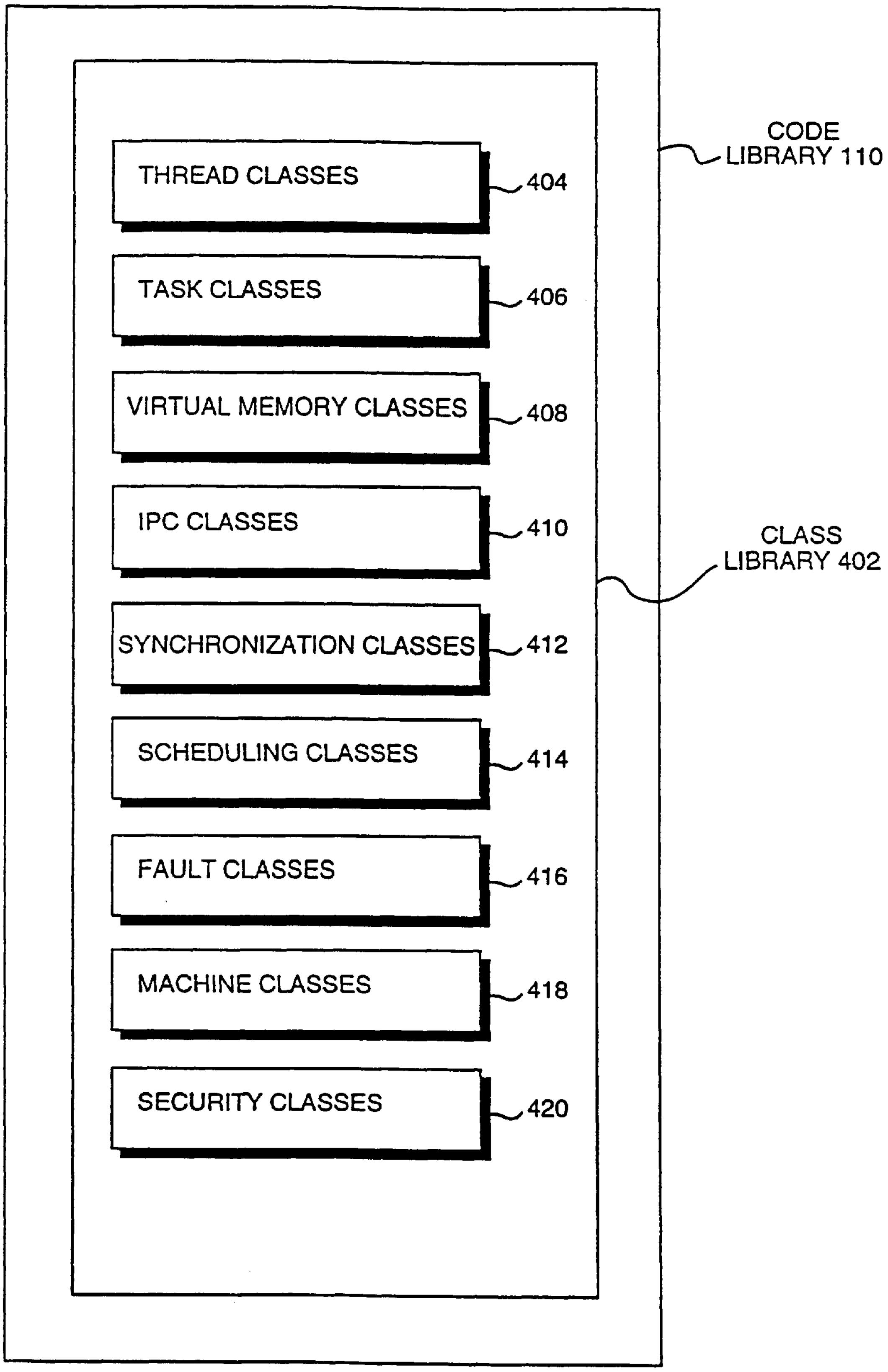

CODE LIBRARY 110
CLASS LIBRARY 402
THREAD CLASSES
404
TASK CLASSES
406
VIRTUAL MEMORY CLASSES
408
IPC CLASSES
410
SYNCHRONIZATION CLASSES
412
SCHEDULING CLASSES
414
FAULT CLASSES
416
MACHINE CLASSES
418
SECURITY CLASSES
420

Figure 4

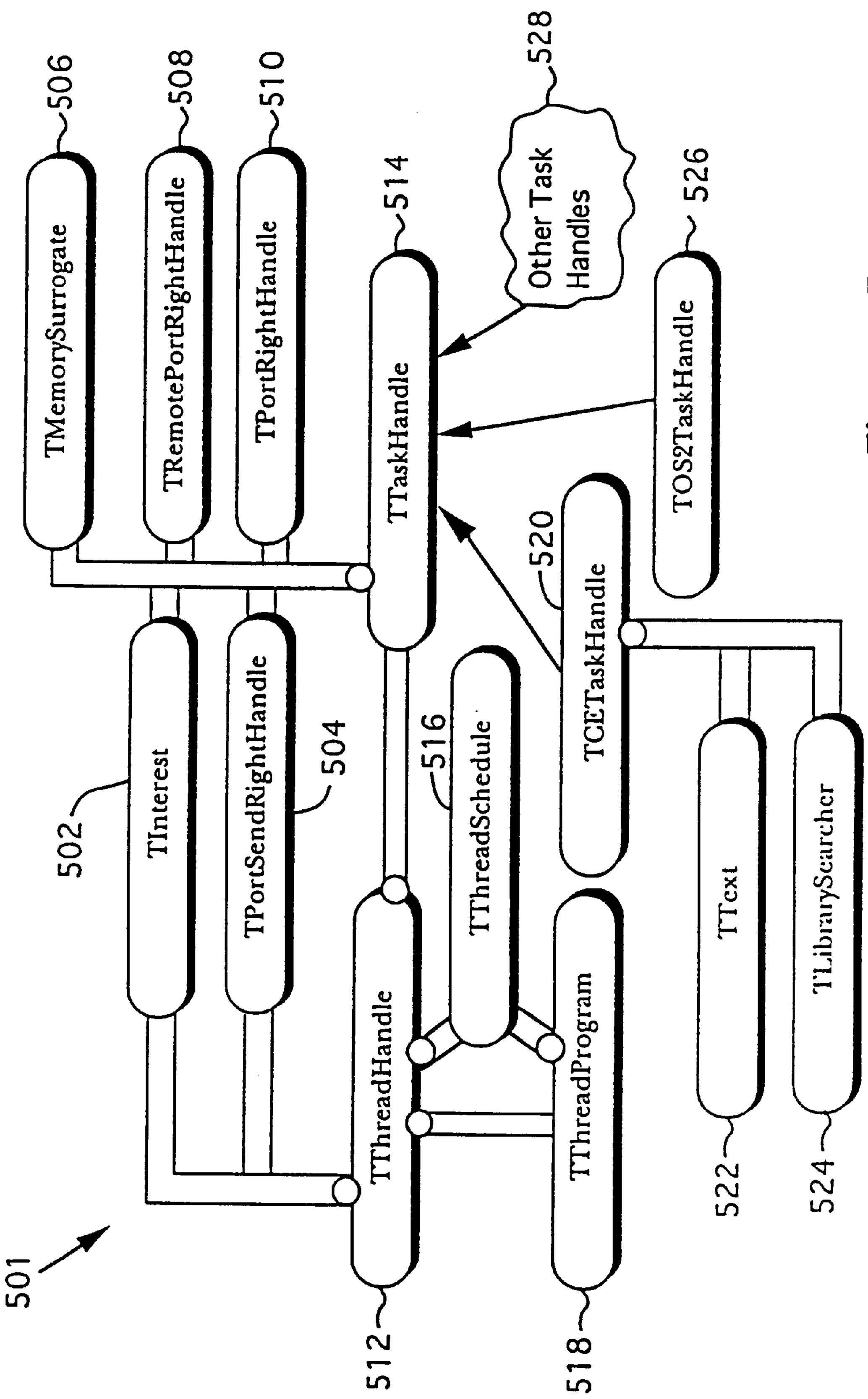

501
TMemorySurrogate
506
TRemotePortRightHandle
508
TPortRightHandle
510
TTaskHandle
514
Other Task Handles
528
TOS2TaskHandle
526
TCETaskHandle
520
TInterest
502
TPortSendRightHandle
504
TThreadSchedule
516
TThreadHandle
512
TThreadProgram
518
TText
522
TLibrarySearcher
524

Figure 5

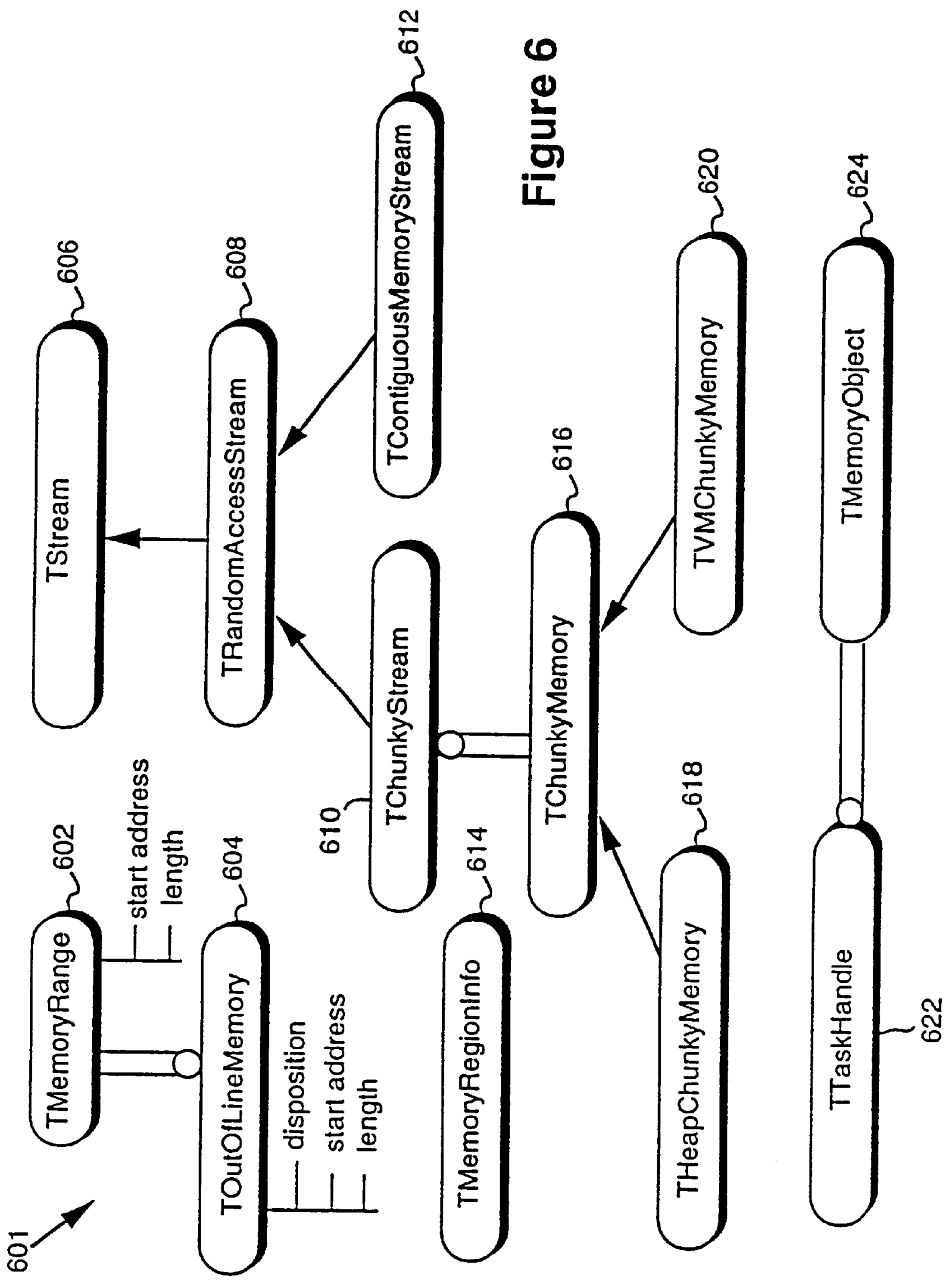

601
TMemoryRange
602
start address
length
TOutOfLineMemory
604
disposition
start address
length
TStream
606
TRandomAccessStream
608
TChunkyStream
610
TContiguousMemoryStream
612
TMemoryRegionInfo
614
TChunkyMemory
616
THeapChunkyMemory
618
TVMChunkyMemory
620
TTaskHandle
622
TMemoryObject
624

Figure 6

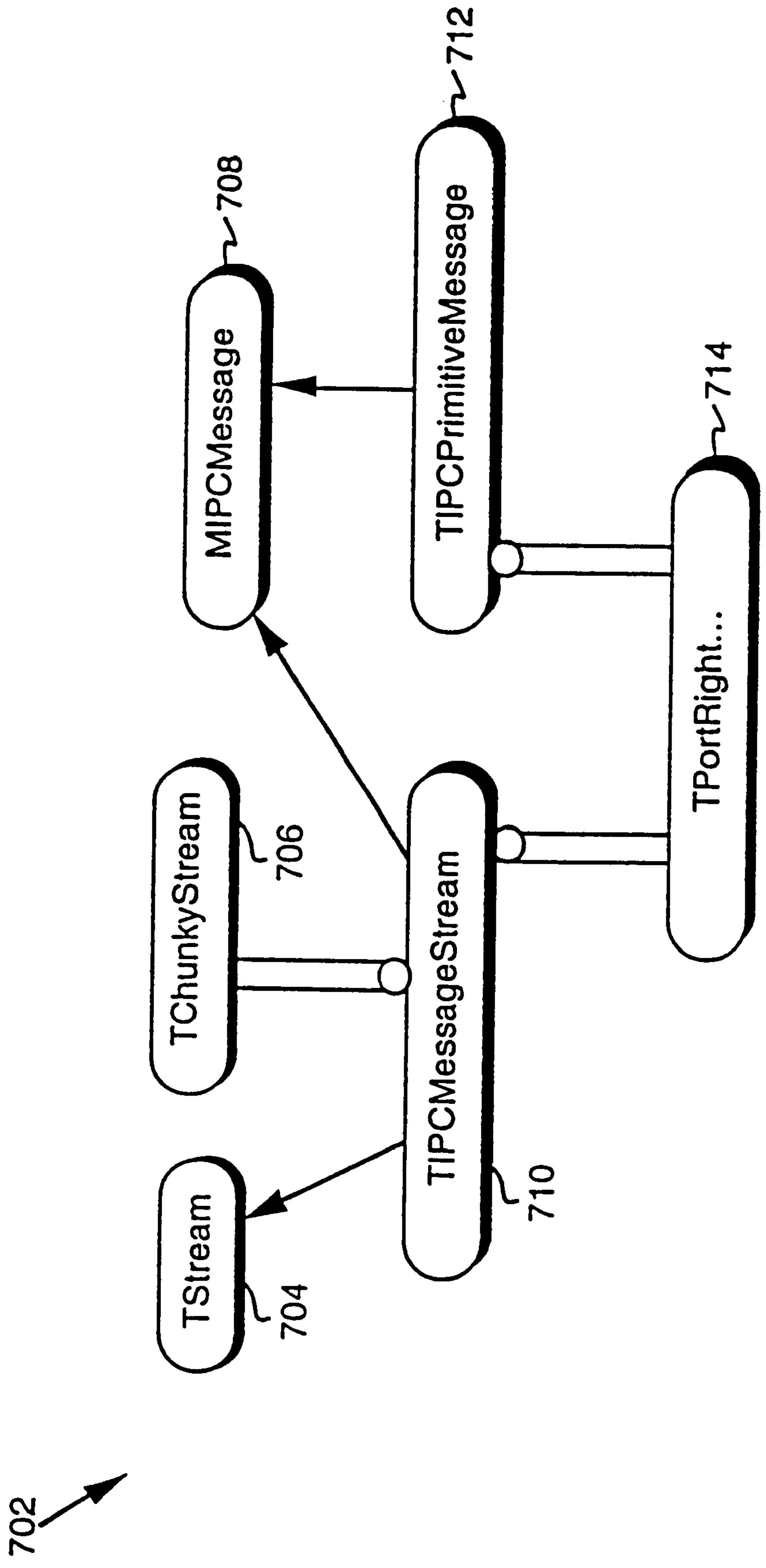

702
TStream
704
TChunkyStream
706
TIPCMessageStream
710
MIPCMessage
708
TIPCPrimitiveMessage
712
TPortRight...
714

Figure 7

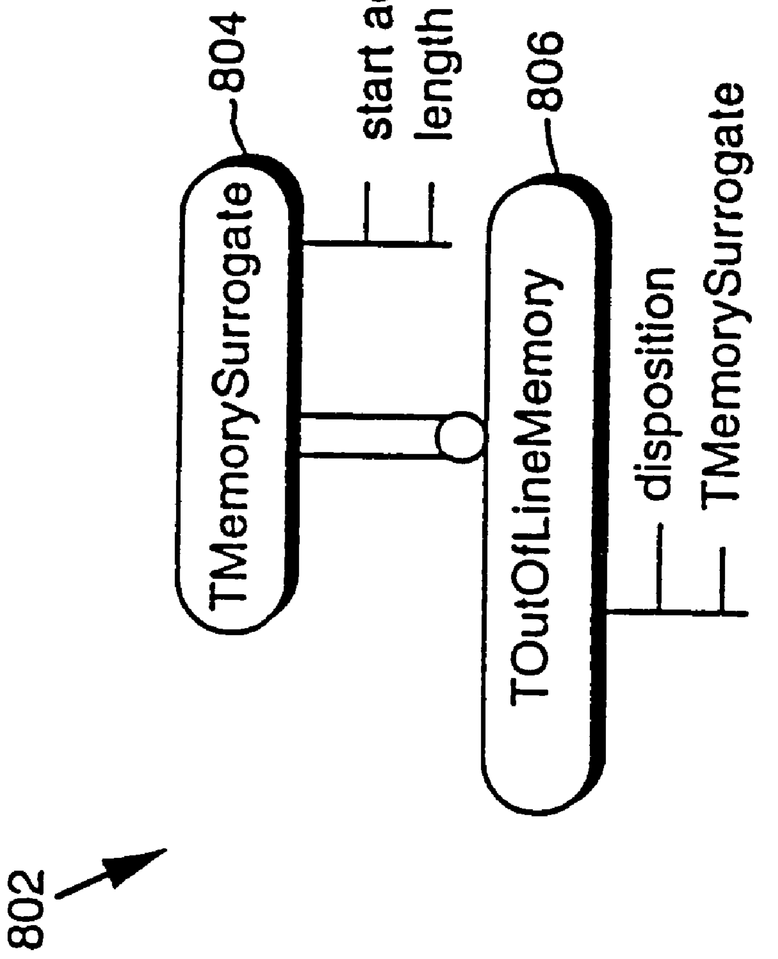
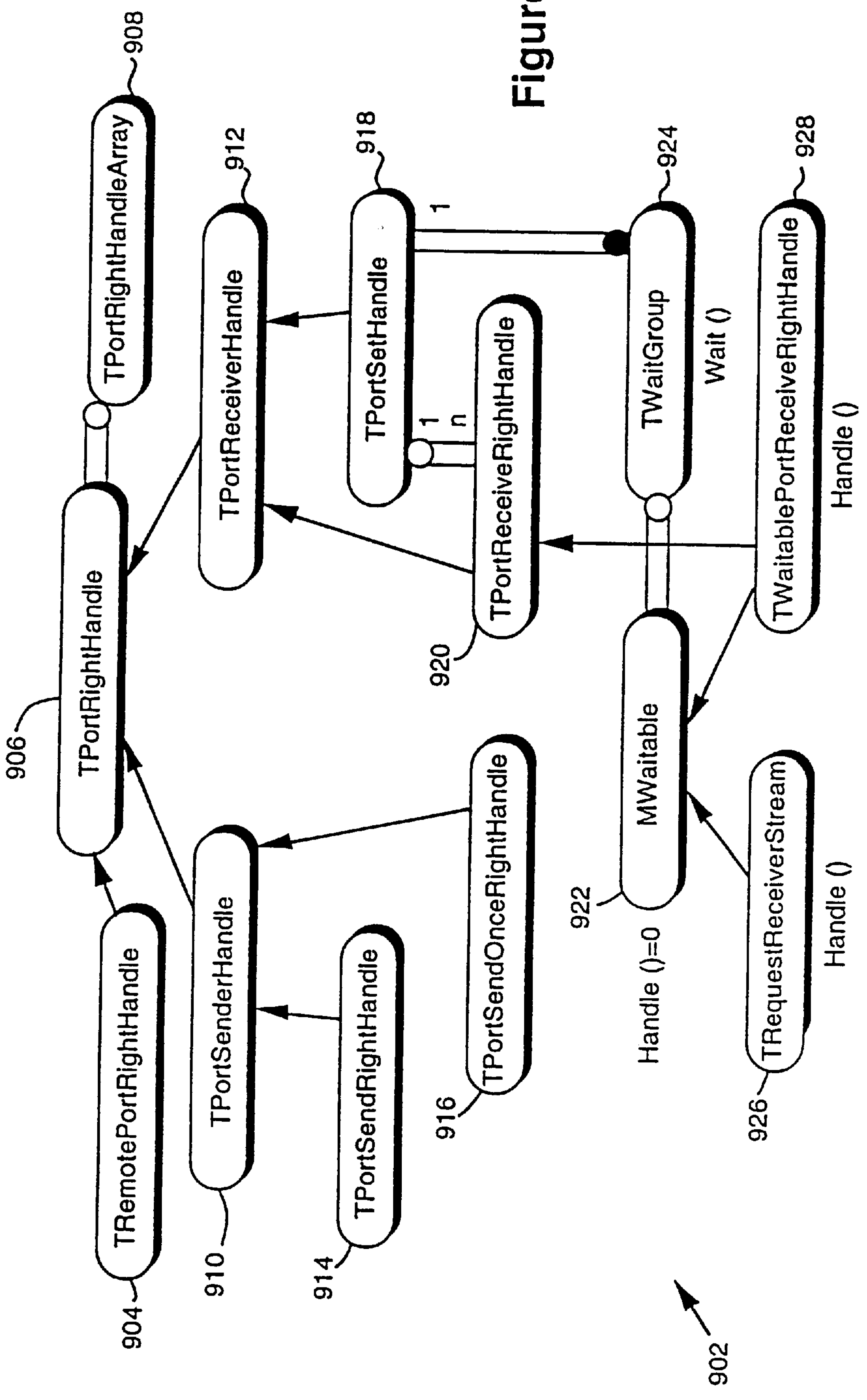

802
TMemorySurrogate
804
start address
length
TOutOfLineMemory
806
disposition
TMemorySurrogate

Figure 8

Figure 9


TRemotePortRightHandle
904
TPortRightHandle
906
TPortRightHandleArray
908
TPortSenderHandle
910
TPortReceiverHandle
912
TPortSendRightHandle
914
TPortSendOnceRightHandle
916
TPortSetHandle
918
TPortReceiveRightHandle
920
1
n
1
MWaitable
922
Handle ()=0
TWaitGroup
924
Wait ()
TRequestReceiverStream
926
Handle ()
TWaitablePortReceiveRightHandle
928
Handle ()
902

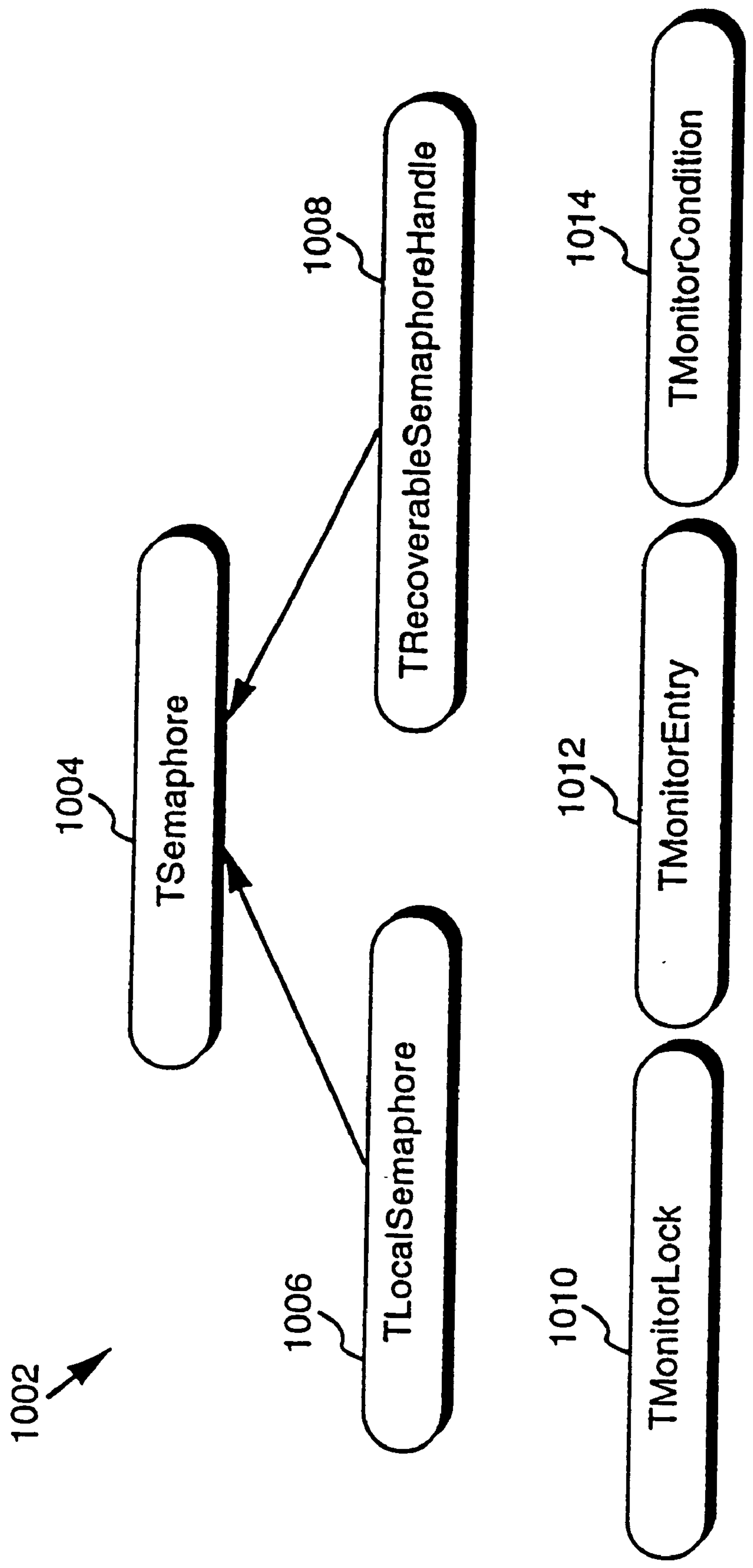

1002
1004
TSemaphore
1006
TLocalSemaphore
1008
TRecoverableSemaphoreHandle
1010
TMonitorLock
1012
TMonitorEntry
1014
TMonitorCondition

Figure 10

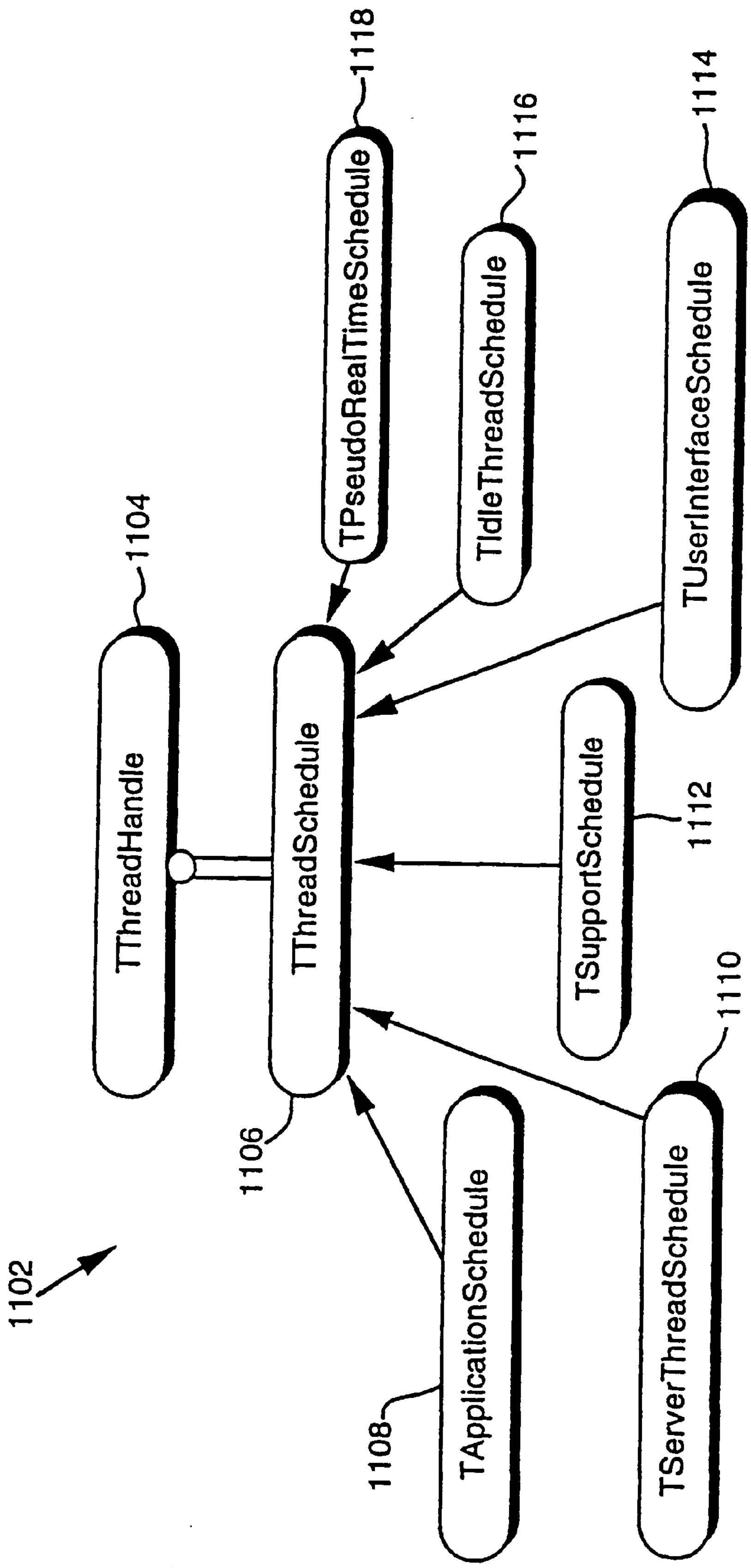
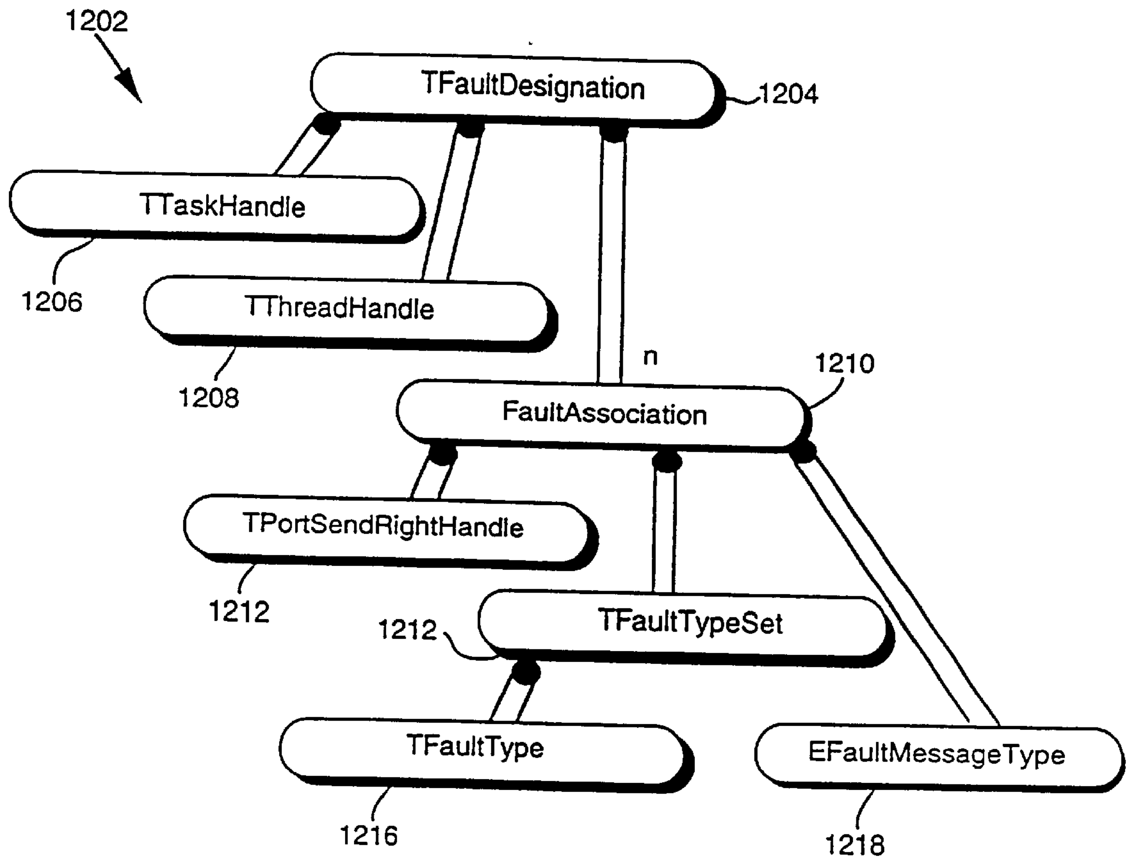

1102
TThreadHandle
1104
TThreadSchedule
1106
TPseudoRealTimeSchedule
1118
TIdleThreadSchedule
1116
TUserInterfaceSchedule
1114
TSupportSchedule
1112
TServerThreadSchedule
1110
TApplicationSchedule
1108

Figure 11


1202
TFaultDesignation
1204
TTaskHandle
1206
TThreadHandle
1208
n
1210
FaultAssociation
TPortSendRightHandle
1212
1212
TFaultTypeSet
TFaultType
1216
EFaultMessageType
1218

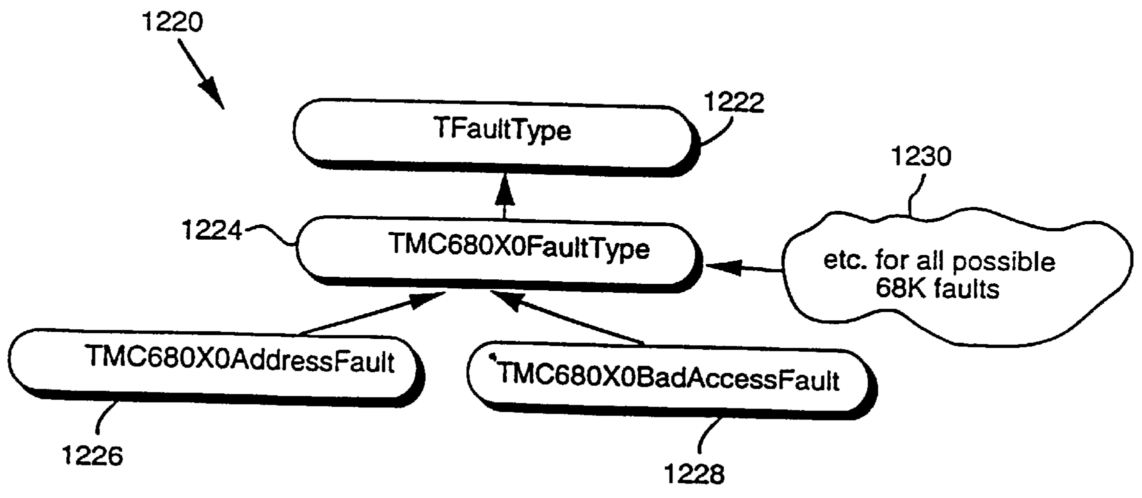
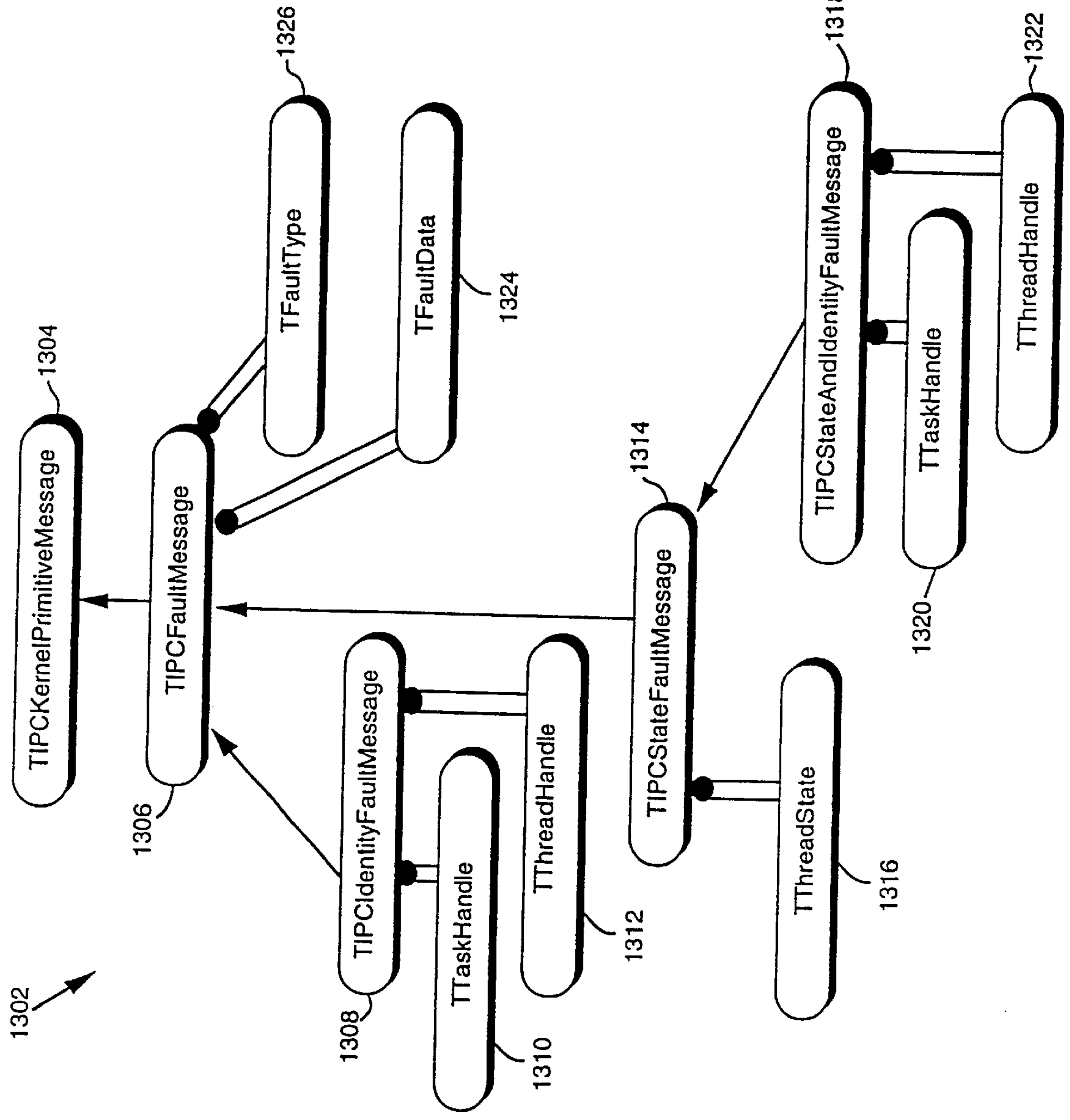

1220
TFaultType
1222
1230
1224
TMC680X0FaultType
etc. for all possible 68K faults
TMC680X0AddressFault
TMC680X0BadAccessFault
1226
1228

Figure 12

Figure 13


1302
TIPCKernelPrimitiveMessage
1304
TIPCFaultMessage
1306
TFaultType
1326
TFaultData
1324
TIPCIdentityFaultMessage
1308
TTaskHandle
1310
TThreadHandle
1312
TIPCStateFaultMessage
1314
TThreadState
1316
TIPCStateAndIdentityFaultMessage
1318
TTaskHandle
1320
TThreadHandle
1322

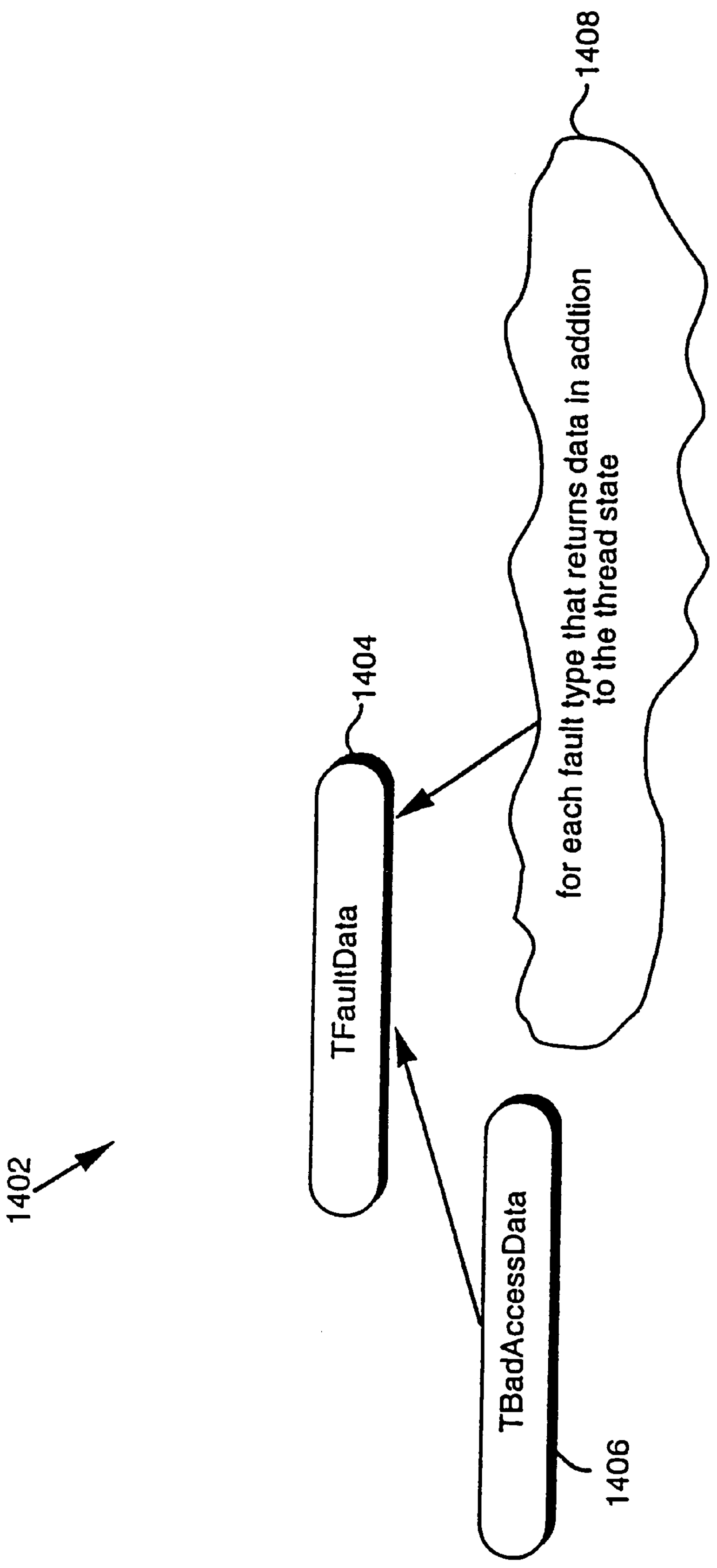

1402
TFaultData
1404
TBadAccessData
1406
for each fault type that returns data in addtion to the thread state
1408

Figure 14

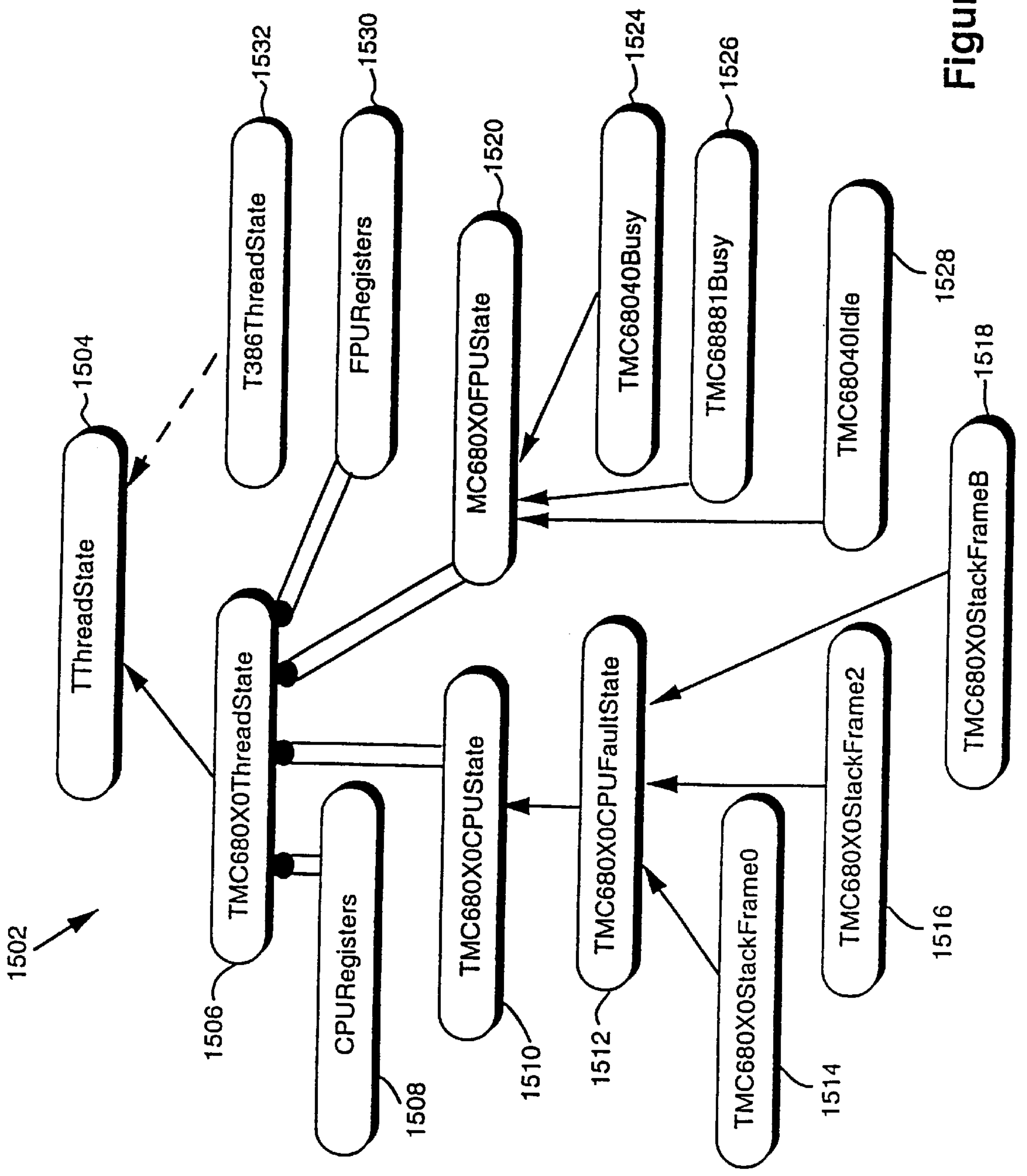

1502
TThreadState
1504
T386ThreadState
1532
FPURegisters
1530
MC680X0FPUState
1520
TMC68040Busy
1524
TMC68881Busy
1526
TMC68040Idle
1528
TMC680X0ThreadState
1506
CPURegisters
1508
TMC680X0CPUState
1510
TMC680X0CPUFaultState
1512
TMC680X0StackFrame0
1514
TMC680X0StackFrame2
1516
TMC680X0StackFrameB
1518

Figure 15

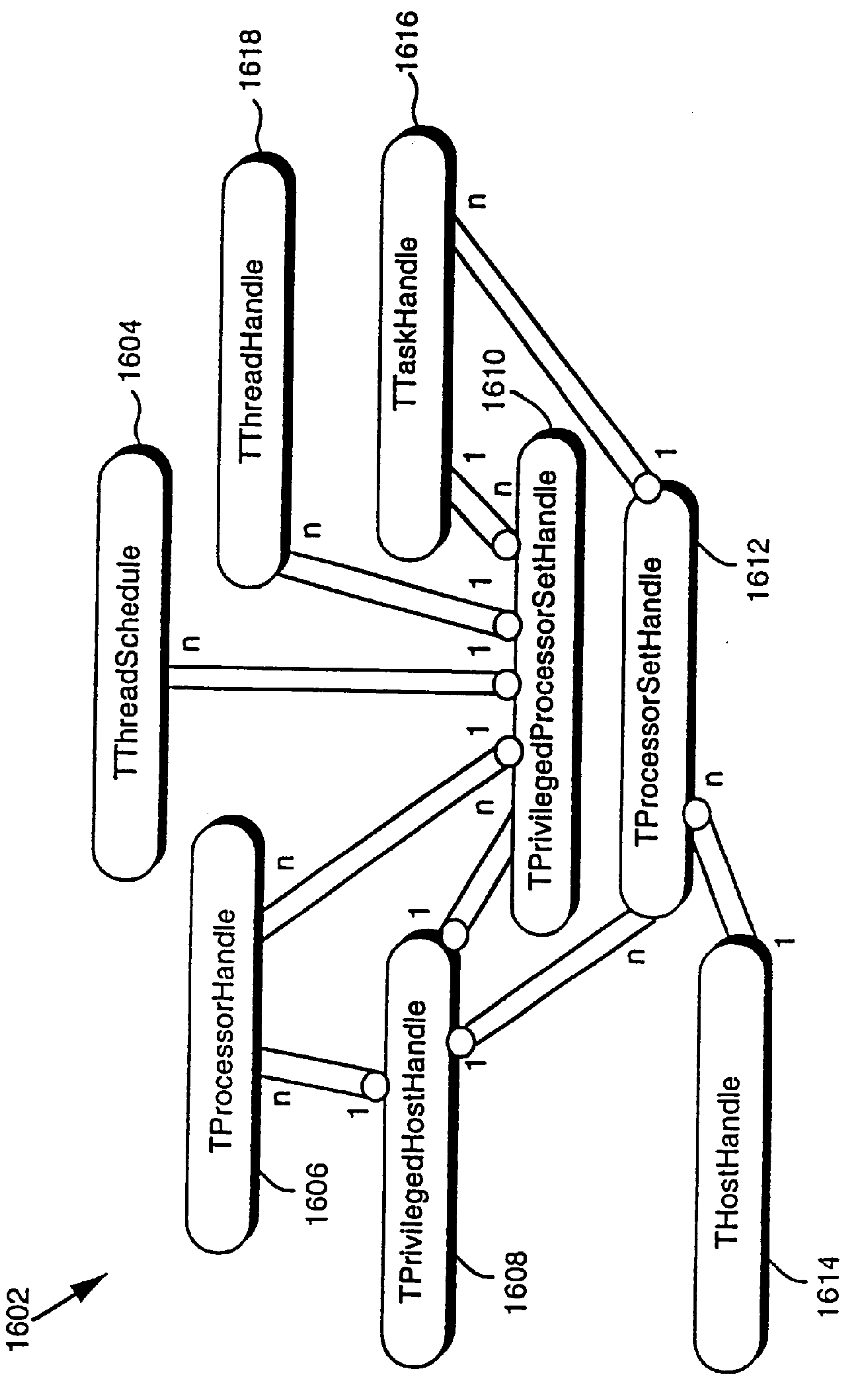

1602
1604
TThreadSchedule
1618
TThreadHandle
1616
TTaskHandle
1610
TPrivilegedProcessorSetHandle
1612
TProcessorSetHandle
TProcessorHandle
1606
TPrivilegedHostHandle
1608
THostHandle
1614
n
1

Figure 16

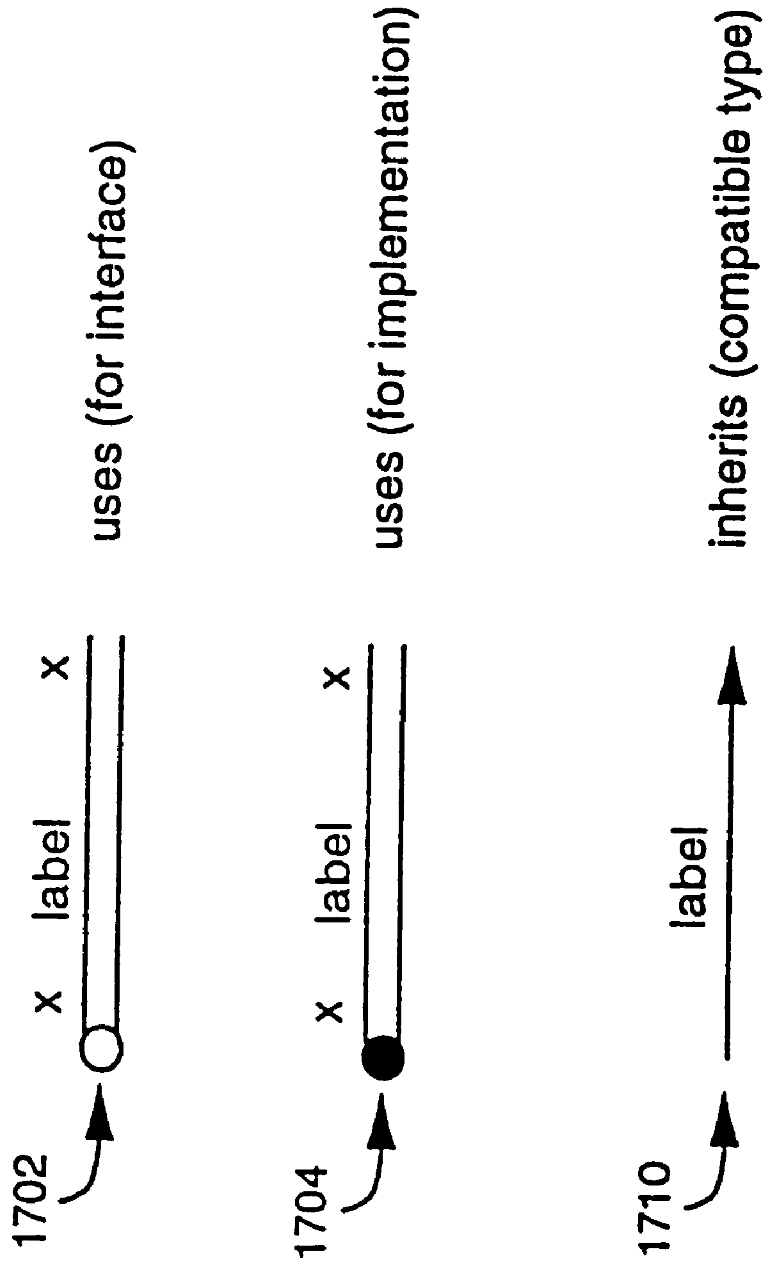

1702
x
label
x
uses (for interface)
1704
x
label
x
uses (for implementation)
1710
label
inherits (compatible type)

Figure 17

OBJECT-ORIENTED OPERATING SYSTEM

RELATED APPLICATIONS

This application is a continuation of U.S. patent application Ser. No. 08/521,085, filed Aug. 29, 1995, now abandoned.

OBJECT-ORIENTED OPERATING SYSTEM

FIELD OF THE INVENTION

The present invention relates generally to object-oriented computing environments, and more particularly to a system and method for providing an object-oriented interface for a procedural operating system.

BACKGROUND OF THE INVENTION

Object-oriented technology (OOT), which generally includes object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP), is earning its place as one of the most important new technologies in software development. OOT has already begun to prove its ability to create significant increases in programmer productivity and in program maintainability. By engendering an environment in which data and the procedures that operate on the data are combined into packages called objects, and by adopting a rule that demands that objects communicate with one another only through well-defined messaging paths, OOT removes much of the complexity of traditional, procedure-oriented programming.

The following paragraphs present a brief overview of some of the more important aspects of OOT. More detailed discussions of OOT are available in many publicly available documents, including *Object Oriented Design With Applications* by Grady Booch (Benjamin/Cummings Publishing Company, 1991) and *Object-Oriented Requirements Analysis and Logical Design* by Donald G. Firesmith (John Wiley & Sons, Inc., 1993). The basic component of OOT is the object. An object includes, and is characterized by, a set of data (also called attributes) and a set of operations (called methods) that can operate on the data. Generally, an object's data may change only through the operation of the object's methods.

A method in an object is invoked by passing a message to the object (this process is called message passing). The message specifies a method name and an argument list. When the object receives the message, code associated with the named method is executed with the formal parameters of the method bound to them corresponding values in the argument list. Methods and message passing in OOT are analogous to procedures and procedure calls in procedure-oriented software environments. However, while procedures operate to modify and return passed parameters, methods operate to modify the internal state of the associated objects (by modifying the data contained therein). The combination of data and methods in objects is called encapsulation. Perhaps the greatest single benefit of encapsulation is the fact that the state of any object can only be changed by well-defined methods associated with that object. When the

behavior of an object is confined to such well-defined locations and interfaces, changes (that is, code modifications) in the object will have minimal impact on the other objects and elements in the system. A second "fringe benefit" of good encapsulation in object-oriented design and programming is that the resulting code is more modular and maintainable than code written using more traditional techniques.

The fact that objects are encapsulated produces another important fringe benefit that is sometimes referred to as data abstraction. Abstraction is the process by which complex ideas and structures are made more understandable by the removal of detail and the generalization of their behavior. From a software perspective, abstraction is in many ways the antithesis of hard-coding. Consider a software windowing example: if every detail of every window that appears on a user's screen in a graphical user interface (GUI)-based program had to have all of its state and behavior hard-coded into a program, then both the program and the windows it contains would lose almost all of their flexibility. By abstracting the concept of a window into a window object, object-oriented systems permit the programmer to think only about the specific aspects that make a particular window unique. Behavior shared by all windows, such as the ability to be dragged and moved, can be shared by all window objects.

This leads to another basic component of OOT, which is the class. A class includes a set of data attributes plus a set of allowable operations (that is, methods) on the data attributes. Each object is an instance of some class. As a natural outgrowth of encapsulation and abstraction, OOT supports inheritance. A class (called a subclass) may be derived from another class (called a base class, a parent class, etc.) wherein the subclass inherits the data attributes and methods of the base class. The subclass may specialize the base class by adding code which overrides the data and/or methods of the base class, or which adds new data attributes and methods. Thus, inheritance represents a mechanism by which abstractions are made increasingly concrete as subclasses are created for greater levels of specialization. Inheritance is a primary contributor to the increased programmer efficiency provided by OOP. Inheritance makes it possible for developers to minimize the amount of new code they have to write to create applications. By providing a significant portion of the functionality needed for a particular task, classes in the inheritance hierarchy give the programmer a head start to program design and creation. One potential drawback to an object-oriented environment lies in the proliferation of objects that must exhibit behavior which is similar and which one would like to use as a single message name to describe. Consider, for example, an object-oriented graphical environment: if a Draw message is sent to a Rectangle object, the Rectangle object responds by drawing a shape with four sides. A Triangle object, on the other hand, responds by drawing a shape with three sides. Ideally, the object that sends the Draw message remains unaware of either the type of object to which the message is addressed or of how that object that receives the message will draw itself in response. If this ideal can be achieved, then it will be relatively simple to add a new kind of shape later (for example, a hexagon) and leave the code sending the Draw message completely unchanged.

In conventional, procedure-oriented languages, such a linguistic approach would wreak havoc. In OOT environments, the concept of polymorphism enables this to be done with impunity. As one consequence, methods can be written that generically tell other objects to do something without requiring the sending object to have any knowledge at all about the way the receiving object will understand the message. Software programs, be they object-oriented, procedure-oriented, rule based, etc., almost always interact with the operating system to access the services provided by the operating system. For example, a software program may interact with the operating system in order to access data in memory, to receive information relating to processor faults, to communicate with other processes, or to schedule the execution of a process.

Most conventional operating systems are procedure-oriented and include native procedural interfaces. Consequently, the services provided by these operating systems can only be accessed by using the procedures defined by their respective procedural interfaces. If a program needs to access a service provided by one of these procedural operating systems, then the program must include a statement to make the appropriate operating system procedure call. This is the case, whether the software program is object-oriented, procedure-oriented, rule based, etc. Thus, conventional operating systems provide procedure-oriented environments in which to develop and execute software. Some of the advantages of OOT are lost when an object-oriented program is developed and executed in a procedure-oriented environment. This is true, since all accesses to the procedural operating system must be implemented using procedure calls defined by the operating system's native procedural interface. Consequently, some of the modularity, maintainability, and reusability advantages associated with object-oriented programs are lost since it is not possible to utilize classes, objects, and other OOT features to their fullest extent possible.

One solution to this problem is to develop object-oriented operating systems having native object-oriented interfaces. While this ultimately may be the best solution, it currently is not a practical solution since the resources required to modify all of the major, procedural operating systems would be enormous. Also, such a modification of these procedural operating systems would render useless thousands of procedure-oriented software programs. Therefore, what is needed is a mechanism for enabling an object-oriented application to interact in an object-oriented manner with a procedural operating system having a native procedural interface.

SUMMARY OF THE INVENTION

The present invention is directed to a system and method of enabling an object-oriented application to access in an object-oriented manner a procedural operating system having a native procedural interface. The system includes a computer and a memory component in the computer. A code library is stored in the memory component. The code library includes computer program logic implementing an object-oriented class library. The object-oriented class library comprises related object-oriented classes for enabling the application to access in an object-oriented manner services provided by the operating system. The object-oriented classes include methods for accessing the operating system services using procedural function calls compatible with the native procedural interface of the operating system. The system also includes means for processing object-oriented statements contained in the application and defined by the class library by executing methods from the class library corresponding to the object-oriented statements.

Preferably, the class library includes:

(1) thread classes for enabling an application to access in an object-oriented manner operating system services to spawn, control, and obtain information relating to threads;

(2) task classes for enabling an application to access in an object-oriented manner operating system services to reference and control tasks, wherein the tasks each represents an execution environment for threads respectively associated with the tasks;

(3) virtual memory classes for enabling an application to access in an object-oriented manner operating system services to access and manipulate virtual memory in a computer;

(4) interprocess communication (IPC) classes for enabling an application to access in an object-oriented manner operating system services to communicate with other threads during run-time execution of the application in a computer;

(5) synchronization classes for enabling an application to access in an object-oriented manner operating system services to synchronize execution of threads;

(6) scheduling classes for enabling an application to access in an object-oriented manner operating system services to schedule execution of threads;

(7) fault classes for enabling an application to access in an object-oriented manner operating system services to process system and user-defined processor faults; and

(8) machine classes for enabling an application to access in an object-oriented manner operating system services to define and modify a host and processor sets.

Further features and advantages of the present invention, as well as the structure and operation of various embodiments of the present invention, are described in detail below with reference to the accompanying drawings, and in the claims. In the drawings, identical reference numbers indicate identical or functionally similar elements.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be described with reference to the accompanying drawings, wherein:

FIG. **1** illustrates a block diagram of a computer platform in which a wrapper of the present invention operates;

FIG. **2** is a high-level flow chart illustrating the operation of the present invention;

FIG. **3** is a more detailed flowchart illustrating the operation of the present invention;

FIG. **4** is a block diagram of a code library containing an object-oriented dass library of the present invention;

FIG. **5** is a class diagram of thread and task classes of the present invention;

FIG. **6** is a class diagram of virtual memory classes of the present invention;

FIGS. **7–9** are class diagrams of interprocess communication classes of the present invention;

FIG. **10** is a class diagram of synchronization classes of the present invention;

FIG. **11** is a class diagram of scheduling classes of the present invention;

FIGS. **12–15** are class diagrams of fault classes of the present invention;

FIG. **16** is a class diagram of host and processor set (machine) classes of the present invention; and

FIG. **17** illustrates well-known icons for representing class relationships and cardinality in class diagrams.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Computing Environment

The present invention is directed to a system and method for providing an object-oriented interface to a procedural operating system having a native procedural interface. The present invention emulates an object-oriented software environment on a computer platform having a procedural operating system. More particularly, the present invention is directed to a system and method of enabling an object-oriented application to access in an object-oriented manner a procedural operating system having a native procedural interface during run-time execution of the application in a computer. The present invention is preferably a part of the run-time environment of the computer in which the application executes. In this patent application, the present invention is sometimes called an object-oriented wrapper since it operates to wrap a procedural operating system with an object-oriented software layer such. that an object-oriented application can access the operating system in an object-oriented manner.

FIG. **1** illustrates a block diagram of a computer platform **102** in which a wrapper **128**, **129** of the present invention operates. It should be noted that the present invention alternatively encompasses the wrapper **128**, **129** in combination with the computer platform **102**. The computer platform **102** includes hardware components **103**, such as a random access memory (RAM) **108** and a central processing unit (CPU) **106**. It should be noted that the CPU **106** may represent a single processor, but preferably represents multiple processors operating in parallel. The computer platform **102** also includes peripheral devices which are connected to the hardware components **103**. These peripheral devices include an input device or devices (such as a keyboard, a mouse, a light pen, etc.), a data storage device **120** (such as a hard disk or floppy disk), a display **124**, and a printer **126**. The data storage device **120** may interact with a removable data storage medium **122** (such as a removable hard disk, a magnetic tape cartridge, or a floppy disk), depending on the type of data storage device used. The computer platform **102** also includes a procedural operating system **114** having a native procedural interface (not shown). The procedural interface includes procedural functions which are called to access services provided by the operating system **102**.

The computer platform **102** further includes device drivers **116**, and may include microinstruction code **210** (also called firmware). As indicated in FIG. **1**, in performing their required functions the device drivers **116** may interact with the operating system **114**. Application programs **130**, **132**, **134** (described further below) preferably interact with the device drivers **116** via the operating system **114**, but may alternatively interact directly with the device drivers **116**. It should be noted that the operating system **114** may represent a substantially full-function operating system, such as the Disk Operating System (DOS) and the UNIX operating system. However, the operating system **114** may represent other types of operating systems. For purposes of the present invention, the only requirement is that the operating system **114** be a procedural operating system having a native procedural interface. Preferably, the operating system **114** represents a limited functionality procedural operating system, such as the Mach micro-kernel developed by CMU, which is well-known to those skilled in the relevant art. For illustrative purposes only, the present invention shall be described herein with reference to the Mach micro-kernel. In a preferred embodiment of the present invention, the computer platform **102** is an International Business Machines (IBM) computer or an IBM-compatible computer. In an alternate embodiment of the present invention, the computer platform **102** is an Apple computer.

Overview of a Wrapper

Various application programs **130**, **132**, **134** preferably operate in parallel on the computer platform **102**. Preferably,

the application programs **130**, **132**, **134** are adapted to execute in different operating environments. For example, the application programs **130**A and **130**B may be adapted to operate in an object-oriented environment. The application program **132** may be adapted to operate in a Microsoft Windows environment, an IBM PS/2 environment, or a Unix environment. As will be appreciated by those skilled in the relevant art, the application programs **130**A, **130**B, and **132** cannot interact directly with the operating system **114** unless the operating system **114** implements an environment in which the application programs **130**A, **130**B, and **132** are adapted to operate. For example, if the application **132** is adapted to operate in the IBM PS/2 environment, then the application **132** cannot directly interact with the operating system **114** unless the operating system **114** is the IBM PS/2 operating system (or compatible). If the application programs **130**A and **130**B are adapted to operate in an object-oriented environment, then the applications **130**A, **130**B cannot directly interact with the operating system **114** since the operating system **114** has a procedural interface. In the example shown in FIG. **1**, the application **134** is adapted to operate in the computing environment created by the operating system **114**, and therefore the application **134** is shown as being connected directly to the operating system **114**.

The wrapper **128** is directed to a mechanism for providing the operating system **114** with an object-oriented interface. The wrapper **128** enables the object-oriented applications **130**A, **130**B to directly access in an object-oriented manner the procedural operating system **114** during run-time execution of the applications **130**A, **130**B on the computer platform **102**. The wrapper **129** is conceptually similar to the wrapper **128**. The wrapper **129** provides an IBM PS/2 interface for the operating system **114**, such that the application **132** can directly access in a PS/2 manner the procedural operating system **114** (assuming that the application **132** is adapted to operate in the IBM PS/2 environmnent). The discussion of the present invention shall be limited herein to the wrapper **128**, which provides an object-oriented interface to a procedural operating system having a native procedural interface.

The wrapper **128** is preferably implemented as a code library **110** which is stored in the RAM **108**. The code library **110** may also be stored in the data storage device **120** and/or the data storage medium **122**. The code library **110** implements an object-oriented class library **402** (see FIG. **4**). In accordance with the present invention, the object-oriented class library **402** indudes related object-oriented classes for enabling an object-oriented application (such as the applications **130**A and **130**B) to access in an object-oriented manner services provided by the operating system **114**. The object-oriented classes comprise methods which indude procedural function calls compatible with the native procedural interface of the operating system **114**. Object-oriented statements defined by the object-oriented class library **402** (such as object-oriented statements which invoke one or more of the methods of the class library **402**) are insertable into the application **130** to enable the application **130** to access in an object-oriented manner the operating system services during run-time execution of the application **130** on the computer platform **102**. The object-oriented class library **402** is further described in sections below.

The code library **110** preferably includes compiled, executable computer program logic which implements the object-oriented class library **402**. The computer program logic of the code library **110** is not linked to application programs. Instead, relevant portions of the code library **110** are copied into the executable address spaces of processes during run-time. This is explained in greater detail below. Since the computer program logic of the code library **110** is not linked to application programs, the computer program logic can be modified at any time without having to modify, recompile and/or relink the application programs (as long as the interface to the code library **110** does not change). As noted above, the present invention shall be described herein with reference to the Mach micro-kernel, although the use of the present invention to wrap other operating systems falls within the scope of the present invention.

The Mach micro-kernel provides users with a number of services with are grouped into the following categories: threads, tasks, virtual memnory, interprocess, communication (IPC), scheduling, synchronization, fault processing, and host/processor set processing. The class library **402** of the present invention includes a set of related classes for each of the Mach service categories. Referring to FIG. **4**, the class library **402** includes:

(1) thread classes **404** for enabling an application to access in an object-oriented manner operating system services to spawn, control, and obtain information relating to threads,

(2) task classes **406** for enabling an application to access in an object-oriented manner operating system services to reference and control tasks, wherein the tasks each represents an execution environment for threads respectively associated with the tasks;

(3) virtual memory classes **408** for enabling an application to access in an object-oriented manner operating system services to access and manipulate virtual memory in a computer;

(4) IPC classes **410** for enabling an application to access in an object-oriented manner operating system services to communicate with other processes during run-time execution of the application in a computer;

(5) synchronization classes **412** for enabling an application to access in an object-oriented manner operating system services to synchronize execution of threads;

(6) scheduling classes **414** for enabling an application to access in an object-oriented manner operating system services to schedule execution of threads;

(7) fault classes **416** for enabling an application to access in an object-oriented manner operating system services to process system and user-defined processor faults; and

(8) machine classes **418** for enabling an application to access in an object-oriented manner operating system services to define and modify a host and processor sets.

The class library **402** may include additional classes for other service categories that are offered by Mach in the future. For example, security services are currently being developed for Mach. Accordingly, the class library **402** may also include security classes **420** for enabling an application to access in an object-oriented manner operating system security services. As will be appreciated, the exact number and type of classes included in the class library **402** depends on the implementation of the underlying operating system.

Operational Overview of a Preferred Embodiment

The operation of the present invention shall now be generally described with reference to FIG. **2**, which illustrates a high-level operational flow chart **202** of the present invention. The present invention is described in the context of executing the object-oriented application **130**A on the computer platform **102**. In step **206**, which is the first substantive step of the flow chart **202**, an object-oriented statement which accesses a service provided by the operating system **114** is located in the application **130**A during the execution of the application **130**A on the computer platform **102**. The object-oriented statement is defined by the object-oriented class library **402**. For example, the object-oriented statement may reference a method defined by one of the classes of the class library **402**. The following steps describe the manner in which the statement is executed by the computer platform **102**.

In step **208**, the object-oriented statement is translated to a procedural function call compatible with the native procedural interface of the operating system **114** and corresponding to the object-oriented statement. In performing step **208**, the statement is translated to the computer program logic from the code library **110** which implements the method referenced in the statement. As noted above, the method includes at least one procedural function call which is compatible with the native procedural interface of the operating system **114**. In step **210**, the procedural function call from step **208** is executed in the computer platform **102** to thereby cause the operating system **114** to provide the service on behalf of the application **130**A. Step **210** is performed by executing the method discussed in step **208**, thereby causing the procedural function call to be invoked.

The operation of a preferred embodiment shall now be described in more detail with reference to FIG. **3**, which illustrates a detailed operational flow chart **302** of the present invention. Again, the present invention is described in the context of executing the object-oriented application **130**A on the computer platform **102**. More particularly, the present invention is described in the context of executing a single object-oriented rstatement of the object-oriented application **130**A on the computer platform **102**. The application **130**A includes statements which access services provided by the operating system **114**, and it is assumed that such statements are defined by the class library **402** (in other words, the programmer created the application **130**A with reference to the class library **402**). As will be discussed in greater detail below, the executable entity in the Mach micro-kernel is called a thread. The processing organization entity.in the Mach micro-kernel is called a task. A task includes one or more threads (which may execute in parallel), and an address space which represents a block of virtual memory in which the task's threads can execute. At any time, there may be multiple tasks active on the computer platform **102**. When executing on the computer platform **102**, the application **130**A could represent an entire task (having one or more threads), or could represent a few threads which are part of a task (in this case, the task would have other threads which may or may not be related to the operation of the application **130**A). The scope of the present invention encompasses the case when the application **130**A is an entire task, or just a few threads of a task.

Referring now to FIG. **3**, in step **308**, it is determined whether the computer program logic (also called computer code) from the code library **110** which implements the method referenced in the statement is present in the task address space associated with the application **130**A. If the computer program logic is present in the task address space, then step **316** is processed (described below). If the computer program logic is not present in the task address space, then the computer program logic is transferred to the task address space in steps **310**,**312**, and **314**. In step **310**, it is determined whether the library server (not shown) associated with the code library **110** is known. The code library **110** may represent multiple code libraries (not shown) related to the wrapper **128**, wherein each of the code libraries include the computer program logic for one of the object-oriented classes of the class library **402**. As those skilled in the relevant art will appreciate, there may also be other code libraries (not shown) completely unrelated to the wrapper **128**.

Associated with the code libraries are library servers, each of which manages the resources of a designated code library. A processing entity which desires access to the computer program logic of a code library makes a request to the code library's library server. The request may include, for example, a description of the desired computer program logic and a destination address to which the computer program logic should be sent. The library server processes the request by accessing the desired computer program logic from the code library and sending the desired computer program logic to the area of memory designated by the destination address. The structure and operation of library servers are well known to those skilled in the relevant art. Thus, in step **310** it is determined whether the library server associated with the code library **110** which contains the relevant computer program logic is known. Step **310** is performed, for example, by referencing a library server table which identifies the known library servers and the code libraries which they service. If the library server is known, then step **314** is processed (discussed below). Otherwise, step **312** is processed. In step **312**, the library server associated with the code library **110** is identified. The identity of the library server may be apparent, for example, from the content of the object-oriented statement which is being processed.

After the library server associated with the code library **110** is identified, or if the library server was already known, then step **314** is processed. In step **314**, a request is sent to the library server asking the library server to copy the computer program logic associated with the method reference in the statement to the task address space. Upon completion of step **314**, the library server has copied the requested computer program logic to the task address space. Preferably, the code library **110** is a shared library. That is, the code library **110** may be simultaneously accessed by multiple threads. However, preferably the computer program logic of the code library **110** is physically stored in only one physical memory area. The library server virtually copies computer program logic from the code library **110** to task address spaces. That is, instead of physically copying computer program logic from one part of physical memory to another, the library server places in the task address space a pointer to the physical memory area containing the relevant computer program logic. In step **316**, the computer program logic associated with the object-oriented statement is executed on the computer platform **102**. As noted above, in the case where the object-oriented statement accesses the operating system **114**, the computer program logic associated with the method contains at least one procedural function call which is compatible with the native procedural interface of the operating system **114**. Thus, by executing the method's computer program logic, the procedural function call is invoked and executed, thereby causing the operating system **114** to provide the service on behalf of the application **130**A.

The above-described performance in the computer platform **102** of steps **306**, **308**, **310**, **312**, and **314** is due, in large part, to the run-time environment established in the computer platform **102**. As will be appreciated by those skilled in the relevant art, the run-time environment of the computer platform **102** is defined by the run-time conventions of the particular compiler which compiles the application program **130**A. For example, the run-time conventions may specify that when an instruction accessing an operating system service is encountered, corresponding code from the code library **110** should be transferred to the task address space (via the associated library server) and executed. Compiler run-time conventions are generally well known. As will be appreciated, run-time conventions are specific to the particular compilers used. The run-time conventions for use with the present invention and with a particular compiler would be apparent to one skilled in the art based on the disclosure of the present invention contained herein, particularly to the disclosure associated with the flow chart **302** in FIG. **3**. As described above, the wrapper **128** of the present invention is implemented as a code library **110** which includes computer program logic implementing the object-oriented class library **402**. Alternatively, the wrapper **128** may be implemented as a hardware mechanism which essentially operates in accordance with the flow chart **302** of FIG. **3** to translate object-oriented statements (defined by the class library **402**) in application programs to procedural function calls compatible with the procedural interface of the operating system **114**. Or, the wrapper **128** may be implemented as a background software process operating on the computer platform **102** which captures all accesses to the operating system **114** (made by object-oriented statements defined by the class library **402**) and which translates the accesses to procedural function calls compatible with the procedural interface of the operating system **114**. Other implementations of the wrapper **128** will be apparent to those skilled in the relevant art based on the disclosure of the present invention contained herein.

Mach Services

This section provides an overview of the abstractions and services provided by the Mach micro-kernel. The services are described for each of the major areas of the Mach micro-kernel. As noted above, these include: threads, tasks, virtual memory, IPC, scheduling, synchronization services, hardware faults, and host/privilege services (also called machine services). The Mach micro-kernel is further discussed in many publicly available documents, including: K. Loepere, editor, "Mach 3 Kernel Principles", *Open Software Foundation and Carnegie Mellon University, Draft Industrial Specification*, September 1992 and November 1992; K. Loepere, editor, "Mach 3 Kernel Interfaces", *Open Software Foundation and Carnegie Mellon University, Draft Industrial Specification*, September 1992 and November 1992; K. Loepere, editor, "Mach 3 Server Writer's Guide", *Open Software Foundation and Carnegie Mellon University, Draft Industrial Specification*, September 1992 and November 1992; K. Loepere, editor, "Mach 3 Server Writer's Interfaces", *Open Software Foundation and Carnegie Mellon University, Draft Industrial Specification*, September 1992 and November 1992; A. Silberschatz, J. Peterson, P. Galvin, *Operating System Concepts*, Addison-Wesley, July 1992; and A. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.

Threads

The executable entity in Mach is known as a thread. Threads have several aspects that enable them to execute in the system. A thread is always contained in a task, which represents most of the major resources (e.g., address space) of which the thread can make use. A thread has an execution state, which is basically the set of machine registers and other data that make up its context. A thread is always in one of several scheduling states: executing, ready to execute, or blocked for some reason. Threads are intended to be lightweight execution entities. This is to encourage the programmer to make use of multiple threads in applications, thus introducing more concurrency into the system than has been found in traditional operating systems. Although threads are not without some cost, they really are fairly minimal and the typical application or server in a Mach environment can take advantage of this capability.

Threads do have some elements associated with them, however. The containing task and address space, as well as the execution state, have already been discussed. Each thread has a scheduling policy, which determines when and how often the thread will be given a processor on which to run. The scheduling services are discussed in more detail in a later section. Closely tied to the scheduling policy of a thread is the optional processor set designation, which can be used in systems with multiple processors to more closely control the assignment of threads to processors for potentially greater application performance. As indicated before, an address space (task) can contain zero or more threads, which execute concurrently. The kernel makes no assumptions about the relationship of the threads in an address space or, indeed, in the entire system. Rather, it schedules and executes the threads according to the scheduling parameters associated with them and the available processor resources in the system. In particular, there is no arrangement (e.g., hierarchical) of threads in an address space and no assumptions about how they are to interact: with each other. In order to control the order of execution and the coordination of threads to some useful end, Mach provides several synchronization mechanisms. The simplest (and coarsest) mechanism is thread-level suspend and resume operations. Each thread has a suspend count, which is incremented and decremented by these operations. A thread whose suspend count is positive remains blocked until the count goes to zero.

Finer synchronization can be obtained through the use of synchronization objects (semaphores or monitors and conditions), which allow a variety of different synchronization styles to be used. Threads can also interact via interprocess communication (IPC). Each of these services is described in more detail in later sections. Basic operations exist to support creation, termination, and getting and setting attributes for threads. Several other control operations exist on threads that can be performed by any thread that has a send right to the intended thread's control port. Threads can be terminated explicitly. They can also be interrupted from the various possible wait situations and caused to resume execution with an indication that they were interrupted. Threads can also be "wired", which means that they are marked as privileged with respect to kernel resources, i.e., they can consume physical memory when free memory is scarce. This is used for threads in the default page-out path. Finally, threads also have several important IPC ports (more precisely, the send or receive rights to these ports), which are used for certain functions. In particular, each thread has a thread self port, which can be used to perform certain operations on the thread by itself. A thread also has a set offault ports which is used when the thread encounters a processor fault during its execution. There is also a distinguished port that can be used for gathering samples of the thread's execution state for monitoring by other threads such as debuggers or program profilers.

Tasks

The basic organizational entity in Mach for which resources are managed is known as a task. Tasks have many objects and attributes associated with them. A task fundamentally comprises three things. A task contains multiple threads, which are the executable entities in the system. A task also has an address space, which represents virtual memory in which its threads can execute. And a task has a port name space, which represents the valid IPC ports through which threads can communicate with other threads in the system. Each of these fundamental objects in a task is discussed in greater detail in the following sections. Note that a task is not, of itself, an executable entity in Mach. However, tasks can contain threads, which are the execution entities. A task has a number of other entities associated with it besides the fundamental ones noted above. Several of these entities have to do with scheduling decisions the kernel needs to make for the threads contained by the task. The scheduling parameters, processor set designation, and host information all contribute to the scheduling of the task's threads. A task also has a number of distinguished interprocess communication ports that serve certain pre-defined functions. Ports and other aspects of interprocess communication are discussed at length in a later section. For now, it is sufficient to know that port resources are accumulated over time in a task. Most of these are managed explicitly by the programmer. The distinguished ports mentioned above generally have to do with establishing connections to several important functions in the system. Mach supplies three "special" ports with each task. The first is the task self port, which can be used to ask the kernel to perform certain operations on the task. The second special port is the bootstrap port, which can be used for anything (it's OS environment-specific) but generally serves to locate other services. The third special port that each task has is the host name port, which allows the task to obtain certain information about the machine on which it is running. Additionally, Mach supplies several "registered" ports with each task that allow the threads contained in the task to communicate with certain higher-level servers in the system (e.g., the Network Name Server, the "Service" Server, and the Environment Server).

Two other useful sets of ports exist for each task that allow fault processing and program state sampling to be performed. Thefault ports of a task provide a common place for processor faults encountered by threads in the task to be processed. Fault processing is described more fully in a later section. The PC sample port allows profiling tools to repeatedly monitor the execution state of the threads in the task. Many operations are possible for tasks. Tasks can be created and terminated. Creation of a new task involves specifying some existing task as a prototype for the initial contents of the address space of the new task. A task can also be terminated, which causes all of the contained threads to be terminated as well. The threads contained in a task can be enumerated and information about the threads can be extracted. Coarse-grain execution of a task (more precisely, the threads in the task) can be controlled through suspend and resume operations. Each task has a suspend count that is incremented and decremented by the suspend and resume operations. The threads in the task can execute as long as the suspend count for the containing task is zero. When the suspend count is positive, all threads in the task will be blocked until the task is subsequently resumed. Finally, the various parameters and attributes associated with a task (e.g., scheduling priority) can be queried and set as desired.

Virtual Memory

Mach supports several features in its virtual memory (VM) subsystem. Both the external client interfaces as well as the internal implementation offer features that are not found in many other operating systems. In broadest terms, the Mach virtual memory system supports a large sparsely populated virtual address space for each of the tasks running in the system. Clients are provided with general services for managing the composition of the address space. Some aspects of the VM system are actually implemented by components that are outside of the micro-kernel, which allows great flexibility in tailoring certain policy functions to different system environments. The internal architecture of the Mach VM system has been divided into machine-independent and machine-dependent modules for maximum portability. Porting to a new processor/MMU architecture is generally a small matter of implementing a number of functions that manipulate the basic hardware MMU structures. Mach has been ported to a number of different processor architectures attesting to the portability of the overall kernel and the virtual memory system in particular. The address space of a Mach task contains a number of virtual memory regions. These regions are pieces of virtual address space that have been allocated in various ways for use by the task. They are the only locations where memory can be legitimately accessed. All references to addresses outside of the defined regions in the address space will result in an improper memory reference fault. A virtual memory region has several interesting attributes. It has a page-aligned starting address and a size, which must be a multiple of the system page size. The pages, in the region all have the same access protections; these access protections can be read-only, read-write, or execute. The pages in a region also have the same inheritance characteristic, which may be used when a new task is created from the current task. The inheritance characteristic for pages in a region can be set to indicate that a new task shoud inherit a read-write copy of the region, that it should inherit a virtual copy of the region, or that it should inherit no copy of the region. A read-write copy of a region in a new address space provides a fully shared mapping of the region between the tasks, while a virtual copy provides a copy-on-write mapping that essentially gives each task its own copy of the region but with efficient copy-on-write sharing of the pages constituting the region.

Every virtual memory region is really a mapping of an abstract entity known as a memory object. A memory object is simply a collection of data that can be addressed in some byte-wise fashion and about which the kernel makes no assumptions. It is best thought of as some pure piece of data that can either be explicitly stored some place or can be produced in some fashion as needed. Many different things can serve as memory objects. Some familiar examples include files, ROMs, disk partitions, or fonts. Memory objects have no pre-defined operations or protocol that they are expected to follow. The data contained in a memory object can only be accessed when it has been tied to a VM region through mapping. After a memory object has been mapped to a region, the data can be accessed via normal memory read and write (load and store) operations. A memory object is generally managed by a special task known as an external memory manager or pager. A pager is a task that executes outside of the micro-kernel much like any other task in the system. It is a user-mode entity whose job is to handle certain requests for the data of the memory objects it supports. As threads in a client task reference the pages in a given region, the kernel logically fills the pages with the data from the corresponding byte addresses in the associated memory object. To accomplish this the kernel actually engages in a well-defined (and onerous) protocol

with the pager whenever it needs to get data for page faults or when it needs to page-out data due to page replacements. This protocol, which is known as the External Memory Management Interface (or EMMI), also handles the initialization sequences for memory objects when they are mapped by client tasks and the termination sequences when any associated memory regions are deallocated by client tasks.

There can be any number of pagers running in the system depending on which memory objects are in use by the various client tasks. Pagers will typically be associated with the various file systems that are mounted at a given time, for example. Pagers could also exist to support certain database applications, which might have needs for operations beyond what is supported by the file system. Pagers could also exist for certain servers that wish to supply data to their clients in non-standard ways (e.g., generating the data computationally rather than retrieving it from a storage subsystem). The micro-kernel always expects to have a certain distinguished pager known as the default pager running in the system. The default pager is responsible for managing the memory objects associated with anonymous virtual memory such stacks, heaps, etc. Such memory is temporary and only of use while a client task is running. As described above, the main entities in the Mach VM system are regions, memory objects, and pagers. Most clients, however, will deal with virtual memory through operations on ranges of memory. A range can be a portion of a region or it could span multiple contiguous regions in the address space. Operations are provided by Mach that allow users to allocate new ranges of virtual memrory in the address space and deallocate ranges as desired. Another important operaition allows a memory object to be mapped into a range of virtual memory as described above. Operations are also available to change the protections on ranges of memory, change the inheritance characteristics, and wire (or lock) the pages of a range into physical memory. It is also possible to read ranges of memory from another task or write into ranges in another task provided that the control port for the task is available. Additional services are available that allow the user to specify the expected reference pattern for a range of memory. This can be used by the kernel as advice on ways to adapt the page replacement policy to different situations. Yet another service is available to synchronize (or flush) the contents of a range of memory with the memory object(s) backing it. Finally services are available to obtain information about regions and to enumerate the contents of a task's address space described in terms of the regions it contains.

Interprocess Communication

Mach has four concepts that are central to its interprocess communications facilities: Ports, Port Sets, Port Rights, and Messages. One of these concepts, Port Rights, is also used by Mach as a means to identify certain common resources in the system (such as threads, tasks, memory objects, etc.).

Ports

Threads use ports to communicate with each other. A port is basically a message queue inside the kernel that threads can add messages to or remove message from, if they have the proper permissions to do so. These "permissions" are called port rights. Other attributes associated with a port, besides port rights, include a limit on the number of messages that can be enqueued on the port, a limit on the maximum size of a message that can be sent to a port, and a count of how many rights to the port are in existence. Ports exist solely in the kernel and can only be manipulated via port rights.

Port Rights

A thread can add a message to a port's message queue if it has a send right to that port. Likewise, it can remove a message from a port's message queue if it has a receive right to that port. Port rights are considered to be resources of a task, not an individual thread. There can be many send rights to a port (held by many different tasks); however, there can only be one receive right to a port. In fact, a port is created by allocating a receive right and a port is destroyed only when the receive right is deallocated (either explicitly or implicitly when the task dies). In addition, the attributes of a port are manipulated through the receive right. Multiple threads (on the same or different tasks) can send to a port at the same time, and multiple threads (on the same task) can receive from a port at the same time. Port rights act as a permission or capability to send messages to or receive messages from a port, and thus they implement a low-level form of security for the system. The "owner" of a port is the task that holds the receive right. The only way for another task to get a send right for a port is if it is explicitly given the right—either by the owner or by any task that holds a valid send right for the port. This is primarily done by including the right in a message and sending the message to another task. Giving a task a send right grants it permission to send as many messages to the port as it wants. There is another kind of port right called a send-once right that only allows the holder to send one message to the port, at which time the send-once right become invalid and can't be used again. Note that ownership of a port can be transferred by sending the port's receive right in a message to another task.

Tasks acquire port rights either by creating them or receiving them in a message. Receive rights can only be created explicitly (by doing a port allocate, as described above); send rights can be created either explicitly from an existing send or receive right or implicitly while being transmitted in a message. A send-once right can be created explicitly or implicitly from a receive right only. When a right is sent in a message the sender can specify that the right is either copied, moved, or a new right created by the send operation. (Receive rights can only be moved, of course.) When a right is moved, the sender looses the right and the receiver gains it. When copied, the sender retains the right but a copy of the right is created and given to the receiver. When created, the sender provides a receive right and a new send or send-once right is created and given to the receiver. When a task acquires a port right, by whatever means, Mach assigns it a name. Note that ports themselves are not named, but their port rights are. (Despite this fact, the creators of Mach decided to refer to the name of a port right with the term port name, instead of the obvious port right name). This name is a scalar value (32-bits on Intel machines) that is guaranteed unique only within a task (which means that several tasks could each have a port name with the same numeric value but that represent port rights to totally different ports) and is chosen at random. Each distinct right held by a task does not necessarily have a distinct port name assigned to it. Send-once rights always have a separate name for each right. Receive and send rights that refer to the same port, however, will have the same name.

Port rights have several attributes associated with them: the type of the right (send, send-once, receive, port set, or dead name), and a reference count for each of the above types of rights. When a task acquires a right for a port to which it already has send or receive rights, the corresponding reference count for the associated port name is incremented. A port name becomes a dead name when its associated port is destroyed. That is, all port names representing send or send-once rights for a port whose receive right is deallocated become dead names. A task can request notification when one of its rights becomes dead. The kernel keeps a system-wide count of the number of send and send-once rights for each port. Any task that holds a receive right (such as a server) can request a notification message be sent when this number goes to zero, indicating that there are no more senders (clients) for the port. This is called a no more senders notification. The request must include a send right for a port to which the notification should be sent.

Port Sets

Port sets provide the ability to receive from a collection of ports simultaneously. That is, receive rights can be added to a port set such that when ai receive is done on the port set, a message will be received from one of the ports in the set. The name of the receive right whose port provided the message is reported by the receive operation.

Messages

A Mach IPC message comprises a header and an in-line data portion, and optionally some out-of-line memory regions and port rights. If the message contains neither port rights nor out-of-line memory, then it is said to be a simple message; otherwise it is a complex message. A simple message contains the message header directly followed by the in-line data portion. The message header contains a destination port send right, an optional send right to which a reply may be sent (usually a send-once right), and the length of the data portion of the message. The in-line data is of variable length (subject to a maximum specified on a per-port basis) and is copied without interpretation. A complex message consists of a message header (with the same format as for a simple message), followed by: a count of the number of out-of-line memory regions and ports, disposition arrays describing the kernel's processing of these regions and ports, and arrays containing the out-of-line descriptors and the port rights

The port right disposition array contains the desired processing of the right, i.e., whether it should be copied, made, or moved to the target task. The out-of-line memory disposition array specifies for each memory range whether or not it should be de-allocated when the message is queued, and whether the memory should be copied into the receiving task's address space or mapped into the receiving address space via a virtual memory copy-on-right mechanism. The out-of-line descriptors specify the size, address, and alignment of the out-of-line memory region. When a task receives a message, the header, in-line data, and descriptor arrays are copied into the addresses designated in the parameters to the receive call. If the message contains out-of-line data, then virtual memory in the receiving task's address space is automatically allocated by the kernel to hold the out-of-line data. It is the responsibility of the receiving task to deallocate these memory regions when they are done with the data.

Message Transmission Semantics

Mach IPC is basically asynchronous in nature. A thread sends a message to a port, and once the message is queued on the port the sending thread continues execution. A receive on a port will block if there are no message, queued on the port. For efficiency there is a combined send/receive call that will send a message and immediately block waiting for a message on a specified reply port (providing a synchronous model). A time-out can be set on all message operations which will abort the operation if the message is unable to be sent (or if no message is available to be received) within the specified time period. A send call will block if it uses a send-right whose corresponding port has reached its maximum number of messages. If a send uses a send-once right, the message is guaranteed to be queued even if the port is full. Message delivery is reliable, and messages are guaranteed to be received in the order they are sent. Note that there is special-case code in Mach which optimizes for the synchronous model over the asynchronous model; the fastest IPC round-trip time is achieved by a server doing a receive followed by repeated send/receive's in a loop and the client doing corresponding send/receive's in a loop on its side.

Port Rights as Identifiers

Because the kernel guarantees both that port rights cannot be counterfeited and that messages cannot be misdirected or falsified, port rights provide a very reliable and secure identifier. Mach takes advantage of this by using port rights to represent almost everything in the system, including tasks, threads, memory objects, external memory managers, permissions to do system-privileged operations, processor allocations, and so on. In addition, since the kernel can send and receive messages itself (it represents itself as a "special" task), the majority of the kernel services are accessed via IPC messages instead of system-call traps. This has allowed services to be migrated out of the kernel fairly easily where appropriate.

Synchronization

Currently, Mach provides no direct support for synchronization capabilities. However, conventional operating systems routinely provide synchronization services. Such synchronization services employ many well-known mechanisms, such as semaphores and monitors and conditions, which are described below. Semaphores are a synchronization mechanism which allows both exclusive and shared access to a resource. Semaphores can be acquired and released (in either an exclusive or shared mode), and they can optionally specify time-out periods on the acquire operations. Semaphores are optionally recoverable in the sense that when a thread that is holding a semaphore terminates prematurely, the counters associated with the semaphore are adjusted and waiting threads are unblocked as appropriate.

Monitors and conditions are a synchronization mechanism which implements a relatively more disciplined (and safer) style of synchronization than simple semaphores. A monitor lock (also called a mutex) is essentially a binary semaphore that enables mutually exclusive access to some data. Condition variables can be used to wait for and signify the truth of certain programmer-defined Boolean expressions within the context of the monitor. When a thread that holds a monitor lock needs to wait for a condition, the monitor lock is relinquished and the thread is blocked. Later, when a another thread that holds the lock notifies that the condition is true, a waiting thread is unblocked and then re-acquires the lock before continuing execution. A thread can also perform a broadcast operation on a condition, which unblocks all of the threads waiting for that condition. Optional time-outs can also be set on the condition wait operations to limit the time a thread will wait for the condition.

Scheduling

Since Mach is multiprocessor capable, it provides for the scheduling of threads in a multiprocessor environment. Mach defines processor sets to group processors and it defines scheduling policies that can be associated with them. Mach provides two scheduling policies: timeshare and fixed priority. The timeshare policy is based on the exponential average of the threads' usage of the CPU. This policy also attempts to optimize the time quantum based on the number of threads and processors. The fixed priority policy does not alter the priority but does round-robin scheduling on the threads that are at the same priority. A thread can use the default scheduling policy of its processor set or explicitly use any one of the scheduling policies enabled for its processor set. A maximum priority can be set for a processor set and thread. In Mach the lower the priority value, the greater the urgency.

Faults

The Mach fault handling services are intended to provide a flexible mechanism for handling both standard and user-defined processor faults. The standard kernel facilities of threads, messages, and ports are used to provide the fault handling mechanism. (This document uses the word "fault" where the Mach documentation uses the word "exception". Such terminology has been changed herein to distinguish hardware faults from the exception mechanism of the C++ language). Threads and task have fault port(s). They differ in their inheritance rules and are expected to be used in slightly different ways. Error handling is expected to be done on a per-thread basis and debugging is expected to be handled on a per-task basis. Task fault ports are inherited from parent to child tasks, while thread fault ports are not inherited and default to no handler. Thread fault handlers take precedence over task fault handlers. When a thread causes a fault the kernel blocks the thread and sends a fault message to the thread's fault handler via the fault port. A handler is a task that receives a message from the fault port. The message contains information about the fault, the thread, and the task causing the fault. The handler performs its function according to the type of the fault. If appropriate, the handler can get and modify the execution state of the thread that caused the fault. Possible actions are to clear the fault, to terminate the thread, or to pass the fault on to the task-level handler. Faults are identified by types and data. Mach defines some machine-independent fault types that are supported for all Mach implementations (e.g., bad access, bad instruction, breakpoint, etc.). Other fault types can be implementation dependent (e.g., f-line, co-processor violation, etc.).

Host and Processor Sets

Mach exports the notion of the host, which is essentially an abstraction for the computer on which it is executing. Various operations can be performed on the host depending on the specific port rights that a task has for the host. Information that is not sensitive can be obtained by any task that holds a send right to the host name port. Examples of such information include the version of the kernel or the right to gain access to the value of the system clock. Almost all other information is considered sensitive, and a higher degree of privilege is required to get or manipulate the information. This added level of privilege is implied when a task holds a send right to the host control port (also known as the host privilege port). This right must be given out very carefully and selectively to tasks, because having this right enables a task to do virtually everything possible to the kernel, thus by-passing the security aspects of the system supported by the IPC services. Various operations can be performed with this added privilege, including altering the system's clock setting, obtaining overall performance and resource usage statistics for the system, and causing the machine to re-boot.

Mach also exports the notions of processors and processor sets, which allow tasks to more carefully specify when and on what processors its threads should execute. Processors and processor sets can be enumerated and controlled with the host privilege port. A processor represents a particular processor in the system, and a processor set represents a collection of processors. Services exist to create new processor sets and to add processors to a set or remove them as desired. Services also exist to assign entire tasks or particular threads to a set. Through these services a programmer can control (on a coarse grain) when the threads and tasks that constitute an application actually get to execute. This allows a programmer to specify when certain threads should be executed in parallel in a processor set. The default assignment for tasks and threads that do not explicitly use these capabilities is to the system default processor set, which generally contains any processors in the system that aren't being used in other sets.

Security

Mach may include other categories of services in addition to those described above. For example, Mach may include services relating to security. In accordance with the Mach security services, every task carries a security tokert, which is a scalar value that is uninterpreted by Mach. There is a port called the hcost security port that is given to the bootstrap task and passed on to the trusted security sever. A task's security token can be set or changed by any task that holds a send right to the host security port, while no special permissions are needed to determine the value of a tasks security token (other than holding the task's control port, of course). At the time a Mach IPC message is received, the security token of the sender of the message is returned as one of the output parameters to the receive function. Tasks that hold the host security port can send a message and assign a different security token to that message, so that it appears to have come from another task. These services can be used by upper layers of the system to implement various degrees of security.

Wrapper Class Library

This section provides an area-by-area description of the object-oriented interface for the services provided by the Mach micro-kernel. This object-oriented interface to the Mach services represents the wrapper class library **402** as implemented by the code library **110**. The wrapper class library **402** includes thread classes **404**, task classes **406**, virtual memory classes **408**, IPC classes **410**, synchronization classes **412**, scheduling classes **414**, fault classes **416**, and machine classes **418** are discussed. The wrapper class library **402** may include additional classes, such as security classes **420**, depending on the services provided by the underlying operating system **114**. Each area is described with a class diagram and text detailing the purpose and function of each class. Selected methods are presented and defined (where appropriate, the parameter list of a method is also provided). Thus, this section provides a complete operational definition and description of the wrapper class library **402**. The implementation of the methods of the wrapper class library **402** is discussed in a later section.

The class diagrams are presented using the well-known Booch icons for representing class relationships and cardinality. These Booch icons are presented in FIG. **17** for

convenience purposes. The Booch icons are discussed in *Object Oriented Design With Applications* by Grady Booch, referenced above. The wrapper class library **402** is preferably implemented using the well-known C++ computer programming language. However, other programming languages could alternatively be used. Preferably, the class descriptions are grouped into SPI (System Programming Interface), API (Application Programming Interface), Internal, and "Noose" methods—indicated by #ifndef statements bracketing the code in question (or by comments for Noose methods). SPI interfaces are specific to the particular computer platform being used. For illustrative purposes, the wrapper class library **402** is presented and described herein with respect to a computer platform operating in accordance with the IBM MicroKernel (which is based on Mach Version 3.0) or compatible. Persons skilled in the relevant art will find it apparent to modify the SPI classes to accommodate other computer platforms based on the teachings contained herein.

API interfaces are included in the wrapper class library **402** regardless of the platform the system is running on. The Internal interfaces are intended for use only by low-level implementors. The Noose methods are provided solely to enable an application **130** operating with the wrapper **128** to communicate with an application **134** (or server) that was written to run on Mach **114** directly. They provide access to the raw Mach facilities in such a way that they fall outside of the intended object-oriented programming model established by the wrapper **128**. Use of Noose methods is highly discouraged. The SPI and API (and perhaps the Internal) classes and methods are sufficient to implement any application, component, or subsystem.

Thread Classes

FIG. **5** is a class diagram **501** of the thread classes **404** and the task classes **406**. The thread classes **404** provide an object-oriented interface to the tasking and threading functionality of Mach **114**. A number of the thread classes **404** are handle classes (so noted by their name), which means that they represent a reference to the corresponding kernel entity. The null constructors on the handle classes create an empty handle object. An empty handle object does not initially correspond to any kernel entity—it must be initialized via streaming, an assignment, or a copy operation. Calling methods on an empty handle will cause an exception to be thrown. Multiple copies of a handle object can be made, each of which point to the same kernel entity. The handle objects are internally reference-counted so that the kernel entity can be deleted when the last object representing it is destroyed.

TThreadHandle is a concrete class that represents a thread entity in the system. It provides the methods for controlling and determining information about the thread. It also provides the mechanism for spawning new threads in the system. Control operations include killing, suspending/resuming, and doing a death watch on it. Constructing a TThreadHandle and passing in a TThreadProgram object causes a new thread to be constructed on the current task. The first code run in the new thread are the Prepare( ) and Run( ) methods of the TThreadProgram object. Destroying a TThreadHandle does not destroy the thread it represents. There may also be a cancel operation on the TThreadHandle object. Note that each TThreadHandle object contains a send right to the control port for the thread. This information is not exported by the interface, in general, but because it does contain a port right the only stream object a TThreadProgram can be streamed into is a TIPCMessageStream.

Attempting to stream into other TStream objects will cause an exception to be thrown.

TThreadHandle provides a number of methods for use by debuggers and the runtime environment, and for supporting interactions with Mach tasks running outside of the environment established by the wrapper **128**. These methods include getting and setting the state of a thread, spawning an "empty" thread in another task, getting the thread's fault ports, returning a right to the thread's control port, and creating a TThreadHandle handle from a thread control port send right.

As noted above, the wrapper **128** establishes a computing environment in which the applications **130** operate. For brevity, this computing environment established by the wrapper **128** shall be called CE. With regard to the wrapper **128**, TThreadHandle spawns a CE runtime thread on the current task. A thread can also be spawned on another task, instead of on the current task, by using the CreateThread methods in the TTaskHandle class and in subclasses of TTaskHandle. (Creating a thread on another task is not recommended as a general programming model, however.) To spawn a CE thread on another CE task, the TCETaskHandle::CreateThread method is used by passing it a TThreadProgram describing the thread to be run. To spawn a non-CE thread (that is, a thread which does not operate in the computing environment established by the wrapper **128**), the CreateThread method is used on the appropriate subclass of TTaskHandle (that is, the subclass of TTaskHandle that has been created to operate with the other, non-CE computing environment). For example, to spawn an IBM OS2 thread on an OS2 task, you might use a TOS2TaskHandle::CreateThread method. It is not possible to run a CE thread on a non-CE task, nor is it possible to run a non-CE thread on a CE task.

TThreadHandle includes the following methods:

TThreadHandle (const TThreadProgram& copyThreadCode): creates a new thread in the calling task—makes an internal COPY of the TThreadProgram, which is deleted upon termination of the thread.

TThreadHandle (TThreadProgram* adoptThreadCode): creates a new thread in the calling task—ADOPTs adoptThreadCode which is deleted upon termination of the thread. The resources owned by the thread are also discarded. A copy of the TThreadProgram is NOT made.

TThreadHandle (EExecution yourself creates a thead handle for the calling thread.

TStream streams in a TThreadHandle object to a TIPCMessageStream.

CopyThreadSchedule ( ) returns a pointer to the Scheduling object (e.g., TServerSchedule, TUISchedule etc) that is used to schedule the objiect. Allocates memory for the TThreadSchedule object which has to be disposed of by the caller.

SetThreadSchedule (const TThreadSchedule& newSchedule) sets the scheduling object in the thread to the newSchedule object. This allows one to control the way a thread is scheduled

GetScheduleState (TThreadHandle& theBlockedOnThread) allows one to query the current state of the thread (theBlockedOnThread) on which this thread is blocked.

CancelWaitAndPostException ( ) const causes a blocking kernel call to be unblocked and a TKernelException to be thrown in the thread (*this).

WaitForDeathOf ( ) const performs death watch on the thread—blocks calling thread until the thread (*this) terminates. CreateDeathInterest ( ) creates a notification interest for the death of the thread (*this). When the thread termi nates the specified TInterest gets a notification.

TThreadProgram is an abstract base class that encapsulates all the information required to create a new thread. This includes the code to be executed, scheduling information, and the thread's stack. To use, it must be subclassed and the Begin and Run methods overridden, and then an instantiation of the object passed into the constructor for TThreadHandle to spawn a thread. The Begin routine is provided to aid startup synchronization; Begin is executed in the new thread before the TThreadHandle constructor completes, and the Run routine is executed after the TThreadHandle constructor completes. The methods CopyThreadSchedule and GetStackSize return the default thread schedule and stack size. To provide values different from the default, these methods should be overridden to return the desired thread schedule and/or stack size. TThreadProgram includes the following methods:

TThreadProgram (const TText& taskDescripteon): TaskDescription provides a text description of a task that can be access via the TTaskHandle::GetTaskDescription method. Only in effect if the object is passed a TTaskHandle constructor. If default constructor is used instead, the interface will synthesize a unique name for TTaskHandle::GetTaskDescription to return.

GetStackSize ( ) returns the size of the stack to be set up for the thread. Override this method if you don't want the "default" stack size.

GetStack ( ): Used to set up the thread's stack. Override this method if you want to provide your own stack.

Run ( ) represents the entry point for the code to be run in the thread. OVERRIDE THIS METHOD to provide the code the thread is to execute.

Task Classes

See FIG. **5** for a class diagram of the task classes **406**.

TTaskHandle is a concrete base class that encapsulates all the attributes and operations of a basic Mach task. It can be used to refer to and control any task on the system. TTaskHandle cannot be used directly to create a task, however, because it doesn't have any knowledge about any runtime environment. It does provide sufficient protocol, via protected methods, for subclasses with specific runtime knowledge to be created that can spawn tasks (TCETaskHandle, below, is an example of such a class). TTaskHandle objects can only be streamed into and out of TIPCMessageStreams and sent via IPC to other tasks, and they are returned in a collection associated with TCETaskHandle. The task control operations associated with a TTaskHandle include killing the task, suspending and resuming the task, and doing a deathwatch on the task. The informational methods include getting its host, getting and setting its registered ports, enumerating its ports or virtual memory regions, getting its fault ports, enumerating its threads, etc. TTaskHandle includes the following methods:

TTaskHandle (EExecutionThread) creates a task handle for the specified thread.

Suspend ( ) suspends the task (i.e., all threads contained by the task). Resume ( ) resumes the task (i.e., all threads contained by the task).

Kill ( ) terminates the task—all threads contained by the task are terminated.

WaitForDeathOf ( ) performs death watch on the task—The calling thread blocks until the task (*this) terminates. CreateDeathInterest ( ) creates a notification interest for the death of the task. The thread specified in the TInterest object gets a notification when the task (*this) terminates.

AllocateMemory (size_t howManyBytes, TMemorySurrogate& newRange) allocates a range of (anonymous) virtual memory anywhere in the task's address space. The desired size in bytes is specified in howManyBytes. The starting address (after page alignment) and actual size of the newly allocated memory are returned in newRange.

AllocateReservedAddressMemory (const TMemorySurregate& range, TMemorySurrogate& newRange) allocates a range of (anonymous) virtual memory at a specified reserved address in the task's address space. The range argument specifies the address and size of the request. The newRange returns the page aligned address and size of the allocated memory.

GetRemotePorts (TCollection<TRemotePortRightHandle>& thePortSet) gets list of ports on *this task. The caller is responsible for de-allocating the memory in the returned Collection.

virtual void CreateFaultAssociationCollection (TCollection<FaultAssociation>& where) return Fault Ports registered for this Task.

TCETaskHandle is a subclass of TTaskHandle that represents a Mach task executing with the CE runtime system (recall that that CE represents the computing environment established by the wrapper **128**), and embodies all the knowledge required to set up the CE object environment. It can be used to spawn a new task by passing a TThreadProgram into its constructor. The new task is created with a single thread, which is described by the TThreadProgram object passed into the TCETaskHandle constructor. There is also a constructor that will allow a TCETaskHandle to be constructed from a TTaskHandle. To insure that a non-CE-runtime task is not wrapped with a TCETaskHandle, the constructor consults the CE loader/library server (that is, the loader/library server operating in the CE environment) to make sure the task being wrapped has been registered with it. This is done automatically (without any user intervention). TCETaskHandle includes the following methods:

TCETaskHandle (const TThreadProgram& whatToRun) creates a new task and a thread to execute specified code. The new thread executes the code in 'whatToRun'.

TCETaskHandle (EExecutionTask) wraps task of currently executing thread.

TCETaskHandle (const TThreadProgram& whatToRun, const TOrderedCollection<TLibrarySearcher>& librarySearchers) creates a new task and a thread to execute specified code with specified ibrary search. The librarysearchers specifies the list of libraries to be used for resolving names.

TCETaskHandle (const TTaskHandle& aTask) creates a CE task object from a generic task object.

AddLibrarySearcher (const TLibrarySearcher& newLibSearcher) adds a library searcher for the task—loader uses newLibrarySearcher first to re.solve lib refernceces i.e. the newLibrarySearcher is put on the top of the collection used to resolve references.

GetTaskDescription (TText& description) const returns a string description of the task—gets the string from the associated TThreadProgram of the root thread (passed to constructor). The string is guaranteed to be unique, and a string will be synthesized by the interface if no description is passed to the TThreadProgram constructor.

NotifyUponCreation (TInterest* notifyMe) synchronously notifies the caller of every new task creation in the system. There is no (*this) task object involved. The task from which this call originates is the receiver of the notification.

Virtual Memory Classes

FIG. **6** is a class diagram **601** for the virtual memory classes **408**. Note that TTaskHandle is a class that represents a task. TTaskHandle has already been discussed under the Task classes **406** section. For virtual memory operations, objects of type TTaskHandle serve to specify the address space in which the operation is to occur. Most of the virtual memory operations that can be performed in Mach are represented as methods of TTaskHandle. The various methods of TTaskHandle that operate on virtual memory take TMemorySurrogate objects as parameters. See the various methods under the TTaskHandle description for further details. A number of the memory classes have copy constructors and/or assignment operators. It should be noted that the memory classes contain references to the memory and not the actual memory itself. Therefore when memory class objects are copied or streamed, only the references within them are copied and not the actual memory. The TMemorySurrogate class contains explicit methods for doing copies of the memory it references.

TMemorySurrogate is a class that represents a contiguous range of memory in the virtual address space. It has a starting address and a size (in bytes). TMemorySurrogates can be used to specify ranges of memory on which certain operations are to be performed. They are typically supplied as arguments to methods of TTaskHandle that manipulate the virtual memory in the address space associated with the task. This class is used to specify/supply a region of memory with a specific size. The class itself does not allocate any memory. It just encapsulates existing memory. It is the responsibility of the caller to provide the actual memory specified in this class (the argument to the constructor). This class is NOT subclassable.

TChunkyMemory is an abstract base class that manages memory in chunks of a specified size. Memory is allocated in chunks (of the specified chunkSize), but the user still views the memory as a series of bytes. TChunkyMemory includes the following methods:

LocateChunk (size_t where, TMemorySurrogate& theContainingRange) looks up in the collection of chunks and returns in theContainingRange the address of the memory and the chunksize.

CutBackTo (size_t where) cuts back to the chunk containing "where" i.e. the chunk at the offset where will become the last chunk in the list.

AllocateMemoryChunk (TMemorySurrogate& theAllocatedRange) is called by clients to allocate new chunks of memory as needed. Returns the allocated range.

THeapChunkyMemory is a concrete class that manages chunky memory on a heap.

TVMChunkymemory is a concrete class that manages chunky memory using virtual memory.

TMemoryRegionInfo is a class used with virtual memory regions in a task's address space. It provides memory attribute information (like Inheritance, Protection etc.). It also provides access to the memory object associated with the region of memory and to the actual memory range encapsulated in the memory region. Nested inside TMemoryRegionInfo is the TMemoryAttributeBundle class that defines all the memory attributes of any memory region. This is useful when one wants to get/set all the memory attributes (or to re-use memory attributes with minimal changes). TMemoryAttributeBundle is also used in the class TTaskHandle to deal with mapping memory objects into a task's address space. TMemoryRegionInfo includes the following methods:

EMemoryProtection {kReadOnly, kReadWrite, kExecute} specifies the protection for the memory.

EMemoryInheritance {kDontInherit, kReadWriteInherit, kCopyInherit} specifies the inheritance attribute for the memory.

EMemoryBehavior {kReferenceSequential, kReferenceReverseSequential, kReferenceRandom} specifies how memory might be referenced.

EMemoryAttribute {kCacheable, kMigrateable} specifies how machine specific properties of memory might be managed.

EMemoryAdvice {kWillUse, kWontUse} specifies how memory will be used.

TMemoryObjectHandle is a base class that represents the notion of a Mach memory object. It embodies the piece of data that can be mapped into virtual memory. System servers that provide TMemoryObjectHandles to clients will subclass from TMemoryObjectHandle in order to define specific types of memory objects such as files, device partitions, etc. For the client of general virtual memory services, the main use of TMemoryObjectHandle and the various subclasses is to provide a common type and protocol for data that can be mapped into a task's address space.

TChunkyStream is a concrete class (derived from TRandomAccessStream) that embodies a random access stream backed by chunks of memory. The chunk size can be specified or a default used. The chunks can be enumerated. This class provides a common function of theTMemory class without incurring the overhead of maintaining the memory as contiguous. If the remaining functionality of TMemory is required other classes could be defined.

TContiguousMemoryStream is a concrete class that uses contiguous memory (supplied by the client). Since it is derived from TRandonAccessStream, all random access operations (like Seeko( )) are applicable to TContiguousMemoryStream objects.

InterProcess Communication (IPC) Classes

The IPC classes **410** represent the Mach IPC message abstraction. Note that all messaging behavior is on the message classes; the port right classes are basically for addressing the message. The usage model is preferably as follows: A TIPCMessageStream is instantiated, objects are streamed into it, and the TIPCMessageStream::Send method is called with an object representing a destination send-right passed as an argument. To receive a message, a TIPCMessageStream is instantiated and its Receive method called, passing in a receive-right object as an argument. When the Receive returns, objects can be streamed out of the TIPCMessageStream object. Note that the TIPCMessageStream objects are reusable. A more detailed description of the IPC classes **410** follow with reference to FIG. **7**, which illustrates a class diagram **702** of IPC message classes, FIG. **8** which illustrates a class diagram **802** of IPC out-of-line memory region classes, and FIG. **9** which illustrates a class diagram **902** of IPC port right classes.

Message Classes

MIPCMessage is an abstract base class that represents a Mach IPC message. It provides all the methods for setting up the fields of the header, the disposition array, and the port and out-of-line memory arrays. It also contains all the protocol for message sending and receiving. It provides rudimentary protocol (exported as a protected interface) to child classes for setting up the in-line message data. The classes TIPCMessageStream and TIPCPrimitiveMessage derive from this class, and provide the public methods for adding data to the message. MIPCMessage includes the following methods:

GetReplyPort (TPortSendSideHandle& replyPort) is valid for receive side only. Returns a reply port object, if one was sent with the message. Only returns it the first time this is called after message is received. Otherwise returns false.

TSecurityToken GetSendersSecurityToken( ) is valid for receive side only. Returns the security token of the task that sent this message.

SetSendersSecurityToken(const TSecurityToken& impostorSecurityToken, const TPortSendRight& hostSecurityPort) is valid for send side only. The next time the message is sent, it will carry the specified security token instead of the one for the task that actually does the send. Takes effect ONLY FOR THE NEXT SEND, and then reverts back to the actual sender's security token value.

Methods for sending/receiving IPC messages (Note that all these methods have an optional TTime timeout value. If you don't want a timeout, specify kPositiveInfinity. All these methods replace any existing value for reply port in msg header. For those methods that allow specification of a reply port, the disposition of the reply port right, as well as the port right itself, is passed via a MIPCMessage::TReplyPortDisposition object. This is the only wa y to set the reply port, since the disposition state is only valid for the duration of the send. Objects for port rights whose dispositions are MOVE become invalid once the send takes place.):

Send (const TPortSendSideHandle& destinationport, const TTime& timeout=kPositiveInfinity) is a one-way, asynchronous send.

Send (const TPortSendSideHandle& destinationport, const TReplyPortDisposition& replyPort, const TTime& timeout=kPositiveInfinity) is an asynchronous send, with send (-once) reply port specified.

Receive (const TPortReceiveSideHandle& sourcePort, const TTime& timeout=kPositiveInfinity) is a "blocking" receive.

SendAndReceive (const TPortSendSideHandle& sendPort const TPortReceiveSideHandle& receivePort, const TTime& timeout=kPositiveInifinity) sends a message, blocks and receives a reply (reply port is a send-once right constructed from receivePort).

SendAndReceive (const TPortSendSideHandle& sendport, const TPortReceiveSideHandle& receivePort, MIPCMessage& receiveMsg, const TTime& timeout=kPositiveInfinity) send message, block and receive reply(reply port is a send-once right constructed from receivePort). Message is received into a new message object to avoid overwrite.

ReplyAndReceive (const TPortSendSideHandle& replyToPort, const TPortReceiveSideHandle&

receivePort, const TTime& timeout=kpositiveInfinity): sends back a reply, blocks and receives a new message.

ReplyAndReceive (const TPortSendSideHandle& replyToPort, const TPortReceiveSideHandle& receiveport, MIPCMessage& receiveMsg, const TTime& timeout=kPositiveInfinity) sends back a reply, blocks and receives a new message.

Subclasses' methods for getting/setting port right fields in header (Remote and Local Ports: On SEND side, REMOTE PORT specifies the destination port, and LOCAL PORT specifies the reply port. On RECEIVE side, REMOTE PORT specifies the reply port (port to be replied to) and LOCAL PORT specifies the port received from. The way the port was (or is to be) transmitted is returned in theDisposition. It can have values: MACH_MSG_TYPE_(MOVE_RECEIVE, MOVE_SEND, MOVE_SEND_ONCE, COPY_SEND, MAKE_SEND, MAKE_SEND_ONCE}.):

GetRemotePort: pass in the remote port right, and specify the disposition.

PORT RIGHT methods:

MovePortRightDescriptor: sender is giving away the port: right to the destination. Works on Send, SendOnce, and Receive rights.

CopyPortSendRightDescriptor: sender is creating a copy of the send right at the destination.

MakePortSendRightDescriptor: a new send right will be created at the destination.

MakePortSendOnceRightDescriptor: a new send once right will be created at the destination.

TIPCMessageStream is a concrete class that provides a stream-based IPC messaging abstraction. This is the recommended class to be used for IPC operations. It derives from MIPCMessageDescriptor and from TStream. To send a message, a user of TIPCMessageStream streams in the data to be sent, including port-rights (TPortRightHandle derivatives), out-of-line memory regions (TOutOfLineMemorySurrogate), port-right arrays (TPortRightHandleArray), objects containing any or all of these, and any other object or data type desired. TIPCMessageStream will automatically set up the appropriate data structures for the port rights, port right arrays, and out-of-line memory in the message header, and put a place holder in the stream so that these elements will be streamed out of the message in the appropriate place in the stream. Once the data has been streamed in, the message is sent using the Send method, supplying the appropriate destination port right (TPortSenderHandle) and optionally a reply port. To receive a message, the Receive method is called, supplying a receive right (TPortReceiverHandle) for the port to be received from. The data just received can then streamed out of the TIPCMessageStream.

TIPCMessageStream also provides two methods for doing a combined send and receive operation, designed to provide commonly-used message transmission semantics (and to take advantage of fast-paths in the Mach microkernel). SendAndReceive does a client-side synchronous-style send and then blocks in a receive to pick up the reply message. ReplyAndReceive does a server-side send of (presumably) a reply message and then immediately blocks in a receive awaiting the next request. Both calls require that a destination port and a receive port be specified. Additionally, the SendAndReceive method automatically creates the appropriate send-once right from the supplied receive right and passes it along as the reply port.

TIPCPrimitiveMessage is a concrete class that derives from MIPCMessage and provides a more rudimentary, low level interface to the Mach message system. Data is provided to and from the message header and body via get and set calls. There is no streaming capability. This is a concrete class that represents a Mach IPC message. In-line data is added to the message by passing in a TMemorySurrogate. Port rights, arrays, and OOLdata must be added and extracted explicitly using the appropriate methods.

TOutOfLineMemorySurrogate represents an out-of-line memory range that is to be included in an IPC message. It uses TMemorySurrogate in its implementation, and only adds disposition information to the startaddress and length information already contained in TMemorySurrogate. This class is the same as a TMemorySurrogate, except it includes disposition information used when sending the message, and may represent the storage associated with the range. This class includes streaming operators, methods to set/get the range, and methods to set/get disposition information.

Port Rights

The following classes represent all the valid types of Mach port rights. These classes all share the following general behaviors: In general, when a port right object is instantiated it increments the kernel's reference count for that right, and when a port right object is destroyed it decrements the kernel's port right reference count. However, note that port right objects are handles for the "real" kernel port right entities. They can be copied, in which case there may be two objects that refer to the same kernel port right entity. These copies are reference counted internally so that when all the objects that refer to a port right are deleted, the kernel's port right reference count is decremented. When a port right becomes a dead name (i.e., when the port it belonged to is destroyed), attempts to use methods on the representative object will throw an exception (excluding those operations, like setting the reference counts, that are valid on dead names).

TPortRightHandle is an abstract base class that represents the notion of a port right. It contains all the protocol common to each type of port right, such as getting the port name, requesting dead name notification, testing to see if the port right is a dead name, etc. (The port name is returned as a mach_port_name_t type, and is provided as a way to interact with Mach servers not written using the object wrappers.) It also serves as a common super class to allow a generic type representing all types of ports to be passed polymorphically. TPortSenderHandle and TPortReceiverHandle derive from these classes. This class includes methods for streaming support (This class and any classes that contain it can only be streamed into or out of the TIPCMessageStream class. Attempting to stream into any other TStream will throw an exception at runtime.), Getters/Setters, and methods for requesting notifications (Must provide a send-once right that the notification is to be sent to. MAKE a send-once right by passing (by reference) a receive right. MOVE a send-once right by ADOPTING a send-once right.)

TPortSenderHandle is an abstract class that represents any port right that an IPC message can be sent to. E.g., this is the type that MIPCMessage::Send takes as the destination and reply ports. The classes TPortSendRightHandle and TPortSendOnceRightHandle derive from this class. This class includes methods for streaming support, and Getters and setters.

TPortSendRightHandle represents a port send right. It supports all the typical operations that can be performed on a send right. It is created by passing a valid TPortReceiveRightHandle or TPortSendRightHandle into the constructor, or by streaming it out of a TIPCMessageStream. This class includes methods that create an empty TPortSendRightHandle object without affecting the kernel reference counts, constructors that create a new Send Right in the current task, methods for Streaming Support, and Getters and setters.

TPortSendOnceRightHandle represents a port send-once right. It supports all the typical operations that can be performed on a send-once right. It is created by passing a valid TPortRecieveRightHandle into the constructor, or by streaming it out of a TIPCMessageStream. When a message is sent to an object of this class, making the send-once right invalid, all subsequent attempts to send to this object will cause an exception to be thrown. In addition, the object will be marked as invalid and attempts to use methods of the object will also cause exceptions to be thrown (except for methods for initializing the object, obviously). This class includes Constructors that create a TPortSendOnceRightHandle object without, Constructors that create a new Send Once right on the current task, methods for Streaming Support, and Getters and setters.

TPortReceiverHandle is an abstract class that represents any port right that an IPC message can be received from. E.g., this is the type that MIPCMessage::Receive takes as the port to receive from. The classes TPortRightReceiveHandle and TPortSetHandle derive from this class. This class includes methods for Streaming Support, and Getters and setters.

TPortReceiveRightHandle represents a port receive right. It supports all the typical operations that can be performed on a receive right, such as requesting no-more-senders notification, setting and getting the port's maximun message size and queue length, getting and setting its make-send count, etc. If a TPortReceiveRightHandle is instantiated (other than with the null or copy constructors) it causes a port and receive right to be created. The copy constructor creates another object (an alias) that references the same receive right. These objects are internally reference counted, such that when the last object referencing a particular receive right is destroyed, it destroys the receive right (and the port) it represents, causing all extant rights to that port to become dead names. This class is a concrete class that represents a port receive right. By definition, the actual kernel port entity is created when a receive right is created, and destroyed when a receive right is destroyed. Since this class is a handle, creation and destruction of the receive right is not necessarily tied to creation and deletion of a TPortReceiveRightHandle. For example, the default constructor does not actually create a receive right, but just an empty object. This class includes Constructors that create a TPortReceiveRightHandle object without creating a port or affecting the kernel reference counts, Constructors that create new Receive Rights and Ports, methods to make an uninitialized object valid, creating a receive right (and therefore a port) in the process, Streaming Support, Receive Right/Port manipulation methods, Getters and setters, and Methods for requesting notifications.

TPortSetHandle represents a port set. It has methods for adding, removing, and enumerating the TPortReceiveRightHandle objects representing the receive rights contained in the port set, methods for getting and setting its make send count, etc. If a TPortSetHandle is instantiated with a default constructor, it causes a port set to be created. If it is instantiated using the copy constructor, an alias is created for the same port set. When the last object representing a particular port set is deleted, it destroys the port set. This class cannot be streamed.

TPortRightHandleArray is a concrete class that represents an array of port rights that can be sent as an out-of-line descriptor in an IPC message. It can contain any kind of port right, and the disposition of the port right (i.e., how it is to be transferred to the target task) is specified for each port right in the array. This class implements an array of port rights that can be sent as an out-of-line descriptor in an IPC message (along with port rights and out-of-line memory). This class includes methods for Streaming Support, Methods to add elements to the array (SEND SIDE), and Methods to remove elements from the array (RECEIVE SIDE).

TRemotePortRightHandle is a concrete class that is used to refer to a port right in another task. It does not contain most of the usual port right methods, since it is not intended to be used to perform those types of functions but merely to act as a name or handle for the remote port right. Constructing this class DOES NOT create a port right—it only represents a port right that already exists in another task.

Wait Groups

MWaitable and TWaitGroup are classes that provide for message dispatching and the ability to wait for more than one type of message source at the same time. TWaitGroup is a class that provides the ability to set up a collection of objects derived from MWaitable such that a thread can use the Wait method to receive a message from any of the MWaitable objects. It also provides for automatic dispatching of the received message. Multi-Wait Operations are called repeatedly by a task to receive messages. They are multi thread safe so there can be more than one thread servicing messages. This class includes methods for manipulating the members of the TWaitGroup. For example, GetListOfWaitables returns a list of MWaitables in this group. MWaitable is an abstract base class that associates a port with an internal handler method (HandleIPCMessage). It also provides a common base class for collecting together via the TWaitGroup class Receive Rights and other classes based on Receive Rights.

TWaitablePortReceiveRightHandle is a convenience class that derives from both TPortReceiveRightHandle and MWaitable. It is an abstract base class whose subclasses can be added to a TWaitGroup to provide for multi-wait/dispatching of Mach message IPC with other MWaitable subclasses.

Synchronization Classes

FIG. **10** is a class diagram **1002** of the synchronization classes **412**, which are used to invoke the synchronization services of Mach. As discussed above, the synchronization classes **412** employ semaphores and monitors and conditions. TSemaphore is a class that provides the general services of a counting semaphore. When acquiring a semaphore, if some other task already has acquired the semaphore, the calling thread blocks (no exception thrown). But if the semaphore is invalid for some reason, an exception is thrown. This class includes the following methods:

Acquire: acquire the semaphore in exclusive mode.

Acquire (const Trime& maximumWait): acquire the semaphore in exclusive mode, with time-out.

AcquireShared ( ): acquire the semaphore in shared mode.

AcquireShared (const TTime& maximumWait): acquire the semaphore in shared mode, with time-out.

Release ( ): release the previously acquired semaphore.

AnyThreadsWaiting ( ): returns true if the semaphore currently has threads waiting to acquire it.

TLocalSemaphore is a class that represents a counting semaphore that can be acquired in an exclusive or shared mode. The major operations are acquire and release. An optional time-out value can be specified on the acquire operation to limit the time spent waiting if desired. This class mplements 'local' semaphores, which may only be used within a task (address space) and have no recovery semantics.

TRecoverableSemaphoreHandle is a class that represents a semaphore that behaves like a TLocalSemaphore with the additional property that the semaphore is "recoverable". Recoverability means that when a thread holding the semaphore terminates abnormally, the counts are adjusted, and any waiting threads are appropriately unblocked. An exception is raised in each such thread indicating that the semaphore was recovered and the integrity of any associated user data may be suspect. Note that for abnormal termination of a thread that had acquired the semaphore in a shared fashion, no exceptions need be raised in other threads since the associated data should only have been accessed in a read-only fashion and should still be in a consistent state. This class includes the following methods:

GetCurrentHolders: returns a collection of the current threads holding the semaphore.

SetRecovered: sets state of the semaphore to 'recovered', removing a previous 'damaged' state.

Destroy: removes the recoverable semaphore from the system

TMonitorEntry is a class that represents the lock (sometimes called a mutex) associated with a monitor. The constructor for this class actually causes the monitor lock to be acquired, and the act of exiting the local scope (which causes the destructor to be called) causes the monitor lock to be relinquished. If another task is already in the monitor, the thread attempting to enter the monitor will be blocked in the TMonitorEntry constructor until the preceding thread(s) leave the monitor. This class includes operators new and delete which are private so that TMonitorEntry's can only be allocated on the stack, thus providing automatic entry and exit (and the associated monitor lock acquire and release) with scope entry and exit.

TMonitorCondition is a class that represents a condition variable that is associated with some monitor. The major operations are wait, notify, and broadcast. The wait operation causes the current thread to wait for the condition to be notified, and while the thread is blocked the monitor lock is relinquished. Notify and broadcast are called by a thread executing inside the monitor to indicate that either one or all of the threads waiting on the condition should be unblocked when the notifying (or broadcasting) thread exits the monitor. When a waiting thread is unblocked, it attempts to reaquire the monitor lock (one thread at a time in the case of a broadcast), at which point it resumes executing in the monitor. An optional time-out value can be specified to limit the time spent waiting for a condition. Other than construction and destruction, all methods of TMonitorCondition must be called only from within the monitor.

TMonitorLock is a class that represents a lock on a monitor. It is passed into the constructors for TMonitorEntry and TMonitorCondition to indicate which monitor is being aquired or to which monitor a condition is to be associated.

Scheduling Classes

FIG. **11** is a class diagram **1102** of the scheduling classes **414**, which are used to invoke the scheduling services of Mach.

TThreadSchedule is a concrete base class that embodies the scheduling behavior of a thread. It defines the thread's actual, default, and maximum priorities. The lower the priority value, the greater the urgency. Each processor set has a collection of enabled TTHreadSchedules and a default one. A thread may be assigned any TThreadSchedule that is enabled on the processor set on which the thread is running. The priority may be set up to the maximum value defined by TThreadSchedule, but use of this feature is strongly discouraged. Specific scheduling classes (TIdleSchedule, TServerSchedule etc.) are made available using this class as the base. However (since there are no pure virtual functions in this class) derived classes are free to create objects of this class if necessary (but it may not be required to do so). TThreadSchedule objects (using polymorphism) are used to specify scheduling policy for threads. The subclasses presented below should be used to determine the appropriate priority and proper range.

TIdleThreadSchedule is a concrete subclass of TThreadSchedule for those threads that are to run when the system is idle. They only run when nothing else in the system can run. This category, in general, would be used for idle timing, maintenance, or diagnostic threads.

TServerSchedule is a concrete subclass of TThreadSchedule for server threads. Server threads must be very responsive. They are expected to execute for a short time and then block. For services that take an appreciable amount of time, helper tasks with a different kind of TThreadSchedule (TSupportSchedule) should be used.

TUserInterfaceSchedule is a concrete subclass of TThreadSchedule for those application tasks that should be responsive and handle the application's human interface. They typically run for a short time and then block until the next interaction.

TApplicationSchedule is a class used with those threads that support an application's longer running parts. Such threads run for appreciable amounts of time. When an application or window is activated, the threads in the associated task become more urgent so that the threads become more responsive.

TPseudoRealTimeThreadSchedule is a class that allows tasks to specify their relative urgency in the fixed priority class by setting their level within its range. The task schedule exports the number of levels that are allowable and the default base level. If a level is requested that would cause the value to be outside the class range an exception will be thrown. This class includes the following methods:

SetLevel (PriorityLevels theLevel): Set the level of the task. A lower number is more urgent.

RetumNumberOfLevels ( ): Return the number of levels of urgency fgr this scheduling object.

ReturnDefaultLevel ( ): Return the default level of urgency for this scheduling object. The default level is relative to the scheduling class's most urgent priority.

Fault Classes

FIGS. **12**, **13**, **14**, and **15** present class diagrams **1202**, **1220**, **1302**, **1402**, and **1502** of the fault classes **416**, which are used to invoke the fault services of Mach. For the classes that represent fault messages (for example, TIPCIdentityFaultMessage, TIPCIdentityFaultMessage, etc.), it is necessary to dedicate a single port for each message type. That is, the user should ensure that only one type of message will be received on any given port that is used for fault handling. Preferbly, the fault classes **416** include a processor-specific set of classes for each processor **106** that the operating system **114** runs on. Alternatively, the fault classes **414** may include generally generic classes which apply to multiple processors. The Motorola-68000-specific classes are presented herein for illustrative purposes, and is not limiting. Persons skilled in the relevant art will find it apparent to generate processor-specific classes for other processors based on the teachings contained herein.

TFaultType is an abstract base class that represents a fault. It is subclassed to provide the processor-unique fault values. It identifies the fault by processor and fault id. The following three classes are subclasses of TFaultType:

TMC680X0FaultType represents a fault type on a Motorola 68K processor. It identifies the possible 68K type values and CPU descriptor.

TMC680X0BadAccessFaultType represents a bad access type on a Motorola 68K processor.

TMC680X0AddressFaultType represents an address error type on a Motorola 68K processor.

TFaultDesignation is a class that encapsulates the destination, the format for a fault message, and the types of faults for which the message should be sent for a task or thread. This class allows you to specify on a task or thread basis that the fault message of the requested type for the specified fault types should be sent to the port indicated by the send right.

TFaultTypeSet encapsulates a set of fault types.

TFaultData is a class that encapsulates fault data provided by the kernel in addition to the processor state. Not all faults have fault data. The fault data is provided in the fault message and is available from the thread state.

TIPCFaultMessage is a class that encapsulates the fault message sent by the kernel on behalf of the thread that got the Fault. It is used to receive and reply to the Fault. Three subclasses (below) are provided for the three possible kinds of data that might be sent with the fault message. The message may include the identification of the faulting task and thread, or the state of the faulting thread, or both sets of information. TIPCIdentityFaultMessage encapsulates the Fault message containing the identity of the thread that got the Fault. It is used to receive and reply to the Fault. TTPCStateFaultMessage encapsulates the Fault message containing the thread state of the thread that got the Fault. It is used to receive and reply to the Fault. TIPCStateAndIdentityFaultMessage encapsulates the Fault message containing the thread state and identity of the thread that got the Fault. It is used to receive and reply to the Fault.

TThreadState is an abstract class that represents the CPU state of a thread. Subclasses actually define the processor specific forms. There is no information in the class. All work is done in the derived classes. All queries for CPU state will return a TMC680X0State pointer which has to be cast at runtime to the correct derived class object. Derived subclasses are specific to particular processors, such as many of the subclasses shown in FIGS. **12**, **13**, **14**, and **15** which are dependent on the Motorola 68xxx line of processors. Such subclasses include TMC680X0State, which is a concrete class that represents the 680x0 CPU state of a thread. Other examples include TMC680X0CPUState, which encapsulates the CPU state available for all 68K states, and TMC680X0CPUFaultState, which encapsulates the 68K fault state available for all 68K states.

Host and Processor Set Classes

FIG. **16** is a class diagram **1602** for the machine classes **418**, which are also called herein the host and processor set

classes. The machine classes **418** are used to invoke the services related to Mach's machine and multiprocessor support.

TPrivilegedHostHandle is a concrete class that embodies the privileged port to the kernel's host object. The privileged host port is the root of Mach's processor management. The holder of the privileged host port can get access to any port on the system. The basic privilege mechanism provided by the kernel is restriction of privileged operations to tasks holding control ports. Therefore, the integrity of the system depends on the close holding of this privileged host port. Objects of this class can: get boot information and host statistics, reboot the system, enumerate the privileged processor sets, communicate with non-CE entities, and enumerate the processors.

THostHandle is a non-privileged concrete class that embcdies the name port to the kernel's host object. Objects of this class can return some host information, and return the default processor set. Objects of this class are useful to get information from the host (such as kernel version, maximum number of CPUs, memory size, CPU type, etc.) but cannot cause any damage to the host. Users should be provided access to objects of this class rather than the highly privileged TPrivilegedHostHandle objects.

TProcessorHandle is a concrete class representing a processor. A processor can be started, exited, added to a TPrivilegedProcessorSetHandle, return information, and be sent implementation-dependent controls.

TPrivilegedProcessorSetHandle is a concrete class providing the protocol for a processor set control port. Objects of this class can: enable and disable scheduling policies, set the maximum priority for the processor set, return statistics and information, enumerate the tasks and threads, and assign thread, and tasks to the processor set. Client access to objects of this class should be highly restricted to protect the individual processors and the processor set.

TProcessorSetHandle is a concrete class providing the protocol for a processor set name port. Objects of this class can return basic information about the processor set (the number of processors in the processor set, etc.) but they cannot cause any damage to the processor set.

Implementation of Wrapper Methods

As noted above, the Mach and the Mach procedural interface are well-known. The wrapper class library **402**, and the operation of the methods of the wrapper class library **402**, have been defined and described in detail above. Implementation of the methods defined by the wrapper class library **402** is described below by considering selected methods from the wrapper class library **402**. Persons skilled in the relevant art will find it apparent to implement the other methods of the wrapper class library **402** based on the well-known specification of the Mach, the discussion above regarding the wrapper class library **402**, and the discussion below regarding the implementation of the wrapper methods. The implementation of the kill( ) method from the TThreadHandle class of the thread classes **404** is shown in Code Example 2, below. A routine called "example1" is shown in Code Example 1, below. The "example1" routine includes a decomposition statement which causes the kill( ) method to be executed. © Copyright, Taligent Inc., 1993 void examplel(TThreadHandle& aThread)

```
void example1(TThreadHandle& aThread)
{
    TRY
    {
        aThread.Kill( );      / / terminates aThread immediatly
    }
    CATCH(TKernelException)
    (
        printf("Couldn't kill thread "); / / error occured trying to kill
    }
    ENDTRY;
    / /...
}
CODE EXAMPLE 1
void TTreadHandle::Kill( )
{
    kern__return__error;
    if((error = thread__terminate(fThreadControlPort)) !=
    KERN__SUCCESS)
        THROW(TKernelException( ));   / / Error indicator
}
CODE EXAMPLE 2
```

Where:

fThreadControlPort is an instance variable of the TThreadHandle class that contains the Mach thread control port for the thread the class represents.

TKernelException is the C++ exception class that is thrown when a kernel routine gets an error.

THROW, TRY, CATCH, and ENDTRY are part of the C++ language that allow you to throw and catch C++ exceptions.

The implementation of the suspend( ) method from the TTaskHandle class of the task classes **406** is shown in Code Example 4, below. A routine called "example2" is shown in Code Example 3, below. The "example2" routine includes a decomposition statement which causes the suspends method to be executed.

```
void example2(TTaskHandle& aTask)
{
    TRY
    {
        aTask.Suspend( );      / / suspend all threads on task a Task
    }
    CATCH(TKernelException)
    (
        printf("Couldn't suspend threads "); / / error occured
    }
    ENDTRY;
    / /...
}
CODE EXAMPLE 3
void TTaskHandle::Suspend( )
{
    kern__return__t error;
    if((error = task__suspend(fTaskControlPort)) !=KERN__SUCCESS)
        THROW(TKernelException( ));   / / Error indicator
}
CODE EXAMPLE 4
```

Where:

fTaskControlPort is an instance variable of the TTaskHandle class that contains the Mach thread control port for the task the class represents.

TKernelException is the C++ exception class that is thrown when a kernel routine gets an error.

THROW, TRY, CATCH, and ENDTRY are part of the C++ language that allow you to throw and catch C++ exceptions.

The implementation of the GetLevel( ) method from the TPseudoRealTimeThreadSchedule class of the scheduling classes **414** is shown in Code Example 6, below. A routine called "example3" is shown in Code Example 5, below. The "example3" routine includes a decomposition statement which causes the GetLevel( ) method to be executed.

```
void example3(TPseudoRealTimeThreadSchedule& aSchedule)
{
    PriorityLevels curPriority;
    curPriority = aSchedule.GetLevel ( );      / / Get thread's
    current priority
    / /...
}
CODE EXAMPLE 5
PriorityLevels TPseudoRealTimeThreadSchedule::GetLevel( )
{
    struct task__thread__sched__infor schedInfo;
    thread__sched__info schedInfoPtr = schedInfo;
    mach__msg__type__number__t returnedSize;
    returnedSize = sizeof (schedInfo);
    void thread__info (fThreadControlPort, THREAD__SCHED__
    INFO,schedInfoPtr, &returnedSize);
    return (schedInfo.cur__priority);
}
CODE EXAMPLE 6
```

Where:

fThreadControlPort is an instance variable of the TPseudoRealTimeThreadSchedule class. It contains the Mach thread control port of the thread for which the class is a schedule.

The implementation of the GetKernelVersion( ) method from the THostHandle class of the machine classes **418** is shown in Code Example 8, below. A routine called "example4" is shown in Code Example 7, below. The "example4" routine includes a decomposition statement which causes the GetKernelVersion( ) method to be executed.

```
void example4(THostHandle& aHost)
{
    kernel__version__t version;
    aHost.GetKernelVersion (&version);      / / get version
    of kernel currently
running
    / /...
}
CODE EXAMPLE 7
void THostHandle::GetKernelVersion (kernel__version__t& the Version)
{
    void host__kernel__version(fHostPort, the Version);
}
CODE EXAMPLE 8
```

Where:

fHostPortis an instance variable of the THostHandleclass that contains the Mach host control port for the host the class represents.

The implementation of the GetMakeSendCount( ) method from the TPortReceiveRightHandle class of the IPC classes **410** is shown in Code Example 10, below. A routine called "example5" is shown in Code Example 9. below. The "example5" routine includes a decomposition statement which causes the GetMakeSendCount( ) method to be executed. As evident by its name, the GetMakeSendCount( ) method accesses the Mach to retrieve a make send count associated with a port. The GetMakeSendCount( ) method includes a statement to call mach__port__get__attributes, which is a Mach procedurally-oriented system call that returns status information about a port. In GetMakeSendCount( ), fTheTask is an instance variable of the TPortReceiveRightHandle object that contains the task control port of the associated task, and fThePortName is an instance variable of the TPortReceiveRightHandle object that contains the port right name of the port represented by the TPortReceiveRightHandle object.

```
void example5(TPortReceiveRightHandle& aReceiveRight)
{
    unsignd long count;
    count = aReceiveRight.GetMakeSendCount( );
    / /...
}
CODE EXAMPLE 9
unsigned long TPortReceiveRightHandle::GetMakeSendCount( )
{
    mach_port_status_t theInfo;              / / port status infor
                                             returned by Mach
    mach_msg_type_number_t theSize;          / / size of info
                                             returned by
    void mach_port_get_attributes(fTheTask, fThePortName,
                              MACH_PORT_RECEIVE_STATUS,
&theInfo, &theSize);
    return(theInfo.mps_mscount);
}CODE EXAMPLE 10
```

Variations on the present invention will be obvious to persons skilled in the relevant art based on the discussion contained herein. For example, the scope of the present invention includes a system and method of enabling a procedural application to access in a procedural manner an object-oriented operating system having a native object oriented interface during run-time execution of the application in a computer. This embodiment of the present invention preferably operates by locating in the application a procedural statement which accesses a service provided by the operating system, and translating the procedural statement to an object-oriented function call (i.e., method) compatible with the native object-oriented interface of the operating system and corresponding to the procedural statement. The object-oriented function call is executed in the computer to thereby cause the operating system to provide the service on behalf of the application. While various embodiments of the present invention have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

What is claimed is:

1. A computer system, comprising:

computer hardware for performing native system services;

a procedural operating system, having a native interface, for controlling the computer hardware to perform the native system services;

object oriented methods requiring native system services;

procedural program logic code, responsive to invocations of the object-oriented methods during runtime, for causing the procedural operating system to control the computer hardware to perform the required native system services;

executable program memory associated with the computer hardware for runtime execution of the procedural operating system, invocations of the object-oriented methods and related portions of the procedural program logic code;

means for making determinations during runtime execution if object-oriented methods to be invoked are present in the executable program memory; and

a runtime loader, responsive to the determinations, to selectively load required object-oriented methods into the executable program memory during runtime before invocation of the object-oriented methods.

2. The computer system of claim **1**, wherein the procedural program logic code further comprises:

procedural program logic code portions specific to each object-oriented method to issue one or more procedural function calls compatible with the native interface to control the native system services performed by the hardware environment to correspond to the native system services required by the object-oriented method.

3. The computer system of claim **1**, wherein the runtime loader further comprises:

means for selectively loading related portions of the procedural program logic code into the executable program memory upon runtime loading of the selected object-oriented methods.

4. The computer system of claim **3**, wherein the procedural operating system further comprises:

an operating system based on Windows or Unix.

5. The computer system of claim **3** wherein the computer hardware further comprises:

a Unix or Apple or IBM compatible computer environment.

6. The computer system of claim **3** wherein the procedural program logic code further comprises:

means for causing the procedural operating system to provide one or more of the following native system services:

thread services, task services, virtual memory services, inter-process communication (IPC) services, synchronization services, scheduling services, fault services, processor and processor set services, port services, security services, file system services and graphical user interface (GUI) services.

7. A method for operating a computer system, comprising the steps of:

executing a procedural operating system on computer hardware, the procedural operating system including a native interface, responsive to procedural function calls, for providing native system services;

issuing calls during runtime, compatible with the native interface, to provide the native system services in response to invocations of object-oriented methods requiring such native system services;

determining during runtime if object-oriented methods to be invoked during runtime execution are present in executable program memory associated with the computer hardware; and

selectively loading the object-oriented methods into the executable program memory during runtime before invocation thereof, if not yet loaded.

8. The method of claim **7**, wherein the step of selectively loading the object-oriented methods further comprises the step of:

loading related portions of a procedural program logic code for issuing the calls, compatible with the native interface, to provide the native system services in response to invocations of the selectively loaded object-oriented methods.

9. The method of claim **8**, wherein the step of executing a procedural operating system on computer hardware further comprises the step of:

executing a procedural operating system, based on Windows or Unix, in a Unix or IBM compatible computer environment.

10. The method of claim **7**, wherein the step of issuing calls, compatible with the native interface, to provide the native system services in response to invocations of object-oriented methods requiring such native system services, further comprises the step of:

adapting the native services provided by the procedural operating system to be compatible with the native system services required by the associated object-oriented method.

11. The method of claim **10** wherein the step of issuing calls compatible with the native interface further comprises the step of:

providing one or more of the following native system services:

thread services, task services, virtual memory services, inter-process communication (IPC) services, synchronization services, scheduling services, fault services, processor and processor set services, port services, security services, file system services and graphical user interface (GUI) services.

12. A method for operating a computer system, comprising the steps of:

executing a procedural operating system, based on Windows or Unix operating systems, on a Unix or IBM compatible computer hardware environment;

providing an object-oriented interface, executing on the computer hardware environment, and responsive to object-oriented programming, for instantiating objects from object-oriented classes, encapsulating data for exclusive use with each object, and invoking object-oriented methods in the objects for operating on the encapsulated data;

providing procedural programming logic code, responsive during runtime to selected ones of said invoked object-oriented methods requiring native system services, for issuing procedural calls, compatible with a native interface of the procedural operating system, to cause the hardware environment to provide the native system services in response to the object-oriented methods; and

loading the methods during runtime before invocation thereof;

whereby a choice of which system implementation to use can be deferred to run-time.

13. The method of claim **12** wherein the step of providing procedural programming logic code for issuing procedural calls further comprises the step of:

issuing procedural function calls to the operating system in response to invocations of selected object-oriented methods for causing the procedural operating system to control the computer hardware environment to provide one or more of the following native system services:

thread services, task services, virtual memory services, inter-process communication (IPC) services, synchronization services, scheduling services, fault services, processor and processor set services, port services, security services, file system services and graphical user interface (GUI) services.

14. The method of claim **13**, wherein the step of issuing procedural function calls to the operating system in response to invocations of selected object-oriented methods further comprises the step of:

providing procedural logic code, responsive to the invocation of each object-oriented method requiring the performance of native system services, to execute predetermined procedural code to control the native system services performed by the hardware environment to correspond to the native system services required by the object-oriented method.

15. The method of claim **13**, wherein the step of issuing procedural function calls to the operating system in response to invocations of selected object-oriented methods further comprises the step of:

providing procedural logic code, responsive to the invocation of each object-oriented method requiring the performance of native system services, to issue, monitor and adapt one or more procedural function calls to control the native system services performed to correspond to the native system services required by the object-oriented method.

16. A method for operating a computer system including a memory, comprising the steps of:

storing in the memory a library of procedural program logic code;

said library including first procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform first type native system services;

said library including second procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform second type native system services different from said first type;

executing a procedural operating system in the memory, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

running an object-oriented program in a task address space of the memory, the program including an object-oriented method requiring the second type native system services;

determining during runtime whether said second type procedural program logic code is available in said task address space; and

loading said second type procedural program logic code into said task address space during runtime.

17. A method for operating a computer system including an executable program memory, comprising the steps of:

storing in the computer system a library of procedural program logic code;

said library including first procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform first type native system services;

said library including second procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform second type native system services different from said first type;

executing a procedural operating system in the executable program memory, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

running an object-oriented program in the executable program memory, the program including an object-oriented method requiring the second type native system services;

determining during runtime whether said second type procedural program logic code is available in the executable program memory; and

loading said second type procedural program logic code into the executable program memory during runtime.

18. A method for operating a computer system including an executable program memory, comprising the steps of:

storing in the computer system a library of procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform native system services;

executing a procedural operating system in the executable program memory, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

running an object-oriented program in the executable program memory, the program including an object-oriented method requiring native system services;

determining during runtime whether procedural program logic code is available in the executable program memory to provide said required native system services; and

loading procedural program logic code from said library into the executable program memory during runtime to provide said required native system services.

19. A method for operating a computer system including an executable program memory, comprising the steps of:

storing in the computer system a library of procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform native system services;

executing a procedural operating system in the computer system, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

running an object-oriented program in the executable program memory, the program including an object-oriented method requiring native system services;

determining during runtime whether procedural program logic code is available in the executable program memory to provide said required native system services; and

loading procedural program logic code from said library into the executable program memory during runtime to provide said required native system services.

20. A computer system including an executable program memory, comprising:

a library of procedural program logic code in the computer system which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform native system services;

a procedural operating system in the computer system, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

an object-oriented program in the executable program memory, the program including an object-oriented method requiring native system services;

a processor in the computer system for determining during runtime whether procedural program logic code is available in the executable program memory to provide said required native system services; and

said processor loading procedural program logic code from said library into the executable program memory during runtime to provide said required native system services.

21. A method for operating a computer system including an executable program memory, comprising the steps of:

storing in the computer system a library of procedural program logic code which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform native system services;

executing a procedural operating system in the computer system, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

running an object-oriented program in the executable program memory, the program including an object-oriented method requiring native system services;

determining during runtime whether procedural program logic code is available in the executable program memory to provide said required native system services;

loading procedural program logic code from said library into the executable program memory during runtime to provide said required native system services;

invoking said object-oriented method of said object-oriented program during runtime; and

responding with said loaded procedural program logic code to said invoking step to cause said procedural operating system to control the computer system to perform said required native system services.

22. A computer system including an executable program memory, comprising:

a library of procedural program logic code in the computer system which is responsive to invocations of object-oriented methods, for causing a procedural operating system to control the computer system to perform native system services;

a procedural operating system in the computer system, the procedural operating system including a native interface responsive to procedural function calls, for providing native system services;

an object-oriented program in the executable program memory, the program including an object-oriented method requiring native system services;

a processor in the computer system for determining during runtime whether procedural program logic code is available in the executable program memory to provide said required native system services;

said processor loading procedural program logic code from said library into the executable program memory during runtime to provide said required native system services;

said processor invoking said object-oriented method of said object-oriented program during runtime; and

said loaded procedural program logic code responding to said invoking to cause said procedural operating system to control the computer system to perform said required native system services.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: 6,275,983 B1	Page 1 of 1
APPLICATION NO.	: 09/140523	
DATED	: August 14, 2001	
INVENTOR(S)	: Debra Lyn Orton et al.	

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

On the title page, item [60] under the heading "Related Application Data"
Delete "Continuation of application No. 08/521,085, filed on Aug, 29, 1995, now abandoned." and insert --Continuation of application No. 08/521,085, filed on Aug. 29, 1995, now Pat. No. 6,684,261, which is a continuation of application No. 08/315,212, filed on Sep. 28, 1994, now Pat. No. 5,475,845, which is a continuation of application No. 08/094,675, filed on July 19, 1993, now Pat. No. 5,379,432.-- therefor.

In column 21, line 60, after "scheduled" insert --.--

In column 31, line 28, after "system" insert --.--

In column 32, line 67, replace "Preferbly" with --Preferably--

In column 35, line 7, replace "immediatly" with --immediately--

In column 35, line 9, replace "(" with --{--

In column 35, line 10, replace "occured" with --occurred--

In column 35, line 46, replace "(" with --{--

In column 35, line 47, replace "occured" with --occurred--

Signed and Sealed this

Eighth Day of December, 2009

David J. Kappos

David J. Kappos
Director of the United States Patent and Trademark Office

EXHIBIT 9

US006343263B1

(12) **United States Patent**
Nichols et al.

(10) **Patent No.: US 6,343,263 B1**
(45) **Date of Patent: Jan. 29, 2002**

(54) REAL-TIME SIGNAL PROCESSING SYSTEM FOR SERIALLY TRANSMITTED DATA

(75) Inventors: **James B. Nichols**, San Mateo; **John Lynch**, San Jose, both of CA (US)

(73) Assignee: **Apple Computer, Inc.**, Cupertino, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/284,061**

(22) Filed: **Aug. 2, 1994**

(51) **Int. Cl.**^{7} **G06F 13/00**
(52) **U.S. Cl.** **702/189**; 709/328
(58) **Field of Search** 364/514, 238.5, 364/924, 576, 725, 726; 395/2.09, 2.1, 2.91, 2.94, 406, 651, 680, 682, 821, 892, 561, 566, 733; 455/84, 3.1, 39; 370/210; 340/825; 342/195, 196; 360/39

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,327,558	A	*	7/1994	Burke et al.	395/650
5,363,315	A	*	11/1994	Weiss et al.	364/514
5,381,346	A	*	1/1995	Monaham-Mitchell et al.	364/514
5,406,643	A	*	4/1995	Burke et al.	395/200
5,438,614	A	*	8/1995	Rozman et al.	379/93
5,440,619	A	*	8/1995	Cann	379/97
5,440,740	A	*	8/1995	Chen et al.	395/650
5,442,789	A	*	8/1995	Baker et al.	395/650
5,487,167	A	*	1/1996	Dinallo et al.	395/650

FOREIGN PATENT DOCUMENTS

EP	218859	4/1987

OTHER PUBLICATIONS

Tanenbaum, *Structured Computer Organization,* 1984, pp. 10–12.*

Frankel, "DSP Resource Manager Interface & Its Role in DSP Multimedia", May 1994.*

Silberschatz et al., *Operating System Concepts,* p. 489, 1994.*

Jon Udell, "Computer Telephony," *Byte,* vol. 19, No. 7, Jul. 1994, pp. 80–96.

* cited by examiner

Primary Examiner—Patrick Assouad
(74) *Attorney, Agent, or Firm*—Burns, Doane, Swecker & Mathis, L.L.P.

(57) ABSTRACT

A data transmission system having a real-time data engine for processing isochronous streams of data includes an interface device that provides a physical and logical connection of a computer to any one or more of a variety of different types of data networks. Data received at this device is presented to a serial driver, which disassembles different streams of data for presentation to appropriate data managers. A device handler associated with the interface device sets up data flow paths, and also presents data and commands from the data managers to a real-time data processing engine. Flexibility to handle any type of data, such as voice, facsimile, video and the like, that is transmitted over any type of communication network with any type of real-time engine is made possible by abstracting the functions of each of the elements of the system from one another. This abstraction is provided through suitable interfaces that isolate the transmission medium, the data manager and the real-time engine from one another.

41 Claims, 6 Drawing Sheets

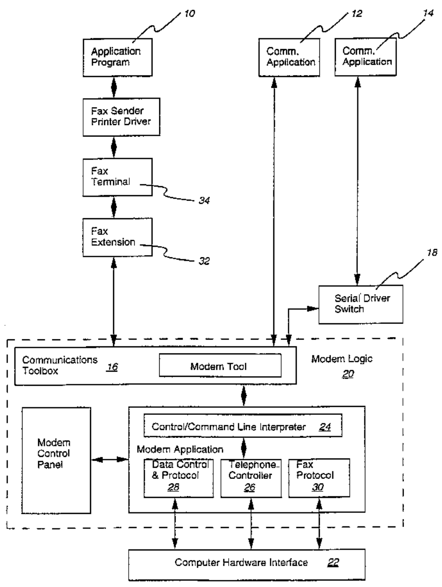


FIGURE 1

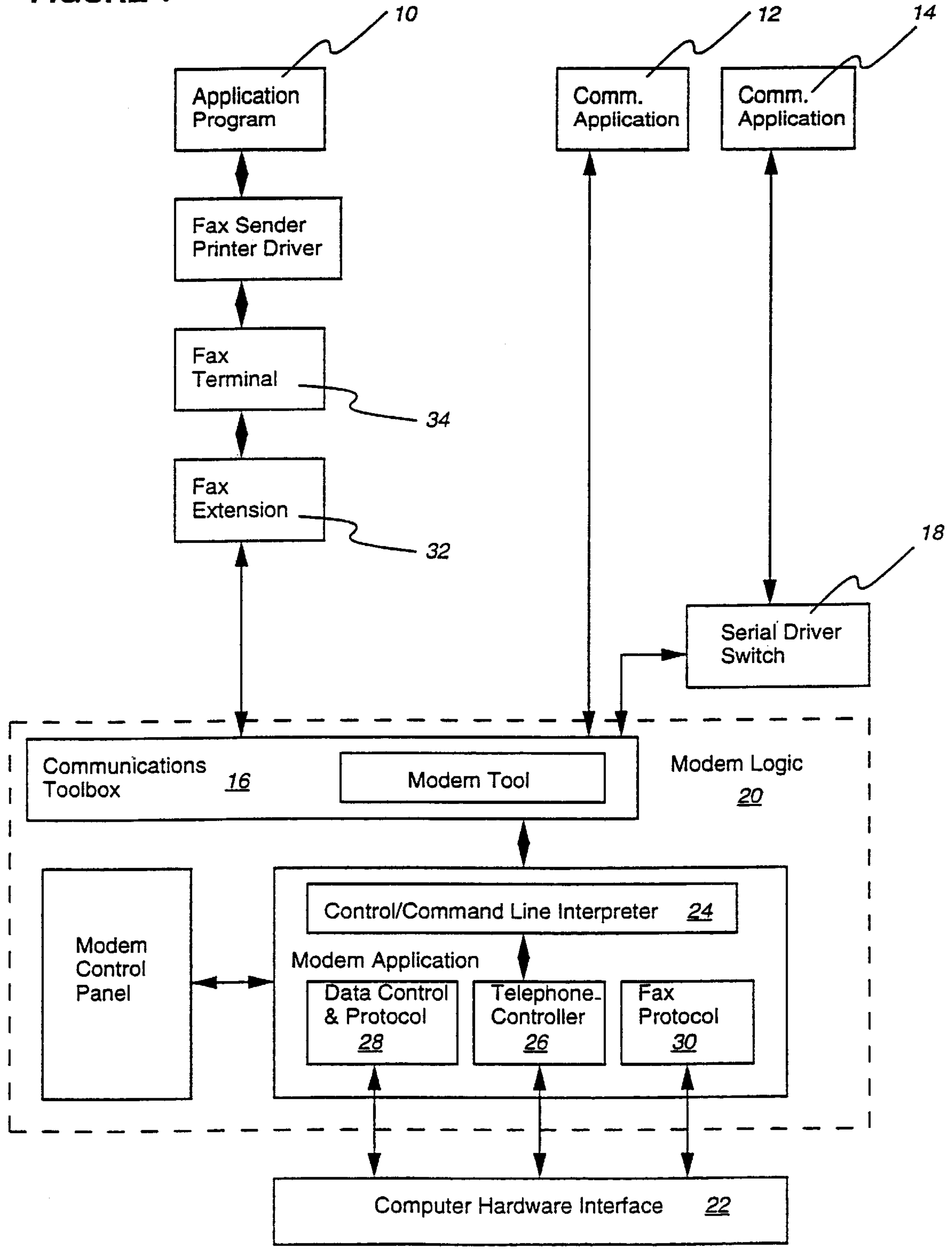

10
Application Program
Fax Sender Printer Driver
Fax Terminal
34
Fax Extension
32
12
Comm. Application
14
Comm. Application
18
Serial Driver Switch
Communications Toolbox 16
Modem Tool
Modem Logic 20
Modem Control Panel
Control/Command Line Interpreter 24
Modem Application
Data Control & Protocol 28
Telephone Controller 26
Fax Protocol 30
Computer Hardware Interface 22

FIGURE 2

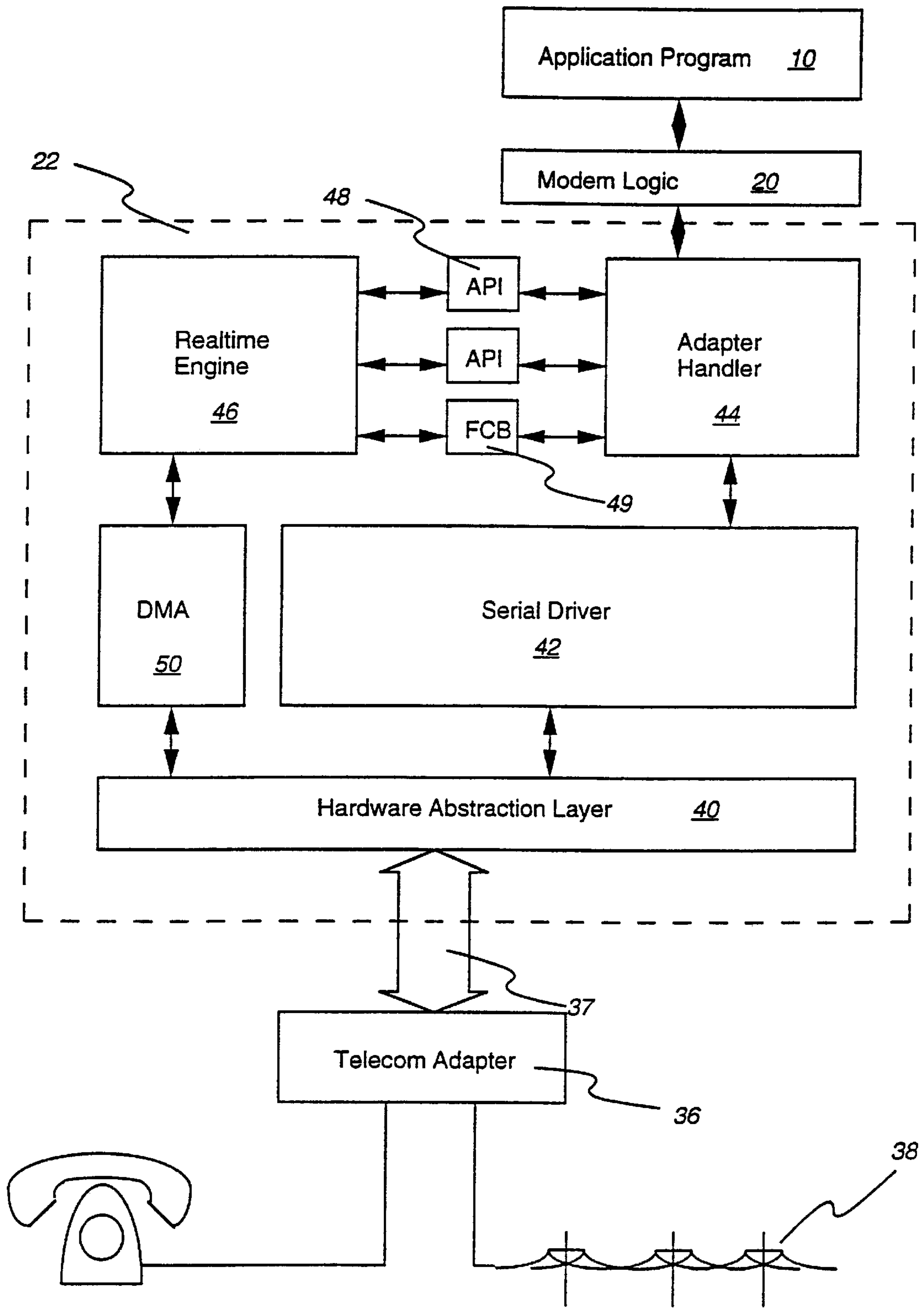
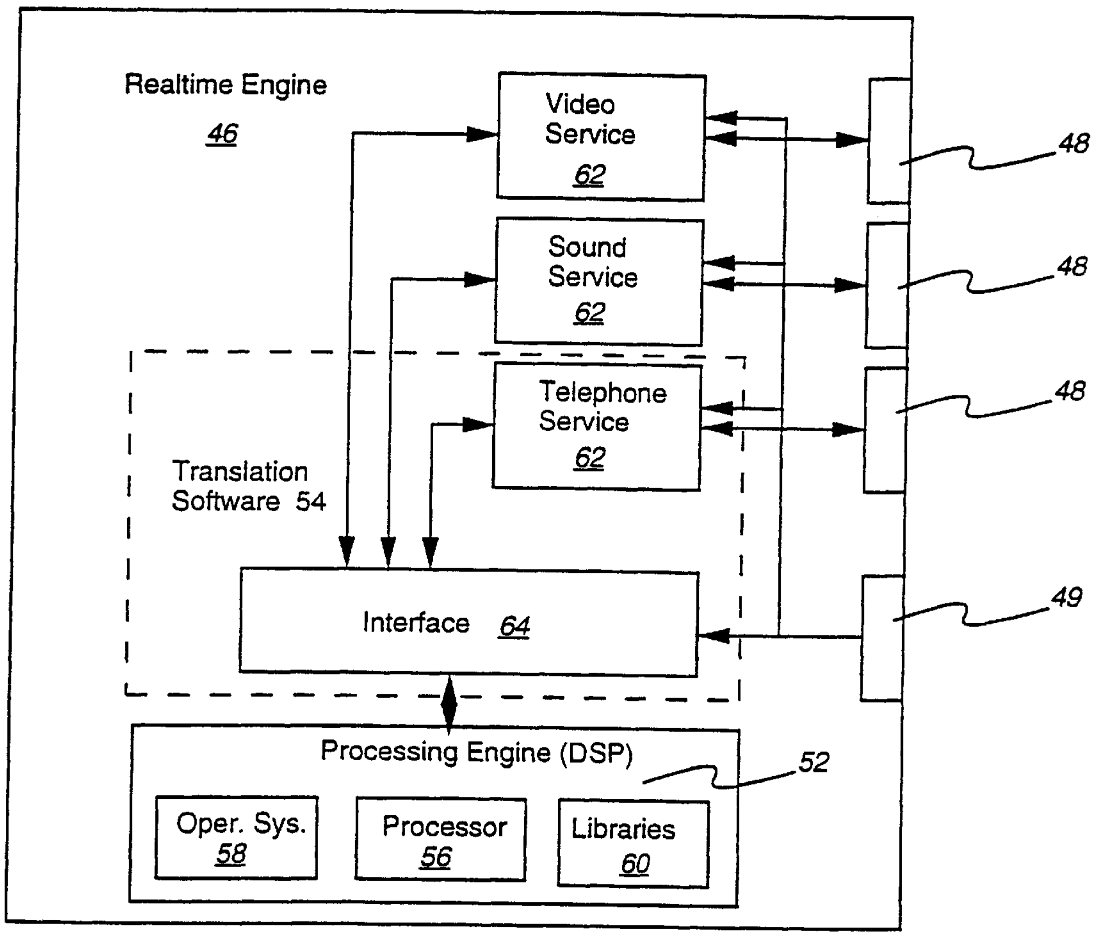

Application Program 10
Modem Logic 20
22
48
API
API
FCB
49
Realtime Engine 46
Adapter Handler 44
DMA 50
Serial Driver 42
Hardware Abstraction Layer 40
37
Telecom Adapter
36
38

FIGURE 3


Realtime Engine
46
Video Service
62
Sound Service
62
Telephone Service
62
Translation Software 54
Interface 64
Processing Engine (DSP)
52
Oper. Sys.
58
Processor
56
Libraries
60
48
48
48
49

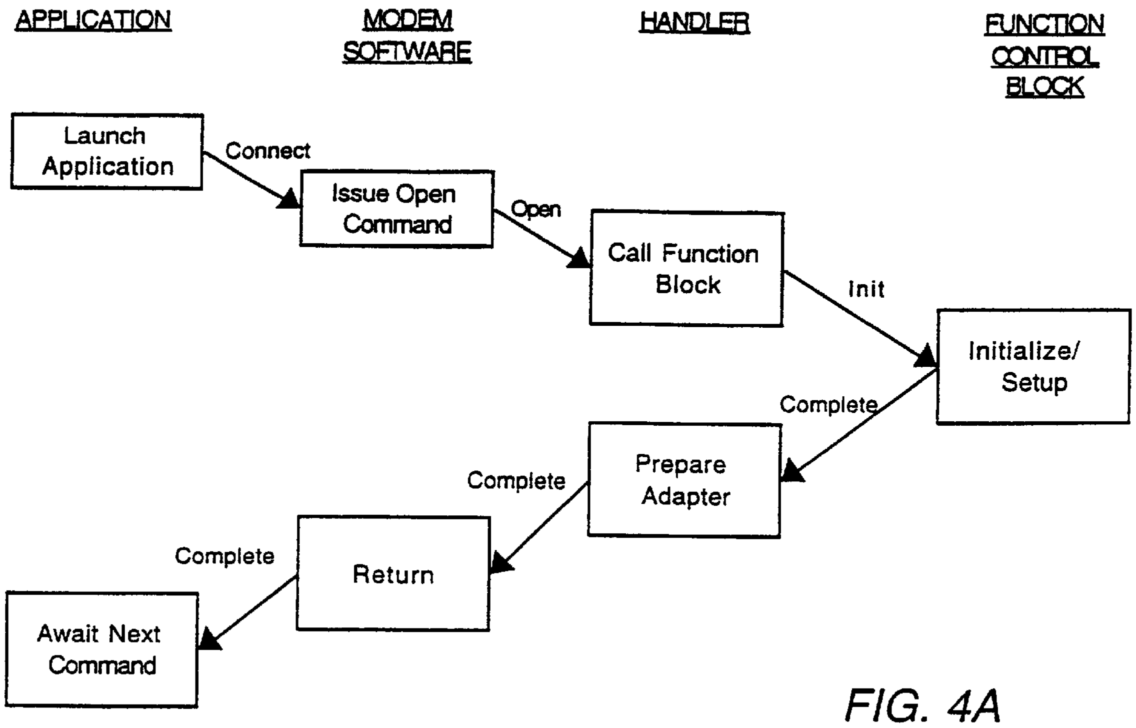

APPLICATION
MODEM SOFTWARE
HANDLER
FUNCTION CONTROL BLOCK
Launch Application
Connect
Issue Open Command
Open
Call Function Block
Init
Initialize/ Setup
Complete
Prepare Adapter
Complete
Return
Complete
Await Next Command

FIG. 4A

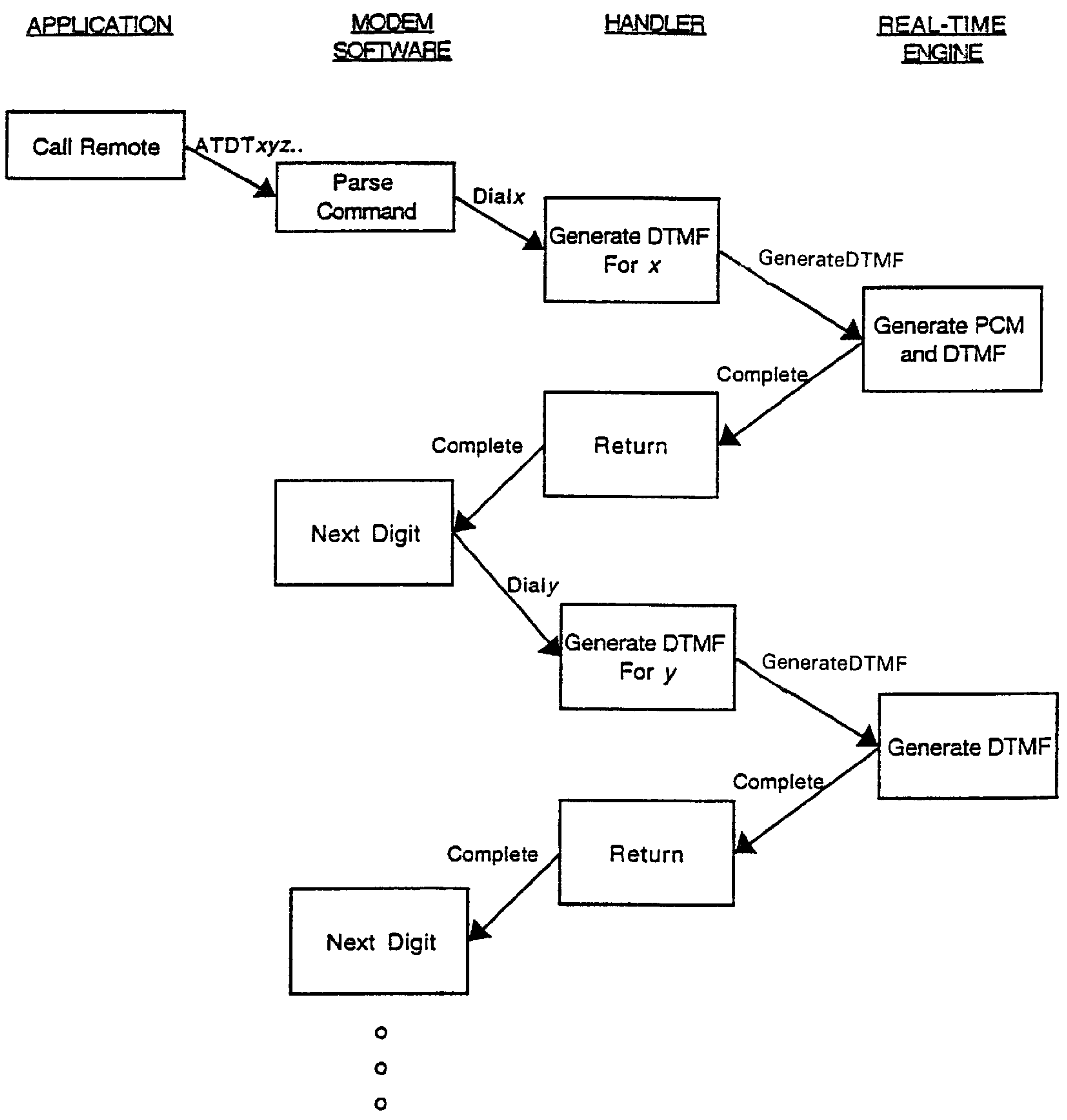

APPLICATION
MODEM SOFTWARE
HANDLER
REAL-TIME ENGINE
Call Remote
ATDT*xyz*..
Parse Command
Dial*x*
Generate DTMF For *x*
GenerateDTMF
Generate PCM and DTMF
Complete
Return
Complete
Next Digit
Dial*y*
Generate DTMF For *y*
GenerateDTMF
Generate DTMF
Complete
Return
Complete
Next Digit

FIG. 4B

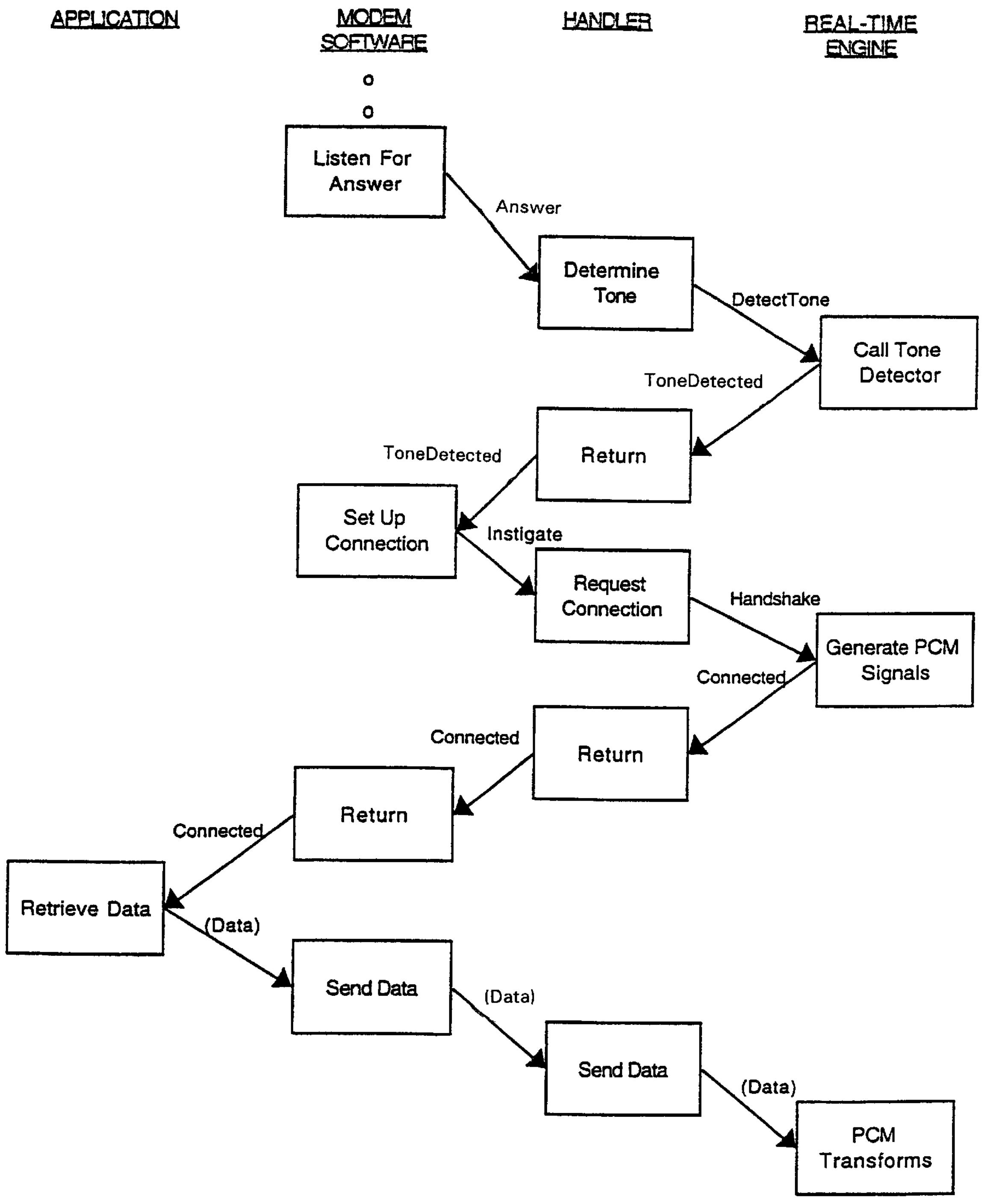

APPLICATION
MODEM SOFTWARE
HANDLER
REAL-TIME ENGINE
Listen For Answer
Answer
Determine Tone
DetectTone
Call Tone Detector
ToneDetected
Return
ToneDetected
Set Up Connection
Instigate
Request Connection
Handshake
Generate PCM Signals
Connected
Return
Connected
Return
Connected
Retrieve Data
(Data)
Send Data
(Data)
Send Data
(Data)
PCM Transforms

FIG. 4C

REAL-TIME SIGNAL PROCESSING SYSTEM FOR SERIALLY TRANSMITTED DATA

FIELD OF THE INVENTION

The present invention is directed to the transmission of data to and from a computer, and more particularly to a system for performing real-time signal processing of data that is serially transmitted to and from a computer.

BACKGROUND OF THE INVENTION

Various devices are known for transmitting data between a computer and a remote site via wide-area telecommunications networks. One of the most widely used devices of this type is the modem, which enables data to be transmitted to and from a computer over a wide-area analog telephone network. Generally speaking, the modem includes one or more sets of registers, typically embodied in an UART or an USART, for storing bits of digital data transmitted to or from the computer, a processor for implementing modem operations, such as dialing a telephone number or answering a ringing signal, in response to commands sent from the computer and stored in the UART, and a modulator/demodulator for converting digital bits of data to be transmitted into analog signals, and vice versa. Originally, all of these features were hardwired in a separate peripheral device that could be externally connected to the computer via a serial I/O port, or internally connected to the computer's data bus. More recently, some of the functions associated with these features, most notably the processing of commands to implement modem operations, have been removed from the hardwired configuration and incorporated into the computer itself. This approach has increased the versatility of the modem. For example, while the hardwired modem configuration had to be specifically designed for the telephone system requirements of a particular country, the later approach could enable a single product to be used in a variety of countries, each of which might have different telephone signaling requirements. Similarly, since the computer itself was handling the data to be transmitted, additional services, such as the ability to send information as a facsimile transmission, in which graphical data is processed, became feasible. However, the presence of the UART, or similar such device through which the data must flow, still limits the effective rate at which the data can be exchanged between the computer and the telephone lines.

To enhance the performance of modems, a digital signal processor (DSP) has been incorporated into its structure. In this arrangement, the modem software was designed to cooperate with the DSP to provide data thereto for processing prior to transmission or after reception over the telephone line. While the addition of the DSP provided increased capabilities in terms of the speed at which the data could be transmitted over a telephone network and the ease with which the modem could be configured, it was still limited in the types of data that could be processed. More particularly, because of the restrictions imposed by passing the data through an UART or the like, even the most modem modems are only capable of effectively transmitting data in the range of 9.6–14.4 Kb/sec. While this rate of data transfer may be useful for transmitting static information such as text files or the like, it is not suitable for many real-time applications in which the data is provided at much higher rates, such as speech or video conferencing.

Further in this regard, the modem control software had to be designed to work with the specific DSP incorporated into the computer. If a different DSP was to be used, the modem control software had to be reprogrammed to work with the new DSP.

While the analog telephone network was the only practical medium for transmitting information between geographically distributed computers for quite some time, more recently other, non-analog transmission mediums have become available. Examples of these other mediums include the integrated services digital network (ISDN), private branch exchange (PBX) telephone systems, and TI digital data links. Since information is transmitted over these mediums as digital data, conventional analog modem circuits are not suited for use with them. Thus, for example, a standard Group III facsimile machine cannot operate on a digital PBX system.

Similarly, digital signal processing systems which are designed to work with PCM-encoded analog data that is received and transmitted via a modem are likewise not suited for use with these other types of transmission mediums. While it is possible to incorporate another DSP system into a computer that can handle data transmitted via any of these digital networks, it would be more desirable to provide a single system that can process data that is received over any type of transmission medium, whether it be digital or analog. Further in this regard, it is desirable to provide a signal processing system that is not limited to one specific DSP, but rather one that can operate with any of a variety of different types of signal processors.

When personal computers communicate with one another through non-modem transport facilities, they typically operate in a burst mode. While this mode of operation enables data to be transferred at much higher rates than with modems, it is still not suitable for applications such as video or speech processing. These types of applications require isochronous data handling, i.e. data that is transmitted at a constant bit rate and that must be processed in real time. Generally speaking, therefore, it is desirable to provide a serial data transmitting and receiving system that is capable of processing real-time isochronous data.

Further in this regard, it is desirable to provide such a system that is capable of handling streams of data pertaining to different functions. For example, in a video conferencing application, speech data is transmitted at the same time as video and other graphic information. However, each of these types of data must be processed separately in real time. It is desirable, therefore, to provide a data transfer system that can handle each of two or more types of data at isochronous rates.

BRIEF STATEMENT OF THE INVENTION

The present invention provides a data transmission system having a real-time data engine for processing isochronous streams of data. An interface device provides a physical and logical connection of the computer to any one or more of a variety of different types of data networks, including analog telephone, ISDN, PBX and the like. Data received at this device is presented to a serial driver, which disassembles different streams of data for presentation to appropriate data managers, such as the operating system of the host computer, a service provider or an application program. A device handler associated with the interface device sets up data paths and issues service requests. The device handler also presents data and commands from the data managers to a real-time data processing engine, that can be used for a variety of applications such as voice recognition, speech compression, and fax/data modems. This real-time engine can be shared by any application program running on the host computer.

The invention enables any arbitrary type of data, such as voice, facsimile, multimedia and the like, which is transmitted over any type of communication network, to be handled with any type of real-time engine, by abstracting the functions of each of the elements of the system from one another. This abstraction is provided through suitable interfaces that isolate the transmission medium, the data managers and the real-time engine from one another. The data is provided to the real-time engine as multiple streams of isochronous data, i.e. it is delivered as it arrives over the network without data handling delays. This feature allows more complex applications, such as speakerphones, videophones and high-speed modems to be readily implemented.

These features of the invention, as well as the advantages offered thereby, are described in greater detail hereinafter with reference to a specific embodiment illustrated in the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of the software architecture for a facsimile/modem communications engine;

FIG. **2** is a block diagram of the computer hardware interface for the engine;

FIG. **3** is a more detailed block diagram of the architecture of the real-time engine; and

FIGS. **4A–4C** are flow diagrams of the steps that are carried out in the operation of a virtual fax/data modem in accordance with the invention.

DETAILED DESCRIPTION

The following description of an embodiment of the present invention is presented in the context of its implementation in a moderate speed (e.g. T1 data rates), low cost, digital interconnect to a wide-area network. An example of such an interconnect is described in U.S. patent application Ser. No. 08/180,926 filed Jan. 11, 1994, now U.S. Pat. No. 5,515,373 the disclosure of which is incorporated herein by reference. The practical applications of the invention are not limited to this particular embodiment, however. For example, it can also be employed as a high-performance interconnect to multimedia devices such as digital imaging equipment and audio appliances.

To facilitate an understanding of the invention, it will be described with reference to the specific example of a telephone-based telecommunication subsystem that provides basic fax/data modem services, plus call management and audio stream handling. This particular example is perhaps one of the more complex applications of the invention, because of the number of different signaling requirements associated with communications over the telephone lines. Further complexity is added by the fact that these requirements are often country-specific, and therefore the handling of a command, such as dialing a telephone number, can vary greatly from one country to the next. Other implementations of the invention, for example in the context of transmitting sounds and video data, will become apparent from an understanding of the principles of the invention explained with respect to this particular example.

FIG. **1** is a block diagram of a communications engine that is designed to provide modem and facsimile services. The top level of the diagram represents communications applications **10–14**. These applications send commands for communications services to a data service provider, or application programming interface (API), such as the Telephone Manager provided by Apple Computer, Inc. or the Telephone Application Programming Interface (TAPI) provided by Microsoft Corp. This interface is represented by the

communications toolbox **16** contained within fax/data modem logic modules **20**. A hardware interface **22** transmits the services provided by these modules over a transmission medium to which the computer is connected, e.g. a telephone line.

Referring to the modem logic **20**, a Modem Control and Command Line Interpreter (CLI) **24** accepts command inputs from the application programs to configure the modem, dial, receive calls, initiate data or fax transmission, hang up, and so on. Any operation of the modem must be initiated by issuing commands to the CLI. The CLI **24** can be an interface that is often referred to as the “AT command set”, which uses simple printable character sequences and constitutes a de facto standard among modems.

A Telephone Controller module **26** dials, answers, and hangs up the telephone line. The Telephone Controller can be “worldwide” in nature. In such a case it configures the modem to conform to the standards of telephone systems in most major economic markets. This can be carried out by storing a country code identifier that allows the Telephone Controller to configure the modem properly.

A Data Control and Protocol module **28** provides all data capabilities for the modem. It supports standard asynchronous text read and write, as well as the standard CCITT V.42 and V.42 bis and Microcom MNP class 2 through 5 reliable link and data compression protocols, for example.

The Fax Protocol module **30** performs the functions of a T.30 fax engine in a dedicated fax modem. It communicates with a Fax Extension driver **32**, a component of a Fax Terminal **34**, for example using an extension of the AT command set known as +F (“plus F”), or TR.**29**.

The hardware interface **22** contains the appropriate transport-dependent protocol components. The logical and physical interface to the wide-area network is hidden in this layer. This allows the fax/data modem modules to be used on any wide-area connection, including PBX, T1 and ISDN, as well as traditional POTS (“plain old telephone system”) channels. The components of this interface are illustrated in the block diagram of FIG. **2**.

Referring now to FIG. **2**, the hardware interface includes an external adapter **36** that provides the physical and logical connection of the computer to the telephone line **38** or other communications network, such as ISDN, PBX or T1 link. This connection can be provided through a serial port **37** of the computer. An example of such an adapter is described in copending U.S. patent application Ser. No. 08/078,890 filed May 10, 1993, and Ser. No. 08/180,925 filed Jan. 11, 1994, the disclosures of which are incorporated herein by reference. Such an adapter preferably includes processing capabilities, as disclosed in concurrently filed U.S. patent application Ser. No. 08/288,144 now U.S. Pat. No 5,799,190 entitled Intelligent Communications Coprocessor, which enables it to provide a constant stream of data to the computer from one or more communications networks. This data can be delivered in a time-division multiplexed manner, or it can be delivered in a packetized form.

Data received at the adapter **36** from the telephone line **38** can be provided to a driver **40** which functions as a hardware abstraction layer. This driver is a hardware-dependent implementation of a serial controller, and is configured in accordance with the particular characteristics of the communications port **37** to which the adapter **36** is connected. It isolates the remainder of the software from the different implementations for connecting the adapter to the computer, e.g. serial port, parallel port or bus device.

A serial driver **42** operates to separate multiple incoming data streams from one another, or to combine multiple outgoing data streams that are intended for respective transmission to separate destinations, whether logical or physical. For example, the adapter **36** might be connected to a desktop telephone for voice communications and to the telephone line for wide-area communications. Data streams for these two connections can be combined in a multiplexed manner by the serial driver, to be sent to the adapter **36** through the computer's serial port **37**.

An adapter handler **44** is specific to the particular adapter **36** and carries out features associated with that adapter. For example, if the adapter is one that is designed to be connected to a telephone network, the handler implements functions germane to that network such as ring detection, pulse dialing, on/off hook control and the like, in response to commands received from the application programs **10**. One example of the manner in which data can be exchanged between the adapter **36** and the handler **44** is described in U.S. patent application Ser. No. 08/058,750 filed May 7, 1993, now abandoned the disclosure of which is incorporated herein by reference.

A real-time engine **46** can perform transforms on data streams provided to and received from the adapter **36**. The particular transforms to be performed are sent as commands to the real-time engine from the adapter handler **44** via suitable application programming interfaces **48**. For communicating with the real-time engine, each interface includes shared command/control mailboxes in the computer's RAM, as well as bi-directional first-in, first-out (FIFO) buffers for transferring data. As an example, if the system is set up to operate as a fax/data modem, the real-time engine functions as a virtual telephone. In such a case, the handler may instruct the engine **46** to send a facsimile in response to a command from an application program. For this purpose, the real-time engine is configured with a suite of modem and call processing functions. This configuration is implemented by a real-time function control block **49**, which initializes and manages the operation of the real-time engine. Generally speaking, whenever a new task is to be carried out by the real-time engine **46**, the real-time function control block **49** issues commands that are specific to the operating system of the real-time engine. These commands cause the engine to start up, if it is not already running, and to configure itself with a library of routines that are necessary for it to implement the task.

When the handler **44** requests a facsimile transmission, for example, the real-time function block issues commands to start the real-time engine and install the various modules that are needed for it to function as a virtual telephone. Binary facsimile image data is transferred to the real-time engine via the FIFO buffers, where it is encoded as PCM data which is further encoded according to the transport medium over which it is to be transmitted. If the adapter is connected to a telephone line, for example, these signals can be encoded as 16-bit pulse-code modulated (PCM) samples, and forwarded directly to the adapter **36** via the serial driver **42**. Alternatively, if the transport medium is an ISDN line, the modem signals are encoded as mulaw-companded 8-bit PCM signals. The different types of encoding are stored in different tables, and the appropriate one to be used by the real-time engine is installed by the real-time function block during the initial configuration of the engine and/or designated by the API **48** at the time the command to transform the data is issued.

Transformed signals from the real-time engine that are to be transmitted via the transmission medium are provided to the hardware abstraction layer **40** through direct memory access (DMA) **50**. When data is being received from the communications network, it is provided to the real-time engine as pulse-code modulated (PCM) data through the DMA **50**. If the computer does not have DMA capabilities, the data can be transferred between the adapter and the handler as packetized data, as described in application Ser. No. 08/058,750 filed May 7, 1993 now abandoned.

As another example, the real-time engine may operate as a virtual speech recognition device. To do so, the real-time function block initializes the engine and installs the modules necessary to carry out this function. In this mode of operation, the engine may convert pulse-code modulated (PCM) data received from the adapter into vector-quantized speech components. The engine transforms this PCM data into data appropriate for subsequent processing by a speech recognition application program, according to the commands from the handler **44**, and the results of the transforms are provided through the shared FIFOs.

With this configuration, the handler does not need to have any knowledge of the real-time engine implementation. Communications with the real-time engine are carried out in a robust fashion. In essence, the architecture of the system provides a message-passing capability between devices that know nothing about the configuration or existence of one another.

As illustrated in FIG. **2**, there can be a number of interfaces **48** situated between the handler **44** and the real-time engine **46**. Each interface represents services for a particular class of functionality. For example, one interface may relate to the operation of the engine as a virtual telephone, another interface can be associated with a virtual sound device, e.g. stereo, and a third interface can pertain to a virtual video device. Each interface receives commands from an application program, through the handler **44**, and instructs the real-time engine to carry out the necessary transforms which relate to the function of the virtual device being implemented, e.g. text-to-speech conversion, video image processing, etc.

The architecture of the real-time engine **46** is illustrated in further detail in FIG. **3**. Referring thereto, when configured as a virtual device, the real-time engine is made up of two main components, a processing engine **52**, such as a DSP, and translation software **54**. The DSP comprises a processor **56**, an operating system **58** for that processor, and a set of libraries **60** which enable the processor to perform designated signal processing functions. There are three possible implementations of the DSP, respectively identified as hard, soft and native. In the hard implementation, all three components of the DSP are fixed within a piece of hardware, i.e. an IC chip. In other words, the libraries and the operating system are embedded as firmware, and cannot be reprogrammed or updated without changing the chip. An example of a hard implementation is the Rockwell 9623 data-pump. This type of DSP might be able to perform only one class of virtual device operation, i.e. Function as a modem. When a hard DSP is employed for the real-time engine, the function control block **49** operates to initialize the processor at the outset of the operation of the real-time engine.

The soft implementation differs from the hard implementation in that the libraries, and possibly also the operating system, are resident as software in the computer's memory. In this implementation, the libraries are programmable and can be updated as desired. The processor, however, is still resident as a separate piece of hardware. Because of this programmability, the DSP can carry out more functions than a hard DSP, such as sound processing and Fourier transforms. An example of a soft DSP is the AT&T 3210. When a soft DSP is employed for the real-time engine, the function control block **49** operates to configure the appropriate libraries for the transforms that are to be carried out by the DSP, in addition to initializing the processor.

In the native implementation of the DSP, the processor does not reside in a separate piece of hardware. Rather, it's functions are carried out by the CPU of the host computer. As in the case of the soft implementation, the DSP operating system and the libraries are resident in the computer's memory. When this implementation is employed, the function control block **49** operates to allocate system resources to the DSP function, such as to enforce system time management, to ensure that adequate processing time is given to DSP operations.

The translation software **54** is made up of two parts. The main part of the software comprises a generic service provider **62** which functions as a device driver. This part of the software receives the commands from one of the APIs **48** and issues the instructions to the DSP to perform the transforms that are required in the operation of the virtual device being implemented. This part of the software is labeled as being generic because it is independent of the actual hardware that is used in the implementation of the DSP **52**. To enable the service provider to communicate with the DSP, an interface **64** is provided. This interface is specific to the particular DSP that is employed as the processing engine for the real-time engine. In other words, the generic service provider does not need to know whether the processing engine is a hard, soft or native DSP. In essence, therefore, the interface **64** functions as an additional layer of abstraction which virtualizes the DSP, i.e. the generic service provider is aware of the existence of a DSP, but does not need to know how it is actually being implemented in order to operate.

In a practical embodiment of the invention, separate generic service providers can be employed for the different virtual devices to be implemented. For example, one service provider can be employed to provide the services of a virtual telephone. Such a service provider might include a set of calls which enable it to determine the capabilities of the hardware being employed, e.g. whether it can support line current detection, remote wakeup, etc. Other sets of calls are used for control and status information, tone generation and detection, data transfer, and power management.

Another service provider can be used for sound applications, and a third service provider for video applications. Depending upon the particular virtual device to be implemented, the function control block **49** calls up the appropriate service provider when configuring the real-time engine. Each service provider communicates with the handler **44** through a respective one of the APIs **48**, and with the DSP **52** through the same interface **64**.

An example of a suitable interface **48** for telephony applications will now be described in detail. The interface basically operates to transmit high-level requests for service. The functionality of such an interface can be divided into two main categories, namely functions that are used only on public service telephone network (PSTN) lines, such as ring detection, and those functions used on any telephone line, such as DTMF generation and detection. This arrangement allows the interface to be used for ISDN and PBX lines as well as traditional analog lines for call progress and modem functions. For PSTN lines, the interface generates commands for setting the appropriate electrical parameters, such as voltage levels that comply with a particular country's regulations. For this purpose, the interface can include data tables containing information on all country-specific parameters.

The interface provides for the origination and answering of calls routed through traditional analog switches. To answer calls, the interface monitors incoming signals, as reported by the real-time engine, for appropriate frequency and cadence consistent with a particular country's requirements. The interface also includes facilities for tracking call progress, such as detection of dial tone, ring back and busy signals. It further includes the necessary information relating to the generation and detection of DTMF signals.

The interface generates calls that can be classified into two general categories, originating calls and callback calls. Originating calls are those which are generated in response to commands from the application program. Callback calls are used to report progress information to the application program. Most originating calls might take time to complete, and are therefore asynchronous, so that the host processor can suspend servicing of the calling application program until the task associated with the call is complete. This allows the interface to be called from within an interrupt handler as well as freeing the processor while waiting for some hardware to execute a task. Completion of the process is indicated by executing a completion routine for an associated callback.

The originating calls are of two types, system task calls and general purpose calls. The system task calls can include those such as "Open", which causes system resources to be allocated to the real-time engine, and "Close", which deallocates the resources. General purpose calls can include such calls as "State", which returns the current state of the virtual device, e.g. on-hook, ringing, off-hook or on-line, "GenerateDtmf" which causes DTMF tones to be generated, and "SetAutoAnswer", which instructs the engine to answer a call after a predetermined number of rings. Other examples of general purpose calls include "SetSilenceDuration", which passes to the engine the length of a silence to be detected, "Hook", which is used to take the virtual phone off-hook and on-hook, and "DialNumber", which dials a number in a designated string. Similar types of general purpose calls can be included for functions associated with facsimile types of operations.

Examples of suitable callback calls include "DtmfDetected", which indicates that a particular DTMF digit has been detected, "RingIndicate", which identifies when a valid ring has been detected on the line, and "DialToneDetected", which is called when a valid dial tone is detected. Other appropriate callback calls will be readily apparent from these examples.

The flow of events that occur when the fax/data modem is activated will now be described. These events are illustrated in the flow diagrams of FIGS. **4A–4C**. From the perspective of an application program **10**, it is "talking" to an external modem connected to the serial port. In fact, however, it is actually obtaining communications services from an internal virtual modem **20** and the hardware interface **22**.

At boot time, the computer's operating system determines whether the computer is capable of supporting a communications system. This allows the operating system to notify the user on application program activation that a communications session is not possible on the computer. This determination is made by assessing system-dependent factors such as presence of a data stream processor, sufficient system resources, and so forth. In the following discussion it is assumed that the requisite resources are available.

At the outset, with reference to FIG. **4A**, a communications application **10** is launched. A communications "connection" is opened either implicitly on launch, or by command. This directs the communications subsystem to initialize itself. The communications application's connection establishment request is passed to the communications toolbox **16**. This in turn causes a driver command "Open" to be issued by the fax/data modem logic **20** to the hardware interface **22**.

The hardware interface driver command "Open" is received by the hardware interface adapter handler **44**. As noted, the handler has previously—typically at boot time—determined that the host computer has the resources to establish a communications session, in this example being an analog modem over a telephone service line.

The handler calls function control block **49**, to initialize the real-time engine. The action taken by function control block depends on the real-time engine's implementation. If a programmable DSP is used for the real-time engine, the function control block might issue a series of DSP operating system specific commands to download and initialize the DSP subsystem, followed by commands to download the DSP algorithms that perform the modem's analog modulation. A native-mode DSP implementation may result in the function control block simply allocating system memory and host processor resources needed for the modem algorithms. In either case, it is significant to note that the hardware interface is designed to use a virtual real-time engine. The entire real-time engine implementation is "hidden" from the handler **44** by the function control block. The handler does not communicate with the real-time engine directly, via DSP-specific commands. Rather, all communications take place over the virtual real-time engine interface **48** via the mailboxes and the full-duplex data FIFO registers.

The handler **44**, as part of the Open process, prepares the attached telecom adapter **36** for operation. After initialization, the hardware interface telecom adapter delivers a full duplex, isochronous data stream.

If the real-time engine is successfully configured, the hardware interface is initialized properly, and all other necessary resources are available, the handler Open operation will be successful, and an analog modem communications link over the hardware interface adapter can begin at any time. Otherwise, communications are impossible and an error is reported to the application program.

Referring now to FIG. **4**B, the application program now initiates a modem connection with a remote station. For this purpose, the character sequence "ATDT5551212" might be issued by the application program, signifying (in the "AT" modem command standard) that the communications subsystem should dial the remote station at number 5551212, and instigate a modem connection with the answering modem, if present.

The dial command string is passed by the communications toolbox **16** to the CLI **24**, where it is parsed and converted into a virtual phone dial command. The virtual phone dial command is passed to the adapter handler **44** as a driver-level control call, where it is translated into a virtual real-time engine command and placed on the virtual real-time engine's command/response interface **48**. This causes the real-time engine to generate a PCM data stream that directs the hardware interface telecom adapter to go off-hook, then to generate the DTMF tones corresponding to the entered phone number. The handler **44** "sleeps", waiting for the real-time engine to signal that the real-time DTMF command has been completed. In this operation, the handler has no involvement with the isochronous data stream created by the real-time engine.

Referring now to FIG. **4**C, upon completion of the virtual phone dial command, the fax/data module issues a command to the virtual phone, i.e. the hardware interface, directing it to listen for an answering modem sequence. Again, this command is received by the handler **44** and is translated into a virtual real-time engine command to detect tones (signal energy) at certain specified frequencies and levels. Optionally, the real-time engine may be commanded to concurrently listen for voice energy in case a human answers the phone.

Assuming that the answering station presents valid answering modem tones, the handler will then be directed to instigate a modem connection. This results in another command to the virtual real-time engine, this time requesting a modem connection compatible with the answer tone sequence received. The actual modulation and demodulation of the hardware interface adapter's isochronous PCM data stream is accomplished entirely by the real-time engine.

Once the modem connection is established, the handler notifies the fax/data modem module that data transmission may begin. Digital data now flows between remote and local computers via handler Read and Write calls. Data is passed between the real-time engine and the handler via full-duplex FIFOs. This data is in turn passed between the handler and the application program through the modem logic **20**.

The ability to communicate over different types of transmission mediums in this single system is made possible by the fact that each of the various components is isolated from the particular features of the other through suitable levels of abstraction implemented via the application programming interfaces. For example, to change the transmission medium from the telephone lines to an ISDN line, the telecom adapter **36** is disconnected from the serial port **37**, and a new adapter appropriate for ISDN is plugged into the serial port. The associated adapter handler **44** is also loaded into the system. Thereafter, whenever the adapter handler issues a command to the real-time engine to perform a transform, it identifies the fact that the transformed data must be suitable for ISDN format. In response thereto, the API **48** which receives these commands supplies the real-time engine with the appropriate parameters for performing the transforms in the required format, e.g. the proper number of bits per word, etc.

Similarly, if the computer is transported from one country to another, the only change that needs to be implemented to carry out telephone communications in the new country is to switch the adapter and its handler. Upon initialization, the adapter identifies the fact that it is designed for a specific country. Whenever commands are to be sent to the real-time engine, the handler instructs the API **48** of the country as well as the command itself. For example, the command might be to generate a dial tone for country X. In response, the API **48** instructs the real-time engine to generate the dial tone, and provides it with the parameters pertinent to dial tones in country X. The real-time engine then generates the necessary PCM signals and supplies them to the adapter **36** via the DMA. The adapter takes care of converting those signals into the necessary electrical signals for transmission on the telephone lines of that country.

In essence, the real-time engine allows any type of transform to be performed on any type of data delivered over any type of transmission medium. The application program which receives the transformed data does not have to have any knowledge of the fact that the transmissions are being carried out over an ISDN line, rather than the telephone lines that it might have been originally programmed for. Thus, for example, a modem connection can be established over an ISDN line without the application being aware of a change in the transmission medium. The adapter **36**, the hardware abstraction layer **40**, the serial driver **42** and the adapter handler **44** function to configure a real-time data stream from the transmission medium to the real-time engine, and vice versa. The speed at which this data can be delivered, as well as the format of the data, is no longer limited by hardware devices that are employed in conventional hard-wired modems, particularly UARTs and the like. Rather, the data is delivered at a real-time rate, where it is handled by the computer's CPU.

The foregoing examples of the invention have been presented to facilitate an understanding of its features and operation. It will be appreciated, however, that the practical applications of the invention are not limited to these specific embodiments. Rather, the invention will find utility in any environment in which it is desirable to transmit and process data at real-time rates. Thus, while the invention has been described in the context of communications over a wide-area network, it can be used in any type of data acquisition system. The preceding description should therefore be viewed as exemplary, rather than restrictive. The scope of the invention is indicated by the following claims, rather than the foregoing description, and all changes which come within the meaning and range of equivalents thereof are intended to be embraced therein.

We claim:

1. A signal processing system for providing a plurality of realtime services to and from a number of independent client applications and devices, said system comprising:

a subsystem comprising a host central processing unit (CPU) operating in accordance with at least one application program and a device handler program, said subsystem further comprising an adapter subsystem interoperating with said host CPU and said device;

a realtime signal processing subsystem for performing a plurality of data transforms comprising a plurality of realtime signal processing operations; and

at least one realtime application program interface (API) coupled between the subsystem and the realtime signal processing subsystem to allow the subsystem to interoperate with said realtime services.

2. The signal processing system as set forth in claim **1**, wherein said signal processing system receives and transmits a plurality of datatypes over a plurality of different wide area networks (WANs).

3. A signal processing system for providing a plurality of realtime services over a wide area network (WAN), said system comprising:

a telecommunications subsystem comprising a host central processing unit (CPU) and a wide area network interface, where said wide area network interface is comprised of a hardware interface to the network and driver software which executes on the host

a telecommunications subsystem comprising a host central processing unit (CPU) operating in accordance with at least one application program and a datastream handler program, said telecommunications subsystem further comprising a telecommunications adapter subsystem interoperating with said host CPU and said WAN;

a realtime signal processing subsystem for performing a plurality of transforms comprising a plurality of realtime signal processing operations; and

at least one realtime application program interface (API) coupled between the telecommunications subsystem and the realtime signal processing subsystem to allow the telecommunications subsystem to interoperate with said realtime services.

4. The signal processing system as set forth in claim **3**, wherein the realtime signal processing subsystem comprises:

at least one realtime communications module coupled to receive a plurality of communications commands from said applications programs via said datastream handler program and said realtime APIs, said realtime communications module in response to said communications commands issuing a plurality of requests for realtime services to at least one realtime service provider;

a translation interface program coupled to receive said requests for realtime services from said communications modules; and

a realtime processor including a realtime operating system interoperating with said translation program for executing a plurality of realtime operations comprising realtime functions in response to said requests.

5. The realtime data processing system as set forth in claim **4**, wherein the translation interface comprises a plurality of realtime features to access a modem unit for communicating over said WAN.

6. The signal processing system as set forth in claim **3** further comprising a direct memory access (DMA) unit coupled between said realtime signal processing subsystem and a hardware abstraction portion of said telecommunications subsystem, said DMA unit providing for transfer of datablocks from said telecommunications adapter module to said realtime signal processing subsystem.

7. A signal processing system for providing a plurality of realtime services over a wide area network (WAN), said system comprising:

a telecommunications subsystem comprising a host central processing unit (CPU) operating in accordance with at least one applications program and a datastream handler program, said telecommunications subsystem further comprising a telecommunications adapter subsystem interoperating with said host CPU and said WAN;

a virtual realtime device enabling a plurality of realtime signal processing operations in accordance with at least one realtime service request issued by said applications program; and

at least one realtime application program interface (API) interoperating with the telecommunications subsystem and the virtual realtime device to enable the telecommunications subsystem to interoperate with said realtime signal processing operations.

8. The signal processing system as set forth in claim **7**, wherein the virtual realtime device comprises a realtime translation interface program and virtual realtime engine, said virtual realtime engine enabling said realtime services by performing a number of data translation operations in accordance with said realtime service request and said realtime translation interface program.

9. The signal processing system as set forth in claim **8**, wherein the virtual realtime engine comprises:

a realtime processor including a realtime operating system, and

a plurality of realtime function libraries interoperatively coupled with said realtime processor for providing a plurality of processing steps comprising said realtime signal processing operations,

whereby said virtual realtime engine responds to communications commands initiated by said applications programs.

10. The signal processing system as set forth in claim **7** further comprising a direct memory access (DMA) unit coupled between said virtual realtime signal processing subsystem and a hardware abstraction portion of said telecommunications subsystem, said DMA unit providing for transfer of data from said telecommunications adapter module to said virtual realtime signal processing subsystem.

11. The signal processing system as set forth in claim **7**, wherein the virtual realtime translation interface comprises a plurality of realtime features to access a modem unit for communicating over said WAN.

12. The signal processing system as set forth in claim **11**, wherein the modem unit comprises at least a serial communications controller, a programmable timer, and a plurality of input/output (I/O) lines.

13. The signal processing system as set forth in claim **7**, wherein realtime service requests are selected from the group of realtime service request devices consisting of telephone answering machines, automatic telephone dialing machines, and remote control systems.

14. The signal processing system as claimed in claim **6**, wherein the realtime signal processing operations are selected from the group of telecommunications transactions consisting of fax (send/receive) and data transmission transactions.

15. The signal processing system as claimed in claim **14**, wherein the data transmission transaction comprises at least one data framing format and at least one data protocol.

16. The virtual realtime data processing system as set forth in claim **9**, wherein the realtime processor comprises a programmable processing unit which is controlled by said realtime communications applications and said realtime communications interface.

17. The virtual realtime data processing system as set forth in claim **9**, wherein the realtime processor comprises said host CPU.

18. The signal processing system as set forth in claim **9**, wherein the realtime processor comprises a hard digital signal processor in which the realtime operating system and the realtime function libraries are fixedly embodied in a hardware element.

19. The realtime data processing system of claim **4** comprising a plurality of realtime communications modules which are respectively associated with different realtime services.

20. The realtime data processing system of claim **19** wherein at least some of said realtime communications modules provide a service which implements a virtual realtime device.

21. The realtime data processing system of claim **19** wherein one of said realtime services comprises a video processing service.

22. The realtime data processing system of claim **19** wherein one of said realtime services comprises a sound processing service.

23. The realtime data processing system of claim **19** wherein one of said realtime services comprises a telephone service.

24. The signal processing system of claim **1**, wherein said realtime signal processing subsystem comprises:

a realtime processor including an operating system for executing a plurality of realtime functions;

a realtime communications module which is independent of said realtime processor and is coupled to receive a plurality of communications commands from said application programs via said device handler program and said realtime API, said realtime communications module operating in response to said communications commands to issue a plurality of requests for realtime services to said realtime processor; and

a translation interface program which is specific to said realtime processor and is coupled to receive said requests for realtime services from said communications module and provide said requests to said realtime processor.

25. The signal processing system of claim **24** comprising a plurality of realtime communications modules which are respectively associated with different realtime services.

26. The signal processing system of claim **25** wherein at least some of said realtime communications modules provide a service which implements a virtual realtime device.

27. The signal processing system as set forth in claim **24**, wherein the realtime processor comprises a hard digital signal processor in which said operating system and realtime function libraries are fixedly embodied in a hardware element.

28. The signal processing system as set forth in claim **24**, wherein the realtime processor comprises said host CPU.

29. The signal processing system of claim **24**, wherein said realtime processor is embodied in a hardware device and includes realtime function libraries that are embodied in programmable software.

30. The signal processing system of claim **29** wherein said operating system is also embodied in programmable software.

31. A signal processing system, comprising:

an input/output device for sending and/or receiving isochronous streams of data transmitted over a communications path;

a realtime engine for performing data transformations on the isochronous streams of data, said realtime engine being independent of said input/output device;

a device handler program associated with said input/output device, for generating requests to the realtime engine to perform data transformations on the isochronous streams of data; and

at least one application programming interface for receiving the requests generated by said device handler program and issuing commands to said realtime engine to perform the requested data transformations.

32. The signal processing system of claim **31** comprising a plurality of said application programming interfaces which are respectively associated with different types of services to be provided by said realtime engine with respect to isochronous streams of data.

33. The signal processing system of claim **32** wherein one of said application programming services relates to the operation of the realtime engine as a virtual telephone device.

34. The signal processing system of claim **32** wherein one of said application programming services relates to the operation of the realtime engine as a virtual sound device.

35. The signal processing system of claim **32** wherein one of said application programming services relates to the operation of the realtime engine as a virtual video device.

36. The signal processing system of claim **31** wherein said application programming interface includes command/control registers that are shared between said realtime engine and said device handler program for transferring said requests and responding thereto, and a buffer for transferring isochronous streams of data between said device handler program and said realtime engine.

37. The signal processing system of claim **31**, wherein said realtime engine comprises:

a realtime processor including an operating system for executing a plurality of realtime functions;

a communications module which is independent of said realtime processor and is coupled to receive said commands from said application programming interface, said communications module operating in response to said commands to issue a plurality of requests for realtime services to said realtime processor; and

a translation interface program which is specific to said realtime processor and is coupled to receive said requests for realtime services from said communications module and provide said requests to said realtime processor.

38. The signal processing system as set forth in claim **37**, wherein the realtime processor comprises a hard digital signal processor in which said operating system and realtime function libraries are fixedly embodied in a hardware element.

39. The signal processing system as set forth in claim **37**, wherein said processing system is incorporated in a data processing system having a host central processing unit (CPU), and wherein the realtime processor comprises said host CPU operating in accordance with software instructions relating to said realtime functions.

40. The signal processing system of claim **37**, wherein said realtime processor is embodied in a hardware device and includes realtime function libraries that are embodied in programmable software.

41. The signal processing system of claim **40** wherein said operating system is also embodied in programmable software.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,343,263 B1 Page 1 of 1
DATED : January 29, 2002
INVENTOR(S) : James B. Nichols and John Lynch

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [75], Inventors, "**James B. Nichols**", please delete "San Mateo" as city and insert -- Los Altos -- in it's place.
"**John Lynch**", please delete "San Jose" as city and insert -- Monte Sereno -- in it's place.

Signed and Sealed this

Twenty-sixth Day of November, 2002

Attest:

[illegible]

JAMES E. ROGAN

Attesting Officer *Director of the United States Patent and Trademark Office*

EXHIBIT 10

US006424354B1

(12) **United States Patent**
Matheny et al.

(10) **Patent No.: US 6,424,354 B1**
(45) **Date of Patent: Jul. 23, 2002**

(54) **OBJECT-ORIENTED EVENT NOTIFICATION SYSTEM WITH LISTENER REGISTRATION OF BOTH INTERESTS AND METHODS**

(75) Inventors: **John R. Matheny**; **Christopher White**, both of Mountain View; **David R. Anderson**, Cupertino; **Arn J. Schaeffer**, Belmont, all of CA (US)

(73) Assignee: **Object Technology Licensing Corporation**, Cupertino, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/287,172**

(22) Filed: **Apr. 1, 1999**

Related U.S. Application Data

(63) Continuation of application No. 07/996,775, filed on Dec. 23, 1992, now Pat. No. 6,259,446.

(51) **Int. Cl.**^{7} **G06F 13/00**
(52) **U.S. Cl.** **345/619**; 345/700; 345/764
(58) **Field of Search** 345/619, 621, 345/623, 624, 625, 700, 716, 764

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,658,427 A	4/1972	DeCou
3,881,605 A	5/1975	Grossman

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 150 273	8/1985
EP	150 273 A	8/1985
EP	352 908 A	1/1990
EP	0 352 908	1/1990
EP	398 646 A	11/1990
EP	499 404 A	8/1992
EP	0 506 102	9/1992
EP	506 102 A	9/1992
EP	529 770 A	3/1993
EP	0 529 770	3/1993
WO	WO 92/15934	9/1972
WO	WO 92/15934 A	9/1992

OTHER PUBLICATIONS

IBM Programming Guide, Sep., 1989, First Edition, "Operating System/2 Programming Tools and Information" Version 1.2, pp. 3–7 through 3–18 and 7–1 through 7–28.

Schumaker, Kurt J., "Object–Oriented Languages: MACAPP: An Application Framework", Byte, Aug., 1986, pp. 189–193.

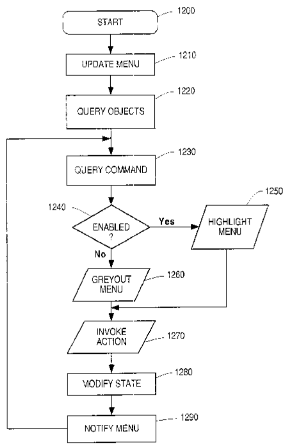
(List continued on next page.)

Primary Examiner—Matthew Luu
(74) *Attorney, Agent, or Firm*—Morgan & Finnegan, LLP

(57) **ABSTRACT**

An event notification system for propagating object-change information. The notification system supports change notification without queues in an object-based application or operating system and can be scaled to propagate large numbers of events among a large plurality of objects. The event notification system interconnects a plurality of event source and event receiver objects. Any object, such as a command object, may operate as either an event receiver object, an event source object or both. A notification object is created by a source object to transport, from a source to a receiver, descriptive information about a change, which includes a particular receiver object method and a pointer to the source object that sent the notification. A receiver object must register with a connection object its "interest" in receiving notification of changes; specifying both the event type and the particular source object of interest. After establishing such connections, the receiver object receives only the events of the specified type for the source objects "of interest" and no others. This delegation of event selection avoids central event queuing altogether and so limits receiver object event processing that the invention can be scaled to large systems operating large numbers of objects.

59 Claims, 15 Drawing Sheets



U.S. PATENT DOCUMENTS

4,082,188	A	4/1978	Grimmell et al.	
4,635,208	A	1/1987	Coleby et al.	
4,677,576	A	6/1987	Berlin, Jr. et al.	
4,679,137	A	7/1987	Lane et al.	364/188
4,686,522	A	8/1987	Hernandez et al.	
4,704,694	A	11/1987	Czerniejewski	
4,742,356	A	5/1988	Kuipers	
4,760,386	A	7/1988	Heath et al.	340/709
4,821,220	A	4/1989	Duisberg	
4,823,283	A	4/1989	Diehm et al.	364/518
4,831,654	A	5/1989	Dick	381/51
4,835,685	A	5/1989	Kun	364/200
4,843,538	A	6/1989	Lane et al.	364/188
4,853,843	A	8/1989	Ecklund	364/200
4,868,744	A	9/1989	Reinsch et al.	364/280.3
4,885,717	A	12/1989	Beck et al.	
4,891,630	A	1/1990	Friedman et al.	
4,931,783	A	6/1990	Atkinson	340/710
4,939,648	A	7/1990	O'Neill et al.	
4,943,932	A	7/1990	Lark et al.	364/513
4,953,080	A	8/1990	Dysart et al.	
4,982,344	A	1/1991	Jordan	364/521
5,008,810	A	4/1991	Kessel et al.	364/200
5,040,131	A	8/1991	Torres	364/521
5,041,992	A	8/1991	Cunningham et al.	
5,050,090	A	9/1991	Golub et al.	
5,060,276	A	10/1991	Morris et al.	
5,075,848	A	12/1991	Lai et al.	
5,083,262	A	1/1992	Haff, Jr.	395/500
5,093,914	A	3/1992	Coplien et al.	
5,119,475	A	6/1992	Smith et al.	
5,125,091	A	6/1992	Staas, Jr. et al.	
5,129,084	A	7/1992	Kelly et al.	395/650
5,133,075	A	7/1992	Risch	
5,136,705	A	8/1992	Stubbs et al.	
5,140,677	A	8/1992	Fleming et al.	395/159
5,151,987	A	9/1992	Abraham et al.	
5,163,130	A	11/1992	Hullot	395/148
5,168,411	A	12/1992	Fujii	
5,168,441	A	12/1992	Onarheim et al.	364/146
5,177,685	A	1/1993	Davis et al.	
5,181,162	A	1/1993	Smith et al.	
5,198,802	A	3/1993	Bertram et al.	340/709
5,206,951	A	4/1993	Khoyi et al.	395/650
5,228,123	A	7/1993	Heckel	395/155
5,230,063	A	7/1993	Hoeber et al.	395/156
5,237,654	A	8/1993	Shakelford et al.	395/160
5,239,287	A	8/1993	Siio et al.	340/706
5,241,655	A	8/1993	Mineki et al.	395/156
5,265,206	A	11/1993	Shackelford et al.	
5,276,775	A	1/1994	Meng	395/55
5,276,816	A	1/1994	Cavendish et al.	395/275
5,280,610	A	1/1994	Travis et al.	395/600
5,287,448	A	2/1994	Nicol et al.	395/159
5,291,587	A	3/1994	Kodosky et al.	395/500
5,295,222	A	3/1994	Wadhwa et al.	395/1
5,295,256	A	3/1994	Bapat	395/500
5,297,253	A	3/1994	Meisel	395/160
5,297,284	A	3/1994	Jones et al.	395/700
5,301,301	A	4/1994	Kodosky et al.	395/500
5,301,336	A	4/1994	Kodosky et al.	395/800
5,309,566	A	5/1994	Larson	395/275
5,313,629	A	5/1994	Abraham et al.	395/600
5,313,636	A	5/1994	Noble et al.	395/700
5,315,703	A	5/1994	Matheny et al.	
5,315,709	A	5/1994	Alston et al.	395/600
5,317,741	A	5/1994	Schwanke	395/700
5,321,841	A	6/1994	East et al.	395/725
5,325,481	A	6/1994	Hunt	395/159
5,325,522	A	6/1994	Vaughn	395/600
5,325,524	A	6/1994	Black et al.	395/600
5,325,533	A	6/1994	McInerney et al.	395/700
5,327,529	A	7/1994	Fults et al.	
5,329,446	A	7/1994	Kugimiya et al.	364/419.04
5,339,433	A	8/1994	Frid-Nielsen	395/700
5,345,550	A	9/1994	Bloomfield	395/156
5,347,626	A	9/1994	Hoeber et al.	395/156
5,367,633	A	11/1994	Matheny et al.	
5,371,846	A	12/1994	Bates	395/157
5,371,851	A	12/1994	Pieper et al.	395/164
5,371,886	A	12/1994	Britton et al.	395/600
5,375,164	A	12/1994	Jennings	379/88
5,386,556	A	1/1995	Hedin et al.	395/600
5,390,314	A	2/1995	Swanson	395/500
5,414,812	A	5/1995	Filip et al.	395/200
5,416,903	A	5/1995	Malcolm	395/155
5,434,965	A	7/1995	Matheny et al.	395/159
5,446,902	A	8/1995	Islam	
5,479,601	A	12/1995	Matheny et al.	395/155
5,497,319	A	3/1996	Chong et al.	364/419.02
5,517,606	A	5/1996	Matheny et al.	395/156
5,530,864	A	6/1996	Matheny et al.	395/700
5,550,563	A	8/1996	Matheny et al.	345/168
5,551,055	A	8/1996	Matheny et al.	395/882
5,583,982	A	12/1996	Matheny et al.	395/326
5,717,877	A	2/1998	Orton et al.	395/326

OTHER PUBLICATIONS

Wang et al., "An Event–Object Recovery Model For Object–Oriented User Interfaces", Fourth Annual Symposium on User Inteface Software and Technology: Proceedings of the ACM Symposium on User Interface Software and Technology, Nov. 11, 1991, pp. 107–115.

Microsoft Systems Journal, Jan., 1990, vol. 5. No. 1, "Software Architecture Object–Oriented Programming Design", p. 14 (3).

Coop–Berre, "An Object Oriented Framework for Systems Integration", pp. 104–107.

Microsoft Corp., Windows User's Guide for Version 3.1, 1990–1992, pp. 52, 83–85.

Microsoft Corp., "A Presentation Manager Primer," *Microsoft Systems Journal*, Jan. 1990, v5, n1, pp. 14–17.

Berre, Arne–Jørgen, "COOP—An Object Oriented Framework for Systems Integration," *ICSI'92 Proc. 2nd Int'l Conf. On Systems Integration*, Jun. 15, 1992, Morristown, NJ, pp. 104–113.

Hirakawa et al, "A Framework for Construction of Icon Systems," IEEE, 1998, pp. 70–77.

IBM Corp., "Systems Application Architecture, Common User Access, Advanced Interface Design Guide," Jun. 1989, pp. 55–81, 97–99.

Apple Computer, Inc., "System 7–Macintosh Reference Guide," 1992, Cupertino, CA, pp. 30, 70, 72, 75.

Booch, Grady, "Object Oriented Design with Applications," 1991, pp. 45–6, 65 & 494.

Campbell et al., "Choices, Frameworks and Refinement," *Proc. Int'l Workshop on Object Orientation in Operating Systems*, Oct. 17, 1991, Palo Alto, CA, pp. 9–15.

Cobb et al, "Examining NewWave, Hewlett–Packard's Graphical Object–Oriented Environment," *Microsoft Systems Journal*, Nov. 1989, pp. 1–18 and Exhibits A–B.

Embry et al., "An Open Network Management Architecture: OSI/NM Forum Architecture and Concepts," *IEEE Network Magazine*, Jul. 1990, pp. 14–22.

Franz, Marty, "Object–Oriented Programming Featuring ACTOR," 1990, Chapters 1–2 & 19–22.

IBM Corp., "Dynamic Icon Presentation," *IBM Technical Disclosure Bulletin*, V.35, N.4B, Sep. 1992, Armonk NY, pp. 227–232.
IBM Corp., "Pause Review: a Technique for Improving the Interactivity of Direct Manipulation," *IBM Technical Disclosure Bulletin*, V.34, N.7A, Dec. 1991, Armonk, NY, pp. 20–25.
IBM Corp., "Auto Scroll During Direct Manipulation," *IBM Technical Disclosure Bulletin*, V.33, N.11, Apr. 1991, Armonk NY, p. 312.
IBM Corp., "Volume 3: Presentation Manager and Workplace Shell," O/S/2 Version 2.0, Apr. 1992, IBM Corporation International Technical Support Center, Boca Raton, FL, p. 53.
IBM Corp., "Presentation Manager Programming Reference," Volume III, OS/2 Technical Library, Mar. 1992.
IBM Corp., "Programming Guide," *Operating System/2 Programming Tools and Information Version 1.2*, Sep. 1989, pp. 3–7 to 3–18 and 7–1 to 7–28.
IBM Corp., "Getting Started: Using IBM Risc System/6000," Jan. 1992.
Khoshafian, Setrag, "Intelligent Offices, Object–Oriented Multi–Media Information Management in Client/Server Architectures," 1992, Chapter 8, pp. 235–304.
Meyrowitz, Norman, "Intermedia: The Architecture and Construction of an Object–Oriented Hypermedia System and Applications Framework," OOPSLA '86 Conference Proceedings, Sep. 29–Oct. 2, 1986, Portland, OR, pp. 186–201.
Microsoft Corp., "Window User's Guide for Version 3.0," 1990, pp. 128–133.
Microsoft Corp., "MS–DOS User's Guide," 1988, pp. 21–25, 77–80 & 165–170.
Miyauchi et al., "An Implementation of Management Information Base," IEEE, 1991, pp. 318–321.
Myers et al, "Environment for Rapidly Creating Interactive Design Tools," *The Visual Computer*, v.8, No. 2, Feb. 1992, Berlin, DE, pp. 94–116.
Myers, Brad, "Creating Interaction Techniques by Demonstration," *IEEE Computer Graphics and Applications*, V.7, N.9, Sep. 1987, New York, US, pp. 55–61.
Reiss, Steven P., "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, Jul. 1990, pp. 57–66.
Schmucker, Kurt, "MACAPP: An Application Framework," *Byte Magazine*, Aug. 1986, pp. 189–193.
Smith, R.B., "The Alternate Reality Kit," *IEEE, Proceedings of Workshop on Visual Languages*, Jun. 25, 1986, Dallas, TX, pp. 99–106.
Microsoft Corp., Windows User Guide for Version 3.1, 1990–1992, pp. 52, 83–85.
Microsoft Corp., "A Presentation Manager Primer", Microsoft Systems Journal, Jan. 1990, v5, n1, pp. 14–17.
Apple Computer, Inc., "System 7–Macintosh Reference Guide, " 1992, Cupertino, CA, pp. 30, 70, 72, 75.
Booch, Grady, "Object Oriented Design with Applications", 1991, pp. 45–6, 65 and 494.
Campbell et al., "Choices, Frameworks and Refinement, " Proc. Int'l Workshop on Object Orientation in Operating Systems, Oct. 17, 1991, Palo Alto, CA. pp. 9–15.
Cobb et al., "Examining NewWave, Hewlett–Packard's Graphical Object–Oriented Environment, " Microsoft Systems Journal, Nov. 1989, pp. 1–18 and Exhibits A–B.
Dodani et al., "Separation of Powers, " Byte Magazine, v. 143, Mar. 1989, pp. 255–271.
Embry et al., "An Open Network Management Architecture: OSI/NM Forum Architecture and Concepts," IEEE Network Magazine, Jul. 1990, pp. 14–22.
IBM Corp., "Dynamic Icon Presentation, " IBM Technical Disclosure Buletin, V35, no4B, Sep. 1992, Armonk, NY, pp. 227–232.
IBM Corp., "Pause Review: A Technique for Improving the Interactivity of Direct Manipulation, " IBM Technical Disclosure Buletin, v34, n7A, Dec. 1991, Armonk, NY, pp. 20–25.
IBM Corp., "Auto Scroll During Direct Manipulation, " IBM Technical Disclosure Buletin, v33, n11, Apr. 1991, Armonk, NY, p. 312.
IBM Corp., "vol. 3: Presentation Manager and Workplace Shell, " O/S/2 Version 2.0, Apr. 1992, IBM Corporation International Technical Support Center, Bocal Raton, FL, p. 53.
IBM Corp., "Presentation Manager Programming Reference, " vol. III, OS/2 Technical Library, Mar. 1992.
IBM Corp., "Getting Started: Using IBM RISC System/6000", Jan. 1992.
Microsoft Corp., "Window User's Guide for Version 3.0" 1990, pp. 128–133.
Microsoft Corp., "MS–DOS User's Guide", 1988, pp. 21–25, 77–80 and 165–170.
Miyauchi et al., "An Implementation of Management Information Base, " IEEE, 1991, pp. 318–321.
Reiss, Steven P., "Connecting Tools Using Message Passing in the Field Environment, " IEEE Software, Jul. 1990, pp. 57–66.
Smith, R.B., "The Alternate Reality Kit, " IEEE, Proceedings of Workshop on Visual Languages, Jun. 25, 1986, Dallas, TX, pp. 99–106.
Williams, Greg, "Software Frameworks, " Byte Magazine, Dec. 1984, pp. 124–127 and 394–410.

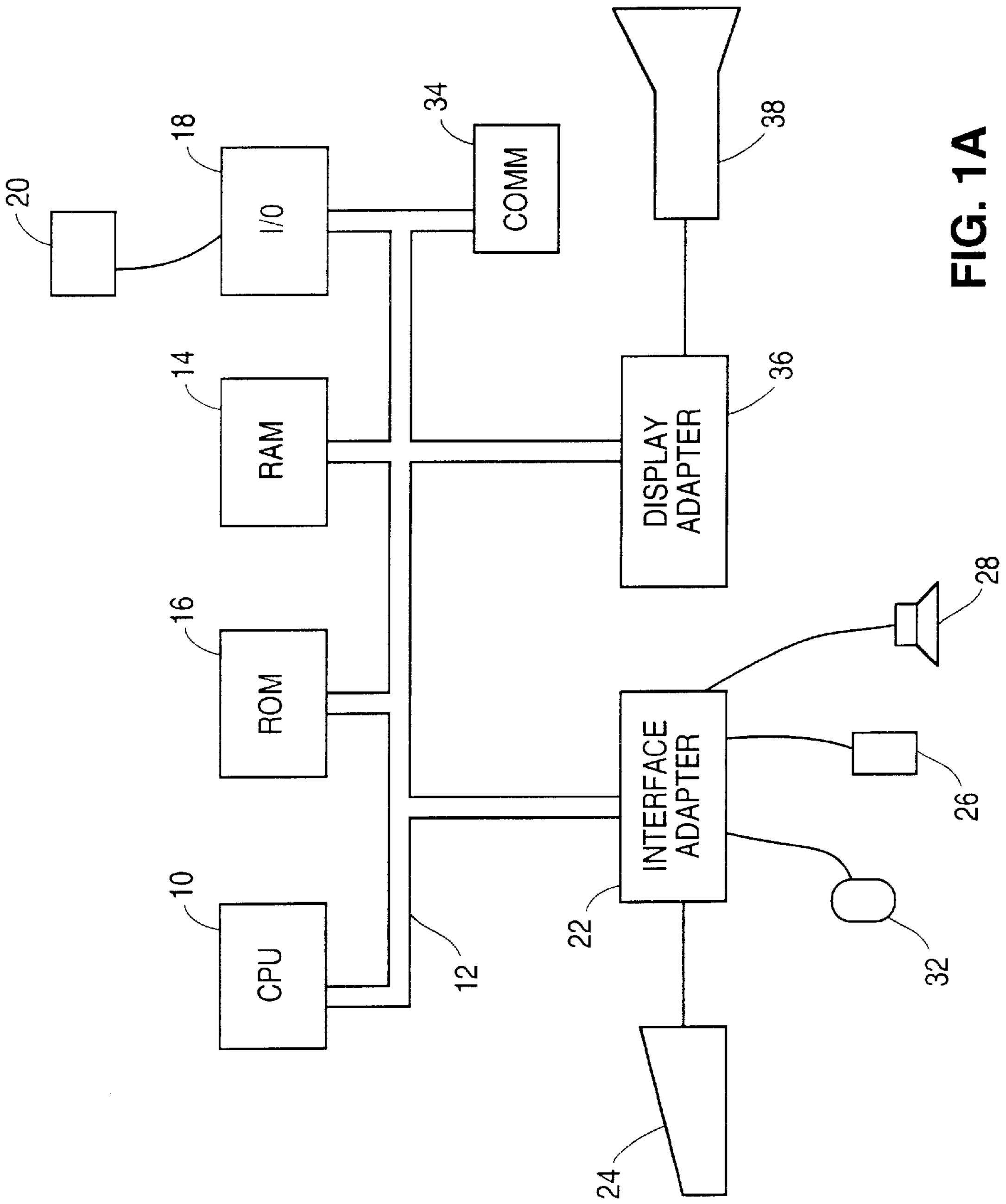
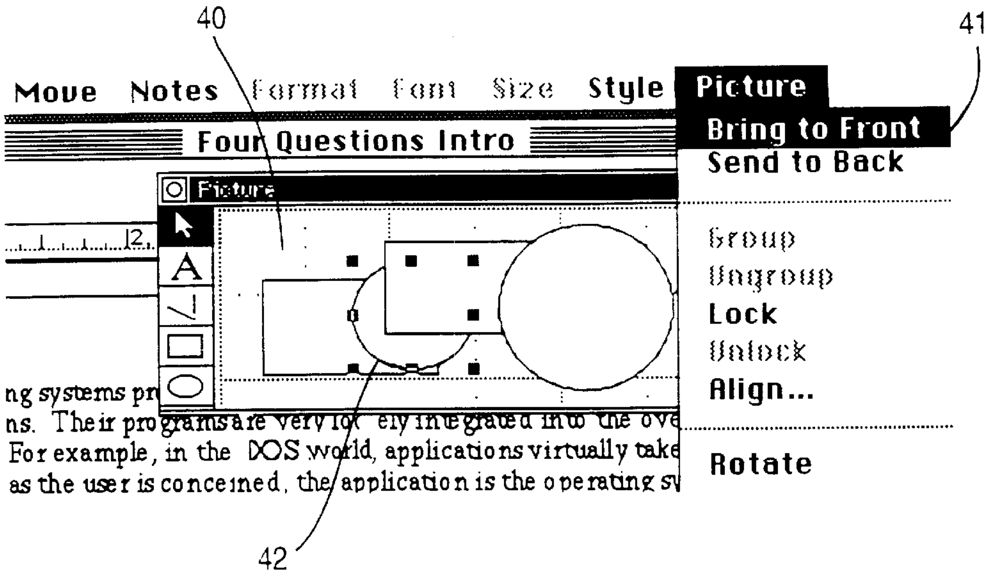

CPU
10
ROM
16
RAM
14
I/O
18
20
COMM
34
12
DISPLAY ADAPTER
36
38
INTERFACE ADAPTER
22
24
26
28
32

FIG. 1A



ng systems pr
ns. Their programs are very lo ely integrated into the ove
For example, in the DOS world, applications virtually take
as the user is concerned, the application is the operating sy

FIG. 1B

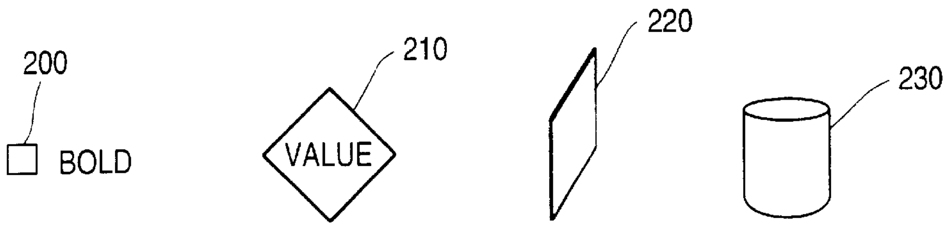


FIG. 2

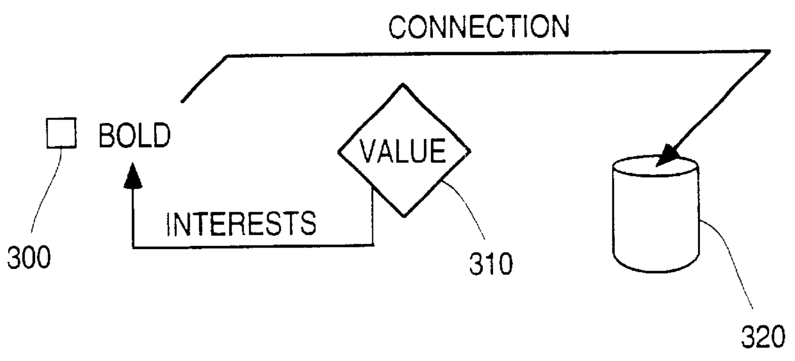


FIG. 3

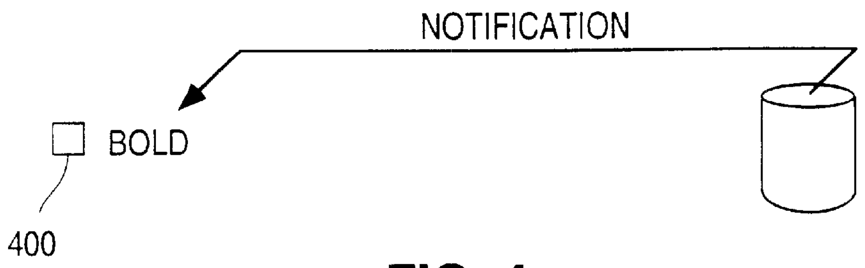


FIG. 4

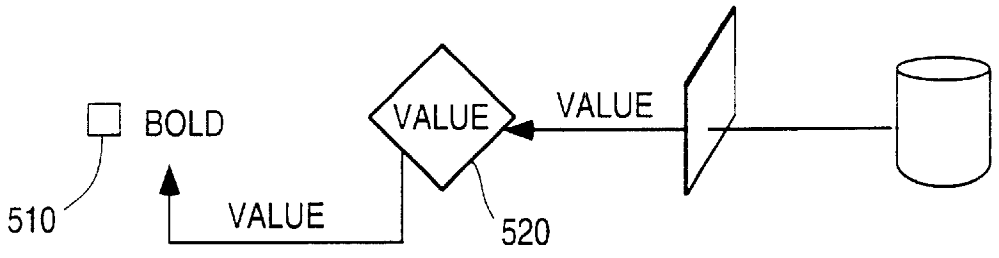


FIG. 5

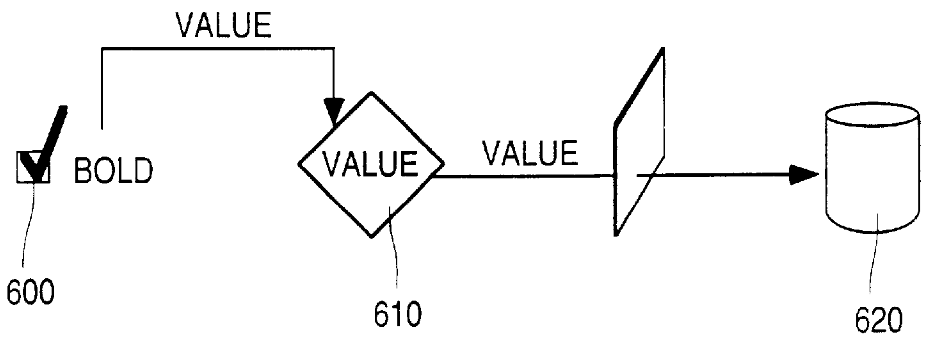


FIG. 6

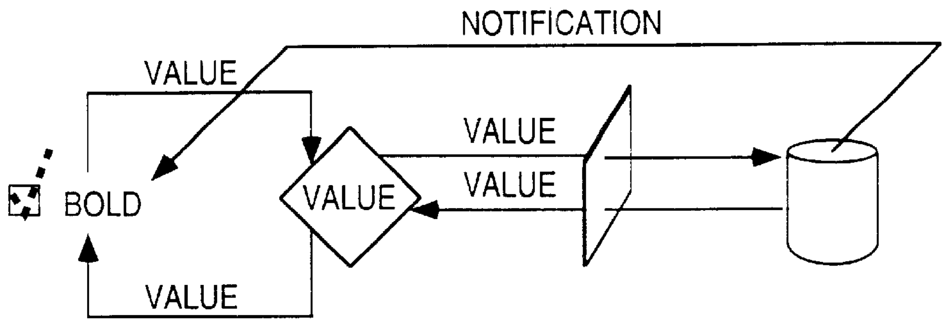


FIG. 7

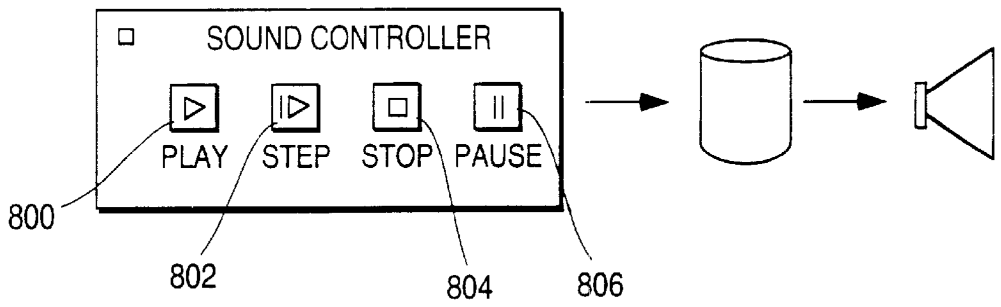


FIG. 8

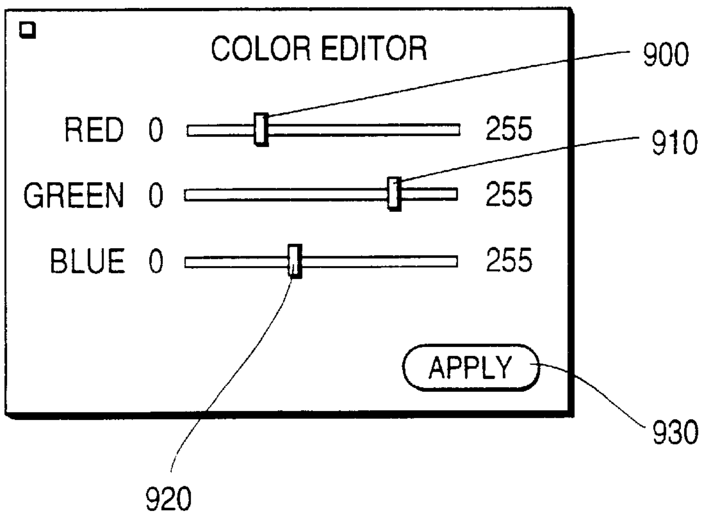


FIG. 9

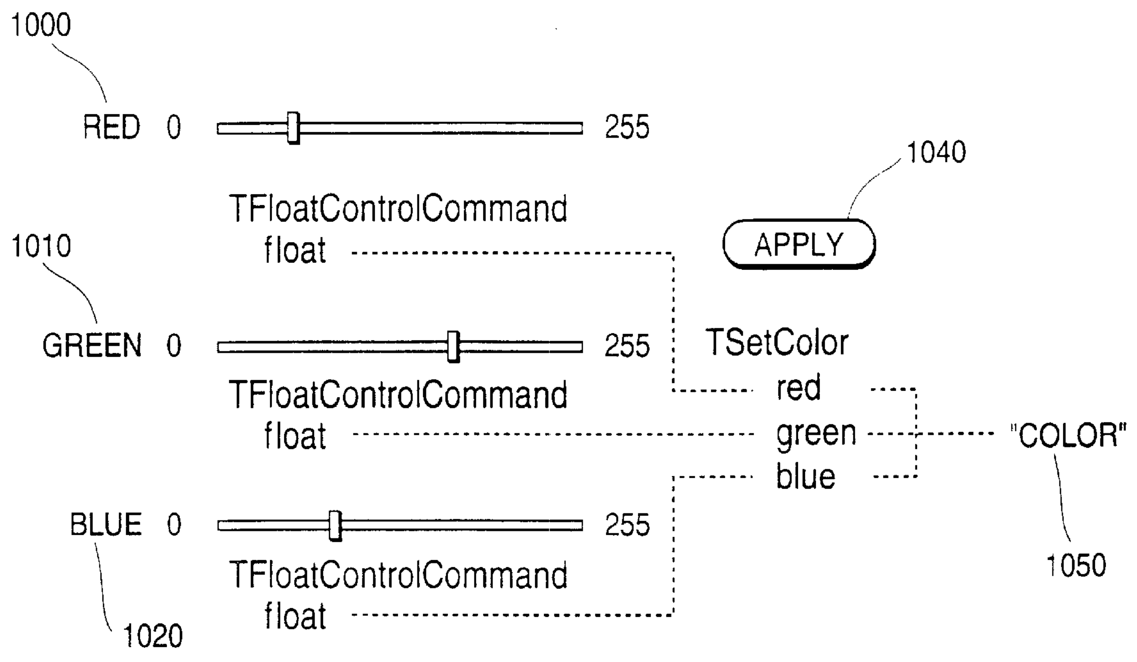

1000
RED 0
255
TFloatControlCommand
float
1010
GREEN 0
255
TFloatControlCommand
float
BLUE 0
255
TFloatControlCommand
float
1020
1040
APPLY
TSetColor
red
green
blue
"COLOR"
1050

FIG. 10

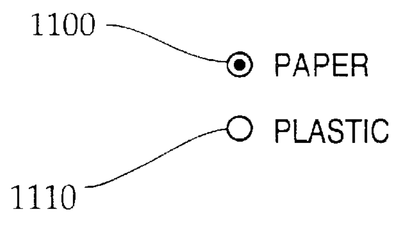

1100
PAPER
PLASTIC
1110

FIG. 11

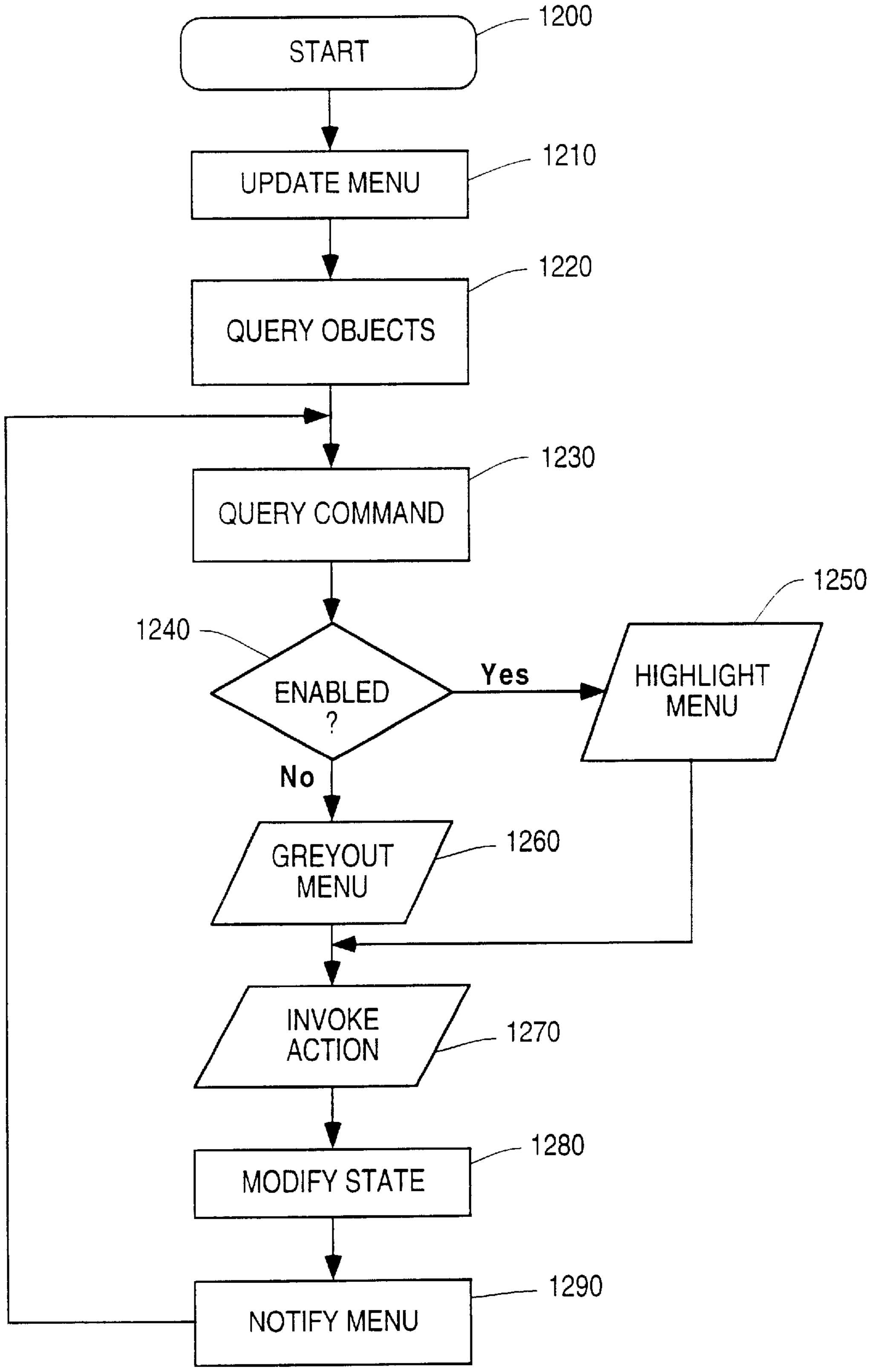

START
1200
UPDATE MENU
1210
QUERY OBJECTS
1220
QUERY COMMAND
1230
1240
ENABLED ?
Yes
HIGHLIGHT MENU
1250
No
GREYOUT MENU
1260
INVOKE ACTION
1270
MODIFY STATE
1280
NOTIFY MENU
1290

FIG. 12

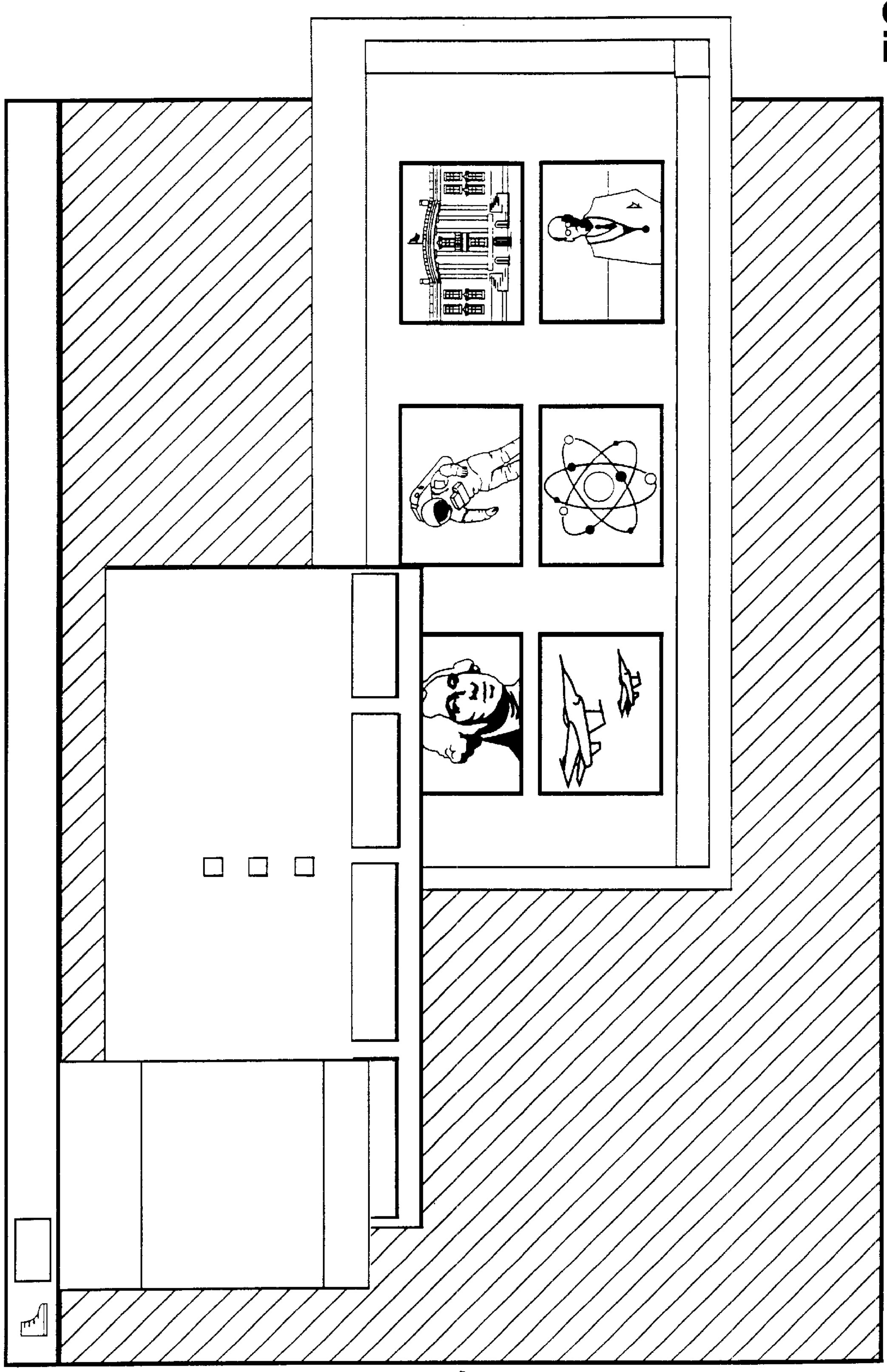


FIG. 13

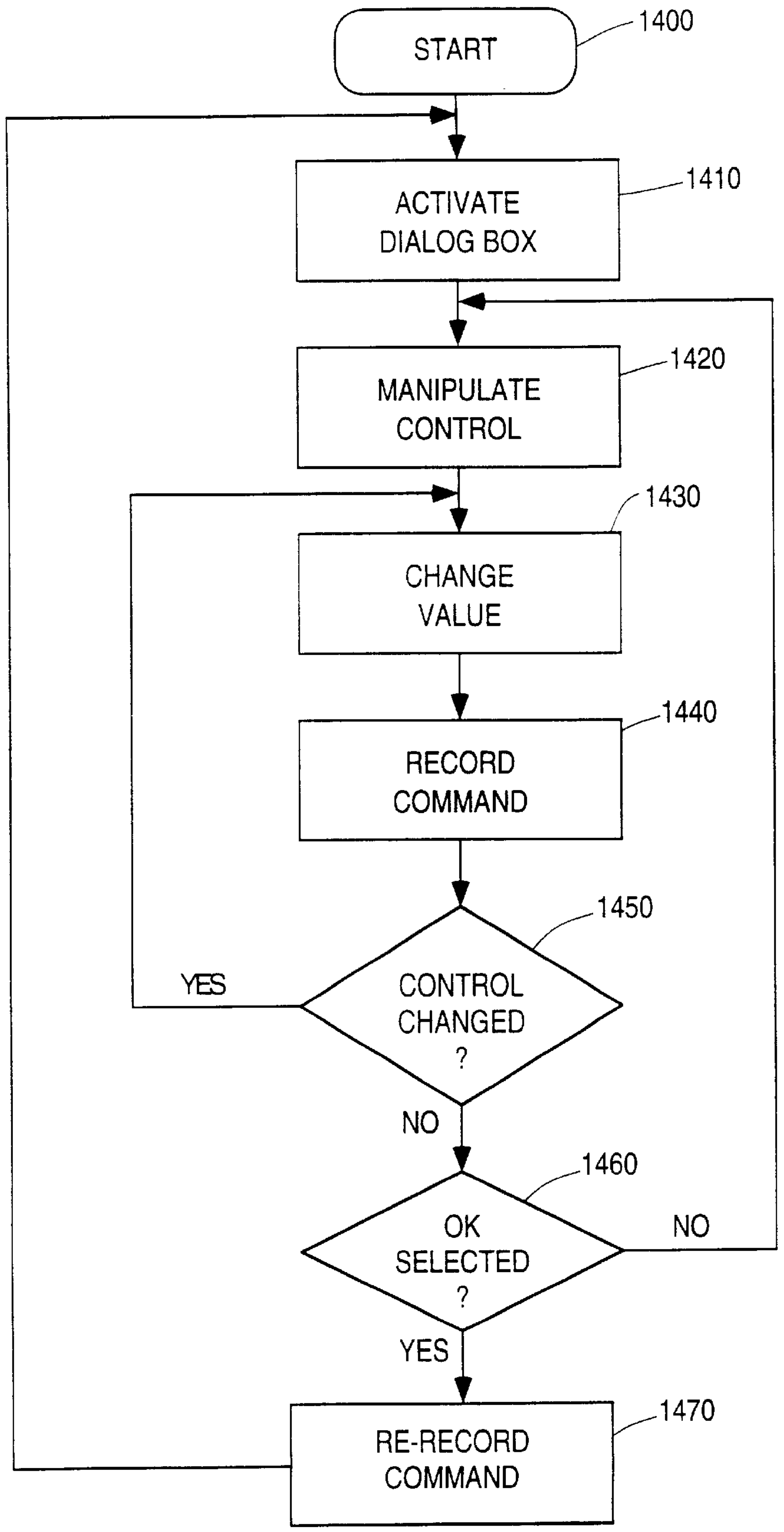

START
1400
ACTIVATE DIALOG BOX
1410
MANIPULATE CONTROL
1420
CHANGE VALUE
1430
RECORD COMMAND
1440
CONTROL CHANGED ?
1450
YES
NO
OK SELECTED ?
1460
NO
YES
RE-RECORD COMMAND
1470

FIG. 14

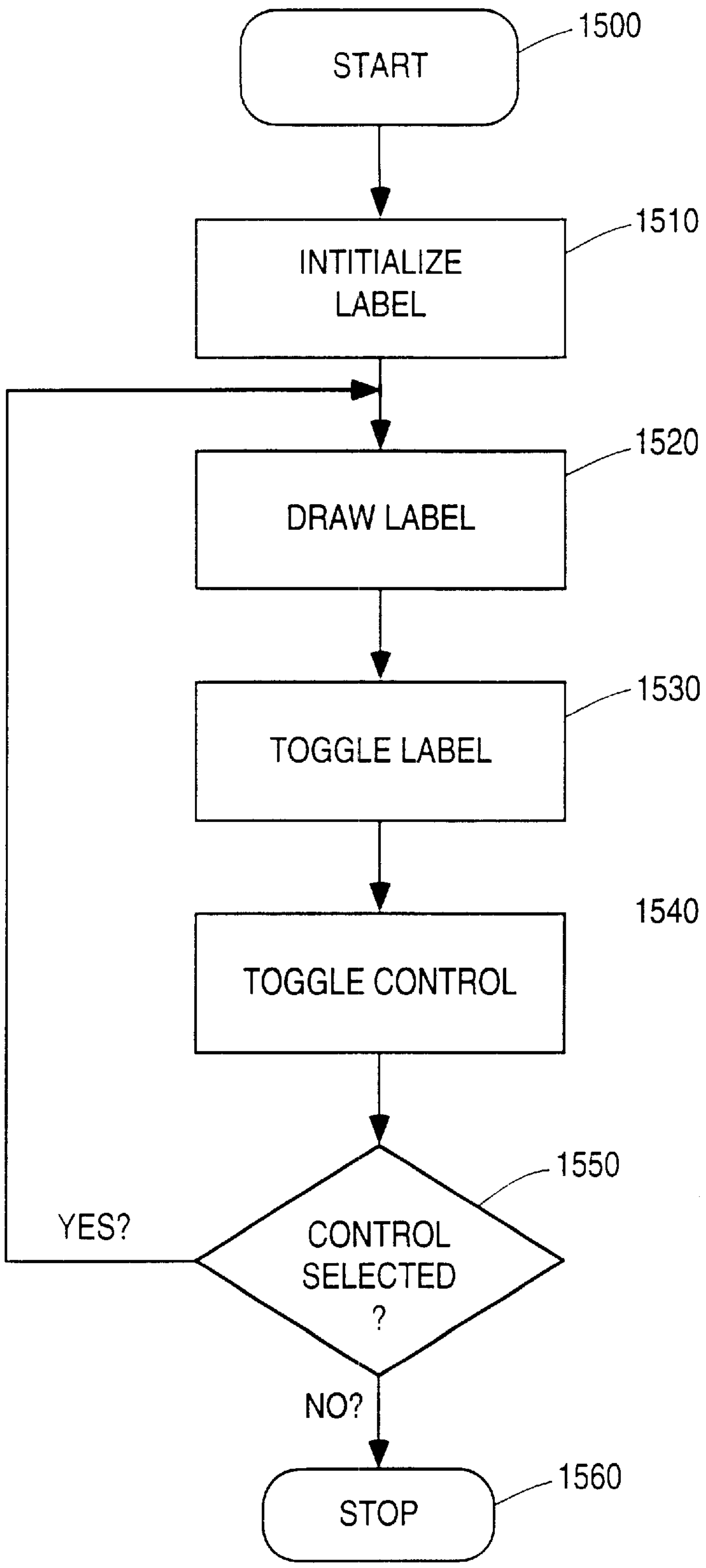

START
1500
INTITIALIZE LABEL
1510
DRAW LABEL
1520
TOGGLE LABEL
1530
TOGGLE CONTROL
1540
CONTROL SELECTED ?
1550
YES?
NO?
STOP
1560

FIG. 15

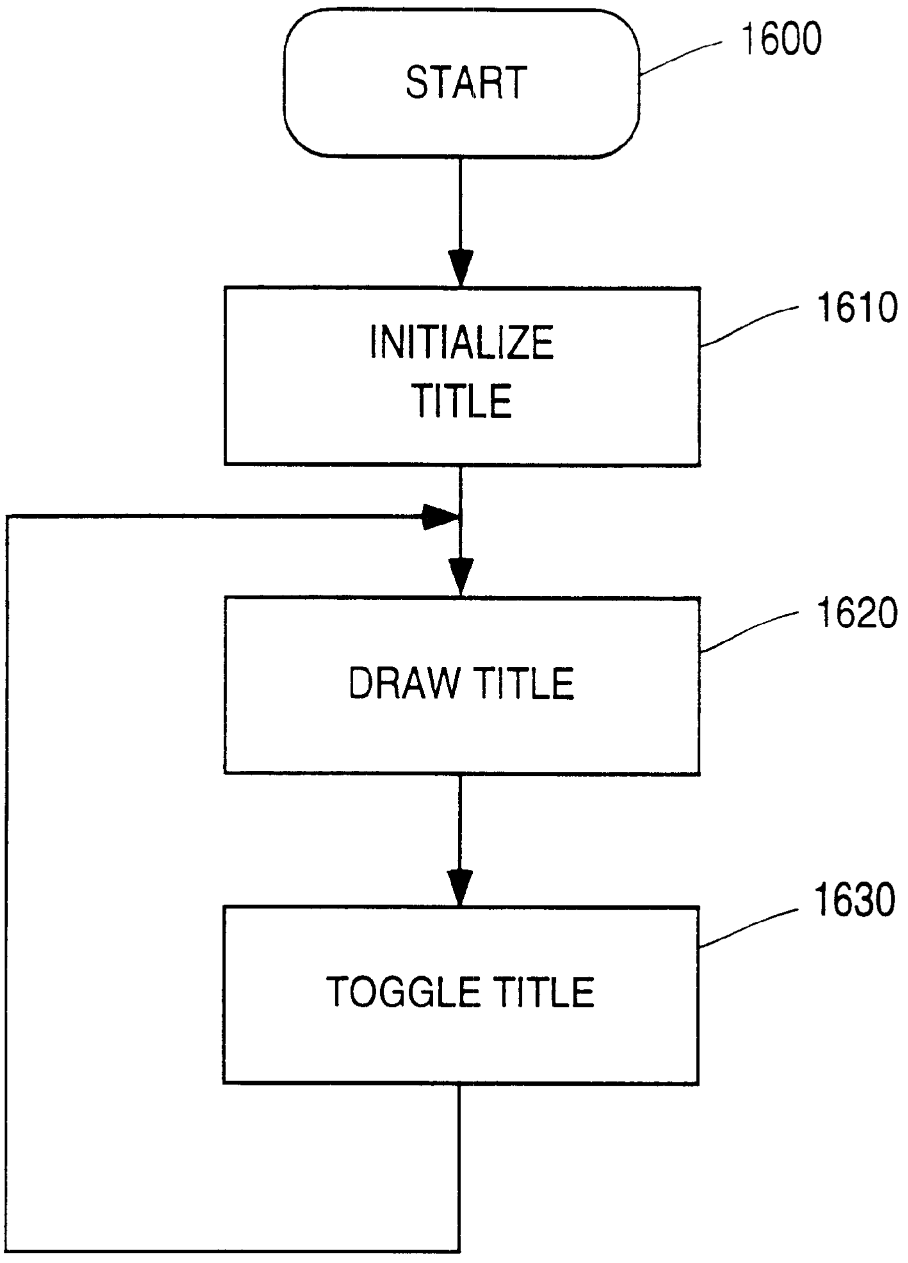

START
1600
INITIALIZE TITLE
1610
DRAW TITLE
1620
TOGGLE TITLE
1630

FIG. 16

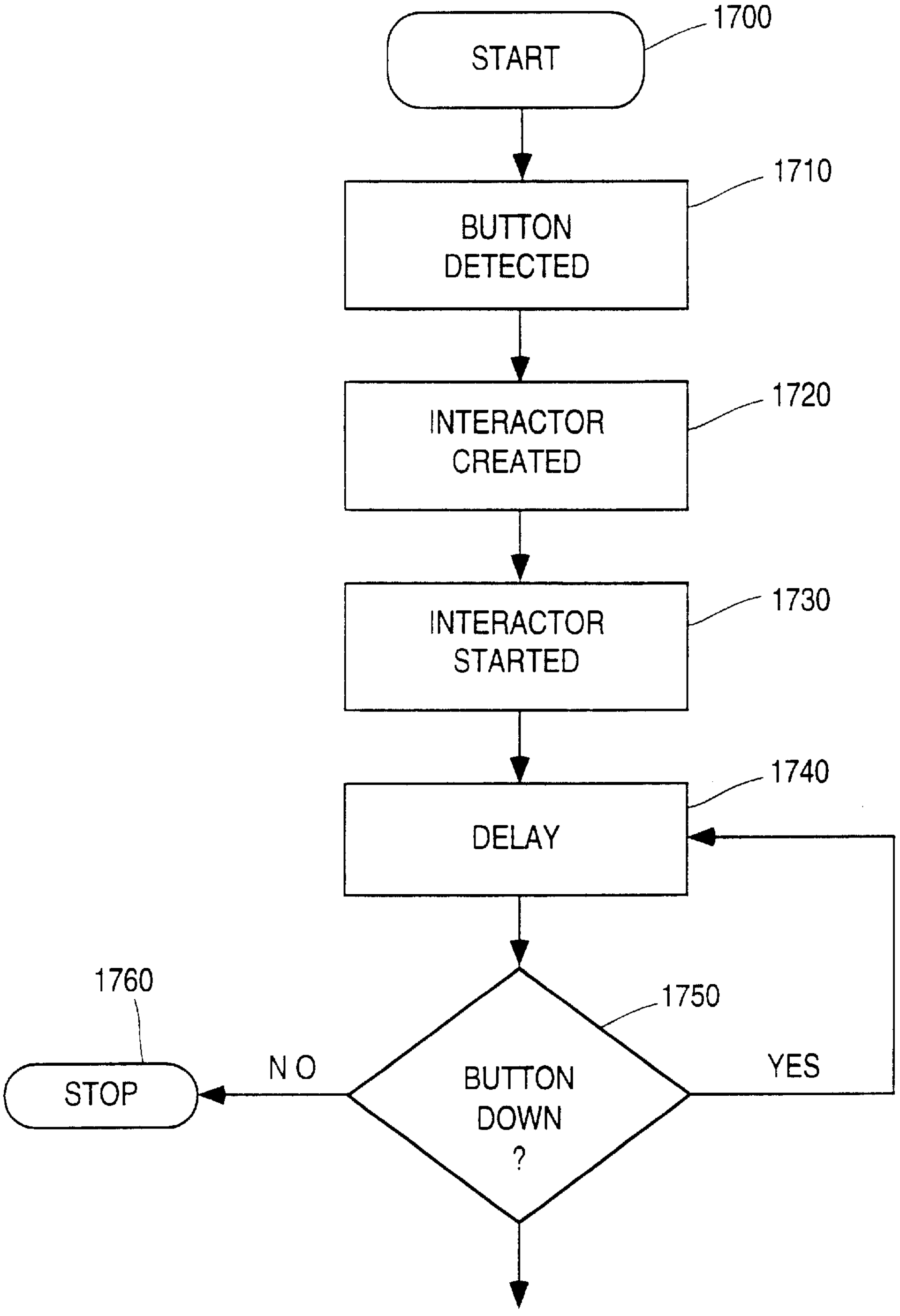

START
1700
BUTTON DETECTED
1710
INTERACTOR CREATED
1720
INTERACTOR STARTED
1730
DELAY
1740
BUTTON DOWN ?
1750
YES
NO
STOP
1760

FIG. 17

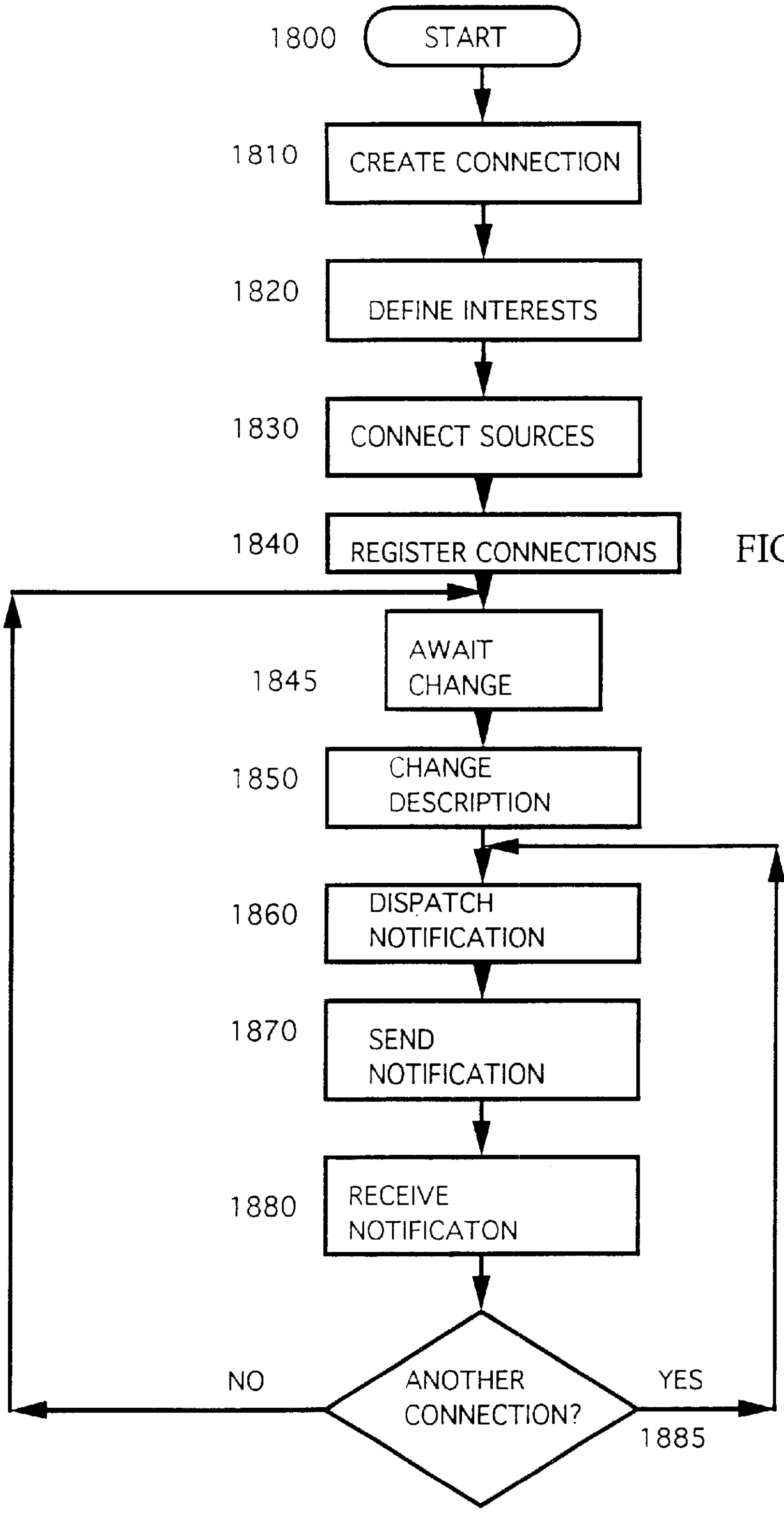

1800
START
1810
CREATE CONNECTION
1820
DEFINE INTERESTS
1830
CONNECT SOURCES
1840
REGISTER CONNECTIONS
1845
AWAIT CHANGE
1850
CHANGE DESCRIPTION
1860
DISPATCH NOTIFICATION
1870
SEND NOTIFICATION
1880
RECEIVE NOTIFICATON
NO
ANOTHER CONNECTION?
YES
1885

FIGURE 18

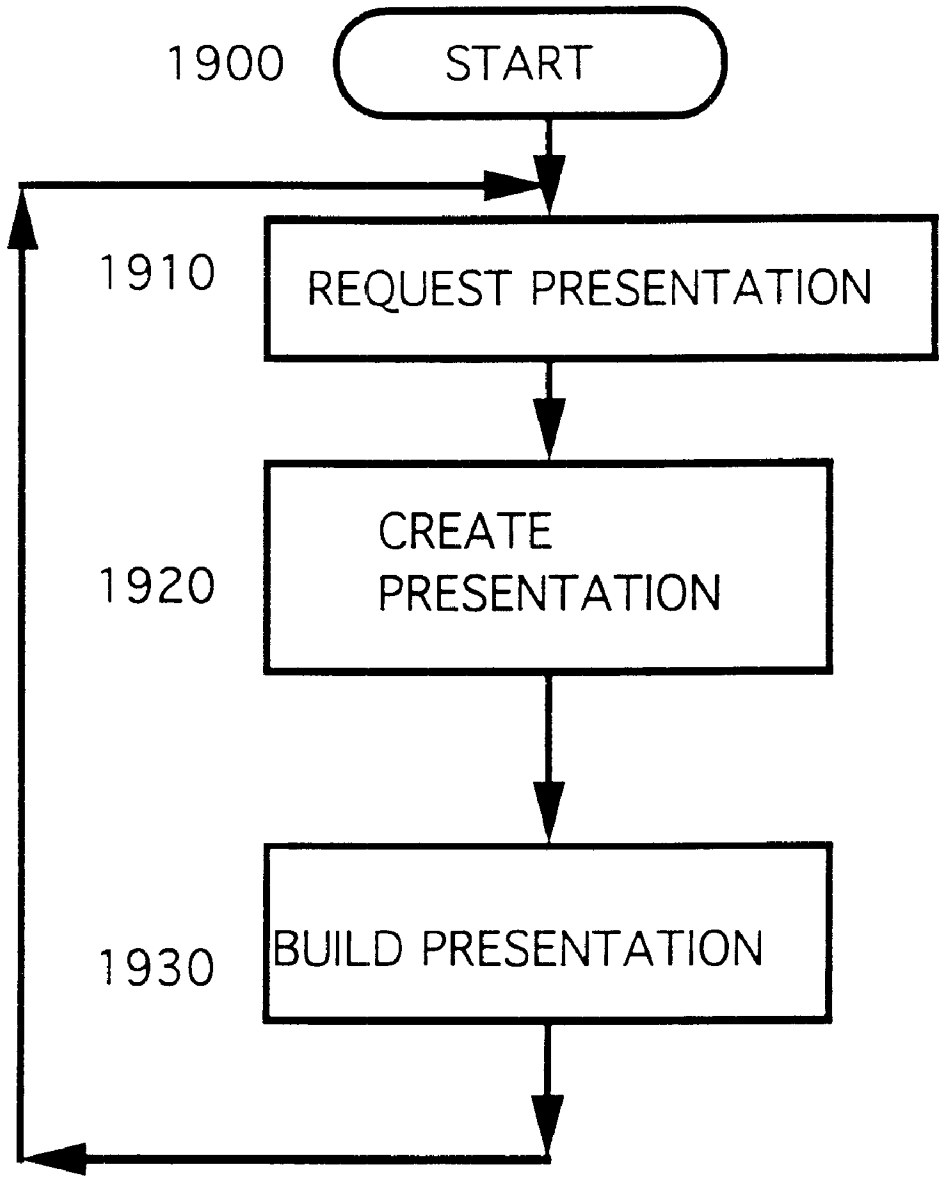

1900
START
1910
REQUEST PRESENTATION
1920
CREATE PRESENTATION
1930
BUILD PRESENTATION

FIGURE 19

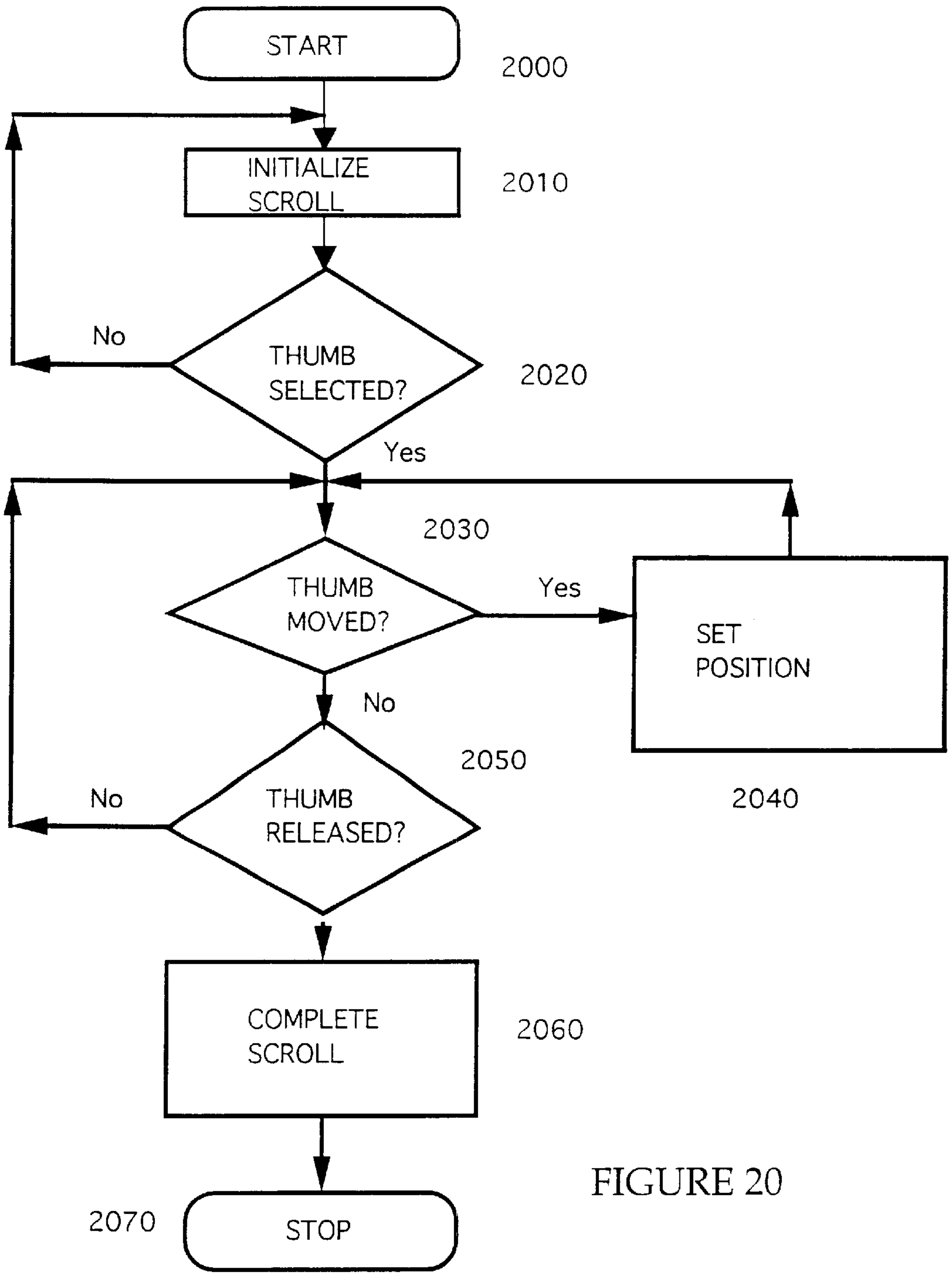

START
2000
INITIALIZE SCROLL
2010
THUMB SELECTED?
2020
No
Yes
2030
THUMB MOVED?
Yes
SET POSITION
2040
No
2050
THUMB RELEASED?
No
COMPLETE SCROLL
2060
2070
STOP

FIGURE 20

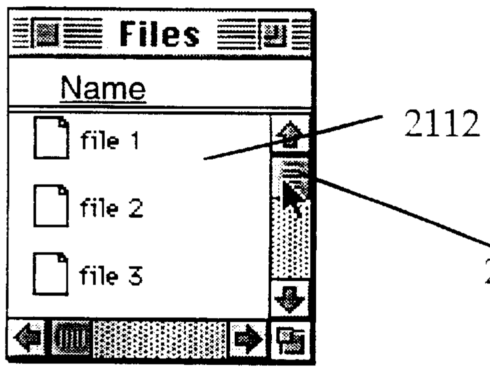


FIGURE 21A

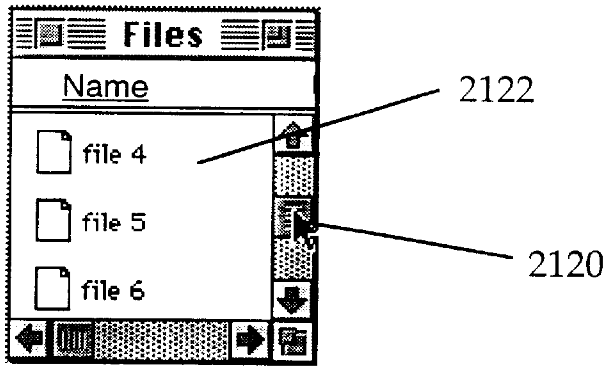


FIGURE 21B

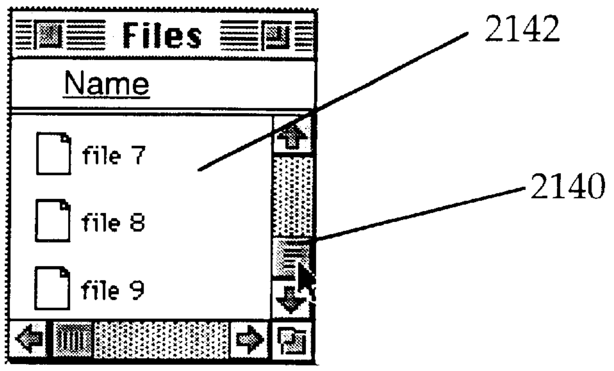


FIGURE 21C

OBJECT-ORIENTED EVENT NOTIFICATION SYSTEM WITH LISTENER REGISTRATION OF BOTH INTERESTS AND METHODS

CROSS-REFERENCES TO RELATED APPLICATIONS

This is a 37 C.F.R. §1.53(b)continuation of U.S. patent application Ser. No. 07/996,775 filed on Dec. 23, 1992, now U.S. Pat. No. 6,259,446.

FIELD OF THE INVENTION

This invention generally relates to improvements in display systems and more particularly to a globally scalable method for notification of change events arising in an object-oriented environment such as an automated menu state processing by integrating menu processing operations into the operating system.

BACKGROUND OF THE INVENTION

Among developers of workstation software, it is increasingly important to provide a flexible software environment while maintaining consistency in the user's interface. An early attempt at providing this type of an operating environment is disclosed in U.S. Pat. No. 4,686,522 to Hernandez et al. This patent discusses a combined graphic and text processing system in which a user can invoke a dynamic menu at the location of the cursor and invoke any of a variety of functions from the menu. This type of natural interaction with a user improves the user interface and makes the application much more intuitive.

Menu selection should also reflect a consistent interface with the user regardless of what application is currently active. None of the prior art references applicant is aware of provides the innovative hardware and software system features which enable all application menus to function in a consistent manner.

SUMMARY OF THE INVENTION

Accordingly, it is a primary objective of the present invention to provide a scalable method for notification of change events arising in an object-oriented environment such as an automated menu-based system containing size, state, status and location information. For example, a preferred embodiment of a menu contains a list of menu items containing a command and variables that reflect the command's current appearance. This includes status information determinative of the menu item's state (enabled/disabled), its name, its associated graphic, and whether its appearance is currently valid. Each of these are initialized when the menu item was created. The exemplary embodiment creates a menu item from a command, where a menu item is another object data structure containing a command sequence. The menu item is added to a list of menu items, and initialized as an invalid appearance. Later when the menu item is selected from a pull down menu, the appearance state is recomputed and validated based on the system state and its status information.

Next, the invention queries a command object for notification. In an exemplary embodiment, each command object has four methods to connect for different types of notifications:

i) notifications that affect it's name,
ii) notifications that affect is graphic,
iii) notifications that affect whether it's active, and
iv) notifications that affect any data it provides.

In this exemplary embodiment, the menu item just created for the command connects for active notification. It does this by passing a connection object to the event notification system. The command is then responsible for connecting the connection object to notifiers affecting whether the command is active.

Then, the exemplary menu system queries the command for the enabled state before presenting the menu item on the display. This processing is accomplished by examining the current system state to ascertain if the function is active in the current context. Then, the internal state of the menu item is updated and the menu item is displayed based on the appropriate appearance state (grayed out or normal).

When a user invokes a command from a menu item, a control or though the direct manipulation of an object, a document state is modified and notification of the event is sent to the system. This event automatically informs any active menu items and assures current status information is consistent across the operating environment. The notification message includes the name of the change and a pointer to the object that sent the notification message.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1A** is a block diagram of a personal computer system in accordance with the subject invention;

FIG. **1B** is a display in accordance with the subject invention;

FIG. **2** illustrates the tools used to create an application in accordance with the subject invention;

FIG. **3** is a flow diagram of a command process in accordance with the subject invention;

FIG. **4** is a checkbox control in accordance with the subject invention;

FIG. **5** is a checkbox control activation in accordance with the subject invention;

FIG. **6** is a checkbox update in accordance with the subject invention;

FIG. **7** is a summary of checkbox control processing in accordance with the subject invention;

FIG. **8** is an illustration of a control panel in accordance with the subject invention;

FIG. **9** is an illustration of a dialog box in accordance with the subject invention;

FIG. **10** is an illustration of a dialog box color controller in accordance with the subject invention;

FIG. **11** is an illustration of a radio button in accordance with the subject invention;

FIG. **12** is a detailed flowchart of menu state processing in accordance with the subject invention;

FIG. **13** is a picture of a display in accordance with the subject invention;

FIG. **14** illustrates the detailed logic of atomic execution in accordance with the subject invention;

FIG. **15** sets forth the detailed logic associated with smart label processing in accordance with the subject invention;

FIG. **16** presents the detailed logic of smart window label processing in accordance with the subject invention;

FIG. **17** illustrates how objects are created and how the objects communicate with each other during a typical interaction with an object that can be moved and selected in accordance with the subject invention;

FIG. **18** is an object generating notification flowchart for a notification source object in accordance with the subject invention;

FIG. **19** presents a flowchart illustrating the detailed logic associated with selecting the proper user interface element in accordance with the subject invention;

FIG. **20** is a flowchart illustrating the detailed logic associated with scrolling in accordance with the subject invention; and

FIGS. **21**A, **21**B and **21**C illustrate window scrolling in accordance with the subject invention.

DETAILED DESCRIPTION OF THE INVENTION

The invention is preferably practiced in the context of an operating system resident on a personal computer such as the IBM® PS/2® or Apple® Macintosh® computer. A representative hardware environment is depicted in FIG. **1**A, which illustrates a typical hardware configuration of a workstation in accordance with the subject invention having a central processing unit **10**, such as a conventional microprocessor, and a number of other units interconnected via a system bus **12**. The workstation shown in FIG. **1**A includes a Random Access Memory (RAM) **14**, Read Only Memory (ROM) **16**, an I/O adapter **18** for connecting peripheral devices such as disk units **20** to the bus, a user interface adapter **22** for connecting a keyboard **24**, a mouse **26**, a speaker **28**, a microphone **32**, and/or other user interface devices such as a touch screen device (not shown) to the bus, a communication adapter **34** for connecting the workstation to a data processing network and a display adapter **36** for connecting the bus to a display device **38**. The workstation has resident thereon an operating system such as the IBM OS/2® operating system or the Apple System/7® operating system.

The subject invention is a new object-oriented system software platform comprised of an operating system and development environment designed to revolutionize personal computing for end-users, developers, and system vendors. The system is a complete, standalone, native operating system and development environment architected from the ground up for high-performance personal computing. The invention is a fully object-oriented system including a wealth of frameworks, class libraries, and a new generation object programming environment, intended to improve fundamentally the economics of third party application software development. The subject invention is a fully portable operating system.

Traditional operating systems provide a set of services which software developers can use to create their software. Their programs are very loosely integrated into the overall operating system environment. For example, DOS applications take over the entire machine. This means that as far as the user is concerned, the application is the operating system. In Macintosh® and Windows operating systems, applications feel and look similar and they typically support cutting and pasting between applications. This commonalty makes it easier for users to use multiple applications in a single environment. However, because the commonalty is not factored into a set of services and frameworks, it is still very difficult to develop software.

In the subject invention, writing an "application" means creating a set of objects that integrate into the operating system environment. Software developers rely on the operating system for both a sophisticated set of services and a framework to develop software. The frameworks in the subject invention provide powerful abstractions which allow software developers to concentrate on their problem rather than on building up infrastructure. Furthermore, the fundamental abstractions for the software developer are very close to the fundamental concepts that a user must understand to operate her software. This architecture results in easier development of sophisticated applications.

This section describes four steps to writing software employing the subject invention. A user contemplating the development of an application is typically concerned with the following questions:

What am I modeling?

For a word processor, this is the text I am entering; for a spreadsheet, it is the values and formulas in the cells.

How is the data presented?

Again, for a word processor, the characters are typically displayed in a what-you-see-is-what-you-get (wysiwyg) format on the screen with appropriate line and page breaks; in a spreadsheet it is displayed as a table or a graph; and in a structured graphics program (e.g. MacDraw), it is displayed as a set of graphics objects.

What can be selected?

In a word processing application, a selection is typically a range of characters; in a structured graphics program it is a set of graphic objects.

What are the commands that can operate on this selection?

A command in a word processor might be to change the style of a set of characters to bold. A command in a structured graphic program might be to rotate a graphic object. FIG. **1**B is an illustration of a display in accordance with the subject invention. A command is illustrated at **41** for bringing a picture to the front of a display. A presentation of graphic information is illustrated at **40**. Finally, a selection of a particular graphic object, a circle, is shown at **42**.

A developer must answer the same four questions asked by the user. Fortunately, the subject invention provides frameworks and services for addressing each of these four questions. The first question that must be answered is: What am I modeling? In a word processing program, the data includes the characters that make up a document. The data in a spreadsheet includes the values and formulas in the cells. In a calendar program, the data includes the times and appointments associated with a given day. The invention provides facilities that help to model data. There are classes for modeling specific data types including: text, structured graphics, sound and video. In addition to these specific classes, the invention provides a number of other abstractions that support problem modeling, including: collection classes, concurrency control, recovery framework, and the C++ language. The class that encapsulates the data model for a particular data type provides a specific protocol for accessing and modifying the data contained in the data encapsulator, support for overriding a generic protocol for embedding other data encapsulators and for being embedded in other data encapsulators, generating notification to all registered objects when the data changes, and overriding a generic protocol for creating presentations of the data.

The next question that must be answered is: how is the data presented? In a structured graphic program, the set of graphic objects are typically rendered on a canvas. In a spreadsheet, it is typically a table of cells or a graph; and in a presentation program it is a set of slides or an outline. The subject invention provides a "view" of the data contained in a data encapsulator. The view is created using a "view system" and graphic system calls. However, playing a sound or video clip is also considered a presentation of the data.

Next: what can be selected? In a word processing program, a selection is a range of characters; in a structured graphics program, it is a set of graphics objects; and in a spreadsheet it is a range of cells. The invention provides selection classes for all of the fundamental data types that the system supports. The abstract baseclass that represents a selection made by a user provides an address space independent specification of the data selected. For text, this would be a numeric range of characters rather than a pair of pointers to the characters. This distinction is important because selections are exchanged between other machines when collaborating (in real-time) with other users. The baseclass also overrides a generic protocol for creating a persistent selection corresponding to this selection. Persistent selections are subclasses of an anchor object and may be heavier weight than their corresponding ephemeral selections because persistent selections must survive editing changes. For example, a persistent text selection must adjust itself when text is inserted before or after it. Anchors are used in the implementation of hypermedia linking, dataflow linking and annotations.

The baseclass also provides an override generic protocol for absorbing, embedding and exporting data contained in a data encapsulator. Baseclasses are independent of the user interface technique used to create them. Selections are typically created via direct manipulation by a user (e.g. tracking out a range of text or cells) but can be created via a script or as a result of a command. This orthogonality with the user interface is very important. Baseclasses also provide specific protocol for accessing the data encapsulator. There is a very strong relationship between a particular subclass of the encapsulator class and its subclass of a model selection class.

Finally: what are the commands that can operate on this selection? In a word processing program, a command might change the style of a selected range of characters and in a structured graphics program, a command might rotate a graphic object. The subject invention provides a large number of built-in command objects for all of the built-in data types as well as providing generic commands for Cut, Copy, Paste, Starting HyperMedia Links, Completing Links, Navigating Links, Pushing Data on Links, Pulling Data on Links, as well as many user interface commands. The abstract baseclass that represents a command made by the user is responsible for capturing the semantics of a user action, determining if the command can be done, undone, and redone. Command objects are responsible for encapsulating all of the information necessary to undo a command after a command is done. Before a command is done, command objects are very compact representations of a user action. The baseclass is independent of the user interface technique used to create them. Commands are typically created from menus or via direct manipulation by the user (e.g. moving a graphic object) but could be created via a script. This orthogonality with the user interface is very important.

Benefits Of Frameworks

The benefits of plugging into the abstractions in the invention are greater than providing a conceptual model. Plugging into the framework provides many sophisticated features architected into the base operating system. This means that the framework implements major user features by calling relatively small methods. The result is that an investment in coding for the framework is leveraged over several features.

Multiple Data Types

Once a new kind of data is implemented, the new data type becomes a part of the system. Existing software that can handle data encapsulators can handle your new data type

without modification. This differs from current computer systems, such as the Macintosh computer system. For example, a scrapbook desk accessory can store any kind of data, but it can only display data that has a text or quickdraw picture component. In contrast, the subject invention's scrapbook displays any kind of data, because it deals with the data in the form of an object. Any new data type that is created behaves exactly like the system-provided data types. In addition, the data in the scrapbook is editable since an object provides standard protocol for editing data.

The scrapbook example highlights the advantages of data encapsulators. If software is developed such that it can handle data encapsulators, an application can be designed to simply handle a new data type. A new application can display and edit the new kind of data without modification.

Multi-level Undo

The invention is designed to support multi-level undo. Implementing this feature, however, requires no extra effort on the part of a developer. The system simply remembers all the command objects that are created. As long as the corresponding command object exist, a user can undo a particular change to the data. Because the system takes care of saving the commands and deciding which command to undo or redo, a user does not implement an undo procedure.

Document Saving, Reliability, and Versioning

A portion of the data encapsulator protocol deals with filing the data into a stream and recreating the data at another place and/or time. The system uses this protocol to implement document saving. By default, a user's data objects are streamed to a file when saved. When the document is opened, the data objects are recreated. The system uses a data management framework to ensure the data written to disk is in a consistent state. Users tend to save a file often so that their data will be preserved on disk if the system crashes. The subject invention does not require this type of saving, because the system keeps all the command objects. The state of the document can be reconstructed by starting from the last disk version of the document and replaying the command objects since that point in time. For reliability, the system automatically logs command objects to the disk as they occur, so that if the system crashes the user would not lose more than the last command.

The invention also supports document versioning. A user can create a draft from the current state of a document. A draft is an immutable "snapshot" of the document at a particular point in time. (One reason to create a draft is to circulate it to other users for comments.) The system automatically takes care of the details involved with creating a new draft.

Collaboration

As mentioned above, a document can be reconstructed by starting with its state at some past time and applying the sequence of command objects performed since that time. This feature allows users to recover their work in the case of a crash, and it can also be used to support real-time collaboration. Command objects operate on selections, which are address-space independent. Therefore, a selection object can be sent to a collaborator over the network and used on a remote machine. The same is true of command objects. A command performed by one collaborator can be sent to the others and performed on their machines as well. If the collaborators start with identical copies of the data, then their copies will be remain "in sync" as they make changes.

Creating a selection is done using a command object, so that all collaborators have the same current selection.

The system uses a feature known as "model based tracking" to perform mouse tracking on each collaborator's machine. The tracker object created to handle a mouse press creates and performs a series of incremental commands as a user moves the mouse. These commands are sent to collaborators and performed by each collaborator. The result is that each collaborator sees the tracking feedback as it occurs. The system also establishes a collaboration policy. A collaboration policy decides whether users are forced to take turns when changing data or can make changes freely. The invention handles the mechanics of collaboration which removes the responsibility from an application developer.

Scripting

Designing a system to manage the sequence of command objects also makes it possible to implement a systemwide scripting facility. The sequence of command objects is equivalent to a script of the local actions. The scripting feature simply keeps track of command objects applied to any document. The scripting facility also uses selection objects in scripts. This feature provides customization of a script by changing the selection to which the script applies. Since command objects include a protocol for indicating whether they can apply to a particular selection, the system ensures that a user's script changes are valid.

Hypermedia Linking

Persistent selections, also known as anchors, can be connected by link objects. A link object contains references to the two anchors that form its endpoints. To the system, the link is bidirectional; both ends have equal capabilities. Certain higher-level uses of links may impose a direction on the link. The single link object supports two standard features: navigation and data flow. A user can navigate from one end of the link to the other. Normally, this will involve opening the document containing the destination anchor and highlighting the persistent selection. The exact behavior is determined by the anchor object at the destination end. For example, a link to an animation may play the animation. A link to a database query may perform the query.

Links also facilitate data flow. The selected data at one end of the link can be transferred to the other end to replace the selection there. In most cases, the effect is the same as if the user copied the selection at one end, used the link to navigate to the other end, and pasted the data. The system takes care of the details involved with navigating from one end of a link to the other (e.g., locating the destination document, opening it, scrolling the destination anchor into view, etc.). Similarly, the system handles the details of transferring data across the link. The latter is done using the selection's protocol for accessing and modifying the data to which it refers.

Annotations

The invention supports a system-wide annotation facility. This facility allows an author to distribute a document draft for review. Reviewers can attach posted notes to the document, and when done, return the document to the author. The author can then examine the posted notes and take action on each. (An author can also create posted notes in the document.) A reviewer need not have the same software as the author. Instead, the reviewer can use a standard annotation application. This application reads the data from the author's draft, and creates an annotatable presentation of the data. (Creating such a presentation is part of the standard data encapsulator protocol.)

The reviewer can create selections in the document, and link posted notes to the selection. The link between the posted note and selection allows the system to position the posted note "near" the selection to which it refers. The links also make the annotation structure explicit, so that the system can implement standard commands to manipulate annotations. The contents of the posted note can be any data type implemented in the system, not simply text or graphics. The contents of a note is implemented using a data encapsulator, and opening a note results in creating an editable presentation on that data.

Data Representation

Data representation is concerned with answering the question of what is the data that I am modeling? The subject invention provides facilities that help to model data. There are classes for modeling specific data types, including: text, structured graphics, sound and video. In addition to these specific classes, the invention provides a number of other abstractions that help to model a problem: the collection classes, the concurrency control and recovery framework, and the C++ language itself. In the subject invention, the class that encapsulates the data model for a particular data type is a subclass of the encapsulator class.

The Encapsulator Class

A developer creates a container for a particular type of data representation by creating a derived class of the encapsulator class. For each type of data in the system, (e.g. graphic objects, styled text, spreadsheet cells) a different derived class must exist which acts as the container for a type's data. Each class of encapsulator provides a type specific protocol for accessing and modifying the data contained therein. This protocol is typically used by presentations for displaying the data and by commands for modifying the data. In addition to type specific protocol, the encapsulator class provides generic protocol that supports the embedding of data encapsulators as "black-boxes" into other alien types. This protocol must be implemented in the derived class to support the creation of presentations, editors and selections for the encapsulated data. A container need only understand this generic protocol to support the embedding of any alien data type.

Choosing A Representation For Data

The data type designer has both the C++ object model, and a rich set of standard classes to choose from when designing a representation for a particular type of data. The classes provided by the invention should always be considered before designing unique classes to represent the data. This minimizes any duplication of effort which may occur by creating new classes which provide similar or identical function to classes already existing in the system. The most basic of these is the C++ object model. A designer can create a class or classes which closely match the mental model of the user to represent the classes the user deals with.

The invention's foundation classes provide many standard ways to represent data. Collection classes provide a number of ways for collecting together related objects in memory, ranging from simple sets to dictionaries. Disk-based collections, providing persistent, uncorrupted collections of objects, are also available. A data type requiring two (2D) and three dimensional (3D) graphic modeling, such as a graphical editor, is also supported. Numerous 2D and 3D modeling objects are provided along with transformation, matrix classes and 3D cameras. Similarly, the invention provides a sophisticated text data type that supports full international text, aesthetic typography, and an extensible style mechanism. The invention also provides support for time based media such as sound and video. Sophisticated time control mechanisms are available to provide synchronization between various types of time based media.

Presentation Protocol

The encapsulator class provides a protocol for the creation of various classes of presentations on the data contained within the encapsulator. The presentations include a thumbnail presentation, a browse-only presentation, a selectable presentation, and an editable presentation. There is also a protocol for negotiating sizes for the presentations and fitting the data into the chosen size. Subclasses of the encapsulator class are responsible for overriding and implementing this protocol to support the embedding of the data in other encapsulators. The presentations currently supported include:

Thumbnail—This presentation is intended to give the user a "peek" at what is contained in the encapsulator. It is typically small in size and may scale-down and/or clip the data to fit the size.

Browse-only—This presentation allows the user to view the data in its normal size but the user is unable to select or modify any of the data.

Selectable—This presentation adds the ability to select data to the capabilities provided by the browse-only presentation. It is used in annotating to allow annotations to be tied to selections in the data without allowing modification to the data itself. The selectable presentation is typically implemented as a subclass of the browse-only presentation.

Editable—This presentation adds the ability to modify data to the capabilities provided by the selectable presentation. This is the presentation that allows the user to create new data and edit existing data. Currently, this presentation provides its own window for editing. It is likely that in the future support will be added for presentations which allow editing in place. The editable presentation is typically implemented as a subclass of the selectable presentation.

Change Notification

When the data contained in an encapsulator class is changed, it is necessary to provide clients (e.g. a view on the data) with notification of the change. Encapsulators rely on a built-in class for standard notification support to allow the encapsulator to notify clients of changes to the data representation. A client can connect to an encapsulator for notification on specific changes or for all changes. When a change occurs the encapsulator asks the model to propagate notification about the change to all interested clients.

Data Presentation

This section addresses how the system presents data to a user. Once the data has been represented to the system, it is the role of the user interface to present the data in an appropriate and meaningful way to a user. The user interface establishes a dialogue between the user and the model data. This dialogue permits a user to view or otherwise perceive data and gives a user the opportunity to modify or manipulate data. This section focuses on data presentation.

The User Interface

A developer creates a class to facilitate the presentation of data to interact with a data encapsulator. By separating the

data model from the presentation, the invention facilitates multiple presentations of the same data. Some applications, like the Apple® Macintosh Finder, already support a limited form of multiple presentations of the same data. Sometimes it is useful to be able to display different views of the same data at the same time. These different views might be instances of the same class—as in a 3D CAD program which shows four different view of the same data. For each kind of presentation, a user was previously required to write a view which can display the model and a set of trackers and tracking commands which can select and modify the model.

Static Presentations

The simplest presentation type is the name of the data. The name is a text string that indicates the data content or type. Examples include "Chapter 4", "1990 Federal Income Taxes", "To Do". Another simple presentation type, an icon, is a small graphical representation of the data. It usually indicates the data type. Examples are a book, a report, a financial model, a sound or video recording, a drawing. However, they may also display status, such as a printer that is printing, or indicate content, such as a reduced view of a drawing. Finally, the thumbnail, is a small view of the model data. This view may show only a portion of the data in order to fit the available space. Examples are a shrunken drawing, a book's table of contents, a shrunken letter, or the shrunken first page of a long document. A browse-only presentation allows a user to view the data in its normal size but the user is unable to select or modify any of the data.

Selectable Presentations

Selectable presentations allow a user to view, explore, and extract information from the data. These presentations provide context: what the data is, where the data is, when the data was. It may help to present the data in a structured way, such as a list, a grid, as an outline, or spatially. It is also useful to display the relationships among the data elements, the data's relationship to its container or siblings, and any other dependencies.

Selectable presentations may also display meta data. An example is the current selection, which indicates the data elements a user is currently manipulating. Another type of meta data is a hypermedia link between data elements. The view may also indicate other users who are collaborating on the data.

Selectable presentations are usually very specific to the type of the data. They are made up of windows, views, and other user interface objects which may be customized to best reflect the data type. Some examples are:

Sound recording—A control panel would facilitate an audible presentation. Views would display the sound as a musical score or as a series of waveforms. Views may include a sample number or time indications.

Financial model.—The model could be viewed as the set of formulas and other parameters. It could display values from the model at a particular instance of time or with specific input values as a spreadsheet or in various graphical forms.

Book.—The model could be viewed as a table of contents, an index, a list of illustrations. It could be viewed as a series of pages, a series of chapters, or a continuous text flow.

Video recording—The model could be viewed as a series of individual frames or as a continuous presentation. Views may include track marks, frame number, and time indications.

Container containing other objects—The objects could be displayed alphabetically by name, by type or other attribute, as a set of icons, as a set of thumbnails.

Editable Presentations

Editable presentations are similar to interactive presentations except that they also facilitate data modification. They do this by allowing direct manipulation of the data with the mouse or other pointer. They also allow the data to be manipulated symbolically through menu items and other controls.

Data Access

Presentations interact with data encapsulators in order to determine the data and other information to present. Presentations query the model for the data that is required. The presentation may present all or only part of the data that is contained or can be derived from the data in the data encapsulator.

Change Notification

Because there can be many presentations of a single model active at once, the data can be changed from many sources, including collaborators. Each presentation is responsible for keeping itself up to date with respect to the model data. This is accomplished by registering for notification when all or a portion of a model changes. When a change occurs to data in which the presentation is interested, the presentation receives notification and updates its view accordingly. Change notification can be generated in any of the ways listed below. First, change notification can be generated from the method in the data encapsulator which actually changes the model data. Second, change notification can be generated from the command which caused the change. As mentioned earlier, there are benefits to these two approaches. Generating the notification from within the data encapsulator guarantees that clients will be notified whenever the data changes. Generating the notification from the command allows "higher-level" notification, and reduces the flurry of notifications produced by a complicated change.

Notification Framework Overview

The Notification framework provides a mechanism for propagating change information between objects. The framework allows objects to express interest in, and receive notification about changes in objects on which they depend. A standard interface is provided for classes that provide notification to clients. Notifier classes provide notification source objects with the means to manage lists of clients and dispatch notifications to those clients. Notifier objects require no special knowledge of the class of objects receiving notifications. Connection objects provide the dispatch of notifications from the notifier to specific notification receiver objects. These connection objects allow specialization of how notifications are delivered to different classes of receivers. Finally, Notification objects transport descriptive information about a change, and interests describe a specific notification from a notification source object.

Notification Propagation Flow Chart

FIG. **18** is an object generating notification flowchart for a notification source object. Processing commences at terminal **1800** and immediately passes to function block **1810** where a notification receiver object creates a connection to itself. Then, at function block **1820** the notification receiver object adds appropriate interests for one or more notifications from one or more notification source objects. These interests are defined by the notification source object(s).

The client object asks the connection object to connect to the notification source(s) for notifications specified by the interests in the connection in function block **1830**. Then, in function block **1840**, for each interest in connection, the connection is registered as interested in the notification with the notifier in the interest. Next, at function block **1845**, the system enters a wait state until a change is detected. When a system change occurs, control immediately passes to **1850** where a notification source object changes and calls notify on its notifier with a notification describing the change.

For each connection registered with the notifier as interested in the notification, at function block **1860**, the connection is asked to dispatch the notification. In turn, at function block **1870**, the connection dispatches the notification to the appropriate method of the notification receiver. Finally, at function block **1880**, the notification receiver takes the appropriate action for the notification, and a test is performed at decision block **1885** to determine if another connection is registered with the notifier as interested in notification. If there is another connection, then control passes to **1850**. If there is not another connection to service, then control passes to function block **1845** to await the next change.

Data Specification

Data specification addresses the selection issue of data processing. If a user must manipulate data contained in a representation, the data must be able to specify subsets of that data. The user typically calls this specification a "selection," and the system provides a base class from which all selection classes descend. The invention also provides selection classes for all of the fundamental data types that the system supports.

Model Selection

The object which contains the specification of a subset of data in a representation is a model selection class. In the case of a text representation, one possible selection specification is a pair of character offsets. In a structured graphics model, each shape must be assigned a unique id, and the selection specification is a set of unique ids. Neither of the specifications point directly at the selection data and they can be applied across multiple copies of the data.

Accessing Specified Data

A selection understands the representation protocol for accessing and modifying data and knows how to find data in a local address space. Command objects access a representation's data through data selection, and therefore require no knowledge of converting from specification to the real data in the local model. It is the job of the selection object to provide access to the real data from the address space independent specification. In a text encapsulator, this processing may require querying the encapsulator for the actual characters contained in a range. In a base model such as a graphical editor the selection will typically hold surrogates for the real objects. The encapsulator must provide a lookup facility for converting the surrogate to the real object.

Standard Editing Protocol

The model selection class provides a protocol for the exchange of data between selections. By implementing the protocol for type negotiation, absorbing, embedding and exporting data, derived classes provide support for most of the standard editing commands. This means that the editing commands (Cut, Copy, Paste, Push Data, etc.) provided by the system will function for the represented data type and will not require reimplementation for each application. The model selection class also provides support directly for the exchange of anchors and links but relies on the derived class's implementation of several key methods to support the exchange of the representation's data:

CopyData must be implemented by the derived class to export a copy of the specified data. The implementation creates and returns a new data encapsulator of the requested type containing a copy of the specified data.

AdoptData must be implemented by the derived class to support absorbing or embedding data into the specification's associated representation. If the data is to be absorbed it must be of a type which can be incorporated directly into the receiver's representation. The absorbed data is added to the representation as defined by the specification. It is common for many data types to replace the currently specified data with the newly absorbed data. Any replaced data is returned in a data encapsulator to support Undo. If the data is to be embedded, the encapsulator is incorporated as a black box and added as a child of the representation.

ClearData must be implemented by the derived class to delete the specified data from the associated representation. An encapsulator of the representation's native type containing the deleted data must be returned.

User Interface

The user interface for creating specifications is typically the responsibility of a presentation on the data. A number of mechanism are available depending on data type and presentation style. The most favored user interface for creating a selection is direct manipulation. In a simple graphics model, objects may be selected by clicking directly on the object with the mouse or dragging a selection box across several objects using a mouse tracker. In text, a selection may be created by as the result of a find command. Another common way that selections are created is as a result of a menu command such as "find." After the command is issued, the document is scrolled to the appropriate place and the text that was searched for is selected.

Finally, selections can come from a script (or programmatically generated) and the result would be the same as if a user created the selection directly. "Naming" selections for scripts involve creating a language for describing the selection. For example, in text, a selection could be "the second word of the fourth paragraph on page two." The invention's architecture provides support for scripting.

Data Modification

Data Modifications addresses the question: what are the commands that can operate on this selection? If a user is to modify the data contained in a representation, the system must be able to specify exactly the type of modification to be made. For example, in a word processing program, a user may want to change the style of a selected range of characters. Or, in a structured graphics program, a user may desire rotation of a graphic object. All user actions that modify the data contained in a data encapsulator are represented by "command objects."

The Model Command Object

The abstract base class that represents a command made by the user is the model command object. Subclasses of the model command object capture the semantics of user actions, such as: can be done, undone, and redone. These subclasses are independent of the user interface technique used to create them. Unlike MacApp, as soon as the semantics of a user action is known, device events are translated into command objects by the system.

HandleDo, HandleUndo, and HandleRedo

Creating a new class of command involves overriding a number of methods. The most important three methods to override are: HandleDo, HandleUndo and HandleRedo. The HandleDo method is responsible for changing the data encapsulator appropriately based on the type of command that it is and the selection the command is applied to. For example, if the command involves a style change to a range of characters in a word processor, the HandleDo method would call a method (or set of methods) in the data encapsulator to specify a character range and style to change. A more difficult responsibility of the HandleDo method is saving all of the information necessary to "undo" this command later. In the style change example, saving undo information involves recording the old style of the character range. The undo information for most commands is very simple to save. However, some commands, like find and change may involve recording a great deal of information to undo the command at a later time. Finally, the HandleDo method is responsible for issuing change notification describing the changes it made to the data encapsulator.

The HandleUndo method is responsible for reverting a document back to the state it was in before the command was "done." The steps that must be applied are analogous to the steps that were done in the HandleDo method described above. The HandleRedo method is responsible for "redoing" the command after it had been done and undone. Users often toggle between two states of a document comparing a result of a command using the undo/redo combination. Typically, the HandleRedo method is very similar to the HandleDo method except that in the Redo method, the information that was derived the last time can be reused when this command is completed (the information doesn't need to be recalculated since it is guaranteed to be the same).

User Interface

Command objects capture the semantics of a user action. In fact, a command represents a "work request" that is most often created by a user (using a variety of user interface techniques) but could be created (and applied) in other ways as well. The important concept is that command objects represent the only means for modifying the data contained in a data encapsulator. All changes to the data encapsulator must be processed by a command object if the benefits of infinite undo, save-less model, and other features of the invention are to be realized.

The most favored user interface for issuing commands involves some sort of direct manipulation. An object responsible for translating device events into commands and "driving" the user feedback process is known as a tracker. The invention provides a rich set of "tracking commands" for manipulating the built-in data types. For example, there are tracking commands for rotating, scaling and moving all the 2D objects in Pink such as lines, curves, polygons, etc.

A common user interface for issuing commands is via controls or the menu system. Menus are created and a set of related commands are added to the menu. When the user chooses an item in the menu, the appropriate command is "cloned" and the Do method of the command is called. The programmer is never involved with device events at all. Furthermore, because commands know what types of selections they can be applied to, menu items are automatically dimmed when they are not appropriate.

Finally, commands can be issued from a script (or programmatically generated) and the result would be the same as if a user issued the command directly. The Pink architecture provides support for scripting; however, at this time, there is no user interface available for creating these scripts.

Built-in Commands

The invention provides a large number of built-in command objects for all of the built-in data types as well as providing generic commands for Cut, Copy, Paste, Starting HyperMedia Links, Completing Links, Navigating Links, Pushing Data on Links, Pulling Data on Links, as well as many user interface commands. One of the advantages of using the frameworks is that these built-in command objects can be used with any data encapsulators.

More Features

The previous sections of this document concentrated on the foundational features of the invention. There are many additional facilities in the invention that implement advanced features. Specifically, these facilities include: model-based tracking, filing, anchors, and collaboration.

Model Based Tracking

Tracking is the heart of a direct-manipulation user interface. Tracking allows users to select ranges of text, drag objects, resize objects, and sketch objects. The invention extends tracking to function across multiple views and multiple machines by actually modifying the model. The tracker issues commands to the model, which posts change notifications to all interested views.

Model based tracking is the best solution for tracking in documents, but it does have the drawbacks that: (1) the model's views must be optimized to provide quick response to change events and (2) the model must be capable of expressing the intermediate track states.

Anchors

Persistent selections or "anchors" are very similar to selections in that they are specifications of data in a representation. The difference is that anchors must survive editing changes since by definition anchors persist across changes to the data. The implementation of graphics selections described earlier in the document is persistent. The implementation of text selections, however, is not. If a user inserts or deletes text before a selection, then the character offsets must be adjusted. There are a couple of approaches for implementing text anchors. First, the text representation maintains a collection of markers that point within the text, similar to the way styles are maintained. The anchors include an unique id that refers to a marker. When the text is changed, the appropriate markers are updated, but the anchors remain the same. Another approach is to maintain an editing history for the text. The anchor could contain a pair of character positions, as well as a time stamp. Each time the text was edited, the history would be updated to record the change (e.g., 5 characters deleted from position X at time T). When the anchor is used, the system would have to correct its character positions based on editing changes that happened since the last time it was used. At convenient times, the history can be condensed and the anchors permanently updated.

The system provides a large number of features for "free" through the anchor facility. All of the HyperMedia commands (CreateLink, PushData, PullData, and Follow) all use anchors in their implementation. The implementation of the system wide annotation facility uses anchors in its implementation. The base data encapsulator provides services for keeping track of anchors and links. However, the user is responsible for making anchors visible to the user via presentations. The application must also issue the proper command object when a user selects an anchor. After a user interface for anchors and links is nailed down, the document framework provides additional support to simplify processing.

Filing

Filing is the process of saving and restoring data to and from permanent storage. All a user must do to make filing work is to implement the streaming operators for a data encapsulator. The invention's default filing is "image" based. When a user opens a document, the entire contents of the document are read into memory. When a user closes a document, the entire contents of the document are written back to disk. This approach was selected because it is simple, flexible, and easy to understand. To store data in a different format, perhaps for compatibility with a preexisting standard file format, two approaches are possible. First, an encapsulator class can stream a reference to the actual data, then use the reference to find the actual data, or a new subclass can be defined to create and return a file subclass.

The advantage of the first approach is a data encapsulator can be encapsulated in other documents. The advantage of the second approach is the complete freedom afforded to exactly match an existing file format for the complete document.

Collaboration

Same-time network collaboration means that two or more people edit the same document at the same time. The system also establishes the collaboration policy; that is, whether users are forced to take turns when changing the data or can make changes freely. A developer does not have to worry about the mechanics of collaboration or the collaboration policy.

Supporting Collaborator Selection Styles

To assist in the reduction of confusion and enhance model selection, the document architecture provides a collaborator class which contains information about the collaborator's initials and preferred highlight bundle.

Supporting Multiple Selections

To support multiple selections a user must modify presentation views because each collaborator has a selection. When the active collaborator's selection changes the standard change notification is sent. When a passive collaborator's selection changes a different notification event is sent. A view should register for both events. Since the action taken to respond to either event is usually the same, economy can be realized by registering the same handler method for both events.

User Interface In Accordance With The Invention

This portion of the invention is primarily focused on innovative aspects of the user interface building upon the foundation of the operating system framework previously discussed. The first aspect of the user interface is a mechanism allowing a user to manage interactions with various objects or data referred to as controls.

Control

The object with which users interact to manipulate other objects or data is called a control. Controls use a command to determine the current state of the object or data. Following appropriate interactions with the user, the control updates the command's parameters and causes it to be executed. Example controls are menus, buttons, check boxes and radio buttons.

Controls use a command to determine the current state of the object or data. Following appropriate interactions with the user, the control updates the command's parameters and causes it to be executed. For example, a checkbox sets a command parameter to on or off and then executes the command to change a data value.

Many controls display the current value of the data they manipulate. For example, a check box displays a check only when a Boolean data value is TRUE. As the data changes, the control's appearance is kept up to date using a notification system described here. The process is similar to the process used to enable/disable menu items.

When a control is created a command must be specified. The control makes a copy of this command and stores it in field fCommand. If the command supplies any data values, a pointer to appropriate Get and Set methods of the command must also be specified. The control stores these method pointers in fields fGetMethod and fSetMethod, respectively. Then, the control connects for notifications that indicate its data value may be out of date. Each command provides a method called ConnectData for this purpose.

Each control contains a connection object called fDataConnection indicating the object and method to receive the notification. This connection object passed as an argument to the command. The command object calls the connection object's Connect method to add each notifier and interest that may affect its data value. When complete, the control calls the connection object's Connect method to establish the connections as shown in FIG. **3**. The control updates its data value from its command. It does this by calling the Get method of the command (fCommand->(*fGetMethod)( )). The control stores this value in an appropriate field (e.g. a checkbox stores it in a Boolean field named fChecked) as depicted in FIG. **5**. Then, the control updates its appearance. It performs this action by calling the view system's invalidate method, indicating which portion of the screen needs updating.

Finally, the data changes and notification is sent. At some point, a command is executed which changes the value of the data being reflected by the control. This command could be executed from a control, menu item, or through direct manipulation. The control receives the notification as shown in FIG. **4**, and control is passed to await the next user selection.

Control Panel

One collection of controls is called a control panel. The controls in a control panel typically operate upon actual data (this is the default, not a requirement). Their actions are usually immediate and are independent from one another. Control panels manage the progression of the input focus among its controls as necessary. It is likely that control panels will be shared across all user interfaces in the system.

Dialog Box

Another collection of controls is called a dialog box. The controls in a dialog box typically operate upon prototypical data (this is the default, not a requirement). Their actions are usually collected together into a group and then performed together when the user presses an Apply button. Dialog boxes manage the progression of the input focus among its controls as necessary.

A Control in Action

We would now like to present a play in three acts to illustrate a control in action. FIG. **2** illustrates the various controls. A play example will be used by way of analogy to illustrate a control (in this case a checkbox), a command, a selection, and a data encapsulator.

Checkbox **200** The role of the checkbox is to display a Boolean value stored in the data encapsulator and to facilitate its change. The value is represented by the presence or absence of a check.

Command **210** The role of the command is to obtain the value from the data encapsulator and change it upon direction from the checkbox.

Selection **220** The role of the selection is to be an interface between the command and the data.

Data **230** Data is employed as a target for actions.

Getting to Know You

Everyone gets to know each other a little better as shown in FIG. **3**. The command **310** tells the checkbox **300** which notifications the data may send in which the control is certain to be interested (how the command **310** knows is none of anyone else's business). The checkbox **300**, in turn, connects to the data **320** for the notifications.

Unknown to anyone else, the director told the checkbox **300** the best way to interact with the command **310**. Specifically, it was told about the command's get value method and a set value method. The checkbox will take advantage of this a little bit later.

Reflecting the Data

Something happens to the data—it sends notifications as depicted in FIG. **4**. The checkbox **400** hears about those for which it has expressed an interest. In FIG. **4**, the notification from the data expresses to bold the information which is reflected by placing an X in the checkbox.

The checkbox **510** received notification from the data, and the processing to display the checkbox **510** correctly is depicted in FIG. **5**. It does this by using the command's **520** get value method it happens to know about. Before telling the checkbox **510** what the correct value is, the command **520** goes through the selection to the data to make sure it really knows the correct value. The checkbox **510** updates itself as necessary.

Changing the Data

The user now enters the scene and gives the checkbox **600** a nudge as shown in FIG. **6**. The checkbox **600** uses the command's **610** set value method to set the data's **620** value through the selection. The entire process is reviewed in FIG. **7**.

A Control Panel in Action

A control panel is nothing more than a simple window that contains a set of controls as shown in FIG. **8**. These controls contain a command that operates upon the current selection. The control is enabled if the command is active. Following appropriate interaction with the user, the control executes the command, causing the data to change.

A Sound Control Panel

As an example control panel, consider the sound controller illustrated in FIG. **8**. This control panel contains four buttons **800**, **802**, **804** and **806** for controlling sound playback. Each button performs as described in the "A Control in Action" section above.

Play **800** This control contains a TPlay command. This command is active only under certain conditions, making the control enabled only under those conditions. First, a sound must be selected in the appropriate data encapsulator. Next, it must not be playing already. Finally, the current sound position must be somewhere before the end. When pressed, the Play button executes the TPlay command, causing the selected sound to come out of the speaker.

Step **802** This control contains a TPlay command, too. How is this, you ask? Well, since I am making this up, we can pretend that the TPlay command takes a parameter indicating the duration it is to play. For the purposes of the step button, it is set to a single sample. The Step button is enabled only under the same conditions as described for the Play button. When pressed, the Step button executes the TPlay command, causing the selected sound to come out of the speaker.

Stop **804** This control contains a TStop command. The Stop button is enabled only if the selected sound is currently playing. When pressed, the Stop button executes the TStop command, causing the selected sound to stop playing and to set the current sound position to the beginning.

Pause **806** This control contains a TStop command, too. Unlike the Stop button, however, this TStop command is set to not rewind the sound to the beginning. Pressing the Play or Step buttons continue from where the playback left off.

A Dialog Box in Action

A dialog box is similar to a control panel, in that it is a simple window containing a set of controls. However, instead of the controls operating upon the selected data, they operate upon parameters of another command. Only until the Apply button is pressed is the real data modified.

A Color Editor

As an example dialog box, consider the color editor set forth in FIG. **9**. It contains three sliders, one for the red **900**, blue **910**, and green **920** components of the color. After adjusting the sliders to the desired values, the user presses Apply **930** to change the color of the selection.

Red **900**, Green **910**, Blue **920** To the user, these sliders are identical, except for their label. As with all controls, each slider contains a command that is executed following user interaction. Unlike many controls, especially those in a control panel that immediately affect the selected data, the command contained by these sliders displays and modifies the value of a parameter of another command. In this case, it is one of the red, green, or blue parameters of the command contained within the Apply button.

Apply **930** The Apply button contains a TSetColor command that changes the color of the selection when executed. It has three parameters, one for each of the red, green, and blue components of the color. These parameters are displayed and set by the sliders in response to user interaction. When the Apply button is pressed, this command is executed and the new color is set. The internal actions accompanying the color editor example, are depicted in FIG. **10**. The Red **1000**, Green **1010**, and Blue **1020** slides contain a TFloatControlCommand. These commands contain a single floating point value which the control displays. As the user adjusts the slider, it updates this value and executes the command.

The selection for the TFloatControlCommand specifies the TSetColor command within the Apply **1040** button. One of its parameters is set when each TFloatControlCommand is executed. Finally, when the user presses the Apply **1040** button, the TSetColor command is executed and the selected color **1050** is changed.

Classes

The following section describes the classes of the controls and dialog areas and their primary methods.

Control

A control is the user interface to one or more commands. The control displays information about a command, such as its name and whether it is active in the current context. Following appropriate user interaction, the control causes a command to be executed. When appropriate, the control obtains the current value of data the command modifies and displays it to the user. It may set a command parameter that indicates a new value of this data before executing the command.

Methods to create a selection on the control, with additional specification of a command within the control as an option. Lookup command is a pure virtual function in order to give subclasses flexibility in how many commands they contain and how they are stored.

Methods that are called when the presentation is opened and closed. When the presentation is opened the control connects for notifications that may affect its state. When the presentation is closed these connections are broken.

Methods that are called when the presentation is activated and deactivated. When the presentation is activated, some controls connect for notifications that are valid only when active. Deactivating the presentation breaks these connections.

Methods that control uses to connect to and disconnect from notifiers that affect whether the control is enabled. ConnectEnabledNotifiers connects to the notifiers specified by commands when the control is opened. DisconnectEnabledNotifiers breaks these connections when the control is closed.

Methods that receive notifications indicating that something happened affecting the control's presentation of a data value. This method does nothing by default.

Methods for notification. Create interest creates an interest specialized by the control instance. Notify is overloaded to send a notification and swallow the interest.

The Control Interest

A single notifier is shared among many subclasses of controls. In order to express interest in a particular control instance, the interest must be specialized. A control interest is an interest that contains a pointer to a specific control. This class is an internal class that is usually used as is, without subclassing.

The Control Notification

A single notifier is shared among many subclasses of controls. In order to distinguish which control sent the notification, the notification must be specialized. A control notification is a notification containing a pointer to the control that sent the notification. This class is usually used as-is, without subclassing.

The Control Presenter

A control presenter wraps up a control so it can be contained by a presentation data encapsulator. It implements standard behaviors that all presenter objects implement. This class is usually used as-is, without subclassing.

Methods that are called when the presentation is opened and closed. They do nothing by default. A subclass must implement these methods for the object it wraps. For controls, these methods are delegated directly to the control. When the presentation is opened, the control connects for notifications that may affect its state. When closed, the connections are broken.

Methods that are called when the presentation is activated and deactivated. They do nothing by default. A subclass must implement these methods for the object it wraps. For controls, these methods are delegated directly to the control. When the presentation is activated, some controls connect for notifications that are valid only when active. When deactivated, the connections are broken.

TControlSelection

A control selection specifies a single control, and optionally a command within it, that is wrapped in a control presenter and stored in a presentation.

Methods to access a command within the control. These may return an invalid value if no command was specified.

TUniControl

A unicontrol is the abstract base class for controls that present a single command and causes it to be executed following appropriate user interaction. Examples of this type of control are buttons and checkboxes.

Methods to specify the command that is presented and executed by the control. Notification is sent to registered connections when the command is changed.

Methods the control uses to connect to and disconnect from notifiers that affect whether the control is enabled. ConnectEnabledNotifiers connects to the notifiers specified by commands when the control is opened. DisconnectEnabledNotifiers breaks these connections when the control is closed.

Method that receives notifications indicating that something happened affecting whether the control should be enabled. UpdateEnabled checks whether the command is active and calls Enable and Disable as appropriate.

Methods that control uses to connect to and disconnect from notifiers that affect the control's presentation of a data value. ConnectDataNotifiers connects to the notifiers specified by commands when the control is opened. DisconnectDataNotifiers breaks these connections when the control is closed. Controls that do not display a data value (e.g. button) may override connect data notifiers to do nothing.

TButton

A button is a unicontrol that executes its command when pressed. This class is normally used without subclassing; just set the command and away you go.

Methods that are called when the presentation is activated and deactivated. When the presentation is activated, some controls connect for notifications that are valid only when active. When deactivated, these connections are broken. When the presentation is activated, buttons register for key equivalent notification. This connection is broken when the presentation is deactivated.

Methods that control users connecting to and disconnecting from notifiers that affect the control's presentation of a data value. Connect data notifiers connects to the notifiers specified by commands when the control is opened. Disconnect data notifiers breaks these connections when the control is closed. Controls that do not display a data value (e.g. button) may override connect data notifiers to do nothing.

The Checkbox

A checkbox is the user interface to a command that sets a Boolean value. Following appropriate user interaction, the checkbox calls a command method to change the value and executes the command. This class is normally used without subclassing; just set the command, its value getter and setter, and away you go.

The Slider

A slider is a unicontrol that displays a single floating point value and allows it to be changed following appropriate user interaction. Examples of sliders were presented in FIGS. **9** and **10**.

TMultiControl

A multicontrol is the abstract base class for controls that present several commands and causes them to be executed following appropriate user interaction. Examples of this type of control are radio buttons and menus.

TRadioButton

A radio button is a multicontrol that displays two or more Boolean values and allows them to be changed following appropriate user interaction. The radio button enforces the constraint that exactly one button is selected as shown in FIG. **11**. If Paper is selected, then the circle at **1100** is blackened. If Plastic is selected, then the circle at **1110** is selected. Both cannot be selected.

TCommand

A command encapsulates a request to an object or set of objects to perform a particular action. Commands are usually executed in response to an end-user action, such as pressing a button, selecting a menu item, or by direct manipulation. Commands are able to provide various pieces of information about themselves (e.g. name, graphic, key equivalent, whether they are active) that may be used by a control to determine its appearance. Subclasses must implement a method to examine the current selection, active user interface element, or other parameters in order to decide whether the command is active. Subclasses must override get active interest list to return notification interests that may affect whether this command is active.

FIG. **12** is a flowchart depicting the detailed logic in accordance with the subject invention. The flowchart logic commences at **1200** and control passes directly to function block **1210** where a command objects are added to a menu. The steps carried out by this function block are: 1) create menu item from a command, where a menu item is another object data structure containing a command, 2) add a menu item to a list of menu items, and 3) mark the menu's appearance is invalid in data structure fValid. Then, later when the menu is pulled down, the appearance is recomputed based on the system states

Each menu is a view. Views contain size and location information. Each menu contains a list of menu items. Each menu item contains a command and variables that reflect its current appearance. This includes whether the menu item is enabled (Boolean fEnabled), its name (TTextLabel fName), its graphic (TGraphicLabel fGraphic), and whether its appearance is currently valid (Boolean fValid). Each of these variables are determined by asking the command when the menu item was created.

Next, a query is sent to the command object for notification interests as depicted in function block **1220**. Each command has four methods to connect for different types of notifications: i) notifications that affect it's name, ii) notifications that affect a graphic, iii) notifications that affect whether the command is active, and iv) notifications that affect any data. In this case, the menu item just created for the command connects for active notification. It does this by passing a connection object to ConnectActive. The command is then responsible for connecting the connection object to notifiers affecting whether the command is active. Then control is passed to function block **1230** to query a command for the enabled state when it is necessary to draw a menu item. To draw a menu item, menu item calls method "IsActive" for its command. The command looks at whatever system state it wants to and returns whether it is active as depicted in decision block **1240** in the current context (e.g. some commands only are active when a particular type of window is in front, or when a particular type of object is selected). Then, a menu item updates its internal state (a Boolean value in each menu item) and appearance as shown in function block **1250** and **1260** to match the value returned by the command.

Whenever a user action invokes any command as shown in input block **1270**, a user causes a command to be executed. This could be from a menu item, control, or through direct manipulation of an object. This action causes a document state to be modified as shown in function block **1280**, and a document sends notification as shown in function block **1290**. When a document sends notification, the following steps are executed: 1) any menu item (or other control) connected for the notification sent by the document receives a notification message. This message includes the name of the change as well as a pointer to the object that sent the notification) a menu item then updates its state, and control is passed back to function block **1230** for further processing.

FIG. **13** is an illustration of a display in accordance with the subject invention. The menu item is Edit **1300** and has a number of sub-menu items associated with it. Undo **1310** is an active menu item and can thus be selected to carry out the associated functions. Redo**1320** is inactive and is thus presented in a greyed out fashion and cannot be selected at this time. A checkbox is also shown at **1360** as part of the debugging control panel **1350**.

Presentation Templates and Persistence

Data presentations are created from templates and saved across sessions in a user interface object. The container for all data in the system is a model. A model contains and facilitates the manipulation of data. Data exchange is facilitated through cut, copy, and paste operations. Data reference is provided by selections, anchors, and links. Data models may be embedded into any other. Users interact with models through presentations (e.g. icon, thumbnail, frame, window, dialog, control panel) that are provided by an associated user interface. Data models delegate all presentation creation and access methods to another object, called the user interface.

A user interface is a model containing a set of presentations (e.g. icon, thumbnail, frame, window) for a particular model. When required, presentations are selected from those already created based on the type of presentation desired, the user's name, locale, and other criteria. If the desired presentation is not found, a new presentation is created and added to the user interface by copying one from an associated archive. Presentations may be deleted when persistent presentation information (e.g. window size and location, scroll positions) is no longer required.

A presentation contains a set of presentable objects that wrap user interface elements (e.g. menus, windows, tools) used to view and manipulate data. Presentations provide a reference to the data these objects present. Presentations install or activate presentable objects when the presentation is activated. Similarly, these objects are removed or deactivated when the presentation is deactivated. Presentations are identified according to their purpose (e.g. icon, thumbnail, frame, window) and retain yet-to-be-determined criteria (e.g. user identity) for later selection.

A presentation is made up of a collection of presentable objects (e.g. user interface elements) that are displayed on the screen or are otherwise available when the presentation is open or active.

Presentations are created from template presentations contained in an archive. These are made up of objects such as user interface elements, which are, in turn, made up of smaller objects such as graphics and text strings.

An archive is a model containing a set of template objects, including user interface elements (e.g. windows, menus, controls, tools) and presentations (e.g. icon, thumbnail, frame, window).

Dialog Boxes & Control Panels

By using command objects in different ways, we can control two independent behaviors of a group of controls. The first is whether they affect the data immediately, or whether the user must press OK before the settings take effect. The second is whether they are independent from one another, or whether the settings represent an atomic operation.

Controls contain commands. As the user manipulates the control, the control sets parameters in the commands and cause it to be executed. Commands operate on model data specified by a selection.

Immediate

Controls that affect the data immediately contain a command that contains a selection that specifies real model data. As the user manipulates the control, the command causes this data to change. As the data changes, it sends change notification so that views and controls depending on the state of the data can accurately reflect the current state.

Delayed

Controls that are designed to not change the real data must operate on prototypical data, instead. The real model data is not changed until the user performs another action, such as pressing the OK button. This is accomplished in two ways:

The control contains a command that contains a selection that specifies the control itself. As the user manipulates the control, the command causes the control's value to change, but no other model data. When the user presses OK, a command in the OK button changes the real model data to match the values in each control the user may have manipulated.

The control contains a command that contains a selection that specifies a parameter of the command contained by the OK button. As the user manipulates the control, the command causes the OK button's command to change. When the user presses OK button, the OK button's command changes the real model data to match the values contained in itself.

Independent

Controls that act independently from one another require represent actions that can be individually undone after the. control panel or dialog session is complete. This is the normal behavior of commands once they are executed by controls.

Atomic

Other sets of controls are designed to work together and should be undone and redone as an atomic operation. This is accomplished by putting a mark on the undo stack when the dialog box or control is started. When finished, either by dismissing the control panel or when the user presses an OK button (as in II B above), all of the commands executed since the mark was placed on the undo stack are collected together into a single command group. This group can then be undone or redone as a single group.

Cancel

Control panels containing a CANCEL button (usually accompanied by an OK button, as in II B above) us a technique similar to that described III B above. A mark is put on the undo stack when the dialog box or control panel is started. If the user presses the CANCEL button, all commands placed on the undo stack since the mark are undone. This technique works regardless of whether the controls affect the data immediately or not.

Atomic Command Execution in Dialog Boxes

The object with which users interact to manipulate other objects or data is called a control. Example controls are menus, buttons, check boxes, and radio buttons. Each control contains a command, which implements an end**35** user action. Commands operate on data that is specified by a selection object. As the user manipulates the control it sets parameters in the command and causes it to be executed, thus changing the data value.

Controls that act independently from one another require represent actions that can be individually undone after the control panel or dialog session is complete. This is the normal behavior of commands once they are executed by controls. Other sets of controls are designed to work together and should be undone and redone as an atomic operation. This is the subject of this patent.

The detailed logic of the atomic execution is set forth in the flowchart presented in FIG. **14**. Processing commences at terminal **1400** where control is immediately passed to function block **1410** where a dialog box is activated. When the dialog box is activated, a mark is placed on the undo stack. The undo stack is a list of all commands the user has executed. When undo is pressed, the command on the top of the stack is undone. If not immediately redone, it is thrown away. Then, at function block **1410**, a user manipulation of a control is detected. The manipulation of a control changes the command's data value, as appropriate as set forth in function block **1430**, and executes the control. For example, a checkbox toggles the command's fChecked field between 0 and 1. Finally, the command is recorded on the undo stack so it can be subsequently undone as shown in function block **1440**.

As a user subsequently manipulates each control in the dialog box, as detected in decision block **1450**, then control passes to function block **1430**. However, if a user presses OK as detected in decision block **1460**, then control passes to function block **1420**. Finally, when each control in the dialog box is set to the user's satisfaction, the user presses the OK button. All of the commands executed since the mark was placed on the undo stack in function block **1440** are collected together into a single command group and placed back onto the undo stack as depicted in function block **1470**. A command group is a command that collects many commands together. When executed, undone, or redone, the command group executes, undoes, or redoes each command in sequence. The command group is then placed back onto the undo stack where it can be undone or redone as a single atomic operation.

Delayed Command Execution in Dialog Boxes

The object with which users interact to manipulate other objects or data is called a control. Example controls are menus, buttons, check boxes, and radio buttons. Each control contains a command, which implements an end-user action. Commands operate on data that is specified by a selection object. As the user manipulates the control it sets parameters in the command and causes it to be executed, thus changing the data value. Delaying changing of data until the user performs another action is one aspect of the subject invention. For example, controls in a dialog box may not want to change any data values until the user presses the OK button.

When a control is created a command must be specified. The control makes a copy of this command and stores it in field fCommand. If the command supplies any data values, a pointer to appropriate Get and Set methods of the command must also be specified. The control stores these method pointers in fields fGetMethod and fSetMethod, respectively. The data that is modified by a command is specified by a selection object. Normally, this selection object specifies real model data. Instead, a selection object that specifies the data value within the command of the OK button.

When a user manipulates the control, the control's command is executed and a data value within the command of the OK button is changed. As the user manipulates each control in the dialog box, the control's command is executed and a data value within the command of the OK button is changed. Thus, when a user presses the OK button, the command in the OK button updates the real model data to match the data values contained within itself as manipulated by the control's commands. This processing is repeated until control processing is completed.

Labels

Labels are graphical objects that contain a graphic or text string. They are used to identify windows, menus, buttons, and other controls. Labels are able to alter their appearance according to the state of their container. They are drawn on a medium-gray background and appear naturally only when no special state must be indicated. Labels modify their appearance when inactive, disabled, or selected.

Inactive

Window titles are set to be inactive when the window is not front-most. Similarly, control labels are set to be inactive when the control is not in the front-most window or other container. Graphic labels are blended with 55% white when inactive, in order to appear dimmed. For text labels, the inactive paint is derived from the natural paint by manipulating the saturation component of the HSV color model. The saturation is multiplied by 0.45 when inactive.

Disabled

Control labels are dimmed when the control does not apply in a particular context. Graphic labels are blended with 46% white when inactive, in order to appear dimmed. For text labels, the disabled paint is derived from the natural paint by manipulating the saturation component of the HSV color model. The saturation is multiplied by 0.54 when disabled.

Selected

Control labels are highlighted as the control is being manipulated. Graphics and text are drawn in their natural state, but on a white background, when highlighted.

Smart Control Labels

Controls use a command to determine the current state of the object or data. Following appropriate interactions with the user, the control updates the command's parameters and causes it to be executed. For example, a checkbox sets a command parameter to on or off and then executes the command to change a data value. Controls display a label to indicate its function. This label is a graphical object containing a graphic or a text string. As the control changes state, the label automatically adjusts its appearance, without requiring the developer to write additional code. These states include active/inactive, enabled/disabled, and selected/unselected.

FIG. **15** sets forth the detailed logic associated with smart label processing which commences at the start terminal **1500** where control is immediately passed to **1510** for smart label initialization. When the control is created, its label is initialized with a text string or graphic provided by its associated command. Each command provides methods called GetGraphic and GetName for this purpose. The control tells the label whether it is currently active or inactive by calling method SetActive. Similarly, the control calls method SetEnabled to tell the label whether it is enabled, and SetSelected to tell the label whether it is currently being selected by a user.

The next step in smart label processing occurs at function block **1520** when the label is drawn. When the control is activated, it calls the Draw method of its label, causing the label to appear on the screen. If inactive, the label is drawn more dimly than normal. This is done by manipulating the saturation components of the HSV color model. The saturation is multiplied by 0.45 when inactive. If disabled, the label is drawn more dimly than normal. This is done by manipulating the saturation components of the HSV color model. The saturation is multiplied by 0.54 when the label is disabled. If selected, the label on a highlighted background. Labels are normally drawn on a medium-gray background. When highlighted, labels are drawn on a white background. Otherwise, the label is drawn normally.

The next processing occurs when a label is activated/deactivated as shown in function block **1530**. When the control is activated or deactivated, it tells the label by calling the SetActive method. The control then indicates its appearance needs updating by calling Invalidate with an argument indicating the portion of the screen that needs to be redrawn. Then, at function block **1540**, processing occurs when a control is enabled/disabled. When the control is enabled or disabled, it tells the label by calling the SetEnabled method. The control then indicates its appearance needs updating by calling Invalidate with an argument indicating the portion of the screen that needs to be redrawn.

A test is then performed at decision block **1550** to determine if a control is selected or unselected. When the control is selected or unselected, it tells the label by calling the SetSelected method. The control then indicates its appearance needs updating by calling Invalidate with an argument indicating the portion of the screen that needs to be redrawn, and control is passed to function block **1520** for further processing.

Smart Window Labels

A title is displayed in a window in order to indicate its purpose. For example, the title for a window to edit a document is usually the name of the document. A label object is used to keep track of the title. This label is a graphical object containing a graphic or a text string. As the window changes state, the label automatically adjusts its appearance, without requiring the developer to write additional code. Windows can be either active or inactive. Smart Window label processing is flowcharted in FIG. **16** and the detailed logic is explained with reference thereto.

Processing commences in FIG. **16** at terminal **1600** where control is immediately passed to function block **1610** for the title to be initialized. A window title is specified by a developer when a window is created. This title is stored in a TLabel object called fTitle. The control tells the title whether it is currently active or inactive by calling method SetActive. Then, the at function block **1620**. When a window is drawn, it calls the Draw method of its fTitle object, causing the title to appear on the screen. If inactive, the title is drawn dimmer than normal. This is done by manipulating the saturation components of the HSV color model. The saturation is multiplied by 0.45 when inactive. Otherwise, the title is drawn normally.

The next step is processed at function block **1630** when the title is activated/deactivated. When a window is activated or deactivated, it tells its fTitle object by calling the SetActive method. The window then indicates its appearance needs updating by calling Invalidate with an argument indicating the portion of the screen that needs to be redrawn. Then, control is passed back to function block **1620** for redrawing the title to reflect its new state.

Decorations

Many of the visual aspects of user interface elements are common among many elements. Examples are shadows, borders, and labels. The individual visual features are referred to as decorations. Decorations can be combined with other graphics to form the visual appearance of specific user interface elements, such as windows and controls. The subject invention supports many different types of decorations.

Backgrounds

A decoration that is drawn behind another object is called a background. One type of background is drawn so as to appear flush with the surrounding drawing surface. It may be drawn with or without a frame. Another type of background is drawn with highlighting and shadow so it appears to be raised above the surrounding drawing surface. The final type of background is drawn with highlighting and shadow so it appears to be recessed beneath the surrounding drawing surface.

An example use of these backgrounds is a button. Normally the text or graphic that describes the button is drawn on a raised background. When pressed by the user, the text or graphic is redrawn on a recessed background. If the button is inactive, such as when another window is active, the text or graphic of the button could be drawn dimly on a flush background.

Borders

A decoration that surrounds another object or area is called a border. Example borders are frames and shadows. A frame is a border that surrounds another graphic, much like a frame encloses a painting in the real world. Like backgrounds, frames can be drawn to appear recessed below, flush with, or raised above a surrounding drawing surface. A shadow is a special type of border that adds a shadow around an object to make it appear as if it floats above the surrounding surface.

Decoration Colors

Many of the visual aspects of user interface elements are common among many elements. Examples are shadows, borders, and labels. Each of these individual visual features are referred to as a decoration. Decorations can be combined with other graphics to form the visual appearance of specific user interface elements, such as windows and controls. Some decorations use highlighting and shadows to appear as if they are above or below the surrounding drawing surface. Decorations are able to derive automatically these highlighting and shadow paints.

Fill Paint

The fill paint represents the decoration's primary color. All other paints are derived from the fill paint. The fill paint is stored by the directoration in a TColor field called fFillPaint. The fill paint is normally specified by the developer when the decoration is created. However, if no color is specified, a medium gray is selected.

Frame Paint

The frame paint is used to draw a line around the decoration to provide visual contrast. The frame paint is stored by the decoration in a TColor field called fFramePaint. The frame paint may be specified by the developer when the decoration is created. However, if no frame paint is specified, it is computed automatically from the fill paint. This is accomplished by manipulating the saturation and value components of the HSV color model. The saturation is multiplied by four, with a maximum value of 1. The value is divided by four.

Highlight Paint

The highlight paint is used to draw lines where light would hit the object if it were an actual three-dimensional object. The highlight paint is stored by the decoration in a TColor field called fHighlightPaint. The highlight paint may be specified by the developer when the decoration is created. However, if no highlight paint is specified, it is computed automatically from the fill paint. This is accomplished by manipulating the saturation and value components of the HSV color model. The saturation is multiplied by 0.8. The value is multiplied by 1.25, with a maximum value of 1.

Shadow Paint

The shadow paint can be used to draw lines where the object would be shaded if it were an actual three-dimensional object. The shadow paint is stored by the decoration in a TColor field called fShadowPaint. The shadow paint may be specified by the developer when the decoration is created. However, if no shadow paint is specified, it is computed automatically from the fill paint. This is accomplished by manipulating the saturation and value components of the HSV color model. The saturation is multiplied by 2 with a maximum value of 1. The value is divided by 2.

Separating Input Syntax From Semantics

A graphical user interface is manipulated by moving a mouse, clicking on objects to select them, dragging objects to move or copy then, and double-clicking to open them. These operations are called direct manipulations, or interactions. The sequence of events corresponding to a user pressing, moving, and releasing a mouse is called an input syntax. Certain sequences of events are used to indicate particular actions, called semantic operations.

The separation of the code that understands the input syntax from the code that implements semantic operations is the subject of this patent. This processing is embodied in objects called Interacts and Intractable, respectively. FIG. **17** illustrates how these objects are created and how the objects communicate with each other during a typical interaction with an object that can be moved and selected.

Processing commences at terminal **1700** where control is passed immediately to function block **1710** to determine if the mouse button has been pressed. An event is sent to the object responsible for the portion of the screen at the location where the mouse button was pressed. This object is called a View. Then, at function block **1720** the Interactor is created to parse the input syntax. This is done by calling the CreateInteractor method of the view. When the Interactor is created, pointers to objects that implement possible user actions are passed as parameters.

For the purposes of this discussion, assume the user pressed the mouse button down on an object that can be selected and moved. In this case, an object that implements selection and an object that implements movement for the target object are passed as parameters to the Interactor. The initial View could implement both of these behaviors, or they could be implemented by one or two separate objects. The object or objects are referred to collectively as the Interactable.

The Interactor is started at function block **1730**. This processing returns the Interactor to the View and commences processing of the Interactor. This is accomplished by calling the Interactor's Start method and passing the initial mouse event as a parameter. The Start method saves the initial mouse event in field fInitialEvent. Since only one mouse event has been processed thus far, the only action possible is selecting. The Interactor enters select mode by setting variable fInteractionType to constant kSelect. It asks the Interactable to begin the selection operation by calling its SelectBegin method.

Then, the Interactor waits for a short time to pass as shown in function block **1740**. A new mouse event is sent to the Interactor when the time is up which indicates the current state of the mouse. Then, if the system detects that the mouse is still down at decision block **1750**, control is passed to function block **1740**. Otherwise, control is passed to terminal **1760**. If the mouse button is still down, the interactor makes sure it is still in the correct state and asks the Interactable to implement the correct operation. The Interactor is Selecting if fInteractionType is kSelecting. It is Moving if the fInteractionType is kMoving.

If selecting, the Interactor compares the current mouse location with the initial mouse location. The current mouse location is obtained by calling the GetCurrentLocation method. The initial mouse location is obtained by calling the GetInitialLocation method. If the two are the same or differ by only a small amount, the user is still selecting the object. The Interactor then asks the Interactable to continue the selection operation by calling its SelectRepeat method. However, if the two points differ beyond a predetermined threshold, the user has begun moving the object. In this case, the Interactor asks the Interactable to terminate the selection operation by calling its SelectEnd method. It then asks the Interactable to begin the move operation by callings its MoveBegin method. In each case, the current mouse location is passed as an argument. If Moving, the Interactor asks the Interactable to continue the move operation by calling its MoveRepeat method. It passes the current mouse location as an argument.

When the user releases the mouse button, it signals the end of the current operation. If Selecting, the Interactor asks the Interactable to terminate the selection operation by calling its SelectEnd method. If moving, the Interactors asks the Interactable to terminate the move operation by calling its MoveEnd method.

Localized Presentations

Localization is the process of updating an application to conform to unique requirements of a specific locale. It may involve language translation, graphic substitution, and interface element reorientation. For example, the text used in labels, titles, and messages depends upon the selected language. Its direction and orientation may affect the placement and orientation of a menu, menubar, title, scrollbar, or toolbar. Similarly, the selection of icons and other graphical symbols may be culturally dependent. Unfortunately, having many localized versions of user interface elements in memory is very expensive. Instead, localized versions of user interface elements are kept on disk until required in memory.

Further, it is very error-prone and expensive to keep track of all of the user interface elements and decide which version to use. Instead, when a user interface element is required, the appropriate one is selected automatically by the system, according to the current language and other cultural parameters, and read into memory.

Once localized, user interface elements are stored in a disk dictionary. A disk dictionary is an object that, when given a key, returns a value after reading it in from disk. This disk dictionary is managed by an object called an archive. An archive is responsible for putting together the individual user interface elements that make up a particular presentation. The process of selecting the proper user interface element is presented in FIG. **19**.

Processing commences at terminal **1900** and immediately passes to function block **1910** when a user requests a presentation. A TOpenPresentation Command is sent to the data model, indicating that the user wants to view or edit this data. A command is sent to the data model to indicate that the user wants to view or edit the data. This command is called a TOpenPresentationCommand. A presentation is a set of user interface elements that, together, allow the user to view or edit some data. Presentations are stored across sessions in User Interface object, thus maintaining continuity for the user. User interface elements are stored on disk until needed in memory. They may be required as part of a data presentation the user has requested, or they may be needed for translation or another localization process. Each user interface element contains an ID which uniquely references that element. However, all localized versions of the same user interface element share a single ID.

In order to differentiate the localized versions, the particular language, writing direction, and other cultural parameters are stored with each localized user interface element. Together, these parameters are referred to as the locale. All of the user interface elements are stored in a file. This file is organized like a dictionary, with one or more key/value pairs. The key is an object which combines the ID and the locale. The value is the user interface element itself.

A new presentation must be created next at function block **1920**. If an appropriate presentation does not already exist, a new one must be created from a template by the user interface Archive. A new presentation is created from a template stored in the archive by calling its CreatePresentation method. A presentation type is passed to this method as a parameter. This type includes such information as the type of data to be displayed, whether it is to be in its own window or part of another presentation, and so on. Finally, at function block **1930**, an Archive builds the presentation, selecting user interface elements according to locale. If the Archive is able to build a presentation of the specified type, it collects together each user interface element that makes up the presentation and returns this to the user interface object.

For each presentation the archive is able to make, it has a list of user interface element IDs that together make up the presentation. The user interface elements are stored on disk maintained by a disk dictionary object called. Given a key, the disk dictionary will return the corresponding user interface element. The user interface element ID makes up the primary component of this key. A secondary component of the key is the desired locale. A locale is an object that specifies the natural language and other cultural attributes of the user. The locale obtained automatically by the Archive from a Preferences Server. This server contains all of the individual preferences associated with the user.

The locale, as obtained from the preferences server, is combined with the ID into a single object called a TUserInterfaceElementKey. This key passed as a parameter to the GetValue method of the disk dictionary. If a user interface element with a matching ID and locale is found, it is returned and included as part of the presentation. Otherwise, the locale parameter must be omitted from the key, or another locale must be specified until an appropriate user interface element is found.

Interaction Framework System

Users of an object oriented operating system's graphical user interface often move a mouse, click on objects to select them, drag objects to move or copy then, and double-click to open an object. These operations are called direct manipulations, or interactions. The sequence of events corresponding to a user pressing, moving, and releasing the mouse is called the input syntax. Certain sequences of events are used to indicate particular actions, called semantic operations. This invention discloses the method and apparatus for translating input syntax into semantic operations for an object that supports Select, Peek, Move, AutoScroll, and Drag/Drop (Copy).

The invention detects a mouse button depression and then employs the following logic:

(a) If an Option key was depressed when the user pressed the mouse button, the system enters drag mode by setting variable fInteractionType to constant kDrag. The system then commences a drag operation using the selected object as the target of the operation; o r

(b) if the Option key was not depressed, then the system enters selection mode by setting variable fInteractionType to constant kSelect. Then, the select operation is commenced.

If a user already had the mouse button depresses and continues to hold the mouse button down, then the following logic is engaged. If the system is in select mode, then the system first determines whether the user has moved the mouse beyond a certain threshold, called the move threshold. This is done by comparing the initial mouse location, returned by the GetInitialLocation method, with the current mouse location, returned by the GetCurrentLocation method. If the mouse has moved beyond the move threshold, the system ends select mode and enters move mode. It does this by setting variable fInteractionType to constant kMove. The system then queries the object to terminate the select operation by calling its SelectEnd method. The system then initiates a move operation by calling its MoveBegin method.

Otherwise, if the mouse has not moved, the system checks how long the mouse has been down. It does this by comparing the initial mouse down time, returned by the GetInitialTime method, with the current time, returned by the GetCurrentTime method. If the mouse has been down beyond a certain threshold, called the peek threshold, the system ends select mode and enters peek mode. It does this by setting variable fInteractionType to constant kPeek. It asks the object to end the select operation by callings its SelectEnd method, and begins a peek operation by calling its PeekBegin method. Otherwise, if the mouse has not moved, or it has not been down beyond the peek threshold, the system continues the select operation by calling the object's SelectRepeat method. If the system detects that a user is in Move mode, the system first determines whether the user has moved the mouse within the window, on the border of the window, or outside the window. It does this by comparing the current mouse location, returned by the GetCurrentLocationMethod, with the bounds of the object's container, returned by GetContainerBounds.

If the mouse is still within the bounds of the window, the system continues the move operation by calling the object's MoveRepeat method. If the mouse is on the border of the window, this indicates an AutoScroll operation. The system asks the object's container to scroll in the direction indicated by the mouse location. This is done by calling the container's AutoScroll method and passing the current mouse location as a parameter. Once complete, the system continues the move operation by calling the object's MoveRepeat method.

If the mouse has moved outside the window, the system ends move mode and enters drag mode. It does this by setting variable fInteractionType to constant kDrag. It asks the object to end the move operation by calling its MoveEnd method. It asks the object to begin the drag operation by calling its DragBegin method. If the system is in drag mode, the system continues the drag operation by calling the object's DragRepeat method. If the system is in peek mode, the system first determines whether the user has moved the mouse beyond a certain threshold, called the move threshold. This is done by comparing the initial mouse location, returned by the GetInitialLocation method, with the current mouse location, returned by the GetCurrentLocation method.

If the mouse has moved beyond the move threshold, the system ends peek mode and enters move mode. It does this by setting variable fInteractionType to constant kMove. It asks the object to end the peek operation by calling its PeekEnd method. It asks the object to begin the move operation by calling its MoveBegin method. Otherwise, if the mouse has not moved, the system continues the peek operation by calling the object's PeekRepeat method.

If the system detects that a user releases the mouse button, then if the system is in select mode, the system ends select mode. It does this by setting variable fInteractionType to constant kNone. The system queries the object to end the select operation by calling its SelectEnd method. If the system is in move mode, the system ends move mode. It does this by setting variable fInteractionType to constant kNone. Then, the system queries the object to end the move operation by calling its MoveEnd method and ends drag mode by setting variable fInteractionType to constant kNone. It asks the object to end the drag operation by calling its DragEnd method. If the system is in peek mode, the system ends peek mode. It does this by setting variable fInteractionType to constant kNone. It asks the object to end the peek operation by calling its PeekEnd method.

Accordingly, it is a primary objective of the present invention to provide an innovative hardware and software system which enables the contents of a window to update dynamically as a user moves a scrollbar thumb. The system detects when a user presses down on a scrollbar thumb. When the user presses down on the scrollbar thumb, the system begins initiation of a scroll command to change the portion of the data that is exposed in the window. A command is an object that implements an end-user action, such as scrolling. A scroll command has one parameter, the position to which the content view should be scrolled. The system sets this position to the current scroll position. This is accomplished by calling the command's SetScrollPosition and setting the scroll to position to the value returned by the scrollbar's method GetScrollPosition.

When a user moves the mouse within the scrollbar, the system continues the execution of the scroll command to dynamically change the portion of the data exposed in the window. The system sets the scroll position of the command to the new scroll position. This is accomplished by calling the command's SetScrollPosition and setting the value equal to the value returned by the scrollbar's method GetScrollPosition. The execution of the command is then repeated by calling its DoRepeat method. This causes the content view to scroll to the new position. This processing is continued while a user continues to hold the mouse button down.

When a user releases the mouse button, the system ends the execution of the scroll command to dynamically change the portion of the data exposed in the window. The system sets the scroll position of the command to the final scroll position. This processing is accomplished by calling the command's SetScrollPosition and setting it equal to the value returned by the scrollbar's method GetScrollPosition.

FIG. **20** is a flowchart illustrating the detailed logic associated with scrolling in accordance with the subject invention. Processing commences at terminal block **2000** and immediately passes to function block **2010** where the current scroll position is initialized based on the current cursor location. Then, at decision block **2020**, a test is performed to detect if the scrollbar thumb has been selected. An example of a scrollbar thumb is shown in FIG. **21**A at label **2110**. If the scrollbar thumb has been selected, then control passes to decision block **2030** to determine if the scrollbar thumb has been moved. If so, then the scroll position is set to the new position of the scrollbar thumb and the display is reformatted to reflect the immediate scroll operation and displayed for the user. If the scrollbar thumb has not moved, another test is performed at decision block **2050** to determine if the scrollbar thumb has been released. If not, then control is returned to decision block **2030**. If the scrollbar thumb has been released, then control passes to function block **2060** to end the scroll operation and return the system to a nonscroll operational status and processing is completed at terminal **2070**.

FIGS. **21**A, **21**B and **21**C illustrate window scrolling in accordance with the subject invention. In FIG. **21**A, the scrollbar thumb **2110** is located at the top of the window **2112**. FIG. **21**B shows the scrollbar thumb **2120** moved to the middle of the window and the window's contents **2122** updated accordingly. FIG. **21**C shows the scrollbar thumb **2140** moved to the bottom of the window and the bottom most portion of the window **2142** displayed.

While the invention has been described in terms of a preferred embodiment in a specific system environment, those skilled in the art recognize that the invention can be practiced, with modification, in other and different hardware and software environments within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new, and desire to secure by Letters Patent is:

1. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a first object, connection information representing the first object's interest in, and an associated object method for, receiving notification of a change to a second object;

(b) registering the connection information with a connection object;

(c) creating an event representing a change in the second object, responsive to the change in the second object; and

(d) notifying the first object of the event by invoking the associated object method for receiving notification registered with the connection object only if the event information corresponds to an interest registered on behalf of the first object.

2. The operating method of claim **1**, wherein the connection object is associated with status information, the operating method further comprising the step of:

(b. 1) using the connection information in the connection object to configure the status information to represent whether the notifying step (d) is activated or inactivated.

3. The operating method of claim **1**, wherein the connection information is associated with a notification type corresponding to a connection object method, the operating method further comprising the step of:

(c. 1) invoking the connection object method corresponding to the notification type specified by the connection information in the connection object.

4. The operating method of claim **3** wherein:

each of a notification type plurality corresponds to a unique connection object method different from the connection object method corresponding to another of the notification type plurality.

5. The operating method of claim **3** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a name associated with the first object.

6. The operating method of claim **3** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a graphic associated with the first object.

7. The operating method of claim **3** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to create or modify data associated with the first object.

8. The operating method of claim **3** further comprising the step of:

(C. 1.1) invoking a connection object method responsible for using the connection information in the connection object to read data associated with the first object.

9. The operating method of claim **8** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute an undo function associated with the first object.

10. The operating method of claim **8** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute an redo function associated with the first object.

11. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of an event listener object, connection information representing the event listener object's interest in, and an associated object method for, receiving notification of a change to an event source object;

(b) registering the connection information with a connection object;

(c) creating an event representing a change in the event source object, responsive to the change in the event source object; and

(d) notifying the event listener object of the event by invoking the associated object method for receiving notification registered with the connection object only if the event information corresponds to an interest registered on behalf of the event listener object.

12. The operating method of claim **11**, wherein the connection object is associated with status information, the operating method further comprising the step of:

(b. 1) using the connection information in the connection object to configure the status information to enable or disable the notifying step (d).

13. The operating method of claim **11** wherein the connection information is associate with a notification type corresponding to a connection object method, the operating method further comprising the step of:

(c. 1) invoking the connection object method corresponding to the notification type specified by the connection information in the connection object.

14. The operating method of claim **13**, wherein each of a notification type plurality corresponds to the same single connection object method, the operating method further comprising the step of:

(c. 1.1) transferring notification type information between two objects.

15. The operating method of claim **13** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a name associated with the event listener object.

16. The operating method of claim **13** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a graphic icon associated with the event listener object.

17. The operating method of claim **13** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to read data associated with the event listener object.

18. The operating method of claim **13** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to create or modify data associated with the event listener object.

19. The operating method of claim **18** wherein the data associated with the event listener object includes descriptive textual data.

20. The operating method of claim **18** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute an undo function associated with the event listener object.

21. The operating method of claim **18** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute an redo function associated with the event listener object.

22. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a consumer object, connection information representing the consumer object's interest in, and an associated object method for, receiving notification of a change to a supplier object;

(b) registering the connection information with a channel object;

(c) creating an event representing a change in the supplier object, responsive to the change in the supplier object; and

(d) notifying the consumer object of the event by invoking the associated object method for receiving notification registered with the channel object only if the event information corresponds to an interest registered on behalf of the consumer object.

23. The operating method of claim **22**, wherein the channel object is associated with status information, the operating method further comprising the step of:

(b. 1) using the connection information in the channel object to configure the status information to make the notifying step (d) active or passive.

24. The operating method of claim **22**, wherein the connection information is associated with a notification type corresponding to a channel object method, the operating method further comprising the step of:

(c.1) invoking the channel object method corresponding to the notification type specified by the connection information in the channel object.

25. The operating method of claim **24**, wherein a notification type plurality all correspond to the same single channel object method, the operating method further comprising the step of:

transferring notification type information-between two objects.

26. The operating method of claim **24** further comprising the step of:

(c. 1.1) invoking a channel object method responsible for using the connection information in the channel object to create or modify data associated with the consumer object.

27. The operating method of claim **24** further comprising the step of:

(c. 1.1) invoking a channel object method responsible for using the connection information in the channel object to read data associated with the consumer object.

28. The operating method of claim **24** wherein the event has an associated type attribute.

29. The operating method of claim **22** wherein the creating step (c) is initiated by the channel object.

30. The operating method of claim **22** wherein the creating step (c) is initiated by the supplier object.

31. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a receiver object, connection information representing the receiver object's interest in, and an associated object method for, receiving notification of a change to a source object;

(b) registering the connection information using a connection object;

(c) creating an event representing a change in the source object, responsive to the change in the source object; and

(d) notifying the receiver object of the event by invoking the associated object method for receiving notification registered using the connection object only if the event information corresponds to an interest registered on behalf of the receiver object.

32. The operating method of claim **31**, wherein the connection object is associated with status information, the operating method further comprising the step of:

(b. 1) using the connection information in the connection object to configure the status information to represent whether the notifying step (d) is activated or inactivated.

33. The operating method of claim **31**, wherein the connection information is associated with a notification type corresponding to a connection object method, the operating method further comprising the step of:

(c. 1) invoking the connection object method corresponding to the notification type specified by the connection information in the connection object.

34. The operating method of claim **33** wherein:

each of a notification type plurality corresponds to a unique connection object method different from the connection object method corresponding to another of the notification type plurality.

35. The operating method of claim **33** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a name associated with the receiver object.

36. The operating method of claim **33** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a graphic associated with the receiver object.

37. The operating method of claim **33** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to create or modify data associated with the receiver object.

38. The operating method of claim **33** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to read data associated with the receiver object.

39. The operating method of claim **38** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute an undo function associated with the receiver object.

40. The operating method of claim **38** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute a redo function associated with the receiver object.

41. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a receiver object, connection information representing the receiver object's interest in, and an associated object method for, receiving notification of a change to a source object:

(b) registering the connection information using a connection object;

(c) creating an event representing a change in the source object, responsive to the change in the source object;

(d) notifying the receiver object of the event by invoking the associated object method for receiving notification registered using the connection object only if the event information corresponds to an interest registered on behalf of the receiver object; and

(e) using the connection information in the connection object to configure status information to enable the notifying step (d).

42. A method for operating a computer-implemented event notification system for propagating, among a plurality

of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a receiver object, connection information representing the receiver object's interest in, and an associated object method for, receiving notification of a change to a source object;

(b) registering the connection information using a connection object;

(c) creating an event representing a change in the source object, responsive to the change in the source object;

(d) notifying the receiver object of the event by invoking the associated object method for receiving notification registered using the connection object only if the event information corresponds to an interest registered on behalf of the receiver object; and

(e) using the connection information in the connection object to configure status information to disable the notifying step (d).

43. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a receiver object, connection information representing the receiver object's interest in, and an associated object method for, receiving notification of a change to a source object;

(b) registering the connection information using a connection object;

(c) creating an event representing a change in the source object, responsive to the change in the source object;

(d) notifying the receiver object of the event by invoking the associated object method for receiving notification registered using the connection object only if the event information corresponds to an interest registered on behalf of the receiver object;

said connection information being associated with a notification type corresponding to a connection object method;

(e) invoking the connection object method corresponding to the notification type specified by the connection information in the connection object;

each of a notification type plurality corresponding to the same single connection object method; and

(f) transferring notification type information between two objects.

44. The operating method of claim **43** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a name associated with the receiver object.

45. The operating method of claim **43** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to modify a graphic icon associated with the receiver object.

46. The operating method of claim **43** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to read data associated with the receiver object.

47. The operating method of claim **43** further comprising the step of:

(c. 1.1) invoking a connection object method responsible for using the connection information in the connection object to create or modify data associated with the receiver object.

48. The operating method of claim **47** wherein the data associated with the receiver object includes descriptive textual data.

49. The operating method of claim **47** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute an undo function associated with the receiver object.

50. The operating method of claim **47** further comprising the step of:

(c. 1.2) invoking a connection object method responsible for using the connection information in the connection object to execute a redo function associated with the receiver object.

51. A method for operating a computer-implemented event notification system for propagating, among a plurality of objects, events representing changes in the objects, the operating method comprising the steps of:

(a) creating, on behalf of a receiver object, connection information representing the receiver object's interest in, and an associated object method for, receiving notification of a change to a source object;

(b) registering the connection information with a notifier object;

(c) creating an event representing a change in the source object, responsive to the change in the source object; and

(d) notifying the receiver object of the event by invoking the associated object method for receiving notification registered with the notifier object only if the event information corresponds to an interest registered on behalf of the receiver object.

52. The operating method of claim **51**, wherein the notifier object is associated with status information, the operating method further comprising the step of:

(b. 1) using the connection information in the notifier object to configure the status information to make the notifying step (d) active or passive.

53. The operating method of claim **51**, wherein the connection information is associated with a notification type corresponding to a notifier object method, the operating method further comprising the step of:

(c. 1) invoking the notifier object method corresponding to the notification type specified by the connection information in the notifier object.

54. The operating method of claim **53**, wherein a notification type plurality all correspond to the same single notifier object method, the operating method further comprising the step of:

transferring notification type information between two objects.

55. The operating method of claim **53** further comprising the step of:

(c. 1.1) invoking a notifier object method responsible for using the connection information in the notifier object to create or modify data associated with the receiver object.

56. The operating method of claim **53** further comprising the step of:

(c. 1.1) invoking a notifier object method responsible for using the connection information in the notifier object to read data associated with the receiver object.

57. The operating method of claim **53** wherein the event has an associated type attribute.

58. The operating method of claim **51** wherein the creating step (c) is initiated by the notifier object.

59. The operating method of claim **51** wherein the creating step (c) is initiated by the source object.

* * * * *

EXHIBIT 11

US00RE39486E

(19) United States
(12) **Reissued Patent**
Cleron et al.

(10) **Patent Number: US RE39,486 E**
(45) **Date of Reissued Patent: Feb. 6, 2007**

(54) **EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM**

(75) Inventors: **Michael A. Cleron**, Menlo Park, CA (US); **Stephen Fisher**, Menlo Park, CA (US); **Timo Bruck**, Mountain View, CA (US)

(73) Assignee: **Apple Computer, Inc.**, Cupertino, CA (US)

(21) Appl. No.: **10/408,789**

(22) Filed: **Apr. 3, 2003**

(Under 37 CFR 1.47)

Related U.S. Patent Documents

Reissue of:

(64)	Patent No.:	**6,212,575**
	Issued:	**Apr. 3, 2001**
	Appl. No.:	**08/435,377**
	Filed:	**May 5, 1995**

(51) **Int. Cl.**
G06F 9/00 (2006.01)
G06F 9/46 (2006.01)

(52) **U.S. Cl.** **719/328**; 719/329; 709/201; 709/202; 709/203

(58) **Field of Classification Search** 719/328–329; 709/200–203
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,297,249	A	*	3/1994	Bernstein et al.
5,339,430	A	*	8/1994	Lundin et al.
5,481,666	A	*	1/1996	Nguyen et al.
5,530,852	A	*	6/1996	Meske, Jr. et al.
5,537,526	A	*	7/1996	Anderson
5,548,722	A	*	8/1996	Jalalian
5,581,686	A	*	12/1996	Koppolu et al.
5,584,035	A	*	12/1996	Duggan et al.
5,634,129	A	*	5/1997	Dickinson
5,669,005	A	*	9/1997	Curbow

FOREIGN PATENT DOCUMENTS

EP	0 631 456 A2	*	12/1994
GB	2 242 293	*	1/1990

OTHER PUBLICATIONS

Reinhardt, Andy, "The Network with Smarts" BYTE, Oct. 1994, pp. 51–64.*

Lippman, Stanley B., "C++ Primer" 2nd edition, Addison–Wesley, 1991, pp. 394–397.*

Potel et al; The Architecture of the Taligent System; Dr. Dobbs Journal on CD–ROM, SP 94.*

Rush, Jeff; OpenDoc; Dr. Dobb's Journal on CD–ROM, SP 94.*

Piersol, Kurt; A Close–Up of OpenDoc; AIXpert, Jun. 1994.*

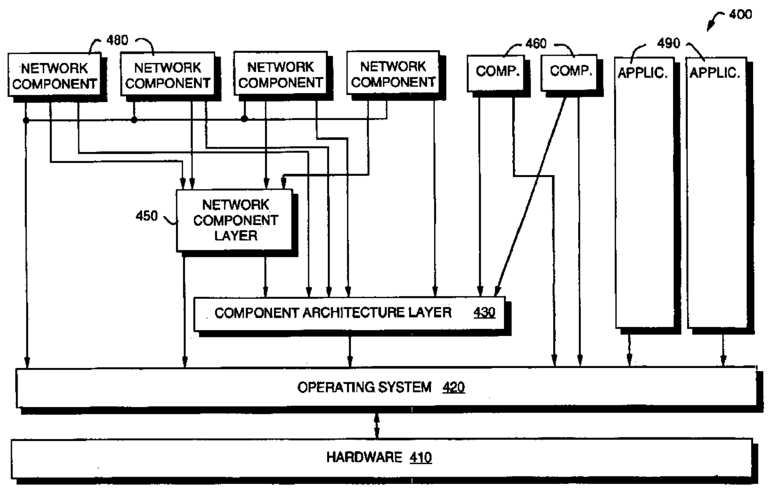
(Continued)

Primary Examiner—William Thomson
(74) *Attorney, Agent, or Firm*—Fenwick & West LLP

(57) **ABSTRACT**

An extensible and replaceable network-oriented component system provides a platform for developing networking navigation components that operate on a variety of hardware and software computer systems. These navigation components include key integrating components along with components configured to deliver conventional services directed to computer networks, such as Gopher-specific and Web-specific components. Communication among these components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a high-modular cooperating layered-arrangement between the network component system and the component architecture allows any existing component to be replaced, and allows new components to be added, without affecting operation of the network component system.

20 Claims, 8 Drawing Sheets



OTHER PUBLICATIONS

Schmidt et al; "An object–oriented framework for developing network server daemons", C+++ World Conference, pp. 1–15, Oct. 1993.*

"Leveraging object–oriented frameworks", Taligent white paper, 1993.*

Andert, Glerk; "Object–Frameworks in the Taligent OS", IEEE electronic Library, pp. 112–121, 1994.*

Helm et al, "Integrating information retrieval and domain specific approaches for browsing and retrieval in object–oriented class libraries", ACM Digital Library, 1991.*

Monnard et al; An object–oriented scripting environment for the WEBSs electronic book system' ACM Digital Library, 1992.*

Norr, Henry. "Cyberdog could be a breakthrough if it's Kept on a leash", MacWeek, Nov. 14, 1994, v8, n45, p. 50.*

Hess, Robert, "Cyberdog to fetch Internet Resources for Open Doc apps." MacWeek, Nov. 7, 1994, v8, n44, p. 44.*

Harkey et al, "Object component suites", Datamation, Feb. 15, 1995, v41, n3, p. 44.*

Prosise, Jeff, "Much ado about object", PC Magazine, Feb. 7, 1995, v14, n3, p. 257.*

Bonner, Paul, "Component software: putting the pieces together", Computer Shopper, Sep. 1994, v14, n9, p. 532.*

Gruman, Galen, "OpenDoc & OLE 2.0", MacWorld, Nov. '94, v11, n11, p. 96.*

Spiegel, Leo "OLE promises barrier–free computing", InfoWorld, Mar. 6, '95, v17, n10, p. 53.*

Develop, The Apple Technical Journal, "Building an OpenDoc Part Handler", Issue 19, Sep. 1994, pp. 6–16.*

S.H. Goldberg and J.A. Mounton, Jr. A Base for Portable Communications Software, IBM Systems Journal, vol. 30 (1991) No. 3, Armonk, NY, pp. 259–279.*

E.C. Arnold and D.W. Brown, Object Oriented Software Technologies Applied to Switching System Architecture and Software Development Processes, AT&T Bell Laboratories, Naperville, IL, vol. II, pp. 97–106.*

* cited by examiner

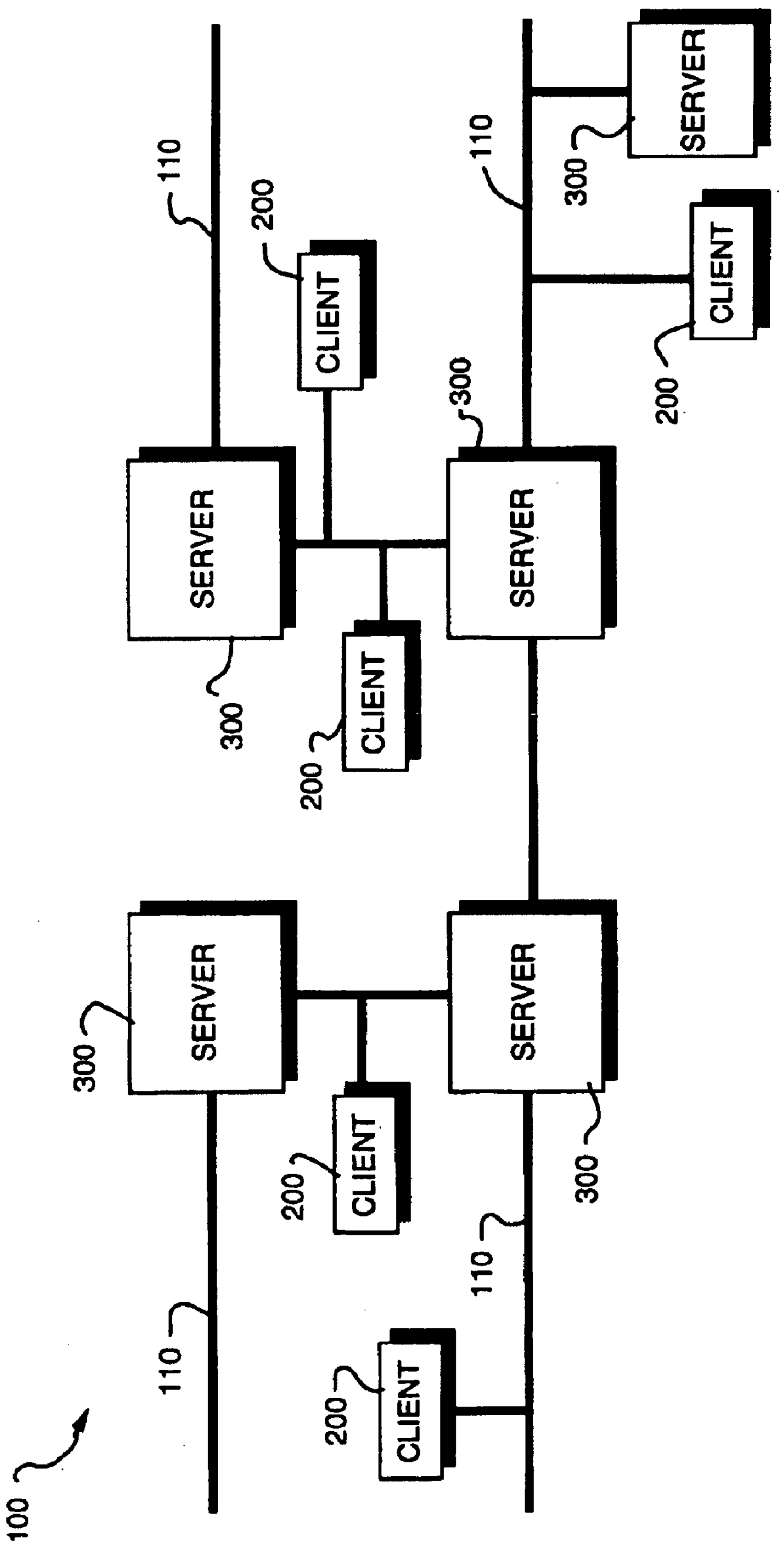

100
110
110
110
110
200
200
200
200
200
300
300
300
300
300
300
SERVER
SERVER
SERVER
SERVER
SERVER
CLIENT
CLIENT
CLIENT
CLIENT
CLIENT

FIG. 1

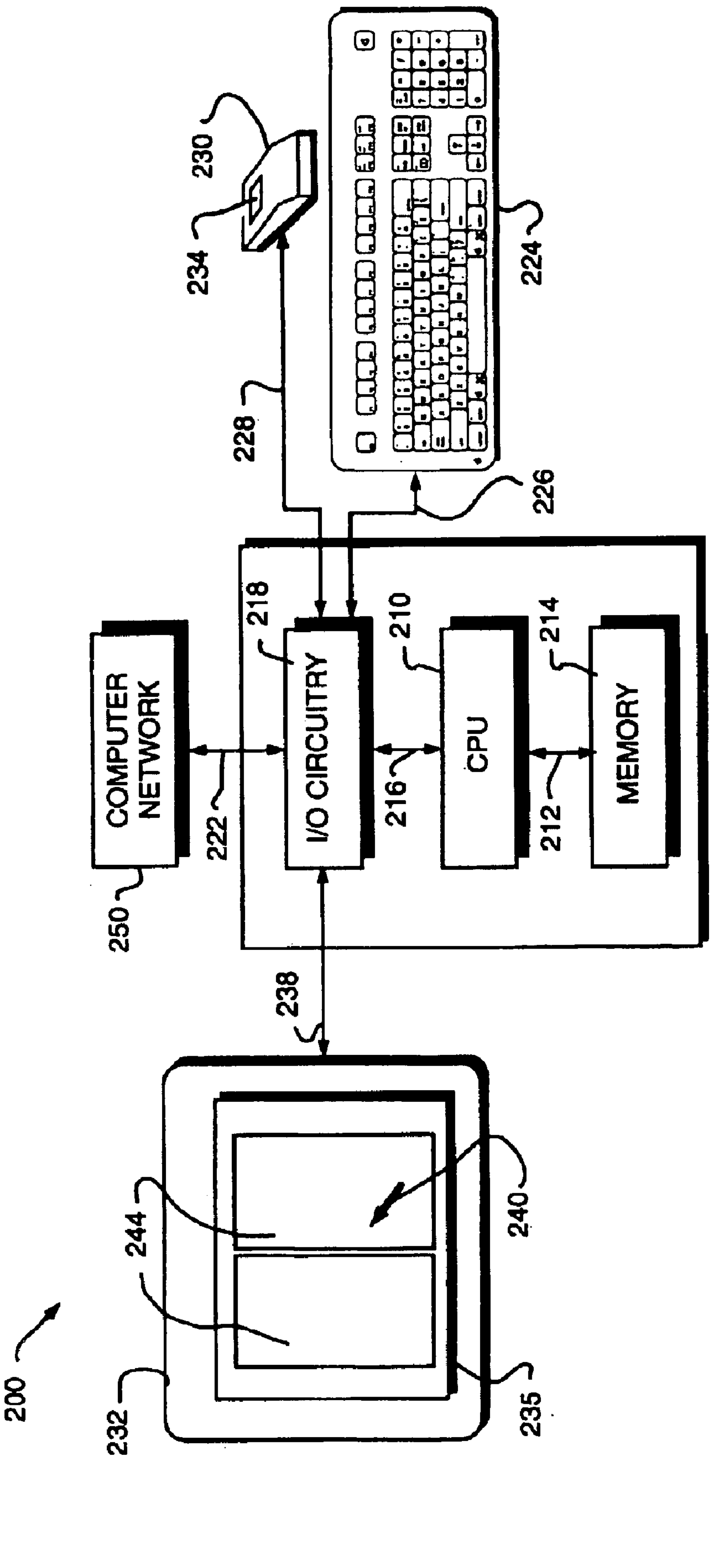

200
232
235
244
240
238
250
COMPUTER NETWORK
222
I/O CIRCUITRY
218
216
CPU
210
212
MEMORY
214
228
226
230
234
224

FIG. 2

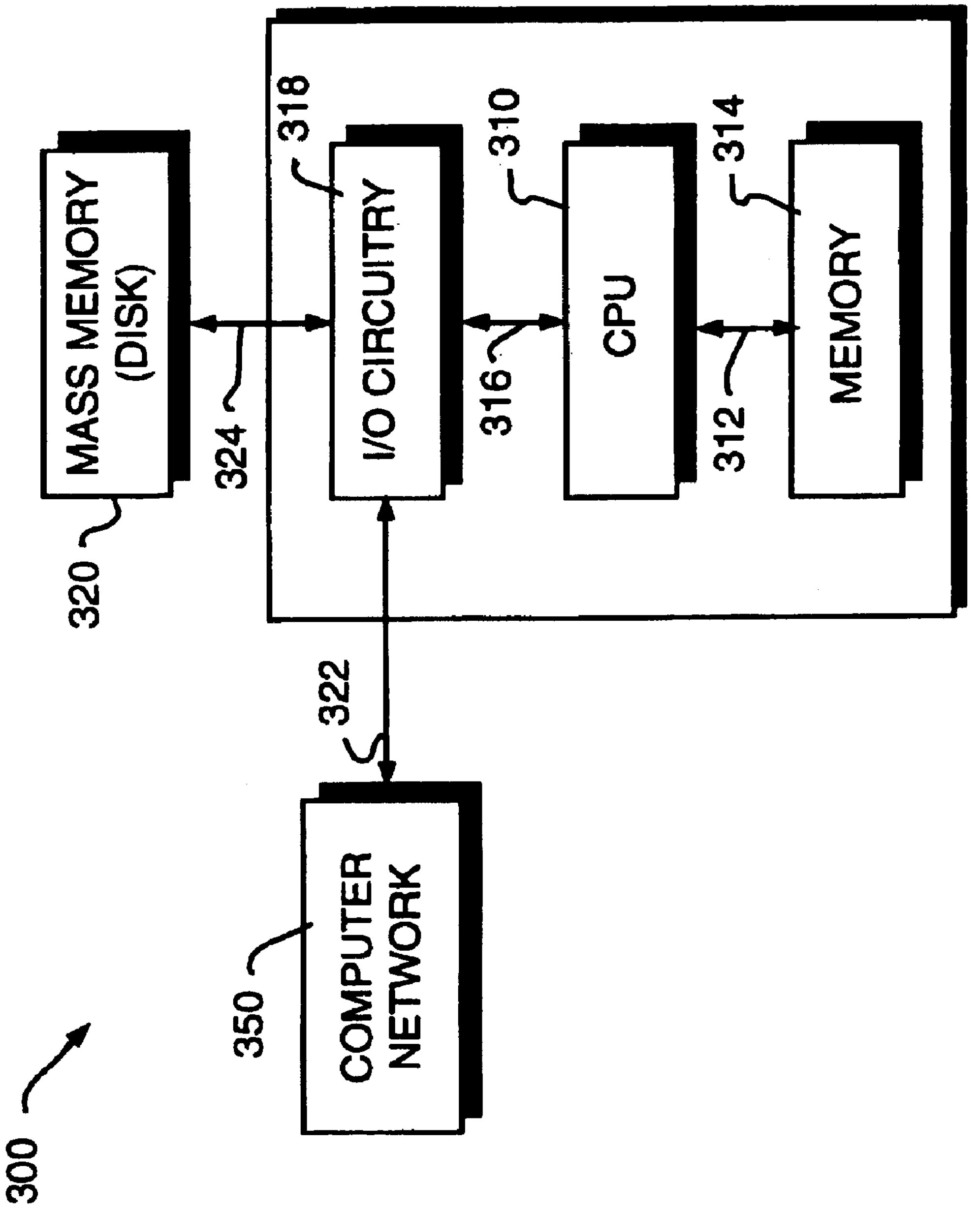

300
MASS MEMORY (DISK)
320
324
I/O CIRCUITRY
318
316
CPU
310
312
MEMORY
314
322
COMPUTER NETWORK
350

FIG. 3

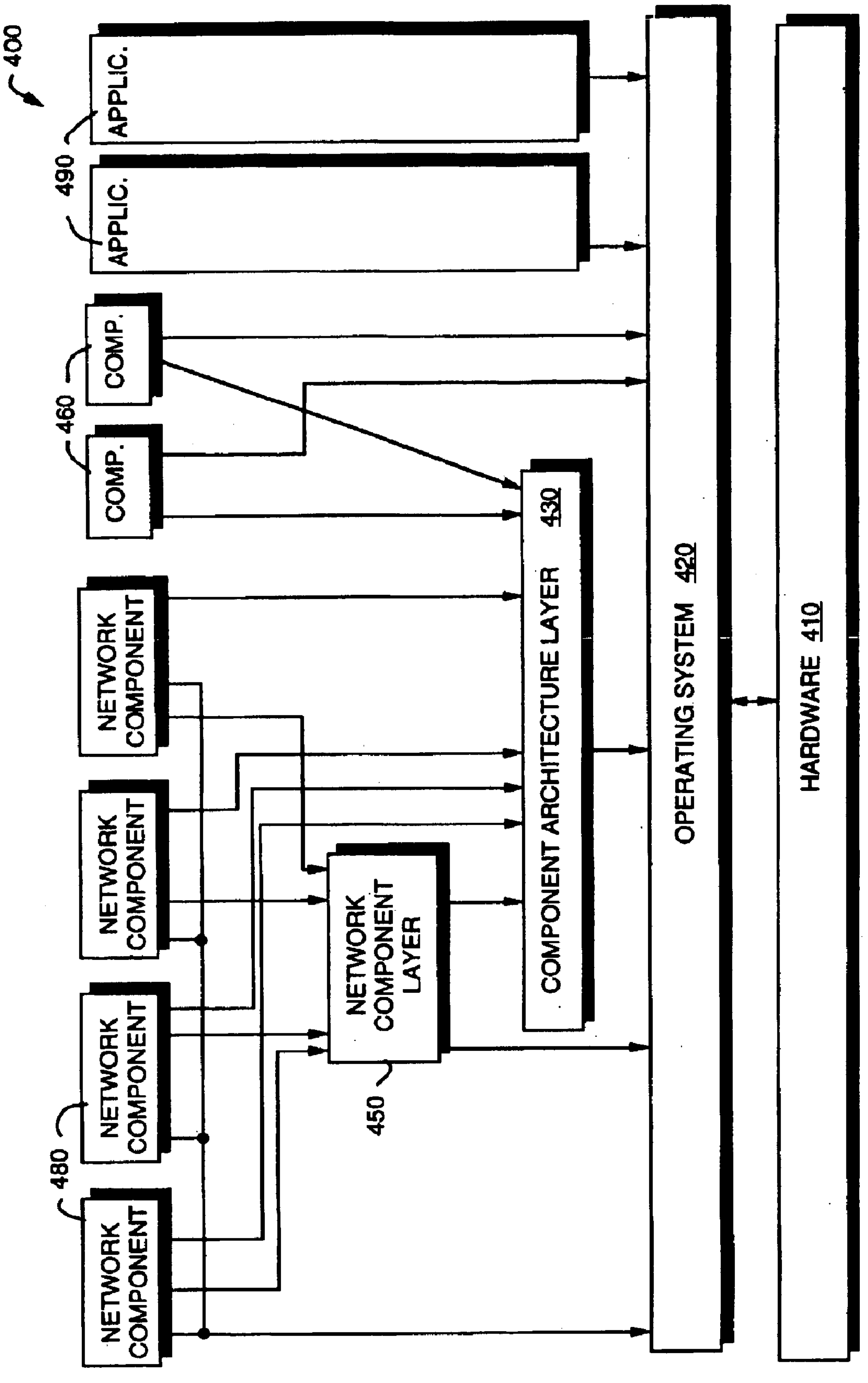

400
APPLIC.
APPLIC.
490
COMP.
COMP.
460
COMPONENT ARCHITECTURE LAYER 430
OPERATING SYSTEM 420
HARDWARE 410
NETWORK COMPONENT
NETWORK COMPONENT
NETWORK COMPONENT
NETWORK COMPONENT
480
NETWORK COMPONENT LAYER
450

FIG. 4

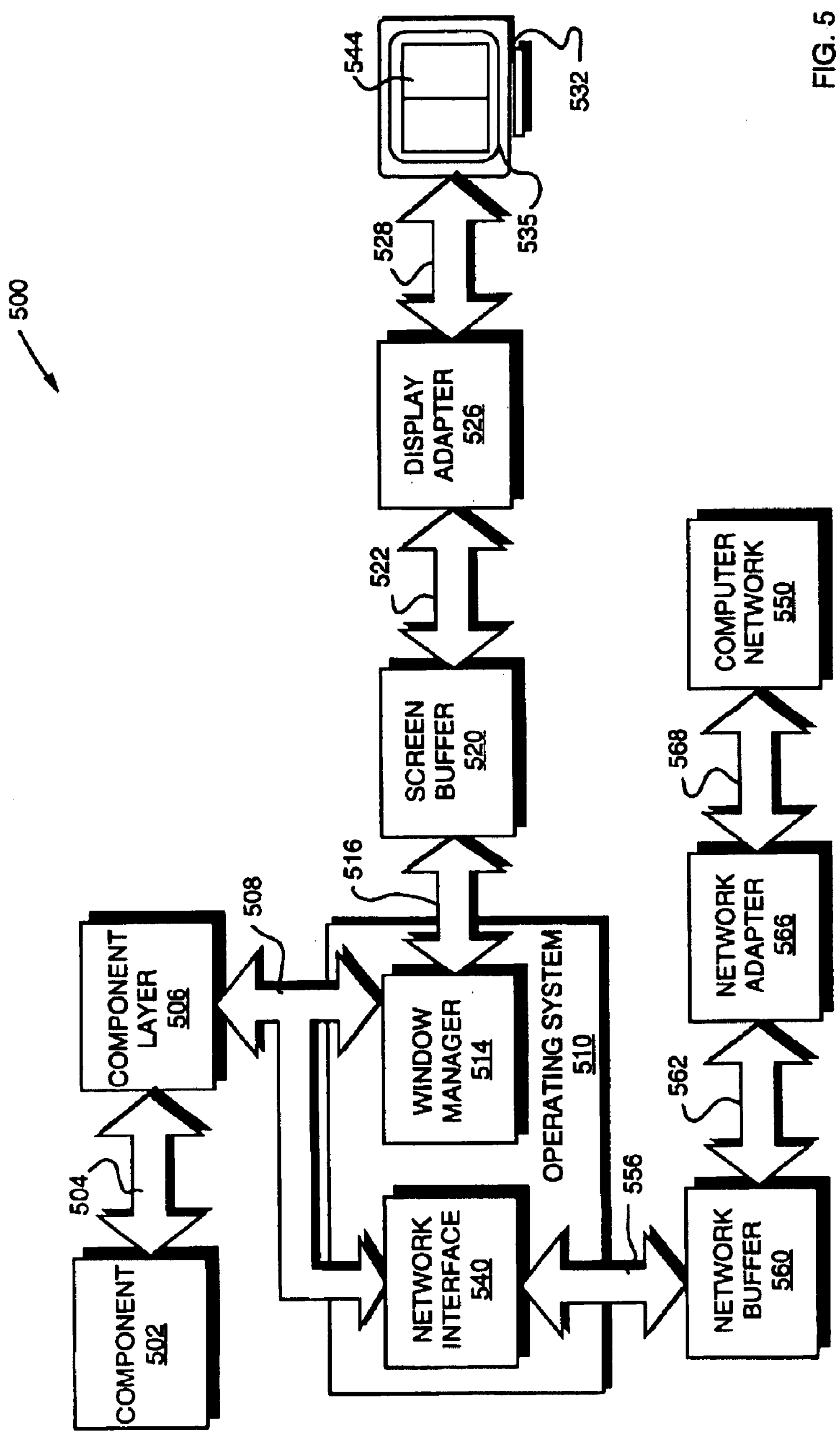

500
COMPONENT 502
504
COMPONENT LAYER 506
508
WINDOW MANAGER 514
NETWORK INTERFACE 540
OPERATING SYSTEM 510
516
556
SCREEN BUFFER 520
522
DISPLAY ADAPTER 526
528
535
544
532
NETWORK BUFFER 560
562
NETWORK ADAPTER 566
568
COMPUTER NETWORK 550

FIG. 5

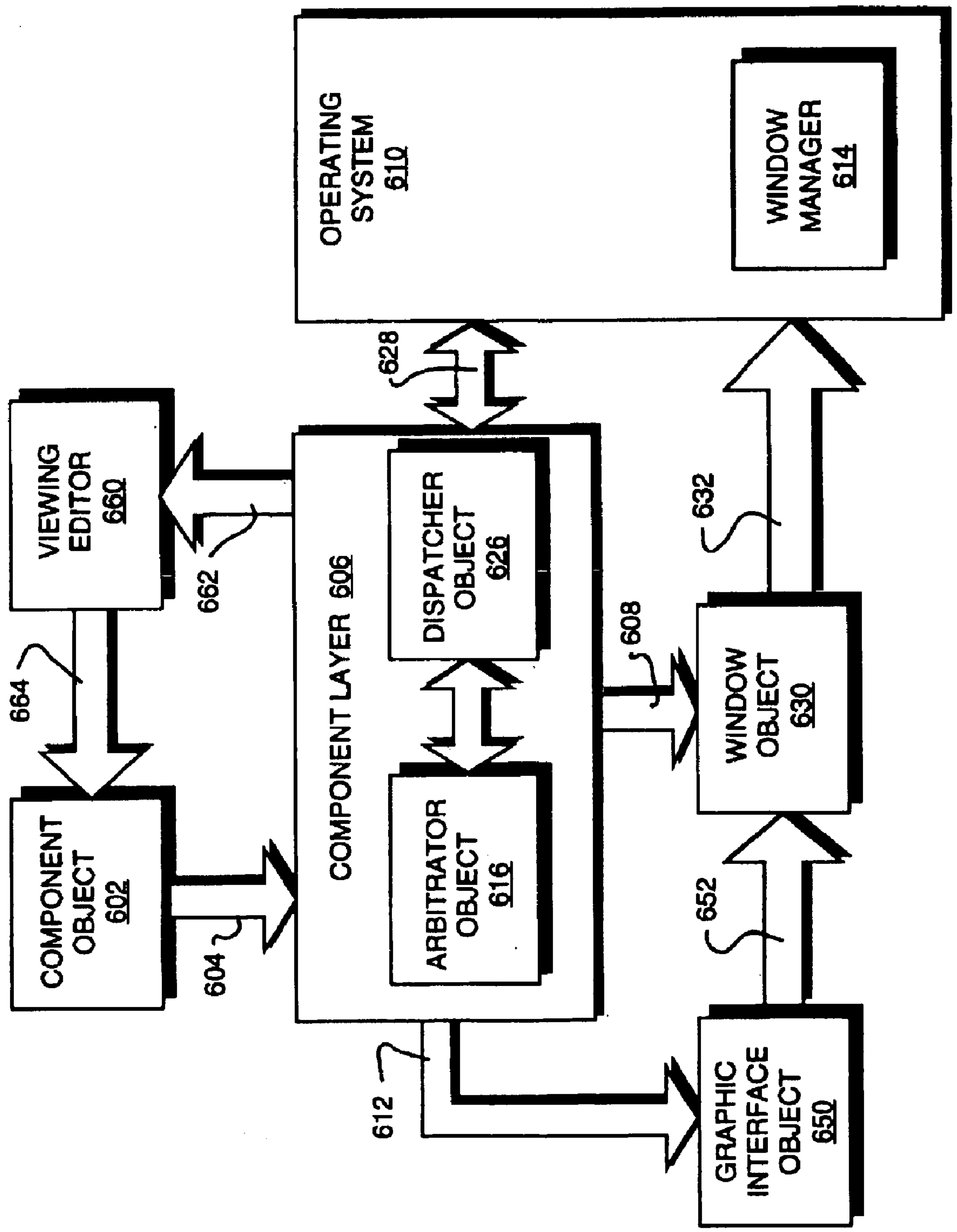

OPERATING SYSTEM 610
WINDOW MANAGER 614
VIEWING EDITOR 660
COMPONENT OBJECT 602
COMPONENT LAYER 606
DISPATCHER OBJECT 626
ARBITRATOR OBJECT 616
WINDOW OBJECT 630
GRAPHIC INTERFACE OBJECT 650
628
662
664
604
608
612
632
652

FIG. 6

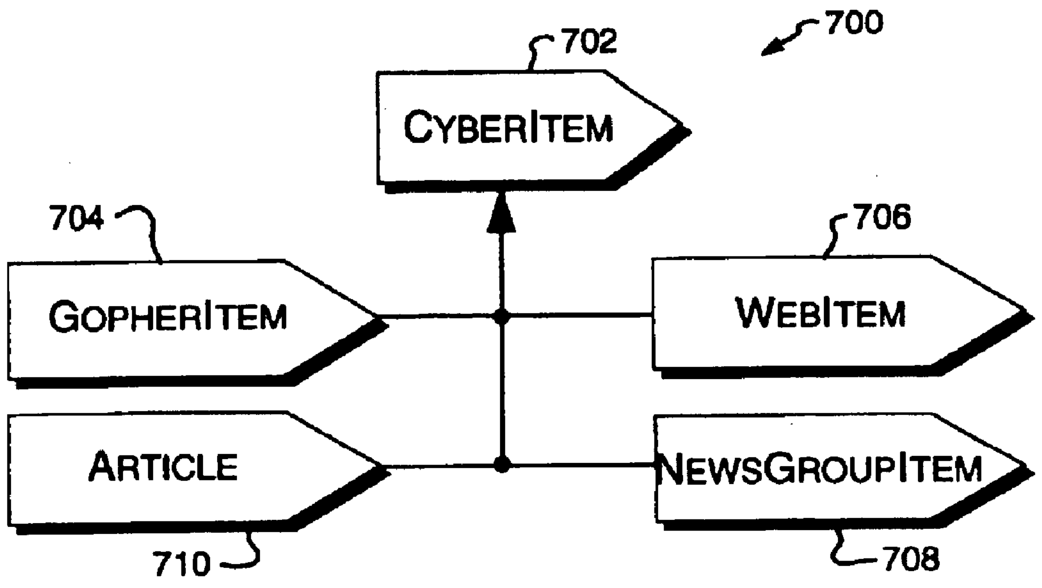

700
702
CYBERITEM
704
GOPHERITEM
706
WEBITEM
ARTICLE
NEWSGROUPITEM
710
708

FIG. 7

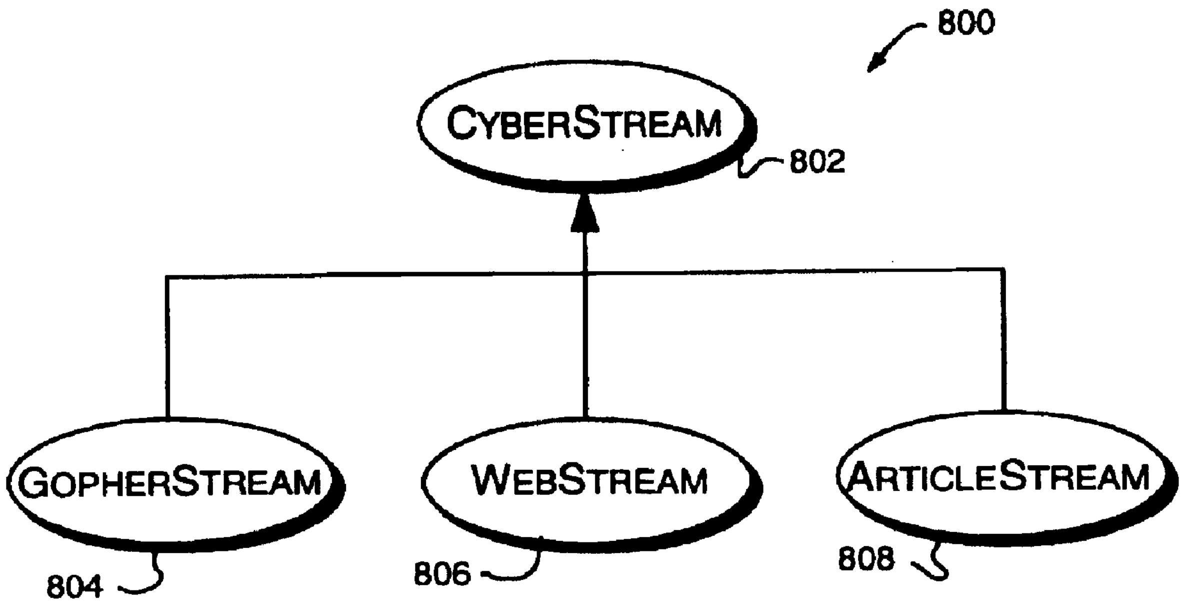

800
CYBERSTREAM
802
GOPHERSTREAM
WEBSTREAM
ARTICLESTREAM
804
806
808

FIG. 8

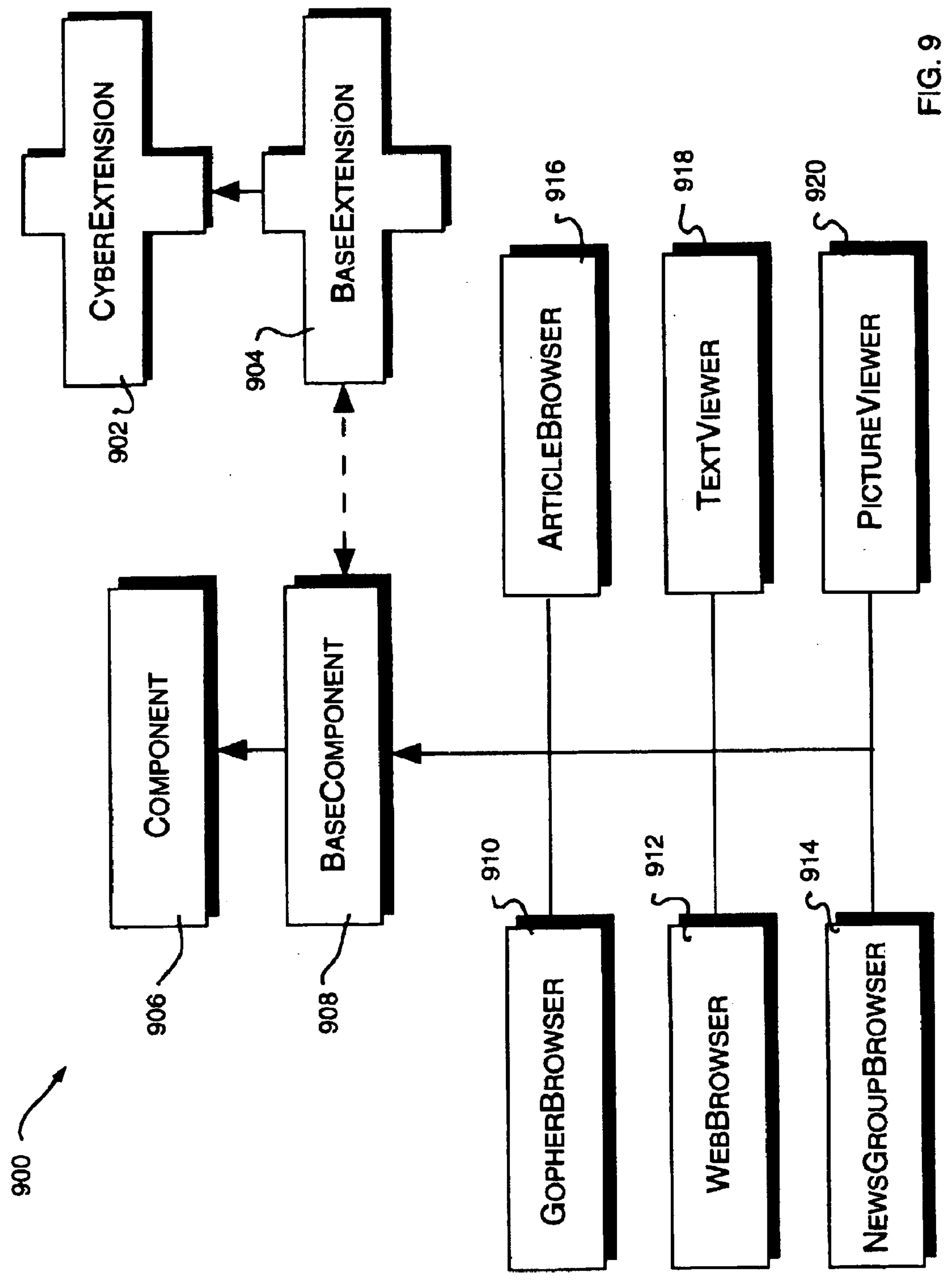

900
CyberExtension
902
BaseExtension
904
Component
906
BaseComponent
908
GopherBrowser
910
WebBrowser
912
NewsGroupBrowser
914
ArticleBrowser
916
TextViewer
918
PictureViewer
920

FIG. 9

EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM

Matter enclosed in heavy brackets [ ] appears in the original patent but forms no part of this reissue specification; matter printed in italics indicates the additions made by reissue.

CROSS REFERENCE TO RELATED APPLICATIONS

This invention is related to the following copending U.S. Patent Applications:

U.S. patent application Ser. No. 08/435,374, titled REPLACEABLE AND EXTENSIBLE NOTEBOOK COMPONENT OF A NETWORK COMPONENT SYSTEM.

U.S. patent application Ser. No. 08/435,862, titled REPLACEABLE AND EXTENSIBLE LOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,213, titled REPLACEABLE AND EXTENSIBLE CONNECTION DIALOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,671, titled EMBEDDING INTERNET BROWSER/BUTTONS WITHIN COMPONENTS OF A NETWORK COMPONENT SYSTEM; and

U. S. patent application Ser. No. 08/435,880, tilted ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM, each of which was filed on May 5, 1995 and assigned to the assignee of the present invention.

FIELD OF THE INVENTION

This invention relates generally to computer networks and, more particularly, to an architecture for building Internet-specific services.

BACKGROUND OF THE INVENTION

The Internet is a system of geographically distributed computer networks interconnected by computers executing networking protocols that allow users to interact and share information over the networks. Because of such wide-spread information sharing, the Internet has generally evolved into an "open" system for which developers can design software for performing specialized operations, or services, essentially without restriction. These services are typically implemented in accordance with a client/server architecture, wherein the clients, e.g., personal computers or workstations, are responsible for interacting with the users and the servers are computers configured to perform the services as directed by the clients.

Not surprisingly, each of the services available over the Internet is generally defined by its own networking protocol. A protocol is a set of rules governing the format and meaning of messages or "packets" exchanged over the networks. By implementing services in accordance with the protocols, computers cooperate to perform various operations, or similar operations in various ways, for users wishing to "interact" with the networks. The services typically range from browsing or searching the information-having a particular data format using a particular protocol to actually acquiring information of a different format in accordance with a different protocol.

For example, the file transfer protocol (FTP) service facilitates the transfer and sharing of files across the Internet. The Telnet service allows users to log onto computers coupled to the networks, while the network protocol provides a bulletin-board service to its subscribers. Furthermore, the various data formats of the information available on the Internet include JPEG images, MPEG movies and μ-law sound files.

Coincided with the design of these services has been the development of applications for implementing the services on the client/server architecture. Accordingly, applications are available for users to obtain files from computers connected to the Internet using the FTP protocol. Similarly, individual applications allow users to log into remote computers (as though they were logging in from terminals attached to those computers) using the Telnet protocol and, further, to view JPEG images and MPEG movies. As a result, there exists a proliferation of applications directed to user activity on the Internet.

A problem with this vast collection of application-specific protocols is that these applications are generally unorganized, thus requiring users to plod through them in order to satisfyingly, and profitably, utilize the Internet. Such lack of uniformity is time consuming and disorienting to users that want to access particular types of information but are forced to use unfamiliar applications. Because of the enormous amount of different types of information available on the Internet and the variety of applications needed to access those information types, the experience of using the Internet may be burdensome to these users.

An alternative to the assortment of open applications for accessing information on the Internet is a "closed" application system, such as Prodigy, CompuServe or America Online. Each of these systems provide a fill range of well-organized services to their subscribers; however, they also impose restrictions on the services developers can offer for their systems. Such constraint of "new" service development may be an unreasonable alternative for many users.

Two fashionable services for accessing information over the Internet are Gopher and the World-Wide Web ("Web"). Gopher consists of a series of Internet servers that provide a "list-oriented" interface to information available on the networks, the information is displayed as menu items in a hierarchical manner. Included in the hierarchy of menus are documents, which can be displayed or saved, and searchable indexes, which allow users to type keywords and perform searches.

Some of the menu items displayed by Gopher are links to information available on other servers located on the networks. In this case, the user is presented with a list of available information documents that can be opened. The opened documents may display additional lists or they may contain various data-types, such as pictures or text, occasionally, the opened documents may "transport" the user to another computer on the Internet.

The other popular information services on the Internet is the Web. Instead of providing a user with a hierarchical list-oriented view of information, the Web provides the user with a "linked-hypertext" view. Metaphorically, the Web perceives the Internet as a vast book of pages, each of which may contain pictures, text, sound, movies or various other types of data in the form of documents. Web documents are written in HyperText Markup Language (HTML) and Web servers transfer HTML documents to each other through the HyperText Transfer Protocol (HTTP).

The Web service is essentially a means for naming sources of information on the Internet. Armed with such a general naming convention that spans the entire network system, developers are able to build information servers that potentially any user can access. Accordingly, Gopher servers, HTTP servers, FTP servers, and E-mail servers have been developed for the Web. Moreover, the naming convention enables users to identify resources (such as directories and documents) on any of these servers connected to the Internet and allow access to those resources.

As an example, a user "traverses" the Web by following hot items of a page displayed on a graphical Web browser. These hot items are hypertext links whose presence are indicated on the page by visual cues, e.g., underlined words, icons or buttons. When a user follows a link (usually by clicking on the cue with a mouse), the browser displays the target pointed to by the link which, in some cases, may be another HTML document.

The Gopher and Web information services represent entirely different approaches to interacting with information on the Internet. One follows a list-approach to information that "looks" like a telephone directory service, while the other assumes a page-approach analogous to a tabloid newspaper. However, both of these approaches include applications for enabling users to browse information available on Internet servers. Additionally, each of these applications has a unique way of viewing and accessing the information on the servers.

Netscape Navigator™ ("Netscape")is an example of a monolithic Web browser application that is configured to interact with many of the previously-described protocols, including HTFF, Gopher and FTP. When instructed to invoke an application that uses one of these protocols, Netscape "translates" the protocol to hypertext. This translation places the user farther away from the protocol designed to run the application and, in some cases, actually thwarts the user's Internet experience. For example, a discussion system requiring an interactive exchange between participants may be bogged down by hypertext translations.

The Gopher and Web services may further require additional applications to perform specific functions, such as playing sound or viewing movies, with respect to the data types contained in the documents. For example, Netscape employs helper applications for executing applications having data formats it does not "understand". Execution of these functions on a computer requires interruption of processing and context switching (i.e., saving of state) prior to invoking the appropriate applications. Thus, if a user operating within the Netscape application "opens" an MPEG movie, that browsing application number must be saved (e.g., to disk) prior to opening an appropriate MPEG application, e.g., Sparkle, to view the image. Such an arrangement is inefficient and rather disruptive to processing operations of the computer.

Typically, a computer includes an operating system and application software which, collectively, control the operations of the computer. The applications are preferably task-specific and independent, e.g., a word processor application edits text, a drawing application edits drawings and a database application interacts with information stored on a database storage unit. Although a user can move data from one application to the other, such as by copying a drawing into a word processing file, the independent applications must be invoked to thereafter manipulate that data.

Generally, the application program presents information to a user through a window of a graphical user interface by drawing images, graphics or text within the window region. The user, in turn, communicates with the application by "pointing" at graphical objects in the window with a pointer

that is controlled by a hand-operated pointing device, such as a mouse, or by pressing keys of a keyboard.

The graphical objects typically included with each window region are sizing boxes, buttons and scroll bars. These objects represent user interface elements that the user can point at with the pointer (or a cursor) to select or manipulate. For example, the user may manipulate these elements to move the windows around on the display screen, and change their sizes and appearances so as to arrange the window in a convenient manner. When the elements are selected or manipulated, the underlying application program is informed, via the window environment, that control has been appropriated by the user.

A menu bar is a further example of a user interface element that provides a list of menus available to a user. Each menu, in turn, provides a list of command options that can be selected merely by pointing to them with the mouse-controlled pointer. That is, the commands may be issued by actuating the mouse to move the pointer onto or near the command selection, and pressing and quickly releasing, i.e., "clicking" a button on the mouse.

In contrast to this typical application-based computing environment, a software component architecture provides a modular document-based computing arrangement using tools such as viewing editors. The key to document-based computing is the compound document, i.e., a document composed of many different types of data sharing the same file. The types of data contained in a compound document may range from text, tables and graphics to video and sound. Several editors, each designed to handle a particular data type of format, can work on the contents of the document at the same time, unlike the application-based computing environment.

Since many editors may work together on the same document, the compound document is apportioned into individual modules of context for manipulation by the editors. The compound-nature of the document is realized by embedding these modules within each other to create a document having a mixture of data types. The software component architecture provides the foundation for assembling documents of differing contents and the present invention is directed to a system for extending this capability to network-oriented services.

Therefore, it is among the objects of the present invention to simplify a user's experience on computer networks without sacrificing the flexibility afforded the user by employing existing protocols and data types available on those networks.

Another object of the invention is to provide a system for users to search and access information on the Internet without extensive understanding or knowledge of the underlying protocols and data formats needed to access that information.

Still another object of the invention is to provide a document-based computing system that enables users to develop modules for services directed to information available on computer networks.

Still yet another object of the invention is to provide a platform that allows third-party developers to extend a layered network component system by building new components that seamlessly interact with the system components.

SUMMARY OF THE INVENTION

Briefly, the invention comprises an extensible and replaceable network-oriented component system that provides a platform for developing network navigation components that operate on a variety of hardware and software computer system. These navigation components include key integrating components along with components, such as Gopher-specific and Web-specific components, configured to deliver conventional services directed to computer networks. Communication among these components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a highly-modular cooperating layered-arrangement between the network component system and the component architecture allows any existing component to be replaced, and allows new components to be added, without affecting operation of the novel network component system.

According to one aspect of the present invention, the novel system provides a network navigating service for browsing and accessing information available on the computer networks. The information may include various data types available from a variety of sources coupled to the computer networks. Upon accessing the desired information, component viewing editors are provided to modify or display, either visually or acoustically, the contents of the data types regardless of the source of the underlying data. Additional components and component viewing editors may be created in connection with the underlying software component architecture to allow integration of different data types and protocols needed to interact with information on the Internet.

In accordance with another aspect of the invention, the component system is preferably embodied as a customized framework having a set of interconnected abstract classes for defining network-oriented objects. These abstract classes include CyberItem, CyberStream and CyberExtension, and the objects they define are used to build the novel navigation components. Interactions among these latter components and existing components of the underlying software architecture provide the basis for the extensibility and replaceability features of the network component system.

Specifically, CyberItem is an object abstraction which represents a "resource on a computer-network", but which may be further expanded to include resources available at any accessible location. CyberStream is an object abstraction representing a method for downloading information from a remote location on the computer network, while CyberExtension represents additional behaviors provided to the existing components for integration with the network component system.

The novel network system captures the essence of a "component-based" approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems. Such a component-based system has a number of advantages. First, if a user does not like the way a particular component operates, that component can be replaced with a different component provided by another developer. In contrast, if a user does not like the way a monolithic application handles certain protocols, the only resource is to use another service because the user cannot modify the application to perform the protocol function in a different manner. Clearly, the replaceability feature of the novel network component system provides a flexible alternative to the user.

Second, the use of components is substantially less disruptive than using helper applications in situations where a monolithic application confronts differing data types and formats. Instead of "switching" application layers, the novel

network system merely invokes the appropriate component and component viewing editor configured to operate with the data type and format. Such "seamless" integration among components is a significant feature of the modular cooperating architecture described herein.

A third advantage of the novel network system is directed to the cooperating relationship between the system and the underlying software computer architecture. Specifically, the novel network components are based on the component architecture technology to therefore ensure cooperation between all components in an integrated manner. The software component architecture is configured to operate on a plurality of computers, and is preferably implemented as a software layer adjoining the operating system.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which:

FIG. **1** is a block diagram of a network system including a collection of computer networks interconnected by client and server computers;

FIG. **2** is a block diagram of a client component, such as a personal computer, on which the invention may advantageously operate;

FIG. **3** is a block diagram of a server computer of FIG. **1**;

FIG. **4** is a highly schematized block diagram of a layered component computing arrangement in accordance with the invention;

FIG. **5** is a schematic illustration of the interaction of a component, a software component layer and an operating system of the computer of FIG. **2**;

FIG. **6** is a schematic illustration of the interaction between a component, a component layer and a window manager in accordance with the invention;

FIG. **7** is a simplified class hierarchy diagram illustrating a base class CyberItem, and its associated subclasses, used to construct network component objects in accordance with the invention;

FIG. **8** is a simplified class hierarchy diagram illustrating a base class CyberStream, and its associated subclasses, in accordance with the invention; and

FIG. **9** is a simplified class hierarchy diagram illustrating a base class CyberExtension, and its associated subclasses, in accordance with the present invention.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

FIG. **1** is a block diagram of a network system **100** comprising a collection of computer networks **110** interconnected by client computers ("clients") **200**, e.g., workstations or personal computers, and server computers ("servers") **300**. The servers are typically computers having hardware and software elements that provide resources or services for use by the clients **200** to increase the efficiency of their operations. It will be understood to those skilled in the art that, in an alternate embodiment, the client and server may exist on the same computer; however, for the illustrative embodiment described herein, the client and server are separate computers.

Several types of computer networks **110**, including local area networks (LANs) and wide area networks (WANs), may be employed in the system **100**. A LAN is a limited area network that typically consists of a transmission medium, such as coaxial cable or twisted pair, while a WAN may be a public or private telecommunications facility that interconnects computers widely dispersed. In the illustrative embodiment, the network system **100** is the Internet system of geographically distributed computer networks.

Computers coupled to the Internet typically communicate by exchanging discrete packets of information according to predefined networking protocols. Execution of these networking protocols allow users to interact and share information across the networks. As an illustration, in response to a user's request for a particular service, the client **200** sends an appropriate information packet to the server **300**, which performs the service and returns a result back to the client **200**.

FIG. **2** illustrates a typical hardware configuration of a client **200** comprising a central processing unit (CPU) **210** coupled between a memory **214** and input/output (I/O) circuitry **218** by bidirectional buses **212** and **216**. The memory **214** typically comprises random access memory (RAM) for temporary storage of information and read only memory (ROM) for permanent storage of the computer's configuration and basic operating commands, such as portions of an operating system (not shown). As described further herein, the operating system controls the operations of the CPU **210** and client computer **200**.

The I/O circuitry **218**, in turn, connects the computer to computer networks, such as the Internet computer networks **250**, via a bidirectional bus **222** and to cursor/pointer control devices, such as keyboard **224** (via cable **226**) and a mouse **230** (via cable **228**). The mouse **230** typically contains at least one button **234** operated by a user of the computer. A conventional display monitor **232** having a display screen **235** is also connected to I/O circuitry **218** via cable **238**. A pointer (cursor) **240** is displayed on windows **244** of the screen **235** and its position is controllable via the mouse **230** or the keyboard **224**, as is well-known. Typically, the I/O circuitry **218** receives information, such as control and data signals, from the mouse **230** and keyboard **224**, and provides that information to the CPU **210** for display on the screen **235** or, as described further herein, for transfer over the Internet **250**.

FIG. **3** illustrates a typical hardware configuration of a server **300** of the network system **100**. The server **300** has many of the same units as employed in the client **200**, including a CPU **310**, a memory **314**, and I/O circuitry **318**, each of which are interconnected by bidirectional buses **312** and **316**. Also, the I/O circuitry connects the computer to computer networks **350** via a bidirectional bus **322**. These units are configured to perform functions similar to those provided by their corresponding units in the computer **200**. In addition, the server typically includes a mass storage unit **320**, such as a disk drive, connected to the I/O circuitry **318** via bidirectional bus **324**.

It is to be understood that the I/O circuits within the computers **200** and **300** contain the necessary hardware, e.g., buffers and adapters, needed to interface with the control devices, the display monitor, the mass storage unit and the networks. Moreover, the operating system includes the necessary software drivers to control, e.g., network adapters within the I/O circuits when performing I/O operations, such as the transfer of data packets between the client **200** and server **300**.

The computers are preferably personal computers of the Macintosh® series of computers sold by Apple Computer Inc., although the invention may also be practiced in the context of other types of computers, including the IBM®G) series of computers sold by International Business Machines Corp. These computers have resident thereon, and are controlled and coordinated by, operating system software, such as the Apple® System 7®, IBM OS2®, or the Microsoft® Windows® operating systems.

As noted, the present invention is based on a modular document computing arrangement as provided by an underlying software components architecture, rather than the typical application-based environment of prior computing systems. FIG. **4** is a highly schematized diagram of the hardware and software elements of a layered component computing arrangement **400** that includes the novel network-oriented component system of the invention. At the lowest level there is the computer hardware, shown as layer **410**. Interfacing with the hardware is a conventional operating system layer **420** that includes a window manager, a graphic system, a file system and network-specific interfacing, such as a TCP/IP protocol stack and an Appletalk protocol stack.

The software component architecture is preferably implemented as a component architecture layer **430**. Although it is shown as overlaying the operating system **420**, the component architecture layer **430** is actually independent of the operating system and, more precisely, resides side-by-side with the operating system. This relationship allows the component architecture to exist on multiple platforms that employ different operating systems.

In accordance with the present invention, a novel network:oriented component layer **450** contains the underlying technology for implementing the extensible and replaceable network component system that delivers services and facilitates development of navigation components directed to computer networks, such as the Internet. As described further herein, this technology includes novel application programming interfaces (APIs) that facilitate communication among components to ensure integration with the underlying component architecture layer **430**. These novel APIs are preferably delivered in the form of objects in a class hierarchy.

It should be noted that the network component layer **450** may operate with any existing system-wide component architecture, such as the Object Linking and Embedding (OLE) architecture developed by the Microsoft Corporation; however, in the illustrative embodiment, the component architecture is preferably OpenDoc, the vendor-neutral, open standard for compound documents developed by, among others, Apple Computer, Inc.

Using tools such as viewing editors, the component architecture layer **430** creates a compound document composed of data having different types and formats. Each differing data type and format is contained in a fundamental unit called a computing part or, more generally, a "component" **460** comprised of a viewing editor along with the data content. An example of the computing component **460** may include a MacDraw component. The editor, on the other hand, is analogous to an application program in a conventional computer. That is, the editor is a software component which provides the necessary functionality to display a component's contents and, where appropriate, present a user interface for modifying those contents. Additionally, the editor may include menus, controls and other user interface elements.

According to the invention, the network component layer **450** extends the functionality of the underlying component architecture layer **430** by defining network-oriented components **480**. Included among these components are key integrating components (such as notebook, log and connection dialog components) along with components configured to deliver conventional services directed to computer networks, such as Gopher-specific and Web-specific components. Moreover, the components may include FTP-specific components for transferring files across the networks. Telnet-specific components for remotely logging onto other computers, and JPEG-specific and MPEG-specific components for viewing image and movie data types and formats.

A feature of the invention is the ability to easily extend, or replace, any of the components of the layered computing arrangement **400** with a different component to provide a user with customized network-related services. As described herein, this feature is made possible by the cooperating relationship between the network component layer **450** and its underlying component architecture layer **430**. The integrating components communicate and interact with these various components of the system in a "seamlessly integrated" manner to provide basic tools for navigating the Internet computer networks.

FIG. **4** also illustrates the relationship of applications **490** to the elements of the layered computing arrangement **400**. Although they reside in the same "user space" as the components **460** and network components **480**, the applications **490** do not interact with these elements and, thus, interface directly to the operating system layer **420**. Because they are designed as monolithic, autonomous modules, applications (such as previous Internet browsers) often do not even interact among themselves. In contrast, the components of the arrangement **400** are designed to work together via the common component architecture layer **430** or, in the case of the network components, via the novel network component layer **450**.

Specifically, the invention features the provision of the extensible and replaceable network-oriented component system which, when invoked, causes actions to take place that enhance the ability of a user to interact with the computer to search for, and obtain, information available over computer networks such as the Internet. The information is manifested to a user via a window environment, such as the graphical user interface provide by System 7 or Windows, that is preferably displayed on the screen **235** (FIG. **2**) as a graphical display to facilitate interactions between the user and the computer, such as the client **200**. This behavior of the system is brought about by the interaction of the network components with a series of system software routines associated with the operating system **420**. These system routines, in turn, interact with the components architecture layer **430** to create the windows and graphical user interface elements, as described further herein.

The window environment is generally part of the operating system software **420** that includes a collection of utility programs for controlling the operation of the computer **200**. The operating system, in turn, interacts with the components to provide higher levels functionality, including a direct interface with the user. A component makes use of operating system functions by issuing a series of task commands to the operating system via the network component layer **450** or, as is typically the case, through the component architecture layer **430**. The operating system **420** then performs the requested task. For example, the component may request that a software driver of the operating system initiate transfer of a data packet over the networks **250** or that the operating system display certain information on a window for presentation to the user.

FIG. **5** is a schematic illustration of the interaction of a component **502**, software component layer **506** and an operating system **510** of a computer **500**, which is similar to, and has equivalent elements of, the client computer **200** of FIG. **2**. As noted, the network component layer **450** (FIG. **4**) is integrated with the computer architecture layer **430** to provide a cooperating architecture that allows any component to be replaced or extended, and allows new components to be added, without affecting operation of the network component system, accordingly, for purposes of the present discussion, the layers **430** and **450** may be treated as a single software component layer **506**.

The component **502**, component layer **506** and operating system **510** interact to control and coordinate the operations of the computer **500** and their interaction is illustrated schematically by arrows **504** and **508**. In order to display information on a screen display **535**, the component **502** and component layer **506** cooperate to generate and send display commands to a window manager **514** of the operating system **510**. The window manager **514** stores information directly (via arrow **516**) into a screen buffer **520**.

The window manager **514** is a system software routine that is generally responsible for managing windows **544** that the user views during operation of the network component system. That is, it is generally the task of the window manager to keep track of the location and size of the window and window areas which must be drawn and redrawn in connection with the network component system of the present invention.

Under control of various hardware and software in the system, the contents of the screen buffer **520** are read out of the buffer and provided, as indicated schematically by arrow **522**, to a display adapter **526**. The display adapter contains hardware and software (sometimes in the form of firmware) which converts the information in the screen buffer **520** to a form which can be used to drive a display screen **535** of a monitor **532**. The monitor **532** is connected to display adapter **526** by cable **528**.

Similarly, in order to transfer information as a packet over the computer networks, the component **502** and component layer **506** cooperate to generate and send network commands, such as remote procedure calls, to a network-specific interface **540** of the operating system **510**. The network interface comprises system software routines, such as "stub" procedure software and protocol stacks, that are generally responsible for forming the information into a predetermined packet format according to the specific network protocol used, e.g., TCP/IP or Apple-talk protocol.

Specifically, the network interface **540** stores the packet directly (via arrow **556**) into a network buffer **560**. Under control of the hardware and software in the system, the contents of the network buffer **560** are provided, as indicated schematically by arrow **562**, to a network adapter **566**. The network adapter incorporates the software and hardware, i.e., electrical and mechanical interchange circuits and characteristics, needed to interface with the particular computer networks **550**. The adapter **566** is connected to the computer networks **550** by cable **568**.

In a preferred embodiment, the invention described herein is implemented in an object-oriented programming (OOP) language, such as C++, using System Object Model (SOM) technology and OOP techniques. The C++ and SOM languages are well-known and many articles and texts are available which describe the languages in detail. In addition, C++ and SOM compilers are commercially available from several vendors. Accordingly, for reasons of brevity, the details of the C++ and SOM languages and the operations of their compilers will not be discussed further in detail herein.

As will be understood by those skilled in the art, OOP techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity that can be created, used and deleted as if it were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data elements, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like computers, while also modeling abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct an actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a "constructor" which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a "destructor". Objects may be used by manipulating their data and invoking their functions.

The principle benefits of OOP techniques arise out of three basic principles encapsulation, polymorphism and inheritance. Specifically, objects can be designed to hide, or encapsulate all, or a portion of, its internal data structure and internal functions. More specifically, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables causes the compiler to generate an error during program compilation which error stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions that have the same overall format, but that work with different data, to function differently in order to produce consistent results. Inheritance, on the other hand, allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these functions appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective modification) of another class allows software developers to easily customize existing code to meet their particular needs.

In accordance with the present invention, the component **502** and windows **544** are "objects" created by the component layer **506** and the window manager **514**, respectively, the latter of which may be an object-oriented program. Interaction between a component, component layer and a window manager is illustrative in greater detail in FIG. **6**.

In general, the component layer **606** interfaces with the window manager **614** by creating and manipulating objects. The window manger itself may be an object which is created when the operating system is started. Specifically, the component layer creates window objects **630** that create the window manager to create associated windows on the display screen. This is shown schematically by an arrow **608**. In addition, the component layer **606** creates individual graphic interface objects **650** that are stored in each window object **630**, as shown schematically by arrows **612** and **652**. Since many graphic interface objects may be created in order to display many interface elements on the display screen, the window object **630** communicates with the window manager by means of a sequence of drawing commands issued from the window object to the window manager **614**, as illustrated by arrow **632**.

As noted, the component layer **606** functions to embed components within one another to form a component document having mixed data types and formats. Many different viewing editors may work together to display, or modify, the data contents of the documents. In order to direct keystrokes and mouse events initiated by a user to the proper components and editors, the component layer **606** includes an arbitrator **616** and a dispatcher **626**.

The dispatcher is an object that communicates with the operating system **610** to identify the correct viewing editor **660**, while the arbitrator is an object that informs the dispatcher as to which editor "owns" the stream of keystrokes or mouse events. Specifically, the dispatcher **626** receives these "human-interface" events from the operating system **610** (as shown schematically by arrow **628**) and delivers them to the correct viewing editor **660** via arrow **662**. The viewing editor **660** then modifies or displays, either visually or acoustically, the contents of the data types.

Although OOP offers significant improvements over other programming concepts, software development still requires significant outlays of time and effort, especially if no pre-existing software is available for modulation. Consequently, a prior art approach has been to provide a developer with a set of preferred, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such predefined classes and libraries are typically called "frameworks" and essentially provide a pre-fabricated structure for a working document.

For example, a framework for a user interface might provide a set of predefined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these interface objects. Since frameworks are based on object-oriented techniques, the predefined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of that original program. In addition, developers are not blindly working through layers of code because the framework providers architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of framework available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application-type frameworks include MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXT Step App Kit (NeXT) and Smalltalk-80 MVC (ParcPlace).

While the framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. These difficulties are caused by the fact that it is easy for developers to reuse their own objects, but it is difficult for the developers to use objects generated by other programs. Further, frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying system by means of awkward procedure calls.

In the same way that a framework provides the developer with prefab functionality for a document, a system framework, such as that included in the preferred embodiment, can provide a prefab functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art framework. For example, consider a customizable network interface framework which can provide the foundation for browsing and accessing information over a computer network. A software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristic and behavior of the finished output, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the document, component, component layer and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework, such as MacApp, can be leveraged not only at the application level for things such as text and graphical user interfaces, but also at the system level for such services as printing, graphics, multi-media, file systems and, as described herein, network-specific operations.

Referring again to FIG. **6**, the window object **630** and the graphic interface object **650** are elements of a graphical user interface of a network component system having a customizable framework for greater enhancing the ability of a user to navigate or browse through information stored on servers coupled to the network. Moreover, the novel network system provides a platform for developing network navigation components for operation on a variety of hardware and software computer systems.

As noted, the network components are preferably implemented as objects and communication among the network component objects is effected through novel application programming interfaces (APIs). These APIs are preferably delivered in the form of objects in a class hierarchy that is extensible so that developers can create new components and editors. From an implementation viewpoint, the objects can be subclassed and can inherit from base classes to build

customized components allow users to see different kinds of data using different kinds of protocols, or to create components that function differently from existing components.

In accordance with the invention, the customized framework has a set of interconnected abstract classes for defining network-oriented objects used to build these customized network components. These abstract classes include CyberItem, CyberStream and CyberExtension and the objects they define are used to build the novel network components. Interactions among these latter components and existing components of the underlying software architecture provide the basis for the extensibility and replaceability features of the network component system.

In order to further understand the operations of these network component objects, it may be useful to examine their construction together with the major function routines that comprise the behavior of the objects. In examining the objects, it is also useful to examine the classes that are used to construct the objects (as previously mentioned the classes serve as templates for the construction of objects). Thus, the relation of the classes and the functions inherent in each class can be used to predict the behavior of an object once it is constructed.

FIG. **7** illustrates a simplified class hierarchy diagram **700** of the base class CyberItem **702** used to construct the network component object **602**. In general, CyberItem is an abstraction that may represent resources available at any location accessible from the client **200**. However, in accordance with the illustrative embodiment, a CyberItem is preferably a small, self-contained object that represents a resource, such as a service, available on the Internet and subclasses of the CyberItem base class are used to construct various network component objects configured to provide such services for the novel network-oriented component system.

For example, the class GopherItem **704** may be used to construct a network component object representing a "thing in Gopher space", such as a Gopher directory, while the subclass WebItem **706** is derived from CyberItem and encapsulates a network component object representing a "thing in Web space, e.g. a Web page. Similarly, the subclass NewsGroupItem **708** may be used to construct a network object representing a newsgroup and the class Article **710** is configured to encapsulate a network component object representing an article resource on an Internet server.

Since each of the classes used to construct these network component objects are subclasses of the CyberItem base class, each class inherits the functional operators and methods that are available from that base class. For example, methods associated with the CyberItem base class for returning an icon family and a name are assumed by the subclasses to allow the network components to display CyberItem objects in a consistent manner. The methods associated with the CyberItem base class include (the arguments have been omitted for simplicity):

```
GetRefCount ();
IncrementRefCount ();
Release ();
SetUpFromURL ();
ExternalizeContent ();
StreamToStorageUnit ();
StreamFromStorageUnit ();
Clone ();
Compare ();
GetStringProperty ();
```

```
SetDefaultName ();
GetURL ();
GetIconSuite ();
CreateCyberStream ();
Open ();
OpenInFrame ();
FindWindow ().
```

In some instances, a CyberItem object may need to spawn a CyberStream object in order to obtain the actual data for the object it represents. FIG. **8** illustrates a simplified class hierarchy diagram **800** of the base class CyberStream **802**. As noted, CyberStream is an abstraction that serves as an API between a component configured to display a particular data format and the method for obtaining the actual data. This allows developers to design viewing editors that can display the content of data regardless of the protocol required to obtain that data.

For example, a developer may design a picture viewing editor that uses the CyberStream API to obtain data bytes describing a picture. The actual data bytes are obtained by a subclass of CyberStream configured to construct a component object that implements a particular protocol, such as Gopher and HTTP. That is, the CyberStream object contains the software commands necessary to create a "data stream" for transferring information from one object to another. According to the invention, a GopherStream subclass **804** is derived from the CyberStream base class and encapsulates a network object that implements the Gopher protocol, while the class WebStream **806** may be used to construct a network component configured to operate in accordance with the HTTP protocol.

The methods associated with the CyberStream class, and which are contained in the objects constructed from the subclasses, include (the arguments have been omitted for simplicity):

```
GetStreamStatus ();
GetTotalDataSize ();
GetStreamError ();
GetStatusString ();
OpenWithCallback ();
Open ();
GetBuffer ();
ReleaseBuffer ();
Abort ().
```

FIG. **9** is a simplified class hierarchy diagram of the base class CyberExtension **902** which represents additional behaviors provided to components of the underlying software component architecture. Specifically, CyberExtension are the mechanisms for adding functionality to, and extending the APIs of, existing components so that they may communicate with the novel network components. As a result, the CyberExtension base class **902** operates in connection with a Component base class **906** through their respective subclasses BaseExtension **904** and BaseComponent **908**.

The CyberExtension base class provides an API for accessing other network-specific components, such as notebooks and logs, and for supporting graphical user interface elements, such as menus. CyberExtension objects are used by components that display the contents of CyterItem objects. This includes browser-like components such as a Gopher browser or Web browser, as well as JPEG-specific components which display particular types of data such as pictures. The CyberExtension objects also keep track of the CyberItem objects which these components are responsible for displaying.

In accordance with the invention, the class GopherBrowser **910** may be used to construct a Gopher-like network browsing component and the class WebBrowser **912** may be able to construct a Web-like network browsing component. Likewise, a TextViewer subclass **918** may encapsulate a network component configured to display text and a PictureViewer subclass **920** may construct a component for displaying pictures. The methods associated with the CyberExtension class include (the arguments have been omitted for simplicity):

```
ICyberExtension ( );
Components displaying the contents of CyberItem object

SetCyberItem ( );
GetCyberItem ( );
GetCyberItemWindow ( );
IsCyberItemSelected ( );
GetSelectedCyberItems ( );
Notebook and Log Tools

AddCyberItemToLog ( );
ShowLogWindow ( );
IsLogWindowShown ( );
AddCyberItemToNotebook ( );
AddCyberItemsToNotebook ( );
ShowNotebookWindow ( );
IsNotebookWindowShown ( );
SetLogFinger ( );
ClearLogFinger ( );
Notebook and Log Menu Handlers

InstallServicesMenu ( );
AdjustMenus ( );
DoCommand ( ).
```

In summary, the novel network system described herein captures, the essence of a "comprehensive-based" approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems. Advantages of such a component-based system include the ability to easily replace and extend components because of the cooperating relationship between the novel network-oriented component system and the underlying component architecture. This relationship also facilitates "seamless" integration and cooperation between components and component viewing editors when confronted with differing data types and formats.

While there has been shown and described an illustrative embodiment for implementing an extensible and replaceable network component system, it is to be understood that various other adaptations and modifications may be made within the spirit and scope of the invention. For example, additional system software routines may be used when implementing the invention in various applications. These additional system routines include dynamic link libraries (DLL), which are program files containing collections of window environment and networking functions designed to perform specific classes of operations. These functions are invoked as needed by the software component layer to perform the desired operations. Specifically, DLLs, which are generally well-known, may be used to interact with the component layer and window manager to provide network-specific components and functions.

The foregoing description has been directed to specific embodiments of this invention. It will be apparent, however, that other variations and modifications may be made to the described embodiments, with the attainment of some or all of their advantages. Therefor, it is the object of the appended claims to cover all such variations and modifications as come within the true spirit and scope of the invention.

What is claimed is:

1. An extensible and replaceable layered component computing arrangement residing on a computer coupled to a computer network, the layered arrangement comprising:

a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components; and

a network component layer for developing network navigation components that provide services directed to the computer network, the network component layer includes application programming interfaces; and

a first class included in the application programming interfaces to construct a first network navigation object that represents different network resources available on the computer network, wherein the network component layer coupled to the software component architecture layer in integrating relation to facilitate communication among the computing and network navigation components.

2. The computing arrangement of claim **1** wherein the network navigation components are objects.

3. The computing arrangement of claim **1** wherein the application programming interfaces further comprise a second class for constructing a second network navigation object representing a data stream for transferring information among objects of the arrangement.

4. The computing arrangement of claim **3** wherein the first network navigation object is an Item object and the second network navigation object is a Stream object, and wherein the Item object spawns the Stream object to obtain information from the network resource that the Item object represents.

5. The computing arrangement of claim **3** wherein the application programming interfaces further comprise a third class for constructing a third network navigation object representing additional behaviors provided to computing components of the software component architecture layer to thereby enable communication between the computing components and the network navigation components.

6. An extensible and replaceable layered component computing arrangement for providing services directed to information available on computer networks, the computing arrangement comprising:

a processor;

an operating system;

a software component architecture layer coupled to the operating system to control the operations of the processor, the software component architecture layer defining a plurality of computing components; and

a network component layer for creating network navigation components configured to search and obtain information available on the computer networks, the network component layer includes application programming interfaces; and

means for constructing a network navigation component that represents different resources available on the computer network, wherein the network component layer is integrally coupled to the software component architecture layer to ensure communication among the computing and network navigation components.

7. The computing arrangement of claim **6** wherein the network component layer and software component architecture layer comprise means for embedding components within one another to form a compound document having mixed data types and formats.

8. The computing arrangement of claim **6** wherein the application programming interfaces comprise means for constructing a network navigation component that implements a protocol.

9. The computing arrangement of claim **6** wherein the application programming interfaces comprise means for constructing a network navigation component that provides additional functionality to existing computing components to enable communication among the components.

10. The computing arrangement of claim **9** wherein the computing component comprises a computing part having a viewing editor and data content.

11. The computing arrangement of claim **10** wherein the computing component functions to one of transfer files over the networks, remotely log onto another computer coupled to the networks and view images on a screen of the computing arrangement.

12. The computing arrangement of claim **10** wherein the network navigation component comprises a browsing component.

13. The computing arrangement of claim **10** wherein the network navigation component comprises a component for one of displaying text and displaying movies on a screen of the computing arrangement.

14. An extensible and replaceable layered component computing arrangement residing on a computer adapted to be coupled on a computer network, the layered arrangement comprising:

a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components;

a network component layer adapted to be coupled to at least one network navigation component that provides a service directed to the computer network, the network component layer including an application programming interface; and

a number of interconnected abstract classes included in the application programming interface, at least on abstract class for defining a network navigation object that represents a resource available on the computer network, the network component layer coupled to the software component architecture layer to facilitate communication among the network navigation component and at least one computing component.

15. The layered arrangement of claim 14, wherein the abstract class defines a network navigation object that represents a method of downloading information from a remote location on the computer network.

16. The layered arrangement of claim 14, wherein the abstract class defines a network navigation object that represents additional behaviors provided to the computing components of the software component architecture layer for integrating with the network component layer.

17. The layered arrangement of claim 14, wherein the network navigation object is adapted to browse the computer network.

18. The layered arrangement of claim 14, wherein the network navigation object is adapted to display text on a computer display.

19. The layered arrangement of claim 14, wherein the network navigation object is adapted to display images on a computer display.

20. The layered arrangement of claim 14, wherein the network navigation object includes software commands for creating a datastream for transferring information between objects in the layered component computing arrangement.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: RE 39,486 E	Page 1 of 1
APPLICATION NO.	: 10/408789	
DATED	: February 6, 2007	
INVENTOR(S)	: Michael A. Cleron, Stephen Fisher and Timo Bruck	

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 18, Claim 14, Line 15, please change "on" to -- one --.

Signed and Sealed this

Second Day of September, 2008

Jon W. Dudas

JON W. DUDAS
Director of the United States Patent and Trademark Office

EXHIBIT 12

US005481721A

United States Patent [19]

Serlet et al.

[11] **Patent Number: 5,481,721**

[45] **Date of Patent: Jan. 2, 1996**

[54] **METHOD FOR PROVIDING AUTOMATIC AND DYNAMIC TRANSLATION OF OBJECT ORIENTED PROGRAMMING LANGUAGE-BASED MESSAGE PASSING INTO OPERATION SYSTEM MESSAGE PASSING USING PROXY OBJECTS**

[75] Inventors: **Bertrand Serlet**; **Lee Boynton**, both of Palo Alto; **Avadis Tevanian**, Mountain View, all of Calif.

[73] Assignee: **NeXT Computer, Inc.**, Redwood City, Calif.

[21] Appl. No.: **332,486**

[22] Filed: **Oct. 31, 1994**

Related U.S. Application Data

[63] Continuation of Ser. No. 731,636, Jul. 17, 1991, abandoned.

[51] **Int. Cl.**^{6} **G06F 9/44**

[52] **U.S. Cl.** **395/700**; 364/DIG. 1; 364/280; 364/284.3; 364/284

[58] **Field of Search** .... 395/700, 650; 364/DIG. 1, DIG. 2

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,060,150	10/1991	Simor	364/200
5,230,051	7/1993	Quan	395/700
5,305,461	4/1994	Feigenbaum et al.	395/775

OTHER PUBLICATIONS

Bennet, J. K., "The design and implementation of Distributed Smalltalk", SIGPLAN Notices, vol. 22, No. 12, pp. 318–320, OOPSLA '87 Proceedings, Dec. 1987.

McCullough, P. L., "Transparent Forwarding: First Steps", SIGPLAN Notices, vol. 22, No. 12, pp. 331–341, OOPSLA '87 Proceedings, Dec. 1987.

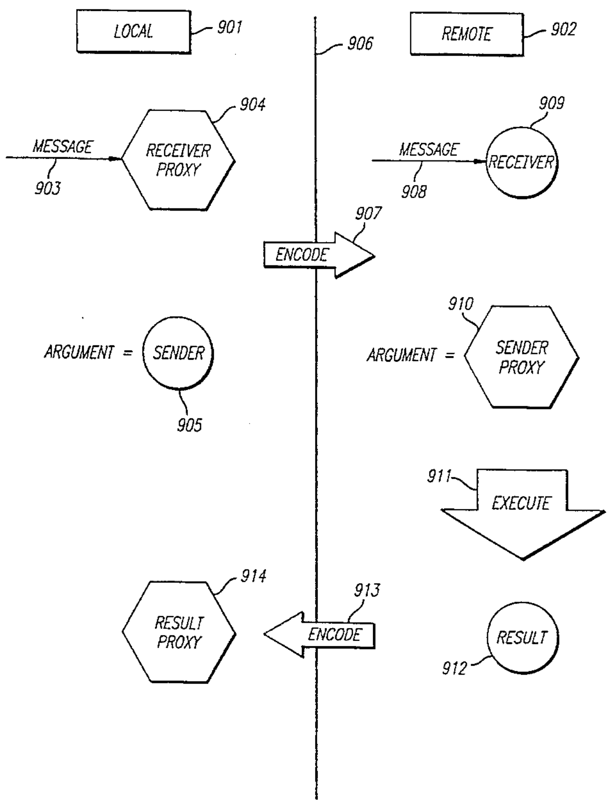
Shapiro, M., "The Design of a Distributed Object–Oriented Operating System For Office Applications", ESPRIT '88. Putting the Technology to Use Proceedings of the 5th Annual ESPRIT Conference, pp. 1020–1027, vol. 2, Nov. 1988.

Primary Examiner—Kevin A. Kriess
Attorney, Agent, or Firm—Hecker & Harriman

[57] **ABSTRACT**

The present invention provides a method and apparatus for the distribution of objects and the sending of messages between objects that are located in different processes. Initially, a "proxy" object is created in the same process as a sender object. This proxy acts as a local receiver for all objects in the local program. When the proxy receives a message, the message is encoded and transmitted between programs as a stream of bytes. In the remote process, the message is decoded and executed as if the sender was remote. The result follows the same path, encoded, transmitted, and then decoded back in the local process. The result is then provided to the sending object.

24 Claims, 6 Drawing Sheets



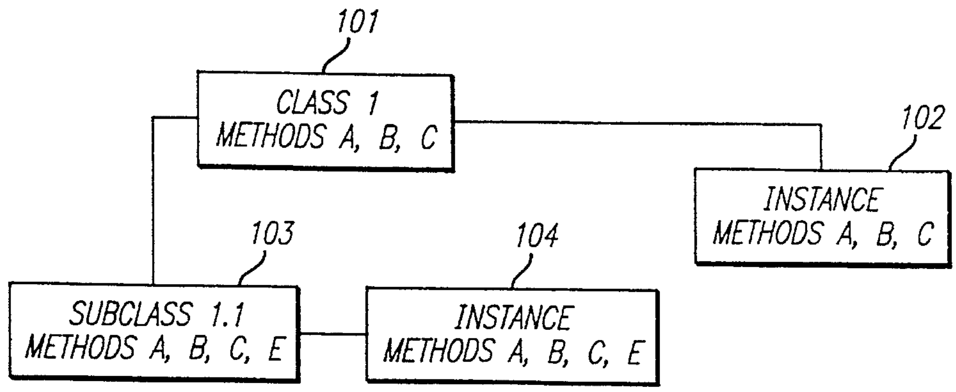


FIG. 1 PRIOR ART

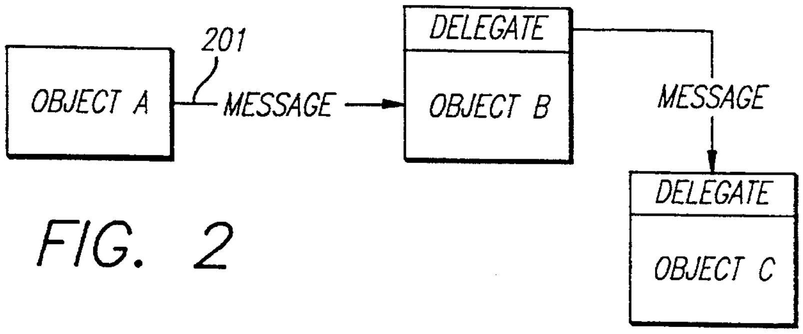


FIG. 2

FIG. 4

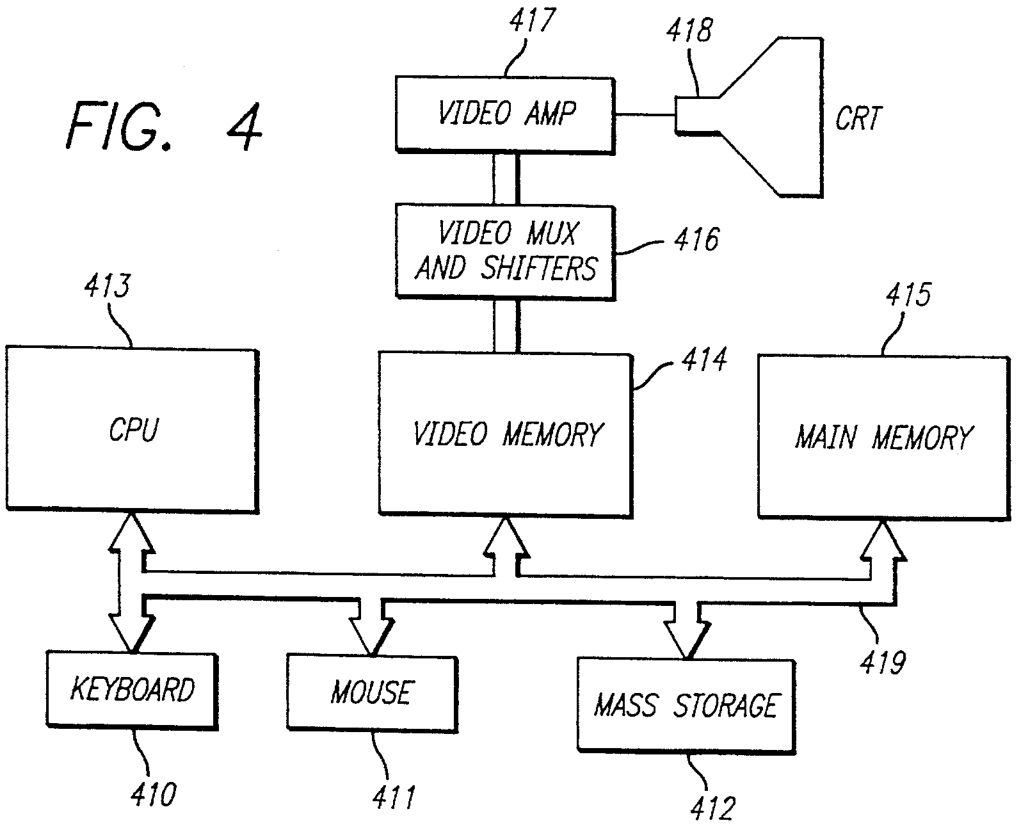


FIG. 3A

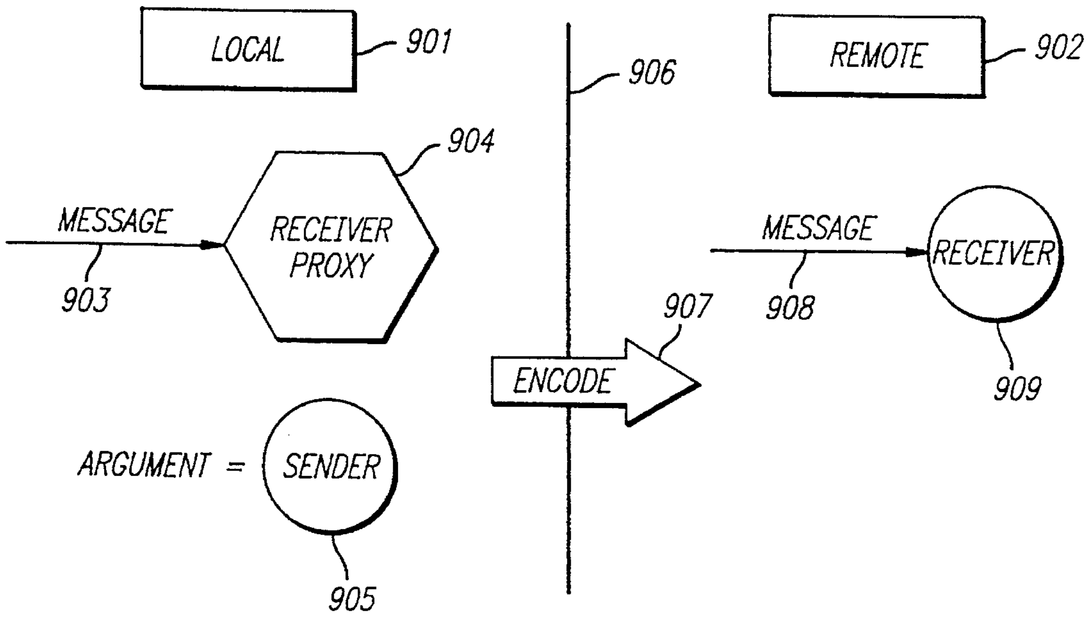

LOCAL
901
906
REMOTE
902
904
MESSAGE
RECEIVER PROXY
903
MESSAGE
RECEIVER
907
908
ENCODE
909
ARGUMENT =
SENDER
905

FIG. 3B

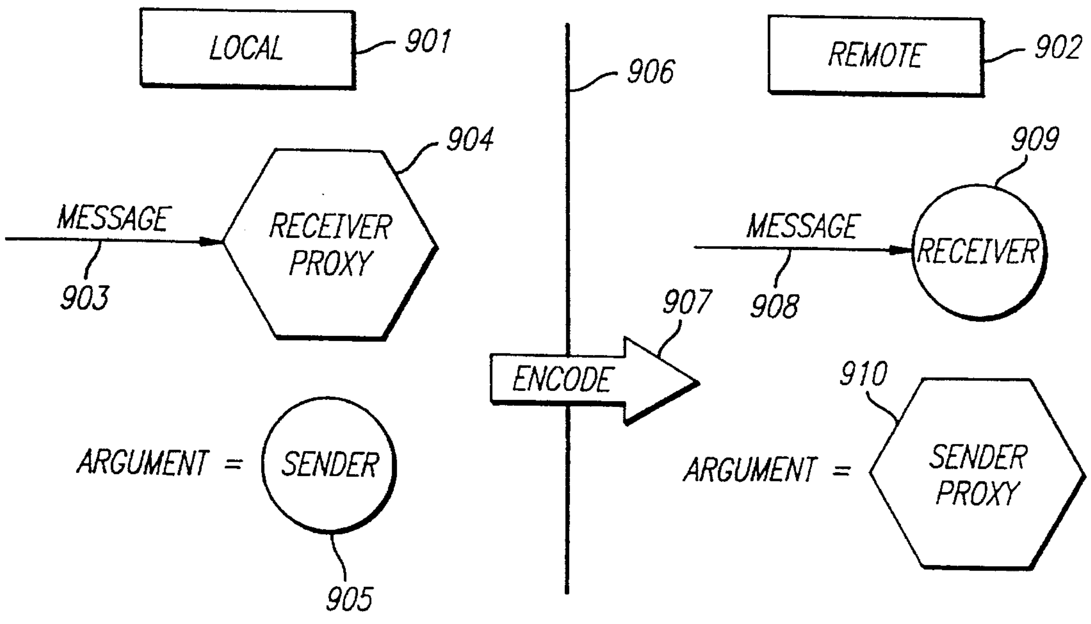

LOCAL
901
906
REMOTE
902
904
909
MESSAGE
RECEIVER PROXY
903
MESSAGE
RECEIVER
907
908
ENCODE
910
ARGUMENT =
SENDER
ARGUMENT =
SENDER PROXY
905

FIG. 3C

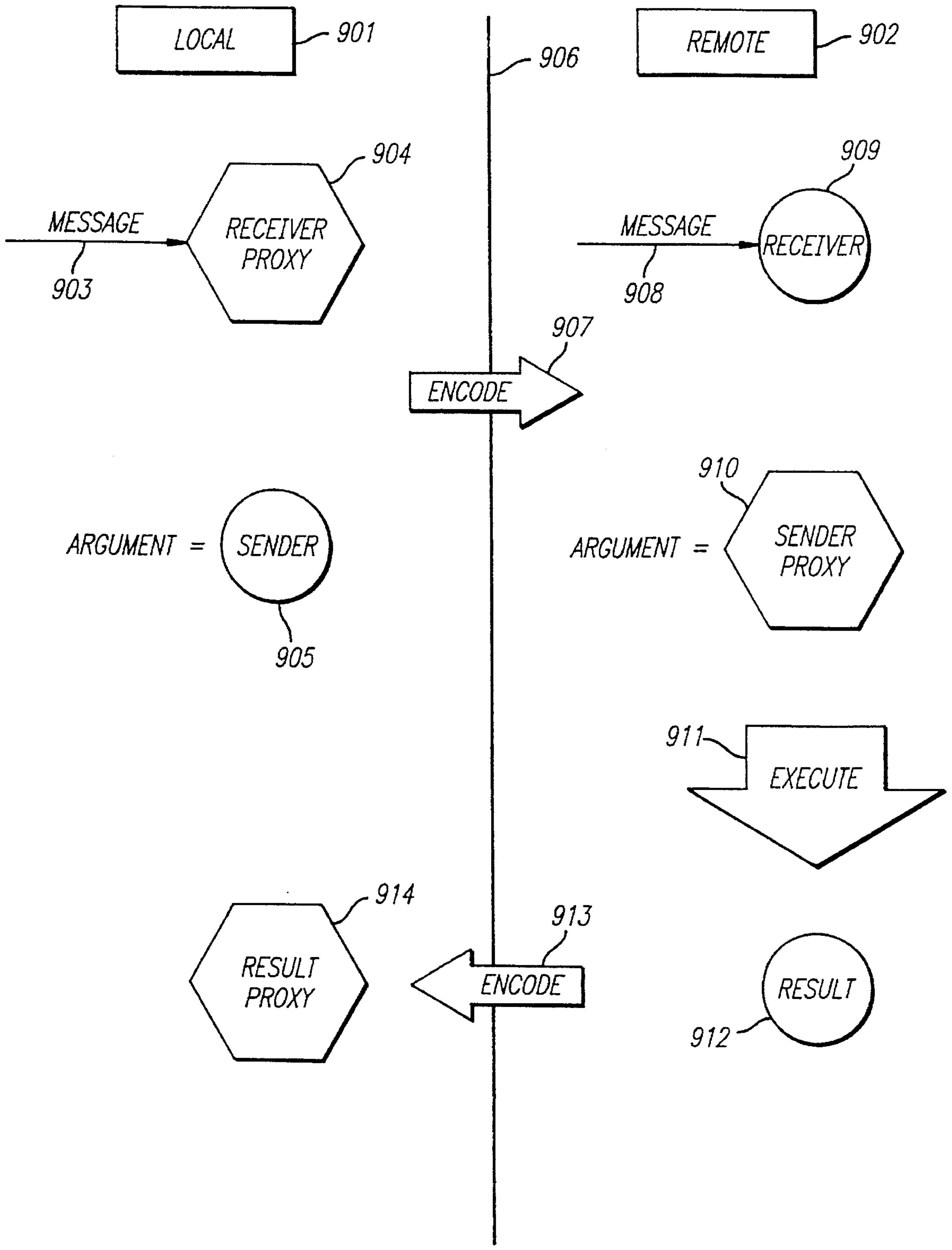

LOCAL
901
906
REMOTE
902
904
909
MESSAGE
RECEIVER PROXY
903
MESSAGE
RECEIVER
908
907
ENCODE
910
ARGUMENT =
SENDER
905
ARGUMENT =
SENDER PROXY
911
EXECUTE
914
913
RESULT PROXY
ENCODE
RESULT
912

FIG. 5

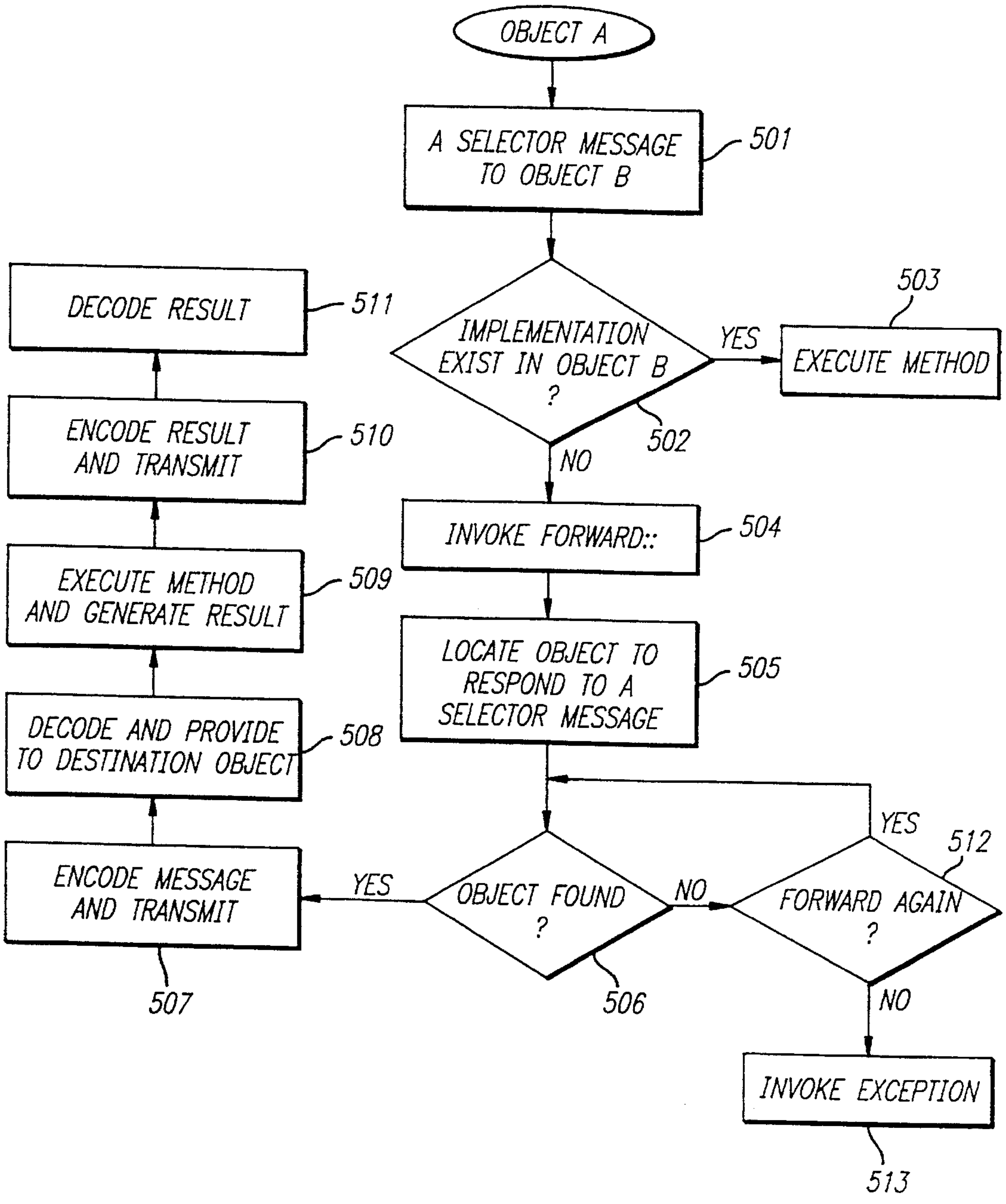

OBJECT A
A SELECTOR MESSAGE TO OBJECT B
501
IMPLEMENTATION EXIST IN OBJECT B ?
502
YES
EXECUTE METHOD
503
NO
INVOKE FORWARD::
504
LOCATE OBJECT TO RESPOND TO A SELECTOR MESSAGE
505
OBJECT FOUND ?
506
YES
ENCODE MESSAGE AND TRANSMIT
507
DECODE AND PROVIDE TO DESTINATION OBJECT
508
EXECUTE METHOD AND GENERATE RESULT
509
ENCODE RESULT AND TRANSMIT
510
DECODE RESULT
511
NO
FORWARD AGAIN ?
512
YES
NO
INVOKE EXCEPTION
513

FIG. 6 PRIOR ART

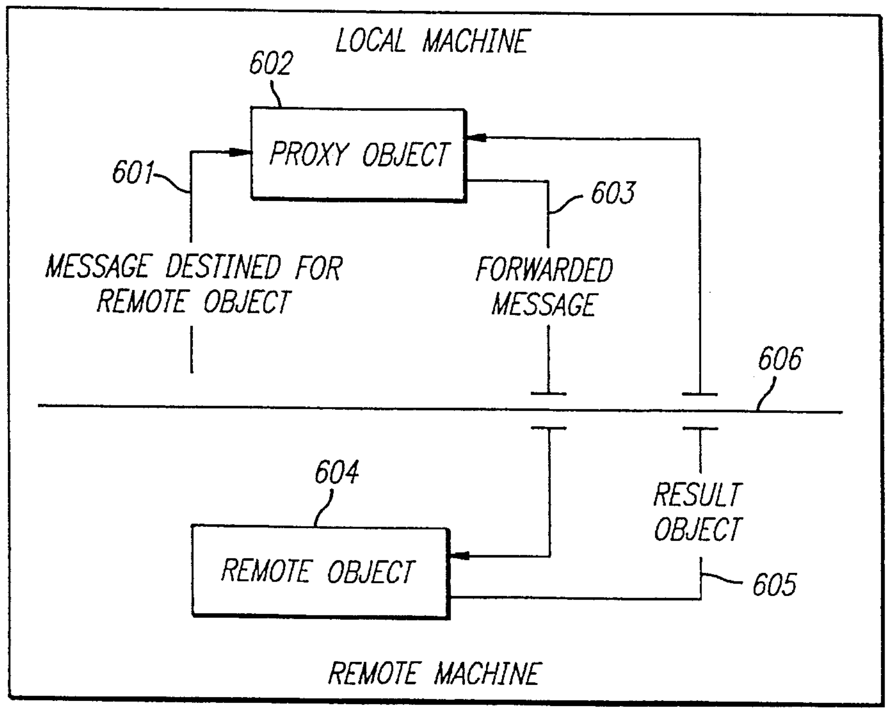


FIG. 7 PRIOR ART

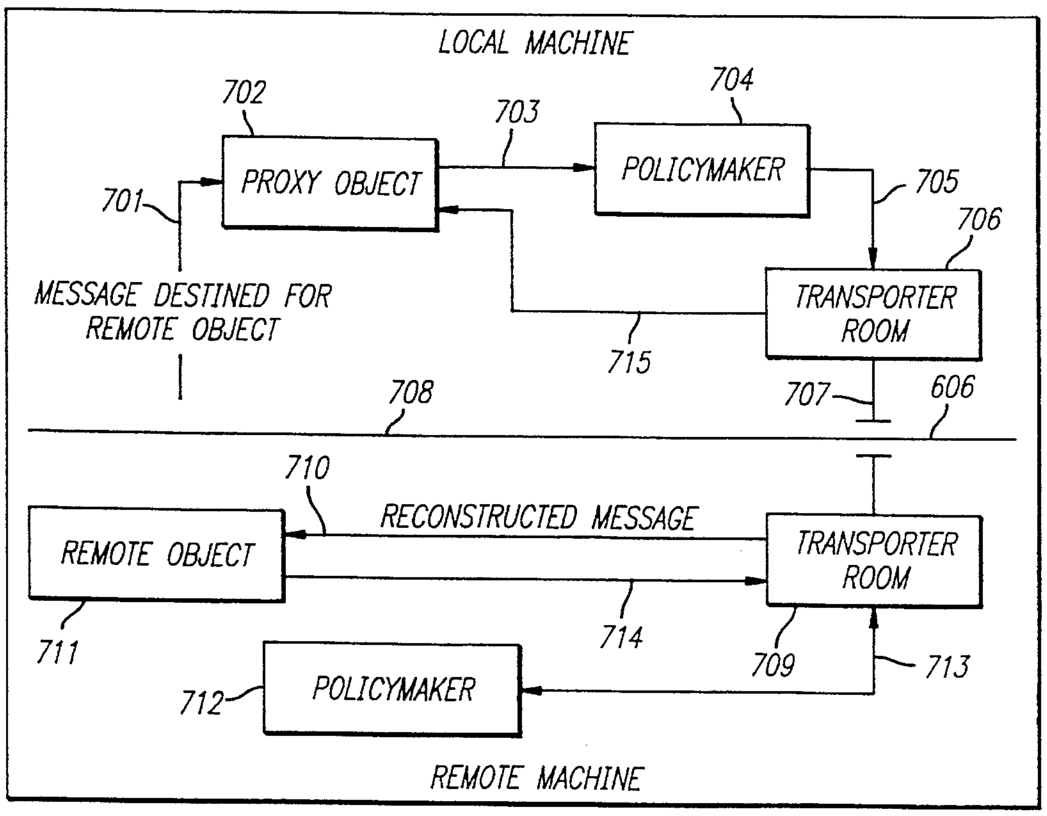


FIG. 8A

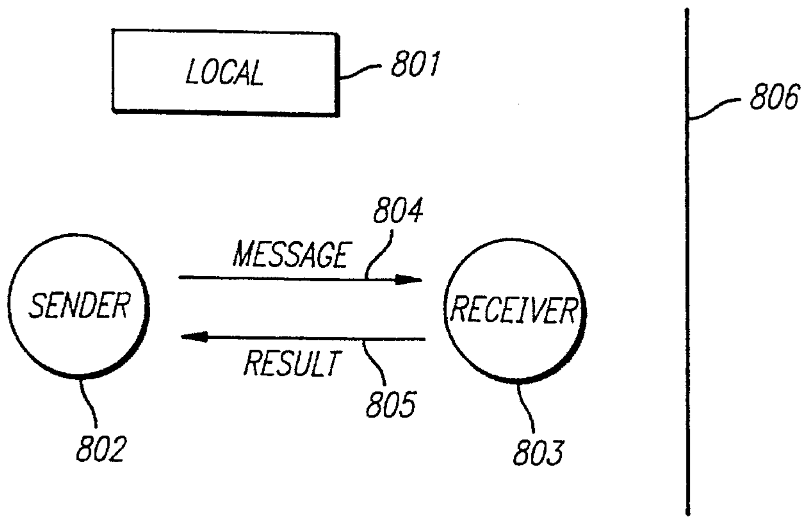
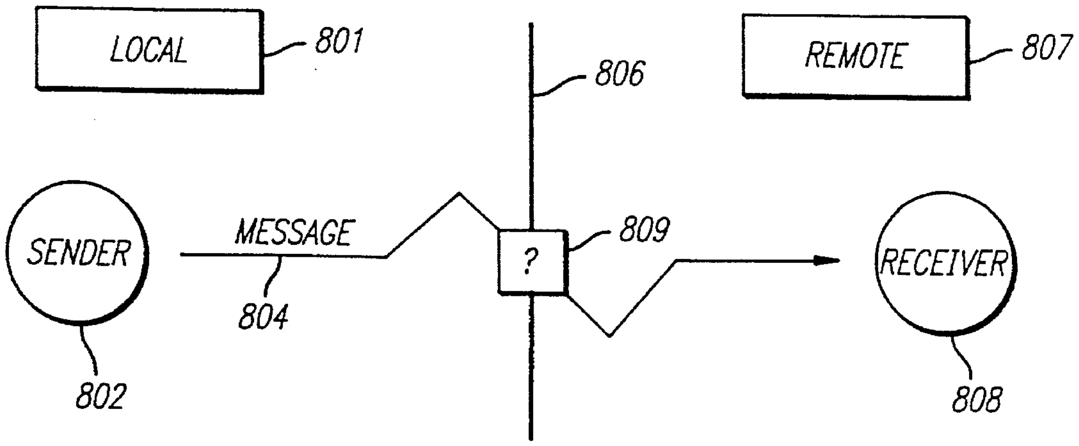


FIG. 8B PRIOR ART



METHOD FOR PROVIDING AUTOMATIC AND DYNAMIC TRANSLATION OF OBJECT ORIENTED PROGRAMMING LANGUAGE-BASED MESSAGE PASSING INTO OPERATION SYSTEM MESSAGE PASSING USING PROXY OBJECTS

BACKGROUND OF THE PRESENT INVENTION

This is a continuation of application Ser. No. 07/731,636 filed Jul. 17, 1991, now abandoned.

FIELD OF THE INVENTION

This invention relates to the field of object-oriented programming and distributed computing.

BACKGROUND ART

Object-oriented programming is a method of creating computer programs by combining certain fundamental building blocks, and creating relationships among and between the building blocks. The building blocks object-oriented programming systems are called "objects." An object is a programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data. Thus, an object consists of data and one or more operations or procedures that can be performed on that data. The joining of data and operations into a unitary building block is "encapsulation." In object-oriented programming, operations that can be performed on the data are referred to as "methods."

An object can be instructed to perform one of its methods when it receives a "message." A message is a command or instruction to the object to execute a certain method. It consists of a method selection (name) and arguments that are sent to an object. A message tells the receiving object what to do.

One advantage of object-oriented programming is the way in which methods are invoked. When a message is sent to an object, it is not necessary for the message to instruct the object how to perform a certain method. It is only necessary to request that the object execute the method. This greatly simplifies program development.

Object-oriented programming languages are generally based on one of two schemes for representing general concepts and sharing knowledge. One scheme is known as the "class" scheme. The other scheme is known as the "prototype" scheme. Both the set-based and prototype-based object-oriented programming schemes are generally described in Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," OOPSLA 86 Proceedings, September 1986, pp. 214–223.

Class Scheme

An object that describes behavior is called a "class." Objects that acquire a behavior and that have states are called "instances." Thus, in the objective C language, which is the computer language in which the preferred embodiment of the present invention is implemented, a class is a particular type of object. In objective C, any object that is not a class object is said to be an instance of its class. The classes form a "hierarchy." Each subclass in the hierarchy may add to or modify the behavior of the object in question and may also add additional states. Inheritance is a fundamental property of the class scheme and allows objects to acquire

behavior from other objects.

The inheritance hierarchy is the hierarchy of classes defined by the arrangement of superclasses and subclasses. Except for the root classes, every class has a superclass, and any class may have an unlimited number of subclasses. Each class inherits from those classes above it in the hierarchy. Thus, a superclass has the ability to pass its characteristics (methods and instance variables) onto its subclasses.

FIG. **1** is a block diagram that illustrates inheritance. Class **1** (generally indicated by block **101**) defines a class of objects that have three methods in common, namely, A, B and C. An object belonging to a class is referred to as an "instance" of that class. An example of an instance of class **1** is block **102**. An instance such as instance **102** contains all the methods of its parent class. Block **102** contains methods A, B and C.

As discussed, each class may also have subclasses, which also share all the methods of the class. Subclass **1.1** (indicated by block **103**) inherits methods A, B and C and defines an additional method, E. Each subclass can have its own instances, such as, for example, instance **104**. Each instance of a subclass includes all the methods of the subclass. For example, instance **104** includes methods A, B, C and E of subclass **1.1**.

Not all object oriented programming languages permit new methods to be added per instance. For, example, in Objective C, an instance can not have methods that are not contained in its parent class.

A disadvantage of an inheritance-based, object-oriented programming language is object size. Because each subclass must, by definition, include all methods of its parent class and super classes, instances are larger at the bottom of the inheritance hierarchy.

Object-oriented programming languages that utilize the class/instance/inheritance structure described above implement a set-theoretic approach to sharing knowledge. This approach is used in object-oriented programming languages, such as Simula, SmallTalk, Flavors and Loops.

Prototype Scheme

The prototype scheme is an alternate approach to sharing knowledge in an object-oriented system. In prototype scheme systems that use the individual instances, rather than a class, ("prototypes") are created first, instead of a class. The prototypes are then generalized by defining aspects of their concepts that are permitted to vary. The mechanism for implementing this process is known as "delegation." Examples of prototype languages include the actor language and lisp-based object-oriented systems, such as director, t, and orbit.

Delegation removes the distinction between classes and instances. To create another object that shares knowledge with a prototype, an "extension" object is created that has a list containing its prototypes that may be shared with other objects and containing personal behavior limited to the object itself. When an extension object receives a message, it attempts to respond to the message using the behavior stored in its personal aspect. If the object's personal characteristics are not suitable to answer the message, the object forwards the message on to other prototypes to see if one can respond to the message. This method of forwarding is called "delegating the message."

An example of delegation is illustrated in FIG. **2**. Object A provides a message **201** to object B. The message includes a method and arguments to the method. Object B does not have the method required by the message. Therefore, object B cannot respond to the message. Instead, object B sends the method and message to its delegate. The delegate of object B is object C. Object C has the method requested in the message. The method can then be executed and a response provided to object A.

Distributed Programming

A disadvantage of current object-oriented programming systems is that all objects are required to exist in a single program or process. This prohibits utilizing an object-oriented programming system when writing distributed applications. In addition, these prior art limitations prevent the creation of applications that are distributed physically over networks of machines.

The difficulty of creating distributed object-oriented programs is illustrated in FIGS. **8**A and **8**B. In FIG. **8**A, all objects are resident in the same program, namely program LOCAL **801**. A sender object **802** sends a message **804** to a receiver **803**. The message may include a method and an argument. The receiver **803** executes the method of the message **804** and returns a result **805**. The result **805** is provided back to the sender **802**. In the example of FIG. **8**A, the object-oriented program resides entirely on one side of a boundary **806**.

FIG. **8**B illustrates a prior art object-oriented programming system attempting to communicate across a boundary between processes. A local process **801** includes a sender object **802** that generates a message **804** destined for receiver object **808**. However, object **808** is on the opposite side of boundary **806**. That is a separate process identified as REMOTE **807**. An object, such as receiver **808** which resides in a different process than a sender object, is known a "remote object." The language and run time support of the local process **801** does not provide a mechanism to send a message **804** directly to the remote object **808**. At the transition point **809** of boundary **806**, the message is stopped.

One prior art approach for writing distributed applications consists of explicitly defining the boundaries between different programs by specifying protocols, generating client/server stubs, and communicating between processes or programs by using function calls that automatically transport arguments and return values between the client layer and the server layer. At such boundaries, the object-oriented developer can no longer treat items as objects as soon as the boundary of the process is crossed. This defeats the purpose and advantage of using object-oriented programming in the first place.

Another prior art method for providing distributed object oriented programming is described in "Design of a Distributed Object Manager for the SmallTalk-80System," D. Decouchant, OOPSLA 86 Proceedings, September, 1986, pp. 444–452. The Decouchant reference describes the design of a distributed object manager that allows several SmallTalk-80 systems to share objects over a local area network. When a local object desires to communicate with a remote object, the local object communicates with a "proxy" that locally represents the remote object. A proxy is part of the private data of the object manager. The proxy has two fields that describe a remote object, namely, the resident site of the remote object and a virtual pointer to the object in the resident site. If the referenced object migrates, the contents of the referencing object are not modified. The proxy is updated accordingly by the object manager. In this implementation, a proxy is functionally equivalent to a Unix link, except that a proxy is not visible to the programmer. It is a private data structure which is handled by the object manager like other Small-Talk objects.

In the Decouchant reference, three processes cooperate to perform the object manager functions. These are the network manager, the main memory manager and the secondary manager. SmallTalk interpreter processes which access the objects to perform SmallTalk actions may also be present on the site. The network manager is the master process of a SmallTalk site. It ensures consistency of the shared objects with the other network sites and controls the local processes. The main memory manager is in charge of the object management in main memory. It resolves object faults by allocating free space for the missing object and sending an object load request to the secondary storage manager. The secondary memory manager takes care of the object management in secondary storage. This storage is represented by two files, one of which contains the SmallTalk object table and the other one contains the object space.

Another prior art method to provide distributed object-oriented programming is described in "The Design and Implementation of Distributed SmallTalk," John K. Bennett, OOPSLA, Oct. 4–8, 1987, pp. 318–330. SmallTalk itself is a language environment that provides a single user with access to a single object address space. Only rudimentary support exists within SmallTalk for cooperation among users, and no support exists within SmallTalk for object sharing between users or between different machines or between processes on the same or different machines. The Bennett references describes "distributed SmallTalk" (DS) as a method of providing improved communication and interaction among geographically remote SmallTalk users, direct access to remote objects, the ability to construct distributed applications in a SmallTalk environment, and a degree of object sharing among users.

The system described in the Bennett reference does not allow remote classes. Instead, the system requires that classes and instances be co-resident on all processes and machines. This impacts object mobility adversely. Instances can only move to hosts with compatible classes and insuring class compatibility is difficult. In addition, the system of Bennett does not operate in an object-oriented programming system that utilizes class inheritance and reactiveness. (Reactiveness describes the ability of a system to present objects for inspection or modification).

The system of Bennett uses ProxyObjects and a RemoteObjectTable to implement distributed message passing. A ProxyObject represents a remote object to all objects in a local address space. There is one ProxyObject per host per remote object referenced by that host. ProxyObjects cause a remote object's message interface to appear to local objects as if the remote object were locally resident. ProxyObjects redefine the doesNotUnderstand: message of object. This is the primary message defined for ProxyObjects. In other words, messages sent to ProxyObjects are intended to fail. The system responds to this failure by sending the message doesNotUnderstand: to the receiver with the message that was not understood as an argument. The ProxyObject's response to the doesNotUnderstand: message is to forward the original message to the RemoteObjectTable on the appropriate machine or process. The location of the remote object is part of the internal state of the ProxyObject.

The RemoteObjectTable is responsible for receiving and replying to messages forwarded by ProxyObjects. There is one RemoteObjectTable per host. It is the sole instance of class RemoteObjectTable. The RemoteObjectTable can be thought of as a set of extensions to the object tables (if present) of all remote machines. The RemoteObjectTable keeps track of all local objects that are remotely referenced. When the RemoteObjectTable receives a message from some ProxyObject, it schedules a process that will contain the execution context of the actual message receiver by sending the message perform to the receiver with the forwarded selector and arguments (if any) as arguments to the perform message. The value returned by the perform message is returned to the remote sender in a reply message constructed by the RemoteObjectTable.

Before an object can be sent between processes, the classes must be checked for compatibility. The three cases to consider are:

1. The required class is already present and is compatible;
2. The required class is present, but it is determined to be incompatible; and
3. The required class is not present.

In case 1, the system proceeds normally. In the second case, the attempted move fails and the user is notified of the error. In case 3, the user is asked whether the desired object's class should be moved. If the response is affirmative, the object's super class is checked for compatibility. This procedure continues up the class hierarchy until class object is reached. However, class object may not be moved.

Another method for providing distributed object-oriented programming is described in "Transparent Forwarding: First Steps" Paul L. McCullough, OOPSLA 1987 Proceedings, Oct. 4–8, 1987, pp. 331–341. As in the Bennett system, the McCullough system utilizes ProxyObjects and the doesNotUnderstand: message for identifying and transmitting messages. In the McCullough system, the implementation of doesNotUnderstand: creates an ethernet packet containing the original message and forwards it to the machine containing the remote object. The proxy contains information in its instance variables about where the remote object resides.

FIG. **6** illustrates an overview of the operation of the McCullough system. A message **601** destined for a remote object is provided to a ProxyObject **602**. The ProxyObject instance forms a representation of the message, including both the selector and the arguments, suitable for transmission to a remote machine. This message **603** is forwarded across the process boundary **606** to a remote object **604**. The remote object **604** receives the representation of the message, extracts the message selector and arguments and executes the message send as though it originated on the same machine as the remote object. This result object **605** is transmitted across the process boundary **606** to the ProxyObject **602**. The ProxyObject **602** uses the return representation to reconstruct the result object and returns it to the sender of the original message **601**.

The McCullough system implements four possible message parameter passing schemes, namely, pass by value, pass by reference, pass by proxy and pass by migration. In pass by value, a representation of the object is shipped to the remote machine, which in turn reconstructs the object. Pass by reference cannot be used in a SmallTalk environment because the compiler prevents assignment to formal parameter variables. In pass by proxy, a proxy for the object and any messages which are sent to the proxy are automatically forwarded to the remote object. In pass by migration, we move an object from one machine to another, leaving a proxy object in its prior home.

A centralized control scheme, referred to as PolicyMaker, is used to deliver messages to remote objects. Individual proxies need not record the current network location of a shared object, that is the responsibility of the PolicyMaker. PolicyMaker responsibilities include the decision of whether to pass objects by value, proxy or by migration, and whether to forward a message to a remote object or whether to migrate the object to the local machine for execution. In addition, PolicyMaker keeps track of open connections between machines. For each connection to a remote machine, the PolicyMaker creates an instance of class TransporterRoom. The TransporterRoom takes care of communications protocols between machines, as well as the linearization of messages and objects.

FIG. **7** illustrates the flow control of sending a message from a machine to a remote object using the scheme of the McCullough system. A message **701**, destined for a remote object, is provided to a ProxyObject **702**. The sender of the message **701** believes it is sending to a local object, but in reality it is sending to a remote object. The ProxyObject **702** sends a message **703** to the local PolicyMaker **704**. The PolicyMaker **704** determines whether the arguments of the message should be sent by copying or by proxy to the remote object. The PolicyMaker establishes a connection to the remote machine via transporter room **706**. The PolicyMaker **704** provides the message **705** to the TransporterRoom **706**. The TransporterRoom **706** linearizes and transmits the message as message **707** to the remote machine across process boundary **708**.

The TransporterRoom **709** of the remote machine receives the message **707**. The TransporterRoom **709** sends the reconstructed message **710** to the remote object **711**. The remote object **711** returns a message **714** to the TransporterRoom **709**. The PolicyMaker **712** considers the resulting object and determines whether to return it by value or by proxy and communicates to the TransporterRoom **709** on path **713**. The TransporterRoom **709** sends the message **707** to the TransporterRoom **706** across process boundary **708**. The TransporterRoom **706** reconstructs the result object and provides it as message **715** to proxy object **702**, which can then return it to the sending context.

The use of migration limits the performance and ease of use of these prior art schemes. Migration of objects from their home process adds to the complexity of the system. Another disadvantage of these prior art schemes is that each process and thread must be forked to anticipate each expected iteration. There is no provision for dynamic recursive communication between processes. In addition, these prior art schemes rely on a pure, large object oriented language/environment, such as SmallTalk. This requires substantial run time support to implement communication between processes. In addition, the prior art schemes do not implement suitable object collection methods.

SUMMARY OF THE INVENTION

The present invention permits the distribution of objects and sending of messages between objects that are located in different processes. Initially, a "proxy" object is created in the same process as a sender object. This proxy acts as a local receiver for all objects in the local program. When the proxy receives a message, the message is encoded and transmitted between programs as a stream of bytes. In the remote process, the message is decoded and executed as if the sender was remote. The result follows the same path, encoded, transmitted, and then decoded back in the local process. The result is then provided to the sending object.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram illustrating the concept of inheritance in object-oriented programming.

FIG. **2** is a block diagram illustrating delegation in object-oriented programming.

FIGS. **3**A–**3**C illustrate the distributed processing object oriented programming system of the present invention.

FIG. **4** is a block diagram illustrating a general purpose computer system for implementing the present invention.

FIG. **5** is a flow diagram of the forwarding method of the present invention.

FIG. **6** is a block diagram illustrating the prior art distributed processing object-oriented programming system.

FIG. **7** is a block diagram illustrating another prior art distributed processing object-oriented programming system.

FIGS. **8**A and **8**B illustrate non-distributed programming systems.

DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus for distributed execution of methods is described. In the following description, numerous specific details, such as object-oriented programming language, operating system, etc., are set forth in detail in order to provide a more thorough understanding of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well known features have not been described in detail so as not to obscure the present invention.

The present invention may be implemented on any conventional or general purpose computer system. An example of one embodiment of a computer system for implementing this invention is illustrated in FIG. 4. A keyboard **410** and mouse **411** are coupled to a bi-directional system **419**. The keyboard and mouse are for introducing user input to the computer system and communicating that user input to CPU **413**. The computer system of FIG. 4 also includes a video memory **414**, main memory **415** and mass storage **412**, all coupled to bi-directional system bus **419** along with keyboard **410**, mouse **411** and CPU **413**. The mass storage **412** may include both fixed and removable media, such as magnetic, optical or magnetic optical storage systems or any other available mass storage technology. The mass storage may be shared on a network, or it may be dedicated mass storage. Bus **419** may contain, for example, 32 address lines for addressing video memory **414** or main memory **415**. The system bus **419** also includes, for example, a 32-bit data bus for transferring data between and among the components, such as CPU **413**, main memory **415**, video memory **414** and mass storage **412**. Alternatively, multiplex data/address lines may be used instead of separate data and address lines.

In the preferred embodiment of this invention, the CPU **413** is a 32-bit microprocessor manufactured by Motorola, such as the 68030 or 68040. However, any other suitable microprocessor or microcomputer may be utilized. The Motorola microprocessor and its instruction set, bus structure and control lines are described in MC68030 User's Manual, and MC68040 User's Manual, published by Motorola Inc. of Phoenix, Ariz.

Main memory **415** is comprised of dynamic random access memory (DRAM) and in the preferred embodiment of this invention, comprises 8 megabytes of memory. More or less memory may be used without departing from the scope of this invention. Video memory **414** is a dual-ported video random access memory, and this invention consists, for example, of 256 kbytes of memory. However, more or less video memory may be provided as well.

One port of the video memory **414** is coupled to video multiplexer and shifter **416**, which in turn is coupled to video amplifier **417**. The video amplifier **417** is used to drive the cathode ray tube (CRT) raster monitor **418**. Video multiplexing shifter circuitry **416** and video amplifier **417** are well known in the art and may be implemented by any suitable means. This circuitry converts pixel data stored in video memory **414** to a raster signal suitable for use by monitor **418**. Monitor **418** is a type of monitor suitable for displaying graphic images, and in the preferred embodiment of this invention, has a resolution of approximately 1020×832. Other resolution monitors may be utilized in this invention.

The computer system described above is for purposes of example only. The present invention may be implemented in any type of computer system or programming or processing environment.

The preferred embodiment of the present invention implements an object-oriented programming system using objective C language. Objective C is an extension to ANSI C that supports the definition of classes of objects and provides syntactic and run-time support for sending messages to objects. This language model is partially derived from SmallTalk and has been described in "Object-Oriented Programming; An Evolutionary Approach," Brad J. Cox, Addison-Wesley 1986 and in "SmallTalk-80: The Language and its Implementation," Adele Goldberg, Dave Robson, Addison-Wesley 1983.

One feature of objective C is "dynamic binding" of messages to the actual methods to be invoked, depending on the class of the receiver. A programmer writing code in objective C can create code that sends a message "doSomething" to an object. The actual method corresponding to the class of the target object does not need to be determined until the message must be sent. This allows objects of any classes that implementing the doSomething method to be substituted for the target object at run time without having to modify the part of the program that sends the message. Also, in objective C, programs have run time access to method "signatures," that encode a method's argument and return types for each class. The method signature provides a way for two programs to agree on the format of messages. Moreover, there is a way to extract arguments from the stack using the signature.

In its preferred embodiment, the present invention is implemented in a computer system using an object-oriented operating system. One such object-oriented operating system is known as the "Mach" operating system and is implemented on computers manufactured by NeXT, Inc., the Assignee of the present invention.

The Mach operating system is an object-oriented operating system that supports distributed programming. It is a multi-tasking operating system kernel, allowing multiple, independent "tasks," which provide the basic environment of a program in the form of a demand-paged virtual "address space" and one or more "threads" of execution. Mach supports message-passing within and between tasks. This support is distinct from objective C messaging described earlier.

Fundamental to Mach's ability to deliver messages between different programs is an abstraction called a "port." A Mach port is a buffered communication channel over which messages are sent. This channel, which is maintained by the operating system, may be local, may span two tasks on the same machine, or may span tasks on different machines. The physical location of the receiving end of a port has no effect on the sender, which always sees a local reference to the port.

The messages sent on a port are buffered, that is, a sender writes messages to a port with a "send" primitive, and a receiver accepts messages with a "receive" primitive. The messages themselves may be of any size, and consist of a header followed by zero or more data objects. For efficiency, large array arguments are passed out-of-line with copy-on-write semantics. Of particular interest is the ability to pass Mach ports themselves as data objects in a message. By this means, one task may pass a port to another, with the kernel maintaining address translation along the way. This allows tasks to learn about the existence of new external objects, or make new "acquaintances," by receiving their ports in a message. Thus, ports may also be viewed as a reference for an object that is independent of any particular space, and may be freely passed between programs.

For a task to communicate with a "receiver" object in another address space, it must first establish a connection with that program, and then create a local "proxy" for the object. When a message is sent to this proxy, the elements of the objective C message are encoded into a Mach message, which is then forwarded through a Mach port to the other program. On the receiving side, the message is received, decoded, and then forwarded it to the target objective C object. The return value of the objective C method is then encoded and sent back to the originator, where it is decoded and returned as the value. Each part of this model is now described.

In order to communicate with an object in a different program or process, that program or process must be known. The present invention uses Mach ports to represent "domains" of objects, and a token to identify objects within that domain. This two-part address is easily communicated, since the port maintains its identity as it moves between domains. In Mach, acquiring ports is synonymous with acquiring privileges to communicate. The present invention requires that the local domain has send rights to a port that the remote domain has receive rights to, and also that the remote domain has send rights to a port that the local domain has receive rights to, before any communication can take place. Thus, mutual consent is required to communicate.

In addition to learning of each other's ports, each domain must also provide the other with the token corresponding to its "first proxy." A first proxy is required to bootstrap the communication. There must be at least one known object in a remote domain before a message can be sent to it. Because a connection allows messages in either direction and initiated by either party, each side of a connection must have a first proxy. This first proxy may be viewed as the "receptionist" for the remote domain, as other objects are obtained (discovered) by asking this object. This object also is a candidate for implementing sender authentication.

The present invention provides a means for implementing an extensible, distributed program in which one task is responsible for creating other tasks to communicate with. This is a master/slave relationship; the master can provide the slave with send rights to the master's port as part of the creation process. When the slave starts executing, it sends a Mach message containing send rights to its port and a token for its first proxy back to the master. The master then replies with an indication of whether the connection is granted, and what token to use for the first proxy. This "bootstrap-meta-protocol" results in both tasks knowing about each other, allowing communication to ensue.

Distributed Object Oriented Programming

The present invention provides a method for different processes to communicate, using a traditional language-

based, message-passing paradigm. The present invention has a number of advantages over prior art methods of distributed object-oriented programming. These advantages include no pre-defined set of messages, transparent to the programmer, no code generation step and a method for bridging the gap between object-oriented languages and object-oriented operating systems.

The present invention differs from the prior art approaches of Decouchant, Bennett and McCullough. The present invention uses an object-oriented superset of ANSI C with minimal run time support to implement transparent messaging between application programs as opposed to the prior art systems, that rely on a pure, large, dynamic object-oriented language/environment, such as SmallTalk. The present invention either implements a client/server setup or a master/slave setup that involves forking tasks to perform background operations and using the present invention to communicate with these tasks. Because communications are serialized by the operating system, remote messages performed by a slave task appear to the program just like other asynchronous events, such as mouse clicks, allowing the design of a consistent user interface. Modularity, extensibility and safety are gained by spawning new tasks. The ability to implement recursive remote messaging is an important feature of the present invention, for example when user interaction is required to perform the desired task.

The present invention can also be implemented in a client/server setup. In the client/server setup, both the client and the server begin independently. Communication is through an agreed upon method. The client and server can communicate with each other by looking for a manager for the communications channel (e.g. network).

The present invention provides a new alternative for developing extensible programs. Instead of adding functionality by adding code that defines new object classes and then loading that code into a main program, a new program is created and forked. If there are any errors in the new program, they reside outside the main program improving performance. In addition, processing is parallel and asynchronous, leading to improved performance.

In one embodiment, the first time a message is sent to a proxy, the receiver is asked for the method signature in order to allow the proxy's domain to encode the arguments. This can increase communication time. One alternative embodiment provides for prior agreement between processes so that two tasks know in advance the method signatures for their proxies. Alternatively, when prior agreement is not possible due to the dynamic nature of the required exchange, the atomicity of the signature request can be changed to communicate all the public signatures for a given proxy, resulting in a single "meta-protocol" transaction for the proxy.

When an object is passed by reference, a new proxy is typically created. The present invention includes safeguards to guarantee that if a remote object is encoded twice, the same proxy (in the pointer equality sense) will be obtained in the local domain. This unicity of proxies is maintained by a table which maps tokens of remote objects to their local proxies and looks for previously created proxies when an object pass by reference is decoded. The present invention keeps all proxies until the communication with the remote domain ends. At that time, the table is used to de-allocate all proxies.

The operation of the present invention is illustrated in FIGS. 3A–3C. Referring first to FIG. 3 A, a local process **901** is separated from a remote process **902** by boundary **906**. The boundary **906** could be a separation between programs on the same computer or it could represent the separation of two different machines on a network. The local process **901** includes a sender object **905** that sends a message to receiver object **909**. The object **909** is located in the remote process **902**. However, the sender object **905** can send its message **903** to the remote object as if it were a local object.

The local process **901** includes a receiver proxy **904** that accepts the message **903**. The receiver proxy **904** is an object that executes a forward:: method. The receiver proxy **904** encodes the message and transmits it across the process boundary to the remote object. In the preferred embodiment of the present invention, the proxy **904** encodes the message, (which is a language based message such as, for example, an objective C message), as an operating system message, such as a Mach message **907**, and transmits it to the receiver object **909** in the remote process **902**. The receiver object **909** decodes the Mach message into a language based message for execution or handling in the remote process **902**.

The present invention supports nested, recursive, remote messages. That is, when a message is sent to a remote object, that remote object may send other messages back to the local process (which may again send other remote messages), as part of its calculations before providing a reply to the initial message. These messages may be nested arbitrarily deep. If only the sender itself is sent as an argument, (such as illustrated in FIG. **3** A), the receiver determines what further information is required from the sender and sends messages back to the local process to obtain that information before generating a reply.

Referring to FIG. **3** B, the receiver object **909** requires additional information from the sender object so it generates a request message to the sender object **904**. However, since the sender object **904** is not resident in the remote process **902**, a sender proxy **910** is created in the remote process **902**. The sender proxy **910** encodes the request into a Mach message and transmits it across process boundary **906** to sender object **905**. The Mach message is decoded into a language based message and generates a response. This response is sent back to the receiver object **909**, (again via receiver proxy **904**).

Referring to FIG. **3** C, the receiver object **909** then performs an execute **911** on the original message to generate a result object **912**. The result object **912** is encoded **913** and transmitted across process boundary **906** to result proxy **914** in local process **901**.

In the present invention, proxies are not required to be created in advance. Once one proxy of a remote process exists, N proxies can be created for communication with that process. The present invention also permits the sending of objects themselves across process boundaries.

The recursive nature of the present invention is useful when the remote object does not recognize a method sent to it by a local object. For example, the local object may send a message to a remote object requesting it to execute the method "foo". If the remote object does not recognize the method, it can ask the sending object "what is foo?" All objects recognize the method "what is". The sender object can then respond with instructions concerning the nature of the method being investigated. For example, the sender can reply that foo is a method that requires an integer. If no integer has been provided, the remote object can then request the integer.

The creation of the first proxy is provided automatically in the present invention. The first time a process is accessed, it must call a process referred to as the "proxy receptionist". By definition, the first call to a new process is to be at sequence number 0. The first proxy by definition has a sequence number of 0. Subsequent proxies, as needed, are defined and the sequence numbers are provided to the other process.

In the preferred embodiment of the present invention, communication between processes is implemented in a master/slave or client/server relationship. In the case of a client/server relationship, the server may "publish" its port in an appropriate place on the system. The client looks up this port and then uses it to initiate the bootstrap-meta-protocol described above.

The establishment of a connection defines only the first proxy on each side. Proxies for any other objects that need to be communicated are created dynamically as they are encountered. For example, if a remote method returns a new object, a new proxy is created when decoding the result locally, such that the local program could send a message to this new remote object without any explicit setup.

Once a connection is established, a message may be sent (in either direction). To send a message, the arguments must be encoded into a form that is representable in a Mach message, so that the receiver may decode them correctly. To do this, the sender must know the method signature of the message. The method signature is part of the "protocol" that both sender and receiver must understand in order to communicate, and is a domain-independent encapsulation of the method name, its argument types, and its return value type.

Although a protocol may be arranged by prior agreement in many cases, the present invention determines the protocol dynamically by asking the receiver how to encode the arguments for each message as it is encountered. During the first attempt to send a given message to a remote object, the local domain consults the remote domain to get the signature corresponding to the actual class implementation that will ultimately receive the message. This method signature is cached on a per-connection basis, with the assumption that the class of a remote object will not change for the duration of the connection. Thus, subsequent messages use the cached signature, and do not have the overhead of this "meta-protocol" transaction. This dynamic aspect is useful for type checking, and allows one program to "learn" how to talk to another program.

The argument encoding for standard C data types is explicit and strictly pass-by-value, and maps substantially directly onto a Mach message. Pointers to C data structures are not encoded. Arguments that are first-class objects (as opposed to simple C data types) are handled specially; they are asked to encode themselves. The default encoding scheme that most objects inherit is to allocate a token (if one does not already exist) in the local domain, and encode only that token. Along with the port of the local domain, this constitutes an object reference (via proxy), and when it is decoded on the receiving side, results in a new proxy for that remote object.

Thus, first-class objects by default get passed by reference instead of by value. For example, when domain A sends a message with object X as an argument to a remote object in domain B, a new proxy is created in domain B to represent X. Any subsequent message that domain B sends to X results in another remote object transaction. In the present invention, an implementation of an object class is free to choose to implement a different encoding scheme, for example one that encodes the object by value, though this requires that both sender and receiver implement that class of object. The "pass-by-proxy" scheme does not have this restriction and is generally preferred.

Because objective C implements functional messages, a return value is always returned in a reply message. It is encoded exactly the same way as arguments are. In particular, if the result is an object or a proxy, it is encoded as a proxy or an object, respectively, in the other domain. The reply message also encodes information about errors or exceptions that may have occurred, so that they can be raised in the local context. That is, an error occurring while executing a message sent to a remote object is caught and returned to the caller, so that the exception is raised in the local domain. This makes error handling transparent. Remote exceptions may be handled the same way as local exceptions.

The use of a single port to represent a domain of objects allows an efficient and simple implementation. A one-way message is used when forwarding a message to a remote domain. Messages are received on the local domain's single published port while waiting for the reply. Since all messages (including both the reply message and any other messages initiated from remote domains) arrive on the same port, each one can be handled serially. There is no global state to keep track of (the logic is implemented in a reentrant manner), and each message and reply have matching sequence numbers, so it can be determined when the correct reply has been received.

In effect, the low level routine to actually forward the message to the remote domain becomes the main loop of the program until the reply is received. There may be many nested levels of this low level routine at one time. Outside the scope of a locally-initiated remote message (i.e., the "idle" state of waiting asynchronous messages to arrive), the local domain's port is listened to by the main program, along with other non-remote-object related events. This approach contrasts with that of the prior art McCullough system, where a process or thread is forked to field every expected reply. The present invention achieves an order of magnitude in performance by avoiding the forking when communicating.

An example of a computer program listing that may be used to implement the present invention is described in Appendix A. This computer program listing is given by way of example only. The present invention may be practiced using other programs and methods as well.

Automatic Forwarding of Messages

The present invention, in its preferred embodiment, takes advantage of a method referred to as "automatic forwarding of messages." This method is the subject of copending patent application Ser. No. 07/695,316 filed May 3, 1991, entitled "METHOD FOR PROVIDING AUTOMATIC FORWARDING OF MESSAGES AND METHODS" and assigned to the assignee of the present invention. This method is described below.

In objective C, when an object receives a message that contains a method that the object does not recognize, an exception is provoked leading to an error. The present invention, instead of provoking an exception, redirects the message to an acquaintance that can understand the message. For example, if an object receives a message containing a method that the receiving object does not contain, the message is forwarded to an acquaintance object that does contain the method. This provides the advantage of inheriting the method from the acquaintance object but does not

require the first receiving object to actually have the method itself. This reduces code size the memory requirements.

The present invention has a plurality of uses. For example, a new object class can be defined so that one of its instances (attributed object) adds an attribute to another object (its forwardee). In that situation, automatic forwarding occurs when an attributed object receives a message that is irrelevant to the attribute, and the method is forwarded to the forwardee. In another situation, some functionality is applied before and/or after the forwarding. This can be used in the case of a locking data structure where all methods must be redirected to the locked data after acquiring the lock, and where the lock must be released after the execution of the forwarded method. The present invention also has use in the case of forwarding messages in a distributed environment where there is no explicit forwardee. Rather, there is a network address of the forwardee.

The implementation of the present invention in the preferred embodiment requires trapping the "message not recognized" exception of objective C, retrieving all of the arguments of the unrecognized message, and sending the "forward::" message to the object.

The resulting automatic forwarding system of the present invention is more powerful than multiple inheritance and is transparent to the programmer and developer. The system is general, because the forwardee is not explicit, thus permitting solutions to a large class of problems.

The present invention uses the forward:: command so that subclasses can forward messages to other objects. The format is forward: (SEL) aSelector:(marg_list) argFrame. When an object is sent an aSelector message, and the run time system cannot find an implementation of the method for the receiving object, the run time system sends the object a forward:: message to give it an opportunity to delegate the message to another object. If the forwardee object cannot respond to the message either, it also has the opportunity to forward the message. A forward:: message is generated only if a selector method is not implemented by the receiving object's class or by any of the classes it inherits from.

The forward:: message thus allows an object to establish relationships with other objects that will, for certain messages, act on its behalf. The forwarding object is, in a sense, able to "inherit" some of the characteristics of the object it forwards the message to. The forwarding object is not limited to the forwardees it may select, and a forwardee relationships may be formed with more than one object at the same hierarchical level. Therefore, the present invention provides the advantages of multiple inheritance without the code size problem.

In addition to forwarding messages, the forward:: method can locate code that responds to a variety of different messages, thus avoiding the necessity of having to write a separate method for each selector.

If implemented to forward messages, a forward:: method has two tasks. First, to locate an object that can respond to the aSelector message (this need not be the same object for all messages). Second, to send the message to that object using the performv:: and performv method.

The operation of the present invention is illustrated in the flow diagrams of FIG. **5**. At step **501**, Object A sends an aSelector message to object B. At decision block **502**, the argument "Implementation exists in Object B?" is made. If the argument is true, the method of the aSelector message can be executed by object B. If that is the case, the system proceeds to step **503** and the method is executed. If the argument is not true, the system proceeds to step **504** and invokes the forward:: method.

The forward:: method then performs the first of its two tasks at step **505**. Namely, it attempts to locate an object to respond to the aSelector message. At decision block **506**, the argument "Object found?" is made. If the argument is true, then forward:: has successfully found an object to respond to the aSelector message. In the present invention, the forwarding object is typically a proxy object.

The system then proceeds to step **507**, the message is encoded and transmitted as an operating system message to another process. At step **508**, the operating system message is decoded and provided to the destination object. At step **509**, the destination object executes the method of the message to generate a result. At step **510**, the result is encoded and transmitted to the first process as an operating system message. At step **511**, the message is decoded and the result is provided to the sending object.

If the argument at decision block **506** is not true, the system proceeds to decision block **512**. At decision block **512**, the argument "forward again?" is made. If the argument is true, the message is forwarded again and a search for an object to respond to the message is made. If the argument is false, the system proceeds to step **513** and an exception (error) is invoked.

In the case in which an object forwards messages to just one destination, a forward:: method could appear as follows:

```
- forward: (SEL)aSelector :(marg_list)argFrame
{
    if ([friend respondsTo:aSelector])
            return [friend performv:aSelector:argFrame}:
        return [self doesNotRecognize:aSelector];
}
```

ArgFrame is a pointer to the arguments included in the original aSelector message. It is passed directly to performv:: without change. The default version of forward:: implemented in the object class invokes the does not recognize: method. It does not forward messages. Thus, if a user chooses not to implement forward:: methods, unrecognized messages will be handled in the usual way.

The objective C run time code routines for implementing automatic forwarding of messages is as follows:

```
// provide a default error handler for unrecognized messages
static id-forward (id self, SEL sel, . . .)
{
    id     retval;
    // the following test is not necessary for Objects (instances of
    Object)
    // because forward:: is recognized.
    if (sel ==@selector (forward::)) }
            _objc_error (self, _errDoesntRecognize, SELNAME
    (sel));
            return nil;
    }
    retval =[self forward: sel : &self];
    return retval;
    {
            Method smt =(Method) objc_malloc (sizeof (struct
            objc_method));
            smt->method_name sel;
            smt->method_types = "";
            smt->method_imp = (IMP)_forward;
            _cache_fill (savCls, smt);
    }
    {
            Method smt = (Method) objc_malloc (sizeof (struct
            objc_method));
            smt->method_name = sel;
            smt->method_types = "";
            smt->method_imp = (IMP)_forward;
                    cache_fill (savCls, smt);
    }
    // the class does not respond to forward: (or, did not supply a
    dest)
    {
            Method smt = (Method) objc_malloc (sizeof (struct
            objc_method));
            smt->method_name = sel;
            smt->method_types = "";
            smt->method_imp = (IMP)_forward;
                    _cache_fill (savCls, smt);
    }
    return (IMP)_forward;
}
```

Thus, a method and apparatus for providing distributed processes is described.

APPENDIX A

```
#import <stdlib.h>
#import <stdarg.h>
#import <objc/HashTable.h>
#import <objc/hashtable.h>
#import <sys/message.h>

/* Declarations that will go away */
extern SEL _sel_registerName(STR key);

/*****************       Definitions      **************************/

extern int defaultCommTimeout; /* in millisecs, <0 means infinite */

extern int remoteMessageReceiveCount; /* increments when a msg is received */

/*      The following type of function may be passed to the beginListeningOn:rootObject: m
ethod. It gets asynchronously called during remote message sending. For example, an app co
uld register the port and function with DPSAddPort when the boolean is YES (and DPSRemoveP
ort when NO). */
typedef void (*remote_message_handler_t)(msg_header_t *msg, void *userData);

typedef void (*receive_enable_proc_t)(port_t port, remote_message_handler_t fun, BOOL shou
ldEnable);

typedef enum {
    #define REMOTE_EXCEPTION_BASE 36000 /* less than appkit base */
    /*  Format of exceptions is a label and a message string    */
    GENERIC_REMOTE_EXCEPTION = REMOTE_EXCEPTION_BASE,
    TIMEOUT_REMOTE_EXCEPTION,
    LAST_REMOTE_EXCEPTION
} RemoteException;

/*****************       Communication     *************************/

@interface Communication: Object {
    @public
    port_t      sendPort;
    int         timeout;        /* in milliseconds */
    NXHashTable *objectsGivenAway;
    NXHashTable *allProxies;
}
+ beginListeningOn:(port_t)listenPort enableProc:(receive_enable_proc_t)aProc;
    /* Initialize the remote object system.
    enableProc is called immediately with a boolean of YES. */

+ new:(port_t)port timeout:(int)aTimeout;
+ findCommForPort:(port_t)aPort;

@end

/*****************       Remote Objects  *************************/

@interface RemoteObject: Object {
    Communication       *comm;  /* nil means "local" */
    unsigned            name;           /* object name; 0 means localRoot */
    HashTable           *knownSelectors;        /* cache */
}

+ messageReceived:(msg_header_t *)msg;
+ newRemote:(unsigned)remoteName withCommunication:(Communication *)communication;
    /* This is used only for bootstrap */

+ registerLocalRoot:root;
    /* Register the local root, and return a remote object with name 0;
    Can only be called once */

+ newLocal:local withCommunication:(Communication *)communication;
    /* search for a local-RemoteObject that corresponds to local id.
    If none found, creates one */

- (unsigned)remoteObjectName;

- (unsigned) methodArgSize: (SEL) sel;
    /* Size of the arguments of the remote object, including self and sel;
    0 iff error */
- forward: (SEL) sel : (void *) args;
@end

/*****************        Encoding Protocol        **************************/

@interface Object (Object_MakeRemote)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag;
    /* This method is called for each object being encoded; By default, it consists in cre
ating a new "local" remote object (i.e. passing the object by reference).  To pass the obj
ect by value, just return the object.  To substitute another object, just return it.
    Iff flag is set, the returned object will be freed after encoding. */

- afterPortReading:(Communication *)communication;
    /* This method is called after decoding an object to give an opportunity to replace it
; original object can be freed */

- instantiateObject:(const char *)className;
- setOutlet:(const char *)outletName with:dest;
@end
```

```
#import "RemoteObject.h"
#import "NXPortStream.h"

#import <string.h>
#import <stdio.h>
#import <libc.h>
#import <cthreads.h>
#import <mach.h>
#import <syslog.h>
#import <objc/error.h>
#import <objc/List.h>
#import <objc/objc-runtime.h>
#import <kern/mach_param.h>
#import <sys/message.h>

/*****************        Forward Definitions        ***************************/

static void handleRemoteMessage(msg_header_t *m, void *userData);
static void remoteAsk (port_t port, msg_header_t *msg, int timeout);
#if 0 // Never tried
static void remoteTell(port_t target, msg_header_t *msg, port_t sender, int timeout);
#endif

/*****************        Utilities        **************************/

int defaultCommTimeout = 15000;
BOOL enableWarning = NO;

static void logv(const char *format, va_list args) {
    printf("RemoteObjects[pid %d]:\t", getpid());
    vprintf(format, args);
}

static void ROLog(const char *format, ...) {
    va_list     args;
    va_start(args, format);
    logv(format, args);
    va_end(args);
}

static void warning(const char *format, ...) {
    va_list     args;
    va_start(args, format);
    if (enableWarning) logv(format, args);
    va_end(args);
}

static void error(const char *format, ...) {
    va_list     args;
    va_start(args, format);
    logv(format, args);
    va_end(args);
    NX_RAISE(GENERIC_REMOTE_EXCEPTION, "Internal error", NULL);
}

static int generateSequenceNumber() {
    static int number=0;
    number++;
    if (!number) number = 1;    /* useless */
    return number;
}

@interface Object (Private_Imports)
- reallyFree;
@end

/*****************      Selector Info    *************************/

@interface RemoteMethodInfo: Object {
    NXAtom      typedesc;
}

static RemoteMethodInfo *knownRemoteMethodInfo = nil;

+ localMethodInfoFor:(Class)class :(SEL)sel;
    /* Return RemoteMethodInfo for a local method;
    Can return nil */

- encodeMethodParams:(void *)args onto:(NXPortStream *)stream;
    /* encode the method frame onto stream (excluding self and sel) */

- (void *)decodeMethodParamsFrom:(NXPortStream *)stream;
    /* decode the method frame from stream;
    return a freshly malloced pointer, never NULL (unless error) */

- encodeMethodRet:result onto:(NXPortStream *)stream;
    /* encode the method return value */

- decodeMethodRetFrom:(NXPortStream *)stream;
    /* decode the return value. */

- (unsigned)sizeOfParams;
    /* Return the size of all parameters including self and sel */
@end

@implementation RemoteMethodInfo

static unsigned hashMethodInfo(const void *info, const void *data) {
    const RemoteMethodInfo      *rm = data;
    return (unsigned)rm->typedesc;      /* depends on the fact its uniqeud */
}
static int isEqualMethodInfo(const void *info, const void *data1, const void *data2) {
    const RemoteMethodInfo      *rm1 = data1;
    const RemoteMethodInfo      *rm2 = data2;
    return (rm1->typedesc == rm2->typedesc);
}

static NXHashTablePrototype proto = {hashMethodInfo, isEqualMethodInfo, NXNoEffectFree, 0}
;

static NXHashTable *allMethodInfos = NULL;
+ initialize {
    if (! knownRemoteMethodInfo) {
        allMethodInfos = NXCreateHashTable(proto, 0, NULL);
        knownRemoteMethodInfo = [RemoteMethodInfo localMethodInfoFor: (Class) [Object clas
s] :@selector(remoteMethodInfo:)];
    }
    return self;
}

+ localMethodInfoFor:(Class)class : (SEL)sel {
    Method      method = class_getInstanceMethod(class, sel);
    RemoteMethodInfo    *previous;
```

```
     if (! method) {
         error("*** localMethodInfoFor:: for '%s' with '%s' return nil\n", [(id) class name
 ], sel_getName(sel));
         return nil;
     }
     self = [super new];
     typedesc = NXUniqueString(method->method_types);
     previous = NXHashGet(allMethodInfos, self);
     if (previous) {[self free]; return previous; }
     NXHashInsert(allMethodInfos, self);
     return self;
 }

 - encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
     return self;
 }

 - writePortStream: (NXPortStream *) stream {
     [super writePortStream: stream];
     warning("writing a Method:%s\n", typedesc);
     NXWritePortTypes(stream, "%", &typedesc);
     return self;
 }

 - readPortStream: (NXPortStream *) stream {
     [super readPortStream: stream];
     NXReadPortTypes(stream, "%", &typedesc);
     warning("reading a Method:%s\n", typedesc);
     return self;
 }

 - encodeMethodParams:(void *)args onto:(NXPortStream *)stream {
     struct objc_method  met;
     unsigned       nb;
     unsigned       index = 2; /* skip result, self and sel */
     int            offset0;
     char           *type;
     met.method_types = (char *)typedesc;
     nb = method_getNumberOfArguments(&met);
     NXWritePortTypes(stream, "i", &nb); /* just for redundancy */
     method_getArgumentInfo(&met, 0, &type, &offset0);
     while (index < nb) {
         char             *type;
         int              offset = 0;
         void             *arg;
         method_getArgumentInfo(&met, index, &type, &offset);
         if (! offset) error("*** encodeMethodParams:onto: cannot extract %d argument for t
ype desc %s\n", index, typedesc);
         arg = ((char *)args)+offset-offset0;
         warning("encodeMethodParams:onto: type=%s value=0x%x\n", type, *(void **)arg);
         NXWritePortTypeInternal(stream, type, arg);
         index++;
     }
     return self;
}

- (void *)decodeMethodParamsFrom:(NXPortStream *)stream {
    struct objc_method  met;
    unsigned       nb;
    unsigned       index = 2;
    unsigned       count;
    void           *args;
    int            offset0;
    char           *type;
```

```
        met.method_types = (char *)typedesc;
        nb = method_getNumberOfArguments(&met);
        NXReadPortTypes(stream, "i", &count);
        if (count != nb) {
            error("decodeMethodParamsFrom: incompatible method params");
            return NULL;
        }
        method_getArgumentInfo(&met, 0, &type, &offset0);
        args = calloc([self sizeOfParams], 1);
        while (index < nb) {
            char            *type;
            int             offset = 0;
            void            *arg;
            method_getArgumentInfo(&met, index, &type, &offset);
            if (! offset) error("*** decodeMethodParamsFrom: cannot extract %d argument for ty
pe desc %s\n", index, typedesc);
            arg = ((char *)args)+offset-offset0;
            NXReadPortTypeInternal(stream, type, arg);
            warning("decodeMethodParamsFrom: type=%s value=0x%x\n", type, *(void **)arg);
            index++;
        }
        return args;
}

- encodeMethodRet:result onto:(NXPortStream *)stream {
//?? -> SN: way to get return types?
    if (typedesc[0] == 'v') return self;
    if (typedesc[0] == 'c') {
        //?? BOGUS, of course
        NXWritePortTypeInternal(stream, "i", &result);
    } else {
        NXWritePortTypeInternal(stream, typedesc, &result);
    }
    return self;
}

- decodeMethodRetFrom:(NXPortStream *)stream {
    id          result = nil;/* important init. for result less than 4 bytes */
//?? -> SN: what iff result more than 4 bytes?
    if (typedesc[0] == 'v') return nil;
    if (typedesc[0] == 'c') {
        //?? BOGUS, of course
        NXReadPortTypeInternal(stream, "i", &result);
    } else {
        NXReadPortTypeInternal(stream, typedesc, &result);
    }
    return result;
}

- (unsigned)sizeOfParams {
    struct objc_method  met;
    met.method_types = (char *)typedesc;
    return method_getSizeOfArguments(&met);
}

@end

@interface Object (Object_RemoteMethodInfo)
- remoteMethodInfo:(SEL)sel;
@end

@implementation Object (Object_RemoteMethodInfo)

- remoteMethodInfo:(SEL)sel {
    return [RemoteMethodInfo localMethodInfoFor:(Class)[self class] :sel];
}

@end

/*****************       Communication    **************************/

@implementation Communication

typedef struct _LocalToRemote {
    id                  local;
    RemoteObject        *remote;        /* remote->name == (unsigned)local */
} LocalToRemote;

static void freeLocalToRemote(const void *info, void *data) {
    LocalToRemote       *ltr = data;
    [ltr->remote reallyFree];
    free(data);
}

static NXHashTablePrototype proxyProto;

static id commList = nil;

static receive_enable_proc_t enableProc = NULL;
static port_t replyPort = PORT_NULL;

+ beginListeningOn:(port_t)listenPort enableProc:(receive_enable_proc_t)aProc {
    replyPort = listenPort;
    port_set_backlog(task_self(),listenPort,PORT_BACKLOG_MAX);
    enableProc = aProc;
    if (enableProc) (*enableProc)(replyPort,handleRemoteMessage,YES);
    return self;
}

+ new:(port_t)port timeout:(int)aTimeout {
    NXHashTablePrototype        proto1 = NXPtrStructKeyPrototype;
    proto1.free = freeLocalToRemote;
    if (! commList) commList = [List new];
    self = [super new];
    sendPort = port;
    timeout = aTimeout;
    objectsGivenAway = NXCreateHashTable(proto1, 0, NULL);
    allProxies = NXCreateHashTable(proxyProto, 0, NULL);
    [commList addObject:self];
    return self;
}

+ findCommForPort:(port_t)aPort {
    int index = [commList count];
    while (index--)
        if (((Communication *)[commList objectAt:index])->sendPort == aPort)
            return [commList objectAt:index];
    return nil;
}

- free {
    [commList removeObject:self];
    NXFreeHashTable(objectsGivenAway);
    NXFreeHashTable(allProxies);
    return [super free];
}

- beforeEncoding:object onto:(NXPortStream *)stream freeAfterEncoding:(BOOL *)flag {
    return [object encodeRemotelyFor: stream->communication freeAfterEncoding:flag];
}

- afterDecoding:object from:(NXPortStream *)stream {
    return [object afterPortReading: stream->communication];
}

@end

/*****************     Remote Objects     ************************/

static id localRoot = nil;
static id localRemoteForRoot = nil;

@implementation RemoteObject

+ initialize {
    /* We have to initialize knownRemoteMethodInfo! */
    [RemoteMethodInfo initialize];
    return self;
}

+ newRemote:(unsigned)remoteName withCommunication:(Communication *)communication {
    self = [super new];
    comm = communication;
    name = remoteName;
    if (NXHashInsert(communication->allProxies, self)) error("newRemote: already in table!
");
    return self;
}

+ newLocal:local withCommunication:(Communication *)communication {
    LocalToRemote        pseudo;
    LocalToRemote        *ltr;
    if (! local) return nil;
    if (local == localRoot) return localRemoteForRoot;
    pseudo.local = local;
    ltr = NXHashGet(communication->objectsGivenAway, &pseudo);
    if (ltr) return ltr->remote;
    ltr = malloc(sizeof(LocalToRemote));
    ltr->local = local;
    ltr->remote = [self new];
    ltr->remote->name = (unsigned)local;
    NXHashInsert(communication->objectsGivenAway, ltr);
    return ltr->remote;
}

+ registerLocalRoot:root {
    if (localRoot) error("registerLocalRoot: root registered twice!");
    localRoot = root;
    return (localRemoteForRoot = [self new]);
}

+ messageReceived:(msg_header_t *)msg {
    handleRemoteMessage(msg, NULL);
    return self;
}

- reallyFree {
    [knownSelectors free];
    return [super free];
}

- free {
    syslog(LOG_ERR, "Remote Object 0x%x received free", self);
    return nil;
}

static unsigned hashProxy(const void *info, const void *data) {
    return ((RemoteObject *)data)->name;
}

static int isEqualProxy(const void *info, const void *data1, const void *data2) {
    return ((RemoteObject *)data1)->name == ((RemoteObject *)data2)->name;
}

static void freeProxy(const void *info, const void *data) {
    [(id)data reallyFree];
}

static NXHashTablePrototype proxyProto = {hashProxy, isEqualProxy, freeProxy, 0};

- (unsigned)remoteObjectName { return name; }

- remoteMethodInfo:(SEL)sel {
    id          res;
    id          args[4];
    if (sel == @selector(remoteMethodInfo:)) return knownRemoteMethodInfo; /* to avoid inf
inite recursion */
    res = [knownSelectors valueForKey:(void *)sel];
    if (res) return res;
//?? -> SN How to fill args cleanly
    bzero(args, sizeof(id)*4);
    {
        Method  method = class_getInstanceMethod((Class)[Object class], @selector(remoteMe
thodInfo:));
        char            *type;
        int             offset2;
        int             offset0;
        SEL             *ref;
        method_getArgumentInfo(method, 0, &type, &offset0);
        method_getArgumentInfo(method, 2, &type, &offset2);
        ref = (SEL *) (((char *)args)+offset2-offset0);
        *ref = sel;
    }
    res = [(id) self forward:@selector(remoteMethodInfo:) :args];
    if (! knownSelectors) knownSelectors = [HashTable newKeyDesc:":"];
    [knownSelectors insertKey:(void *)sel value:res];
    return res;
}

- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    return self;
}

- writePortStream: (NXPortStream *) stream {
    [super writePortStream: stream];
    NXWritePortTypes (stream, "ii", &comm, &name);
    return self;
}

- readPortStream: (NXPortStream *) stream {
    [super readPortStream: stream];
    NXReadPortTypes (stream, "ii", &comm, &name);
    return self;
}

- afterPortReading:(Communication *)communication {
    if (! comm) {
        id      previous;
        /* it was local for the other guy */
        warning("in afterPortReading - read Remote %d\n", name);
        comm = communication;
        previous = NXHashGet(communication->allProxies, self);
        if (previous) {
            warning("receiving same name=0x%x previous=0x%x self=0x%x\n", name, previous,
self);
            [self reallyFree];
            return previous;
        }
        NXHashInsert(communication->allProxies, self);
        return self;
    } else {
        LocalToRemote   pseudo;
        LocalToRemote   *ltr;
        pseudo.local = (id)name;
        if (! name) {
            if (! localRoot) error("invariant broken in afterPortReading:");
            [self reallyFree];
            return localRoot;
        }
        ltr = NXHashGet(communication->objectsGivenAway, &pseudo);
        if (! ltr) error("afterPortReading:");
        if ((unsigned)ltr->local != name) {
            error("afterPortReading: broken invariant");
        }
        warning("in afterPortReading - converting Remote %d into local 0x%x\n", name, ltr-
>local);
        [self reallyFree];
        return ltr->local;
    }
}

- (unsigned) methodArgSize: (SEL) sel {
    RemoteMethodInfo    *selInfo = [self remoteMethodInfo:sel];
    if (! selInfo) return 0;
    return [selInfo sizeOfParams];
}


- forward:(SEL)sel :(void *)args {
    char                buffer[MSG_SIZE_MAX];
    msg_header_t        *msg = (msg_header_t *)buffer;
    NXAtom              selName = NXUniqueString(sel_getName (sel));
    int                 sequence = generateSequenceNumber();
    NXPortStream        *stream = NXOpenEncodePortStream(msg, sequence, comm);
    RemoteMethodInfo    *selInfo = [self remoteMethodInfo:sel];
    id                  result;
    int                 errorCode;

    if (! selInfo) error("forward:: cannot find remote selector %s", selName);
    warning("entered forward:: self=%d selName=%s\n", name, selName);
    NXWritePortTypes(stream, "@%", &self, &selName);
    [selInfo encodeMethodParams:args onto:stream];
    NXCloseEncodePortStream(stream);
    warning("in forward:: - %d made packet for [0x%x comm:%x %s ...]\n", name, comm, self,
selName);

    remoteAsk(comm->sendPort, msg, comm->timeout);

    /* let's decode the result */
    warning("in forward:: selName=%s received answer\n", selName);
```

```
     stream = NXOpenDecodePortStream(msg, comm);
     NXReadPortTypes (stream, "i", &errorCode);
     if (errorCode) error("forward:: Error occurred during remote execution");
     result = [selInfo decodeMethodRetFrom:stream];
     NXCloseDecodePortStream(stream);
     warning("in forward:: - %d result decoded for [0x%x comm:%x %s ...]\n", name, comm, se
lf, selName);
     return result;
}

@end

/****************         Object Misc        *************************/

@interface Object (Object_MakeRemote)
- setAction: (SEL) theSelector;
@end

@implementation Object (Object_MakeRemote_Import)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    warning("in encodeRemotelyFor - converting local 0x%x (%s) into remote\n", self, [self
 name]);
    return [RemoteObject newLocal:self withCommunication:communication];
}
- afterPortReading:(Communication *)communication {
    return self;
}

- instantiateObject:(const char *)className {
    return [objc_getClass((char *)className) new];
}

- setOutlet:(const char *)outletName with:dest {
    char        methodString[256];
    SEL         sel;
    strcpy(methodString, "set");
    strcat(methodString, outletName);
    strcat(methodString, ":");
    if (methodString[3] >= 'a' && methodString[3] <= 'z')
        methodString[3] += 'A' - 'a';
    sel = _sel_registerName((char *)NXUniqueString(methodString));
#if 0
    if ([self respondsTo:sel]) {
        [self perform:sel with:dest];
    } else {
        object_setInstanceVariable(self, methodString, dest);
    }
#else
    [self perform:sel with:dest];
#endif
    return self;
}

@end

/****************         Message transport        *************************/

#define RO_TELL_MSG_ID (232323)
#define RO_ASK_MSG_ID (323232)
#define RO_REPLY_MSG_ID (233223)

#define DEFAULT_TIMEOUT (15000)

static void remoteReply(port_t rPort, msg_header_t *msg, int timeout);
```

```
  static void handleRemoteAsk(msg_header_t *msg, port_t sender, id communication) {
      /*
      ** perform a remote RPC.
      ** This function may call many callouts to enableRemoteListening and
      ** disableRemoteListening  as it recurses.
      ** exceptions may arise.
      */
      NXAtom                  selName;
      id                      volatile result = nil;
      NXPortStream            *stream = NXOpenDecodePortStream(msg, communication);
      char                    *args;
      id                      self;
      SEL                     sel;
      int                     errorCode = 0;
      RemoteMethodInfo        *selInfo;
      int                     sequence = NXGetPortStreamSequence(stream);
      char                    buffer[MSG_SIZE_MAX];

      warning("in remoteAnswer received packet\n");
      NXReadPortTypes(stream, "@%", &self, &selName);
      warning("in remoteAnswer selName=%s\n", selName);
      sel = sel_getUid((char *) selName);
      if (! sel) error("handleRemoteAsk: received message with unknown sel");
      selInfo = [self remoteMethodInfo:sel];
      args = [selInfo decodeMethodParamsFrom:stream];
      NXCloseDecodePortStream(stream);
      if (! args) { errorCode = -2; goto done; }
      NX_DURING
          result = objc_msgSendv(self, sel, [selInfo sizeOfParams], args);
      NX_HANDLER
          ROLog("**** Error while excuting remote message for [0x%x %s ...]\n", self, selName
 );
          errorCode = -1;
      NX_ENDHANDLER;
      free(args);

      /* let's encode the result */
  done:
      /* we cannot reuse msg here, regrettably, because if we come from DPSClient, we don't
have an 8K buffer but a copy only big enough for the incoming msg; sigh */
      msg = (msg_header_t *)buffer;
      stream = NXOpenEncodePortStream(msg, sequence, communication);
      NXWritePortTypes (stream, "i", &errorCode);
      [selInfo encodeMethodRet:result onto:stream];
      NXCloseEncodePortStream (stream);
      remoteReply(sender, msg, ((Communication*)communication)->timeout);
      warning("in remoteAnswer made return packet\n");
}

static void handleRemoteTell(msg_header_t *msg, port_t sender, id communication) {
    const char          *selName;
    NXPortStream        *stream = NXOpenDecodePortStream(msg, communication);
    char                *args;
    id                  self;
    SEL                 sel;
    int                 volatile errorCode = 0;
    RemoteMethodInfo    *selInfo;

    warning("in handleRemoteTell received packet\n");
    NXReadPortTypes(stream, "@%", &self, &selName);
    warning("in handleRemoteTell selName=%s\n", selName);
    sel = sel_getUid ((char *) selName);
    if (! sel) error("handleRemoteTell: received message with unknown sel");
```

```
        selInfo = [self remoteMethodInfo:sel];
        args = [selInfo decodeMethodParamsFrom:stream];
        NXCloseDecodePortStream (stream);
        if (! args) { errorCode = -2; goto done; }
        NX_DURING
            objc_msgSendv(self, sel, [selInfo sizeOfParams], args);
        NX_HANDLER
            errorCode = -1;
        NX_ENDHANDLER;
        free(args);
    done:
        if (errorCode) warning("**** Error executing handleRemoteTell %d\n", errorCode);
}

int remoteMessageReceiveCount = 0;
static void handleRemoteMessage(msg_header_t *msg, void *userData) {
    port_t        sender = msg->msg_remote_port;
    id            communication = [Communication findCommForPort:sender];
    if (! communication) {
        syslog(LOG_ERR, "Received message from zombie");
        return;
    }
    remoteMessageReceiveCount++;
    if (msg->msg_id == RO_TELL_MSG_ID) {
        handleRemoteTell(msg,sender, communication);
    } else if (msg->msg_id == RO_ASK_MSG_ID) {
        handleRemoteAsk(msg,sender, communication);
    } else
        syslog(LOG_ERR,"Bogus remote message");
}

#if 0 // Should work, but has never been used/tested
static void remoteTell(port_t target, msg_header_t *msg, port_t sender, int timeout) {
    int err, sndOptions = SEND_SWITCH;

    msg->msg_remote_port = target;
    msg->msg_local_port = sender;
    msg->msg_id = RO_TELL_MSG_ID;
    if (timeout >= 0)
        sndOptions |= SEND_TIMEOUT;
    err = msg_send(msg,sndOptions,timeout);
    if (err) error("remoteTell: cannot send");
}
#endif

static void remoteReply(port_t rPort, msg_header_t *msg, int timeout) {
    int err, sndOptions = SEND_SWITCH;
    msg->msg_remote_port = rPort;
    msg->msg_local_port = PORT_NULL;
    msg->msg_id = RO_REPLY_MSG_ID;
    if (timeout >= 0)
        sndOptions |= SEND_TIMEOUT;
    err = msg_send(msg,sndOptions,timeout);
    if (err) error("remoteReply: cannot send");
}

static nestingLevel=0;


static void remoteAsk(port_t target, msg_header_t *msg, int timeout) {
    int err;
    int volatile sndOptions = SEND_SWITCH;
    int volatile rcvOptions = RCV_NO_SENDERS | RCV_INTERRUPT;
    msg->msg_remote_port = target;
    msg->msg_local_port = replyPort;
    msg->msg_id = RO_ASK_MSG_ID;
    if (timeout >= 0) {
        sndOptions |= SEND_TIMEOUT;
        rcvOptions |= RCV_TIMEOUT;
    }
    nestingLevel++;
    if (nestingLevel == 1 && enableProc)
        (*enableProc)(replyPort,handleRemoteMessage,NO);
    NX_DURING
        err = msg_send(msg,sndOptions,timeout);
        if (err) error("remoteAsk: cannot send");
        else {
            while (1) {
                msg->msg_size = MSG_SIZE_MAX;
                msg->msg_local_port = replyPort;
                err = msg_receive(msg,rcvOptions,timeout);
                if (err) {
                    ROLog("remoteAsk: cannot receive or timeout\n");
                    NX_RAISE(TIMEOUT_REMOTE_EXCEPTION, "Cannot receive", NULL);
                } else {
                    if (msg->msg_id == RO_REPLY_MSG_ID) {
                        break;
                    } else
                        handleRemoteMessage((msg_header_t *)msg, NULL);
                        //?? IF OUT OF LINE, DEALLOCATE HERE
                }
            }
        }
        nestingLevel--;
        if (!nestingLevel && enableProc)
            (*enableProc)(replyPort,handleRemoteMessage,YES);
    NX_HANDLER
        nestingLevel--;
        if (!nestingLevel && enableProc)
            (*enableProc)(replyPort,handleRemoteMessage,YES);
        NX_RERAISE ();
    NX_ENDHANDLER;
}
```

```
/* This module simply allows NXStrings to be passed across address spaces.
The principle is: to encode a NXString, make a temporary object holding the string, encode
 its characters, free the temporary; to decode a NXString, decode the temporary object, re
place by an immutable string, free the temporary. */

#import "../lowlevel.subproj/NXString.h"

#import "RemoteObject.h"
#import "NXPortStream.h"

#import <string.h>
#import <stdio.h>

@interface _TemporaryStringHolder: Object {
    @public
    NXString    *string;
}
@end

@implementation _TemporaryStringHolder

- writePortStream:(NXPortStream *)stream {
    unsigned    length = [string length];
    NXChar      *chars = malloc(length*sizeof(NXChar));
    [super writePortStream: stream];
    [string setChars:chars];
    NXWritePortTypes(stream, "i", &length);
    NXPortEncodeBytes(stream, (char *)chars, length);
    free(chars);
    return self;
}

- readPortStream: (NXPortStream *) stream {
    unsigned    length;
    NXChar      *chars;
    [super readPortStream: stream];
    NXReadPortTypes(stream, "i", &length);
    chars = malloc(length*sizeof(NXChar));
    NXPortDecodeBytes(stream, (char *)chars, length);
    string = [NXImmutableString newFor:length chars:chars];
    free(chars);
    return self;
}

- afterPortReading:(Communication *)communication {
    NXString    *res = string;
    [self free];
    return res;
}

@end

@interface NXString (NXString_RemoteObject_Coding)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag;
@end

@implementation NXString (NXString_RemoteObject_Coding)
- encodeRemotelyFor:(Communication *)communication freeAfterEncoding:(BOOL *)flag {
    _TemporaryStringHolder      *new = [_TemporaryStringHolder new];
    new->string = self;
    *flag = YES;
    return new;
}

@end
```

```
*/

#import <objc/Object.h>
#import <objc/hashtable.h>
#import <sys/message.h>

extern SEL _sel_registerName(STR key);
    //?? will go away

/*****************      Definitions      **************************/

typedef struct _NXPortStream {
    msg_header_t        *msg;
//?? DO SIZE TEST LATER
    BOOL        write;         /* writing vs reading */        //REMOVE AFTER DEBUG!
    id          communication;
    char        *chars;
    int         nbchars;
    int         maxchars;
    int         *ints;
    int         nbints;
    int         maxints;
} NXPortStream;

@interface Object (Communication_Calls)
- beforeEncoding: object onto: (NXPortStream *) stream freeAfterEncoding:(BOOL *)flag;
    /* will encode the returned object; if flag is set, returned object will be send 'free
' after encoding */
- afterDecoding: object from: (NXPortStream *) stream;
    /* will replace the decoded object by the returned object */
@end

/*****************      global operations      ***********************/

extern NXPortStream *NXOpenEncodePortStream(msg_header_t *msg, int sequence, id communicat
ion);

extern NXPortStream *NXOpenDecodePortStream(msg_header_t *msg, id communication);
        /*  buffer is char[MSG_SIZE_MAX];

        If mode is NX_WRITEONLY, creates a NXPortStream, ready for writing, given a physic
al stream on which to actually put the bytes. If mode is NX_READONLY, creates a NXPortStre
am, ready for reading, given a physical stream on which to actually get the bytes.  The ca
ller is responsible for closing physical.  If the file format mismatches right from the st
art with stream format, NULL is returned, otherwise an exception might be raised.

        Iff read, have buffers point to msg */

extern void NXCloseEncodePortStream(NXPortStream *stream);
    /* Copy buffers into message and prepare msg for msg_send;
    free stream */

extern void NXCloseDecodePortStream(NXPortStream *stream);
    /* free stream */

extern int NXGetPortStreamSequence(NXPortStream *stream);

/*****************      Read/Write data      **************************/

void NXPortEncodeBytes(NXPortStream *stream, const char *bytes, int count);
```

```
*/

#import <objc/Object.h>
#import <objc/hashtable.h>
#import <sys/message.h>

extern SEL _sel_registerName(STR key);
    //?? will go away

/*****************    Definitions    **************************/

typedef struct _NXPortStream {
    msg_header_t        *msg;
//?? DO SIZE TEST LATER
    BOOL        write;          /* writing vs reading */        //REMOVE AFTER DEBUG!
    id          communication;
    char        *chars;
    int         nbchars;
    int         maxchars;
    int         *ints;
    int         nbints;
    int         maxints;
} NXPortStream;

@interface Object (Communication_Calls)
- beforeEncoding: object onto: (NXPortStream *) stream freeAfterEncoding:(BOOL *)flag;
    /* will encode the returned object; if flag is set, returned object will be send 'free
' after encoding */
- afterDecoding: object from: (NXPortStream *) stream;
    /* will replace the decoded object by the returned object */
@end

/*****************    global operations    **********************/

extern NXPortStream *NXOpenEncodePortStream(msg_header_t *msg, int sequence, id communicat
ion);

extern NXPortStream *NXOpenDecodePortStream(msg_header_t *msg, id communication);
        /*  buffer is char[MSG_SIZE_MAX];

        If mode is NX_WRITEONLY, creates a NXPortStream, ready for writing, given a physic
al stream on which to actually put the bytes. If mode is NX_READONLY, creates a NXPortStre
am, ready for reading, given a physical stream on which to actually get the bytes.  The ca
ller is responsible for closing physical.  If the file format mismatches right from the st
art with stream format, NULL is returned, otherwise an exception might be raised.

        Iff read, have buffers point to msg */

extern void NXCloseEncodePortStream(NXPortStream *stream);
    /* Copy buffers into message and prepare msg for msg_send;
    free stream */

extern void NXCloseDecodePortStream(NXPortStream *stream);
    /* free stream */

extern int NXGetPortStreamSequence(NXPortStream *stream);

/*****************    Read/Write data    **************************/

void NXPortEncodeBytes(NXPortStream *stream, const char *bytes, int count);
void NXPortDecodeBytes(NXPortStream *stream, char *bytes, int count);

extern void NXWritePortTypes(NXPortStream *stream, const char *type, ...);
        /* Restricted to monochar type descriptions;
        Last arguments specify addresses of values to be written.
        It might seem surprising to specify values by address, but this is extremely conve
nient for copy-paste with NXReadPortTypes calls.  A more down-to-the-earth cause for this
passing of addresses is that values of arbitrary size is not well supported in ANSI C for
functions with variable number of arguments. */

extern void NXReadPortTypes(NXPortStream *stream, const char *type, ...);
        /* Restricted to monochar type descriptions;
        Last arguments specify addresses of values to be read.  Expected type is checked a
gainst the type actually present on the stream. */

extern void NXWritePortTypeInternal(NXPortStream *stream, const char *type, const void *da
ta);
extern void NXReadPortTypeInternal(NXPortStream *stream, const char *type, void *data);

/***************          errors          ****************************/

/* several exceptions can occur; the format of all exceptions raised is a label, a message
 string and maybe some extra information */

#define PORTSTREAM_ERROR_BASE 37000      /* less than appkit base */

typedef enum _PortStreamErrors {
    PORTSTREAM_CALLER_ERROR = PORTSTREAM_ERROR_BASE,
    PORTSTREAM_INCONSISTENCY,
    PORTSTREAM_INTERNAL_ERROR
} PortStreamErrors;

@interface Object (Object_PortStream_Calls)
- writePortStream: (NXPortStream *) stream;
- readPortStream: (NXPortStream *) stream;
@end
```

```
/*      NXPortStream.m
        Copyright 1989, NeXT, Inc.
        Bertrand 1989

*/

#import "NXPortStream.h"

#import <stdlib.h>
#import <stdarg.h>
#import <stdio.h>
#import <string.h>
#import <syslog.h>
#import <mach.h>
#import <libc.h>
#import <objc/error.h>
#import <objc/objc-class.h>
#import <objc/objc-runtime.h>

#define DEBUG_LEAKS      0

/*****************       Utilities       ***************************/

static void BUG(const char *str) {
    syslog(LOG_ERR, "*** RemoteObjects: internal error in %s", str);
    NX_RAISE(PORTSTREAM_INTERNAL_ERROR, str, NULL);
}

static void checkExpected(const char *readType, const char *wanted) {
    if (readType == wanted) return;
    if (! readType || strcmp (readType, wanted)) BUG("checkExpected");
}

@implementation Object (Object_PortStream_Calls)
- writePortStream: (NXPortStream *) stream { return self; }
- readPortStream: (NXPortStream *) stream { return self; }
@end

/*****************       Definitions       ***************************/

typedef struct _remote_message_t {
    msg_header_t         header;
    msg_type_t           sequenceType;
    int                  sequence;
    /* following is chars type and chars, int types and ints, etc ... */
} remote_message_t;

static void PortInternalWriteObject(NXPortStream *stream, id object);
    /* write an object without header */
static id PortInternalReadObject(NXPortStream *stream);
    /* read an object without header */

/*****************       Coding data       ***************************/

void NXPortEncodeBytes(NXPortStream *stream, const char *buf, int count) {
    while (stream->nbchars + count > stream->maxchars) {
        stream->maxchars += stream->maxchars + 1;
        stream->chars = realloc(stream->chars, stream->maxchars);
    }
    bcopy(buf, stream->chars + stream->nbchars, count);
    stream->nbchars += count;
}

void NXPortDecodeBytes(NXPortStream *stream, char *buf, int count) {
    if (count > stream->nbchars) BUG("NXPortDecodeBytes");
    bcopy(stream->chars, buf, count);
    stream->chars += count;
    stream->nbchars -= count;
}

static inline void _NXEncodeChar(NXPortStream *stream, signed char ch) {
    if (stream->nbchars + 1 > stream->maxchars) {
        stream->maxchars += stream->maxchars + 1;
        stream->chars = realloc(stream->chars, stream->maxchars);
    }
    stream->chars[stream->nbchars ++] = ch;
}

static inline signed char _NXDecodeChar(NXPortStream *stream) {
    if (stream->nbchars-- <= 0) BUG("_NXDecodeChar");
    return *(stream->chars ++);
}

static void _NXEncodeInt(NXPortStream *stream, int x) {
    if (stream->nbints + 1 > stream->maxints) {
        stream->maxints += stream->maxints + 1;
        stream->ints = realloc(stream->ints, stream->maxints * sizeof(int));
    }
    stream->ints[stream->nbints++] = x;
}

static int _NXDecodeInt(NXPortStream *stream) {
    if (stream->nbints-- <= 0) BUG("_NXDecodeInt");
    return *(stream->ints++);
}

static void _NXEncodeBool(NXPortStream *stream, BOOL x) {
    _NXEncodeInt(stream, (x) ? 1 : 0);
}

static BOOL _NXDecodeBool(NXPortStream *stream) {
    int         x = _NXDecodeInt(stream);
    if (x && x!=1) BUG("_NXDecodeBool");
    return x;
}

static inline void _NXEncodeFloat(NXPortStream *stream, float x) {
    NXPortEncodeBytes(stream, (char *) &x, sizeof(float));
}

static inline float _NXDecodeFloat(NXPortStream *stream) {
    float       x;
    NXPortDecodeBytes(stream, (char *) &x, sizeof(float));
    return x;
}

static inline void _NXEncodeDouble(NXPortStream *stream, double x) {
    NXPortEncodeBytes(stream, (char *) &x, sizeof(double));
}

static inline double _NXDecodeDouble(NXPortStream *stream) {
    double      x;
    NXPortDecodeBytes(stream, (char *) &x, sizeof(double));
    return x;
}

static void _NXEncodeChars(NXPortStream *stream, const char *str) {
    _NXEncodeBool(stream, (str) ? YES : NO);
    if (str) {
        int     len = strlen (str);
        _NXEncodeInt(stream, len);
        NXPortEncodeBytes(stream, str, len);
    }
}

static char *_NXDecodeChars(NXPortStream *stream) {
    if (! _NXDecodeBool(stream)) return NULL;
    else {
        int      len = _NXDecodeInt(stream);
        char     *str = (STR) malloc(len+1);
        NXPortDecodeBytes(stream, str, len);
        str[len] = '\0';
        return str;
    }
}

static NXAtom _NXDecodeUniqueString(NXPortStream *stream) {
    char           *new = _NXDecodeChars(stream);
    NXAtom         atom = NXUniqueString(new);
    free(new);
    return atom;
}

/*******          global operations        *******/

NXPortStream *NXOpenEncodePortStream(msg_header_t *msg, int sequence, id communication) {
    NXPortStream          *stream = calloc(sizeof(NXPortStream), 1);
    remote_message_t      *head = (remote_message_t *)msg;
    if (! communication) BUG("NXOpenEncodePortStream");
    stream->communication = communication;
    stream->write = YES;
    stream->msg = msg;

    head->sequenceType.msg_type_name = MSG_TYPE_INTEGER_32;
    head->sequenceType.msg_type_size = sizeof(int) * 8;
    head->sequenceType.msg_type_number = 1;
    head->sequenceType.msg_type_inline = TRUE;
    head->sequenceType.msg_type_longform = FALSE;
    head->sequenceType.msg_type_deallocate = FALSE;
    head->sequence = sequence;

    return stream;
}

NXPortStream *NXOpenDecodePortStream(msg_header_t *msg, id communication) {
    void                  *head = ((void *)msg)+sizeof(remote_message_t);
    NXPortStream          *stream = calloc(sizeof(NXPortStream), 1);
    if (! communication) BUG("NXOpenDecodePortStream");
    stream->communication = communication;
    stream->write = NO;
    stream->msg = msg;

    if (msg->msg_simple) {
        msg_type_t                *type;
        type = head;
        if (type->msg_type_name != MSG_TYPE_BYTE)
            BUG("NXOpenDecodePortStream-char");
        stream->nbchars = type->msg_type_number;
        head += sizeof(msg_type_t);
        stream->chars = head;
        head += stream->nbchars;
        type = head;
        if (type->msg_type_name != MSG_TYPE_INTEGER_32)
            BUG("NXOpenDecodePortStream-int");
        stream->nbints = type->msg_type_number;
        head += sizeof(msg_type_t);
        stream->ints = head;
        head += stream->nbints * sizeof(int);

    } else {
        msg_type_long_t         *type;
        type = head;
        if (type->msg_type_long_name != MSG_TYPE_BYTE)
            BUG("NXOpenDecodePortStream-char");
        stream->nbchars = type->msg_type_long_number;
        head += sizeof(msg_type_long_t);
        stream->chars = ((char **)head)[0];
        head += sizeof(int);

        type = head;
        if (type->msg_type_long_name != MSG_TYPE_INTEGER_32)
            BUG("NXOpenDecodePortStream-int");
        stream->nbints = type->msg_type_long_number;
        head += sizeof(msg_type_long_t);
        stream->ints = ((int **)head)[0];
        head += sizeof(int);

    }
    return stream;
}

void NXCloseEncodePortStream(NXPortStream *stream) {
    void                *head = ((void *)stream->msg)+sizeof(remote_message_t);
    if (! stream->write) BUG("NXCloseEncodePortStream");
    /* We pad chars to a multiple of 4; what a hack! */
    stream->nbchars = ((stream->nbchars + 3)/4)*4;
    stream->chars = realloc(stream->chars, stream->nbchars);

    /* We test for out-of-line */
    stream->msg->msg_simple = (sizeof(remote_message_t)+sizeof(msg_type_t)+stream->nbchars
+sizeof(msg_type_t)+stream->nbints * sizeof(int) < MSG_SIZE_MAX) && (stream->nbchars <= 40
95) && (stream->nbints <= 4095);

    if (stream->msg->msg_simple) {
        msg_type_t              *type;
        type = head;
        type->msg_type_name = MSG_TYPE_BYTE;
        type->msg_type_size = 8;
        type->msg_type_number = stream->nbchars;
        type->msg_type_inline = TRUE;
        type->msg_type_longform = FALSE;
        type->msg_type_deallocate = FALSE;
        head += sizeof(msg_type_t);
        bcopy(stream->chars, head, stream->nbchars);
        head += stream->nbchars;
        free(stream->chars);

        type = head;
        type->msg_type_name = MSG_TYPE_INTEGER_32;
        type->msg_type_size = sizeof(int) * 8;
        type->msg_type_number = stream->nbints;
        type->msg_type_inline = TRUE;
        type->msg_type_longform = FALSE;
        type->msg_type_deallocate = FALSE;
        head += sizeof(msg_type_t);
```

```
            bcopy(stream->ints, head, stream->nbints * sizeof(int));
            head += stream->nbints * sizeof(int);
            free(stream->ints);

        } else {
            msg_type_long_t         *type;
            vm_address_t            buffer;
            type = head;
            type->msg_type_header.msg_type_name = 0;
            type->msg_type_header.msg_type_size = 0;
            type->msg_type_header.msg_type_number = 0;
            type->msg_type_header.msg_type_inline = FALSE;
            type->msg_type_header.msg_type_longform = TRUE;
            type->msg_type_header.msg_type_deallocate = TRUE;
            type->msg_type_long_name = MSG_TYPE_BYTE;
            type->msg_type_long_size = 8;
            type->msg_type_long_number = stream->nbchars;
            head += sizeof(msg_type_long_t);
            if (vm_allocate(task_self(), (vm_address_t *)&buffer, stream->nbchars, 1) != KERN_
SUCCESS)
                BUG("NXCloseEncodePortStream: can't allocate (chars)");
            bcopy(stream->chars, (char *)buffer, stream->nbchars);
            ((int *)head)[0] = buffer;
            head += sizeof(int);
            free(stream->chars);

            type = head;
            type->msg_type_header.msg_type_name = 0;
            type->msg_type_header.msg_type_size = 0;
            type->msg_type_header.msg_type_number = 0;
            type->msg_type_header.msg_type_inline = FALSE;
            type->msg_type_header.msg_type_longform = TRUE;
            type->msg_type_header.msg_type_deallocate = TRUE;
            type->msg_type_long_name = MSG_TYPE_INTEGER_32;
            type->msg_type_long_size = sizeof(int) * 8;
            type->msg_type_long_number = stream->nbints;
            head += sizeof(msg_type_long_t);
            if (vm_allocate(task_self(), (vm_address_t *)&buffer, stream->nbints * sizeof(int)
, 1) != KERN_SUCCESS)
                BUG("NXCloseEncodePortStream: can't allocate (ints)");
            bcopy(stream->ints, (char *)buffer, stream->nbints * sizeof(int));
            ((int *)head)[0] = buffer;
            head += sizeof(int);
            free(stream->ints);

        }
        stream->msg->msg_type = MSG_TYPE_NORMAL;
        stream->msg->msg_size = head - (void *)stream->msg;

        free(stream);
}

void NXCloseDecodePortStream(NXPortStream *stream) {
    if (stream->write) BUG("NXCloseDecodePortStream");
    if (! stream->msg->msg_simple) {
        void            *head = ((void *)stream->msg)+sizeof(remote_message_t);
        msg_type_long_t *type;
        type = head;
        if (type->msg_type_long_name != MSG_TYPE_BYTE)
            BUG("NXOpenDecodePortStream-char");
        head += sizeof(msg_type_long_t);
        if (vm_deallocate(task_self(), ((int *)head)[0], type->msg_type_long_number) != KE
RN_SUCCESS)
            BUG("NXCloseDecodePortStream: can't deallocate (chars)");
        head += sizeof(int);

        type = head;
        if (type->msg_type_long_name != MSG_TYPE_INTEGER_32)
            BUG("NXOpenDecodePortStream-int");
        head += sizeof(msg_type_long_t);
        if (vm_deallocate(task_self(), ((int *)head)[0], type->msg_type_long_number * size
of(int)) != KERN_SUCCESS)
            BUG("NXCloseDecodePortStream: can't deallocate (ints)");
        head += sizeof(int);

    }
    free(stream);
}

int NXGetPortStreamSequence(NXPortStream *stream) {
    remote_message_t    *head = (remote_message_t *)stream->msg;
    return head->sequence;
}

/*******        Writing and reading arbitrary data      *******/

void NXWritePortTypeInternal(NXPortStream *stream, const char *type, const void *data) {
    switch (*type) {
        case 'c': case 'C': {
            char        cc = 0; /* for padding */
            cc = *((char *)data);
            _NXEncodeInt(stream, cc);
            break;
        }
        case 's': case 'S': {
            short       ss = 0; /* for padding */
            ss = *((short *)data);
            _NXEncodeInt(stream, ss);
            break;
        }
        case 'i': case 'I': case 'l': case 'L':
            _NXEncodeInt(stream, *((int *)data));
            break;
        case 'f':
            _NXEncodeFloat(stream, *((float *)data));
            break;
        case 'd':
            _NXEncodeDouble(stream, *((double *)data));
            break;
        case '@':
            PortInternalWriteObject(stream, *((id *)data));
            break;
        case '*':
#if DEBUG_LEAKS
            syslog(LOG_ERR, "*** Remote Objects: un-freeable string encoding: %s", *((char
**)data));
#endif
            _NXEncodeChars(stream, *((char **)data));
            break;
        case '%':
            _NXEncodeChars(stream, *((NXAtom *)data));
            break;
        case ':': {
            SEL         sel = *((SEL *)data);
            char        *str = NULL;
            if (sel) str = sel_getName (sel);
            _NXEncodeChars(stream, str);
            break;
            }
            case '!':
                break;
            default: BUG("NXWritePortTypeInternal: unknown type descriptor");
        }
    }

    void NXReadPortTypeInternal(NXPortStream *stream, const char *type, void *data) {
        switch (*type) {
            case 'c': case 'C': {
                signed char         *ptr = data;
                *ptr = _NXDecodeInt(stream);
                break;
            }
            case 's': case 'S': {
                short          *ptr = data;
                *ptr = _NXDecodeInt(stream);
                break;
            }
            case 'i': case 'I': case 'l': case 'L': {
                int            *ptr = data;
                *ptr = _NXDecodeInt(stream);
                break;
            }
            case 'f': {
                float          *ptr = data;
                *ptr = _NXDecodeFloat(stream);
                break;
            }
            case 'd': {
                double         *ptr = data;
                *ptr = _NXDecodeDouble(stream);
                break;
            }
            case '@': {
                id             *ptr = data;
                *ptr = PortInternalReadObject(stream);
                break;
            }
            case '*': {
                char           **ptr = data;
                *ptr = _NXDecodeChars(stream);
                break;
            }
            case '%': {
                NXAtom         *ptr = data;
                *ptr = _NXDecodeUniqueString(stream);
                break;
            }
            case ':': {
                SEL            *ptr = data;
                NXAtom         selName = _NXDecodeUniqueString(stream);
                *ptr = _sel_registerName((char *)selName);
                break;
            }
            case '!':
                break;
            default: BUG("NXReadPortTypeInternal: unknown type descriptor");
        }
    }

void NXWritePortTypes(NXPortStream *stream, const char *type, ...) {
    va_list     args;
    va_start(args, type);
    /* We could avoid next line at the cost of more painful debug */
    _NXEncodeChars(stream, type);
    while (*type) {
        NXWritePortTypeInternal(stream, type, va_arg(args, void *));
        type = type++; /* we restrict to monochar type descriptions */
    }
    va_end (args);
}

void NXReadPortTypes(NXPortStream *stream, const char *type, ...) {
    NXAtom      readType;
    va_list     args;
    va_start(args, type);
    readType = _NXDecodeUniqueString(stream);
    /* We could avoid next line at the cost of more painful debug */
    checkExpected(readType, type);
    while (*type){
        NXReadPortTypeInternal(stream, type, va_arg(args, void *));
        type = type++; /* we restrict to monochar type descriptions */
    }
    va_end (args);
}

static void PortInternalWriteObject(NXPortStream *stream, id object) {
    BOOL        flag = NO;
    if (! [stream->communication respondsTo: @selector (beforeEncoding:onto:freeAfterEncod
ing:)]) BUG("PortInternalWriteObject");
    object = [stream->communication beforeEncoding: object onto: stream freeAfterEncoding:
&flag];
    _NXEncodeBool(stream, (object) ? YES : NO);
    if (object) {
        Class   class = (Class)[object class];
        if (! class) BUG("PortInternalWriteObject: found null class");
        _NXEncodeChars (stream, class->name);
        [object writePortStream: stream];
        _NXEncodeBool(stream, YES);
        if (flag) [object free];
    }
}

static id PortInternalReadObject(NXPortStream *stream) {
    id          object;
    if (! _NXDecodeBool(stream)) return nil;
    else {
        NXAtom  className = _NXDecodeUniqueString (stream);
        Class   class = (Class) objc_getClass ((char *) className);
        if (! class) BUG("PortInternalReadObject: class not loaded");
        /* explicit class initialization */
        (void) [(id) class self];
        object = class_createInstance (class, 0);
        [object readPortStream: stream];
        [object awake];
        if (! [stream->communication respondsTo: @selector (afterDecoding:from:)]) BUG("Po
rtInternalWriteObject-1");
        object = [stream->communication afterDecoding: object from: stream];
        if (! _NXDecodeBool(stream)) BUG("PortInternalReadObject-2");
        return object;
    }
}
```

We claim:

1. A method for sending an object oriented programming language based message having dynamic binding from a first object in a first process to a second object in a second process, said method comprising the steps of:

transmitting, using a first processing means, said object oriented programming language based message to a first proxy in said first process;

using said first proxy and said first processing means, encoding said object oriented programming language based message into an operating system based message at run time;

transmitting said operating system based message to said second process in said second processing means at run time;

decoding, using a second process, said operating system based message into a language based message;

transmitting, using said second processing means, said object oriented programming language based message to said second object in said second process;

executing said object oriented programming language based message by said second object in said second process.

2. The method of claim **1** further including the steps of:

said second object in said second process and generating an object oriented programming language based result;

encoding, using said second processing means, said object oriented programming language based result into an operating system based result at run time;

transmitting, using said second processing means, said operating system based result to said first process at run time;

decoding said operating system based result into an object oriented programming language based result at run time, using said first processing means;

transmitting, using said first processing means, said object oriented programming language based result to said first object.

3. The method of claim **1** wherein said object oriented programming language based message comprises a method and an argument.

4. The method of claim **1** wherein said second object executes said method on said argument when executing said message.

5. The method of claim **1** wherein the step of executing said object oriented programming language based message further includes the steps of:

said second object determining, using said second processing means, whether additional information is needed to execute said object oriented programming language based message;

said second object generating, using said second processing means, an object oriented programming language based query if it is determined that additional information is needed;

encoding, using said second processing means, said object oriented programming language based query into an operating system based query at run time if it is determined that additional information is needed;

transmitting said operating system based query to said first process at run time, using said second processing means if it is determined that additional information is needed;

decoding, using said first processing means, said operating system based query into an object oriented programming language based query at run time if it is determined that additional information is needed;

transmitting, using said first processing means, said object oriented programming language based query to said first object if it is determined that additional information is needed.

6. The method of claim **5** further including the steps of:

said first object generating, using said first processing means, an object oriented programming language based reply to said object oriented programming language based query;

encoding said object oriented programming language based reply into an operating system based reply at run time, using said first processing means;

transmitting, using said first processing means, said operating system based reply to said second process at run time;

decoding, using said second processing means, said operating system based reply into an object oriented programming language based reply at run time;

transmitting, using said second processing means, said object oriented programming language based reply to said second object.

7. The method of claim **6** wherein said first processing means and said second processing means are the same processing means.

8. The method of claim **1** wherein said object oriented programming language based message comprises an objective C message.

9. The method of claim **1** wherein said operating system based message comprises a Mach message.

10. The method of claim **1** wherein said first proxy represents said second object.

11. A method for sending an object oriented programming language based message having dynamic binding from a first object in a first process to a second object in a second process, said method comprising the steps of:

transmitting, using a first processing means, said object oriented programming language based message to a first proxy in said first process;

using said first proxy and said first processing means, encoding said object oriented programming language based message into an operating system based message at run time;

transmitting, using said first processing means, said operating system based message to said second process at run time;

decoding, using said second processing means, said operating system based message into an object oriented programming language based message at run time;

transmitting, using said second processing means, said object oriented programming language based message to said second object;

said second object generating an object oriented programming language based query, using said second processing means;

creating, using said second processing means, a second proxy in said second process;

transmitting, using said second processing means, said object oriented programming language based query to said second proxy;

using said second proxy and said second processing means, encoding said object oriented programming language based query into an operating system based query at run time;

transmitting, using said second processing means, said operating system based query to said first process at run time;

decoding, using said first processing means, said operating system based query into an object oriented programming language based query at run time;

transmitting, using said first processing means, said object oriented programming language based query to said first object;

said first object generating an object oriented programming language based reply, using said first processing means;

encoding, using said first processing means, said object oriented programming language based reply into an operating system based reply at run time;

transmitting, using said first processing means, said operating system based reply to said second process at run time;

decoding, using a second processing means, said operating system based reply into an object oriented programming language based reply at run time;

transmitting, using said second processing means, said object oriented programming language based reply, using said second processing means, and generating an object oriented programming language based result;

encoding, using said second processing means, said object oriented programming language based result into an operating system based result at run time;

transmitting, using said second processing means, said operating system based result to said first process at run time;

decoding, using said first processing means, said operating system based result into an object oriented programming language based result;

transmitting, using said first processing means, said object oriented programming language based result to said first object.

12. The method of claim **11** wherein said object oriented programming language based message comprises a method and an argument.

13. The method of claim **12** wherein said second object executes said method on said argument when executing said message.

14. The method of claim **11** wherein said first process and said second process are located on first and second computers respectively.

15. The method of claim **11** wherein said object oriented programming language based message comprises an objective C message.

16. The method of claim **11** wherein said operating system based message comprises a Mach message.

17. The method of claim **11** wherein said first proxy represents said second object.

18. The method of claim **11** wherein said second proxy represents said first object.

19. A method for sending, in a C environment with minimal run time support, an object oriented programming language based message having dynamic binding from a first object in a first process to a second object in a second process, said method comprising the steps of:

transmitting, using a first processing means implementing said C environment, said object oriented programming language based message to a first proxy in said first process;

using said first proxy and said first processing means, encoding said object oriented programming language based message into an operating system based message at run time;

transmitting said operating system based message to said second process at run time;

decoding, using a second processing means implementing said C environment, said operating system based message into a language based message;

transmitting, using said second processing means, said object oriented programming language based message to said second object.

20. The method of claim **19** further including the steps of:

said second object executing said object oriented programming language based message, using said second processing means, and generating an object oriented programming language based result;

encoding, using said second processing means, said object oriented programming language based result into an operating system based result at run time;

transmitting, using said second processing means, said operating system based result to said first process at run time;

decoding said operating system based result into an object oriented programming language based result at run time, using said first processing means;

transmitting, using said first processing means, said object oriented programming language based result to said first object.

21. The method of claim **20** wherein the step of executing said object oriented programming language based message further includes the steps of:

said second object determining, using said second processing means, whether additional information is needed to execute said object oriented programming language based message;

said second object generating, using said second processing means, an object oriented programming language based query if it is determined that additional information is needed;

encoding, using said second processing means, said object oriented programming language based query into an operating system based query at run time if it is determined that additional information is needed;

transmitting said operating system based query to said first process at run time, using said second processing means if it is determined that additional information is needed;

decoding, using said first processing means, said operating system based query into an object oriented programming language based query at run time if it is determined that additional information is needed;

transmitting, using said first processing means, said object oriented programming language based query to said first object if it is determined that additional information is needed.

22. The method of claim **21** further including the steps of:

said first object generating, using said first processing means, an object oriented programming language based reply to said object oriented programming language based query;

encoding said object oriented programming language based reply into an operating system based reply at run time, using said first processing means;

transmitting, using said first processing means, said operating system based reply to said second process at run time;

decoding, using said second processing means, said operating system based reply into an object oriented programming language based reply at run time;

transmitting, using said processing means, said object oriented programming language based reply to said second object.

23. The method of claim **21** wherein said operating system based message comprises a Mach message.

24. The method of claim **21** wherein said first processing means and said second processing means are the same processing means.

* * * * *