# EXHIBIT 1

US005455599A

# United States Patent [19]

Cabral et al.

[11] Patent Number: 5,455,599

[45] Date of Patent: Oct. 3, 1995

[54] **OBJECT-ORIENTED GRAPHIC SYSTEM**

[75] Inventors: **Arthur W. Cabral**; **Rajiv Jain**, both of Sunnyvale; **Maire L. Howard**, San Jose; **John Peterson**, Menlo Park; **Richard D. Webb**, Sunnyvale; **Robert Seidl**, Palo Alto, all of Calif.

[73] Assignee: **Taligent Inc.**, Cupertino, Calif.

[21] Appl. No.: **416,949**

[22] Filed: **Apr. 4, 1995**

**Related U.S. Application Data**

[63] Continuation of Ser. No. 145,840, Nov. 2, 1993, abandoned.

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

[52] **U.S. Cl.** ..... **345/133**; 395/118

[58] **Field of Search** ..... 345/112, 132, 345/133, 153, 154, 155; 395/118, 275

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,821,220 | 4/1989 | Duisberg | 364/578 |
| 4,885,717 | 12/1989 | Beck et al. | 364/900 |
| 4,891,630 | 1/1990 | Friedman et al. | 340/706 |
| 4,953,080 | 8/1990 | Dysart 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 | 12/1992 | 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 |
| 5,151,987 | 9/1992 | Abraham et al. | 395/575 |
| 5,181,162 | 1/1993 | Smith et al. | 364/419 |
| 5,241,625 | 8/1993 | Epard et al. | 395/163 |
| 5,265,206 | 11/1993 | Shackelford et al. | 395/200 |
| 5,297,279 | 3/1994 | Bannon et al. | 395/600 |

FOREIGN PATENT DOCUMENTS

| | | |
|---|---|---|
| 0459683 | 12/1991 | European Pat. Off. . |
| 0603095 | 6/1994 | European Pat. Off. . |
| 91/20032 | 12/1991 | WIPO . |

OTHER PUBLICATIONS

"Object Oriented Approach to Design of Interactive Intelligent Instrumentation User Interface", Nikola Bogunovic, Automatika vol. 34, No. 3–4, May–Dec. 1993, pp. 143–146.

"Object–oriented versus bit–mapped graphics interfaces: performance and preference differences for typical applications", Michael Mohageg, Behaviour & Inforamtion Technology, vol. 10, No. 2, Mar.–Apr. 1991 pp. 121–147.

"Porting Apple© Macintosh© Applications to the Microsoft© Windows Environment", Schulman et al., Microsoft System Journal, vol. 4, No. 1, Jan. 1989, pp. 11–40.

Computer, vol. 22(10), Dec. 1989, Long Beach, US, pp. 43–54, Goodman "Knowledge–Based Computer Vision".

Software–Practice and Experience, vol. 19(10), Oct. 1989, Chicester UK, pp. 979–1013, Dietrich, "TGMS: An Object–Oriented System for Programming Geometry".

Proceedings of the SPIE, vol. 1659, Feb. 12, 1992, US, pp. 159–167, Haralick et al. "The Image Understanding Environment".

Intelligent CAD Oct. 6, 1987, NL, pp. 159–168, Woodbury et al., "An Approach to Geometric Reasoning".

Computer, vol. 22(10), Dec. 1989.

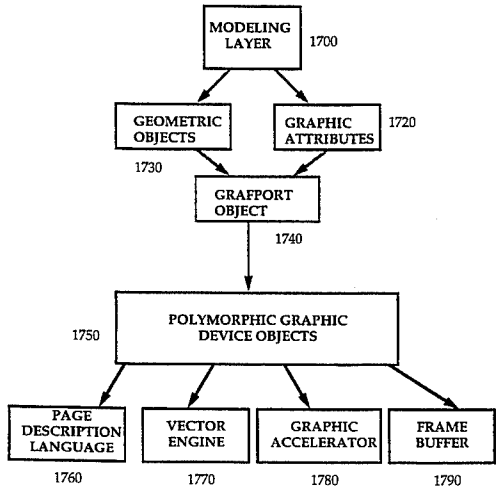
*Primary Examiner*—Jeffery Brier

*Attorney, Agent, or Firm*—Keith Stephens

[57] **ABSTRACT**

An object-oriented graphic system is disclosed including a processor with an attached display, storage and object-oriented operating system. The graphic system builds a component object in the storage of the processor for managing graphic processing. The processor includes an object for connecting one or more graphic devices to various objects responsible for tasks such as graphic accelerators, frame buffers, page description languages and vector engines. The system is fully extensible and includes polymorphic processing built into each of the support objects.

**26 Claims, 16 Drawing Sheets**



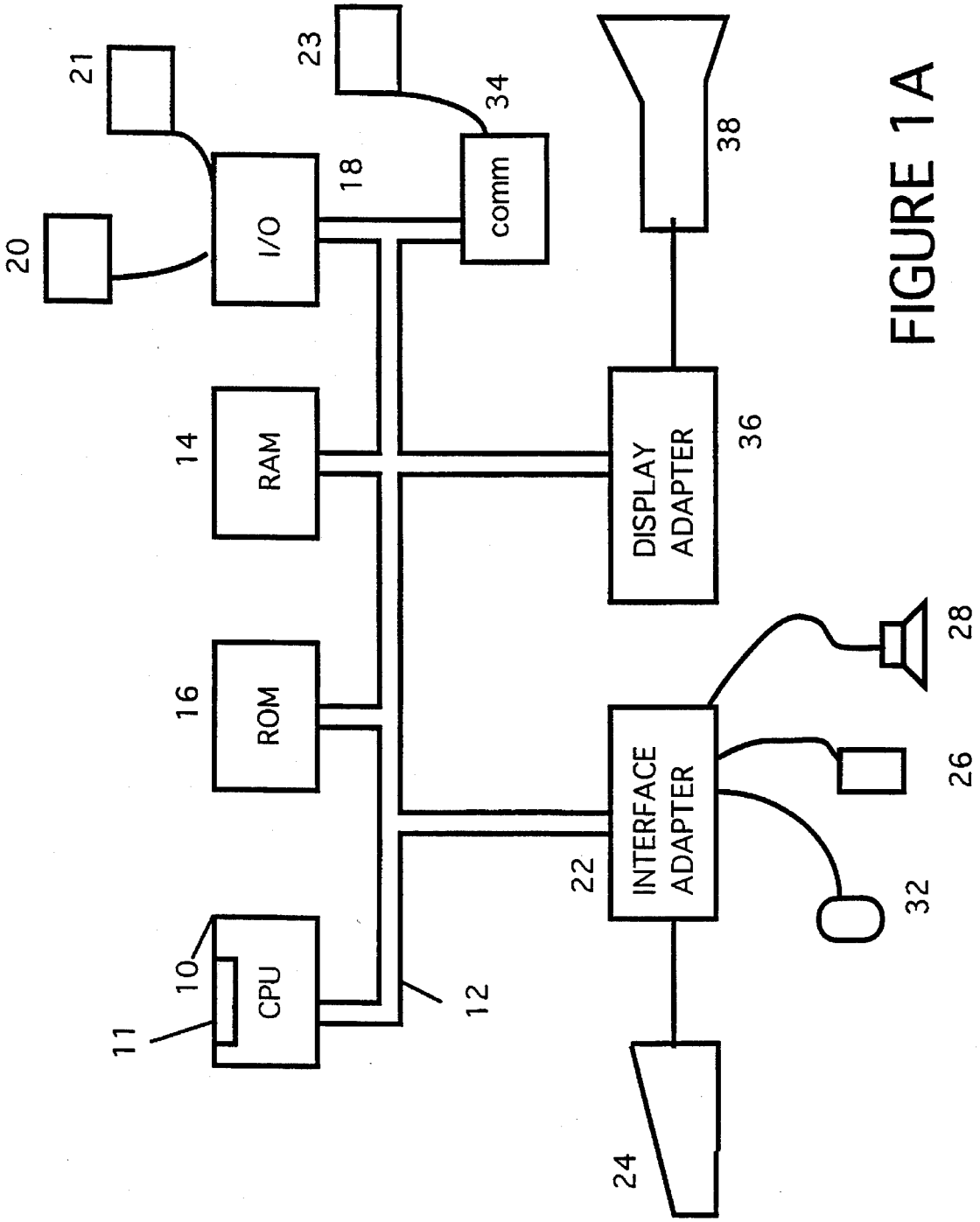

CPU
10
11
12
ROM
16
RAM
14
I/O
18
20
21
comm
34
23
INTERFACE ADAPTER
22
DISPLAY ADAPTER
36
38
24
26
28
32

FIGURE 1A

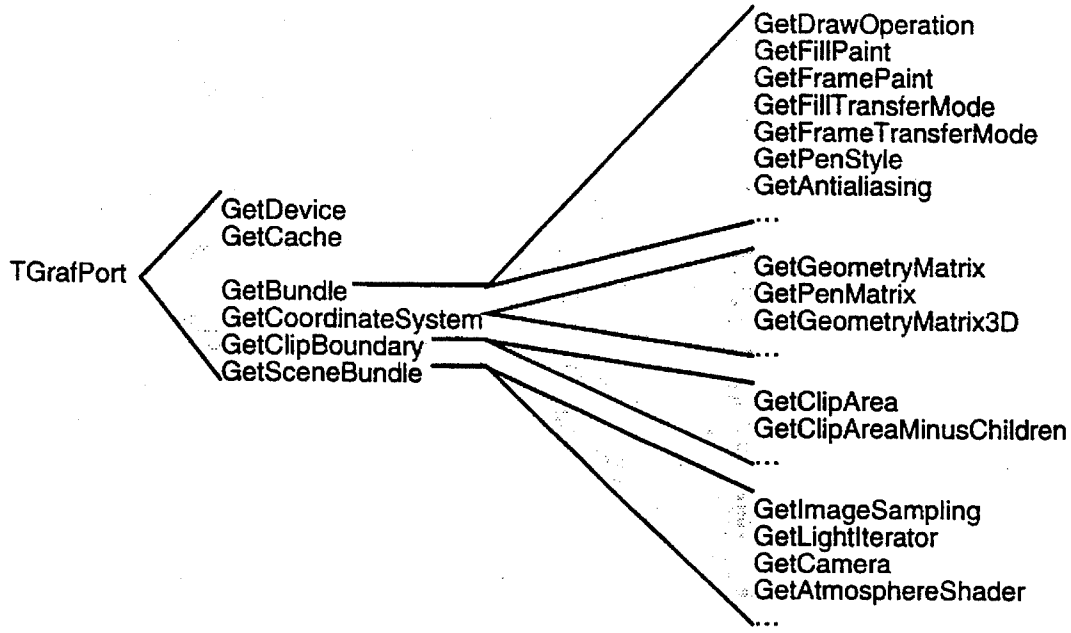

TGrafPort
GetDevice
GetCache
GetBundle
GetCoordinateSystem
GetClipBoundary
GetSceneBundle
GetDrawOperation
GetFillPaint
GetFramePaint
GetFillTransferMode
GetFrameTransferMode
GetPenStyle
GetAntialiasing
...
GetGeometryMatrix
GetPenMatrix
GetGeometryMatrix3D
...
GetClipArea
GetClipAreaMinusChildren
...
GetImageSampling
GetLightIterator
GetCamera
GetAtmosphereShader
...

FIGURE 1B

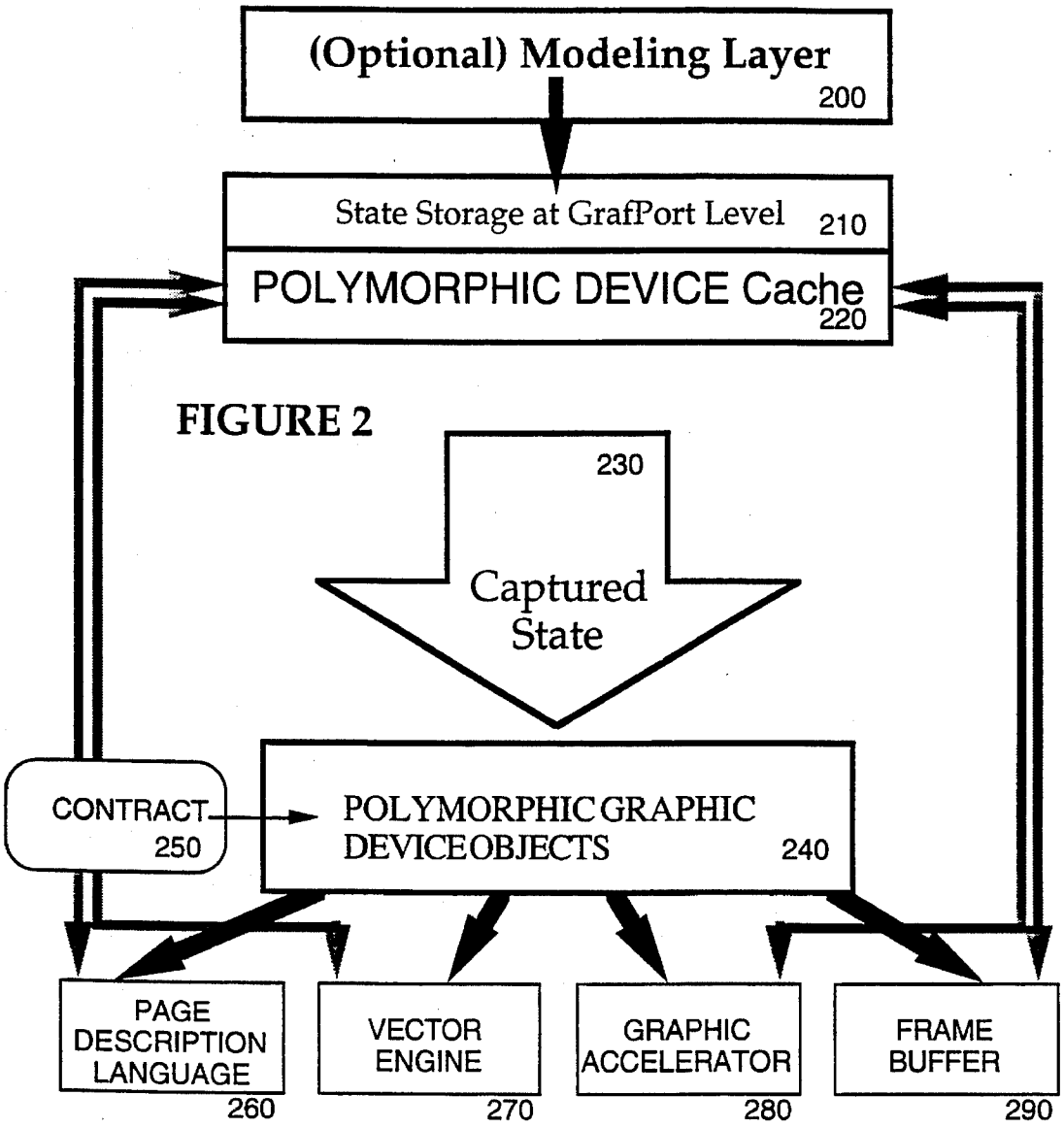

(Optional) Modeling Layer
200
State Storage at GrafPort Level
210
POLYMORPHIC DEVICE Cache
220
230
Captured State
CONTRACT
250
POLYMORPHIC GRAPHIC DEVICE OBJECTS
240
PAGE DESCRIPTION LANGUAGE
260
VECTOR ENGINE
270
GRAPHIC ACCELERATOR
280
FRAME BUFFER
290

FIGURE 2


FIGURE 3

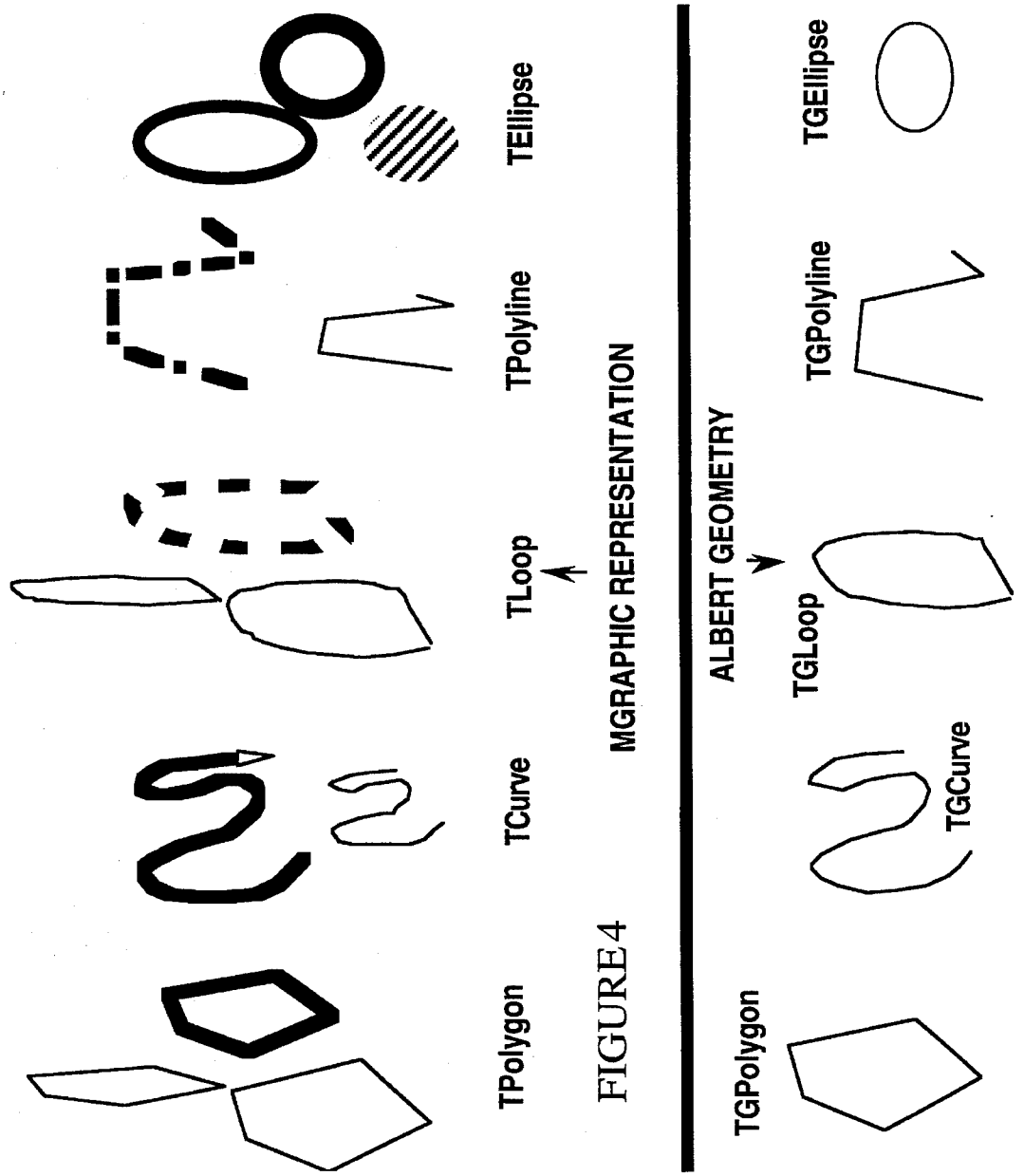

TPolygon
TCurve
TLoop
TPolyline
TEllipse
MGRAPHIC REPRESENTATION
ALBERT GEOMETRY
TGPolygon
TGCurve
TGLoop
TGPolyline
TGEllipse

FIGURE4

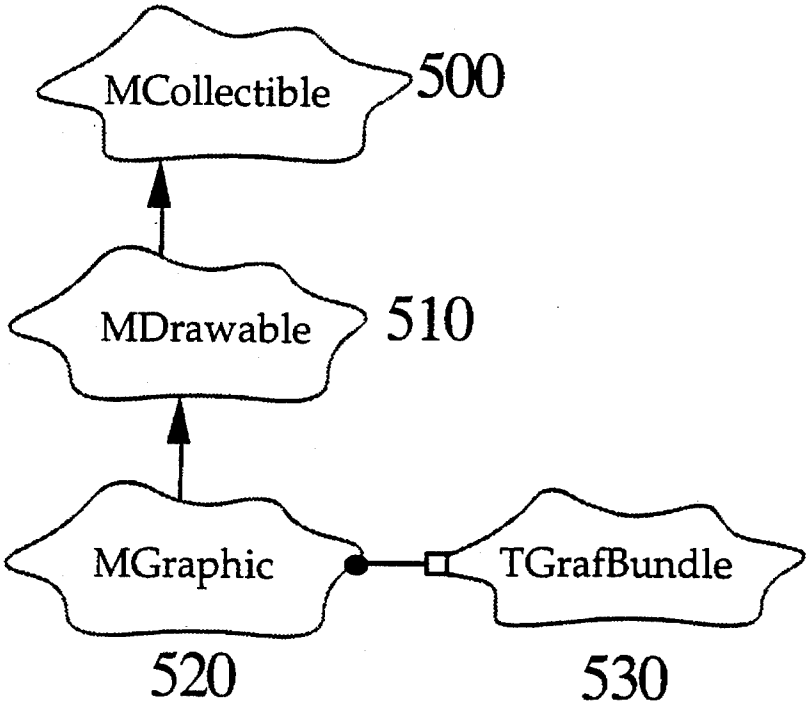



FIGURE 5

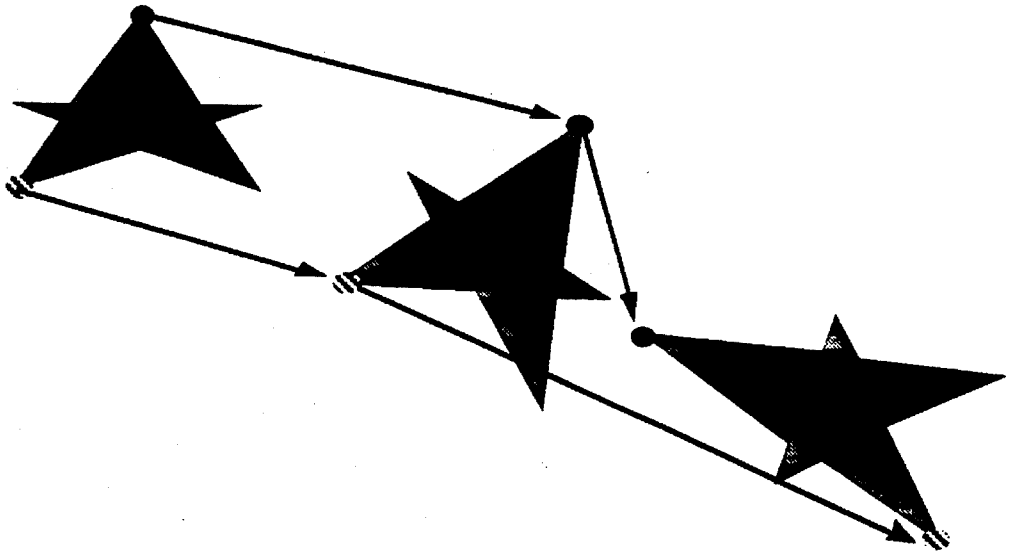


FIGURE 6

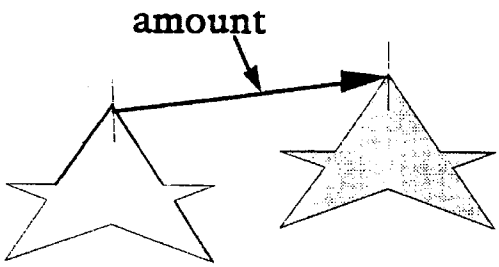
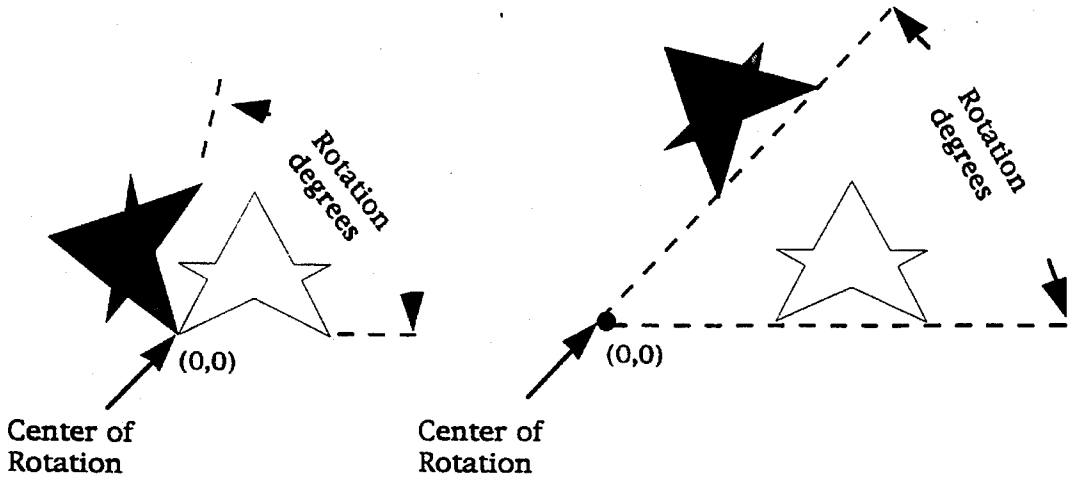



FIGURE 7



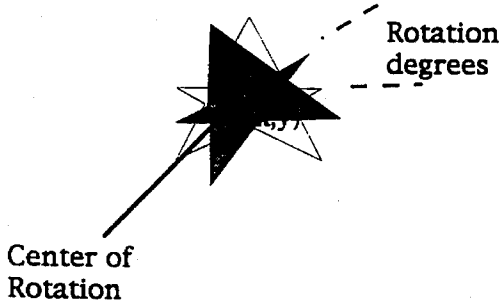




FIGURE 8

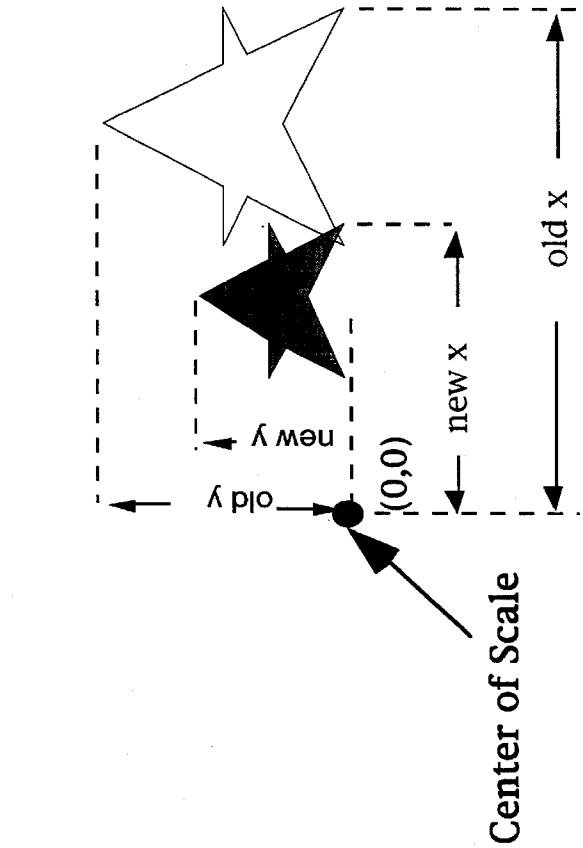
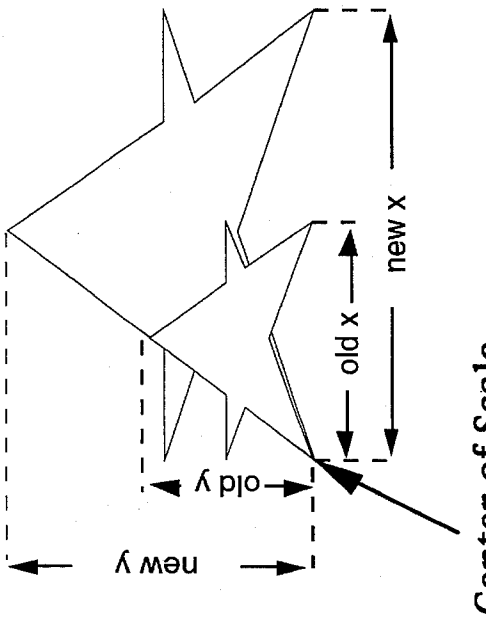



FIGURE 9A



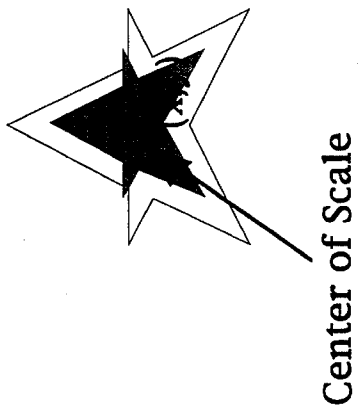




FIGURE 9B

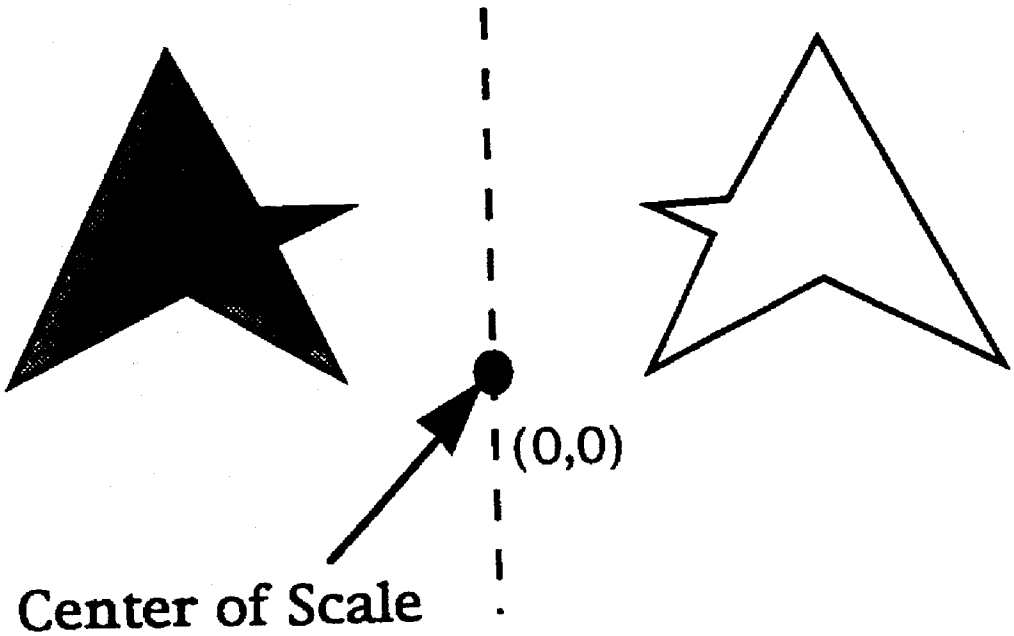



FIGURE 10

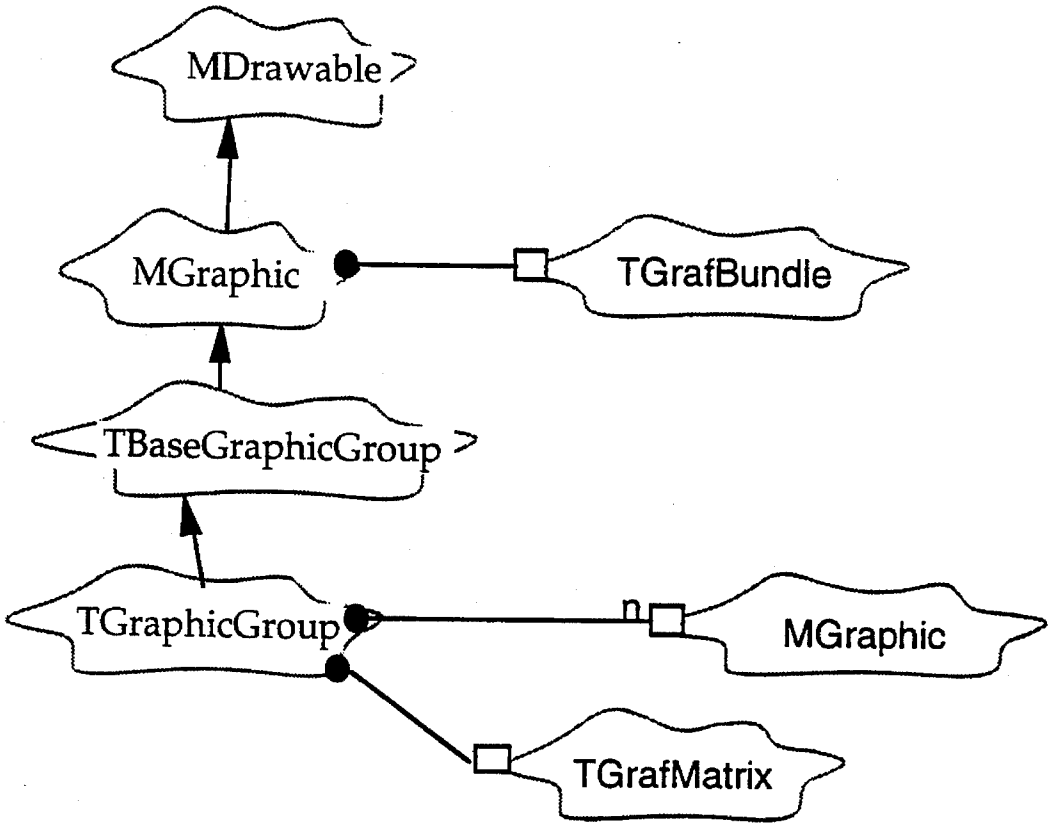

MDrawable
MGraphic
TGrafBundle
TBaseGraphicGroup
TGraphicGroup
n
MGraphic
TGrafMatrix

FIGURE 11

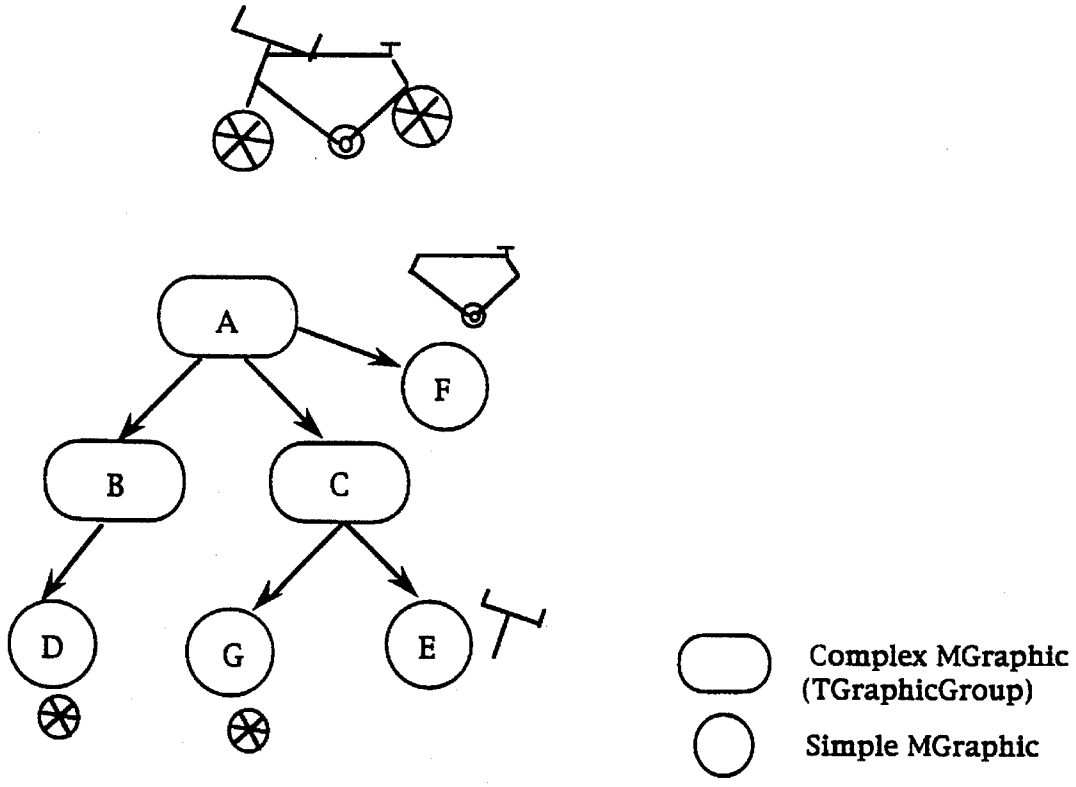



FIGURE 12

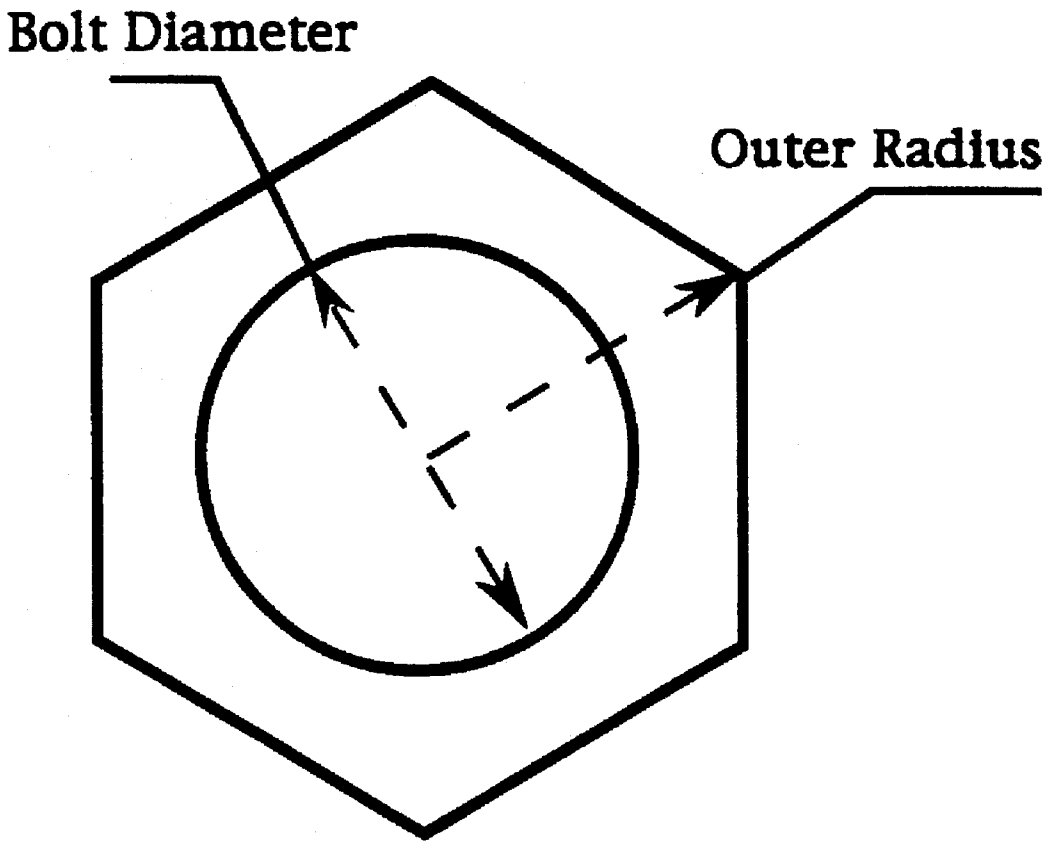



FIGURE 13

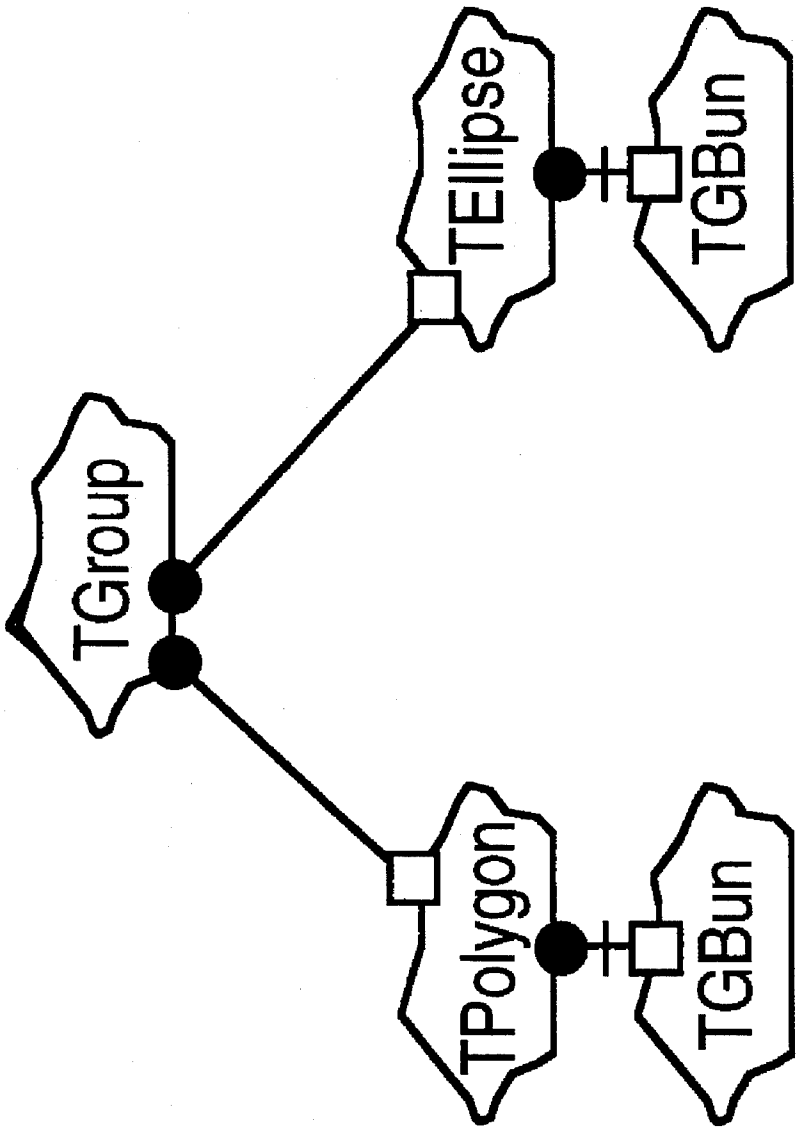



FIGURE 14

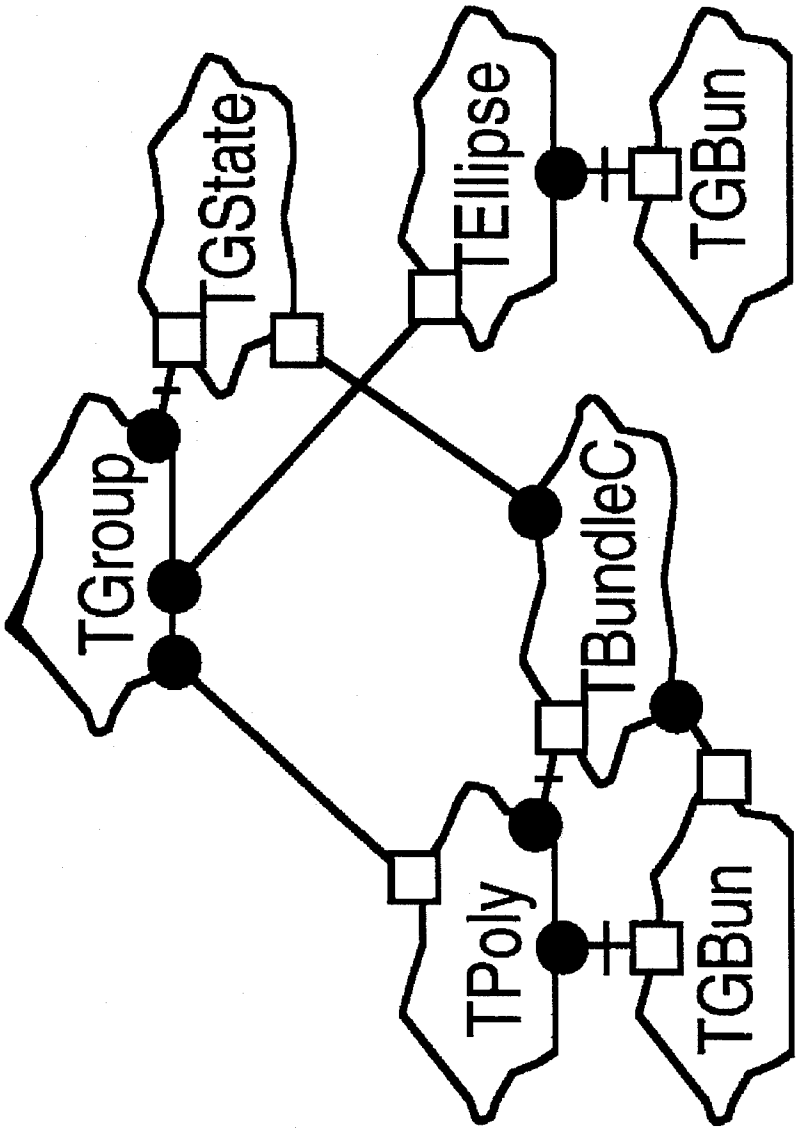



FIGURE 15

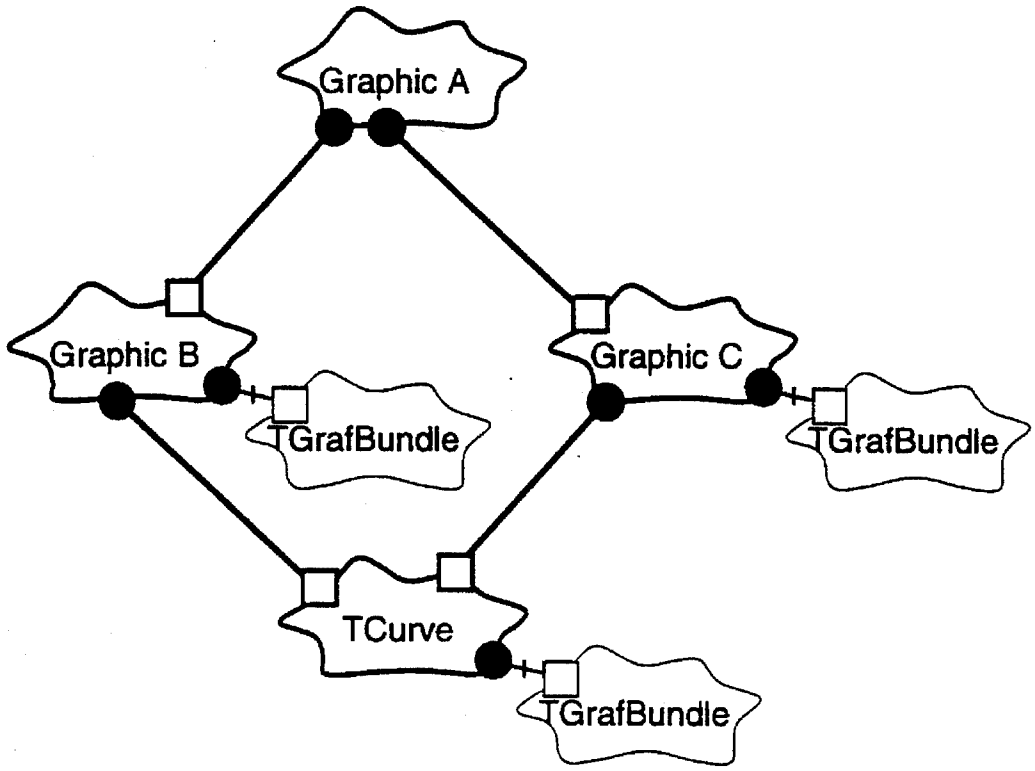

Graphic A
Graphic B
Graphic C
TGrafBundle
TGrafBundle
TCurve
TGrafBundle

FIGURE 16

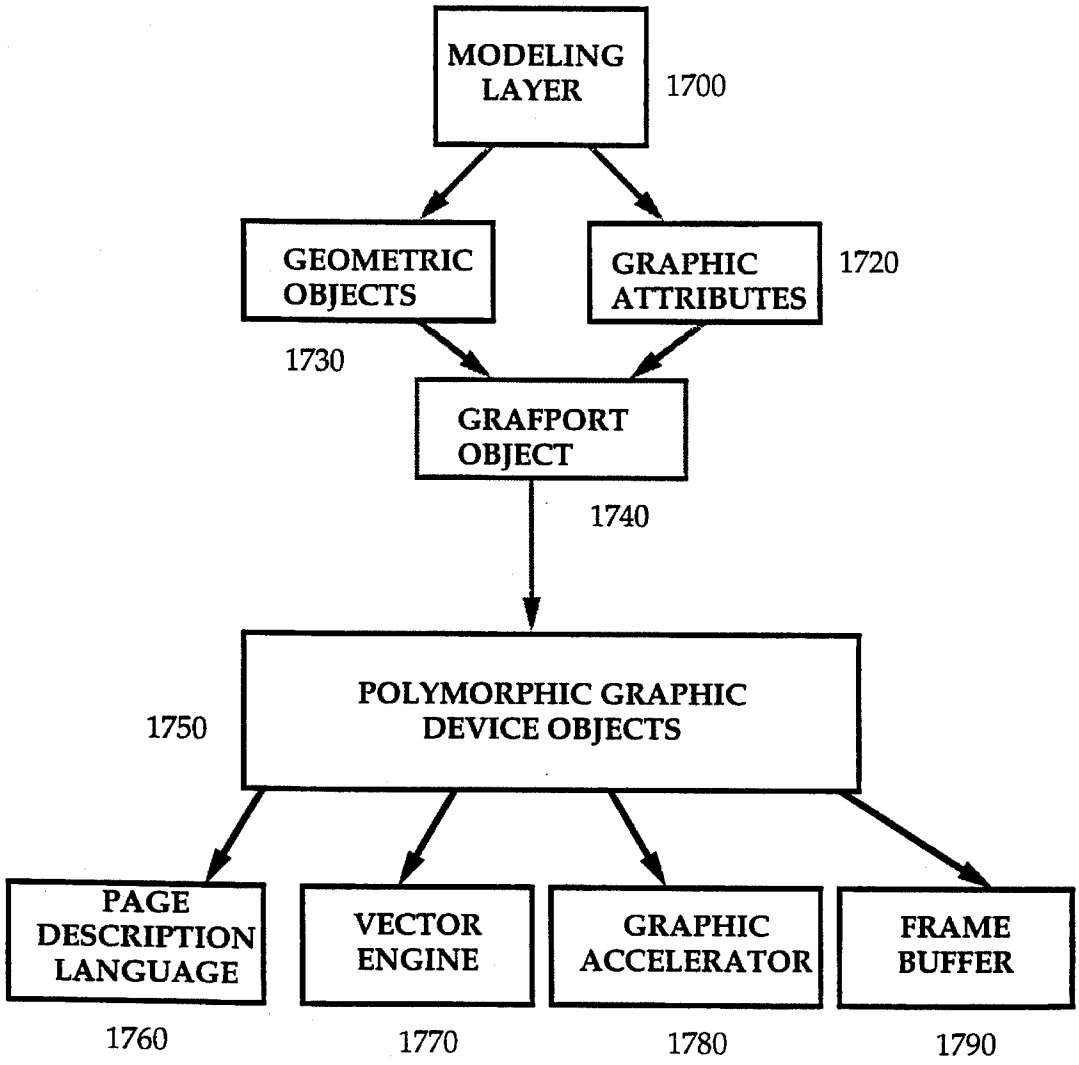

MODELING LAYER
1700
GEOMETRIC OBJECTS
GRAPHIC ATTRIBUTES
1720
1730
GRAFPORT OBJECT
1740
POLYMORPHIC GRAPHIC DEVICE OBJECTS
1750
PAGE DESCRIPTION LANGUAGE
VECTOR ENGINE
GRAPHIC ACCELERATOR
FRAME BUFFER
1760
1770
1780
1790

FIGURE 17

# OBJECT-ORIENTED GRAPHIC SYSTEM

This is a continuation, of application Ser. No. 08/145,840, filed Nov. 2, 1993, abandoned.

## COPYRIGHT NOTIFICATION



1. Field of the Invention

This invention generally relates to improvements in computer systems and more particularly to a system for enabling graphic applications using an object-oriented operating system.

2. Background of the Invention

Computer pictures or images drawn on a computer screen are called computer graphics. Computer graphic systems store graphics internally in digital form. The picture is broken up into tiny picture elements or pixels. Thus, a computer picture or graphic is actually an aggregation of individual picture elements or pixels. Internally, in the digital world of the computer, each pixel is assigned a set of digital values which represent the pixel's attributes. A pixel's attributes may describe its color, intensity and location, for example. Thus to change the color, intensity or location of a pixel, one simply changes the digital value for that particular attribute.

Conventional computer graphic systems utilize primitives known as images, bitmaps or pixel maps to represent computer imagery as an aggregation of pixels. These primitives represent a Two Dimensional (2D) array of pixel attributes and their respective digital values. Typically, such a primitive is expressed as a "struct" (data structure) that contains a pointer to pixel data, a pixel size, scanline size, bounds, and possibly a reference to a color table. Quite often, the pixels are assumed to represent Red, Green, and Blue (RGB) color, luminance, or indices into a color table. Thus, the primitive serves double duty as a framebuffer and as a frame storage specification.

The burgeoning computer graphics industry has settled on a defacto standard for pixel representation. All forms of images that do not fit into this standard are forced into second class citizenship. Conventional graphics systems, however, are nonextendable. They are usually dedicated to a particular application operating on a specific class of images. This is unacceptable in today's rapidly changing environment of digital technology. Every day a new application, and with it the need to process and manipulate new image types in new ways. Thus, the use of a graphics system with a nonextensible graphic specification is not only short sighted, it is in a word, obsolete. Graphical applications, attributes, and organizational requirements for computer output media are diverse and expanding. Thus, dedicated, single-purpose graphic systems fail to meet current application requirements. There is a need for a robust, graphic system that provides a dynamic environment and an extensible graphic specification that can expand to include new applications, new image types and provide for new pixel manipulations.

For example, two applications rarely require the same set of pixel attributes. Three Dimensional (3D) applications store z values (depth ordering), while animation and paint systems store alpha values. Interactive material editors and 3D paint programs store 3D shading information, while video production systems may require YUV 4:2:2 pixel arrays. Hardware clippers store layer tags, and sophisticated systems may store object IDs for hit detection. Moreover, graphical attributes such as color spaces are amassing constant additions, such as PhotoYCC™. Color matching technology is still evolving and it is yet unclear which quantized color space is best for recording the visible spectrum as pixels. Thus, there are a variety of data types in the graphics world. There are also a variety of storage organization techniques. To make matters even worse, it seems that every new application requires a different organization for the pixel memory. For example, Component Interleaved or "Chunky" scanline orientations are the prevailing organization in Macintosh ® video cards, but Component Interleaved banked switched memory is the trend in video cards targeted for hosts with small address spaces. Component planar tiles and component interleaved tiles are the trend in prepress and electronic paint applications, but output and input devices which print or scan in multiple passes prefer a component planar format. Multiresolution or pyramid formats are common for static images that require real-time resampling. Moreover, images that consume large amounts of memory may be represented as compressed pixel data which can be encoded in a multitude of ways.



The variety and growth of graphic applications, data types and pixel memory manipulations is very large. There is a requirement for a multipurpose system that can handle all the known applications and expand to handle those applications that are yet unknown. A single solution is impractical. Although it may handle every known requirement, it would be huge and unwieldy. However, if such an application is downsized, it can no longer handle every application. Thus, there is a need for a general graphic framework that suits the needs of many users, but allows the individual user to customize the general purpose graphic framework.

## SUMMARY OF THE INVENTION

An object-oriented system is well suited to address the shortcomings of traditional graphic applications. Object-oriented designs can provide a general purpose framework that suits the needs of many users, but allows the individual user to customize and add to the general purpose framework to address a particular set of requirements. In general, an object may be characterized by a number of operations and a state which remembers the effect of these operations.

Thus it is a goal of the present invention to provide a method and apparatus which facilitates an object-oriented graphic system. A processor with an attached display, storage and object-oriented operating system builds a component object in the storage of the processor for managing graphic processing. The processor includes an object for connecting one or more graphic devices to various objects responsible for tasks such as graphic accelerators, frame buffers, page description languages and vector engines. The system is fully extensible and includes polymorphic processing built into each of the support objects.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1A is a block diagram of a personal computer system in accordance with a preferred embodiment;

FIG. 1B is a hierarchical layout of a graphic port in accordance with a preferred embodiment;

FIG. 2 is a block diagram of the architecture in accordance with a preferred embodiment;

FIG. **3** illustrates examples of graphic extensions of MGraphic in accordance with a preferred embodiment;

FIG. **4** illustrates MGraphics and their corresponding geometries in accordance with a preferred embodiment;

FIG. **5** is a booch diagram setting forth the flow of control of the graphic system in accordance with a preferred embodiment;

FIG. **6** illustrates a star graphic object undergoing various transformations in accordance with a preferred embodiment;

FIG. **7** depicts a star moved by an amount in accordance with a preferred embodiment;

FIG. **8** illustrates rotating-the star about various centers of rotation in accordance with a preferred embodiment;

FIG. **9** illustrates scaling a star about different centers of scale in accordance with a preferred embodiment;

FIG. **10** shows the effects of scaling an asymmetric star by (−1.0, 1.0) in accordance with a preferred embodiment;

FIG. **11** illustrates a hierarchical graphic in accordance with a preferred embodiment;

FIG. **12** illustrates a bike graphic in accordance with a preferred embodiment;

FIG. **13** illustrates a bolt object in accordance with a preferred embodiment;

FIG. **14** illustrates a hierarchical graphic in accordance with a preferred embodiment;

FIG. **15** illustrates an object that exists inside the TPolygon's Draw call in accordance with a preferred embodiment;

FIG. **16** illustrates a graphic hierarchy that supports sharing of two or more graphics in accordance with a preferred embodiment; and

FIG. **17** is a flowchart setting forth the detailed logic in accordance with a preferred embodiment.

## 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**, 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** 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 Apple System/7® operating system.

In a preferred embodiment, the invention is implemented in the C++ programming language using object oriented programming techniques. As will be understood by those skilled in the art, Object-Oriented Programming (OOP) objects are software entities comprising data structures and operations on the data. Together, these elements enable objects to model virtually any real-world entity in terms of its characteristics, represented by its data elements, and its behavior, represented by its data manipulation functions. In this way, objects can model concrete things like people and computers, and they can model abstract concepts like numbers or geometrical concepts. The benefits of object technology arise out of three basic principles: encapsulation, polymorphism and inheritance.

Objects hide, or encapsulate, the internal structure of their data and the algorithms by which their functions work. Instead of exposing these implementation details, objects present interfaces that represent their abstractions cleanly with no extraneous information. Polymorphism takes encapsulation a step further. The idea is many shapes, one interface. A software component can make a request of another component without knowing exactly what that component is. The component that receives the request interprets it and determines, according to its variables and data, how to execute the request. The third principle is inheritance, which allows developers to reuse pre-existing design and code. This capability allows developers to avoid creating software from scratch. Rather, through inheritance, developers derive subclasses that inherit behaviors, which the developer then customizes to meet their particular needs.

A prior art approach is to layer objects and class libraries in a procedural environment. Many application frameworks on the market take this design approach. In this design, there are one or more object layers on top of a monolithic operating system. While this approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over procedural programming techniques, there are limitations to this approach. These difficulties arise from the fact that while it is easy for a developer to reuse their own objects, it is difficult to use objects from other systems and the developer still needs to reach into the lower, non-object layers with procedural Operating System (OS) calls.

Another aspect of object oriented programming is a framework approach to application development. One of the most rational definitions of frameworks came from Ralph E. Johnson of the University of Illinois and Vincent F. Russo of Purdue. In their 1991 paper, Reusing Object-Oriented Designs, University of Illinois tech report UIUCDCS91-1696 they offer the following definition: "An abstract class is a design of a set of objects that collaborate to carry out a set of responsibilities. Thus, a framework is a set of object classes that collaborate to execute defined sets of computing responsibilities." From a programming standpoint, frameworks are essentially groups of interconnected object classes that provide a pre-fabricated structure of a working application. For example, a user interface framework might provide the support and "default" behavior of drawing windows, scrollbars, menus, etc. Since frameworks are based on object technology, this behavior can be inherited and overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This is a major advantage over traditional programming since the programmer is not changing the original code, but rather extending the software. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling but at the same time frees them to then supply the specific actions unique to the problem domain.

From a business perspective, frameworks can be viewed as a way to encapsulate or embody expertise in a particular knowledge area. Corporate development organizations, Independent Software Vendors (ISV)s and systems integrators have acquired expertise in particular areas, such as manufacturing, accounting, or currency transactions. This

expertise is embodied in their code. Frameworks allow organizations to capture and package the common characteristics of that expertise by embodying it in the organization's code. First, this allows developers to create or extend an application that utilizes the expertise, thus the problem gets solved once and the business rules and design are enforced and used consistently. Also, frameworks and the embodied expertise behind the frameworks, have a strategic asset implication for those organizations who have acquired expertise in vertical markets such as manufacturing, accounting, or bio-technology, and provide a distribution mechanism for packaging, reselling, and deploying their expertise, and furthering the progress and dissemination of technology.

Historically, frameworks have only recently emerged as a mainstream concept on personal computing platforms. This migration has been assisted by the availability of object-oriented languages, such as C++. Traditionally, C++ was found mostly on UNIX systems and researcher's workstations, rather than on computers in commercial settings. It is languages such as C++ and other object-oriented languages, such as Smalltalk and others, that enabled a number of university and research projects to produce the precursors to today's commercial frameworks and class libraries. Some examples of these are InterViews from Stanford University, the Andrew toolkit from Carnegie-Mellon University and University of Zurich's ET++ framework. Types of frameworks range from application frameworks that assist in developing the 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 frameworks are MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXTStep App Kit (NeXT), and Smalltalk-80 MVC (ParcPlace).

Programming with frameworks requires a new way of thinking for developers accustomed to other kinds of systems. In fact, it is not like "programming" at all in the traditional sense. In old-style operating systems such as DOS or UNIX, the developer's own program provides all of the structure. The operating system provides services through system calls-the developer's program makes the calls when it needs the service and control returns when the service has been provided. The program structure is based on the flow-of-control, which is embodied in the code the developer writes. When frameworks are used, this is reversed. The developer is no longer responsible for the flow-of-control. The developer must forego the tendency to understand programming tasks in term of flow of execution. Rather, the thinking must be in terms of the responsibilities of the objects, which must rely on the framework to determine when the tasks should execute. Routines written by the developer are activated by code the developer did not write and that the developer never even sees. This flip-flop in control flow can be a significant psychological barrier for developers experienced only in procedural programming. Once this is understood, however, framework programming requires much less work than other types of programming.

In the same way that an application framework provides the developer with prefab functionality, system frameworks, such as those included in a preferred embodiment, leverage the same concept by providing system level services, which developers, such as system programmers, use to subclass/override to create customized solutions. For example, consider a multimedia framework which could provide the foundation for supporting new and diverse devices such as audio, video, MIDI, animation, etc. The developer that needed to support a new kind of device would have to write a device driver. To do this with a framework, the developer only needs to supply the characteristics and behaviors that are specific to that new device.

The developer in this case supplies an implementation for certain member functions that will be called by the multimedia framework. An immediate benefit to the developer is that the generic code needed for each category of device is already provided by the multimedia framework. This means less code for the device driver developer to write, test, and debug. Another example of using system frameworks would be to have separate I/O frameworks for SCSI devices, NuBus cards, and graphics devices. Because there is inherited functionality, each framework provides support for common functionality found in its device category. Other developers could then depend on these consistent interfaces for implementing other kinds of devices.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system. For the commercial or corporate developer, systems integrator, or OEM, this means all the advantages that have been illustrated for a framework such as MacApp can be leveraged not only at the application level for such things as text and user interfaces, but also at the system level, for services such as graphics, multimedia, file systems, I/O, testing, etc. Application creation in the architecture of a preferred embodiment will essentially be like writing domain-specific pieces that adhere to the framework protocol. In this manner, the whole concept of programming changes. Instead of writing line after line of code that calls multiple API hierarchies, software will be developed by deriving classes from the preexisting frameworks within this environment, and then adding new behavior and/or overriding inherited behavior as desired. Thus, the developer's application becomes the collection of code that is written and shared with all the other framework applications. This is a powerful concept because developers will be able to build on each other's work. This also provides the developer the flexibility to customize as much or as little as needed. Some frameworks will be used just as they are. In some cases, the amount of customization will be minimal, so the piece the developer plugs in will be small. In other cases, the developer may make very extensive modifications and create something completely new.

In a preferred embodiment, as shown in FIG. **1**, a multimedia data routing system manages the movement of multimedia information through the computer system, while multiple media components resident in the RAM **14**, and under the control of the CPU **10**, or externally attached via the bus **12** or communication adapter **34**, are responsible for presenting multimedia information. No central player is necessary to coordinate or manage the overall processing of the system. This architecture provides flexibility and provides for increased extensibility as new media types are added. A preferred embodiment provides an object-oriented graphic system. The object-oriented operating system comprises a number of objects that are clearly delimited parts or functions of the system. Each object contains information about itself and a set of operations that it can perform on its information or information passed to it. For example, an object could be named WOMAN. The information contained in the object WOMAN, or its attributes, might be age, address, and occupation. These attributes describe the object WOMAN. The object also contains a set of operations that it can perform on the information it contains. Thus, WOMAN might be able to perform an operation to change occupations from a doctor to a lawyer.

Objects interact by sending messages to each other. These messages stimulate the receiving object to take some action,

that is, perform one or more operations. In the present invention there are many communicating objects. Some of the objects have common characteristics and are grouped together into a class. A class is a template that enables the creation of new objects that contain the same information and operations as other members of the same class. An object created from a certain class is called an instance of that class. The class defines the operations and information initially contained in an instance, while the current state of the instance is defined by the operations performed on the instance. Thus, while all instances of a given class are created equal, subsequent operations can make each instance a unique object.

Polymorphism refers to object-oriented processing in which a sender of a stimulus or message is not required to know the receiving instance's class. The sender need only know that the receiver can perform a certain operation, without regard to which object performs the operation or what class to which it belongs. Instances inherit the attributes of their class. Thus, by modifying the attribute of a parent class, the attributes of the various instances are modified as well, and the changes are inherited by the subclasses. New classes can be created by describing modifications to existing classes. The new class inherits the attributes of its class and the user can add anything which is unique to the new class. Thus, one can define a class by simply stating how the new class or object differs from its parent class or object. Classes that fall below another class in the inheritance hierarchy are called descendants or children of the parent class from which they descend and inherit. In this polymorphic environment, the receiving object is responsible for determining which operation to perform upon receiving a stimulus message. An operation is a function or transformation that may be applied to or by objects in a class. The stimulating object needs to know very little about the receiving object which simplifies execution of operations. Each object need only know how to perform its own operations, and the appropriate call for performing those operations a particular object cannot perform.

When the same operation may apply to many different classes, it is a polymorphic operation. The same operation takes on a different form in a variety of different classes. A method is the implementation of a particular operation for a given class. For example, the class Document may contain an operation called Read. Depending on the data type of the document, for example, ASCII versus BINARY, a different method might be used to perform the Read operation. Thus while both methods logically perform the same task, Read, and are thus called by the same name, Read, they may in fact be different methods implemented by a different piece of executable code. While the operation Read may have methods in several classes, it maintains the same number and types of arguments, that is, its signature remains the same. Subclasses allow a user to tailor the general purpose framework. It allows for different quantization tradeoffs, sets of pixel attributes, and different pixel memory organizations. Each subclass can encapsulate the knowledge of how to allocate, manage, stream, translate, and modify its own class of pixel data. All subsystems of a preferred embodiment use polymorphic access mechanisms, which enable a user to extend buffer types that can be rendered to or copied.

Fortunately, some commonalty exists among the various types of buffers. As it turns out, there are eight basic functions or categories that are necessary to satisfy the majority of client needs. Most clients want polymorphic management and the ability to specify the relationship between discrete and continuous space. Clients want to characterize color capabilities for use in accurate color reproduction. Clients want mechanisms for pixel memory alteration in the form of Get and SetPixel, specialized "blit loops" tailored for scan converting clients, BitBit, and CopyImage. Clients want mechanisms to supply clients with variants which match a key formed from the combination of client supplied attributes. Clients desire the ability to perform polymorphic queries regarding traits or stored attributes. Clients require mechanisms allowing clients to polymorphically create, maintain, and query buffer caches. And finally, clients require mechanisms which allow them to polymorphically create, and maintain correlated backbuffers.

## Graphic Application Programming Interface (API)

The basic components of a graphic system include a fixed set of Geometric Primitives: Point, Rectangle, Line, Curve, Polygon, Polyline, Area in 2D, Line, Polyline, Curve and Surface in 3D. This set of geometry is not intend to be user extensible. This limits the complexity of the lower level graphic devices, and provides a "contract" between the user-level API and the low level device for consistent data. Discretized data sets: which include 2D raster images with a number of possible components and triangulated 3D datasets. High level modeling tools: that can express hierarchical groups of graphic objects. Transforms: these objects represent the operations available with a traditional 3×3 (in 2D) or 4×4 (in 3D) matrices to rotate, scale, translate, etc. objects. Bundles: these objects encapsulate the appearance of the geometry. Standard attributes include (2D & 3D) frame and/or fill color, pen thickness, dash patterns, etc. In 3D, bundles also define shading attributes. Custom attributes may be specified via a keyword/value pair. All numeric values are expressed in IEEE standard double precision floating point in the graphic system. Graphic Ports: a graphic port is an application-level view that encapsulates the state of the application. The graphic port re-routes any draw calls to an appropriate one of a number of possible devices (monitors, off screen frame buffers, PostScriptPrinter on a network, a window, etc.). Graphical "state" (current transform, bundle, clipping region, etc) is managed at the port level. However, at the device level the system is "stateless". In other words, the complete state for a particular rendering operation is presented to the device when that rendering occurs. Note that a device may turn around and invoke other devices. For example, a device for the entire desktop may first decide which screen the geometry falls on, and then invoke the render call for that particular screen.

## Architectural Introduction

In past graphics architectures, a graphic typically stores its state (such as color, transfer mode, clip area, etc.) privately. When asked to draw, the graphic procedurally copies these state variables into a graphic port, where they are accessed by the rendering code. Thus, the graphic's state is available only during this explicit drawing operation. This is not object-oriented, and is a restriction a modern graphic system cannot afford to make. A preferred embodiment provides a framework for a graphic to store its state. The framework supports a "don't call us, we'll call you" architecture in which clients can get access to the graphic state outside the context of any particular function. This is the purpose of the graphic port class. It is an abstract class that defines the interface for accessing the state variables. Concrete subclasses define the actual storage and concatenation behavior of the state variables.

## Graphic Port Class

A design employing graphic port classes groups the graphic states into four different groups, which then are grouped into a single class called graphic port. The four "sub-states" are TGrafBundle, TCoordinateSystem, TClipBoundary, and TSceneBundle. A graphic port object can be referenced by other classes that need access to the full graphic state. Additionally, a child's graphic state can be concatenated to its parent's graphic port object, producing a new graphic port object. FIG. 1B is a hierarchical layout of a graphic port in accordance with a preferred embodiment. A graphic port class also contains methods to access a device and a device cache. GetDevice returns a pointer to the device to which rendering is done. Typically, this device is inherited from the parent graphic port. GetCache returns a pointer to the cache used by the device to cache devicedependent objects. This cache must have been created by the device at an earlier time. The main purpose for subclassing graphic port and the four sub-states is to define how storage and concatenation of the graphic state, device, and device cache is done. A simpler, flat group of state variables would not be flexible enough to support customization of state concatenation for a subset of the state variables. Also, the sub-states assist in splitting the state variables into commonly used groups. For instance, a simple graphic typically needs only a TGrafBundle; more complex graphic objects may need a matrix and possibly a clip area.

A graphic class, such as MGraphic, must describe itself to a TGrafPortDevice in terms of the basic set of geometries, and each geometry must have a graphic port object associated with the geometry. The graphic port allows a graphic object to conveniently "dump" its contents into a TGrafDevice object. This is accomplished by supplying a set of draw functions in the graphic port class that mirrors a set of render functions in the TGrafDevice class. Each draw function takes a geometry and passes the geometry and the contained graphic state to the appropriate render call in the device. For convenience, an overriding bundle and model matrix are also passed.

FIG. 2 is a block diagram of the architecture in accordance with a preferred embodiment. In the preferred embodiment, a modeling layer **200** generates calls to a Graphic port **210** using the API **210** described above. This GraphPort interface accepts only a specific, fixed set of primitives forming a "contract" **250** between the user level API and the device level API **240**. The graphic port captures state information including transform, appearance ("bundle"), and clipping into a polymorphic cache **220** that is used across multiple types of devices. For each render call, the geometry and all relevant accumulated state information **230** is presented to the device via a polymorphic graphic device object **240**. A device managed by the graphic device object **240** may take the form of a page description language **260** (such as postscript), a vector plotting device **270**, a device with custom electronic hardware for rendering geometric primitives **280**, a traditional framebuffer **290**, or any other graphic device such as a display, printer or plotter.

## Modeling Layer

Above the graphic port and geometry layers there is an optional modeling layer. A preferred embodiment provides a modeling layer, but an application can override the default. The default modeling layer is called a "MGraphic" layer. An MGraphic object encapsulates both geometry and appearance (a bundle). To render an MGraphic, a draw method is used. This method takes the graphic port the MGraphic is drawn into as an argument. The MGraphic draw method turns this information into a graphic port call. The goal behind separating the MGraphic layer from-the graphic port / geometry layer is to avoid a rigid structure suited to only one type of database. If the structure provided by the MGraphic objects does not satisfy the client's requirements, the architecture still permits a different data structure to be used, as long as it can be expressed in terms of primitive geometries, bundles, and transforms.

## MGRAPHIC LAYER

The graphic system provides two distinct ways of rendering geometries on a device. An application can draw the geometry directly to the device. The class graphic port supports a well defined, but fixed set of 2D geometries. It supports these by a set of overloaded draw methods. When using this approach, attributes and transformation matrices are not associated with geometry, making it suitable for immediate mode rendering only. The following pseudo code is an example of how an application may use this approach to create a red line.

```
{
    create a  displayPort an instance of TGrafPort
    TGLine line( TGPoint( 0.0, 0.0 , TGPoint( 1.0, 1.0 ));          //Creates a line
    TGrafBundle redColor( TRGBColor( 1.0, 0.0, 0.0 ));              //Creates a red color bundle
    displayPort->Draw(line, redColor);                  //Render the line on to the GrafPort
}
```

Alternatively, an application can draw the geometry via a higher level abstraction called MGraphic. This is a retained mode approach to rendering of graphical primitives. MGraphic is an abstract base class for representing the 2D primitives of the graphic system. It is a higher level manifestation of graphical objects which can be held in a collection, be transformed and rendered to a graphic device (TGrafDevice). Each MGraphic object holds a set of its own attributes and provides streaming capability (with some restrictions on some of its subclasses). Hit testing methods provide a mechanism for direct manipulation of MGraphic objects such as picking. MGraphic provides extensibility through subclassing that is one of the key features of MGraphics. A particular subclass of MGraphic also creates hierarchies of MGraphic objects and provides the capability to extend the graphic system. FIG. 3 illustrates some examples of graphic extensions of MGraphic in accordance with a preferred embodiment.

MGraphic is a utility class for applications to hold geometry related data that includes geometry definition, grafbundle (set of graphical attributes defining the representation of the geometry) and a set of transformation methods. MGraphic objects also hold any other information required by a user and will copy and stream this user specific data to an application. This class may not be needed for applications

interested in pure immediate mode rendering. For immediate mode rendering of the primitives the applications render geometry by passing an appropriate geometry object, a grafbundle and a transformation matrix to the graphic port. FIG. **4** illustrates MGraphics and their corresponding geometries in accordance with a preferred embodiment. FIG. **5** is a Booch diagram setting forth the flow of control of the graphic system in accordance with a preferred embodiment. In the Booch diagram of FIG. **5**, "clouds" depicted with dashed lines indicate classes or aggregations of classes (e.g. application **500**). Arrows connecting classes are directed from subclass to superclass and indicate a hierarchy including the properties of encapsulation, inheritance and polymorphism as is well understood in object technology and graphic notations accepted in the art which are illustrative thereof. Double lines indicate use of the class in the implementation or interface. A circle at one end of a line segment indicates containment or use in the class with the circle on the end of the line segment. For a more complete description of this notation, reference can be made to "Object Oriented Design" by Grady Booch, published by the Benjamin/Cummings Publishing Co., Copyright 1991. The current MGraphic **520** inherits from MDrawable **510** which inherits from MCollectible **500** to inherit the streaming, versioning and other behaviors of MCollectible **500**. Each MGraphic **520** also has a bundle, TGrafBundle **530**, which holds a set of attributes. These attributes are used by the MGraphic at rendering time.

The MGraphic abstract base class represents only 2D graphical primitives. In general it has been observed that 2D and 3D primitives do not belong to a common set unless users clear the 3D plane on which 2D primitives lie. 2D and 3D primitives have different coordinate systems and mixing them would confuse users. Clients can mix the two sets based upon their specific application requirements. The class MDrawable **510** is the abstract base class common to both MGraphic **520** and MGraphic3D abstracting the common drawing behavior of the two classes. This class is useful for clients interested only in the draw method and do not require overloaded functionality for both 2D and 3D.

## MDrawable Drawing Protocol

All MGraphics (2D and 3D) draw onto the graphic port which is passed to the MGraphic as a parameter. Besides the state information, which is encapsulated by the GrafPort, all other information is contained in the MGraphic object. This information includes the geometry, attribute bundle and any transformation information. All MGraphics draw synchronously and do not handle updating or animating requirements. It is up to the client to create subclasses. When drawing 2D and 3D primitives as a collection, such as in a list of MDrawable objects, the drawing sequence is the same as it would be when 2D and 3D draw calls are made on the graphic port. Thus, drawing a 2D polygon, a 3D box and a 2D ellipse will render differently depending upon the order in which they are rendered. The graphic port passed to this method is a passive iterator which is acted upon by the MGraphic to which it is passed.

## MGraphic Transformations

FIG. **6** illustrates a star undergoing various transformations in accordance with a preferred embodiment. Transformations can alter an MGraphic's shape, by scaling or perspective transformation, and position, by rotating and moving. The transformation methods allow applications to change an existing MGraphic's shape and location without having to recreate the MGraphic. All transformation methods apply only relative transformation to the MGraphic. Methods ScaleBy, MoveBy and RotateBy are special cases of the more general method TransformBy. Subclasses apply the transform directly to the geometry they own to directly change the geometry.

All MGraphic subclasses are closed to arbitrary transformations i.e. a TGPolygon when transformed by an arbitrary transformation will still be a TGPolygon. However, certain geometries do not possess this closure property. For example, a rectangle, when transformed by a perspective matrix, is no longer a rectangle and has no definition for either width or height. The original specification of the rectangle is insufficient to describe the transformed version of the rectangle. All MGraphic subclasses must be closed to arbitrary transformations. Since all transformations are relative, a transformed MGraphic cannot be "untransformed" by passing an identity matrix to the MGraphic method TransformBy().

FIG. **7** depicts a star moved by an amount in accordance with a preferred embodiment. This method moves the MGraphic by an amount relative to its current position. FIG. **8** illustrates rotating the star about various centers of rotation in accordance with a preferred embodiment. The amount of rotation is specified in degrees and is always clockwise. However, subclasses can override the default and optimize for a specific geometry and usage. FIG. **9** illustrates scaling a star about different centers of scale in accordance with a preferred embodiment. The factor is a vector which allows non-uniform scaling namely in X and Y. In FIG. **9** the X coordinate of the parameter amount will be (new x/old x) and the Y coordinate will be (new y/old y). In case of uniform scaling both the X and the Y coordinate will be the same. FIG. **9** also shows scaling about different centers of scale.

Negative scale factors are allowed, and the effects of negative scale factors is the same as mirroring. Scaling by −1.0 in the X direction is the same as mirroring about the Y axis while a negative scale factor in the Y direction is the same as mirroring about the X axis. FIG. **10** shows the effects of scaling an asymmetric star by (−1.0, 1.0) in accordance with a preferred embodiment. Like RotateBy() and TranslateBy(), the effect of this transform is the same as creating a scaling matrix and passing it to TransformBy() and this is the default implementation. Subclasses can override this default implementation and optimize for a specific geometry and usage. TransformBy is a pure virtual member function that transforms the MGraphic by matrix. All concrete subclasses of MGraphics must define this member function. Subclasses that own a TGrafMatrix for manipulation must post multiply the parameter matrix with the local matrix for proper effect.

## MGraphic Attribute Bundles

As seen in FIG. **5**, all MGraphic objects have an associated attribute bundle, TGrafBundle. This bundle holds the attribute information for the graphic object such as its color, pens, filled or framed. When an MGraphic is created, by default, the GrafBundle object is set to NIL. If GrafBundle is equal to a NIL, then the geometry is rendered by a default mechanism. When used in a hierarchy, the parent bundle must be concatenated with the child's bundle before rendering the child. If a child's bundle is NIL, then the child uses the parent's bundle for rendering. For example, in the hierarchy in FIG. **12**, object E will inherit the attributes of

both A, C and E before it is rendered, and a change of attribute in A will trickle down to all its children namely B, C D, E, G and D.

It is important to note that a bundle has a significant amount of information associated with it. Thus, copying of the bundle is generally avoided. Once the bundle is adopted, MGraphic object will take full responsibility to properly destroy the bundle when the MGraphic object is destroyed. When a client wishes to modify an attribute of an MGraphic object, they do so by orphaning the bundle, changing the attribute, and then having the MGraphic adopt the bundle. Also, all caches that depend upon bundles must be invalidated when the bundle is adopted or orphaned. When an object orphans data, it returns a pointer to the data and takes no further data management responsibility for the data. When an object adopts data, it takes in the pointer to the storage and assumes full responsibility for the storage. Default implementations of all bundle related member functions has been provided in the base MGraphic class and subclasses need not override this functionality, unless the subclasses have an attribute based cache which needs to be invalidated or updated whenever the bundle is adopted and orphaned. For example, the loose fit bounds, when cached, need to be invalidated (or reevaluated) when the attributes change.

C++ Application Program Interfaces (API) for Bundle Management

virtual void AdoptBundle(TGrafBundle *bundle)

MGraphic adopts the bundle.

If an MGraphic object already holds a bundle, it is deleted, and the new bundle is attached. As pointers are passed, it is important for the clients not to keep references to the bundle passed as the parameter. The MGraphic object will delete the bundle when it gets destroyed.

virtual const TGrafBundle* GetBundle() const

This method allows users to inquire a bundle and then subsequently inquire its attributes by iterating through them. This method provides an alias to the bundle stored in the MGraphic object.

virtual TGrafBundle* OrphanBundle()

This method returns a bundle to a calling application for its use. Once this method is called, it is the calling application's responsibility to delete the bundle unless it is adopted again by an MGraphic object. When orphaned, the MGraphic bundle is set to NIL, and when the graphic is subsequently drawn, the MGraphic uses the default mechanism of attributes/bundles for its parent's bundle. This kind of MGraphic subclass references other MGraphic objects. Although all manipulative behavior of complex MGraphic objects is similar to a MGraphic object, these objects do not completely encapsulate MGraphic objects they refer to. Of the subclasses supported by a preferred embodiment, the one that falls in this category is TGraphicGroup. TGraphicGroup descends from the abstract base class TBaseGraphicGroup which makes available polymorphically the methods to create iterators for traversing groups. It is important for clients creating groups or hierarchies to descend from the base class TBaseGraphicGroup for making available the iterator polymorphically. FIG. **11** illustrates the class hierarchy in accordance with a preferred embodiment.

TBaseGraphicGroup Iterator Support

Since GraphicGroup facilitates creation of hierarchies, support for iterating the hierarchy is built into this base class and is available polymorphically. This method is virtual in the abstract base class TBaseGraphicGroup and all subclasses provide an implementation. Subclasses which desire a shield for their children may return an empty iterator when this member function is invoked.

Protocol: TGraphicIterator* CreateGraphicIterator() const=0

This method creates a Graphic iterator which iterates through the first level of a hierarchy. For example in FIG. **12**, the graphic iterator created a concrete subclass to iterate over B, C and F. To iterate further, iterators must be created for both B and C as these are TBaseGraphicGroups. All subclasses creating hierarchies must provide a concrete implementation.

TGraphicIterator is an active iterator that facilitates the iteration over the children of a TBaseGraphicGroup.

TGraphicIterator methods include:

const MGraphic *TGraphicIterator::First()

const MGraphic *TGraphicIterator::Next()

const MGraphic *TGraphicIterator::Last()

TGraphicGroup

The graphic system provides a concrete subclass of TBaseGraphicGroup, namely TGraphicGroup, which supports creation of trees. TGraphicGroup creates a collection of MGraphic objects forming a group. As each of the MGraphic objects can be a TGraphicGroup, clients can create a hierarchy of objects. FIG. **12** is an example of a hierarchy created by TGraphicGroup. FIG. **12** contains TGraphicGroups A, B and C. D, E, F and G are different simple MGraphics encapsulating more than one geometry. A has references to B, C and F. B refers to D while C refers to G. Group C also refers to the MGraphic E. FIG. **12** can be considered as an over simplified bike, where A refers to MGraphic F-the body of bike, and groups B and C which refer to the transformations associated with the rear and the front wheel respectively. The two wheels are represented by the primitive geometries D and G. E represents the handle-bar of the bike. Moving node C will move both the front wheel and the handle-bar, and moving node A will move the entire bike.

While applying a transformation matrix to the children at the time of rendering, the group creates a temporary GrafPort object and concatenates its matrix with that stored in the GrafPort. This new GrafPort is used to render its children and is destroyed once the child is completely rendered. The GrafPort objects are created on the stack. TGraphicGroup does not allow its children to have more than one parent in a team. TGraphicGroup inherits directly from MGraphic and thus each of the nodes own its own grafbundle and can affect its own side of the hierarchy. The destructor of TGraphicGroup destroys itself and does not destroy its children. It is up to an application to keep track of references and destroy MGraphic objects when they are not referenced.

GraphicGroup Iterator

Graphic Group provides a concrete implementation for iterating its children. The Graphic Iterator created iterates only one level. Clients interested in iterating more than one level deep can do so by creating iterators on subsequent TGraphicGroups.

Attribute and Transformation Hierarchy

Each TGraphicGroup, if it so chooses, defines its own attributes and transformation. By default, an attribute bundle is NIL and the transformation matrix is set to the identity matrix. As TGraphicGroup is a complex MGraphic, it has

references to other MGraphics, and its children. By definition, each of the children must inherit the attribute traits and transformations of its parent. However, since each child can contain multiple references, it inherits these attributes by concatenating the parents information, without modifying its own, at the time of rendering. The concatenation of these attributes is achieved at the time of the Draw call. Both the attribute and the matrix are concatenated with the TGrafPort object which is passed as a parameter to the Draw call. In FIG. **12**, attributes and transformations of object A (body of bike) are concatenated with the GrafPort object passed to A (as parameter to member function Draw) and a new GrafPort object, APortObject, is created on the stack. APortObject is passed to object C which concatenates its state and creates a new port object, CPortObject. The new CPortObject is passed to object E to be rendered. Object E concatenates its state with CPortObject and renders itself using the new state.

## MGRAPHIC EXAMPLE

As an example, a graphic is subclassed from MGraphic to create a special 2D primitive which corresponds to a top view of a bolt. This class stores a transformation matrix for a local coordinate system, and is a very simple example without taking into account performance and efficiency. FIG. **13** illustrates a bolt object in accordance with a preferred embodiment. The code below is a C++ source listing that completely defines the bolt object in accordance with a preferred embodiment.

```
class TBoltTop : public MGraphic {
public:
        TBoltTop(GCoord BoltDiameter, GCoord outerradius, TGPoint center);
        TBoltTop(const TBoltTop&);
        TBoltTop& operator= (const TBoltTop&);
        virtual void Draw(TGrafPort&) const;
        virtual TGPoint GetAlignmentBasePoint() const;
        virtual TGRect GetLooseFitBounds() const;
        virtual TGRect GetGeometricBounds() const;
        virtual void TransformBy(const TGrafmatrix& matrix);
        virtual Boolean Find(TGrafSearcher& searcher) const;
private:
        TBoltTop();                 / /For streaming purposes only.
        TGrafMatrix fMatrix;
        TGPolygon fPolygon;         / / This is the outer polygon
        TGEllipse fCircle;          / / This is the inner circle
        void ComputePolygon(GCoord outerRad, int numOfSides);
};
TBoltTop::TBoltTop()
{
}
TBoltTop::TBoltTop(GCoord boltDia, GCoord outerDia, TGPoint center)
                                                        : fCircle(boltDia, center)
{
        calculate the hexagon polygon from these paramters
        The side of the polygon = outerDiameter / 2.0
        TGPointArray polygonPoints(6);
        TGPoint tmpPoint;
        for (unsigned long i = 0, theta = 0.0;i < 6;i ++,
                                                        theta += kPi/6) {
                tmpPoint.fX = center.fX + outerDia * sin(theta);
                tmpPoint.fY = center.fY + outerDia * cos(theta);
                polygonPoints.SetPoint(i, tmpPoint);
        }
}
void TBoltTop::Draw(TGrafPort &port) const
{
        /*
        * draw the geometry with the Grafflundle and the matris
        * associated with this primitive
        * /
        port.Draw(fPolygon, fGrafBundle, fMatrix);
        port.Draw(fCircle, fGrafBundle, fMatrix);
        /*
        * If there are a large number of primitives with same attributes
        * it is efficient to construct a local port and then render
        * geometries into this local port.
        * The semantics will be as:
        *
        * TConcatenatedGrafPort newPort(port, fGrafBundle, fMatrix);
        * TConcatenatedGrafPort is a port that concatenates bundle and
        * matrix with the state information of the old port.
        *
        * newPort.Draw(fPolygon);
        * newPort.Draw(fCircle);
        */
}
TGPoint TBoltTop::GetAlignmentBasePoint() const
{
        / / The alignment point is the center of the circle.
```

-continued

```
        TGPoint point;
        point.x = fCircle.GetCenterX();
        point.y = fCircle.GetCenterY();
        return point;
}
TGRect TBoltTop::GetLooseFitBounds() const
{
        TGRect bounds;
        / / Get bounds of the polygon
        / / pass the bounds to the bundle for altering.
        GetGeometricBounds(bounds);
        fGrafBundle->AlterBounds(bounds);
        return bounds;
}
TGRect TBoltTop::GetGeometricBounds() const
{
        / / Get bounds of the polygon
        / / pass the bounds to the bundle for altering.
        bounds = fPolygon.GetBounds();
}
void TBoltTop::TransformBy(const TGrafMatrix& matrix)
{
        fMatrix.ConcatWith(matrix);
}
void TGrafSearch::EFindResult TBoltTop::Find(TGrafSearch& search) const
{
        if (!search.find(fPolygon, fgrafBundle, fMatrix)) {
                return search.find(fCircle, fGrafBundle, fMatrix);
        }
        retum TGrafSearch::kDoneSearching;
}
```

## The Device Cache

The device cache can potentially be a large object, so care must be taken to ensure that device caches do not proliferate throughout the system unexpectedly. If the same base, GrafPort, is utilized for a number of hierarchies, the hierarchies would automatically share the cache in the base GrafPort.

## Graphic State Concatenation

FIG. **14** illustrates a hierarchical graphic in accordance with a preferred embodiment. The graphic consists of a polygon and an ellipse in a group. Each graphic in the hierarchy can store a graphic state. For instance, the polygon and the ellipse each have a TGrafBundle, while the TGroup stores no graphic state. This architecture is easily understood until hierarchical states for matrices are considered. To produce the correct geometry matrix, a graphic's local view matrix must be concatenated with the view matrix of its parent. This concatenated matrix may then be cached by the graphic that provided it. A graphic's state must be "concatenated" to that of its parent graphic, producing a new, full set of states that applies to the graphic. When TGroup::Draw is called, its parent's graphic port object is passed in. Since the TGroup has no state of its own, it doesn't perform any concatenation. It simply passes its parent's graphic port object to the polygon's Draw call and then to the ellipse's Draw call.

The polygon has a TGrafBundle object that must be concatenated to its parent's graphic port object. This is facilitated by creating a local graphic port subclass that can perform this concatenation. It then makes a call to TBundleConcatenator::Draw. FIG. **15** illustrates an object that exists inside the TPolygon's Draw call in accordance with a preferred embodiment. Because the TBundleConcatenator object is created locally to a TPolygon's Draw call, this type of concatenation is transient in nature. This processing is required for particular types of graphic hierarchies. For instance, a graphic hierarchy that allows a particular graphic to be shared by two or more other graphics must implement transient concatenation because the shared graphic has multiple parents. FIG. **16** illustrates a graphic hierarchy that supports sharing of two or more graphics in accordance with a preferred embodiment. The curve object in this example is shared by graphics B and C. Thus, the concatenation must be transient because the results of the concatenation will be different depending on the branch taken (B or C).

Graphic objects in a persistent hierarchy require knowledge of parental information, allowing a graphic to be drawn using its parent's state without drawing its parent. A graphic in the hierarchy cannot be shared by multiple parents. Extra semantics, such as a ConcatenateWithParent call and a Draw call with no parameters, must be added to the graphic classes used in the hierarchy. A graphic may use a graphic port subclass that stores more state, such as a coordinate system and clip boundary. Thus, each graphic may also want to keep its own private device cache.

FIG. **17** is a flowchart of the detailed logic in accordance with a preferred embodiment. Processing commences at function block **1700** where a modeling layer object communicates with the grafport object **1740** with a fixed set of geometric objects **1730** and an extensible set of graphic attribute objects **1720**. The grafport object **1740** passes the geometric object **1730** and graphic attributes **1720** to a polymorphic graphic device object **1750** which manages devices (hardware and software) such as a page description language object **1760**, a vector engine object **1770**, a graphic accelerator object **1780**, a frame buffer object **1790**; or more traditional graphic devices such as displays, printers or plotters as depicted in FIG. **1**.

While the invention has been described in terms of a single preferred embodiment, those skilled in the art will recognize that the invention can be practiced with modifi-

cation 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. An object-oriented graphic system, comprising:

(a) a processor;

(b) a storage under the control of and attached to the processor;

(c) one or more graphic devices under the control of and attached to the processor;

(d) a grafport object in the storage of the processor;

(e) a graphic device object in the storage of the processor for managing one of the one or more graphic devices;

(f) a graphic object in the storage of the processor for managing graphic processing; and

(g) means for connecting the graphic device object to the grafport object to output graphic information on the one of the one or more graphic devices under the control of the graphic object.

2. A system as recited in claim 1, including a graphic accelerator graphic device object.

3. A system as recited in claim 1, including a frame buffer graphic device object.

4. A system as recited in claim 1, including a page description language graphic device object.

5. A system as recited in claim 1, including a vector engine graphic device object.

6. A system as recited in claim 1, wherein the grafport object, the graphic device object and the graphic object are polymorphic.

7. A system as recited in claim 1, wherein the grafport object, the graphic device object and the graphic object are fully extensible.

8. A system as recited in claim 1, including a modeling layer in the graphic object.

9. A system as recited in claim 8, including a geometric object and a graphic attribute object in the modeling layer.

10. A system as recited in claim 1, wherein the geometric object includes geometry for the graphic information.

11. A system as recited in claim 1, wherein the graphic device objects include displays, printers and plotters.

12. A method for graphic processing in an object-oriented operating system resident on a computer with a processor, a storage attached to and under the control of the processor and a graphic device attached to and under the control of the processor, comprising the steps of:

(a) building a modeling layer object in the storage;

(b) generating calls from the modeling layer object to grafport object using a predefined set of graphic primitives;

(c) capturing state information and rendering information at the grafport object; and

(d) passing the state information and the rendering information to a graphic device object for output on the graphic device.

13. The method as recited in claim 12, including state information with transform, appearance and clipping information.

14. The method as recited in claim 12, wherein the graphic device is a software or a hardware graphic processor.

15. An apparatus for graphic processing, comprising:

(a) a processor,

(b) a storage attached to and under the control of the processor;

(c) a graphic device attached to and under the control of the processor;

(d) a modeling layer object in the storage;

(e) a grafport object in the storage;

(f) means for generating calls from the modeling layer object to the grafport object using a predefined set of graphic primitives;

(g) means for capturing state information and rendering information at the grafport object; and

(h) means for passing the state information and the rendering information to a graphic device object for output on the graphic device.

16. The apparatus as recited in claim 15, wherein the state information includes transform, appearance and clipping information.

17. The apparatus as recited in claim 15, wherein the graphic device is a vector engine.

18. The apparatus as recited in claim 15, wherein the graphic device is a graphic accelerator.

19. The apparatus as recited in claim 15, wherein the graphic device is a frame buffer.

20. The apparatus as recited in claim 15, wherein the graphic device is a plotter.

21. The apparatus as recited in claim 15, wherein the graphic device is a printer.

22. The apparatus as recited in claim 15, wherein the graphic device is a display.

23. The apparatus as recited in claim 15, wherein the graphic device is a postscript processor.

24. The apparatus as recited in claim 15, wherein the modeling layer object includes at least one geometric object and at least one graphic attribute object.

25. The apparatus as recited in claim 15, wherein an object includes a method and data.

26. The apparatus as recited in claim 25, wherein the object is polymorphic and extensible.

* * * * *

# EXHIBIT 2

US005519867A

# United States Patent [19]

**Moeller et al.**

[11] **Patent Number: 5,519,867**

[45] **Date of Patent: *May 21, 1996**

[54] **OBJECT-ORIENTED MULTITASKING SYSTEM**

[75] Inventors: **Christopher P. Moeller**, Los Altos; **Eugenie L. Bolton**, Sunnyvale; **Daniel F. Chernikoff**, Palo Alto; **Russell T. Nakano**, Sunnyvale, all of Calif.

[73] Assignee: **Taligent, Inc.**, Cupertino, Calif.

[ * ] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,379,432.

[21] Appl. No.: **94,673**

[22] Filed: **Jul. 19, 1993**

[51] **Int. Cl.**[6] ..... **G06F 9/40**

[52] **U.S. Cl.** ..... **395/700**

[58] **Field of Search** ..... 395/650, 700

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,821,220 | 4/1989 | Duisberg | 364/578 |
| 4,885,717 | 12/1989 | Beck et al. | 364/900 |
| 4,891,630 | 1/1990 | Friedman et al. | 340/706 |
| 4,953,080 | 8/1990 | Dysart 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 | 12/1992 | 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 |
| 5,151,987 | 9/1992 | Abraham et al. | 395/575 |
| 5,181,162 | 1/1993 | Smith et al. | 364/419 |
| 5,379,432 | 1/1995 | Orton et al. | 395/700 |
| 5,404,529 | 4/1995 | Chernihoff et al. | 395/700 |

OTHER PUBLICATIONS

Proc. of the Summer 1988 Usenix Conf. 20 Jun. 1988, San Francisco, US, pp. 1–13 "Using the X Toolkit or How to Write a Widget" by McCormack et al.

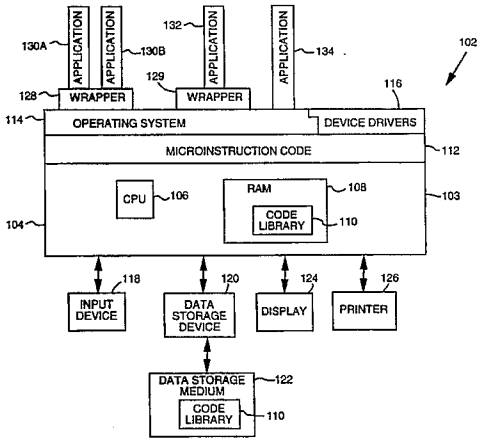
New Directions for Unix. Proc. Autumn 1988 EUUG Conf. Oct. 3, 1988, Cascais, Portugal, pp. 25–37, Bernabeu–Auban et al. "Clouds—A Distributed Object–Based Operating System Architecture and Kernel Implementation".

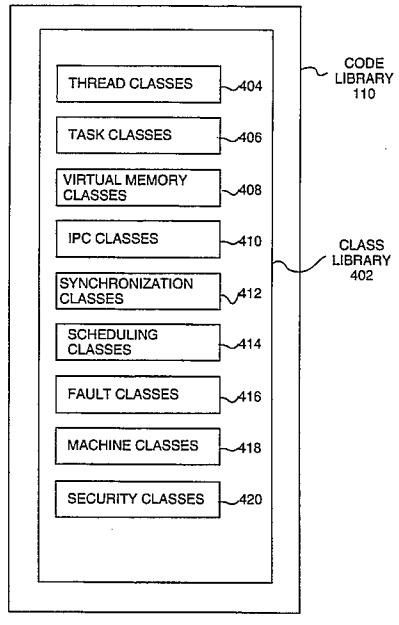
*Primary Examiner*—Alvin E. Oberley
*Assistant Examiner*—John Q. Chavis
*Attorney, Agent, or Firm*—Keith Stephens

[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. The object-oriented application includes support for multi-tasking.

**53 Claims, 17 Drawing Sheets**



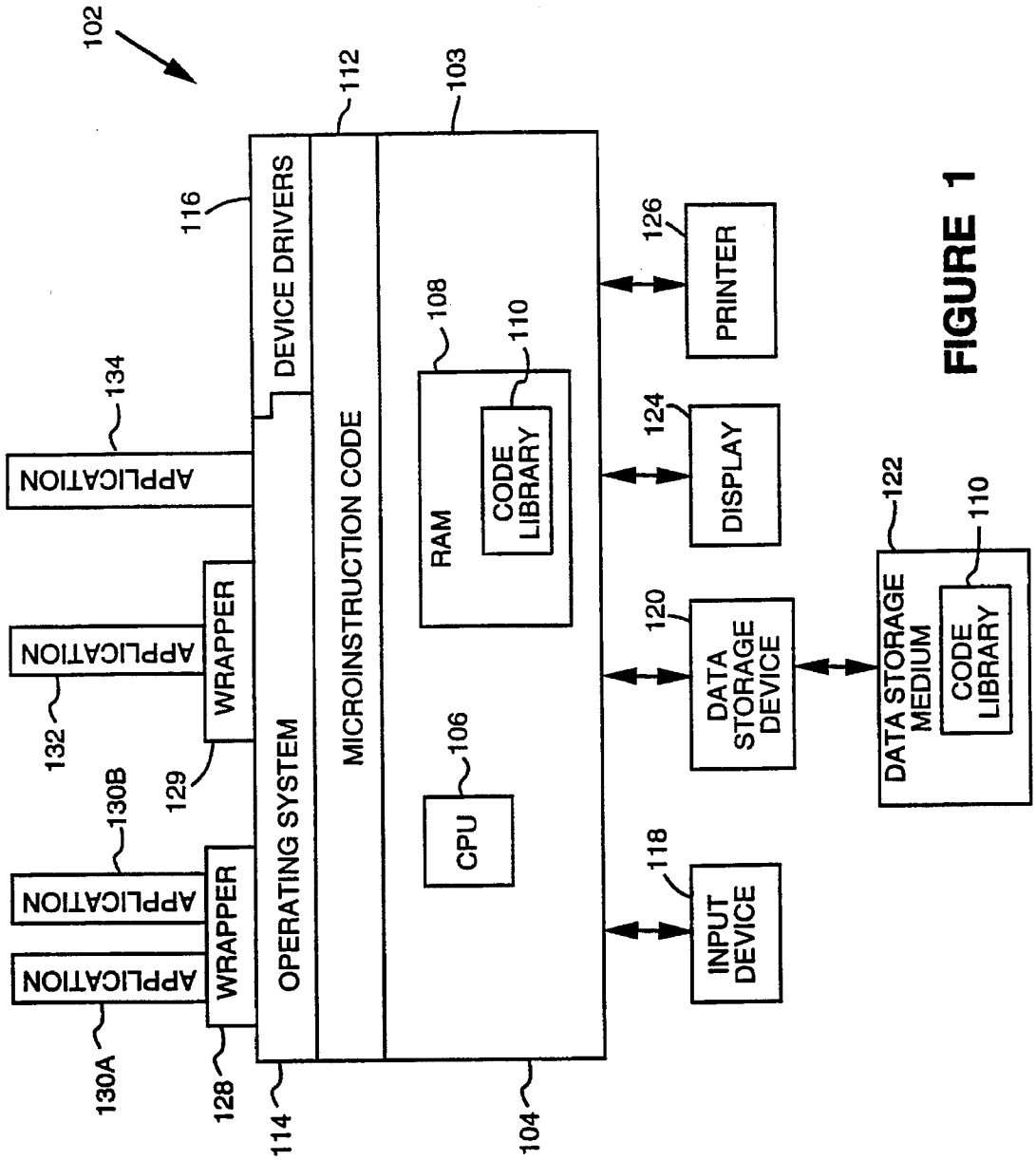

102
112
103
116
DEVICE DRIVERS
MICROINSTRUCTION CODE
OPERATING SYSTEM
108
110
RAM
CODE LIBRARY
106
CPU
134
APPLICATION
132
APPLICATION
WRAPPER
129
130B
APPLICATION
WRAPPER
APPLICATION
130A
128
114
104
126
PRINTER
124
DISPLAY
120
DATA STORAGE DEVICE
118
INPUT DEVICE
122
DATA STORAGE MEDIUM
110
CODE LIBRARY

FIGURE 1

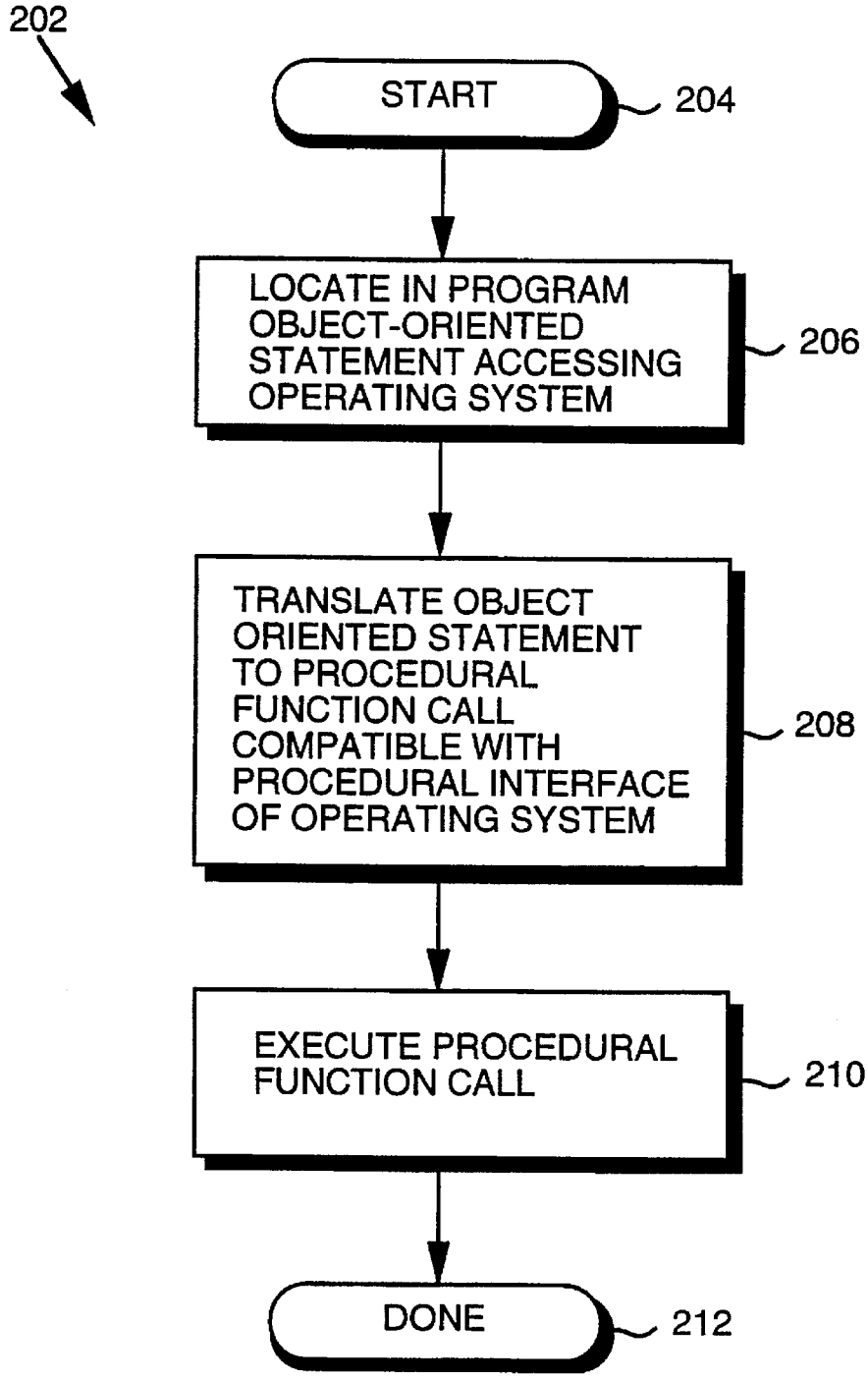

202
START
204
LOCATE IN PROGRAM OBJECT-ORIENTED STATEMENT ACCESSING OPERATING SYSTEM
206
TRANSLATE OBJECT ORIENTED STATEMENT TO PROCEDURAL FUNCTION CALL COMPATIBLE WITH PROCEDURAL INTERFACE OF OPERATING SYSTEM
208
EXECUTE PROCEDURAL FUNCTION CALL
210
DONE
212

Figure 2

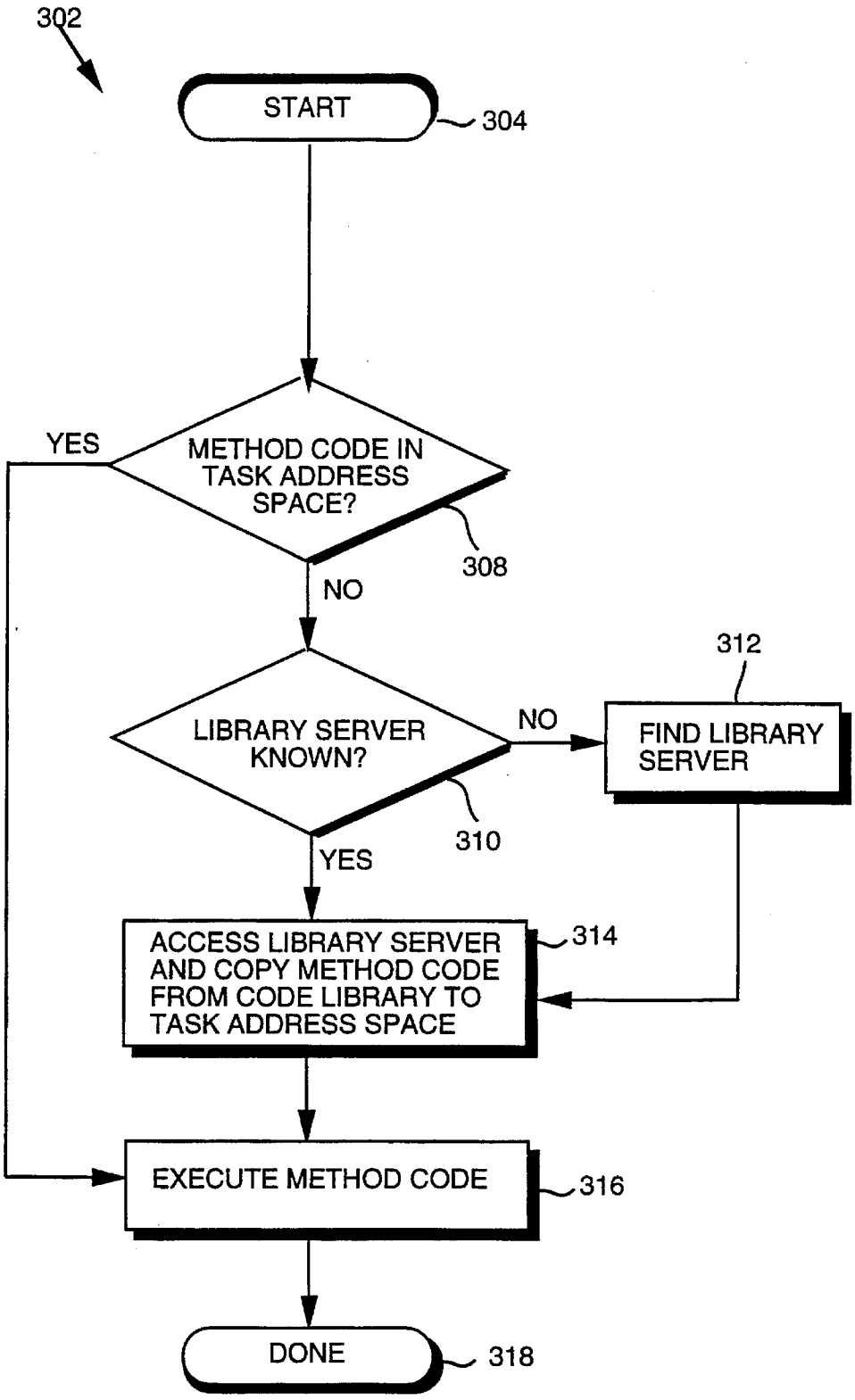

302
START
304
METHOD CODE IN TASK ADDRESS SPACE?
YES
308
NO
LIBRARY SERVER KNOWN?
NO
312
FIND LIBRARY SERVER
310
YES
ACCESS LIBRARY SERVER AND COPY METHOD CODE FROM CODE LIBRARY TO TASK ADDRESS SPACE
314
EXECUTE METHOD CODE
316
DONE
318

Figure 3

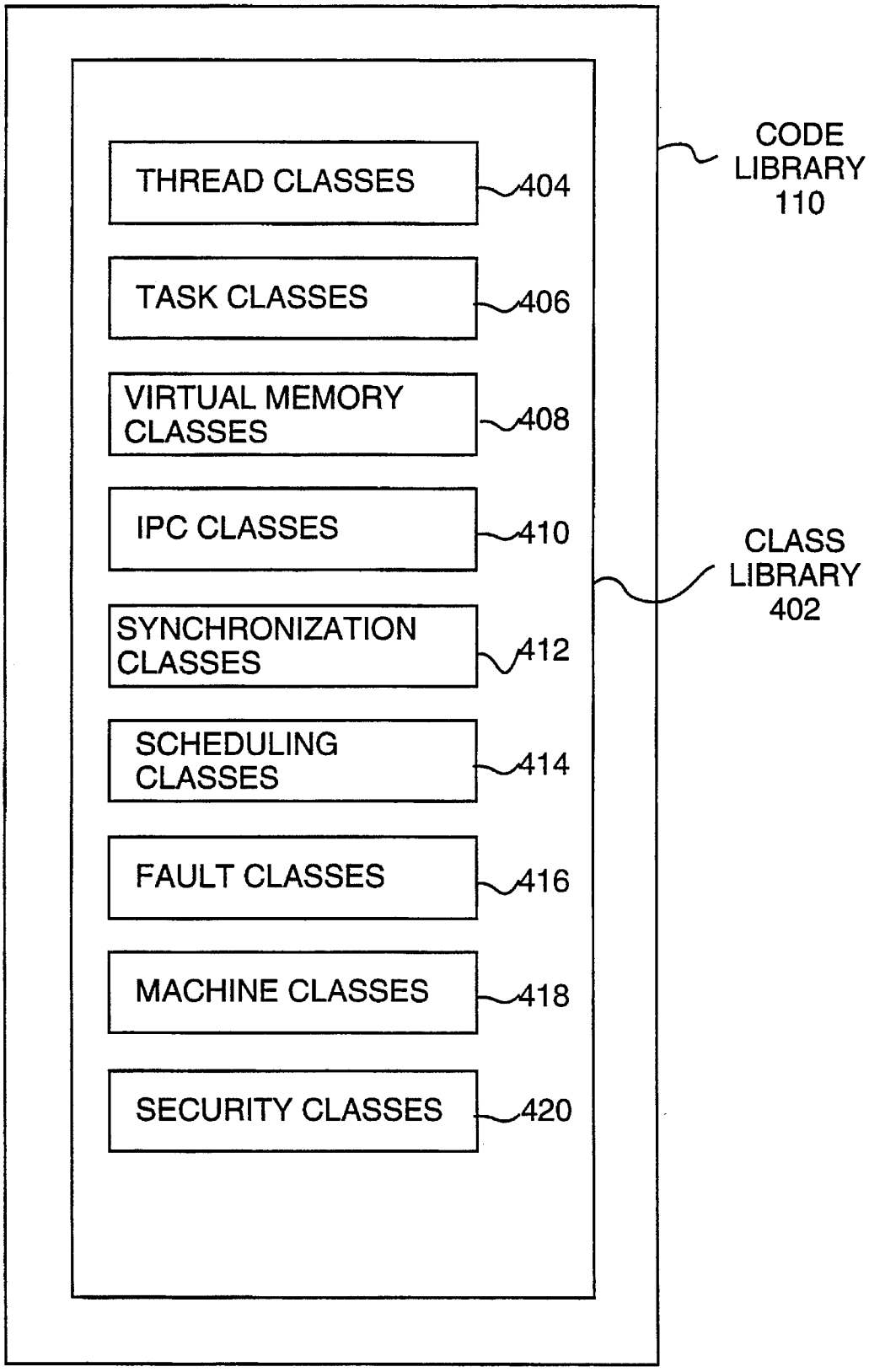

THREAD CLASSES
404
TASK CLASSES
406
VIRTUAL MEMORY CLASSES
408
IPC CLASSES
410
SYNCHRONIZATION CLASSES
412
SCHEDULING CLASSES
414
FAULT CLASSES
416
MACHINE CLASSES
418
SECURITY CLASSES
420
CLASS LIBRARY 402
CODE LIBRARY 110

FIGURE 4

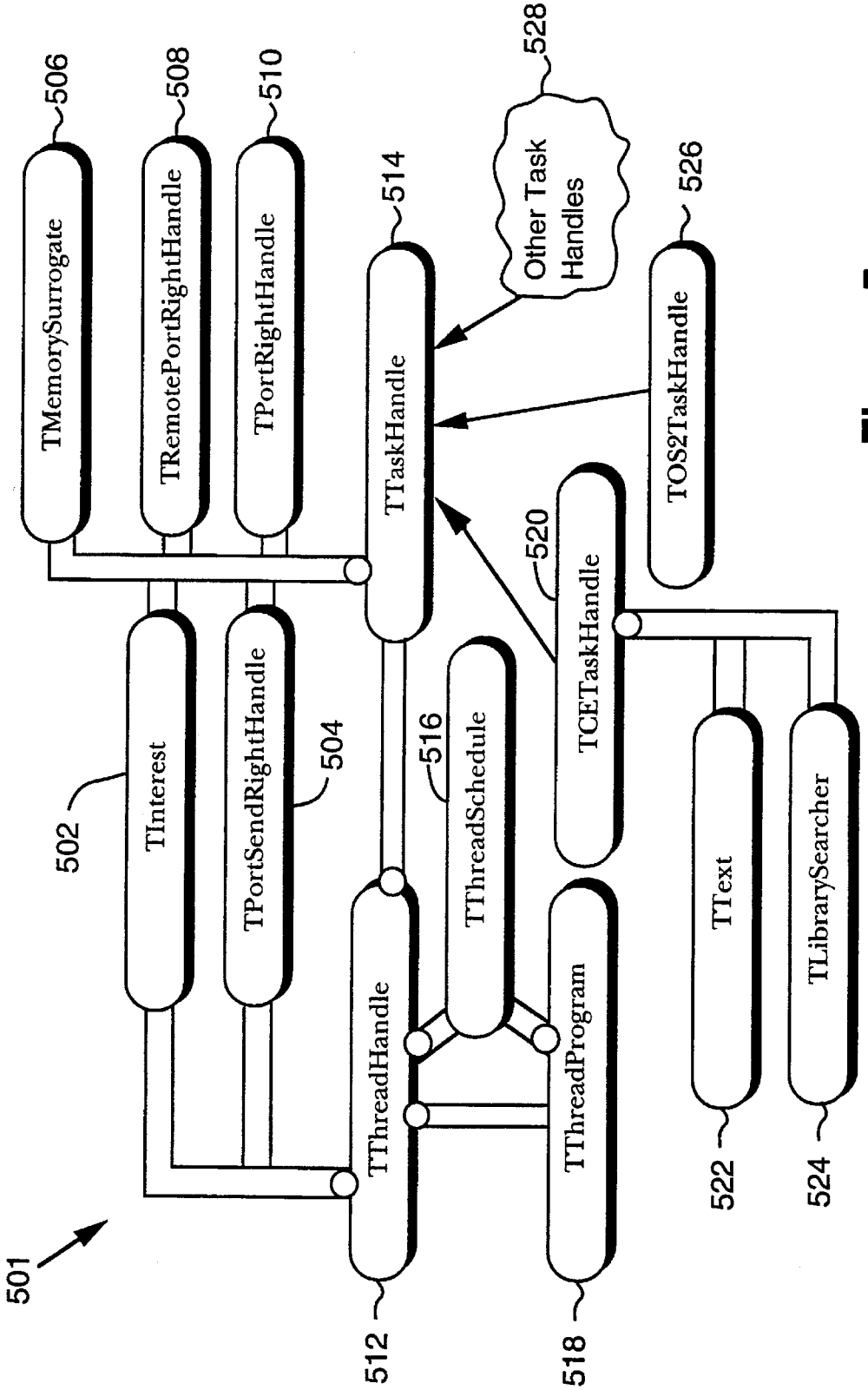

501
502
TInterest
504
TPortSendRightHandle
506
TMemorySurrogate
508
TRemotePortRightHandle
510
TPortRightHandle
512
TThreadHandle
514
TTaskHandle
516
TThreadSchedule
518
TThreadProgram
520
TCETaskHandle
522
TText
524
TLibrarySearcher
526
TOS2TaskHandle
528
Other Task Handles

Figure 5

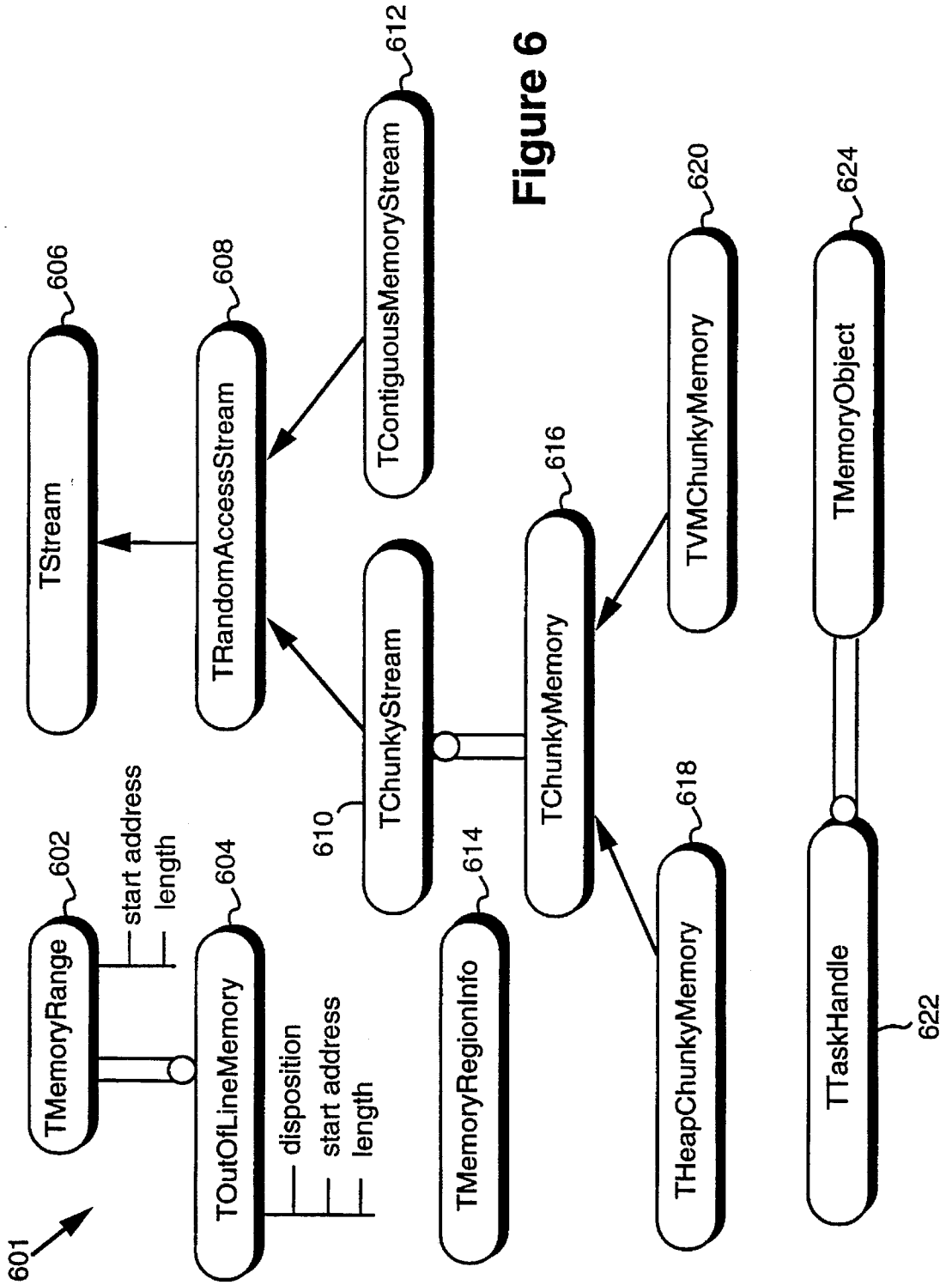

601
TMemoryRange
602
start address
length
TOutOfLineMemory
604
disposition
start address
length
TStream
606
TRandomAccessStream
608
TContiguousMemoryStream
612
610
TChunkyStream
TMemoryRegionInfo
614
TChunkyMemory
616
TVMChunkyMemory
620
THeapChunkyMemory
618
TMemoryObject
624
TTaskHandle
622

Figure 6

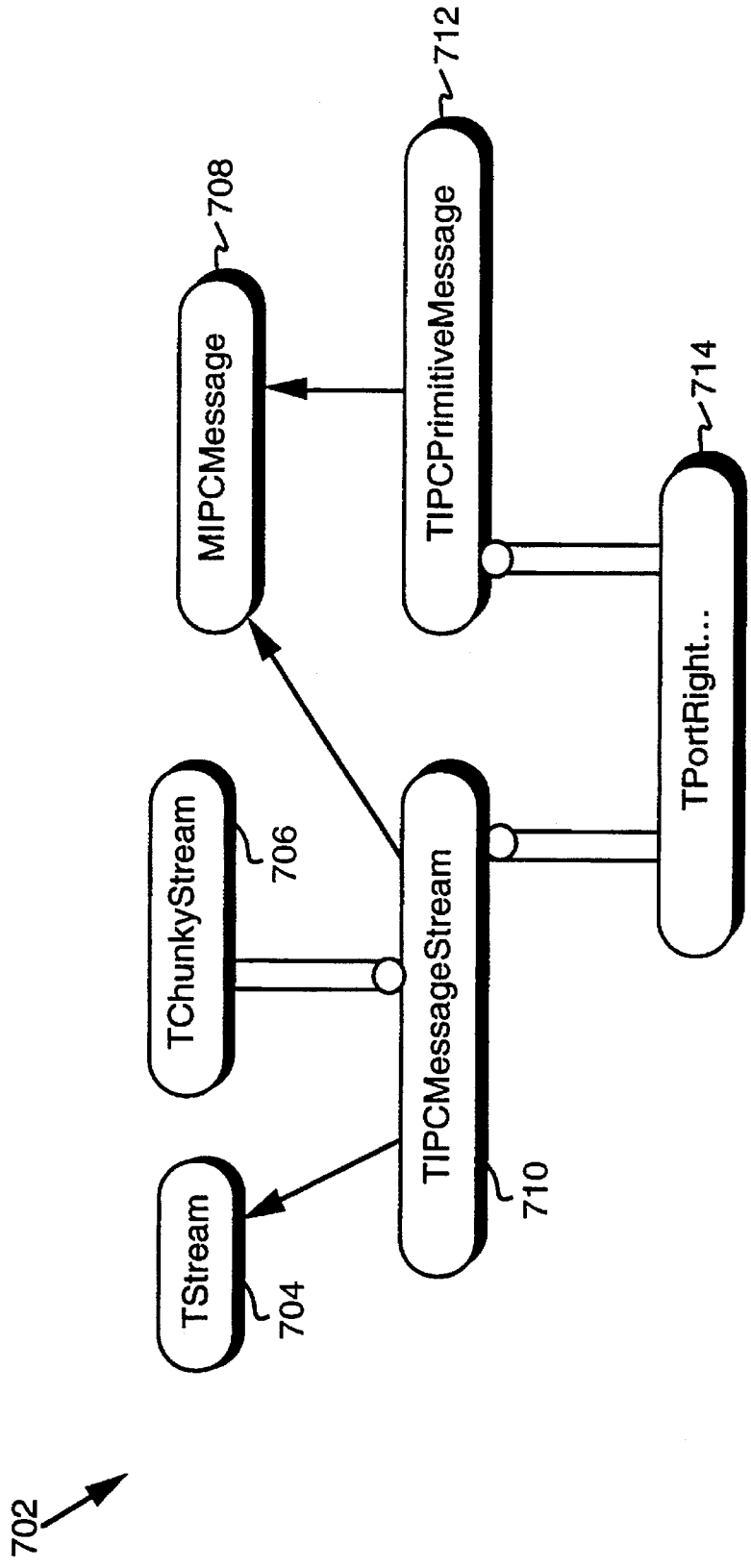

702
TStream
704
TChunkyStream
706
MIPCMessage
708
TIPCMessageStream
710
TIPCPrimitiveMessage
712
TPortRight...
714

Figure 7

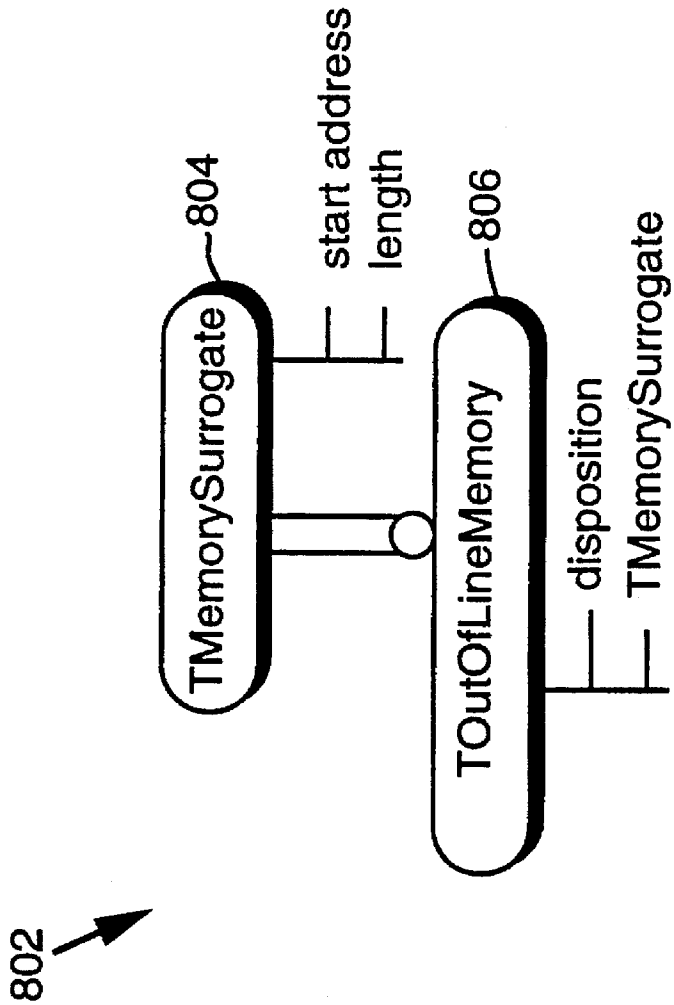

802
804
TMemorySurrogate
start address
length
806
TOutOfLineMemory
disposition
TMemorySurrogate

Figure 8

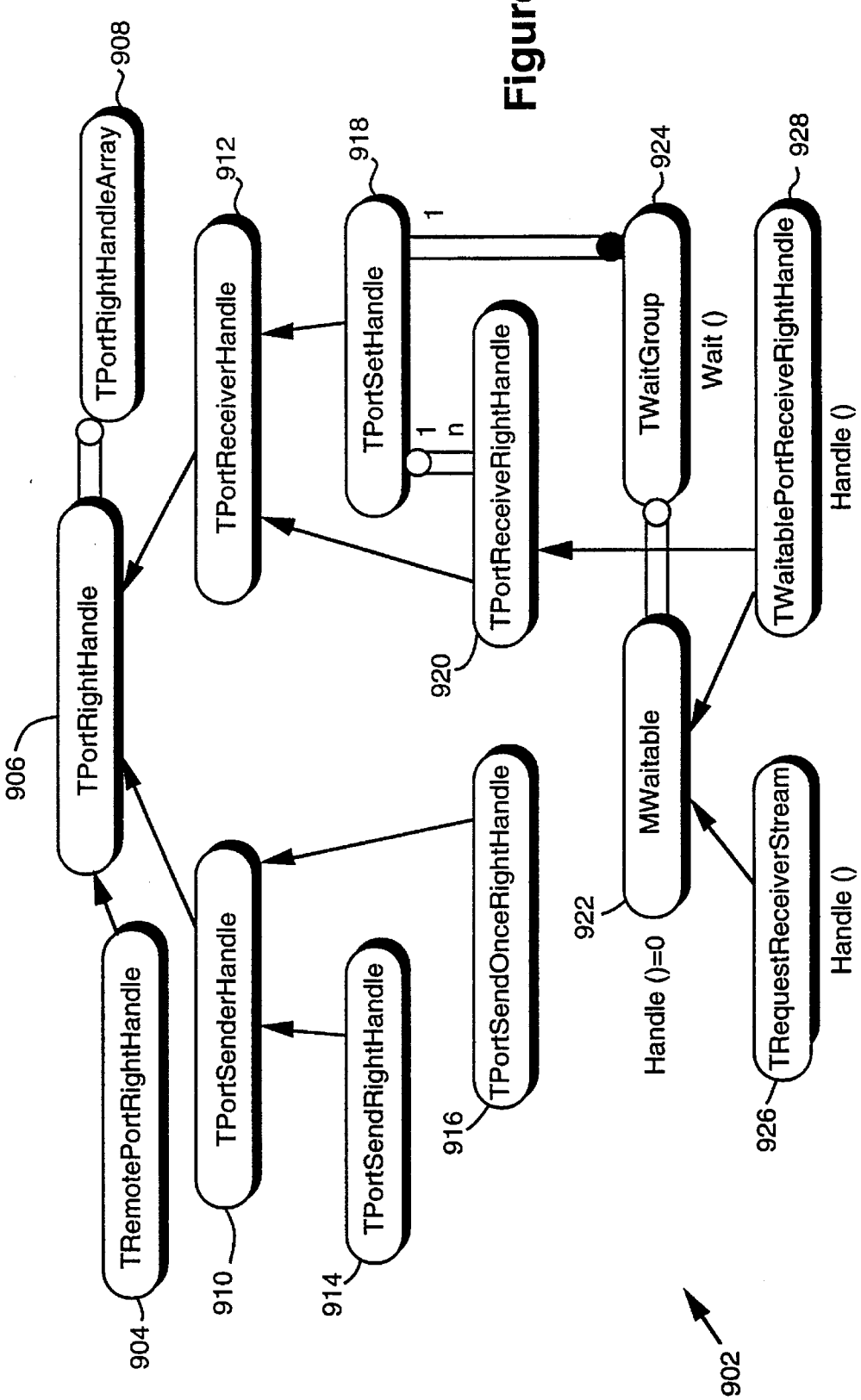

902
904
TRemotePortRightHandle
906
TPortRightHandle
908
TPortRightHandleArray
910
TPortSenderHandle
912
TPortReceiverHandle
914
TPortSendRightHandle
916
TPortSendOnceRightHandle
918
TPortSetHandle
1
1
n
920
TPortReceiveRightHandle
922
MWaitable
Handle ()=0
924
TWaitGroup
Wait ()
926
TRequestReceiverStream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

Figure 9

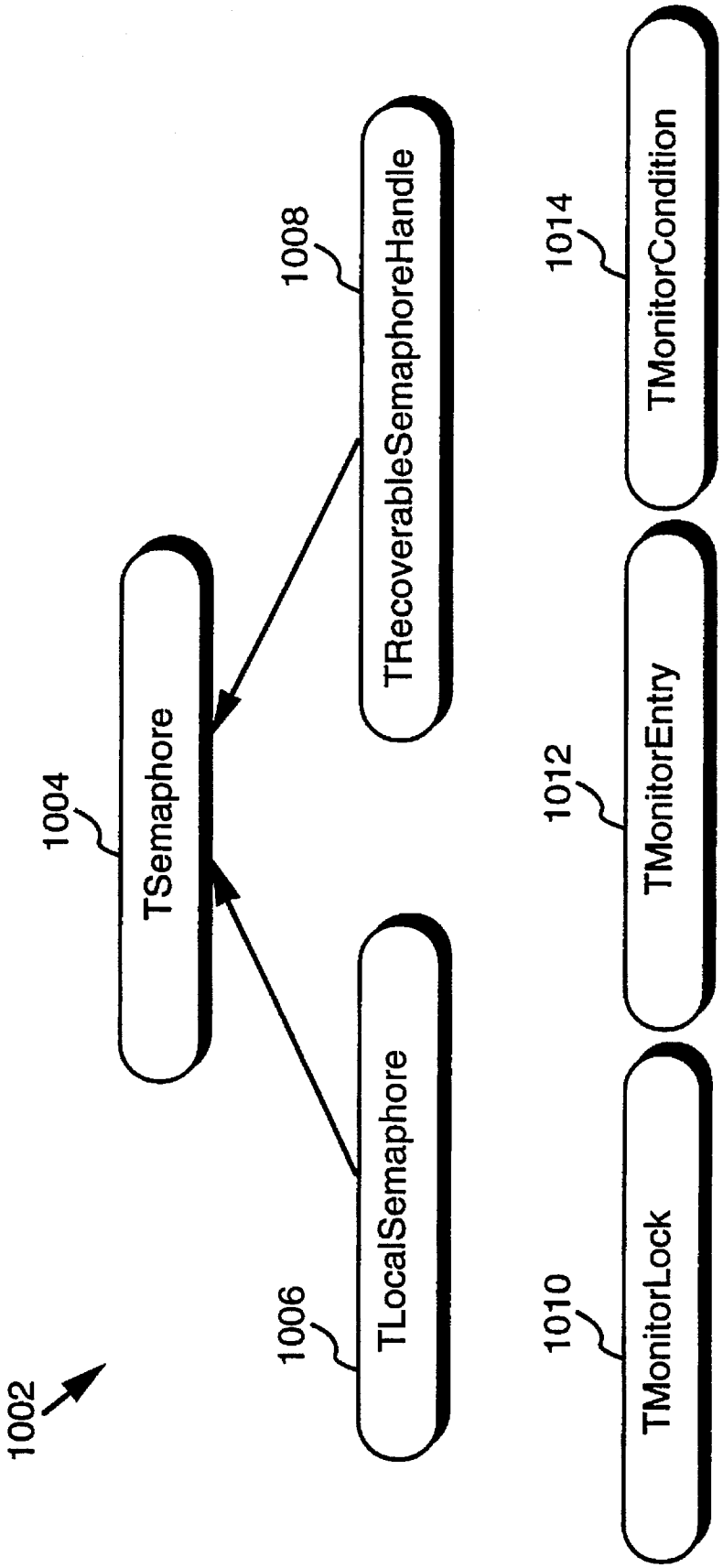

1002
1004
TSemaphore
1006
TLocalSemaphore
1008
TRecoverableSemaphoreHandle
1010
TMonitorLock
1012
TMonitorEntry
1014
TMonitorCondition

Figure 10

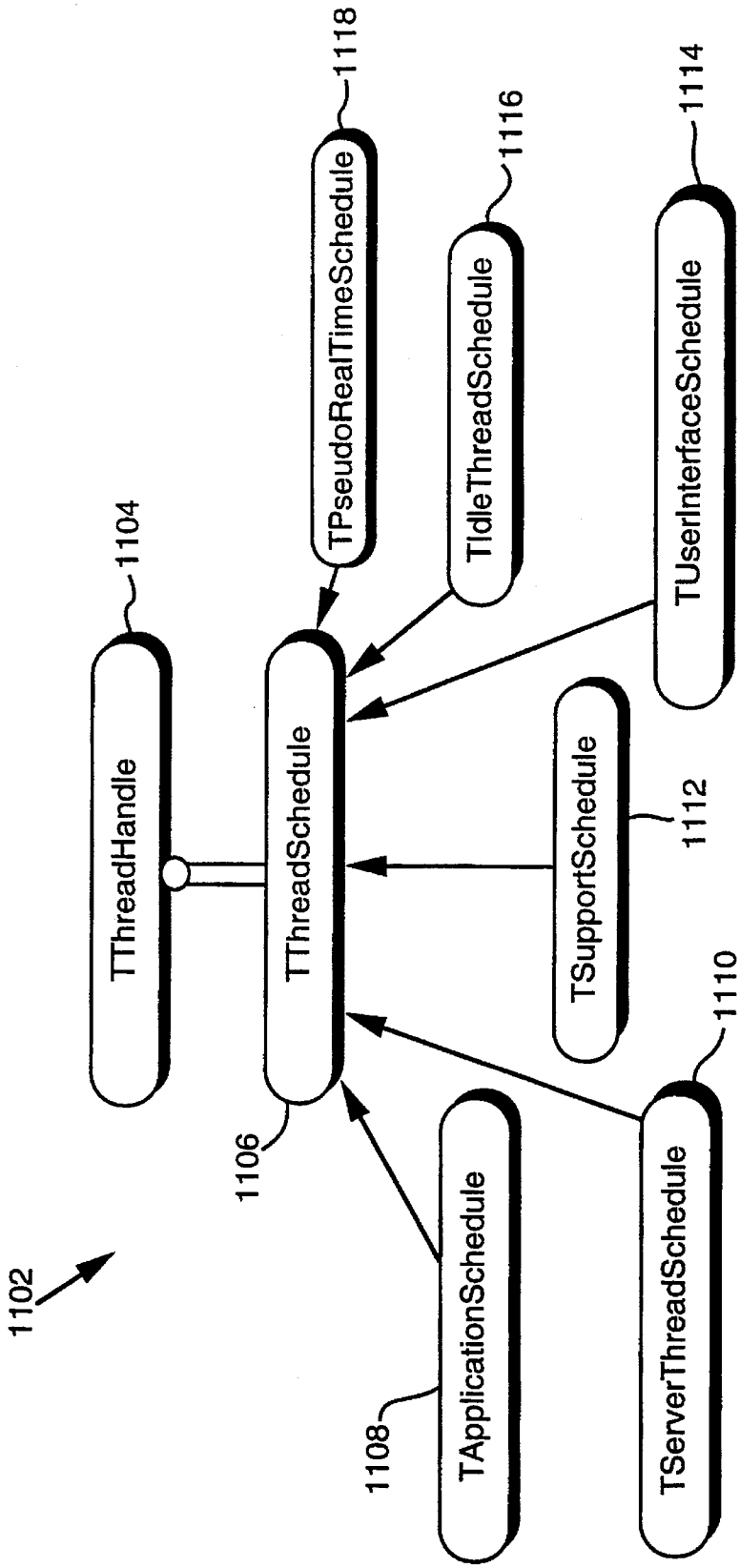

1102
TThreadHandle
1104
TThreadSchedule
1106
TPseudoRealTimeSchedule
1118
TIdleThreadSchedule
1116
TUserInterfaceSchedule
1114
TSupportSchedule
1112
TApplicationSchedule
1108
TServerThreadSchedule
1110

Figure 11

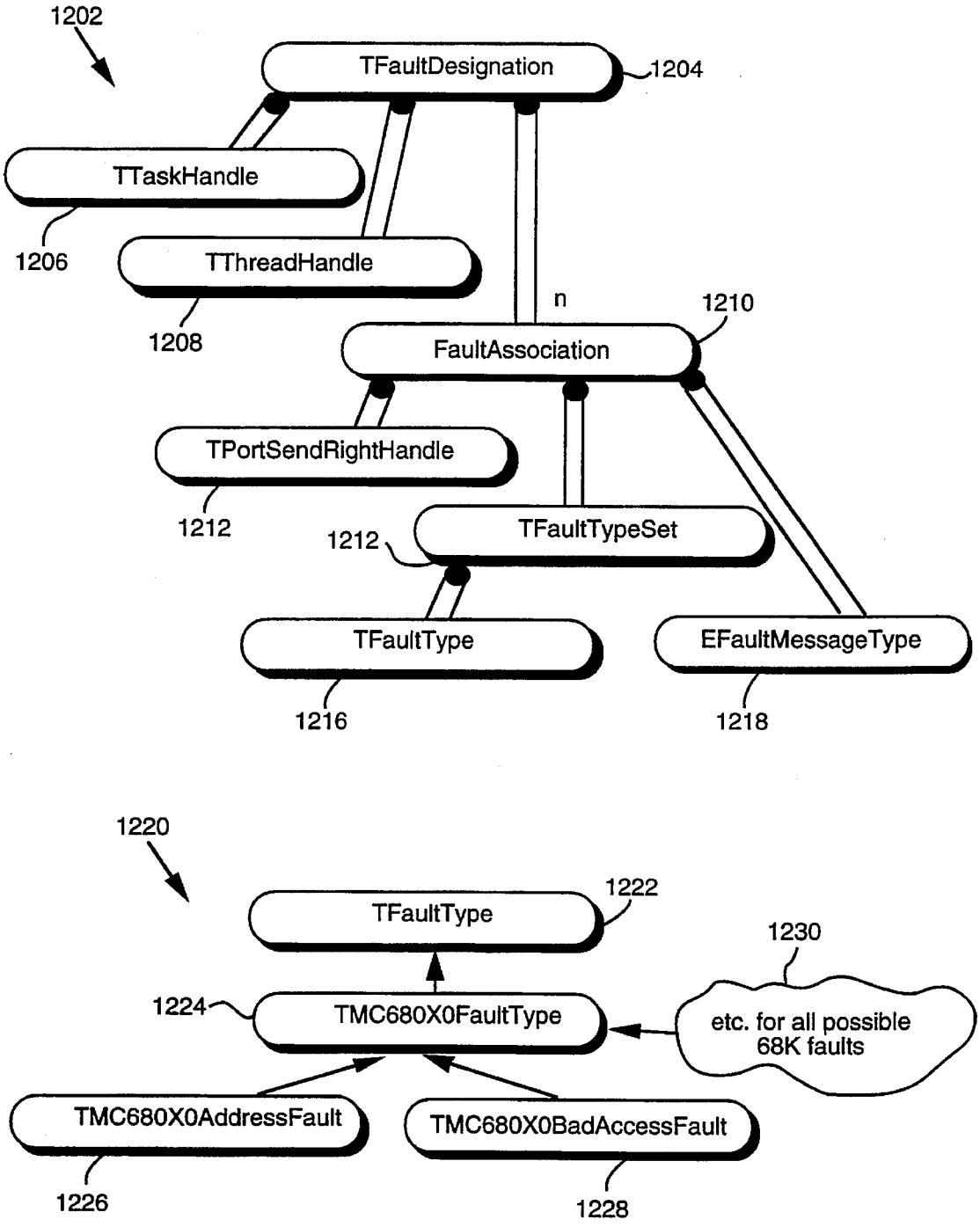

1202
TFaultDesignation
1204
TTaskHandle
1206
TThreadHandle
1208
n
1210
FaultAssociation
TPortSendRightHandle
1212
1212
TFaultTypeSet
TFaultType
1216
EFaultMessageType
1218
1220
TFaultType
1222
1230
1224
TMC680X0FaultType
etc. for all possible 68K faults
TMC680X0AddressFault
1226
TMC680X0BadAccessFault
1228

Figure 12

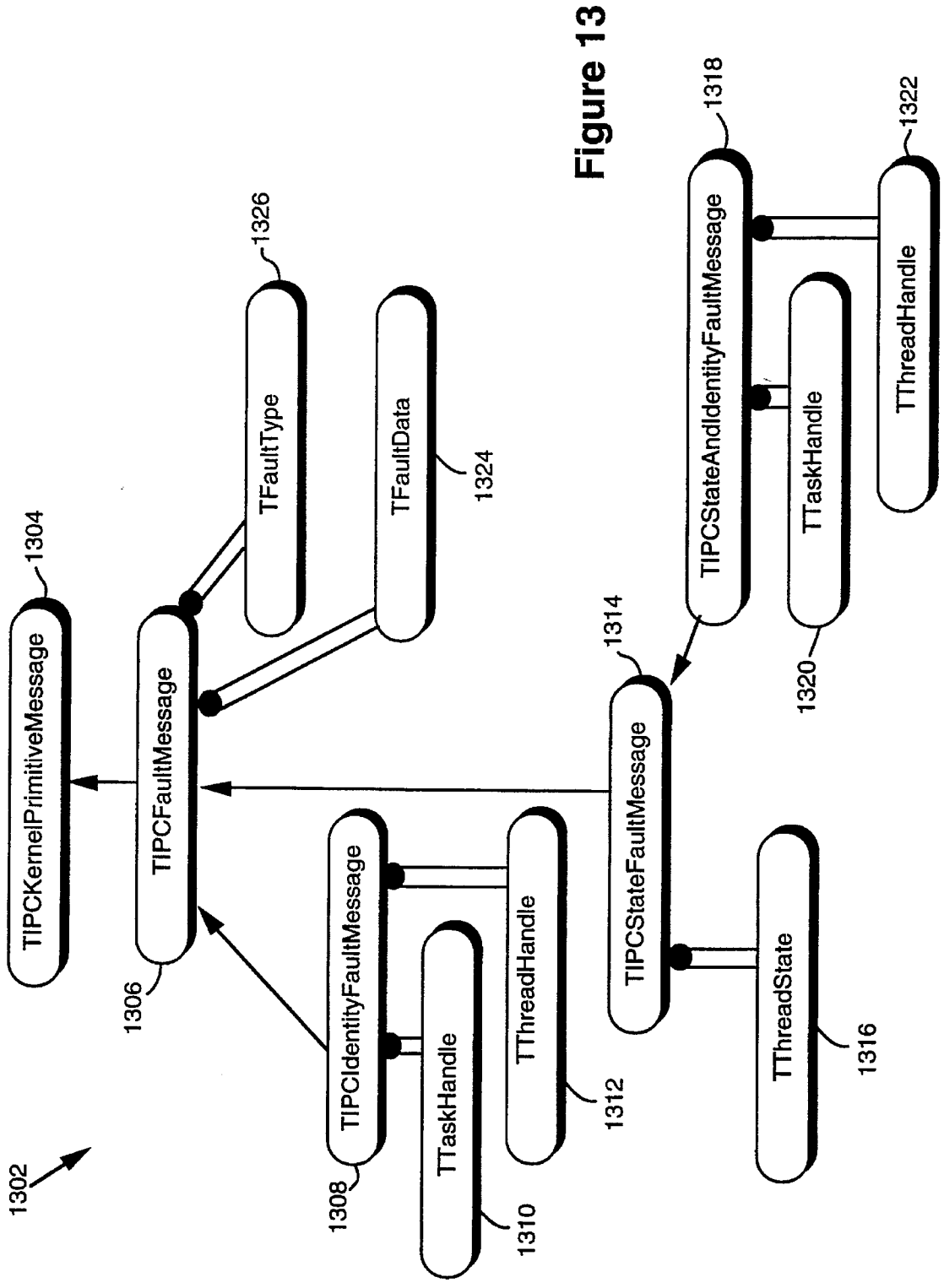

1302
TIPCKernelPrimitiveMessage
1304
TIPCFaultMessage
1306
TFaultType
1326
TFaultData
1324
TIPCIdentityFaultMessage
1308
TTaskHandle
1310
TThreadHandle
1312
TIPCStateFaultMessage
1314
TIPCStateAndIdentityFaultMessage
1318
TTaskHandle
1320
TThreadHandle
1322
TThreadState
1316

Figure 13

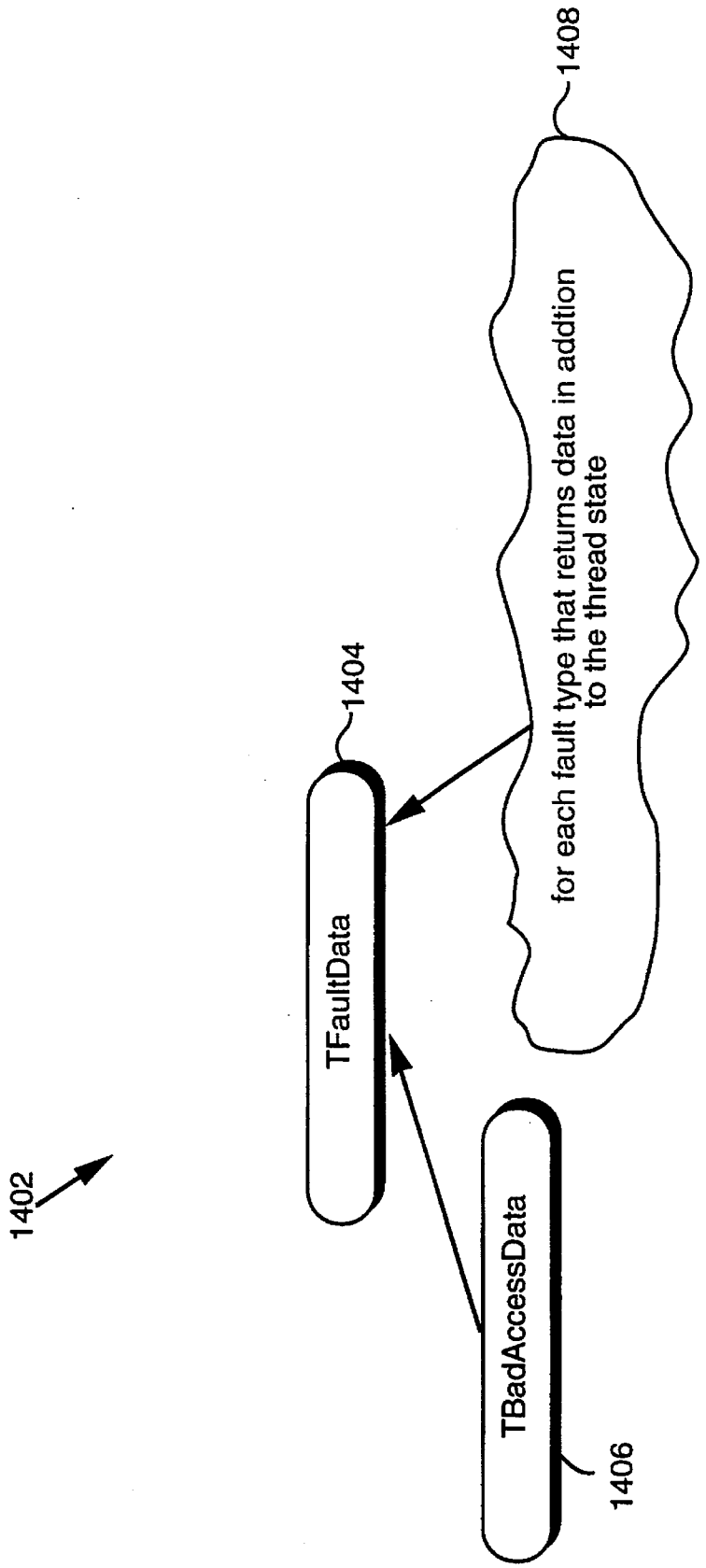

1402
1404
TFaultData
1406
TBadAccessData
1408
for each fault type that returns data in addition to the thread state

Figure 14

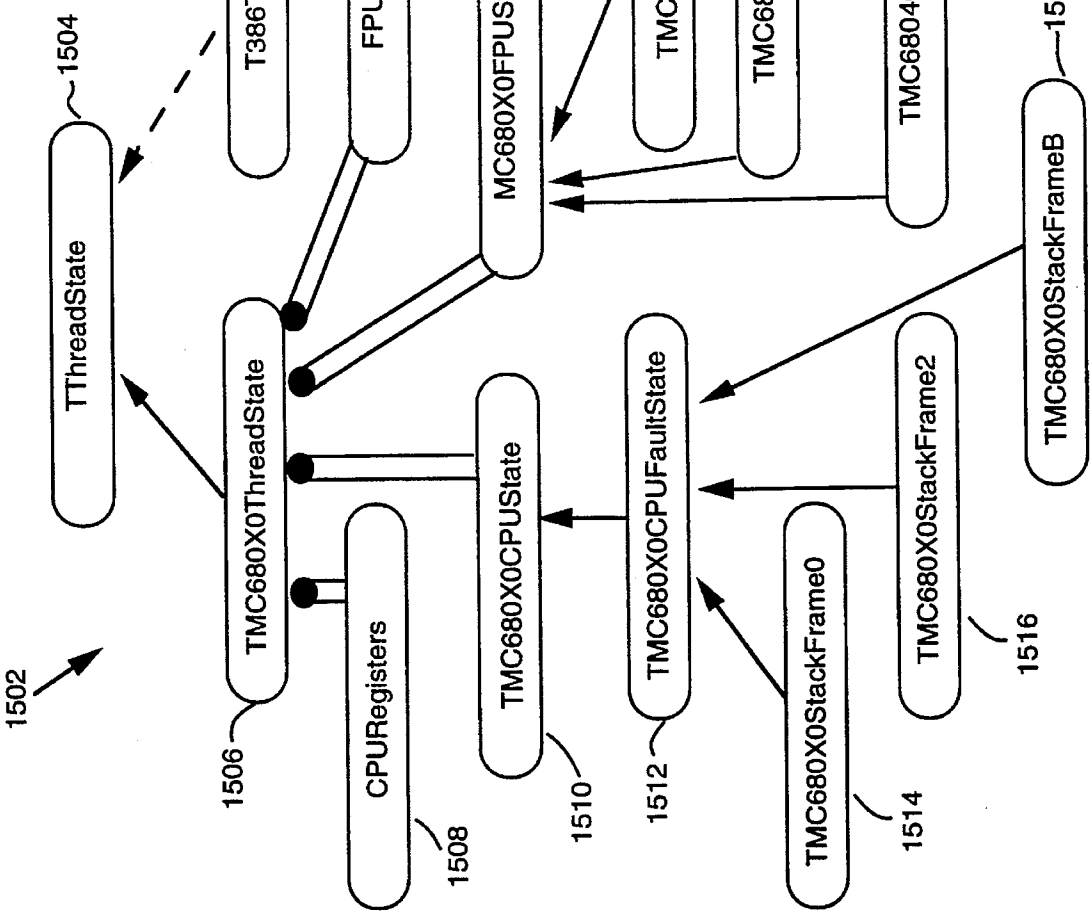

1502
TThreadState
1504
T386ThreadState
1532
FPURegisters
1530
TMC680X0ThreadState
1506
CPURegisters
1508
MC680X0FPUState
1520
TMC68040Busy
1524
TMC68881Busy
1526
TMC68040Idle
1528
TMC680X0CPUState
1510
TMC680X0CPUFaultState
1512
TMC680X0StackFrame0
1514
TMC680X0StackFrame2
1516
TMC680X0StackFrameB
1518

FIGURE 15

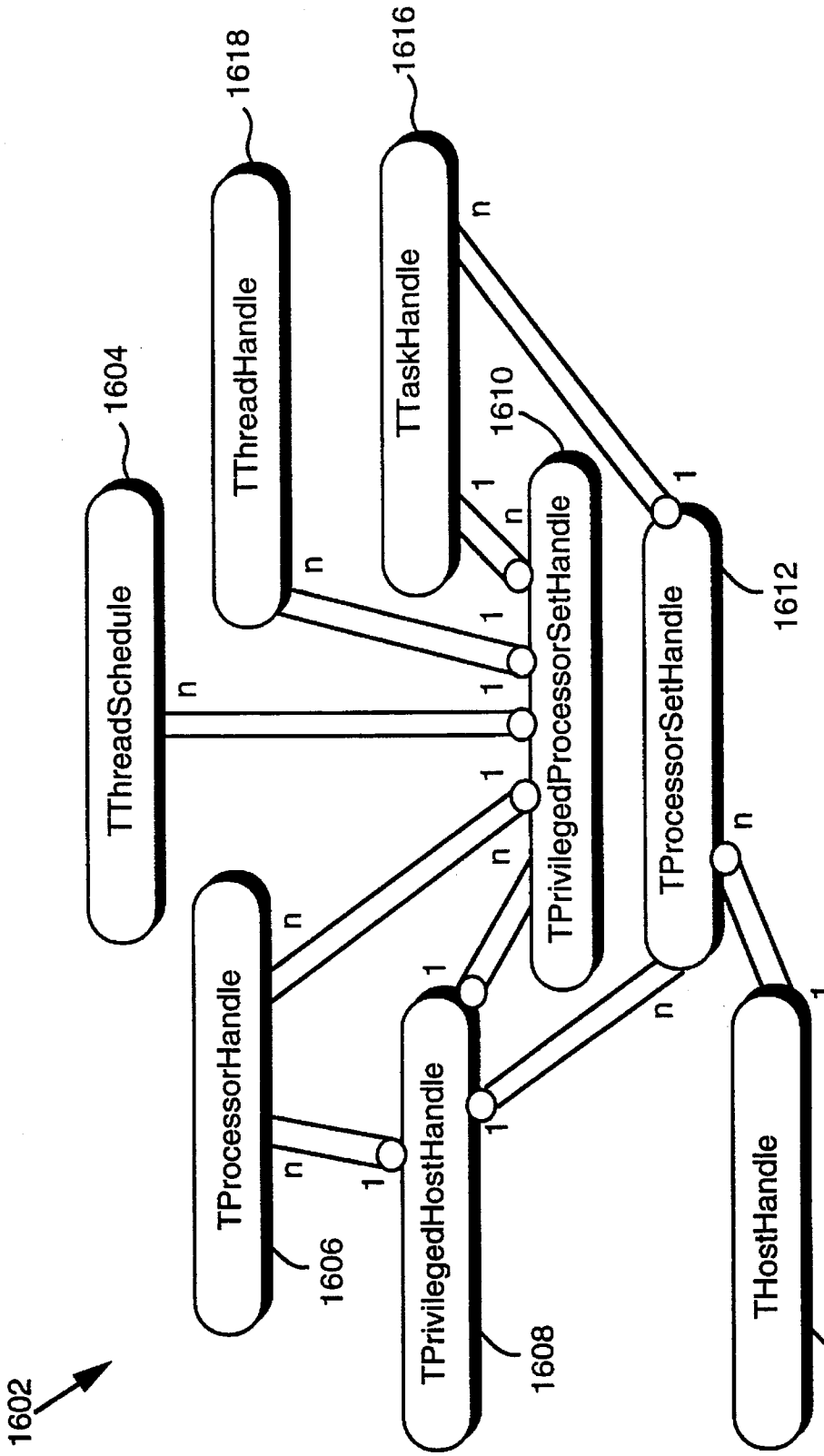

1602
TThreadSchedule
1604
TThreadHandle
1618
TTaskHandle
1616
TPrivilegedProcessorSetHandle
1610
TProcessorSetHandle
1612
TProcessorHandle
1606
TPrivilegedHostHandle
1608
THostHandle
1614
n
1

Figure 16

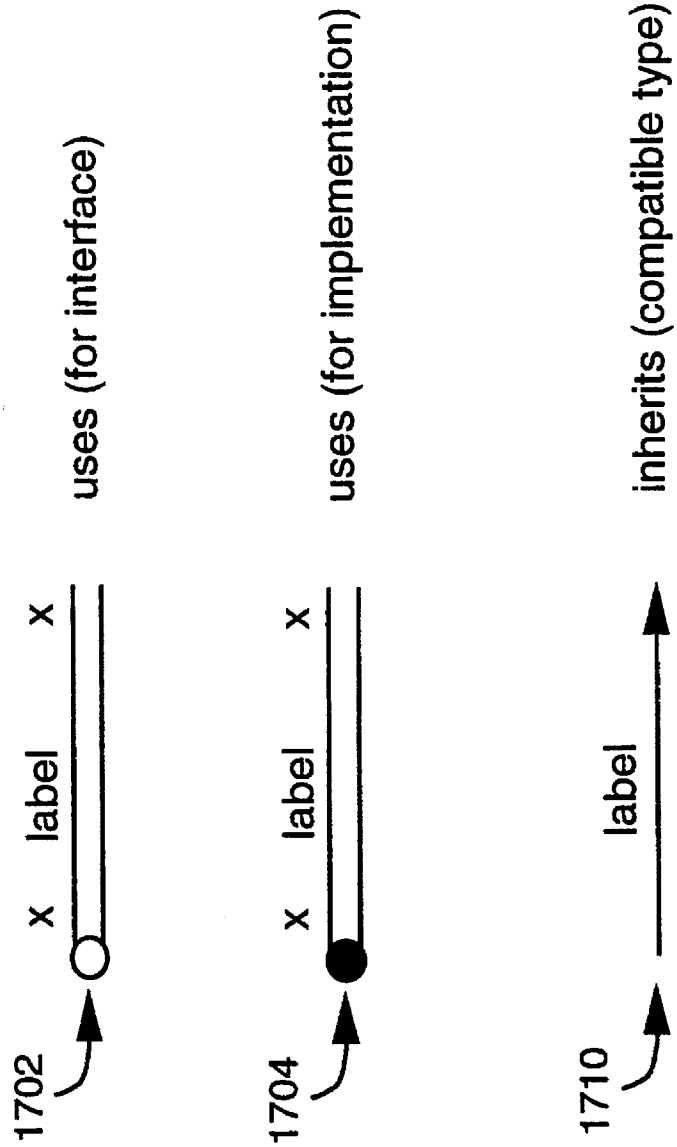

uses (for interface)
x label x
1702
uses (for implementation)
x label x
1704
inherits (compatible type)
label
1710

Figure 17

# OBJECT-ORIENTED MULTITASKING SYSTEM



1. 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.

2. 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 the 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 class 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 **112** (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 environment). 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** includes 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 include 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 memory, 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 statement 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 inter-process 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 of fault 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. The fault 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 should 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 memory in the address space and deallocate ranges as desired. Another important operation 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 mes-

sages 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 a 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 messages 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 timeout 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 token, which is a scalar value that is uninterpreted by Mach. There is a port called the host 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 pro-

vides 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 object. 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 terminates 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& taskDescription): 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 TMemorySurrogate& 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 resolve lib referneces 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 TITaskHandle 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 TRandomAccessStream, all random access operations (like Seek()) 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 tip 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 tip 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 way 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= kPositiveInfinity) 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 maximum 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 TTime& 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, TServerSchdule 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.

ReturnNumberOfLevels (): Return the number of levels of urgency for 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:

TMC680XOFaultType represents a fault type on a Motorola 68K processor. It identifies the possible 68K type values and CPU descriptor.

TMC680XOBadAccessFaultType represents a bad access type on a Motorola 68K processor.

TMC680XOAddressFaultType 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. TIPCStateFaultMessage 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 TMC680XOState 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 TMC680XOState, which is a concrete class that represents the 680x0 CPU state of a thread. Other examples include TMC680XOCPUState, which encapsulates the CPU state available for all 68K states, and TMC680XOCPUFaultState, 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 embodies 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 threads 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 example1(TThreadHandle& aThread)
{
    TRY
    {
        aThread.Kill( );        // terminates aThread immediatly
    }
    CATCH(TKernelException)
    (
        printf("Couldn't kill thread\n");
        // error occured trying to kill
    }
    ENDTRY;
    //. . .
}
CODE EXAMPLE 1
void TThreadHandle::Kill( )
{
    kern_return_t 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 suspend() method to be executed.

```
void example2(TTaskHandle& aTask)
{
    TRY
    {
        aTask.Suspend( );    // suspend all threads on task aTask
    }
    CATCH(TKernelException)
    (
        printf("Couldn't suspend threads\n"); // 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_info 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& theVersion)
{
    void host_kernel_version(fHostPort, theVersion);
}
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)
{
    unsigned long count;
    count = aReceiveRight.GetMakeSendCount( );
    // . . .
}
CODE EXAMPLE 9
unsigned long TPortReceiveRightHandle::GetMakeSendCount( )
{
    mach_port_status_t theInfo;                    // port status info
                                                      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**. An apparatus for enabling an object-oriented application, said application including object-oriented statements, to access in an object-oriented manner a procedural operating system by use of said object-oriented statements, said system providing services, including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

(b) a memory component in said computer;

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; and means for interfacing said object-oriented application to said procedural operating system utilizing said executable program logic;

(d) means, in said computer, for processing said object-oriented statements by executing methods from said object-oriented class library corresponding to said object-oriented statements; and

(e) means, in said object-oriented class library, including object-oriented thread classes, for enabling said object-oriented application to access said services to spawn, control, and obtain information relating to a thread of execution.

**2**. The apparatus of claim **1**, wherein said thread classes comprise a first object-oriented class encapsulating information necessary to create a new thread of execution, said first class being an abstract base class, and a second object-oriented class for enabling said application to spawn a new thread of execution on a task by passing a subclass of said first class to an instance of said second class, and for enabling said application to terminate, suspend, resume, and schedule an existing thread of execution, said second class having instances, said instances representing run-time processing entities in said computer.

**3**. The apparatus of claim **1**, wherein said object-oriented class library comprises object-oriented task classes for enabling said application to access in an object-oriented manner said services to reference and control a task, said task representing an execution environment for at least one thread of execution associated with said task.

**4**. The apparatus of claim **3**, wherein said task classes comprise a first task object-oriented class encapsulating attributes and operations of an existing task, said first task class including protected methods to enable run-time specific subclasses of said first task class to spawn new tasks.

**5**. The apparatus of claim **4**, wherein said first task class comprises methods for enabling said application to determine whether a possible task exists, to suspend and resume said existing task, to terminate said existing task, to receive predetermined information relating to said existing task, to identify a thread of execution contained in said existing task, and to perform predetermined virtual memory operations in an address space associated with said existing task.

**6**. The apparatus of claim **5**, wherein said object-oriented class library comprises an object-oriented thread class for enabling said application to create a new thread of execution, said task classes further comprising a second object-oriented task class, derived from said first task class, for enabling said application to spawn a new run-time specific task having a single thread of execution by passing an instance of said thread class to an instance of said second task class.

**7**. The apparatus of claim **1**, wherein said object-oriented class library comprises object-oriented synchronization classes for enabling said application to access in an object-oriented manner said services to synchronize execution of said thread of execution.

**8**. The apparatus of claim **7**, wherein said synchronization classes define counting semaphores for use in synchronizing the execution of said thread of execution, said synchronization classes comprising methods for enabling said application to acquire one or more of said counting semaphores in an exclusive mode or in a shared mode, and to release said counting semaphores after said counting semaphores are acquired.

**9**. The apparatus of claim **8**, wherein said counting semaphores are recoverable.

**10**. The apparatus of claim **7**, wherein said synchronization classes define a monitor lock for use in synchronizing said thread of execution, said synchronization classes comprising methods for enabling said application to acquire and release said monitor lock, and to block on a specified condition once said monitor lock is acquired, said applica-

tion releasing said monitor lock when said application blocks on said specified condition, and after said application is unblocked, due to satisfaction of said specified condition, said application reacquiring said monitor lock before resuming execution.

**11**. The apparatus of claim **10**, wherein said synchronization classes further comprise methods for enabling said application to perform a broadcast operation on a specified blocking condition when said application has acquired said monitor lock, said broadcast operation unblocking all of said threads of execution that are blocked on said specified blocking condition.

**12**. The apparatus of claim **7**, wherein said object-oriented class library comprises object-oriented scheduling classes for enabling said application to access in an object-oriented manner said services to schedule execution of said thread of execution.

**13**. The apparatus of claim **12** in which an actual scheduling priority, a default scheduling priority, and a maximum scheduling priority are associated with said application, said scheduling classes defining one or more scheduling priorities, said object-oriented class library including methods for setting each of said actual, default, and maximum scheduling priorities of said application to one of said scheduling priorities.

**14**. The apparatus of claim **13**, wherein said scheduling classes comprise an object-oriented class defining an idle scheduling priority adapted for use with said threads of execution that execute when said computer is substantially idle.

**15**. The apparatus of claim **13**, wherein said scheduling classes comprise an object-oriented class defining a responsive scheduling priority adapted for use with highly responsive threads of execution that execute for short time periods, and that block following said short time periods.

**16**. The apparatus of claim **13**, wherein said scheduling classes comprise an object-oriented class defining an interaction scheduling priority adapted for use with highly responsive threads of execution implementing an interface between a human operator and said computer.

**17**. The apparatus of claim **13**, wherein said scheduling classes comprise an object-oriented class defining a long-term scheduling priority adapted for use with threads of execution that execute for long periods of time.

**18**. The apparatus of claim **13**, wherein said scheduling classes comprise methods for enabling a task to specify a relative scheduling urgency of said task.

**19**. The apparatus of claim **1**, wherein said object-oriented class library comprises object-oriented fault classes for enabling said application to access in an object-oriented manner said services to process system and user-defined processor faults.

**20**. The apparatus of claim **19**, wherein said fault classes comprise a first object-oriented class defining a generic fault, said first class having virtual methods for setting a processor computer program logic and a fault computer program logic to thereby identify said generic fault, said first class representing an abstract base class.

**21**. The apparatus of claim **20**, wherein said fault classes comprise a second object-oriented class, derived from said first object-oriented class, comprising non-virtual methods for setting said processor computer program logic and said fault computer program logic in accordance with information specific to a particular fault of a particular processor such that said second class represents a processor-specific fault, said non-virtual methods of said second class overriding said virtual methods of said first class.

**22**. The apparatus of claim **21**, wherein said fault classes comprise an object-oriented class encapsulating information identifying a destination port, a fault message format, and fault types, said object-oriented class comprising methods for enabling said application to specify said destination port, said fault message format, and said fault types, and for enabling said application to instruct said operating system to send messages in said specified fault message format to said specified destination port when said specified fault types occur.

**23**. The apparatus of claim **19**, wherein said fault classes comprise a first object-oriented class comprising methods for obtaining and returning a processing state of said thread of execution.

**24**. The apparatus of claim **23**, wherein said fault classes comprise a second object-oriented class having methods for enabling said application to receive fault messages and to respond to received fault messages, said fault messages comprising information identifying a faulting task and said faulting task's faulting thread of execution, and/or information of a faulting thread of execution's state, said faulting thread of execution's state being obtained by calling said methods of said first class.

**25**. An apparatus for providing an object-oriented interface to a procedural operating system, said system providing services including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

a memory component in said computer;

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; means for interfacing to an object-oriented application; and said object-oriented class library defining object-oriented statements, said statements insertable into said application to enable said application to access said services during run-time execution of said application in said computer; and

(d) means, in said object-oriented class library, including object-oriented, thread classes for enabling said application to access in an object-oriented manner operating system services to spawn, control, and obtain information relating to a thread of execution.

**26**. The apparatus of claim **25**, wherein said object-oriented class library comprises object-oriented task classes for enabling said application to access in an object-oriented manner said services to reference and control a task, said task representing an execution environment for at least one thread of execution associated with said task.

**27**. The apparatus of claim **26**, wherein said object-oriented class library comprises object-oriented synchronization classes for enabling said application to access in an object-oriented manner said services to synchronize execution of threads of execution.

**28**. The apparatus of claim **27**, wherein said object-oriented class library comprises object-oriented, scheduling classes for enabling said application to access in an object-oriented manner said services to schedule execution of said thread of execution.

**29**. The apparatus of claim **28**, wherein said object-oriented class library comprises object-oriented, fault classes for enabling said application to access in an object-oriented manner said services to process system and user-defined processor faults.

**30**. An apparatus for providing an object-oriented interface to a procedural operating system, said system providing

services including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

(b) a memory component in said computer; and

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; means for interfacing to an object-oriented application; and said object-oriented class library comprising object-oriented thread classes, said thread classes comprising methods for accessing said services during run-time execution of said application in said computer, said thread classes having statements, said statements insertable into said application to enable said application to access said services to spawn, control, and obtain information relating to threads of execution.

**31**. The apparatus of claim **30**, wherein said thread classes comprise a first object-oriented class encapsulating information necessary to create a new thread of execution said first class being an abstract base class, and a second object-oriented class for enabling said application to spawn a new thread of execution on a task by passing a subclass of said first class to an instance of said second class, and for enabling said application to terminate, suspend, resume, and schedule an existing thread of execution, said second class having instances, said instances representing run-time processing entities in said computer.

**32**. An apparatus for providing an object-oriented interface to a procedural operating system, said system providing services including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

(b) a memory component in said computer; and

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; means for interfacing to an object-oriented application; and said object-oriented class library comprising object-oriented task classes, said task classes comprising methods for accessing said services during run-time execution of said application in said computer, said task classes having statements, said statements insertable into said application to enable said application to access said services to reference and control a task, said task representing an execution environment for at least one thread of execution associated with said task.

**33**. The apparatus of claim **32**, wherein said task classes comprise a first object-oriented class encapsulating attributes and operations of an existing task, said first class including protected method to enable run-time specific subclasses of said first class to spawn new tasks.

**34**. The apparatus of claim **33**, wherein said first class comprises methods for enabling said application to determine whether a possible task exists, to suspend and resume said existing task, to terminate said existing task, to receive predetermined information relating to said existing task, to identify a thread of execution contained in said existing task, and to perform predetermined virtual memory operations in an address space associated with said existing task.

**35**. The apparatus of claim **34**, wherein the object-oriented class library comprises an object-oriented thread class for enabling said application to create a new thread of execution, and said task classes further comprising a second object-oriented task class, derived from said first class, for enabling said application to spawn a new run-time specific task having a single thread of execution by passing an instance of said thread class to an instance of said second task class.

**36**. An apparatus for providing an object-oriented interface to a procedural operating system, said system providing services including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

(b) a memory component in said computer; and

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; means for interfacing to an object-oriented application; and said object-oriented class library comprising object-oriented synchronization classes, said synchronization classes comprising methods for accessing said services during run-time execution of said application in said computer, said synchronization classes having statements, said statements insertable into said application to enable said application to access said services to synchronize execution of threads of execution.

**37**. The apparatus of claim **36**, wherein said synchronization classes define counting semaphores for use in synchronizing the execution of said thread of execution, said synchronization classes comprising methods for enabling said application to acquire one or more of said counting semaphores in an exclusive mode or in a shared mode, and to release said counting semaphores after said counting semaphores are acquired.

**38**. The apparatus of claim **37**, wherein said counting semaphores are recoverable.

**39**. The apparatus of claim **37**, wherein said synchronization classes define a monitor lock for use in synchronizing said thread of execution, said synchronization classes comprising methods for enabling said application to acquire and release said monitor lock, and to block on a specified condition once said monitor lock is acquired, said application releasing said monitor lock when said application blocks on said specified condition, and after said application is unblocked, due to satisfaction of said specified condition, said application reacquiring said monitor lock before resuming execution.

**40**. The apparatus of claim **39**, wherein said synchronization classes further comprise methods for enabling said application to perform a broadcast operation on a specified blocking condition when said application has acquired said monitor lock, said broadcast operation unblocking all of said threads of execution that are blocked on said specified blocking condition.

**41**. An apparatus for providing an object-oriented interface to a procedural operating system, said system providing services including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

(b) a memory component in said computer; and

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; means for interfacing to an object-oriented application; and said object-oriented class library comprising object-oriented, scheduling classes, said scheduling classes comprising methods for accessing said services during run-time execution of said application in said computer,

said scheduling classes having statements, said statements insertable into said application to enable said application to access said services.

**42**. The apparatus of claim **41** in which an actual scheduling priority, a default scheduling priority, and a maximum scheduling priority are associated with said application, said scheduling classes defining one or more scheduling priorities, said object-oriented class library including methods for setting each of said actual, default, and maximum scheduling priorities of said application to one of said scheduling priorities.

**43**. The apparatus of claim **42**, wherein said scheduling classes comprise an object-oriented class defining an idle scheduling priority adapted for use with said threads of execution that execute when said computer is substantially idle.

**44**. The apparatus of claim **43**, wherein said scheduling classes comprise an object-oriented class defining a responsive scheduling priority adapted for use with highly responsive threads of execution that execute for short time periods, and that block following said short time periods.

**45**. The apparatus of claim **44**, wherein said scheduling classes comprise an object-oriented class defining an interaction scheduling priority adapted for use with highly responsive threads of execution implementing an interface between a human operator and said computer.

**46**. The apparatus of claim **44**, wherein said scheduling classes comprise an object-oriented class defining a long-term scheduling priority adapted for use with threads of execution that execute for long periods of time.

**47**. (Amended) The apparatus of claim **44**, wherein said scheduling classes comprise methods for enabling a task to specify a relative scheduling urgency of said task.

**48**. An apparatus for providing an object-oriented interface to a procedural operating system, said system providing services including procedural functions saved as executable program logic that are called to access said services, said apparatus comprising:

(a) a computer;

(b) a memory component in said computer; and

(c) a code library, stored in said memory component, comprising means for storing said executable program logic in an object-oriented class library; means for interfacing to an object-oriented application; and said object-oriented class library comprising object-oriented fault classes, said fault classes comprising methods for accessing said services during run-time execution of said application in said computer, said fault classes having statements, said statements insertable into said application to enable said application to access said services to process system and user-defined processor faults.

**49**. The apparatus of claim **48**, wherein said fault classes comprise a first object-oriented class defining a generic fault, said first class having virtual methods for setting a processor computer program logic and a fault computer program logic to thereby identify said generic fault, said first class representing an abstract base class.

**50**. The apparatus of claim **49**, wherein said fault classes comprise a second object-oriented class, derived from said first object-oriented class, comprising non-virtual methods for setting said processor computer program logic and said fault computer program logic in accordance with information specific to a particular fault of a particular processor such that said second class represents a processor-specific fault, said non-virtual methods of said second class override said virtual methods of said first class.

**51**. The apparatus of claim **49**, wherein said fault classes comprise an object-oriented class encapsulating information identifying a destination port, a fault message format, and fault types, said object-oriented class comprising methods for enabling said application to specify said destination port, said fault message format, and said fault types, and for enabling said application to instruct said operating system to send messages in said specified fault message format to the specified destination port when said specified fault types occur.

**52**. The apparatus of claim **49**, wherein said fault classes comprise a first object-oriented class comprising methods for obtaining and returning a processing state of said thread of execution.

**53**. The apparatus of claim **52**, wherein said fault classes comprise a second object-oriented class having methods for enabling said application to receive fault messages and to respond to received fault messages, said fault messages comprising information identifying a faulting task and said faulting task's faulting thread of execution, and/or information of a faulting thread of execution's state, said faulting thread of execution'state being obtained by calling said methods of said first class.

* * * * *

# EXHIBIT 3

US005566337A

# United States Patent [19]

Szymanski et al.

[11] Patent Number: 5,566,337

[45] Date of Patent: Oct. 15, 1996

[54] **METHOD AND APPARATUS FOR DISTRIBUTING EVENTS IN AN OPERATING SYSTEM**

[75] Inventors: **Steven J. Szymanski**, Cupertino; **Thomas E. Saulpaugh**, San Jose; **William J. Keenan**, Redwood City, all of Calif.

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

[21] Appl. No.: **242,204**

[22] Filed: **May 13, 1994**

[51] **Int. Cl.**[6] ..... **G06F 9/00**

[52] **U.S. Cl.** ..... **395/733**; 395/650; 395/700

[58] **Field of Search** ..... 395/650, 725, 395/700, 775

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,155,842 | 10/1992 | Rubin | 395/182.2 |
| 5,237,684 | 8/1993 | Record et al. | 395/650 |
| 5,291,608 | 3/1994 | Flurry | 395/725 |
| 5,305,454 | 4/1994 | Record et al. | 395/650 |
| 5,321,837 | 6/1994 | Daniel et al. | 395/650 |
| 5,355,484 | 10/1994 | Record et al. | 395/650 |
| 5,430,875 | 7/1995 | Ma | 395/650 |

FOREIGN PATENT DOCUMENTS

| | | |
|---|---|---|
| 0528222 | 2/1993 | European Pat. Off. . |
| WO91/03017 | 3/1991 | WIPO . |

OTHER PUBLICATIONS

IBM: 'OS/2 2.0 Presentation Manager Programming Guide', Mar. 1992, QUE, USA, p. 31–5, last paragrph, p. 31–6, paragraph 3.

*Primary Examiner*—Jack B. Harvey
*Assistant Examiner*—Sumati Lefkowitz
*Attorney, Agent, or Firm*—Burns, Doane, Swecker & Mathis

[57] **ABSTRACT**

In a computer including an operating system, an event producer for generating an event and detecting that an event has occurred in the computer and an event consumer which need to be informed when events occur in the computer, a system for distributing events including a store for storing a specific set of events of which the at least one event consumer is to be informed, an event manager control unit for receiving the event from the event producer, comparing the received event to the stored set of events, and distributing an appropriate event to an appropriate event consumer, and a distributor for receiving the event from the control unit and directing the control unit to distribute an appropriate event to an appropriate event consumer.

**24 Claims, 10 Drawing Sheets**

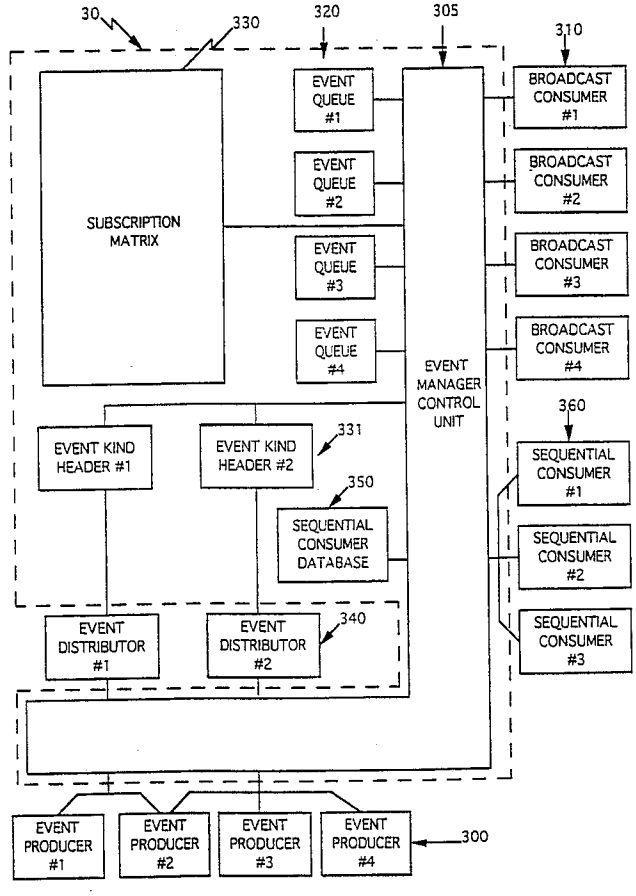
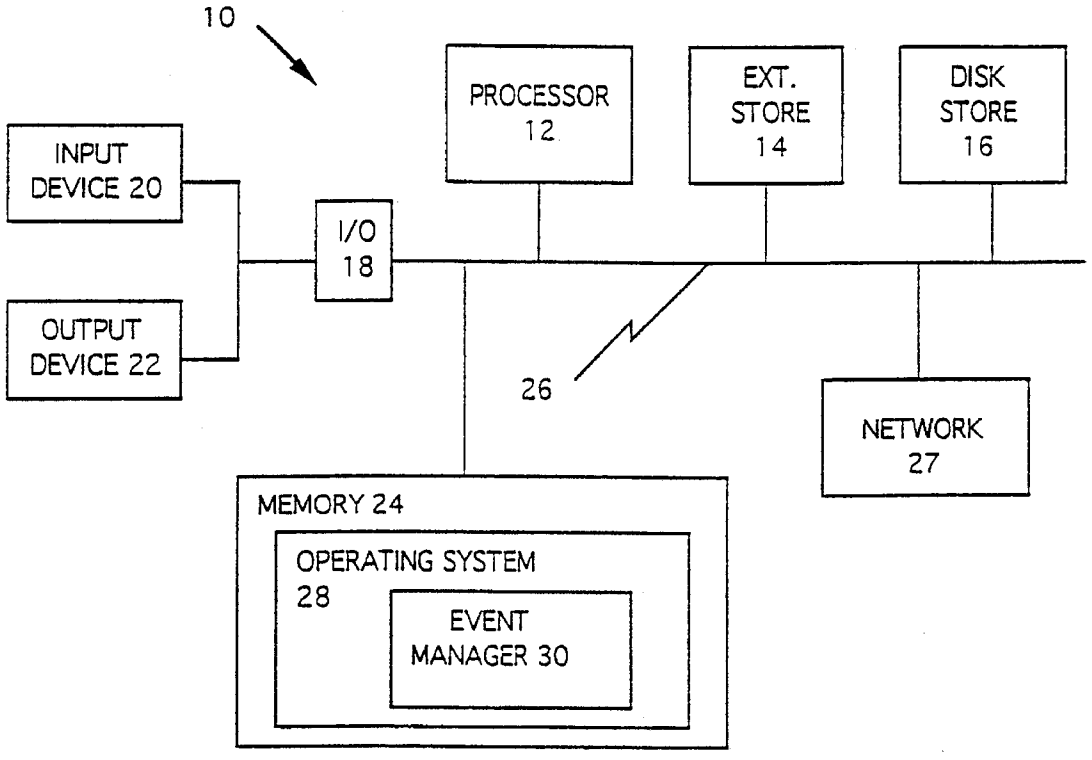


FIG. 1


10
PROCESSOR 12
EXT. STORE 14
DISK STORE 16
INPUT DEVICE 20
I/O 18
OUTPUT DEVICE 22
26
NETWORK 27
MEMORY 24
OPERATING SYSTEM 28
EVENT MANAGER 30

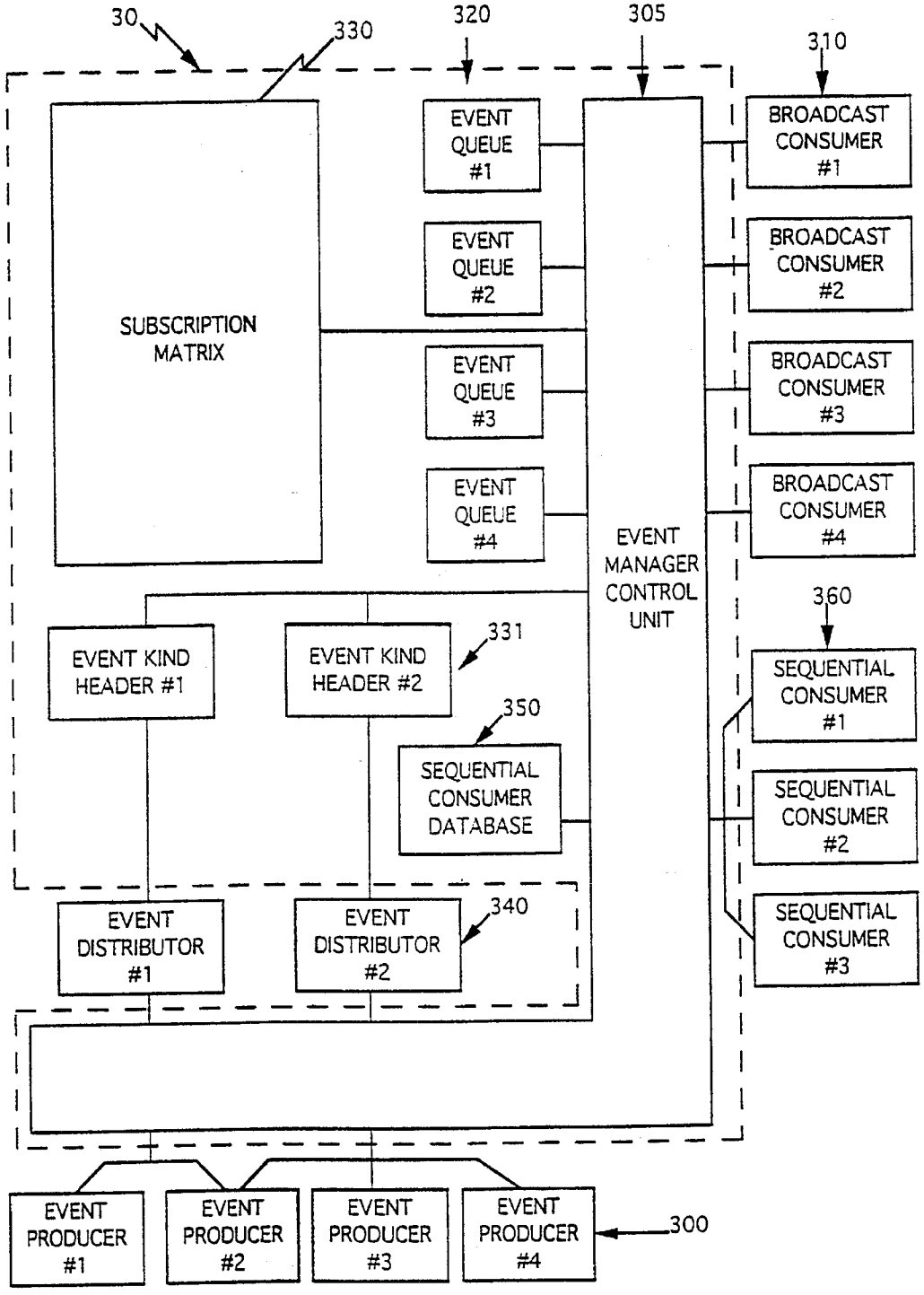

30
330
320
305
310
EVENT QUEUE #1
EVENT QUEUE #2
EVENT QUEUE #3
EVENT QUEUE #4
BROADCAST CONSUMER #1
BROADCAST CONSUMER #2
BROADCAST CONSUMER #3
BROADCAST CONSUMER #4
SUBSCRIPTION MATRIX
EVENT MANAGER CONTROL UNIT
360
SEQUENTIAL CONSUMER #1
SEQUENTIAL CONSUMER #2
SEQUENTIAL CONSUMER #3
EVENT KIND HEADER #1
EVENT KIND HEADER #2
331
350
SEQUENTIAL CONSUMER DATABASE
EVENT DISTRIBUTOR #1
EVENT DISTRIBUTOR #2
340
EVENT PRODUCER #1
EVENT PRODUCER #2
EVENT PRODUCER #3
EVENT PRODUCER #4
300

FIG. 2

FIG. 3

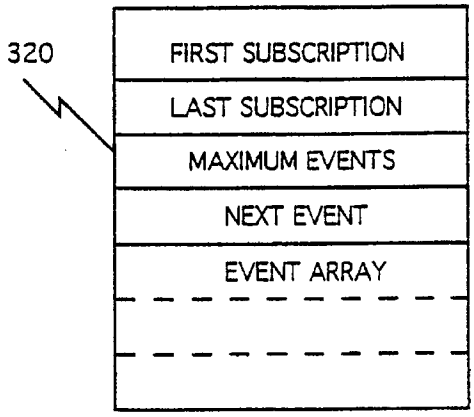



FIG. 6

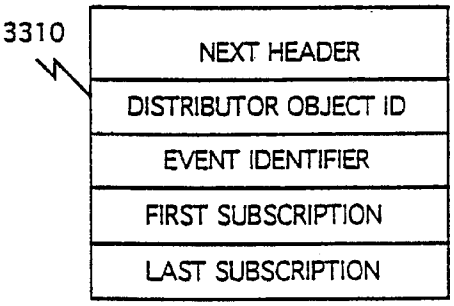



FIG. 5a

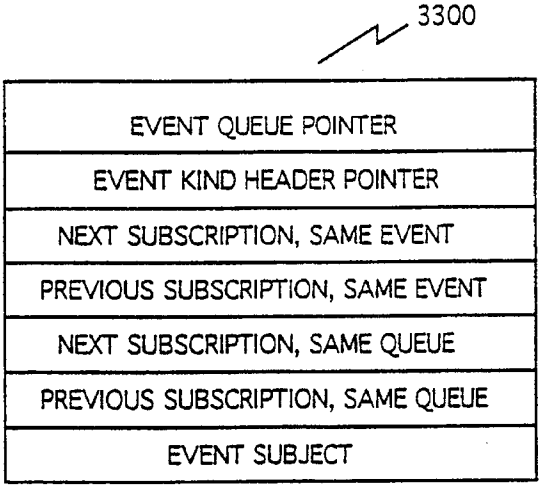



FIG. 7

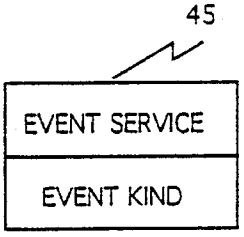



FIG. 8

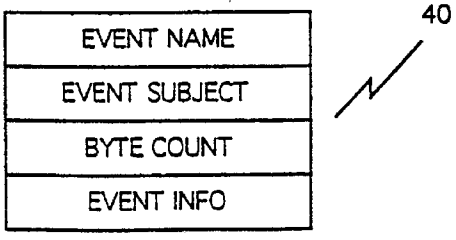
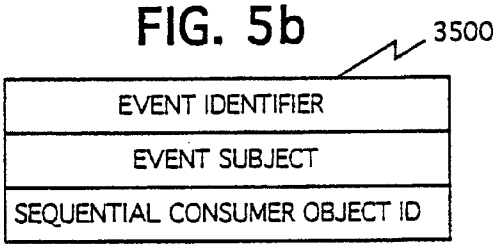



FIG. 5b



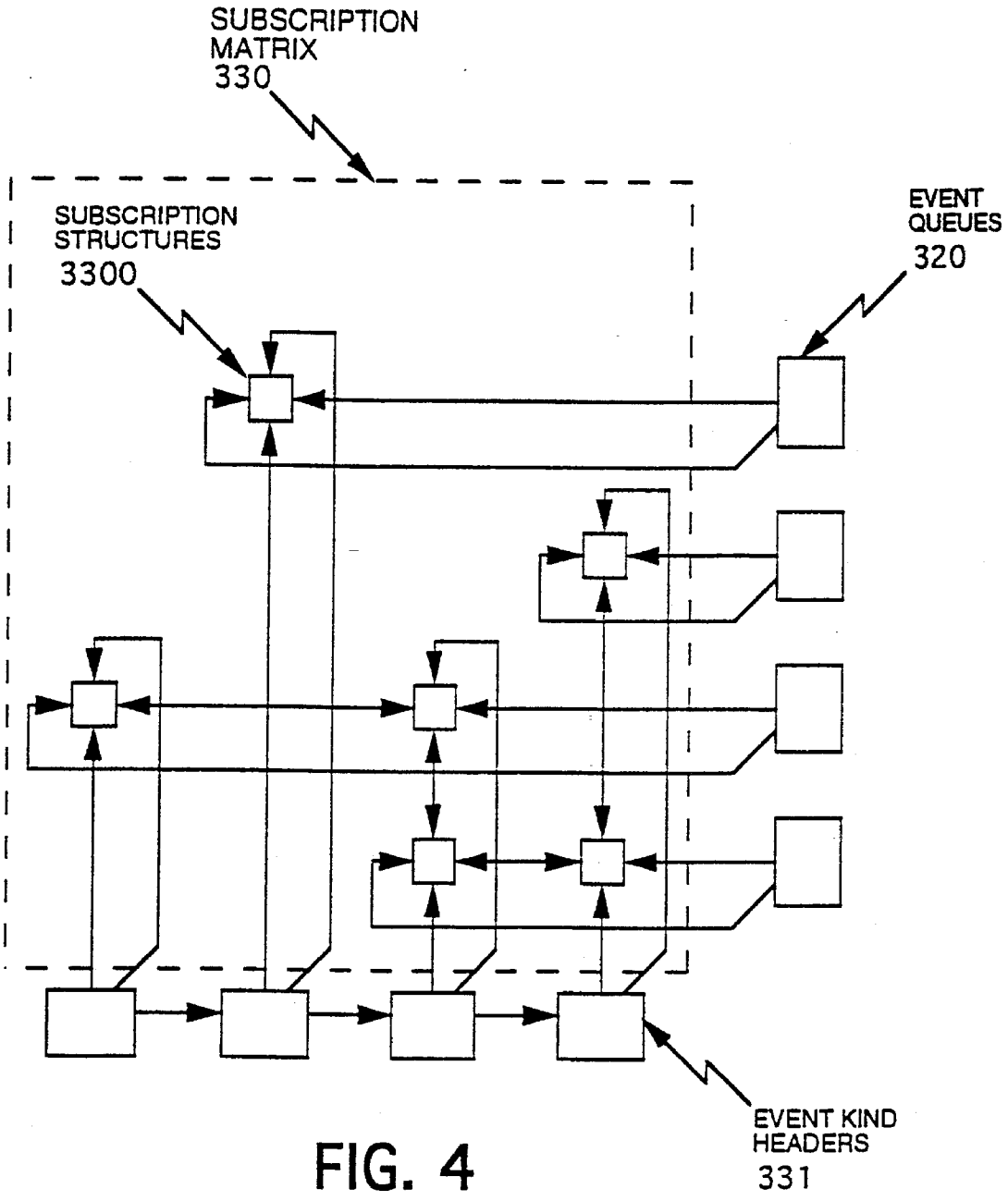

SUBSCRIPTION
MATRIX
330
SUBSCRIPTION
STRUCTURES
3300
EVENT
QUEUES
320
EVENT KIND
HEADERS
331

FIG. 4

FIG. 9A

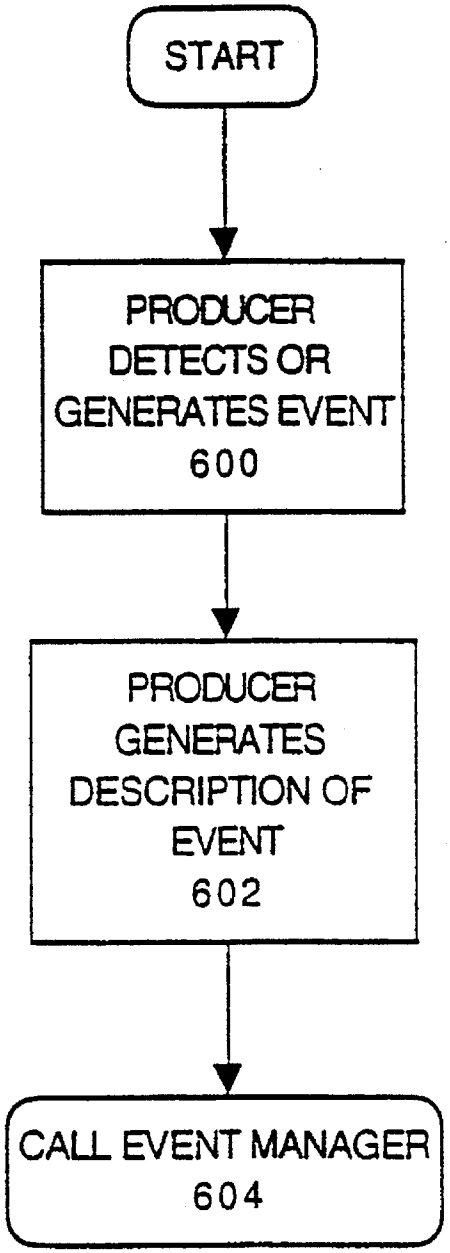

START
PRODUCER DETECTS OR GENERATES EVENT 600
PRODUCER GENERATES DESCRIPTION OF EVENT 602
CALL EVENT MANAGER 604

FIG. 9B

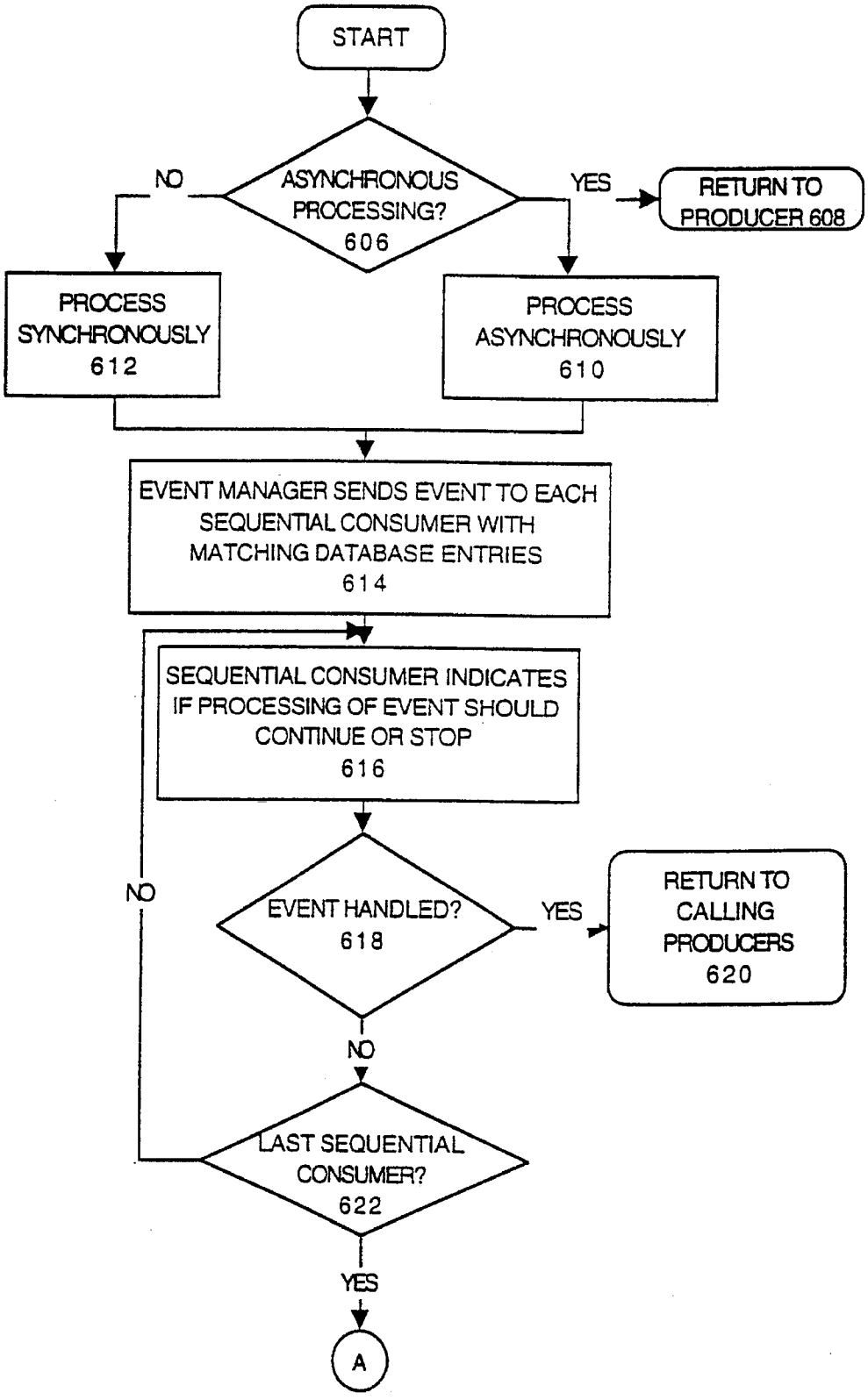

START
ASYNCHRONOUS PROCESSING? 606
NO
YES
RETURN TO PRODUCER 608
PROCESS SYNCHRONOUSLY 612
PROCESS ASYNCHRONOUSLY 610
EVENT MANAGER SENDS EVENT TO EACH SEQUENTIAL CONSUMER WITH MATCHING DATABASE ENTRIES 614
SEQUENTIAL CONSUMER INDICATES IF PROCESSING OF EVENT SHOULD CONTINUE OR STOP 616
EVENT HANDLED? 618
NO
YES
RETURN TO CALLING PRODUCERS 620
LAST SEQUENTIAL CONSUMER? 622
NO
YES
A

FIG. 9C

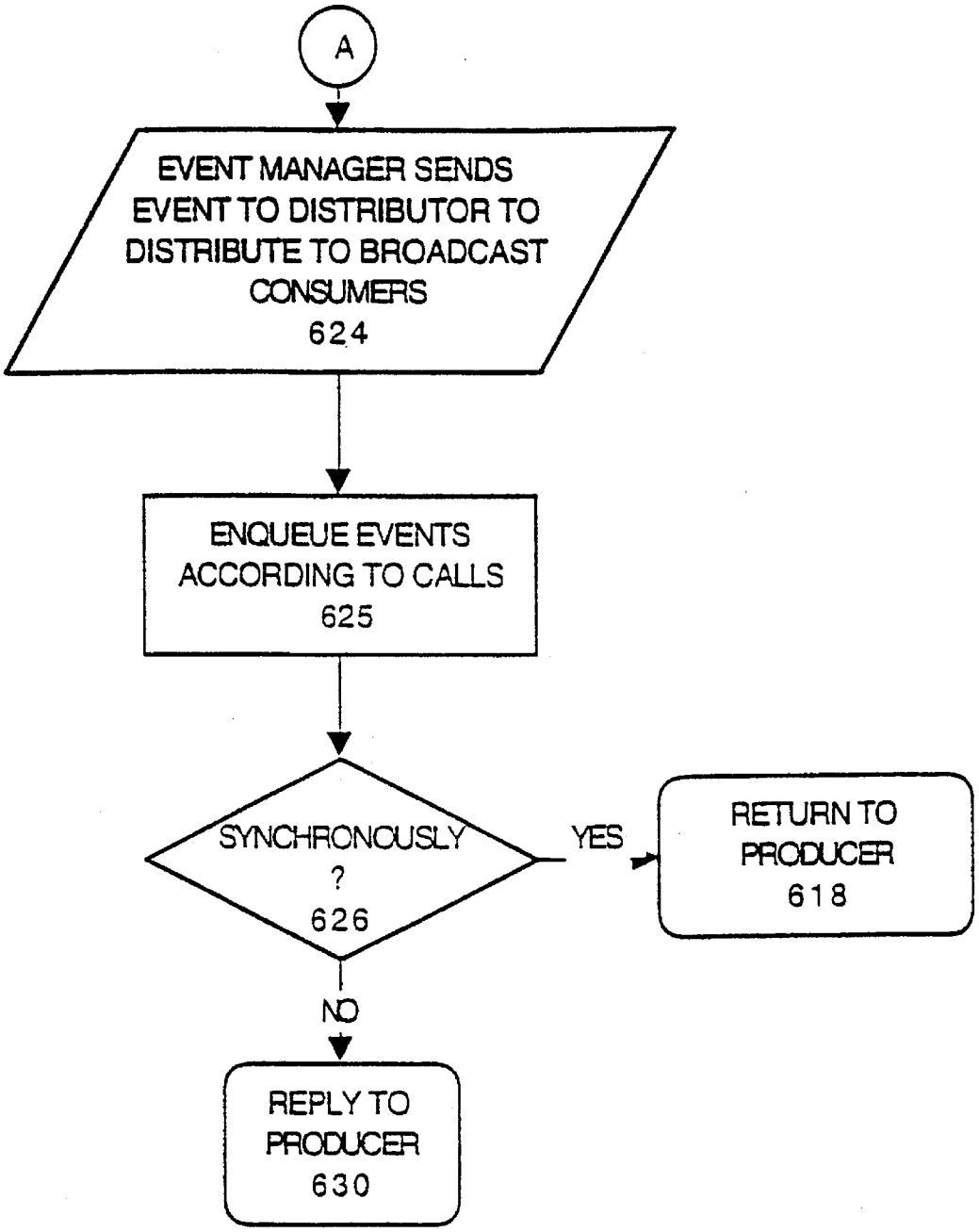

A
EVENT MANAGER SENDS EVENT TO DISTRIBUTOR TO DISTRIBUTE TO BROADCAST CONSUMERS
624
ENQUEUE EVENTS ACCORDING TO CALLS
625
SYNCHRONOUSLY ?
626
YES
RETURN TO PRODUCER
618
NO
REPLY TO PRODUCER
630

FIG. 9D

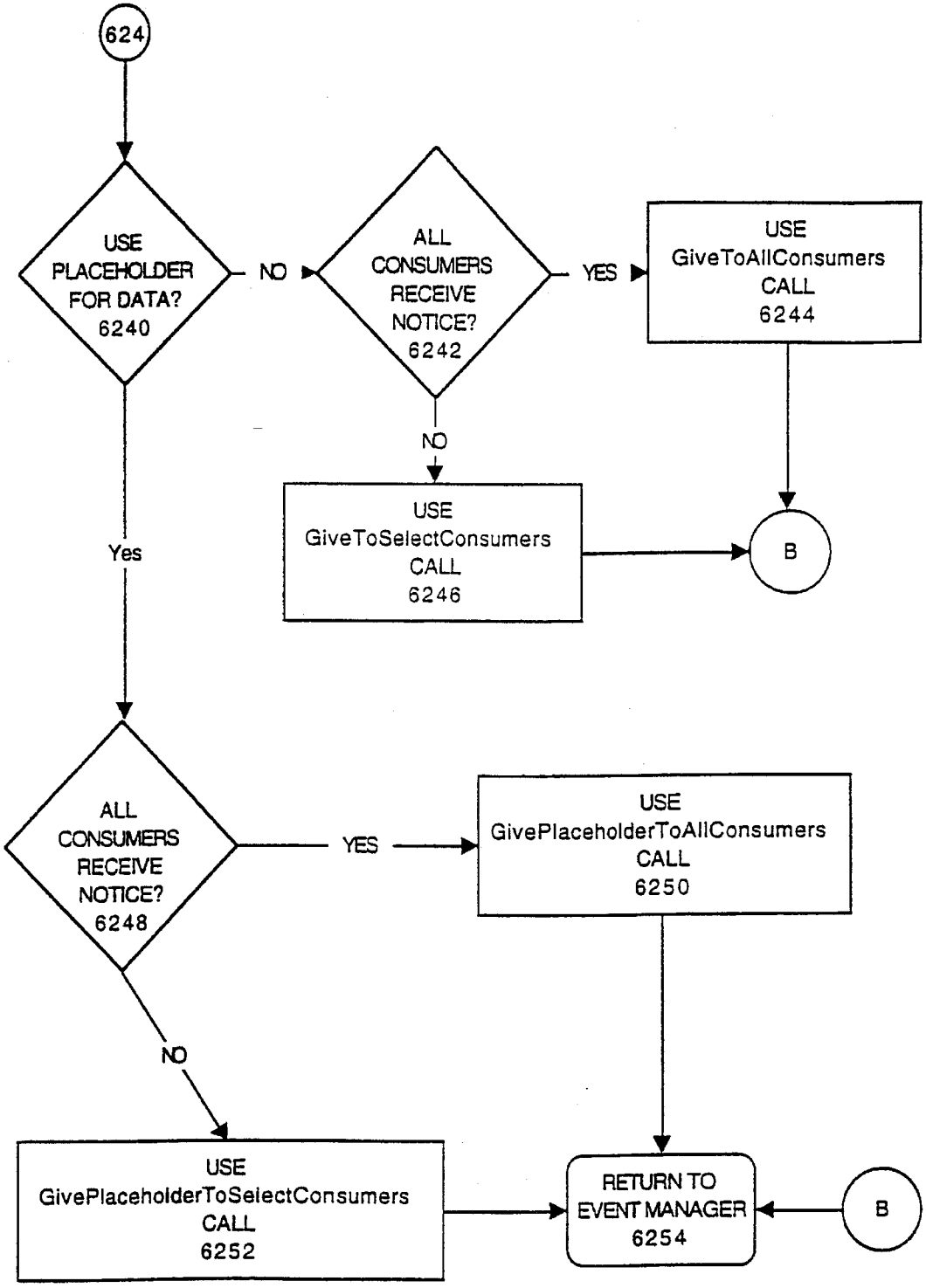

624
USE PLACEHOLDER FOR DATA? 6240
NO
ALL CONSUMERS RECEIVE NOTICE? 6242
YES
USE GiveToAllConsumers CALL 6244
NO
USE GiveToSelectConsumers CALL 6246
B
Yes
ALL CONSUMERS RECEIVE NOTICE? 6248
YES
USE GivePlaceholderToAllConsumers CALL 6250
NO
USE GivePlaceholderToSelectConsumers CALL 6252
RETURN TO EVENT MANAGER 6254
B

FIG. 10

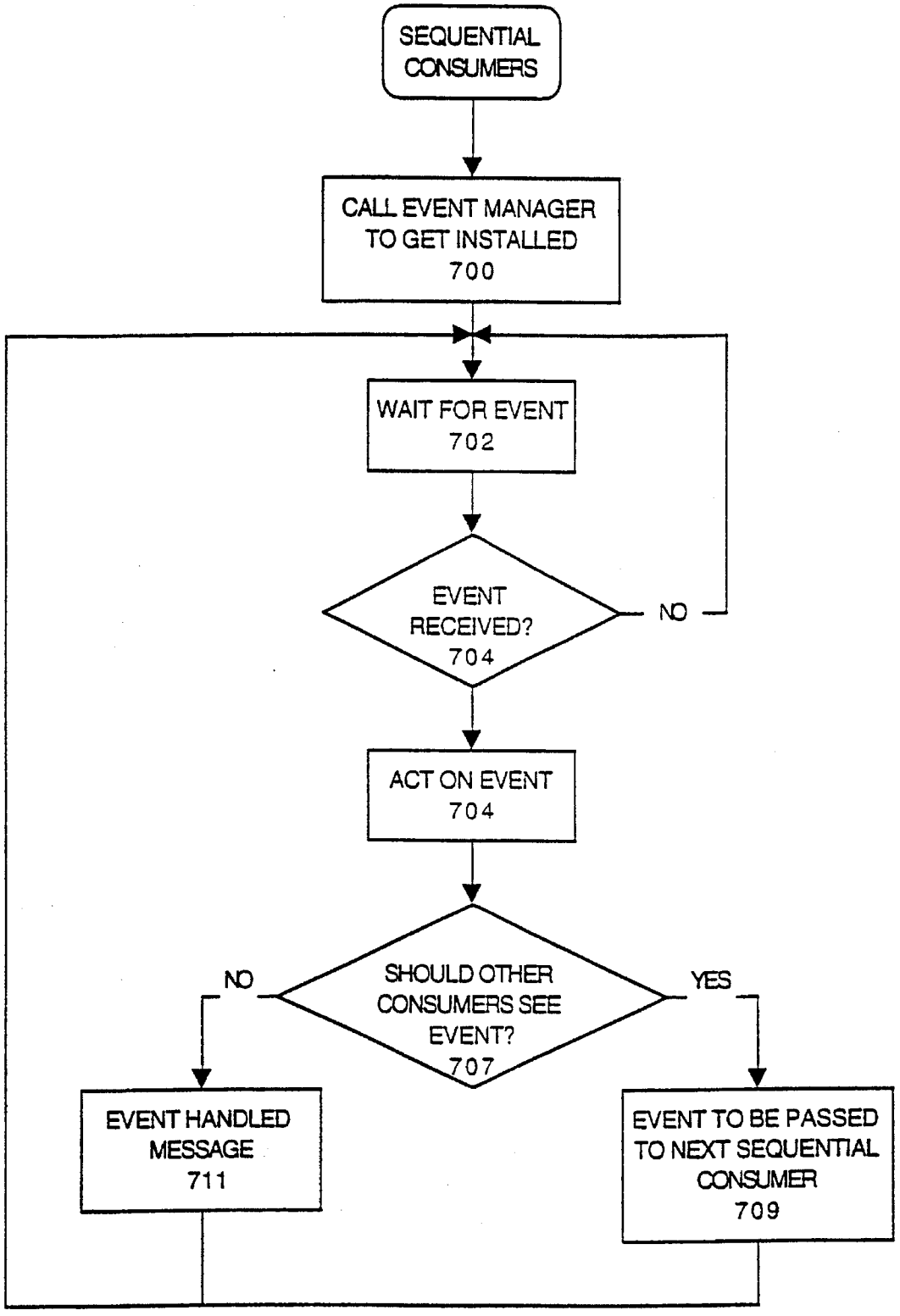
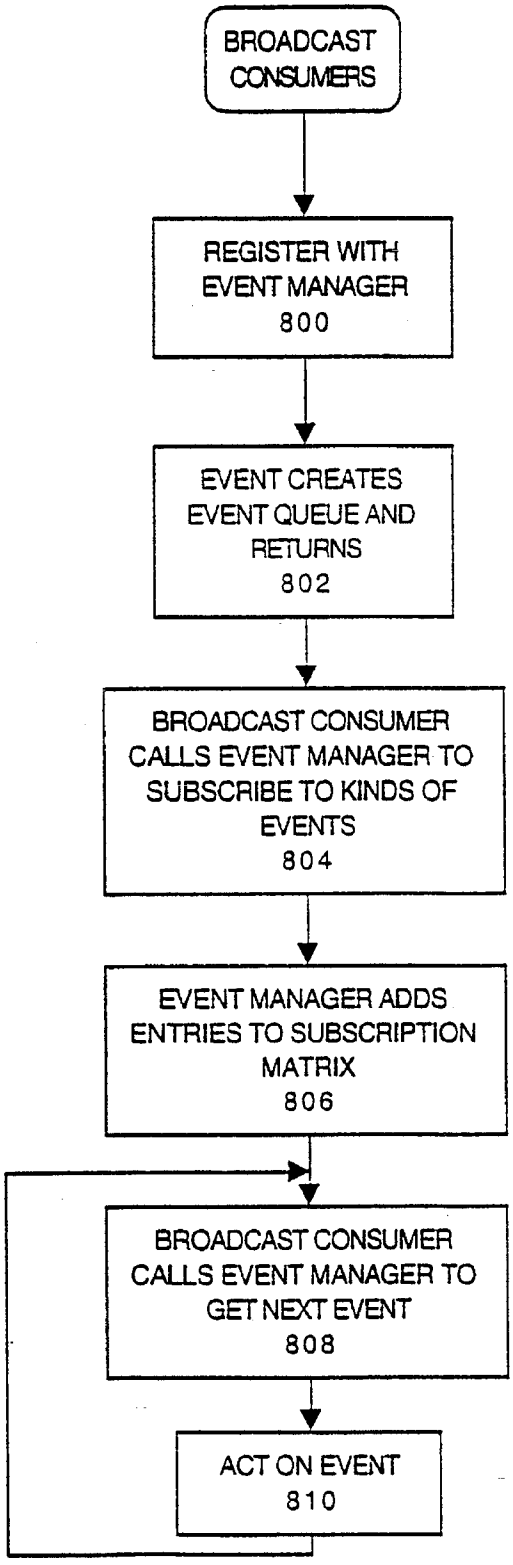

SEQUENTIAL CONSUMERS
CALL EVENT MANAGER TO GET INSTALLED 700
WAIT FOR EVENT 702
EVENT RECEIVED? 704
NO
ACT ON EVENT 704
SHOULD OTHER CONSUMERS SEE EVENT? 707
NO
YES
EVENT HANDLED MESSAGE 711
EVENT TO BE PASSED TO NEXT SEQUENTIAL CONSUMER 709

FIG. 11


BROADCAST CONSUMERS
REGISTER WITH EVENT MANAGER 800
EVENT CREATES EVENT QUEUE AND RETURNS 802
BROADCAST CONSUMER CALLS EVENT MANAGER TO SUBSCRIBE TO KINDS OF EVENTS 804
EVENT MANAGER ADDS ENTRIES TO SUBSCRIPTION MATRIX 806
BROADCAST CONSUMER CALLS EVENT MANAGER TO GET NEXT EVENT 808
ACT ON EVENT 810

# METHOD AND APPARATUS FOR DISTRIBUTING EVENTS IN AN OPERATING SYSTEM

## CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is related to a patent application No. 08/245,141 entitled "Method and Apparatus for Handling Requests Regarding Information Stored in A File System", in the name of Steven James Szymanski and Bill Monroe Bruffey, filed on May 13, 1994, herein incorporated by reference.

## BACKGROUND

The present invention is directed to a method and apparatus for distributing information about events occurring in a computer, and in particular an event manager which manages the distribution of those events to the appropriate entities within the computer.

For purposes of this description, an event is any occurrence in a computer of which software programs running on that computer or on a connected computer might need to be informed. Events may include occurrences such as, for example, a keystroke, a mouse click, disk insertion and ejection, network connection and disconnection, the computer entering a "sleep mode" shutdown, a window uncovered (i.e., the contents of the window need to be redisplayed), a new file created, a directory renamed, the contents of file changed, and the tree space on a volume changed, etc.

Interrupts and error conditions may also be counted as atypical examples of events. In particular, interrupts need to be handled by a program so an event manager is an inappropriate solution. However, the code which does handle the interrupt might generate an event based on the interpretation of the interrupt. For example, the computer might generate an interrupt when the user inserts a floppy disk. The interrupt itself is unlikely to be propagated by the event manager, but it would be reasonable for the interrupt handler to produce a "disk inserted" event. Error conditions are similar. Most of the time it is necessary for one of the computer programs on the system to handle the error, therefore more direct point to point mechanisms are appropriate. However, there are kinds of errors which are more advisory in nature which would be appropriate to be sent via events. For example, some portable computers take various actions to reduce power consumption when the battery gets low. It would be appropriate to produce an event called "battery low" to inform all software programs of the condition, and have all of the software which can reduce power consumption consume these events.

Currently, known operating systems all have some type of mechanism for managing the events that occur within the computer. However, these mechanisms use a point-to-point method of managing the events. That is, the entities producing or detecting events distribute the events to the entities using the events. To accomplish this, all of the entities producing or detecting events must know which entities they must notify when a particular event is generated within the computer. This configuration is very cumbersome and inefficient. Further, it is resource intensive since all entities producing or detecting events must have information on all the events they produce or detect and also on all the entities interested in those events. This information is both extensive and constantly changing, causing modifications to be difficult.



Further, point-to-point mechanisms lack flexibility. Under point-to-point schemes, if there is a new consumer of an event, a new version of the producer must be released which knows about the new consumer. Or if a new kind of event becomes necessary, a new version of the event manager must be released which knows how to distribute the new kind of event.

It is desirable to provide an apparatus for efficiently dealing with all kinds of events in an operating system and for distributing information regarding specific kinds of events to programs which require such information. To this end, it is also desirable to improve the system performance and reduce the resources required to distribute such information. To meet these goals, it is desirable to provide an apparatus for managing events in which communication between the event producers and consumers is facilitated without requiring each event producer to be aware of all of the event consumers.

## BRIEF STATEMENT OF THE INVENTION

In accordance with the present invention, the foregoing objectives, as well as others, are achieved through centralization of event management, and in particular, by providing an event manager for handling the distribution of events within the computer.

According to one embodiment, in a computer including at least one event producer for detecting that an event has occurred in the computer and generating an event and at least one event consumer which need to be informed when events occur in the computer, a system is provided for distributing information about events. The system includes storing means for storing a specific set of events of which the event consumers are to be informed, an event manager control means for receiving the event from the event producer, comparing the received event to the stored set of events, and distributing an appropriate event to an appropriate event consumer, and a distributor for receiving the event from the control means and directing the control means to distribute an appropriate event to an appropriate event consumer.

According to another embodiment, a system is provided for distributing events occurring in a computer. The system comprises event producers for detecting that an event has occurred in the computer, generating an event, and generating a description of the event and event consumers which need to be informed when events occur in the computer, the event consumers comprising a first and a second class of consumers. The system further comprises storing means for storing a specific set of events of which the event consumers am to be informed, event manager control means for receiving the event from the event producers and comparing the received event to the stored set of events, distributor means, responsive to the event control means, for deciding if an event should be passed to an event consumer. The event manager control means comprises first means for sending an event to appropriate event consumers of a first type in accordance with the stored set of events, and second means for sending the event to appropriate event consumers of a second type responsive to the distributor means.

According to another embodiment, a method is provided for distributing events occurring in a computer. The method comprises the steps of determining that an event has been detected by an event producer in the computer, storing, in a storing means, a specific set of events of which an event consumer is to be informed, receiving the event in an event

control means from the event producer and comparing the received event to the stored set of events. The method further comprises receiving the event in a distributor means from the control means, directing the control means to distribute an appropriate event to an appropriate event consumer, and distributing, via the control means, an appropriate event to an appropriate event consumer.

Still other objects, features and attendant advantages of the present invention will become apparent to those skilled in the art from a reading of the following detailed description of the embodiments constructed in accordance therewith, taken in conjunction with the accompanying drawings.

## BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will now be described in more detail with reference to preferred embodiments of the method and apparatus, given only by way of example, and with reference to the accompanying drawings, in which:

FIG. **1** is a block diagram of an exemplary computer on which the present invention can be implemented;

FIG. **2** is a block diagram of the architecture for the event manager according to one embodiment of the present invention;

FIG. **3** is an exemplary embodiment of the event queue data structure according to the present invention;

FIG. **4** is an exemplary embodiment of the subscription matrix according to the present invention;

FIG. **5***a* is an exemplary embodiment of the subscription data structure according to the present invention;

FIG. **5***b* is an exemplary embodiment of the sequential consumer entry structure according to the present invention;

FIG. **6** is an exemplary embodiment of the event kind header data structure according to the present invention;

FIG. **7** is an exemplary embodiment of the event name data structure according to the present invention;

FIG. **8** is an exemplary embodiment of the event data structure according to the present invention;

FIGS. **9**A, **9**B, **9**C and **9**D are flowchart illustrating the event handling process from the event producers' and the event distributors' point of view according to one embodiment of the present invention;

FIG. **10** is a flowchart illustrating the event handling process from the sequential consumers' point of view according to one embodiment of the present invention; and

FIG. **11** is a flowchart illustrating the event handling process from the broadcast consumers' point of view according to one embodiment of the present invention.

## DETAILED DESCRIPTION

In general, the invention recognizes the need for efficient communications between different entities within the computer concerning events occurring within the computer. In particular, communications are required to inform entities within the computer about the events produced by other entities. The method required to handle these communications is complicated by the fact that the entities involved do not know the identity of the other entities. The method is further complicated by the fact that the identity of the entities needing to know about events and the lists of events which can occur are subject to constant change.

One goal for the event manager according to the present invention is to provide a common service which supports a majority of these kinds of communications. The information which needs to be communicated is referred to as the event. According to one embodiment, the event is described by three pans, an event identifier which indicates the kind of event, an event subject, which identifies the entity which the event happened to, and event information which describes how the event occurred. The entities within the computer which are the sources of the information are referred to as the producers of the event. In particular, an event producer is any software on a computer that is responsible for generating an event or for detecting that the computer hardware has generated an event. The event producer then generates a description for each event it produces or detects. The entities within the computer which need to receive the information are referred to as consumers of the event. In particular, an event consumer is any program that needs to be informed when an event has occurred and needs to be informed of the description of the event. Any intermediate service which moderates the connection between the producers and consumers of an event is referred to as the distributor of the event.

According to one embodiment of the present invention, there are two classes of event consumers which differ in their relationships to other consumers of an event, namely broadcast consumers and sequential consumers. Broadcast consumers have no relationship with other consumers. They do not need to know if other consumers exist, nor in what order consumers arc informed of the event, as long as they themselves are eventually informed. Sequential consumers, on the other hand, have very definite relationships with other consumers. They require that no other consumer be told about an event while they themselves are still processing it, and they require the ability to influence when in the sequence they receive the event. In addition, many sequential consumers require the ability to modify the event itself, and even block an event from being received by other consumers. Consumers are defined as sequential or broadcast based on whether they have to react to an event without fail. In particular, a sequential consumer must react to the kinds of events in which it is interested and so a distributor should not withhold those events from the sequential consumer. For example, a communications program would be a sequential consumer of events which notify of the dropping of a connection, since the communication program would need to respond to such an event.

Broadcast consumers typically have a set of kinds of events in which they are interested and want to be notified of the next event from this set as simply as possible. Many broadcast consumers are interested in events that occur only to a limited set of subjects, and so one embodiment of the present invention provides a method for filtering events based on "who" they involve. In addition, the set of events in which the broadcast consumers are interested changes over time as does the immediacy of the interest. Thus, one embodiment of the present invention provides a method which allows the consumer to modify the set of events which will be delivered to the consumer. Lastly, there is no clear pattern as to whether broadcast consumers want to poll the event manager control unit to collect an event, or if they want to be notified asynchronously. Therefore, according to one embodiment, both options are supported.

FIG. **1** is a block diagram showing an exemplary computer on which the software according to the present invention can be implemented. The computer **10** includes a processor (CPU) **12**, an external storage device **14**, a disk storage device **16**, an input/output (I/O) controller **18**, an input device **20**, an output device **22**, a memory **24**, and a common system bus **26** connecting each of the above

elements. Only one input device and one output device is shown in FIG. **1** for ease of readability purposes. However, it will be appreciated that the computer **10** can include more than one such device. The processor **12**, the external storage device **14**, the disk storage device **16** and the memory **24** are also connected through the bus **26** and the I/O controller **18** to the input and output devices **20** and **22**. In addition, the computer **10** can also be adapted for communicating with a network **27**.

Stored within the memory **24** of the computer **10** are a number of pieces of software which can be executed by the processor **12**. One of those pieces of software is an operating system **28**. In one embodiment, the operating system **28** is a microkernel operating system capable of maintaining multiple address spaces. An event manager **30** resides within the operating system. In an exemplary embodiment, the computer system **10** of the present invention is an Apple Macintosh™ computer system made by Apple Computer, Inc., of Cupertino, Calif., and having a microprocessor and a memory wherein a microkernel operating system **28** that includes the event manager **30** resides. The components of the computer system can be changed within the skill of the ordinary artisan once in possession of the instant disclosure. Although the present invention is described in a Macintosh™ environment, it is within the scope of the invention, and within the skill of the ordinarily skilled artisan, to implement the invention in a DOS, Unix, or other computer environment.

The present invention relates to an architecture for an event manager for managing events that occur in an operating system. Clients of the event manager include event consumers and event producers, that is, applications programs and the various parts of the operating system, such as for example, a file manager.

According to one embodiment, the present invention cooperates with an operating system with a microkernel architecture in which the kernel provides a semaphore synchronization mechanism and a messaging system. The messaging system creates and maintains a set of message objects and one or more port objects. The messaging system has a number of features. It allows a creator-defined value to be associated with each object. The messaging system also allows multiple objects to be mapped to the same port and messages to be either received from a port or have a function be called when a message of an appropriate type is sent to that port. Further, the messaging system allows the receiver of the message to determine the object to which the message was originally sent and to derive the creator-defined value for that object. One example of such a microkernel architecture is provided by NuKERNEL™, used in Apple Macintosh™ computers. The NuKERNEL™ system is described in copending U.S. patent application Ser. No. 08/128,706, filed on Sep. 30, 1993, for a "System For Decentralizing Backing Store Control Of Virtual Memory In A Computer", application Ser. No. 08/220,043, filed on Mar. 30, 1994, for an "Object Oriented Message Passing System And Method", and an application entitled "System and Method Of Object Oriented Message Filtering", filed in the name of Thomas E. Saulpaugh and Steven J. Szymanski, on or about the date of filing of the present application. These three patent applications are incorporated by reference herein.

According to the messaging system, a service is provided by a server, and software wishing to make use of the service is called a client of the server. Messages are sent to a target by clients and received by servers. The message directs the server to perform a function on behalf of the client. Message objects are abstract entities that represent various resources to messaging system clients. These objects may represent, for example, devices, files, or windows managed by a server. Clients send messages to objects, which objects are identified by an identification labelled ObjectID.

Message ports are abstract entities that represent a service. These ports may represent, for example, a device driver, a file system, or a window manager. Servers receive messages from ports, which ports are identified by a label PortID. Objects are assigned to a port. A message sent to an object is received at that object's port. Ports and objects are created by the messaging system on behalf of a server. The creation of an object requires designating a port from which messages sent to the object will be retrieved.

According to one embodiment, the messaging system is used by the event manager in that the event distributors have an associated port and object, the sequential consumers have an associated port and object, and the event manager maintains a port and objects assigned to that port for each broadcast consumer. The events are passed as messages by the event producers to the distributor's associated object, the sequential consumers receive the messages from their associated port, and the broadcast consumers send messages to request events to the event manager via the object maintained for that broadcast consumer. Additionally, according to one embodiment, the event manager uses the message filter mechanism provided in a messaging system to implement sequential consumers. For each sequential consumer, a message filter is created on the distributor's ObjectID. The code which receives messages through the filter checks for matches (on the what and who provided) and forwards the appropriate messages to the sequential consumer. One example of such a message filtering mechanism is described in the above-referenced patent application in the name of Thomas E. Saulpaugh and Steven J. Szymanski.

FIG. **2** is a block diagram of the architecture for the event manager **30** shown in FIG. **1**. The event manager **30** (shown by the dashed line in FIG. **2**) is operationally connected to and communicates with a plurality of event distributors **340**, corresponding in number to the different kinds of events possible within the system, a plurality of event consumers, and a plurality of event producers **300**.

According to one embodiment, the plurality of event consumers can include broadcast consumers **310** and sequential consumers **360**. It is appreciated that a given system may have either one or more broadcast consumers or one or more sequential consumers, or both.

According to the present invention, events are grouped into "kinds", for example, all keystrokes are one kind of event, all mouse clicks are another kind of event, all new file creations am another kind of event, etc. It is also possible to group the events differently, for example, all depressions of a particular key are one kind of event. This is not a preferred implementation at least in part because the number of kinds of events would quickly become unmanageable. However, the choice of how to group the events is within the skill of the ordinary artisan once in possession of the present disclosure.

The event manager **30** includes an event manager control unit **305** and data structures. The data structures include a subscription matrix **330**, a sequential consumer database **350**, a plurality of event queues **320** provided in one-to-one correspondence with the broadcast consumers **310**, and a plurality of event kind headers **331** provided in one-to-one correspondence with the event distributors **340**. The event manager control unit **305** consists of at least one software routine which manages the event manager data structures.

Event producers **300** represent any software on a computer that is responsible for generating an event, or is responsible for detecting that other entities in the computer have generated an event. The event producers generate descriptions of each event they produce or detect. Each kind of event can have any number of event producers, and a given event produce may produce or detect multiple kinds of events. Event consumers, including broadcast consumers **310** and sequential consumers **360**, represent any program that needs to be informed when an event has occurred and needs to be informed of the description of that event. Each kind of event can have any number of event consumers, and a given event consumer may need to be notified of multiple kinds of events.

It is possible for an event consumer to be an event producer and for an event producer to be an event consumer. In particular, there are many cases where a piece of software will consume one kind of event, and produce another kind in response. For example, the file manager might consume "disk inserted" events, mount the volumes on the inserted media, and then produce "new volume" events as a result. It is also possible that a given program may subscribe to different kinds of events differently. In particular, the program may subscribe as a sequential consumer for some kinds of events and as a broadcast consumer for other kinds of events.

Event queues **320** are lists of events that are maintained by the event manager control unit **305** for each of the broadcast consumers **310** to hold information about events of interest to the corresponding broadcast consumer. According to one embodiment, the event queues **320** are structured as first-in, first-out lists. Other suitable storage configurations such as lists ordered by event priority or by producer priority, for example, disk inserted events may always have higher priority than mouse clicks or disk event producers may have higher priority than mouse event producers, may also be used. In FIG. **2**, event queue #1 is the list of events of interest to broadcast consumer #1, event queue #2 is the list of events of interest to broadcast consumer #2, and so on. Although four broadcast consumer/queue pairs are shown in FIG. **2**, it is appreciated that any number of broadcast consumer/queue pairs can be defined at a given time.

According to one embodiment, the event queues **320** are stored in the format shown in FIG. **3**, including the following fields: first subscription; last subscription; maximum events; next event; and event array. The first and last subscription fields are pointers to the first and last subscription for the corresponding broadcast consumer **310**. An event subscription is a description of a specific set of events of which a particular broadcast consumer needs to be informed. The maximum events field is the maximum number of events that are to be maintained in the queue for that consumer. According to one embodiment, the size of the event array may be slightly larger than the indicated maximum to provide a cushion. The event array stores the list of events that have occurred in the form of an array of elements, each element being an event to which the corresponding broadcast consumer has subscribed. In one embodiment, the queues are stored in circular buffers, although other suitable storage configurations such as singly and doubly linked lists and dynamic stacks could be used.

Subscription matrix **330** is a structure that maintains the information about all existing event subscriptions. In particular, the subscription matrix **330** is used to keep track of the subscriptions for the events in which the broadcast consumers are interested.

A plurality of sequential consumers **360** may be defined. Although three sequential consumers #1, #2, and #3, are shown in FIG. **2**, it is appreciated that any number of sequential consumers can be defined. The sequential consumer database **350** is composed of a plurality of sequential consumer entries which list the events in which each sequential consumer **360** is interested.

In one embodiment, the subscription matrix **330** can be configured as illustrated in FIG. **4**, where there is one subscription structures **3300** stored for each subscription. Other suitable configurations may be used such as sparse arrays, or dynamic lists of subscriptions on the event queue and/or event kind structures. The subscription structures **3300** stored in the subscription matrix are connected to the event queues **320** and the event kind headers **331**. In particular, there is one event kind header **331** stored for each kind of event known to the system, thus the number of event kind headers is equal to the number of event distributors. The event kind headers **331** store information for the event manager control unit **305** to use to determine which distributor handles the kind of event currently being processed. A RegisterDistributor (described below) call creates the event kind header.

According to one embodiment, the subscriptions **3300** are stored in the subscription matrix **330** in the format shown in FIG. **5***a*, including the following fields: event queue pointer; event kind header pointer; next subscription, same event; previous subscription, same event; next subscription, same queue; previous subscription, same queue; and event subject. The event queue pointer is a pointer to the event queue to which the subscription belongs. The event kind header pointer is the pointer to the event kind header for the kind of event which is the subject of the subscription. The event subject identifies the structure which the event happened to. The rest of the fields are the pointers to the other subscriptions.

An exemplary format for the sequential consumer entries is illustrated in FIG. **5***b*. According to one embodiment, the entries each include the following fields: event identifier, event subject and sequential consumer object ID. The event identifier identifies the kinds of events in which the sequential consumer is interested. The event subject identifies the subject of the events in which the sequential consumer is interested. The sequential consumer object ID identifies the objectID for the sequential consumer which created the entry in the sequential consumer database. According to one embodiment, the sequential consumer database **350** is part of the subscription matrix **330** and there is a pointer to the subscription structure for each sequential consumer. According to another embodiment shown in FIG. **2**, the sequential consumer database **350** is provided separately from the subscription matrix **330** as described above. The event manager control unit **305** performs a comparison between the data structure and the detected event to determine whether the event should go to that sequential consumer.

According to one embodiment, the message filtering mechanism is used to maintain the information required to provide the appropriate sequential consumers with the events as they occur. This mechanism is generally described in the Saulpaugh and Szymanski patent application discussed above.

According to one embodiment, the event kind headers **331** are stored in the format shown in FIG. **6**. The event kind header **331** includes the following fields: next header; DistributorObjectId; event identifier; first subscription; and last subscription. The next header field is a pointer to next header in a singly linked list of event kind headers. The DistributorObjectID is the distributor ObjectID for the correspond-

ing event distributor. The event identifier identifies the kind of event the associated distributor handles. The first and last subscription fields are pointers to first and last subscriptions defined for this kind of event.

Event distributors **340** are programs that are responsible for interpreting event subscriptions so as to distribute appropriate events to appropriate broadcast consumers. According to one embodiment, them is an event distributor for handling each kind of event. In particular, there is an event distributor to handle all keystrokes detected by any of the event producers, another event distributor to handle all mouse clicks, and so on.

According to one embodiment, it is up to the distributors to determine which broadcast consumers am notified of the event. The event manager control unit **305**, using the subscription matrix **330**, keeps track of which consumers want the events, while the distributors have the final say as to which consumers are actually notified. A given distributor might always give the result to all interested consumers, or it might always choose one, or it might ask to see each consumer and choose some and not others. This allows for a very flexible architecture. For example, while all applications will be interested in mouse clicks (and therefore would subscribe to that kind of event), only one (either the frontmost or the application for which a window is clicked on) should actually receive it. The mouse click event distributor determines which of the subscriptions was for the frontmost application and sends it only to that one. Another example would be that there might be several pieces of code which would want to know when a new file was created, and generally all of them should be told. Therefore, the new file event distributor would always send the event to all subscribers.

As shown in FIG. **2**, each broadcast consumer **310** communicates with and is operationally connected to its respective event queue **320** through the event manager control unit **305**. Each broadcast consumer **310** also communicates with and is operationally connected to the subscription matrix **330** through the event manager control unit **305**. The event queues **320** each communicate with and are operationally connected to the subscription matrix **330** through the event manager control unit **305**. The sequential consumer database **350** communicates with and is operationally connected to the event manager control unit **305** and the sequential consumers **360** communicate with the sequential consumer database **350** via the event manager control unit **305**. The plurality of event kind headers **331** are operationally connected to the plurality of event distributors **340**, which are operationally connected to the plurality of event producers **300**.

Each broadcast consumer **310** uses the event manager control unit **305** to cream an event queue **320** to hold events in which it is interested between the time the event is reported and the time the broadcast consumer consumes it. Each broadcast consumer **310** communicates to the event manager control unit **305** a set of event subscriptions that describe all the events of which it needs to be informed. This set of event subscriptions may be changed at any time. The event manager control unit **305** stores that information in the subscription matrix to be used by the event distributors **340**.

In summary, according to one embodiment of the present invention, event producers **300** detect events and build event descriptions. They send those descriptions to the event manager **30** by calling the event manager control unit **305**. The event manager control unit **305** sends the event description to each sequential consumer **360** in turn based on the entries in the sequential consumer database **350**. The event manager control unit **305** then sends the event description to the event distributor **340** who is responsible for distributing that kind of event. The event distributor **340** calls the event manager control unit **305** to send the event description to those broadcast consumers **310** which it decides are appropriate based on the information in the subscription matrix **330**. The event manager control unit **305** gives the event descriptions to the appropriate broadcast consumers **310** by initially storing those descriptions in the event queues **320**. The broadcast consumers **310**, when ready, call the event manager control unit **305** to retrieve the next event description stored in its corresponding event queue **320**.

## APPLICATION PROGRAMMER INTERFACE

According to the present invention, the event consumers, the event producers and the event distributors may be written by third parties other than the manufacturer of the event manager. Therefore, an application programmer interface (API) is defined to provide a specification which allows these third parties to communicate with the event manager. The following is a description of one embodiment of an API which allows communication with the event manager according to the present invention.

### Event Structure

According to one embodiment, an event is composed of three parts: what happened (the event identifier), who it happened to (the event subject), and details of how the event happened (the event information). Since it is desirable for the set of possible events to be easily extensible (by the developers of the other parts of the operating system and by applications programmers), the "what" part of the event structure is defined to be a unique identifier referred to as the event name. In one embodiment, the event name identifier can be implemented as a **4** character code known as an OSType. It is appreciated that other suitable identifiers could be used instead by one of ordinary skill once in possession of the present disclosure. According to one embodiment, it can be a pair of identifiers. In particular, in this disclosure, the implementation described uses two OSTypes. An exemplary structure **45** for the event name identifier is shown in FIG. **7**. The event service identifier serves as the "signature" or name of the service which defined the event, e.g., a word processing program, and the event kind identifier identifies the event itself, e.g., new file created. Thus, the universe of all names, referred to as the namespace, of events is managed by controlling the signatures.

The "who" field of the event structure, defined according to one embodiment as the event subject, is difficult to define since all possible uses of the event manager can not be anticipated and therefore all forms of the hardware and software elements that the "who" might describe also can not be anticipated. Fortunately, the only public operation which needs to be supported for this part of an event is a test for equality, i.e., equality against a "who" which was provided by a broadcast consumer for a subscription, so the event manager control unit **305** need not know the structure of the subscription and it can be defined as an uninterpreted array of bytes. Suitable configurations of the "who" field can be used, such as a fixed length or a variable length field, within the skill of the ordinary artisan once in possession of the present disclosure.

The "details" part of the structure, that is, the info field in FIG. **8**, is totally open ended, and will vary not only with each event, but potentially with each instance of the event. Thus, one embodiment for the structure is an open ended array of bytes.

FIG. **8** is a block diagram of the event structure according to one embodiment of the present invention. The event **40** consists of the event name, event subject, byte count, and event info fields. The event name field specifies the what, the event subject field specifies the who, and the byte count and event into fields specify the how. Event info is the actual data and byte count indicates the length of the data.

## System Calls

The event producers are provided with a single call to submit events to the event manager control unit **305**. In particular, the call sends the message of an event to the event manager control unit **305**. According to one embodiment, the producers then receive confirmation that every consumer who needs to respond to the event has had a chance to process it. In particular, when the call returns, the distributor knows that every sequential consumer has seen it, and it is enqueued for every broadcast consumer. This confirmation is not required and can be omitted according to another embodiment. According to one embodiment, this call has been split into two calls (FindDistributor and ProduceEvent) For efficiency reasons to actor out the identification of who knows how to distribute that kind of event. The API is shown in Table 1.

Table 1 and the succeeding tables present structure definitions written in the C language. While the examples herein are shown in C, it is within the skill of the ordinary artisan to use other suitable programming languages to implement the present invention. In addition, the code shown in the Tables is an example of an implementation of the invention according to one embodiment. It is appreciated that other implementations are possible and within the skill of the ordinary artisan once in possession of the present disclosure.

In Table 1, OSStatus is a type usually used as a return value used to indicate if the call succeeded. OptionBits is a type usually used as an input parameter to a call to allow the user to specify small variations on how the call is to be processed.

TABLE 1

```
typedef Object ID                                   EventDistributorID;
typedef OptionBits                                  ProduceEventOptions;
enum
    {
    kProduceEventsynchronously                      = 0x00000001
    // don't return until all Sequential
    Consumers have completed
    }
OSStatus FindDistributor (      EventName           eventname,
                                EventDistributorID  *inputID);
OSStatus ProduceEvent (         EventInformationPtr event,
                                EventDistributorID  distributor
                                ProduceEventOptions options);
```

## Consumers

According to one embodiment of the present invention, as discussed above, broadcast consumers and sequential consumers differ in their relationships to other consumers of an event. Broadcast consumers do not need to know if other consumers exist, nor in what order consumers are informed of the event, as long as they themselves are eventually informed. Sequential consumers require that no other consumer be told about an event while they themselves are still processing it, and they require the ability to influence when in the sequence they receive the event. In addition, many sequential consumers require the ability to modify the event itself, and even block an event from being received by other consumers.

According to one embodiment, the API provides a method for filtering events based on "who" they involve to allow broadcast consumers to limit the set of subjects for which they will be notified about events. In addition, the API provides a method which allows the broadcast consumer to modify the set of events in which it is interested over time. Lastly, the API supports the ability of broadcast consumers to poll the event manager control unit **305** to collect an event, or to be notified asynchronously by use of a messaging system providing asynchronous notification functionality.

In the interface according to one embodiment, the consumer calls the event manager control unit **305** to create a consumer structure which embodies a list of events in which the consumer is interested. These structures are opaque to the caller (i.e., its details are known only to the event manager control unit **305**).

According to one embodiment, this consumer structure is an event queue **320**. The API allows the broadcast consumers to call the event manager control unit **305** to create an event queue. In particular, the event queues **320** are created by the event manager control unit **305** as a result of a CreateBroadcastConsumer call, and they are eliminated as a result of a DisposeBroadcastConsumer call. One embodiment of the details of the queues are illustrated in FIG. **3**. The event queue includes, among other things, the name of the consumer (used by distributors to identity specific consumers), and the maximum number of events the event manager control unit **305** will buffer by storing in the event queue for that consumer. The consumer is asked to provide this latter value so the event manager control unit **305** is not committed to an unbounded amount of buffering. Should this value be exceeded, additional events are discarded and the next event the consumer will get is an "Overrun" event from the event manager control unit **305** itself. The example code in Table 2 illustrates this feature.

TABLE 2

```
dypedef ObjectID EventConsumerID;
typedef FilterName EventConsumerName;
OSStatus CreateBroadcastConsumer (
    EventConsumerID                 *consumer,
    EventConsumerName               name,
    uint32                          maxPending);
OSStatus DisposeBroadcastConsumer (
    EventConsumerID                 consumer);
```

Once the broadcast consumer has an EventConsumerID, it subscribes to the kinds of events in which it is interested. As part of the subscription process the broadcast consumer can specify both the EventName and the EventSubject to be matched for determining when the consumer needs to be notified of the event. The policies of how the EventSubject is matched by the event distributor **340** is dependent on the kind of event being processed. According to one embodiment, this can be implemented as a byte-by-byte comparison. Other suitable implementations may be used, for example, if there are three different kinds of events that are identical for subscription purposes, i.e., the same consumers are to be notified of them, therefore, these events are

compared as equal even though the actual bits are different. The list of events associated with an EventConsumerID can be expanded at any time using SubscribeBroadcastConsumer, and the events kinds can be removed from the list at any time with UnsubscribeBroadcastConsumer. If a particular kind of event is unsubscribed, any event instances of that kind which have already been collected for that consumer are discarded. This is illustrated in Table 3.

TABLE 3

| | |
|---|---|
| OSStatus SubscribeBroadcastConsumer ( | |
| EventConsumerID | consumer, |
| EventName | eventName, |
| EventSubjectPtr | subject); |
| OSStatus UnsubscribeBroadcastConsumer ( | |
| EventConsumerID | consumer, |
| EventName | eventName, |
| EventSubjectPtr | theSubject); |

## Processing Events

The HoldEvents/UnholdEvents calls are provided to simplify some special cases in the processing of events. When a kind of event is held, instances of those events are still collected for the broadcast consumer; however, they will not be returned to the user by a Consume call until they are unheld. In this way, the event manager handles situations in which particular kinds of events are still of interest to a broadcast consumer, but the processing of those events are temporarily impossible or not a priority. According to another embodiment, the same functionality could be provided by creating two separate broadcast consumer IDs, one for the non-held events and one for the holdable events. The broadcast consumer could then choose when to consume from the second EventConsumerID. However, this embodiment may not be completely practical because it is not always possible to anticipate what will belong in which category, and the decision process for holding is often quite removed from the event processing cycle. The HoldEvents/UnholdEvents calls, according to one embodiment, provide a simple interface for separating that decision process. These calls are illustrated in Table 4.

TABLE 4

| | |
|---|---|
| OSStatus HoldEvents ( EventConsumerID | consumer, |
| EventName | eventName, |
| EventSubjectPtr | subject); |
| OSStatus UnholdEvents ( EventConsumerID | consumer, |
| EventName | eventName, |
| EventSubjectPtr | subject); |

Finally, when the list of interesting events is built, the broadcast consumer consumers the events which are collected using one of two calls, a synchronous call and an asynchronous call. Both calls specify the consumer objectID to identify the queue, and provide a buffer into which the event is copied. The Async call additionally requires a EventNotification structure which describes the action to be performed when a buffer has been filled in with a new event. According to one embodiment, the EventNotification and EventInformationPtr are types defined by an operating system kernel. If the buffer provided is not large enough to hold the event description, the leading part of the description is copied into the buffer. The rest of the description is dropped. These calls are illustrated in Table 5.

TABLE 5

| | |
|---|---|
| OSStatus ConsumeEvent ( | |
| EventConsumerID | conswner, |
| uint32 | maxEventSize, |
| EventInformationPtr | event); |
| OSStatus ConsumeEventASync ( | |
| EventConsumerID | conswner |
| EventNotification | *completion, |
| uint32 | maxEventSize, |
| EventInformationPtr | event); |

FlushEvents is used to remove events which have been collected for the broadcast consumers but which have not been consumed. The broadcast consumer specifies an EventName (which can use wildcards) and EventSubject. The event manager control unit **305** then disposes of any events which match the description and were collected but not processed. This call is illustrated in Table 6.

TABLE 6

| | |
|---|---|
| OSStatus FlushEvents ( | |
| EventConsumerID | consumer, |
| EventName | eventname, |
| EventSubjectPtr | subject); |

Sequential Consumers have very different requirements tier their API. Since they serve as a bottleneck for the transmission of events, their API should be designed for maximum throughput; and should be structured to require a response for each event to indicate when the event manager control unit **305** can pass the event on to the next sequential consumer or the broadcast consumers. Based on this, the API according to one embodiment of the present invention assumes that events are passed to sequential consumers as messages using a suitable communication facility, such as the messaging system service provided by a microkernel operating system as discussed above. According to one embodiment, the sequential consumer sends a message to the event manager control unit **305** asking for the next event and the event itself is returned in the reply to the message. Other implementations are also possible within the skill of the ordinary artisan once in possession of the present disclosure. According to one embodiment, sequential consumers receive these messages by way of an accept function or by asynchronous receives to allow the receiver to overlap program execution with the receive operation to insure a rapid response time. One example of such a messaging system is provided by NuKERNEL™, discussed above.

According to one embodiment, since the messaging system provides a funneling mechanism, having multiple objects associated with a single port, no subscription calls are needed for sequential consumers. Instead, the sequential consumer will be asked to create an object for each interesting kind of event, and those objects can be associated with ports in any way the consumer desires. The objects are then associated with a kind of event using the API described below. Other suitable embodiments of this feature are within the skill of the ordinary artisan once in possession of the present disclosure.

According to one embodiment, the actual sequencing of the consumers is provided by using the message filtering mechanism to transparently list, screen, alter, or re-route messages. Message filters are a software fabricated message interception device which can be ordered in a predetermined priority order to allow certain filters to take precedence over others. One embodiment of the present invention implements the sequencing of the consumers using this ordering

capability. This mechanism is described in detail in the copending application filed in the name of Saulpaugh and Szymanski referenced above.

The InstallSequentialConsumer call is used to tell the event manager control unit **305** that events of the given name and subject should be passed to the given object. The name, ordering and options parameters are used to determine the exact ordering in which those events are given to the sequential consumers that are interested in them. The RemoveSequentialConsumer call is used to remove the installed sequential consumer. The installation and removal calls are illustrated in Table 7.

TABLE 7

```
typedef FilterID SequentialConsumerID;
typedef FilterName
SequentialConsumerName;
OSStatus InstallSequentialConsumer (
    SequentialConsumerID          *consumer,
    SequentialConsumerName        consumersName,
    ObjectID                      consumersObject
    FilterOptions                 consumerOptions,
    FilterobjectPair              consumerOrdering,
    EventName                     eventname,
    EventSubjectPtr               eventSubject);
OSStatus RemoveSequentialConsumer (
    SequentialConsumerID          consumer);
```

Once a sequential consumer has been installed and has received or consumed an event, the sequential consumer needs to indicate when it is done with the event so the event manager control unit **305** can pass it on to the next sequential consumer (or to the broadcast consumers if there am no more sequential ones). In particular, the sequential consumer makes a NextConsumer call to the event manager control unit **305** indicating that it has completed processing of the event. The event manager control unit **305** then passes the event to the next sequential consumer or to the appropriate broadcast consumers. Instead of passing the event on, a sequential consumer has the ability to declare an event handled so that it will not be distributed to any more consumers of any type. In particular, a sequential consumer can make an EventHandled call to prohibit further distribution of the event. These two calls are illustrated in Table 8. In one embodiment, NextConsumer uses the messaging system call ContinueMessage which acts as an automatic forward command to continue to the next object in a filter chain in the messaging system, and EventHandled uses the messaging system call ReplyToMessage which specifies that all the objects in a chain have processed the message. This mechanism is described in detail in the copending application filed in the name of Saulpaugh and Szymanski referenced above.

TABLE 8

```
OSStatus NextConsumer (     MessageID     eventMessageID);
OSStatus EventHandled (     MessageID     eventMessageID);
```

## Distributors

As discussed above, each distributor is responsible for processing different kinds of events. Each distributor must therefore know the events for which it is responsible.

According to one embodiment, most events are routed through the appropriate sequential consumers and then copies are given to all of the broadcast consumers who have expressed interest. According to a default implementation, any matching of event subjects by the event distributors is done by direct byte comparison of the whole value to determine which consumers have to be notified of the event. Other suitable implementations are within the skill of the ordinary artisan once in possession of the present disclosure.

According to one embodiment, a default distributor is provided to pass certain kinds of events to all broadcast consumers. For example, events such as battery low events will always be distributed to all interested consumers. Each event that can be distributed by the default distributor must register with the default distributor by the RegisterEventWithDefaultDistributor call, illustrated in Table 9. This call is made one time per kind of event over the life of the system to tell the default distributor that it is responsible for those events.

TABLE 9

```
OSStatus RegisterEventWithDefaultDistributor(
    EventName          eventName);
```

A custom distributor is registered with the event manager control unit **305** for handling each kind of event that is not handled by the default distributor. According to one embodiment, registering a custom distributor with the event manager control unit **305** is performed by a call which associates an event with an object so that when the associated event occurs, the event manager control unit **305** knows which distributor will handle the distribution of that event. The custom distributors are independently loaded services and all communication between producers and distributors is accomplished by sending and receiving messages. To receive the events to process and distribute, each distributor must provide an ObjectID to which events are sent from the producers. Calls for registering and unregistering custom distributors are illustrated in Table 10.

TABLE 10

```
OSStatus RegisterDistributor(
    EventName                     eventname,
    ObjectID                      distributor);
OSStatus UnregisterDistributor(
    EventName                     eventname,
    ObjectID                      distributor);
```

There are several cases where the distributor needs to apply more complicated heuristics to the process of determining which broadcast consumers should receive notification of events reported by the producers. The information in the subscription matrix is available to the distributor to help determine which broadcast consumers to notify. According to one embodiment, a distributor cannot give an event to a broadcast consumer who has not asked for it, and the distributor looks in the subscription matrix to find out who asked for it. However, distributors are allowed to give the event to only a subset of the broadcast consumers who asked for it. The kinds of ways a distributor might need to effect the distribution of events include:

1. The way in which event subjects are matched is more complicated than direct byte comparison. There may be substructure in the subject identification which needs to be taken into account, or there may be some form of wildcard processing which needs to be handled.
2. Particular events may need to be routed to a subset of the broadcast consumers who have expressed interest, perhaps only one. For instance, mouse events need to be routed to only the one appropriate application or those applications which are interested in them.

3. Related events may need to be combined. For instance, multiple update region events for the same window should be combined until the application actually consumes the event.

Note that none of these possibilities affect sequential consumers. This is because one of the reasons sequential consumers are needed is to influence the distribution process. In other words, sequential consumers are so labelled because for the kinds of events in which they are interested, they must react to the event. Thus, there are no circumstances under which a distributor would choose not to send the event to a particular sequential consumer. Therefore, the event manager control unit **305** automatically sends all produced events through the gauntlet of sequential consumers before handing them off to the distributors.

FIGS. **9**A, **9**B, and **9**C are flowcharts illustrating the event handling process from the event producers' and the event distributors' point of view according to one embodiment of the present invention.

As shown in FIG. **9**A, the event producer first detects or generates an event (step **600**). The event producer generates a description of the event (step **602**) and calls the event manager control unit **305** to send the event to the appropriate event consumers, both broadcast and sequential consumers (step **604**).

FIG. **9**B illustrates the processing of the event by the event manger control unit **305** according to one embodiment of the present invention. It is appreciated that although the following description assumes the use of a messaging system to send the events between elements of the computer, suitable alternative embodiments can be implemented by an ordinarily skilled artisan once in possession of the present disclosure.

At step **606**, a determination is made whether the event producer needs to know when the event has been handled, that is, whether to process the request asynchronously or synchronously. If the event producer does not need to know when the event has been handled (as indicated by an input parameter to the call), it returns to the producer (step **608**) and processes the rest of the steps in this Figure asynchronously (step **610**). If the event producer does need to know, the steps are processed synchronously (step **612**).

The test is implemented by making the message sent in step **606** an asynchronous message, otherwise it is a synchronous message. In particular, the event producer calls the event manager control unit **305** to send a message concerning the detected event. The producer provides an input parameter having a value which indicates whether it wants to wait for a response, in which case the message is sent synchronously. Otherwise, the message is sent asynchronously. By definition, if the message is sent synchronously, the event manager control unit **305** does not return from that call until the message has been processed by all sequential consumers; if the message is sent asynchronously, as soon as the message is sent, the event manager control unit **305** returns back to the event producer so that the event producer does not know whether the message is processed.

At step **614**, the event manager control unit **305** sends the event to each sequential consumer having an entry in the sequential consumer database **350** matching the event description. This is done one at a time, and each sequential consumer can, by calling the event manager control unit **305**, either have the message sent to the next sequential consumer or declare that the event has been handled. Thus, after the sequential consumer completes processing the event, the sequential consumer indicates whether processing of the event should continue or whether the event processing should stop because the event has been handled (step **616**). A determination is then made by the event manager control unit **305** whether that sequential consumer indicated that the event was handled (step **618**). If so, the event is prohibited from being distributed to any other consumers and so the routine returns to the calling producer. If the sequential consumer indicated that the event was not handled, a determination is made whether that sequential consumer is the last one (step **622**). If not, the distributor returns to step **616** to distribute the event to the next sequential consumer.

Referring now to FIG. **9**C, when the event has been distributed to all sequential consumers (step **622**), the event manager control unit **305** sends the event to the event distributor responsible for that kind of event to distribute to the broadcast consumers (process **624**). The event is sent as a message to the distributor. The messaging system automatically passes it on to the distributor when all of the filters have processed it and none have replied (thereby declaring the event handled). Other suitable implementations are possible, such as having the event manager control unit **305** call the event distributor at this point, are within the skill of the ordinary artisan once in possession of the present disclosure.

Process **624** is illustrated in FIG. **9**D. The distributor decides how to distribute the event in this process. According to one embodiment, it can do this by calling the event manager control unit **305** to sequence through all of the subscriptions for that kind of event to see, which of the consumers who have subscribed should get the event.

One situation is where the distributor needs to route an event to all or some of the consumers; but needs to be able to modify the event data itself up until the point at which the consumer receives and consumes it (step **6240**). For example, in a windowing environment, when a window is uncovered (i.e., the topmost window is closed), a window uncovered event is generated. Because it is possible that another portion of the uncovered window may still be uncovered, the window uncovered event distributor wants to be able to modify this event up until the point at which it is consumed. In this case, the distributor submits a place holder structure for the actual message, and gives it to the appropriate consumers.

The place holder structure includes a reference constant field, RefCon, to be used by the distributor to identify the event, and an ObjectID to be used to get the actual data from the distributor. The distributor provides the value of the RefCon field when it creates the placeholder, and is responsible for being able to take that value and provide back the actual event description. Thus, when the consumer actually tries to consume the placeholder, the event manager control unit **305** sends a message to the given ObjectID containing the placeholder RefCon. The distributor then fills in the buffer with the actual, current event description and the consumer's buffer is passed to Consume. Since this functionality needs to be provided for both select and all consumers, two calls are provided as shown in Table 11. If all consumers are to be notified of an event using a place holder (step **6248**), the GivePlaceholderToAllConsumers call is used (step **6250**). If only select consumers are to be notified of an event using a place holder (step **6248**), the GivePlaceholderToSelectConsumers call is used (step **6252**).

TABLE 11

```
OSStatus GivePlaceholderToAllConsumers
    Event InformationPtr        event,
    void*                       placeholderRefCon,
    ObjectID                    fullfilemntID);
```

TABLE 11-continued

```
OSStatus
GivePlaceholderToSelectConsumers
    EventInformationPtr          event,
    ConsumerFilterFunction       fliterFunc,
    void*                        placeholderRefCon,
    ObjectID                     fullfilemntID);
```

If the distributor does not need to modify the event data, the result of the determination at step **6240** is no. Then, in step **6242**, a determination is made if all of the broadcast consumers should receive the event. Note that if this is the case all of the time, it is preferable to use the default distributor which automatically notifies all consumers. Alternatively, if special processing is required only for a subset of events, for any event not in that subset, the GiveEventToAllConsumers call can be used to invoke the distribution mechanism (step **6244**). This call is illustrated in Table 12.

TABLE 12

```
OSStatus
GiveEventToAllConsumers(EventInformationPtr    event);
```

The situation in which the distributor needs to choose a subset of the consumers to receive the message may occur either because the distributor needs to process any subject matches or because the nature of the event requires limiting the distribution is detected by a no response to the test in step **6240**. In particular, in cases where a byte-by-byte comparison of the subject field is not desired, the distributor is responsible for determining if the subjects match. To accomplish this, the GiveEventToSelectConsumers call is used (step **6246**), and a filter function, ConsumerFilterFunction, is called for each consumer who has subscribed to that kind of event (regardless of the subject they specified). The function returns true if the event should be given to that consumer, false if not. This call is illustrated in Table 13.

TABLE 13

```
typedef Boolean (*ConsumerFilterFunction)(
    EventInformationPtr          event,
    EventConsumerName            consumer,
    EventSubjectPtr              subsciptnSubj);
OSStatus GiveEventToSelectConsumers(
    EvenInformationPtr           event,
    ConsumerFilterFunction       filterFunc);
```

After each of the calls at steps **6244**, **6246**, **6250**,and **6252**, the routine returns to the event manager control unit **305** (step **6254**).

Referring now to FIG. **9**C, at step **625**, the event manager control unit **305** enqueues the event according to which call was used in FIG. **9**D. According to one embodiment, the events are enqueued and dequeued in a first in, first out (FIFO) order. When an event consumer is ready to act on an event, it dequeues the top most event in the event queue and handles it as required.

If the steps of FIG. **9**B were processes synchronously (step **626**), the routine returns to the producer at step **628**. Otherwise, the routine replies to the producer's message (step **630**).

FIG. **10** is a flowchart illustrating the event handling process from the sequential consumers' point of view according to one embodiment of the present invention. First, the sequential consumer calls the event manager control unit **305** to get installed (step **700**). According to one embodiment, the consumer provides an ObjectID to which events are to be sent as messages. It is appreciated that other suitable implementations are possible, for example, the consumer could provide a function pointer which is called with a pointer to the event description. Such suitable implementations are within the skill of the ordinary artisan once in possession of the present disclosure.

The sequential consumer does whatever it wants while waiting for an event to occur (step **702**). According to one embodiment, since events are sent as messages, the sequential consumer calls the messaging system to either accept or receive a message through the given Object. Other things may be done by the sequential consumer, depending on whether the process is synchronous or asynchronous.

When an event is received (step **704**), the sequential consumer acts on the event in the appropriate way (step **706**). The sequential consumer then decides whether it wants other consumers (sequential or broadcast) to see this event or if it has handled the event well enough that others should not see it (step **707**). If the event is done, and others should not see it, an event handled message is sent by calling the event manager control unit **305** (step **711**). Otherwise, the sequential consumer calls the event manager control unit **305** to say that the event should be passed to the next consumer (step **709**). The sequential consumer then loops back to the wait state of step **702**.

FIG. **11** is a flowchart illustrating the event handling process from the broadcast consumers' point of view according to one embodiment of the present invention. The broadcast consumer first calls the event manager control unit **305** to register itself using the CreateConsumer call. Once registered, the distributor will start receiving messages through the object referred to by the given ObjectID. The contents of these messages are the descriptions generated by the producers of the events they have detected or generated. The distributor must then indicate to the event manager control unit **305** which of the currently executing consumers should receive the event. In one embodiment, the messaging system routes events from the producers, through the sequential consumers, and into the distributors. The distributor then uses the event manager control unit **305** to put the event on the appropriate queues.

The event manager control unit **305** creates an event queue for the consumer and returns (step **802**). The broadcast consumer then calls the event manager control unit **305** to subscribe to particular kinds of events in which it is interested (step **804**). The event manager control unit **305** adds the entries in the subscription matrix to keep track of the consumer's subscriptions (step **806**).

The consumer calls the event manager control unit **305** to get the next event (step **808**). According to one embodiment, this is implemented by having the consumer send a message to the event manager control unit **305**. The reply to that message will be the next event for that consumer. The consumer has the option of doing this synchronously or asynchronously, and the consumer either sends the message synchronously or asynchronously accordingly. Other suitable implementations are possible within the skill of the ordinary artisan once in possession of the present disclosure. The consumer acts on the event as it sees fit (step **810**), and the process loops back to step **808**.

The foregoing description of the specific embodiments will so fully reveal the general nature of the invention that others can, by applying current knowledge, readily modify and/or adapt for various applications such specific embodiments without departing from the generic concept, and, therefore, such adaptations and modifications should and are

intended to be comprehended within the meaning and range of equivalents of the disclosed embodiments. It is to be understood that the phraseology of terminology employed herein is for the purpose of description and not of limitation.

What is claimed is:

**1**. In a computer including at least one event producer for detecting that an event has occurred in the computer and generating an event and at least one event consumer which needs to be informed when events occur in the computer, a system for distributing events comprising:

storing means for storing a specific set of events of which said at least one event consumer is to be informed;

event manager control means for receiving the event from the event producer, comparing the received event to the stored set of events, and distributing an appropriate event to an appropriate event consumer; and

distributor means for receiving the event from the control means and directing said control means to distribute an appropriate event to an appropriate event consumer.

**2**. The system according to claim **1**, wherein said distributor means comprises a distributor module for each kind of event possible in the computer.

**3**. The system according to claim **1**, wherein a plurality of event consumers are included in the computer and the plurality of consumers comprise:

broadcast consumers having no relationship with other consumers, the broadcast consumers operating independently of other consumers and of the order in which consumers are informed of the event; and

sequential consumers having relationships with other consumers, the sequential consumers requiring that no other consumer be told about an event while they themselves are processing the event and having an ability to influence when they receive the event relative to the other consumers.

**4**. The system according to claim **3**, wherein said distributor means comprises:

means for determining if the event is to be sent to broadcast consumers with a signal indicating that the distributor means maintains a right to modify the event based on subsequent events until the broadcast consumers receive the event; and

means, responsive to a positive determination by said means for determining, for directing said control means to distribute the event to the appropriate broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event.

**5**. The system according to claim **3**, wherein said distributor means comprises:

first means for determining if the event is to be sent to broadcast consumers with a signal indicating that the distributor means maintains a right to modify the event;

second means, responsive to a positive determination by said first means for determining, for determining if the event is to be passed to all broadcast consumers;

means, responsive to a positive determination by said second means for determining, for directing said control means to distribute the event to all broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event;

means, responsive to a negative determination by said second means for determining, for directing said control means to distribute the event to select broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event;

third means, responsive to a negative determination by said first means for determining, for determining if the event is to be sent to all broadcast consumers;

means, responsive to a positive determination by said third means for determining, for directing said control means to distribute the event to all broadcast consumers; and

means, responsive to a negative determination by said third means for determining, for directing said control means to distribute the event to select broadcast consumers.

**6**. The system according to claim **3**, wherein said storing means comprises:

a subscription matrix for storing subscriptions to events in which the broadcast consumers are interested; and

a sequential consumer database for storing entries to events in which the sequential consumers are interested.

**7**. The system according to claim **3**, wherein said storing means comprises an event queue corresponding to each of the broadcast consumers for receiving distributed events from said control means and for storing the distributed events until the events are consumed by the corresponding broadcast consumer.

**8**. The system according to claim **3**, wherein said control means comprises means for passing an event to the sequential consumers in succession in accordance with the entries in the sequential consumer database.

**9**. The system according to claim **8**, wherein said control means comprises means for prohibiting passing of an event upon receiving an event handled message from a sequential consumer.

**10**. A computer system comprising:

event producers for detecting that an event has occurred in the computer, generating an event, and generating a description of the event;

event consumers which need to be informed when events occur in the computer, said event consumers comprising a first and a second class of consumers;

storing means for storing a specific set of events of which the event consumers are to be informed;

event manager control means for receiving the event from the event producers and comparing the received event to the stored set of events;

distributor means, responsive to said event control means, for deciding if an event should be passed to an event consumer;

said event manager control means comprising:

first means for sending an event to appropriate event consumers of a first class in accordance with the stored set of events, and

second means for sending the event to appropriate event consumers of a second class responsive to said distributor means.

**11**. The system according to claim **10**, wherein said distributor means comprises a distributor module for each kind of event possible in the computer.

**12**. The system according to claim **10**, wherein

said first class of consumers comprise sequential consumers having relationships with other consumers, the sequential consumers requiring that no other consumer be told about an event while they themselves are processing it, and having an ability to influence when they receive the event relative to the other consumers; and

said second class of consumers comprise broadcast consumers having no relationship with other consumers, the broadcast consumers operating independently of other consumers and of the order in which consumers are informed of the event.

**13**. The system according to claim **12**, wherein said distributor means comprises:

means for determining if the event is to be sent to broadcast consumers with a signal indicating that the distributor means maintains a right to modify the event based on subsequent events until the broadcast consumers receive the event; and

means, responsive to a positive determination by said means for determining, for directing said control means to distribute the event to the appropriate broadcast consumers with a signal indicating that the distributor means maintains the right to modify the event.

**14**. The system according to claim **12**, wherein said storing means comprises:

a subscription matrix for storing subscriptions to events in which the broadcast consumers are interested; and

a sequential consumer database for storing entries to events in which the sequential consumers are interested.

**15**. The system according to claim **12**, wherein said storing means comprises an event queue corresponding to each of the broadcast consumers for receiving distributed events from said control means and for storing the distributed events until the events are consumed by the corresponding broadcast consumer.

**16**. The system according to claim **12**, wherein said control means comprises means for passing an event to the sequential consumers in succession in accordance with the entries in a sequential consumer database.

**17**. The system according to claim **16**, wherein said control means comprises means for prohibiting passing of an event upon receiving an event handled message from a sequential consumer.

**18**. A method for distributing events occurring in a computer, said method comprising the steps of:

determining that an event has been detected by an event producer in the computer;

storing, in a storing means, a specific set of events of which an event consumer is to be informed;

receiving the event in an event control means from the event producer;

comparing the received event to the stored set of events;

receiving the event in a distributor means from the control means;

directing the control means to distribute an appropriate event to an appropriate event consumer; and

distributing, via the control means, an appropriate event to an appropriate event consumer.

**19**. The method according to claim **18**, wherein the event consumer comprises a plurality of consumers including broadcast consumers which operate independently from one another and of the order in which consumers are informed of events and sequential consumers which require that no other consumer be told about an event while they themselves are processing it and have an ability to influence when they receive the event relative to the other consumers.

**20**. The method according to claim **19**, wherein said step of distributing comprises the steps of:

determining if the event is to be sent to broadcast consumers with a right to modify the event based on subsequent events until the broadcasts consumers receive the event; and

distributing the event to the appropriate broadcast consumers with the right to modify responsive to a positive determination by said step determining.

**21**. The method according to claim **19**, wherein said step of storing comprises the steps of:

storing, in a subscription matrix, subscriptions to events in which the broadcast consumers am interested; and

storing, in a sequential consumer database, entries to events in which the sequential consumers are interested.

**22**. The method according to claim **19**, further comprising the steps of:

receiving distributed events in an event queue corresponding to a broadcast consumer; and

storing the distributed events in the event queue until the events are consumed by the corresponding broadcast consumer.

**23**. The method according to claim **19**, wherein the step of distributing comprises the step of passing an event to the sequential consumers in succession upon receiving a continue message from a sequential consumer indicating that it has completed processing of the event.

**24**. The method according to claim **23**, wherein the step of distributing further comprises the step of prohibiting passing of an event upon receiving an event handled message from a sequential consumer.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 5,566,337

DATED : October 15, 1996

INVENTOR(S) : Steven J. SZYMANSKI, et al.

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

Item [57]

Replace the Abstract as follows:

In a computer including event producers for generating events and detecting that an event has occurred in the computer and event consumers which need to be informed when events occur in the computer, a system distributes information about the events. The system includes a store for storing a specific set of events of which an event consumer is to be informed, an event manager control unit for receiving the event from an event producer, comparing the received event to the stored set of events, and distributing the appropriate event to the appropriate event consumer, and a distributor for receiving the event from the control unit and directing the control unit to distribute the appropriate event to the appropriate event consumer. The system manages events within the computer by facilitating communication between the event producers and the event consumers without requiring each event producer to be aware of all of the event consumers.

**Signed and Sealed this**

**Twenty-eighth Day of January, 1997**

*Attest:* Bruce Lehman

**BRUCE LEHMAN**

*Attesting Officer* *Commissioner of Patents and Trademarks*

# EXHIBIT 4

US005915131A

# United States Patent [19]

Knight et al.

[11] Patent Number: 5,915,131

[45] Date of Patent: Jun. 22, 1999

[54] **METHOD AND APPARATUS FOR HANDLING I/O REQUESTS UTILIZING SEPARATE PROGRAMMING INTERFACES TO ACCESS SEPARATE I/O SERVICES**

[75] Inventors: **Holly N. Knight**, La Honda; **Carl D. Sutton**, Palo Alto; **Wayne N. Meretsky**, Los Altos; **Alan B. Mimms**, San Jose, all of Calif.

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

[21] Appl. No.: **08/435,677**

[22] Filed: **May 5, 1995**

[51] **Int. Cl.**[6] ..... **G06F 9/40**; G06F 13/14

[52] **U.S. Cl.** ..... **395/892**; 395/682; 395/828; 395/702; 707/104; 345/333

[58] **Field of Search** ..... 395/828, 702, 395/834, 200.2, 892, 682, 309; 345/333; 707/104

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 4,593,352 | 6/1986 | Castel et al. | 364/200 |
| 4,727,537 | 2/1988 | Nichols | 370/85 |
| 4,908,859 | 3/1990 | Bennett et al. | 380/10 |
| 4,982,325 | 1/1991 | Tignor et al. | 364/200 |
| 5,129,086 | 7/1992 | Coyle, Jr. et al. | 395/650 |
| 5,148,527 | 9/1992 | Basso et al. | 395/325 |
| 5,197,143 | 3/1993 | Lary et al. | 395/425 |
| 5,430,845 | 7/1995 | Rimmer et al. | 395/275 |
| 5,491,813 | 2/1996 | Bondy et al. | 395/500 |
| 5,513,365 | 4/1996 | Cook et al. | 395/800 |
| 5,535,416 | 7/1996 | Feeney et al. | 395/834 |
| 5,537,466 | 7/1996 | Taylor et al. | 379/201 |
| 5,553,245 | 9/1996 | Su et al. | 395/284 |
| 5,572,675 | 11/1996 | Bergler | 395/200.2 |

OTHER PUBLICATIONS

Forin, A., et al. entitled "An I/O System for Mach 3.0," Proceedings of the Usenix Mach Symposium 20–22, Nov. 1991, Monterey, CA, US, 20–22 Nov. 1991, pp. 163–176.

Steve Lemon and Kennan Rossi, entitled "An Object Oriented Device Driver Model," Digest of Papers Compcon '95, Technologies for the Information Superhighway 5–9, Mar. 1995, San Francisco, CA, USA pp. 360–366.

Glenn Andert, entitled "Object Frameworks in the Taligent OS," Intellectual Leverage: Digest of Papers of the Spring Computer SOCI International Conference (Compcon), San Francisco, Feb. 28–Mar. 4, 1994, Feb. 24, 1994, Institute of Electrical and Electronics Engineers, pp. 112–121.

Hu, 'Interconnecting electronic mail networks: Gateways and translation strategies are proposed for backbone networks to interchange incompatible electronic documents on multivendor networks', Data Communications, p. 128, vol. 17, No. 10, Sep. 1988.

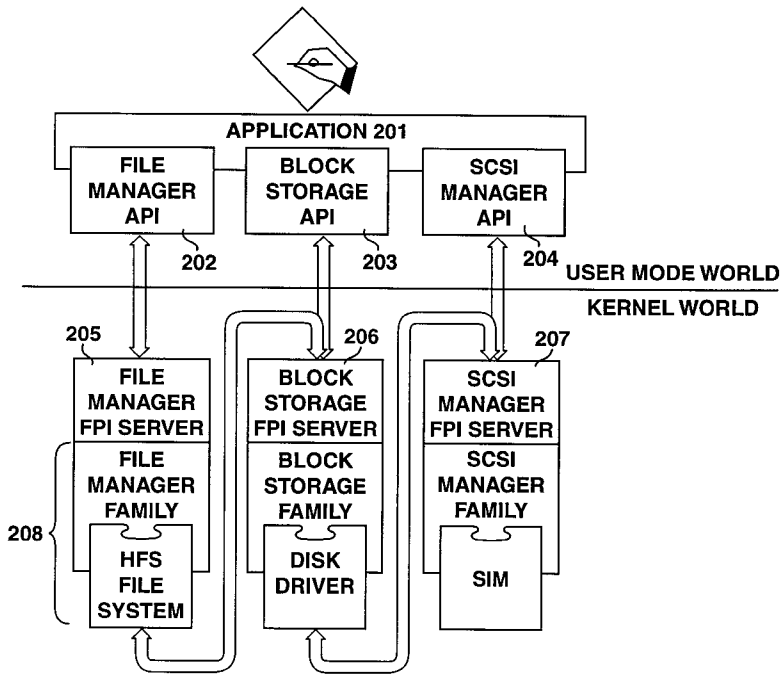
Knibbe, 'IETF's Resource Reservation Protocol to facilitate mixed voice, data, and video nets', Network World, p. 51, Apr. 24, 1995.

*Primary Examiner*—Thomas C. Lee
*Assistant Examiner*—Rehana Perveen
*Attorney, Agent, or Firm*—Blakely, Sokoloff, Taylor & Zafman

[57] **ABSTRACT**

A computer system handling multiple applications wherein groups of I/O services are accessible through separate application programming interfaces. Each application has multiple application programming interfaces by which to access different families of I/O services, such as I/O devices.

**20 Claims, 8 Drawing Sheets**



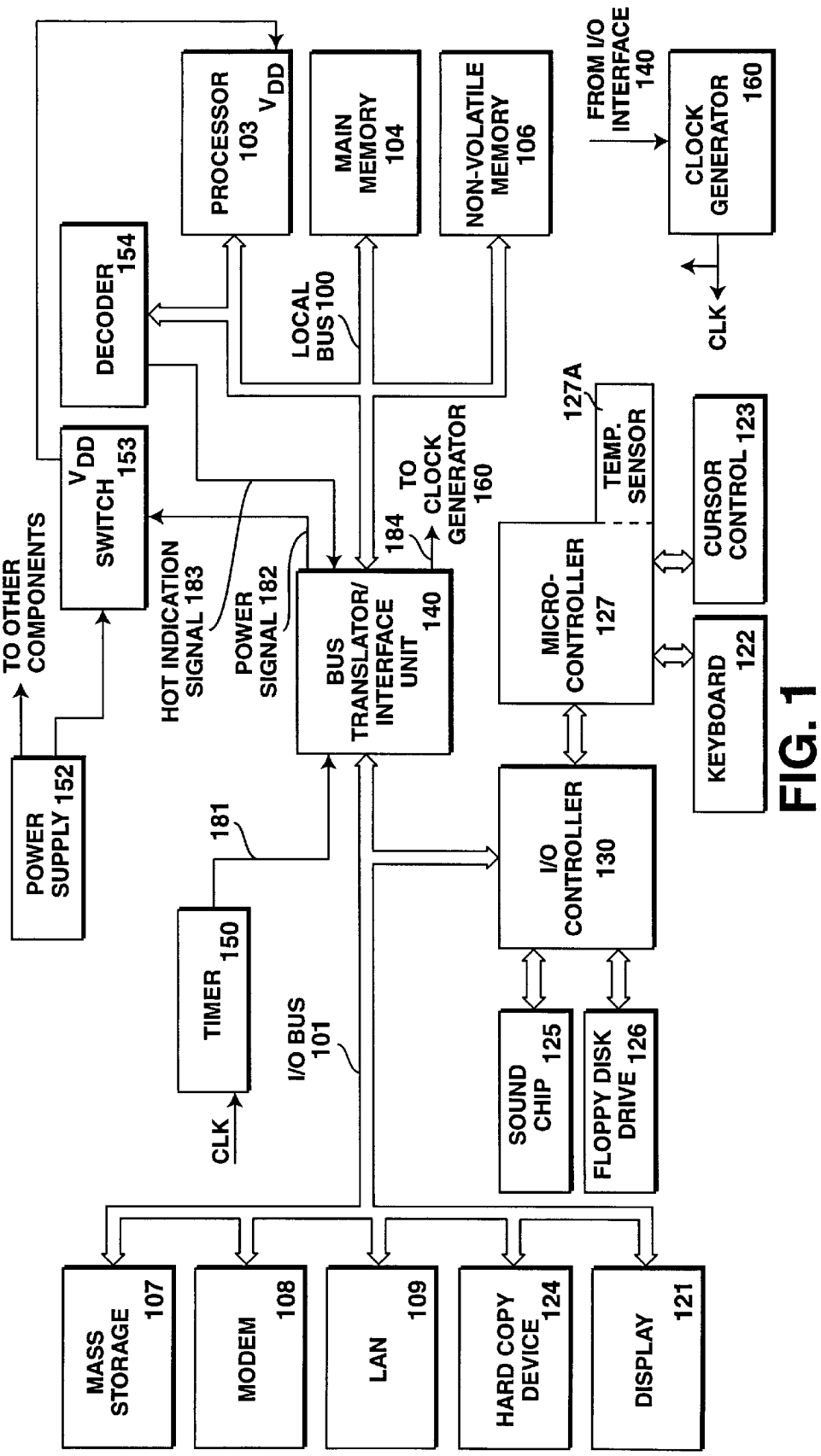

PROCESSOR 103 $V_{DD}$
MAIN MEMORY 104
NON-VOLATILE MEMORY 106
FROM I/O INTERFACE 140
CLOCK GENERATOR 160
DECODER 154
LOCAL BUS 100
CLK
$V_{DD}$ SWITCH 153
TO OTHER COMPONENTS
HOT INDICATION SIGNAL 183
POWER SIGNAL 182
BUS TRANSLATOR/ INTERFACE UNIT 140
184
TO CLOCK GENERATOR 160
127A
TEMP. SENSOR
MICRO-CONTROLLER 127
CURSOR CONTROL 123
KEYBOARD 122
POWER SUPPLY 152
181
I/O CONTROLLER 130
TIMER 150
CLK
I/O BUS 101
SOUND CHIP 125
FLOPPY DISK DRIVE 126
MASS STORAGE 107
MODEM 108
LAN 109
HARD COPY DEVICE 124
DISPLAY 121

FIG. 1

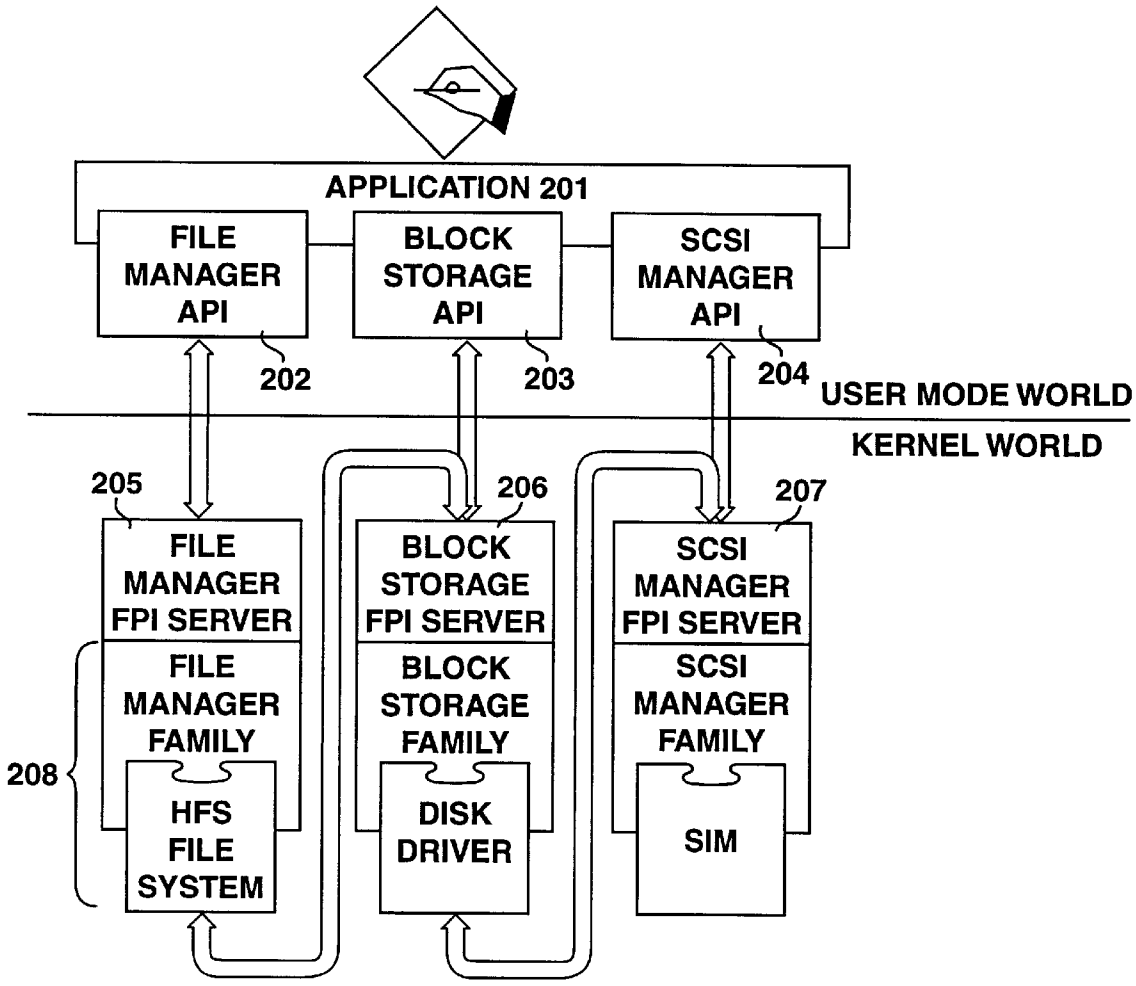

APPLICATION 201
FILE MANAGER API
BLOCK STORAGE API
SCSI MANAGER API
202
203
204
USER MODE WORLD
KERNEL WORLD
205
206
207
FILE MANAGER FPI SERVER
BLOCK STORAGE FPI SERVER
SCSI MANAGER FPI SERVER
FILE MANAGER FAMILY
BLOCK STORAGE FAMILY
SCSI MANAGER FAMILY
208
HFS FILE SYSTEM
DISK DRIVER
SIM

FIG. 2

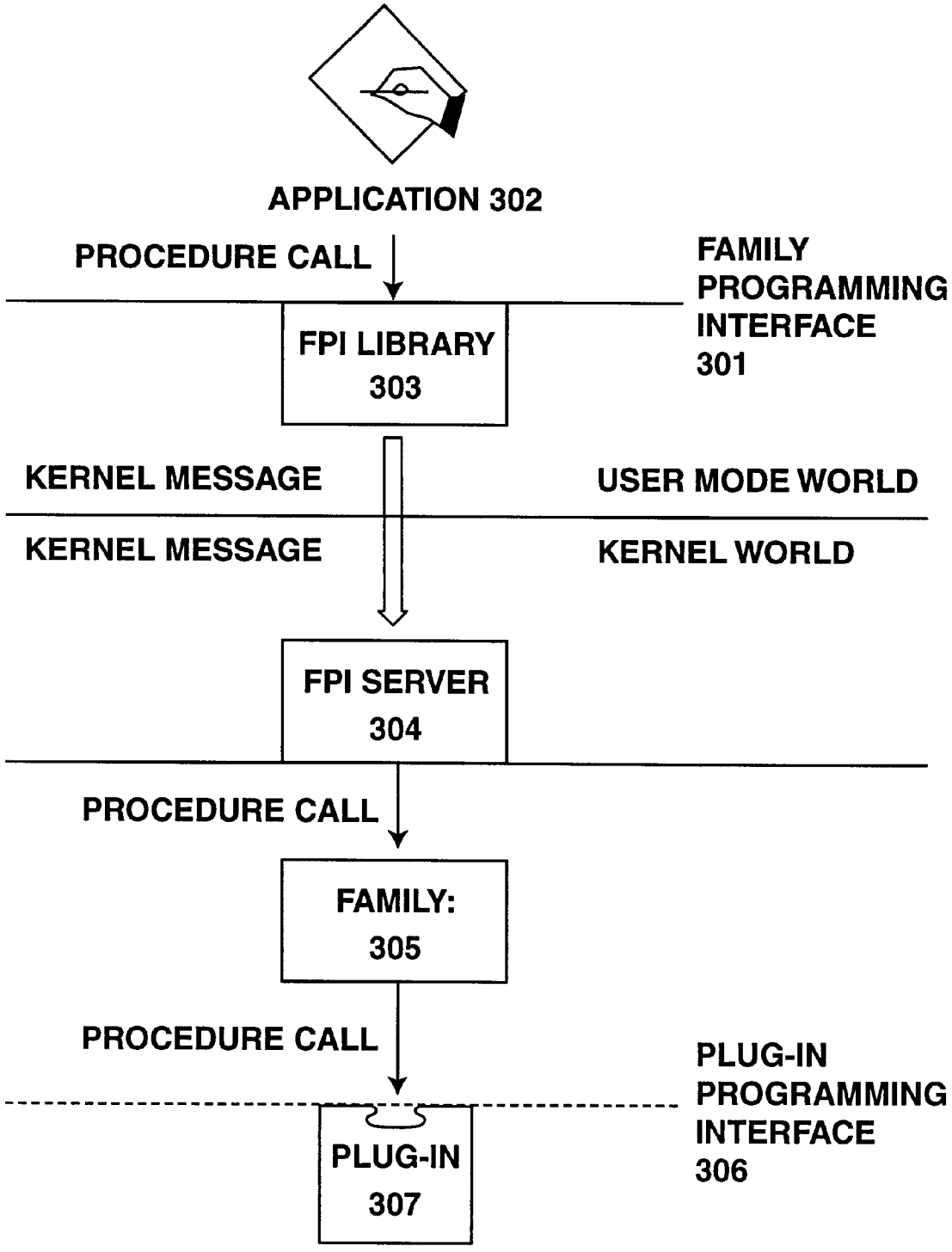

APPLICATION 302
PROCEDURE CALL
FAMILY PROGRAMMING INTERFACE 301
FPI LIBRARY 303
KERNEL MESSAGE
USER MODE WORLD
KERNEL MESSAGE
KERNEL WORLD
FPI SERVER 304
PROCEDURE CALL
FAMILY: 305
PROCEDURE CALL
PLUG-IN PROGRAMMING INTERFACE 306
PLUG-IN 307

FIG. 3

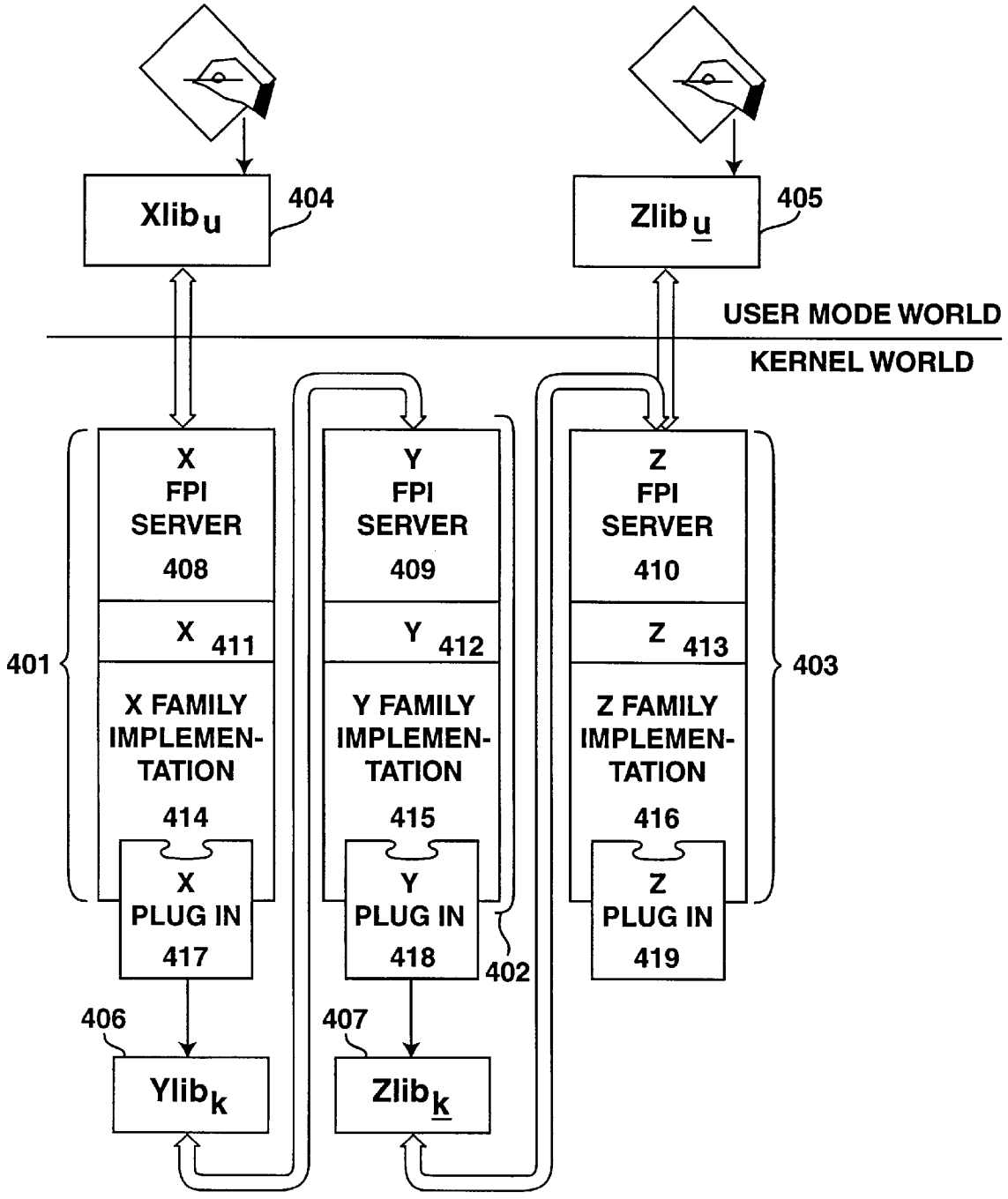

Xlib u
404
Zlib u
405
USER MODE WORLD
KERNEL WORLD
X FPI SERVER 408
Y FPI SERVER 409
Z FPI SERVER 410
X 411
Y 412
Z 413
401
403
X FAMILY IMPLEMEN-TATION 414
Y FAMILY IMPLEMEN-TATION 415
Z FAMILY IMPLEMEN-TATION 416
X PLUG IN 417
Y PLUG IN 418
Z PLUG IN 419
402
406
407
Ylib k
Zlib k

FIG. 4

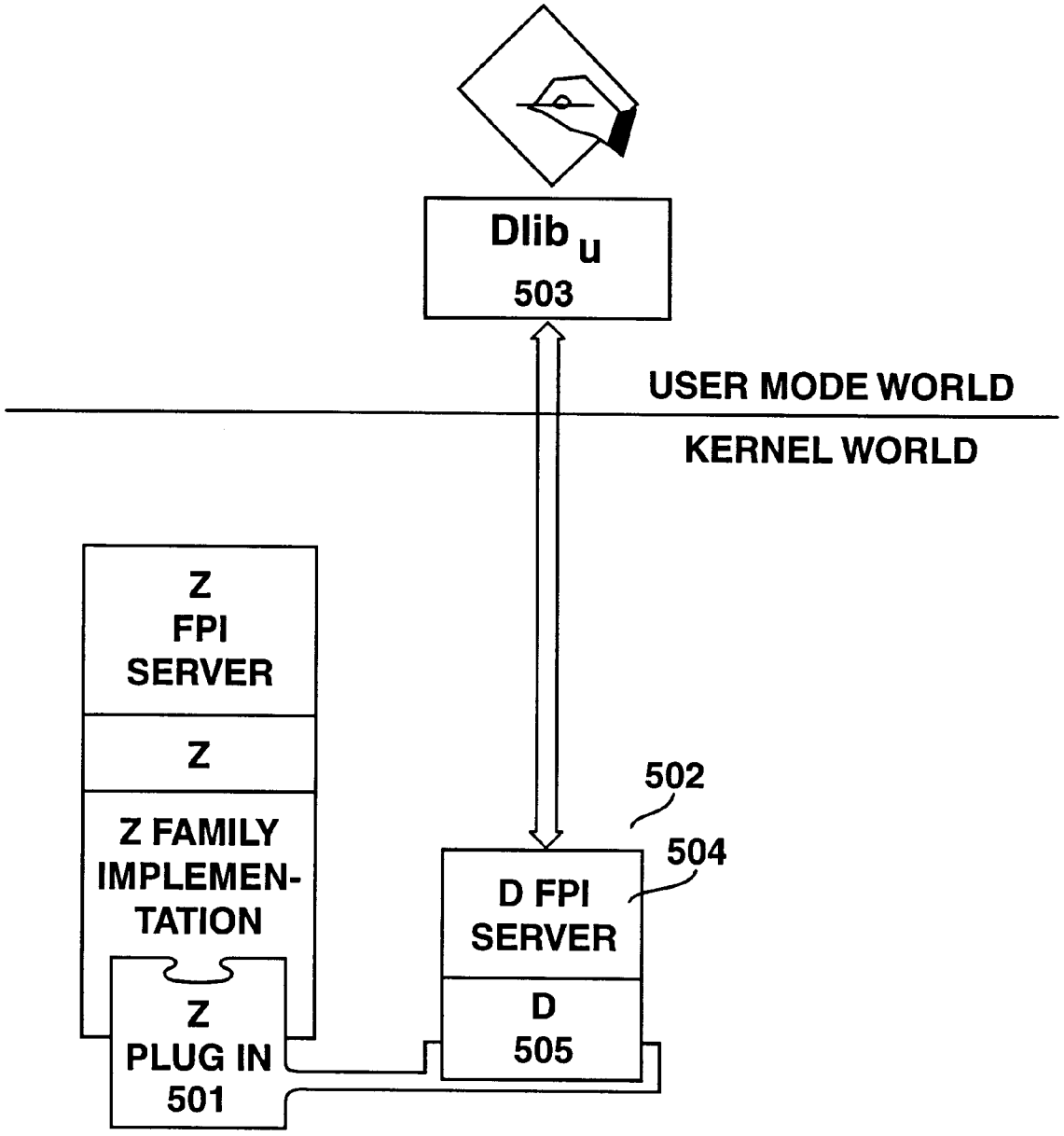

Dlib u
503
USER MODE WORLD
KERNEL WORLD
Z
FPI
SERVER
Z
Z FAMILY
IMPLEMEN-
TATION
Z
PLUG IN
501
502
504
D FPI
SERVER
D
505

FIG. 5

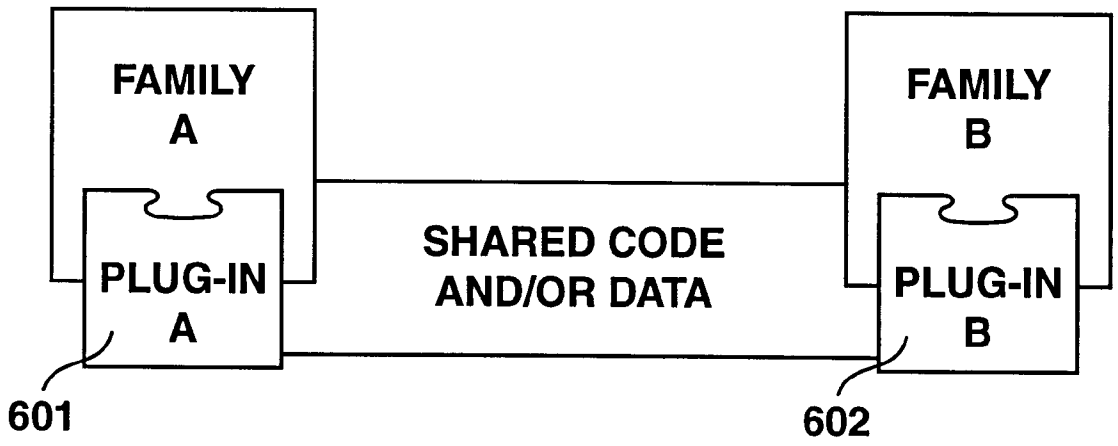

FAMILY A
PLUG-IN A
SHARED CODE AND/OR DATA
FAMILY B
PLUG-IN B
601
602

FIG. 6

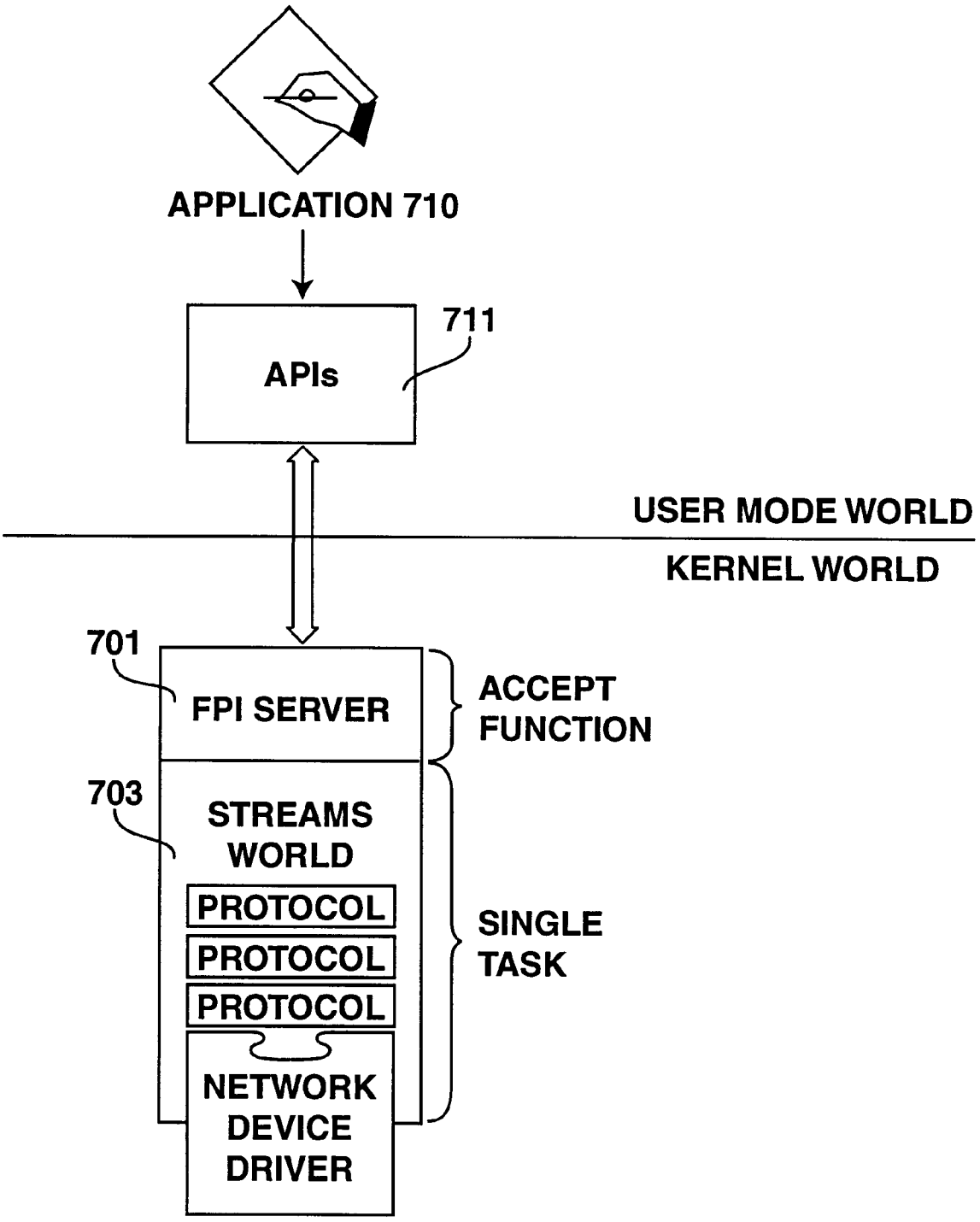

APPLICATION 710
711
APIs
USER MODE WORLD
KERNEL WORLD
701
FPI SERVER
ACCEPT FUNCTION
703
STREAMS WORLD
PROTOCOL
PROTOCOL
PROTOCOL
SINGLE TASK
NETWORK DEVICE DRIVER

FIG. 7

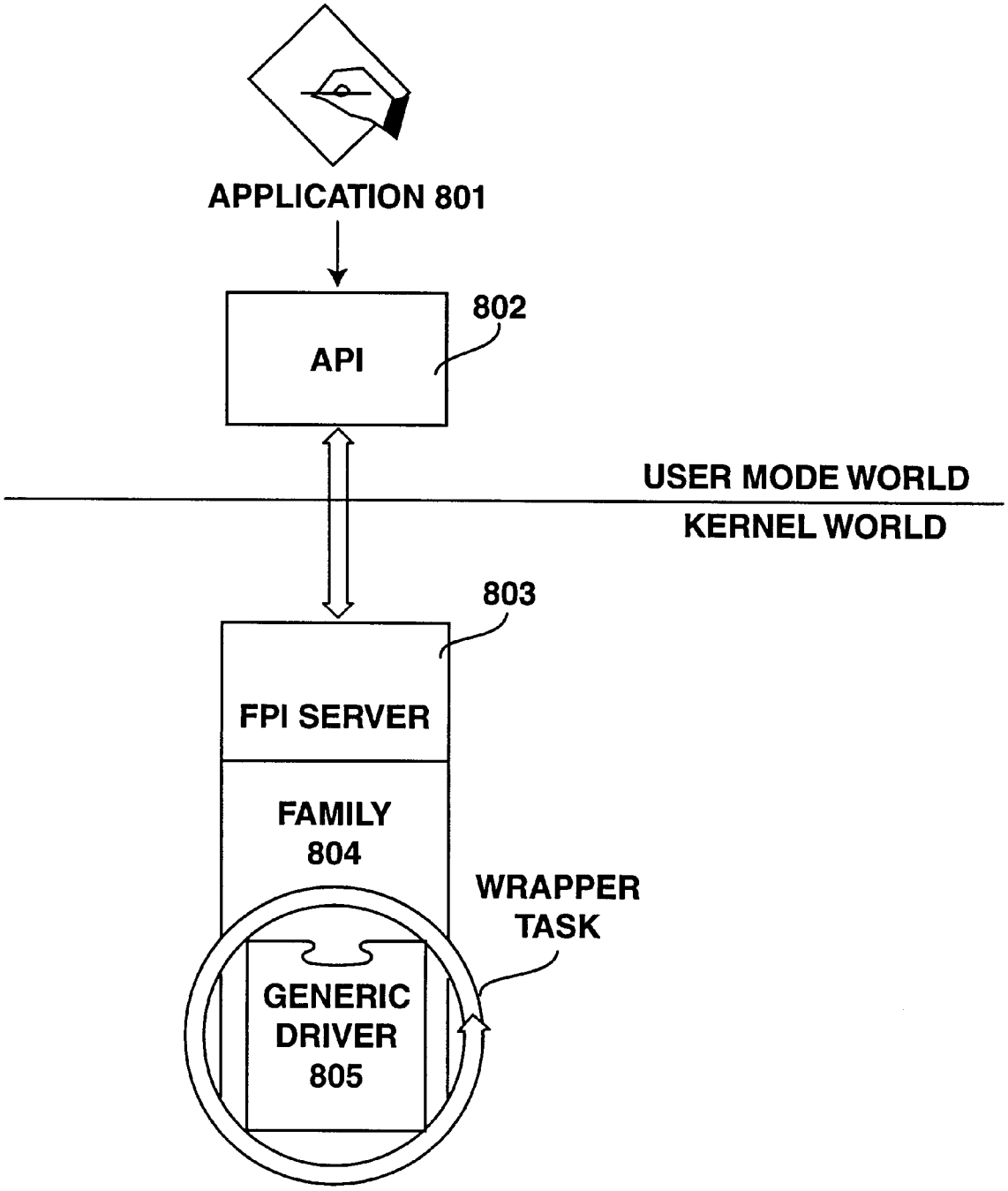

APPLICATION 801
802
API
USER MODE WORLD
KERNEL WORLD
803
FPI SERVER
FAMILY
804
WRAPPER
TASK
GENERIC
DRIVER
805

FIG. 8

# METHOD AND APPARATUS FOR HANDLING I/O REQUESTS UTILIZING SEPARATE PROGRAMMING INTERFACES TO ACCESS SEPARATE I/O SERVICES

## FIELD OF THE INVENTION

The invention relates to the field of computer systems; particularly, the present invention relates to handling service requests generated by application programs.

## BACKGROUND OF THE INVENTION

Application programs running in computer systems often access system resources, such as input/output (I/O) devices. These system resources are often referred to as services. Certain sets of services (e.g., devices) have similar characteristics. For instance, all display devices or all ADB devices have similar interface requirements.

To gain access to I/O resources, applications generate service requests to which are sent through an application programming interface (API). The service requests are converted by the API to a common set of functions that are forwarded to the operating system to be serviced. The operating system then sees that service requests are responded to by the appropriate resources (e.g., device). For instance, the operating system may direct a request to a device driver.

One problem in the prior art is that service requests are not sent directly to the I/O device or resource. All service requests from all applications are typically sent through the same API. Because of this, all of the requests are converted into a common set of functions. These common set of functions do not have meaning for all the various types of I/O devices. For instance, a high level request to play a sound may be converted into a write function to a sound device. However, the write function is not the best method of communicating sound data to the sound device. Thus, another conversion of write data to a sound data format may be required. Also, some functions do not have a one-to-one correspondence with the function set of some I/O devices. Thus, it would be desirable to avoid this added complexity and to take advantage of the similar characteristics of classes of I/O devices when handling I/O requests, while providing services and an environment in which to run those services that is tuned to the specific device needs and requirements.

## SUMMARY OF THE INVENTION

A method and apparatus for handling I/O requests is described. In the present invention, the I/O requests are handled by the computer system having a bus and a memory coupled to the bus that stores data and programming instructions. The programming instructions include application programs and an operating system. A processing unit is coupled to the bus and runs the operating system and application programs by executing programming instructions. Each application programs have multiple separate programming interfaces available to access multiple sets of I/O services provided through the operating system via service requests.

## BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be understood more fully from the detailed description given below and from the accompanying drawings of various embodiments of the invention, which, however, should not be taken to limit the invention to the specific embodiments, but are for explanation and understanding only.



FIG. **1** a block diagram of one embodiment in the computer system of the present invention.

FIG. **2** is an overview of the I/O architecture of the present invention.

FIG. **3** illustrates a flow diagram of I/O service request handling according to the teachings of the present invention.

FIG. **4** illustrates an overview of the I/O architecture of the present invention having selected families accessing other families.

FIG. **5** illustrates extended programming family interface of the present invention.

FIG. **6** illustrates plug-in modules of different families that share code and/or data.

FIG. **7** illustrates a single task activation model according to the teachings of the present invention.

FIG. **8** illustrates a task-per-plug-in model used as an activation model according to the teachings of the present invention.

## DETAILED DESCRIPTION OF THE PRESENT INVENTION

A method and apparatus handling service requests is described. In the following detailed description of the present invention numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form, rather than in detail, in order to avoid obscuring the present invention.

Some portions of the detailed descriptions which follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

The present invention also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may

comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose machines may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

Overview of the Computer System of the Present Invention

Referring to FIG. **1**, an overview of a computer system of the present invention is shown in block diagram form. The present invention may be implemented on a general purpose microcomputer, such as one of the members of the Apple family of personal computers, one of the members of the IBM personal computer family, or one of several other computer devices which are presently commercially available. Of course, the present invention may also be implemented on a multi-user system while encountering all of the costs, speed, and function advantages and disadvantages available with these machines.

As illustrated in FIG. **1**, the computer system of the present invention generally comprises a local bus or other communication means **100** for communicating information, a processor **103** coupled with local bus **100** for processing information, a random access memory (RAM) or other dynamic storage device **104** (commonly referred to as a main memory) coupled with local bus **100** for storing information and instructions for processor **103**, and a read-only memory (ROM) or other non-volatile storage device **106** coupled with local bus **100** for storing non-volatile information and instructions for processor **103**.

The computer system of the present invention also includes an input/output (I/O) bus or other communication means **101** for communication information in the computer system. A data storage device **107**, such as a magnetic tape and disk drive, including its associated controller circuitry, is coupled to I/O bus **101** for storing information and instructions. A display device **121**, such as a cathode ray tube, liquid crystal display, etc., including its associated controller circuitry, is also coupled to I/O bus **101** for displaying information to the computer user, as well as a hard copy device **124**, such as a plotter or printer, including its associated controller circuitry for providing a visual representation of the computer images. Hard copy device **124** is coupled with processor **103**, main memory **104**, non-volatile memory **106** and mass storage device **107** through I/O bus **101** and bus translator/interface unit **140**. A modem **108** and an ethernet local area network **109** are also coupled to I/O bus **101**.

Bus interface unit **140** is coupled to local bus **100** and I/O bus **101** and acts as a gateway between processor **103** and the I/O subsystem. Bus interface unit **140** may also provide translation between signals being sent from units on one of the buses to units on the other bus to allow local bus **100** and I/O bus **101** to co-operate as a single bus.

An I/O controller **130** is coupled to I/O bus **101** and controls access to certain I/O peripherals in the computer system. For instance, I/O controller **130** is coupled to controller device **127** that controls access to an alpha-numeric input device **122** including alpha-numeric and other keys, etc., for communicating information and command selections to processor **103**, and a cursor control **123**, such as a trackball, stylus, mouse, or trackpad, etc., for controlling cursor movement. The system also includes a sound chip **125** coupled to I/O controller **130** for providing audio recording and play back. Sound chip **125** may include a sound circuit and its driver which are used to generate various audio signals from the computer system. I/O controller **130** may also provide access to a floppy disk and driver **126**. The processor **103** controls I/O controller **130** with its peripherals by sending commands to I/O controller **130** via local bus **100**, interface unit **140** and I/O bus **101**.

Batteries or other power supply **152** may also be included to provide power necessary to run the various peripherals and integrated circuits in the computer system. Power supply **152** is typically a DC power source that provides a constant DC power to various units, particularly processor **103**. Various units such as processor **103**, display **121**, etc., also receive clocking signals to synchronize operations within the computer systems. These clocking signals may be provided by a global clock generator or multiple clock generators, each dedicated to a portion of the computer system. Such a clock generator is shown as clock generator **160**. In one embodiment, clock generator **160** comprise a phase-locked loop (PLL) that provides clocking signals to processor **103**.

I/O controller **140** includes control logic to coordinate the thermal management. Several additional devices are included within the computer system to operate with the control logic within I/O controller **140**. A timer **150**, a switch **153** and a decoder **154** are included to function in connection with the control logic. In one embodiment, decoder **154** is included within bus interface unit **140** and timer **150** is included in I/O controller **130**.

Switch **153** is a p-channel power MOSFET, which has its gate connected to the power signal **182**, its source to the power supply and its drain to processor's $V_{DD}$ pin.

In one embodiment, processor **103** is a member of the PowerPC™ family of processors, such as those manufactured by Motorola Corporation of Schaumberg, Ill. The memory in the computer system is initialized to store the operating system as well as other programs, such as file directory routines and application programs, and data inputted from I/O controller **130**. In one embodiment, the operating system is stored in ROM **106**, while RAM **104** is utilized as the internal memory for the computer system for accessing data and application programs. Processor **103** accesses memory in the computer system via an address bus within bus **100**. Commands in connection with the operation of memory in the computer system are also sent from the processor to the memory using bus **100**. Bus **100** also includes a bi-directional data bus to communicate data in response to the commands provided by processor **103** under the control of the operating system running on it.

Of course, certain implementations and uses of the present invention may neither require nor include all of the above components. For example, in certain implementations a keyboard or cursor control device for inputting information to the system may not be required. In other implementations, it may not be required to provide a display device displaying information. Furthermore, the computer system may include additional processing units.

The operating system running on processor **103** takes care of basic tasks such as starting the system, handling interrupts, moving data to and from memory **104** and peripheral devices via input/output interface unit **140**, and managing the memory space in memory **104**. In order to take care of such operations, the operating system provides

multiple execution environments at different levels (e.g., task level, interrupt level, etc.). Tasks and execution environments are known in the art.

Overview of the Present Invention

In one embodiment, the computer system runs a kernel-based, preemptive, multitasking operation system in which applications and I/O services, such as drivers, operate in separate protection domains (e.g., the user and kernel domains, respectively). The user domain does not have direct access to data of the kernel domain, while the kernel domain can access data in the user domain.

The computer system of the present invention uses one or more separate families to provide I/O services to the system. Each I/O family provides a set of I/O services to the system. For instance, a SCSI family and its SCSI interface modules (SIMs) provide SCSI based services, while a file systems family and its installable file systems provide file management services. In one embodiment, an I/O family is implemented by multiple modules and software routines.

Each family defines a family programming interface (FPI) designed to meet the particular needs of that family. An FPI provides access to a given family's plug-ins, which are dynamically loaded pieces of software that each provide an instance of the service provided by a family. For example, within the file systems family (File Manager), a plug-in implements file-system-specific services. In one embodiment, plug-ins are a superset of device drivers, such that all drivers are plug-ins, but not all plug-ins are drivers.

Access to services is available only through an I/O family's programming interface. In one embodiment, hardware is not directly accessible to application software, nor is it vulnerable to application error. Applications have access to hardware services only through an I/O family's programming interface. Also, the context within which an I/O service runs and the method by which it interacts with the system is defined by the I/O family to which it belongs.

FIG. **2** illustrates the relationship between an application, several I/O families, and their plug-ins. Referring to FIG. **2**, an application **201** requests services through one or more family FPIs, shown in FIG. **2** as File Manager API **202**, Block Storage API **203**, and SCSI Manager API **204**. The File Manager API **202**, Block Storage API **203**, and SCSI Manager API **204** are available to one or more applications in the user domain.

In one embodiment, the service requests from application **201** (and other applications) are sent through File Manager API **202**, Block Storage API **203**, and/or SCSI Manager API **204**, etc., and flow as messages to family FPI servers **205–207**, which reside in the kernel domain. In one embodiment, the messages are delivered using a kernel-supplied messaging service.

Any communication method may be used to communicate service requests to I/O families. In one embodiment, kernel messaging is used between the FPI libraries and the FPI server for a given family, between different families, and between plug-ins of one family and another family. The communication method used should be completely opaque to a client requesting a family service.

Each of the FPI servers **205–207** permit access to a distinct set of services. For example, File Manager FPI server **205** handles service for the file manager family of services. Similarly, the Block Storage FPI server **206** handles service requests for the block storage family of services.

Note that FIG. **2** shows three families linked by kernel messages. Messages flow from application level through a family to another family, and so on. For instance, a service request may be communicated from application level to the file system family, resulting in one or more requests to the block storage family, and finally one or more to the SCSI family to complete a service request. Note that in one embodiment, there is no hierarchical relationship among families; all families are peers of each other.

Families in the Present Invention

A family provides a distinct set of services to the system. For example, one family may provide network services, while another provides access to a variety of block storage mediums. A family is associated with a set of devices that have similar characteristics, such as all display devices or all ADB devices.

In one embodiment, each family is implemented in software that runs in the computer system with applications. A family comprises software that includes a family programming interface and its associated FPI library or libraries for its clients, an FPI server, an activation model, a family expert, a plug-in programming interface for its plug-ins, and a family services library for its plug-ins.

FIG. **3** illustrates the interaction between these components. Referring to FIG. **3**, a family programming interface (FPI) **301** provides access to the family's services to one or more applications, such as application **302**. The FPI **301** also provides access to plug-ins from other families and to system software. That is, an FPI is designed to provide callers with services appropriate to a particular family, whether those calls originate from in the user domain or the operating system domain.

For example, when an application generates data for a video device, a display FPI tailored to the needs of video devices is used to gain access to display services. Likewise, when an application desires to input or output sound data, the application gains access to a sound family of services through an FPI. Therefore, the present invention provides family programming interfaces tailored to the needs of specific device families.

Service requests from application **302** (or other applications) are made through an FPI library **303**. In one embodiment, the FPI library **303** contains code that passes requests for service to the family FPI server **304**. In one embodiment, the FPI library **303** maps FPI function calls into messages (e.g., kernel messages) and sends them to the FPI server **304** of the family for servicing. In one embodiment, a family **305** may provide two versions of its FPI library **303**, one that runs in the user domain and one that runs in the operating system kernel domain.

In one embodiment, FPI server **304** runs in the kernel domain and responds to service requests from family clients (e.g., applications, other families, etc.). FPI server **304** responds to a request according to the activation model (not shown) of the family **305**. In one embodiment, the activation model comprises code that provides the runtime environment of the family and its plug-ins. For instance, FPI server **304** may put a request in a queue or may call a plug-in directly to service the request. As shown, the FPI server **304** forwards a request to the family **305** using a procedure call. Note that if FPI library **303** and the FPI server **304** use kernel messaging to communicate, the FPI server **304** provides a message port.

Each family **305** includes an expert (not shown) to maintain knowledge of the set of family devices. In one embodiment, the expert comprises code within a family **305** that maintains knowledge of the set of family plug-ins within the system. At system startup and each time a change occurs, the expert is notified.

In one embodiment, the expert may maintain the set of family services using a central device registry in the system.

The expert scans the device registry for plug-ins that belong to its family. For example, a display family expert looks for display device entries. When a family expert finds an entry for a family plug-in, it instantiates the plug-in, making it available to clients of the family. In one embodiment, the system notifies the family expert on an ongoing basis about new and deleted plug-ins in the device registry. As a result, the set of plug-ins known to and available through the family remains current with changes in system configuration.

Note that family experts do not add or alter information in the device registry nor do they scan hardware. In one embodiment, the present invention includes another level of families (i.e., low-level families) whose responsibility is to discover devices by scanning hardware and installing and removing information for the device registry. These low-level families are the same as the families previously discussed above (i.e., high level family) in other ways, i.e. they have experts, services, an FPI, a library, an activation model and plug-ins. The low-level families' clients are usually other families rather than applications. In one embodiment, families are insulated from knowledge of physical connectivity. Experts and the device registry are discussed in more detail below.

A plug-in programming interface (PPI) **306** provides a family-to-plug-in interface that defines the entry points a plug-in supports so that it can be called and a plug-in-to-family interface that defines the routines plug-ins call when certain events, such as an I/O completion, occur. In addition, PPI **306** defines the path through which the family and its plug-in exchange data.

A family services library (not shown) is a collection of routines that provide services to the plug-ins of a family. The services are specific to a given family and they may be layered on top of services provided by the kernel. Within a family, the methods by which data is communicated, memory is allocated, interrupts are registered and timing services are provided may be implemented in the family services library. Family services libraries may also maintain state information needed by a family to dispatch and manage requests.

For example, a display family services library provides routines that deal with vertical blanking (which is a concern of display devices). Likewise, SCSI device drivers manipulate command blocks, so the SCSI family services library contains routines that allow block manipulation. A family services library that provides commonly needed routines simplifies the development of that family's plug-ins.

Through the PPI **306**, a call is made to a plug-in **307**. In one embodiment, a plug-in, such as plug-in **307**, comprises dynamically loaded code that runs in the kernel's address space to provide an instance of the service provided by a family. For example, within the file systems family, a plug-in implements file-system-specific services. The plug-ins understand how data is formatted in a particular file system such as HFS or DOS-FAT. On the other hand, it is not the responsibility of file systems family plug-ins to obtain data from a physical device. In order to obtain data from a physical device, a file system family plug-in communicates to, for instance, a block storage family. In one embodiment, block storage plug-ins provide both media-specific drivers, such as a tape driver, a CD-ROM driver, or hard disk driver, and volume plug-ins that represent partitions on a given physical disk. Block storage plug-ins in turn may make SCSI family API calls to access data across the SCSI bus on a physical disk. Note that in the present invention, plug-ins are a superset of device drivers. For instance, plug-ins may include code that does not use hardware. For instance, file system and block storage plug-ins are not drivers (in that drivers back hardware).

Applications, plug-ins from other I/O families, and other system software can request the services provided by a family's plug-ins through the family's FPI. Note also that plug-ins are designed to operate in the environment set forth by their family activation model.

In one embodiment, a plug-in may comprises two code sections, a main code section that runs in a task in the kernel domain and an interrupt level code section that services hardware interrupts if the plug-in is, for instance, a device driver. In one embodiment, only work that cannot be done at task level in the main code section should be done at interrupt level. In one embodiment, all plug-ins have a main code section, but not all have interrupt level code sections.

The main code section executes and responds to client service requests made through the FPI. For example, sound family plug-ins respond to sound family specific requests such as sound playback mode setting (stereo, mono, sample size and rate), sound play requests, sound play cancellation, etc. The interrupt level code section executes and responds to interrupts from a physical device. In one embodiment, the interrupt level code section performs only essential functions, deferring all other work to a higher execution levels.

Also because all of the services associated with a particular family are tuned to the same needs and requirements, the drivers or plug-ins for a given family may be as simple as possible.

Family Programming Interfaces

In the present invention, a family provides either a user-mode or a kernel-mode FPI library, or both, to support the family's FPI. FIG. **4** illustrates one embodiment of the I/O architecture of the present invention. Referring to FIG. **4**, three instances of families **401–403** are shown operating in the kernel environment. Although three families are shown, the present invention may have any number of families.

In the user mode, two user-mode FPI libraries, $Xlib_u$ **404** and $Zlib_u$ **405**, are shown that support the FPIs for families X and Z, respectively. In the kernel environment, two kernel-mode FPI libraries, $Ylib_k$ **406** and $Zlib_k$, **407**, for families Y and Z, respectively, are shown.

Both the user-mode and the kernel-mode FPI libraries present the same FPI to clients. In other words, a single FPI is the only way family services can be accessed. In one embodiment, the user-mode and kernel mode libraries are not the same. This may occur when certain operations have meaning in one mode and not the other. For example, operations that are implemented in the user-mode library, such as copying data across address-space boundaries, may be unnecessary in the kernel library.

In response to service requests, FPI libraries **404** and **405** map FPI functions into messages for communication from the user mode to the kernel mode. In one embodiment, the messages are kernel messages.

The service requests from other families are generated by plug-ins that make calls on libraries, such as FPI libraries **406** and **407**. In one embodiment, FPI libraries **406** and **407** map FPI functions into kernel messages and communicate those messages to FPI servers such as Y FPI server **409** and Z FPI server **410** respectively. Other embodiments may use mechanisms other than kernel messaging to communicate information.

In the example, the Z family **403** has both a user-mode library **405** and a kernel-mode library **407**. Therefore, the services of the Z family may be accessed from both the user mode and the kernel mode.

In response to service request messages, X FPI server **408**, Y FPI server **409** and Z FPI server **410** dispatch requests for services to their families. In one embodiment, each of FPI servers **408–410** receives a kernel message, maps the message into a FPI function called by the client, and then calls the function in the family implementation (**414–416**).

In one embodiment, there is a one-to-one correspondence between the FPI functions called by clients and the function called by FPI servers **408–410** as a result. The calls from FPI serves **408–410** are transferred via interfaces **411–413**. For instance, X interface **411** represents the interface presented to the FPI server **408** by the X family **414**. It is exactly the same as the FPI available to applications or other system software. The same is true of Y interface **412** and Z interface **413**.

The X family implementation **414** represents the family activation model that defines how requests communicated from server **408** are serviced by the family and plug-in(s). In one embodiment, X family implementation **414** comprises family code interfacing to plug-in code that completes the service requests from application **400** via server **408**. Similarly, the Y family implementation **415** and Z family implementation **416** define their family's plug-in activation models.

X plug-in **417**, Y plug-in **418** and Z plug-in **419** operate within the activation model mandated by the family and provide code and data exports. The required code and data exports and the activation model for each family of drivers is family specific and different.

Extending Family Programming Interfaces

A plug-in may provide a plug-in-specific interface that extends its functionality beyond that provided by its family. This is useful in a number of situations. For example, a block storage plug-in for a CD-ROM device may provide a block storage plug-in interface required of the CD-ROM device as well as an interface that allows knowledgeable application software to control audio volume and to play, pause, stop, and so forth. Such added capabilities require a plug-in-specific API.

If a device wishes to export extended functionality outside the family framework, a separate message port is provided by the device and an interface library for that portion of the device driver. FIG. **5** illustrates the extension of a family programming interface.

Referring to FIG. **5**, a plug-in module, Z plug-in **501**, extends beyond the Z family boundary to interface to family implementation D **502** as well. A plug-in that has an extended API offers features in addition to those available to clients through it's family's FPI. In order to provide extra services, the plug-in provides additional software shown in FIG. **5** as an interface library $Dlib_{\mu}$ **503**, the message port code D FPI server **504**, and the code that implements the extra features D **505**.

Sharing Code and Data Between Plug-ins

In one embodiment, two or more plug-ins can share data or code or both, regardless of whether the plug-ins belong to the same family or to different families. Sharing code or data is desirable when a single device is controlled by two or more families. Such a device needs a plug-in for each family. These plug-ins can share libraries that contain information about the device state and common code. FIG. **6** illustrates two plug-ins that belong to separate families and that share code and data.

Plug-ins can share code and data through shared libraries. Using shared libraries for plug-ins that share code or data allows the plug-ins to be instantiated independently without encountering problems related to simultaneous instantiation. Referring to FIG. **6**, the first plug-in **601** to be opened and initialized obtains access to the shared libraries. At this point, the first plug-in **601** does not share access. When the second plug-in **602** is opened and initialized, a new connection to the shared libraries is created. From that point, the two plug-ins contend with each other for access to the shared libraries.

Sharing code or data may also be desirable in certain special cases. For instance, two or more separate device drivers may share data as a way to arbitrate access to a shared device. An example of this is a single device that provides network capabilities and real time clock. Each of these functions belong to a distinct family but may originate in a single physical device.

Activation Models in the Present Invention

An activation model defines how the family is implemented and the environment within which plug-ins of the family execute. In one embodiment, the activation model of the family defines the tasking model a family uses, the opportunities the family plug-ins have to execute and the context of those opportunities (for instance, are the plug-ins called at task time, during privileged mode interrupt handling, and so forth), the knowledge about states and processes that a family and its plug-ins are expected to have, and the portion of the service requested by the client that is performed by the family and the portion that is performed by the plug-ins.

Each model provides a distinctly different environment for the plug-ins to the family, and different implementation options for the family software. Examples of activation models include the single-task model, the task-per-plug-in model, and the task-per-request model. Each is described in further detail below. Note that although three activation models are discussed, the choice of activation model is a design choice and different models may be used based on the needs and requirements of the family.

In one embodiment, the activation model uses kernel messaging as the interface between the FPI libraries that family clients link to and the FPI servers in order to provide the asynchronous or synchronous behavior desired by the family client. Within the activation model, asynchronous I/O requests are provided with a task context. In all cases, the implementation of the FPI server depends on the family activation model.

The choice of activation model limits the plug-in implementation choices. For example, the activation model defines the interaction between a driver's hardware interrupt level and the family environment in which the main driver runs. Therefore, plug-ins conform to the activation model employed by its family.

Single-Task Model

One of the activation models that may be employed by a family is referred to herein as the single-task activation model. In the single-task activation model, the family runs as a single monolithic task which is fed from a request queue and from interrupts delivered by plug-ins. Requests are delivered from the FPI library to an accept function that enqueues the request for processing by the family's processing task and wakes the task if it is sleeping. Queuing, synchronization, and communication mechanism within the family follow a set of rules specified by the family.

The interface between the FPI Server and a family implementation using the single-task model is asynchronous. Regardless of whether the family client called a function synchronously or asynchronously, the FPI server calls the family code asynchronously. The FPI server maintains a set

of kernel message IDs that correspond to messages to which the FPI server has not yet replied. The concept of maintaining kernel message IDs corresponding to pending I/O server request messages is well-known in the art;

Consider as an example family **700**, which uses the single-task activation model, shown in FIG. **7**. Referring to FIG. **7**, an application **710** is shown generating a service request to the family's APIs **711**. APIs **711** contain at least one library in which service requests are mapped to FPI functions. The FPI functions are forwarded to the family's FPI server **701**. FPI server **701** dispatches the FPI function to family implementation **703**, which includes various protocols and a network device driver that operate as a single task. Each protocol layer provides a different level of service.

The FPI server **701** is an accept function that executes in response to the calling client via the FPI library (not shown). An accept function, unlike a message-receive-based kernel task, is able to access data within the user and kernel bands directly. The accept function messaging model requires that FPI server **701** be re-entrant because the calling client task may be preempted by another client task making service requests.

When an I/O request completes within the family's environment, a completion notification is sent back to the FPI server **701**, which converts the completion notification into the appropriate kernel message ID reply. The kernel message ID reply is then forwarded to the application that generated the service request.

With a single-task model, the family implementation is insulated from the kernel in that the implementation does it not have kernel structures, IDs, or tasking knowledge. On the other hand, the relationship between FPI server **701** and family code **702** is asynchronous, and has internal knowledge of data structures and communication mechanisms of the family.

The single-task model may be advantageously employed for families of devices that have one of several characteristics: (1) each I/O request requires little effort of the processing unit. This applies not only to keyboard or mouse devices but also to DMA devices to the extent that the processing unit need only set up the transfer, (2) no more than one I/O request is handled at once, such that, for instance, the family does not allow interleaving of I/O requests. This might apply to sound, for example, or to any device for which exclusive reservation is required (i.e., where only one client can use a device at a time). The opposite of a shared resource. Little effort for the processor exists where the processor initiates an I/O request and then is not involved until the request completes, or (3) the family to be implemented provides its own scheduling mechanisms independent of the underlying kernel scheduling. This applies to the Unix™ stream programming model.

Task-Per-Plug-In Model

For each plug-in instantiated by the family, the family creates a task that provides the context within which the plug-in operates.

FIG. **8** illustrates the task-per-plug-in model. Referring to FIG. **8**, an application **801** generates service requests for the family, which are sent to FPI **802**. Using an FPI library, the FPI **802** generates a kernel message according to the family activation model **804** and a driver, such as plug-in driver **805**.

In one embodiment, the FPI server **803** is a simple task-based message-receive loop or an accept function. FPI server **803** receives requests from calling clients and passes those requests to the family code **804**. The FPI server **803** is responsible for making the data associated with a request available to the family, which in turn makes it available to the plug-in that services the request. In some instances, this responsibility includes copying or mapping buffers associated with the original request message to move the data from user address space to the kernel level area.

The family code **804** consists in part of one or more tasks, one for each family plug-in. The tasks act as a wrapper for the family plug-ins such that all tasking knowledge is located in the family code. A wrapper is a piece of code that insulates called code from the original calling code. The wrapper provides services to the called code that the called code is not aware of.

When a plug-in's task receives a service request (by whatever mechanisms the family implementation uses), the task calls its plug-in's entry points, waits for the plug-in's response, and then responds to the service request.

The plug-in performs the work to actually service the request. Each plug-in does not need to know about the tasking model used by the family or how to respond to event queues and other family mechanisms; it only needs to know how to perform its particular function.

For concurrent drivers, all queuing and state information describing an I/O request is contained within the plug-in code and data and within any queued requests. The FPI library forwards all requests regardless of the status of outstanding I/O requests to the plug-in. When the client makes a synchronous service request, the FPI library sends a synchronous kernel message. This blocks the requesting client, but the plug-in's task continues to run within its own task context. This permits clients to make requests of this plug-in even while another client's synchronous request is being processed.

In some cases of a family, a driver (e.g., **805**) can be either concurrent or nonconcurrent. Nevertheless, clients of the family may make synchronous and asynchronous requests, even though the nonconcurrent drivers can handle only one request at a time. The device manager FPI server **803** knows that concurrent drivers cannot handle multiple requests concurrently. Therefore, FPI server **803** provides a mechanism to queue client requests and makes no subsequent requests to a task until the task signals completion of an earlier I/O request.

When a client calls a family function asynchronously, the FPI library sends an asynchronous kernel message to the FPI server and returns to the caller. When a client calls a family function synchronously, the FPI library sends a synchronous kernel message to the FPI server and does not return to the caller until the FPI server replies to the message, thus blocking the caller's execution until the I/O request is complete.

In either case, the behaviors of the device manager FPI server **803** is exactly the same: for all incoming requests, it either queues the request or passes it to the family task, depending on whether the target plug-in is busy. When the plug-in signals that the I/O operation is complete, the FPI server **803** replies to the kernel message. When the FPI library receives the reply, it either returns to the synchronous client, unblocking its execution or it notifies the asynchronous client about the I/O completion.

The task-per-plug-in model is intermediate between the single-task and task-per-request models in terms of the number of tasks it typically uses. The task-per-plug-in model is advantageously used where the processing of I/O requests varies widely among the plug-ins.

Task-Per-Request Model

The task-per-request model shares the following characteristics with the two activation models already discussed:

(1) the FPI library to FPI server communication provides the synchronous or asynchronous calling behavior requested by family clients, and (2) the FPI library and FPI server use kernel messages to communicate I/O requests between themselves. However, in the task-per-request model, the FPI server's interface to the family implementation is completely synchronous.

In one embodiment, one or more internal family request server tasks, and, optionally, an accept function, wait for messages on the family message port. An arriving message containing information describing an I/O request awakens one of the request server tasks, which calls a family function to service the request. All state information necessary to handle the request is maintained in local variables. The request server task is blocked until the I/O request completes, at which time it replies to the kernel message from the FPI library to indicate the result of the operation. After replying, the request server task waits for more messages from the FPI library.

As a consequence of the synchronous nature of the interface between the FPI server and the family implementation, code calling through this interface remains running as a blockable task. This calling code is either the request server task provided by the family to service the I/O (for asynchronous I/O requests) or the task of the requester of the I/O (for certain optimized synchronous requests).

The task-per-request model is advantageously employed for a family where an I/O request can require continuous attention from the processor and multiple I/O requests can be in progress simultaneously. A family that supports dumb, high bandwidth devices is a good candidate for this model. In one embodiment, the file manager family uses the task-per-request model. This programming model requires the family plug-in code to have tasking knowledge and to use kernel facilities to synchronize multiple threads of execution contending for family and system resources.

Unless there are multiple task switches within a family, the tasking overhead is identical within all of the activation models. The shortest task path from application to I/O is completely synchronous because all code runs on the caller's task thread.

Providing at least one level of asynchronous call between an application and an I/O request results in better latency results from the user perspective. Within the file system, a task switch at a file manager API level allows a user-visible application, such as the Finder™, to continue. The file manager creates an I/O tasks to handle the I/O request, and that task is used via synchronous calls by the block storage and SCSI families to complete their part in I/O transaction processing.

The Device Registry of the Present Invention

The device registry of the present invention comprises an operating system naming service that stores system information. In one embodiment, the device registry is responsible for driver replacement and overloading capability so that drivers may be updated, as well as for supporting dynamic driver loading and unloading.

In one embodiment, the device registry of the present invention is a tree-structured collection of entries, each of which can contain an arbitrary number of name-value pairs called properties. Family experts examine the device registry to locate devices or plug-ins available to the family. Low-level experts, discussed below, describe platform hardware by populating the device registry with device nodes for insertion of devices that will be available for use by applications.

In one embodiment, the device registry contains a device subtree pertinent to the I/O architecture of the present invention. The device tree describes the configuration and connectivity of the hardware in the system. Each entry in the device tree has properties that describe the hardware represented by the entry and that contain a reference to the driver in control of the device.

Multiple low-level experts are used, where each such expert is aware of the connection scheme of physical devices to the system and installs and removes that information in the device tree portion of the device registry. For example a low-level expert, referred to herein as a bus expert or a motherboard expert, has specific knowledge of a piece of hardware such as a bus or a motherboard. Also, a SCSI bus expert scans a SCSI bus for devices, and installs an entry into the device tree for each device that it finds. The SCSI bus expert knows nothing about a particular device for which it installs an entry. As part of the installation, a driver gets associated with the entry by the SCSI bus expert. The driver knows the capabilities of the device and specifies that the device belongs to a given family. This information is provided as part of the driver or plug-in descriptive structure required of all plug-ins as part of their PPI implementation.

Low-level experts and family experts use a device registry notification mechanism to recognize changes in the system configuration and to take family-specific action in response to those changes.

An example of how family experts, low-level experts, and the device registry service operate together to stay aware of dynamic changes in system configuration follows: Suppose a motherboard expert notices that a new bus, a new network interface and new video device have appeared within the system. The motherboard expert adds a bus node, a network node, and a video node to the device tree portion of the device registry. The device registry service notifies all software that registered to receive notifications of these events.

Once notified that changes have occurred in the device registry, the networking and video family experts scan the device registry and notice the new entry belonging to their family type. Each of the experts adds an entry in the family subtree portion of the device registry.

The SCSI bus expert notices an additional bus, and probes for SCSI devices. It adds a node to the device registry for each SCSI device that it finds. New SCSI devices in the device registry result in perusal of the device registry by the block storage family expert. The block storage expert notices the new SCSI devices and loads the appropriate drivers, and creates the appropriate device registry entries, to make these volumes available to the file manager. The file manager receives notification of changes to the block storage family portion of the device registry, and notifies the Finder™ that volumes are available. These volumes then appear on the user's desktop.

Whereas, many alterations and modifications of the present invention will no doubt become apparent to a person of ordinary skill in the art after having read the foregoing description, it is to be understood that the particular embodiment shown and described by way of illustration are in no way to be considered limiting. Therefore, reference to the details of the various embodiments are not intended to limit the scope of the claims which themselves recite only those features regarded as essential to the invention.

Thus, a method and apparatus for handling I/O requests in a computer system has been described.

We claim:

1. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that include applications and an operating system; and

a processing unit coupled to the bus and running the operating system and applications by executing programming instructions, wherein an application has a first plurality of tailored distinct programming inter-

faces available to access a plurality of separate sets of computer system services provided through the operating system of the computer system via service requests.

2. The computer system defined in claim 3 wherein each of the first plurality of tailored distinct programming interfaces are tailored to a type of I/O service provided by each set of I/O services.

3. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that include applications and an operating system, wherein the operating system comprises a plurality of servers, and each of the first plurality of programming interfaces transfer service requests to one of the plurality of servers, wherein each of the plurality of servers responds to service requests from clients of the separate sets of I/O services; and

a processing unit coupled to the bus and running the operating system and applications by executing programing instructions, wherein an application has a first plurality of tailored distinct programming interfaces available to access a plurality of separate sets of I/O services provided through the operating system via service requests.

4. The computer system defined in claim 3 wherein service requests are transferred as messages in a messaging system.

5. The computer system defined in claim 4 wherein each of the plurality of servers supports a message port.

6. The computer system defined in claim 3 wherein at least one of the plurality of servers is responsive to service requests from applications and from at least one other set of I/O services.

7. The computer system defined in claim 3 wherein the operating system further comprises a plurality of activation models, wherein each of the plurality of activation models is associated with one of the plurality of servers to provide a runtime environment for the set of I/O services to which access is provided by said one of the plurality of servers.

8. The computer system defined in claim 7 wherein at least one instance of a service is called by one of the plurality of servers for execution in an environment set forth by one of the plurality of activation models.

9. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that comprise applications and an operating system;

a processing unit coupled to the bus and running the operating system and applications by executing programming instructions, wherein the operating system provides computer system services through a tailored distinct one of a plurality of program structures, each tailored distinct program structure comprising:

a first programming interface for receiving service requests for a set of computer system I/O services of a first type,

a first server coupled to receive service requests and to dispatch service requests to the computer system I/O services,

an activation model to define an operating environment in which a service request is to be serviced by the set of computer system I/O services, and

at least one specific instance of the set of computer system I/O services that operate within the activation model.

10. The computer system defined in claim 9 wherein the first programming interface is responsive to request from applications and from other program structures.

11. The computer system defined in claim 9 wherein the first programming interface comprises at least one library for converting functions into messages.

12. The computer system defined in claim 9 wherein the first server receives a message corresponding a service request from the first programming interface, maps the message into a function called by the client, and then calls the function.

13. The computer system defined in claim 9 wherein the message comprises a kernel message.

14. A computer system comprising:

a bus;

at least one memory coupled to the bus for storing data and programming instructions that comprise applications and an operating system;

a processing unit coupled to the bus and running the operating system and applications by executing programming instructions, wherein the operation system provides input/output (I/O) services through a tailored distinct one of plurality of program structures, each tailored distinct program structure comprising:

a first programming interface for receiving service requests for a set of I/O services of a first type,

a first server coupled to receive service requests and to dispatch service requests to the I/O services,

an activation model to define operating environment in which a service request is to be serviced by the set of I/O services, and

at least one specific instance of the set of I/0 services that operate within the activation model, wherein one of the said at least one specific instances comprises a service that accesses another program structure, and further wherein said one of said at least one specific instances communicates to said another program structure of a second type using a message created using a library sent to the server of said another program structure.

15. The computer system defined in claim 9 wherein two or more I/O services share code or data.

16. The computer system defined in claim 15 wherein said two or more I/O services are different types.

17. The computer system defined in claim 9 wherein the program structure further comprises a storage mechanism to maintain identification of available services to which access is provided via the first server.

18. A computer implemented method of accessing I/O services of a first type, said computer implemented method comprising the steps of:

generating a service request for a first type of I/O services;

a tailored distinct family server, operating in an operating system environment and dedicated to providing access to service requests for the first type of I/O service, receiving and responding to the service request based on an activation model specific to the first type of I/O services; and

a processor running an instance of the first type of I/O services that is interfaces to the file server to satisfy the service request.

19. The method defined in claim 18 wherein the service request is generated by an application.

20. The method defined in claim 18 wherein the service request is generated by an instance of an I/O service running in the operating system environment.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 5,915,131
DATED : June 22, 1999
INVENTOR(S) : Knight, et al.

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

In column 15 at line 14 delete "the" and insert -- a --
In column 15 at line 54 delete ":" and insert -- ; --
In column 16 at line 20 delete "operation" and
insert -- operating --
In column 16 at line 58 delete "interfaces" and
insert -- interfaced --

Signed and Sealed this

Eighteenth Day of January, 2000

*Attest:*

*Attesting Officer*

Q. TODD DICKINSON

*Commissioner of Patents and Trademarks*

# EXHIBIT 5

US005929852A

# United States Patent [19]

**Fisher et al.**

[11] **Patent Number: 5,929,852**

[45] **Date of Patent: Jul. 27, 1999**

[54] **ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM**

[75] Inventors: **Stephen Fisher**; **Michael A. Cleron**, both of Menlo Park; **Timo Bruck**, Mountian View, all of Calif.

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

[21] Appl. No.: **09/007,691**

[22] Filed: **Jan. 15, 1998**

**Related U.S. Application Data**

[63] Continuation of application No. 08/435,880, May 5, 1995, abandoned.

[51] **Int. Cl.**[6] ..... **G06T 1/00**

[52] **U.S. Cl.** ..... **345/335**

[58] **Field of Search** ..... 345/335, 339, 345/348, 356; 395/701, 200.47, 200.48, 680, 681, 682, 683, 684

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,202,828 | 4/1993 | Vertelney et al. | 395/936 X |
| 5,481,666 | 1/1996 | Nguyen et al. | 395/762 |
| 5,500,929 | 3/1996 | Dickinson | 395/160 |
| 5,530,852 | 6/1996 | Meske, Jr. et al. | 395/600 |
| 5,537,546 | 7/1996 | Sauter | 395/762 |
| 5,548,722 | 8/1996 | Jalalian et al. | 395/200.1 |
| 5,574,862 | 11/1996 | Marianetti, II | 395/200.08 |
| 5,659,791 | 8/1997 | Nakajima et al. | 345/302 |
| 5,724,506 | 3/1998 | Cleron et al. | 395/200.01 |
| 5,724,556 | 3/1998 | Souder et al. | 345/335 X |
| 5,781,189 | 7/1998 | Holleran et al. | 345/335 |
| 5,819,090 | 10/1998 | Wolf et al. | 345/335 X |

FOREIGN PATENT DOCUMENTS

| | | |
|---|---|---|
| WO A9107719 | 5/1991 | WIPO . |

OTHER PUBLICATIONS

Develop, The Apple Technical Journal, "Building an OpenDoc Part Handler", Issue 19, Sep., 1994, pp. 6–16.

Baker, S. "Mosaic—Surfing at Home and Abroad," Proceedings ACM SIGUCCS User Services Conference XXII, Oct. 16–19, 1994, pp. 159–163.

PCT International Search Report dated Oct. 22, 1996 in corresponding PCT Case No. PCT/US96/06376.

MacWeek, Nov. 7, 1994, vol. 8, No. 44, Cyberdog to Fetch Internet Resources for OpenDoc APPS, Robert Hess.

Opinion, MacWeek Nov. 14, 1994, The Second Decade, Cyberdog Could Be a Breakthrough if it's Kept on a Leash, Henry Norr.

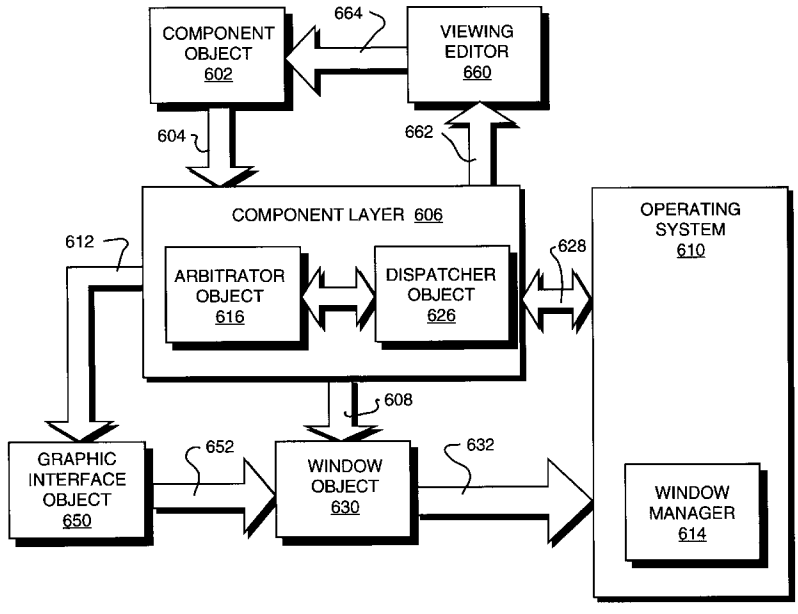
*Primary Examiner*—Joseph H. Feild

*Attorney, Agent, or Firm*—Cesari & McKenna, LLP

[57] **ABSTRACT**

A network-oriented component system efficiently accesses information from a network resource located on a computer network by creating an encapsulated network entity that contains a reference to that resource. The encapsulated entity is preferably implemented as a network component stored on a computer remotely displaced from the referenced resource. In addition, the encapsulated entity may be manifested as a visual object on a graphical user interface of a computer screen. Such visual manifestation allows a user to easily manipulate the entity in order to display the contents of the resource on the screen or to electronically forward the entity over the network.

**20 Claims, 14 Drawing Sheets**



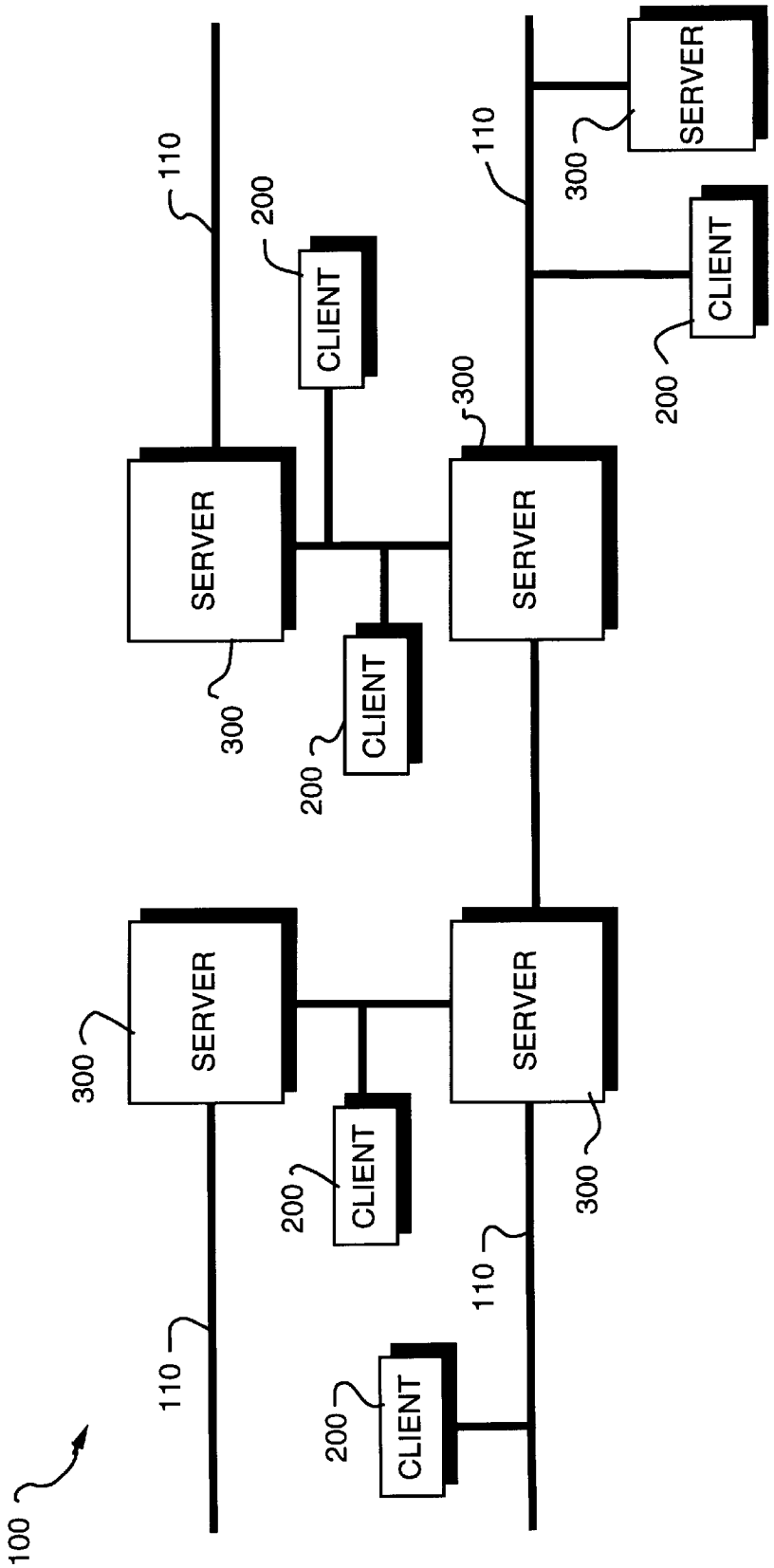

100
110
110
110
110
200
200
200
200
300
300
300
300
300
CLIENT
CLIENT
CLIENT
CLIENT
SERVER
SERVER
SERVER
SERVER
SERVER

FIG. 1

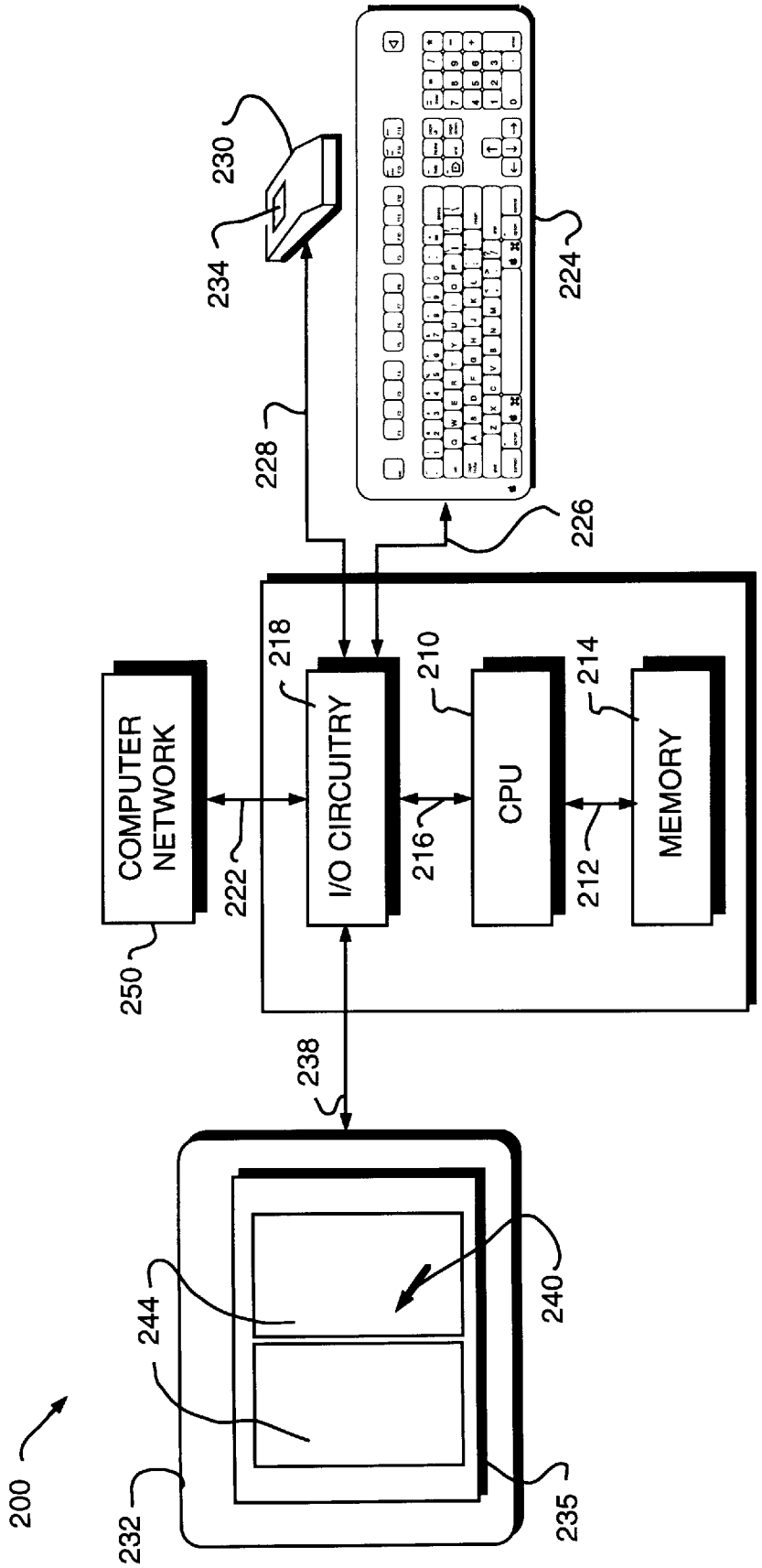

200
232
235
240
244
238
250
COMPUTER NETWORK
222
I/O CIRCUITRY
218
216
CPU
210
212
MEMORY
214
228
226
224
234
230

FIG. 2

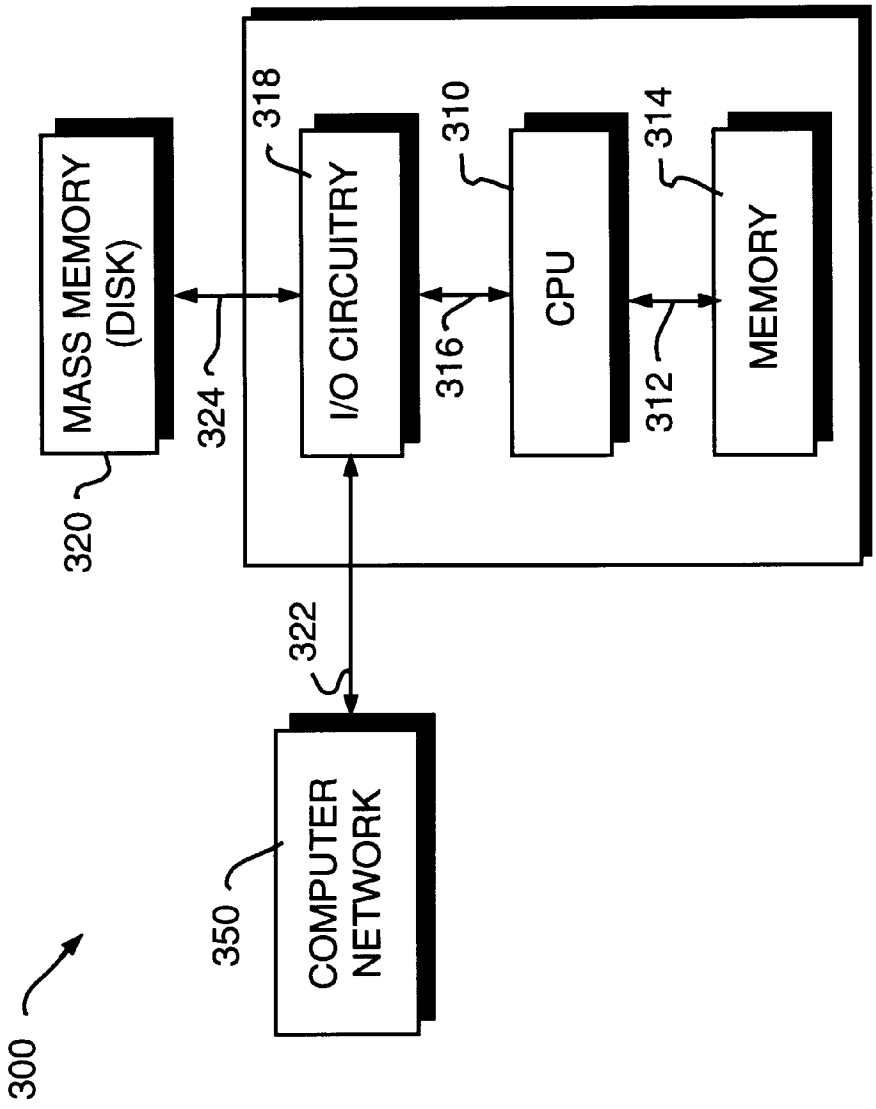

MASS MEMORY (DISK)
320
324
318
I/O CIRCUITRY
316
310
CPU
312
314
MEMORY
322
350
COMPUTER NETWORK
300

FIG. 3

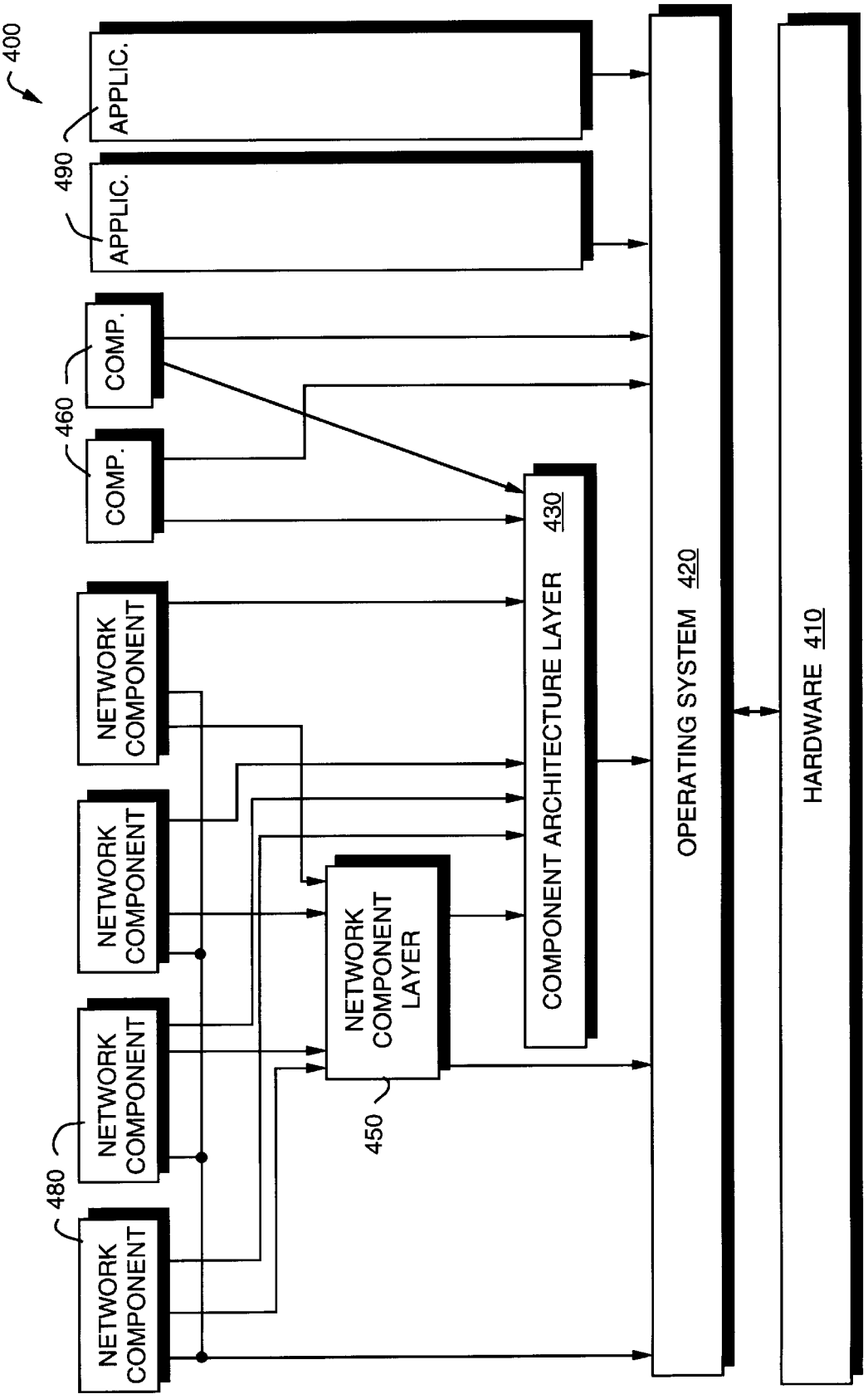

400
APPLIC.
490
APPLIC.
COMP.
460
COMP.
NETWORK COMPONENT
NETWORK COMPONENT
NETWORK COMPONENT
480
NETWORK COMPONENT
NETWORK COMPONENT LAYER
450
COMPONENT ARCHITECTURE LAYER 430
OPERATING SYSTEM 420
HARDWARE 410

FIG. 4

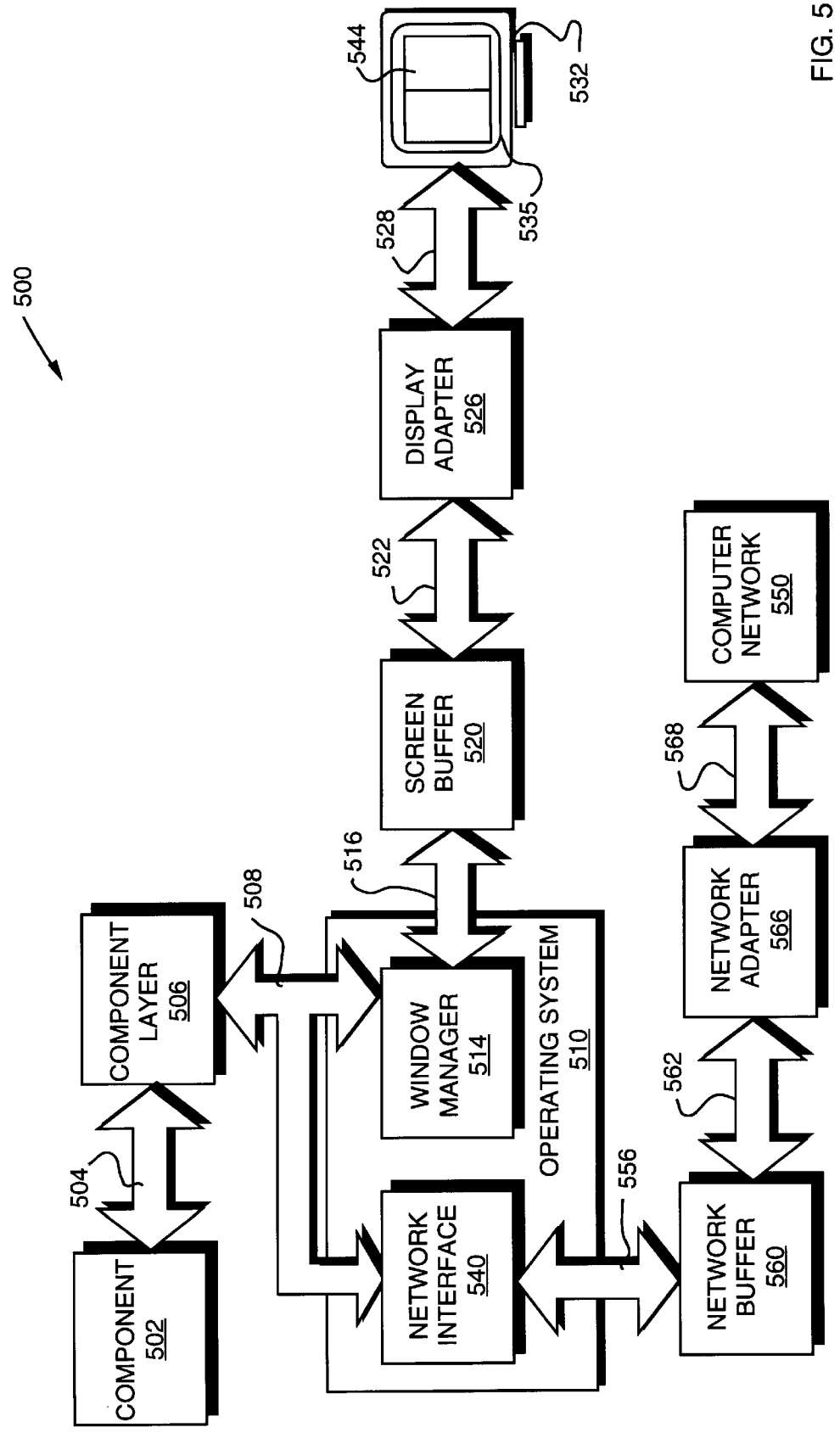

500
COMPONENT 502
504
COMPONENT LAYER 506
508
516
WINDOW MANAGER 514
NETWORK INTERFACE 540
OPERATING SYSTEM 510
556
SCREEN BUFFER 520
522
DISPLAY ADAPTER 526
528
544
532
535
NETWORK BUFFER 560
562
NETWORK ADAPTER 566
568
COMPUTER NETWORK 550

FIG. 5

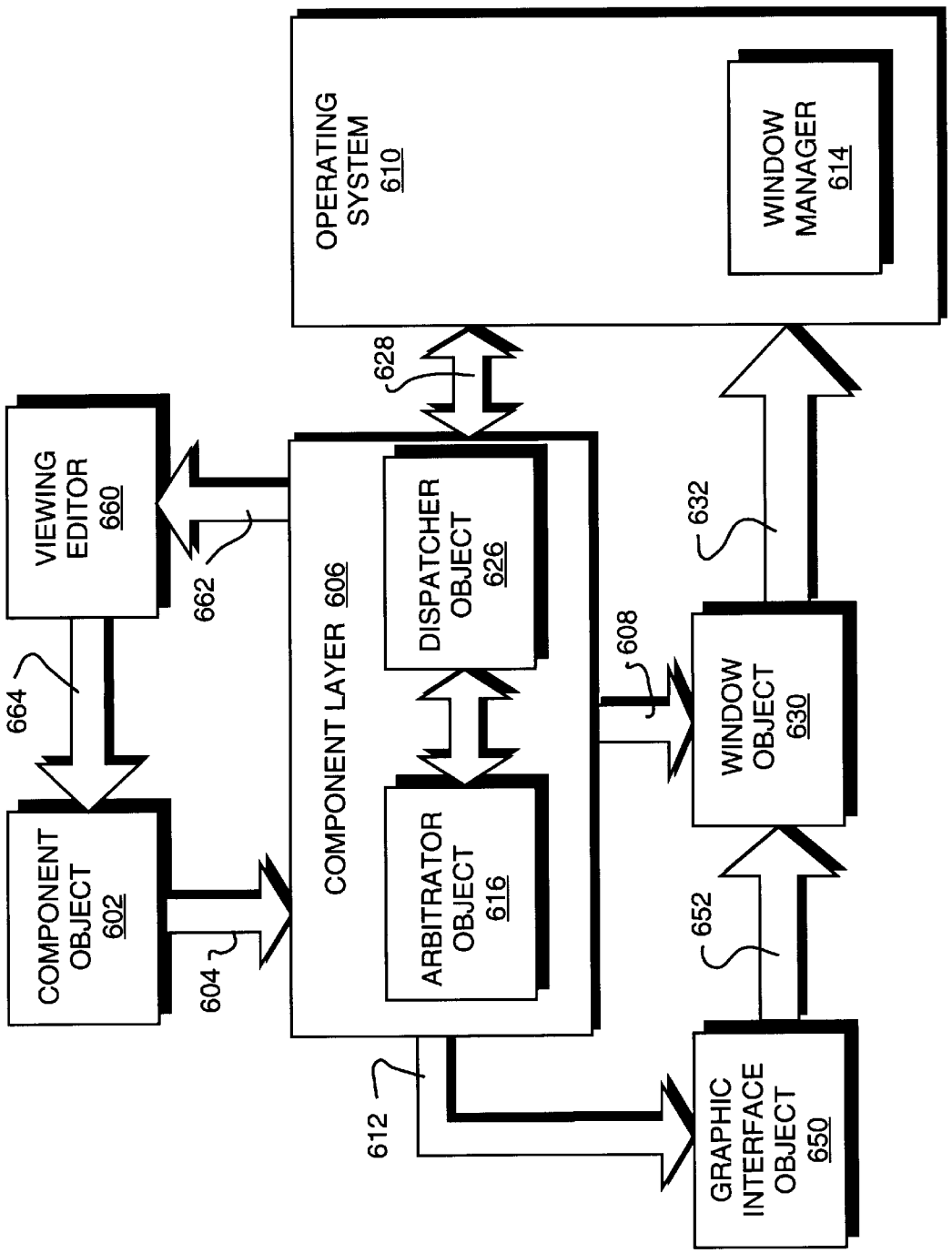

OPERATING SYSTEM 610
WINDOW MANAGER 614
628
VIEWING EDITOR 660
662
664
COMPONENT LAYER 606
DISPATCHER OBJECT 626
ARBITRATOR OBJECT 616
608
632
WINDOW OBJECT 630
COMPONENT OBJECT 602
604
652
612
GRAPHIC INTERFACE OBJECT 650

FIG. 6

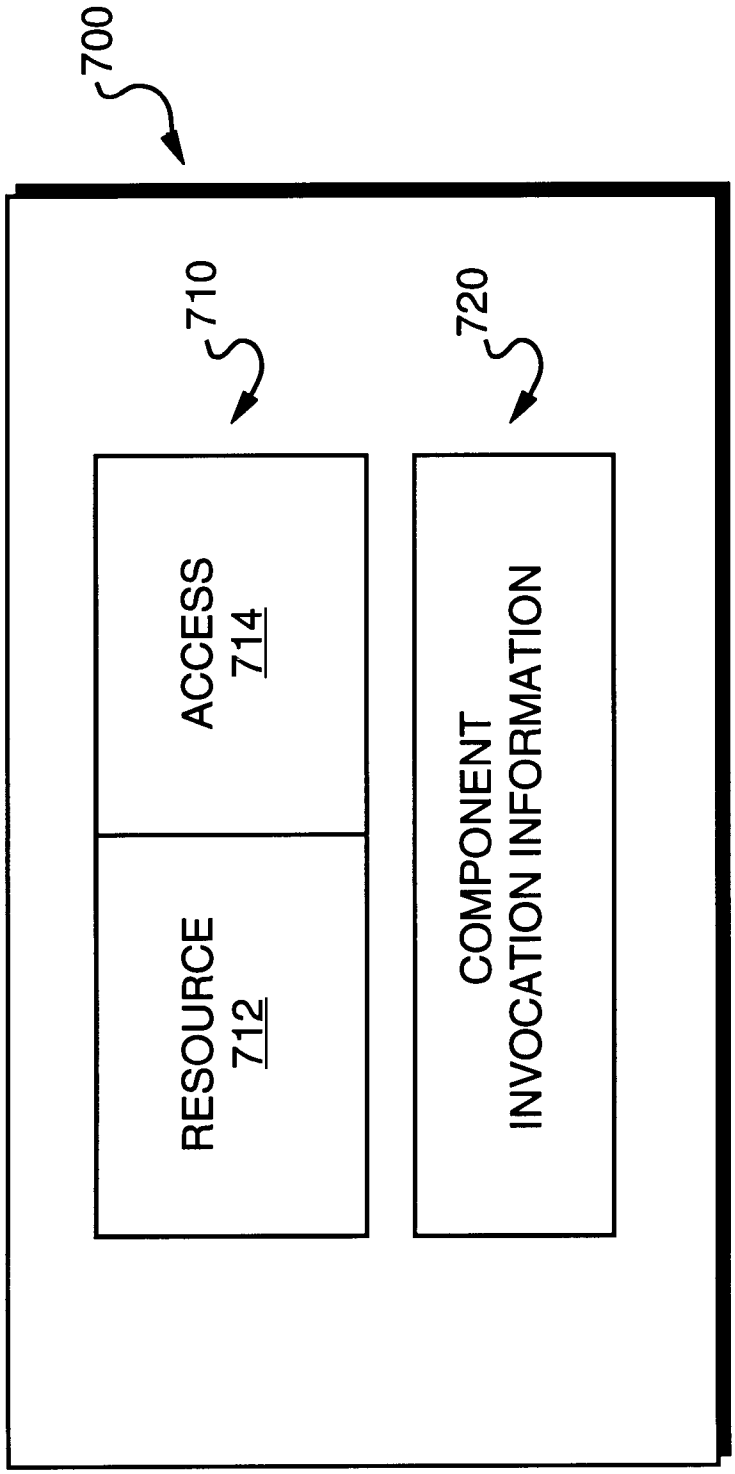

700
710
720
ACCESS
714
RESOURCE
712
COMPONENT
INVOCATION INFORMATION

FIG. 7

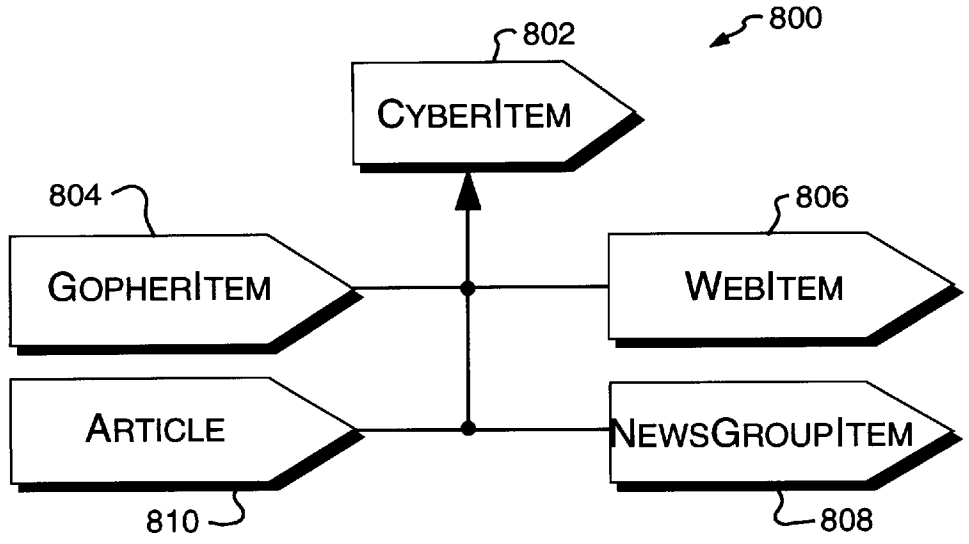

800
802
CYBERITEM
804
GOPHERITEM
806
WEBITEM
ARTICLE
NEWSGROUPITEM
810
808

FIG. 8

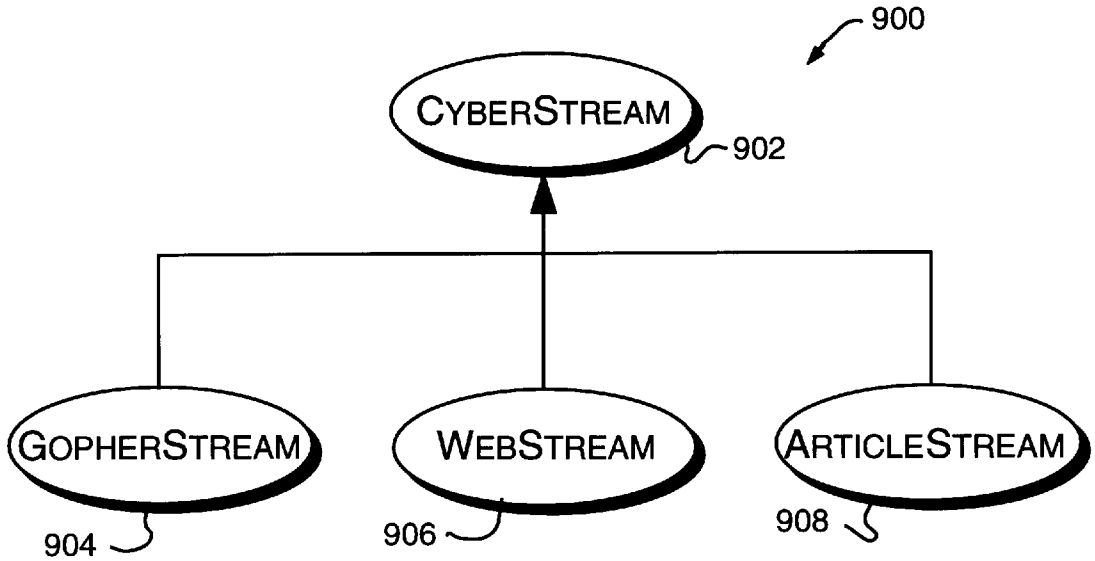

900
CYBERSTREAM
902
GOPHERSTREAM
WEBSTREAM
ARTICLESTREAM
904
906
908

FIG. 9

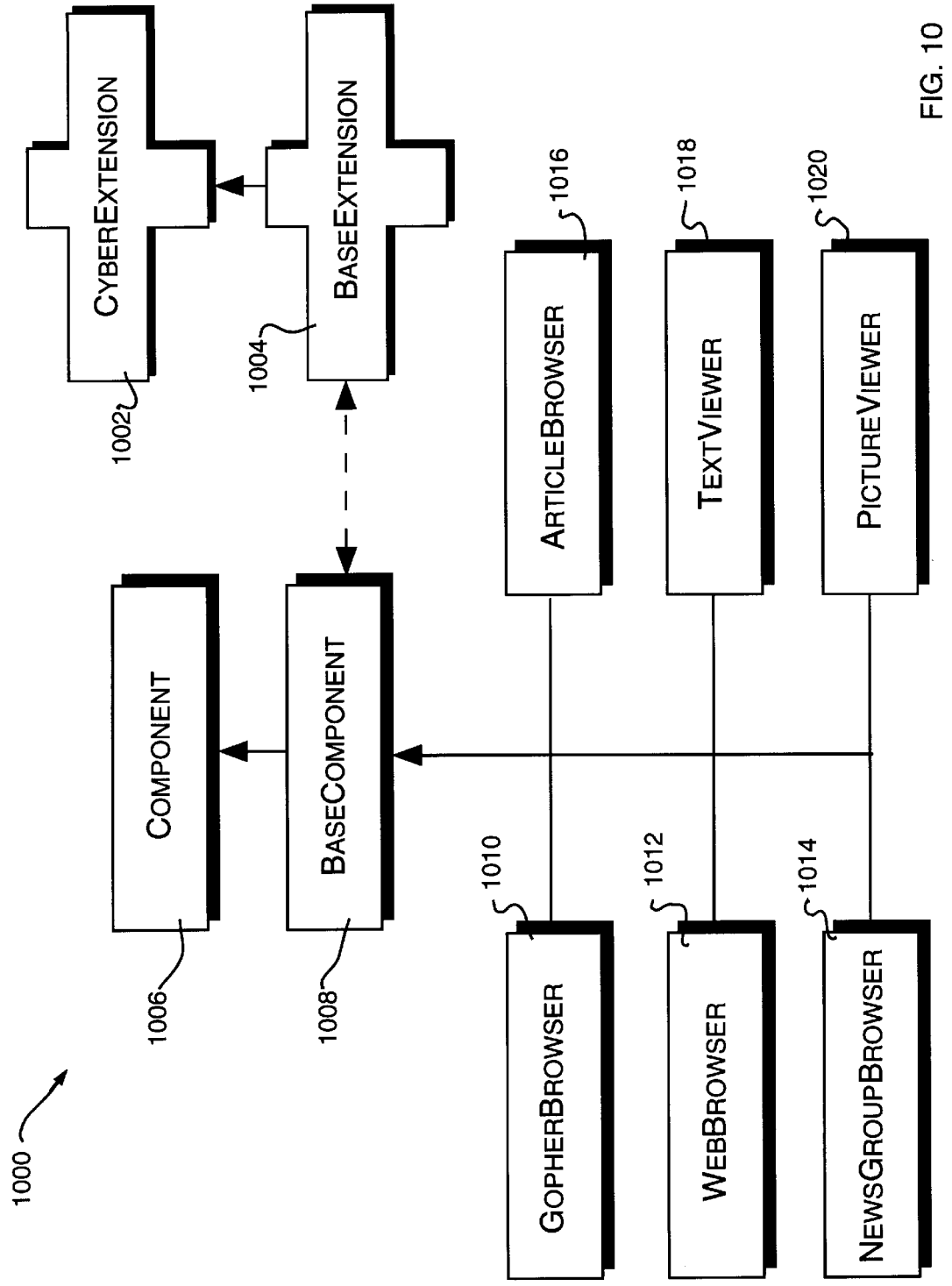

CYBEREXTENSION
BASEEXTENSION
1002
1004
ARTICLEBROWSER
TEXTVIEWER
PICTUREVIEWER
1016
1018
1020
COMPONENT
BASECOMPONENT
1006
1008
GOPHERBROWSER
WEBBROWSER
NEWSGROUPBROWSER
1010
1012
1014
1000

FIG. 10

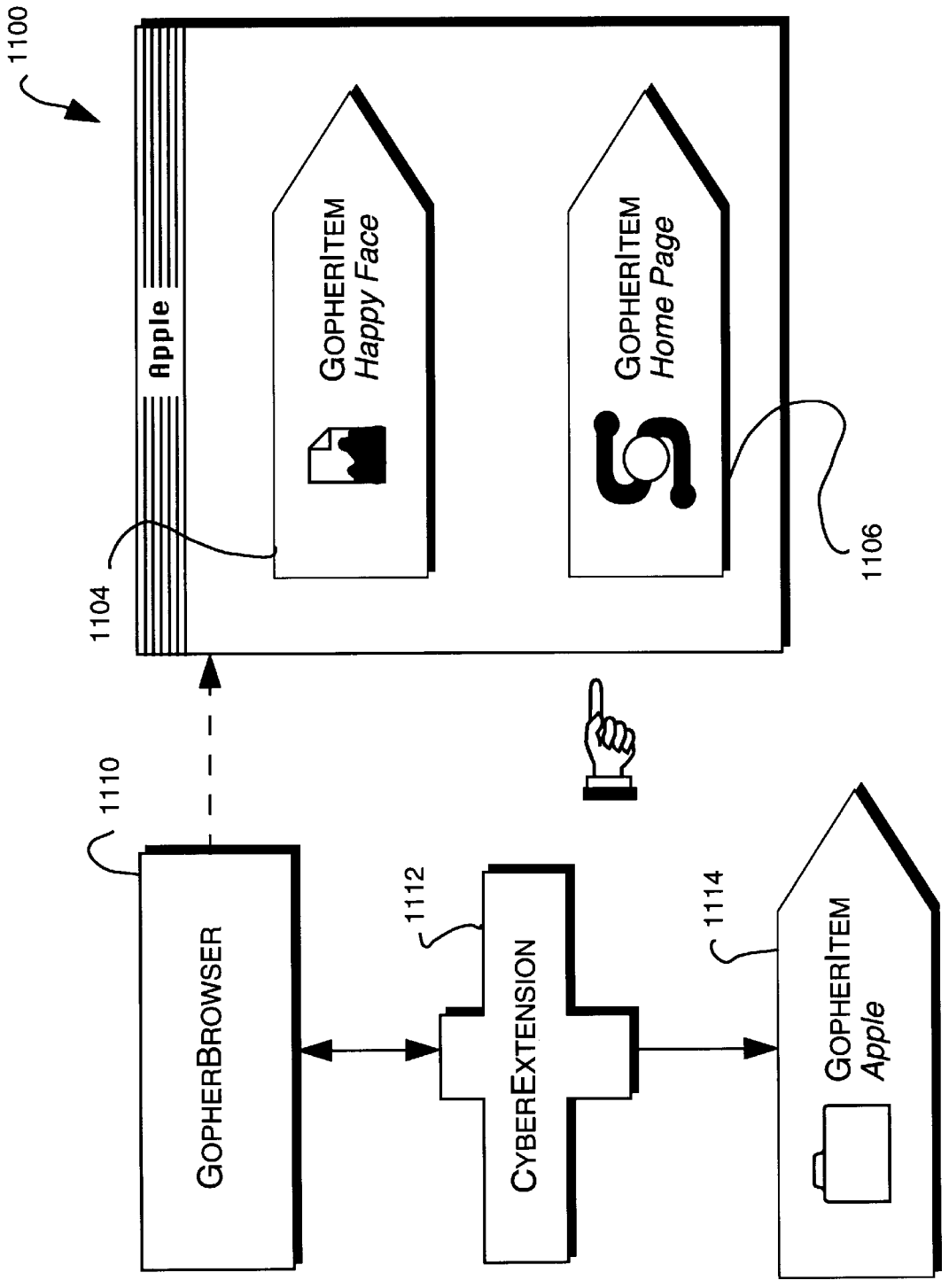

1100
Apple
GopherItem
Happy Face
GopherItem
Home Page
1104
1106
1110
1112
1114
GopherBrowser
CyberExtension
GopherItem
Apple

FIG. 11A

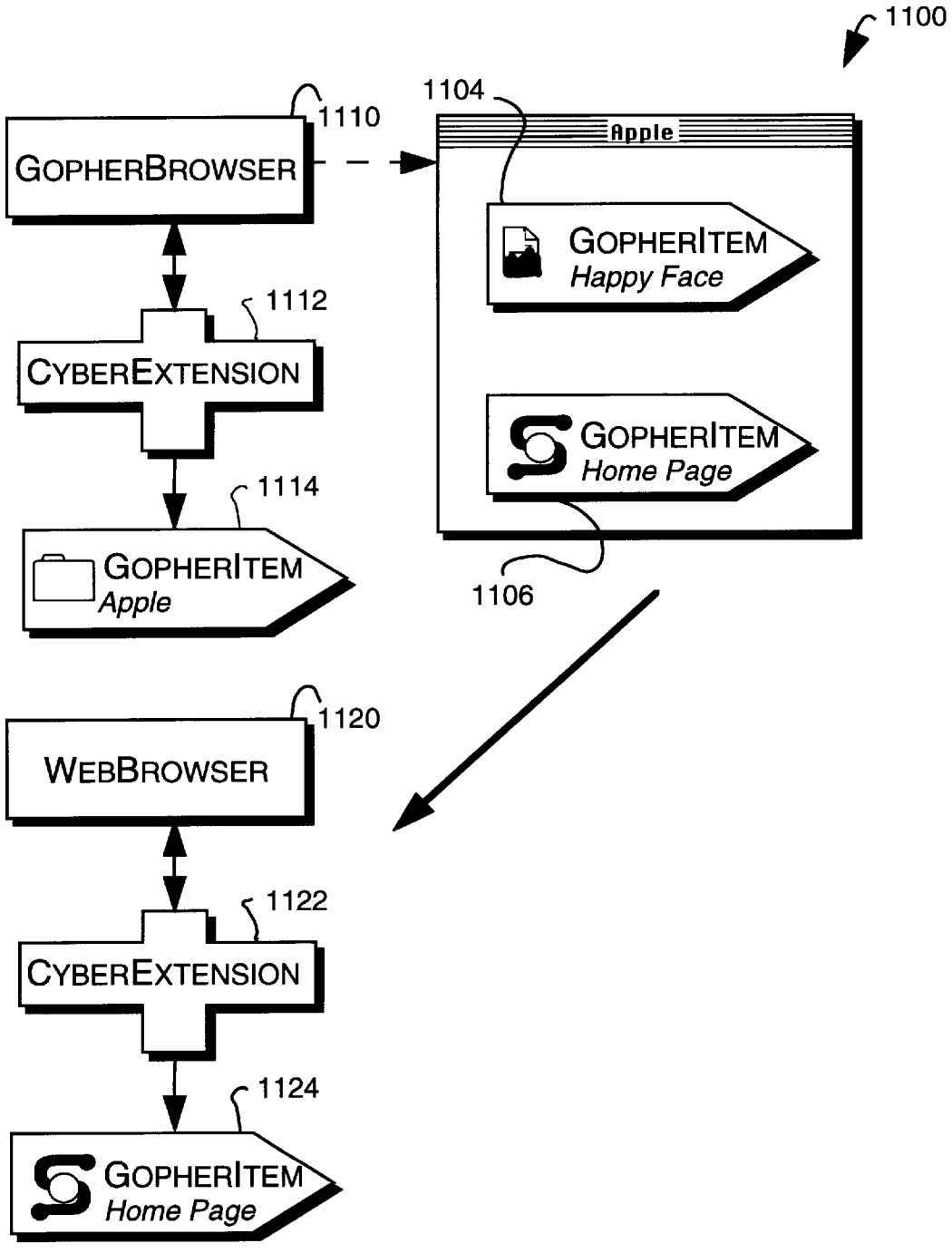

1100
1104
1110
GOPHERBROWSER
Apple
GOPHERITEM
Happy Face
1112
CYBEREXTENSION
GOPHERITEM
Home Page
1114
GOPHERITEM
Apple
1106
1120
WEBBROWSER
1122
CYBEREXTENSION
1124
GOPHERITEM
Home Page

FIG. 11B

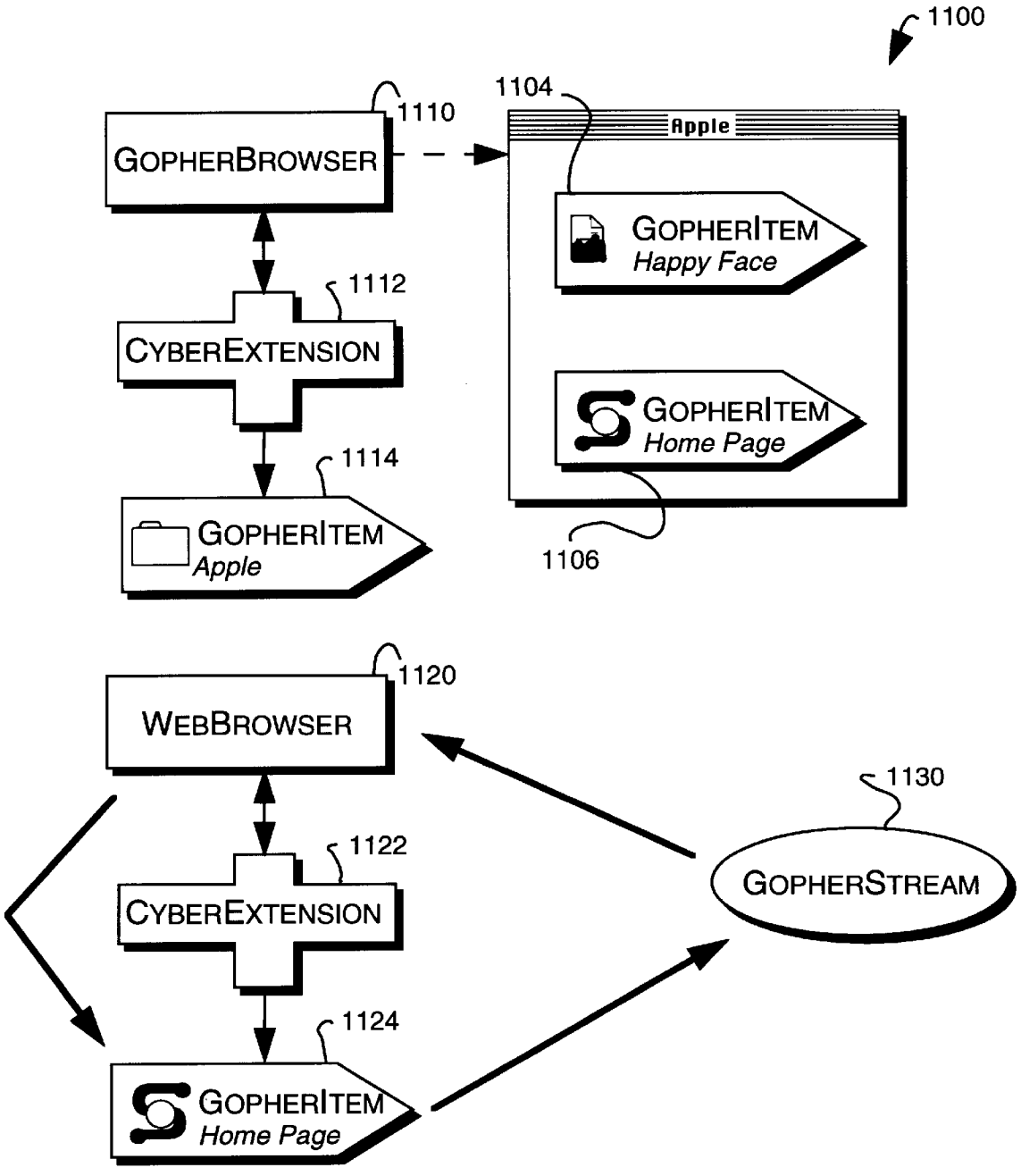

1100
1104
1110
GopherBrowser
Apple
GopherItem
Happy Face
1112
CyberExtension
GopherItem
Home Page
1114
GopherItem
Apple
1106
1120
WebBrowser
1130
1122
CyberExtension
GopherStream
1124
GopherItem
Home Page

FIG. 11C

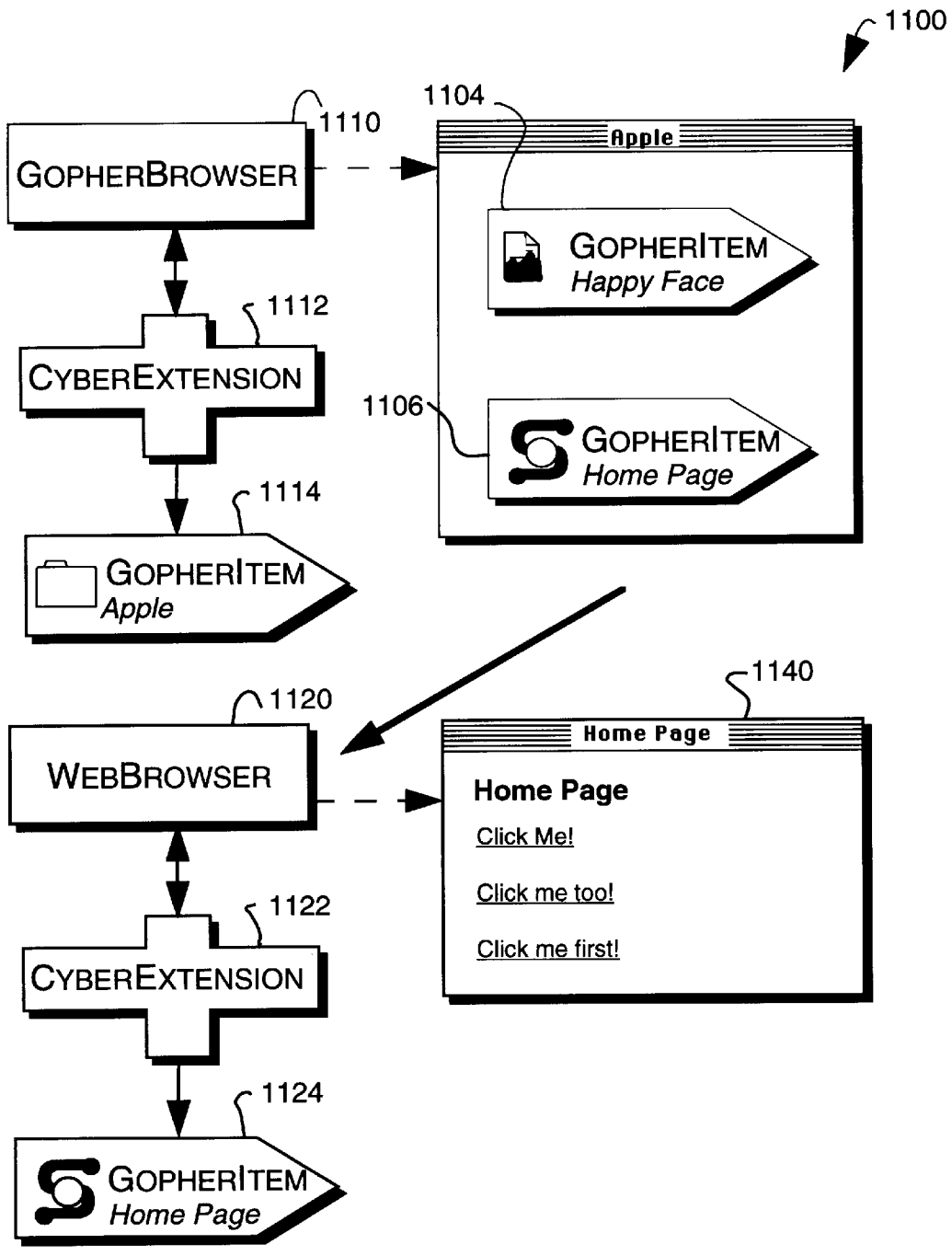

1100
1104
1110
GOPHERBROWSER
Apple
GOPHERITEM
Happy Face
1112
CYBEREXTENSION
1106
GOPHERITEM
Home Page
1114
GOPHERITEM
Apple
1140
1120
Home Page
WEBBROWSER
Home Page
Click Me!
Click me too!
Click me first!
1122
CYBEREXTENSION
1124
GOPHERITEM
Home Page

FIG. 11D

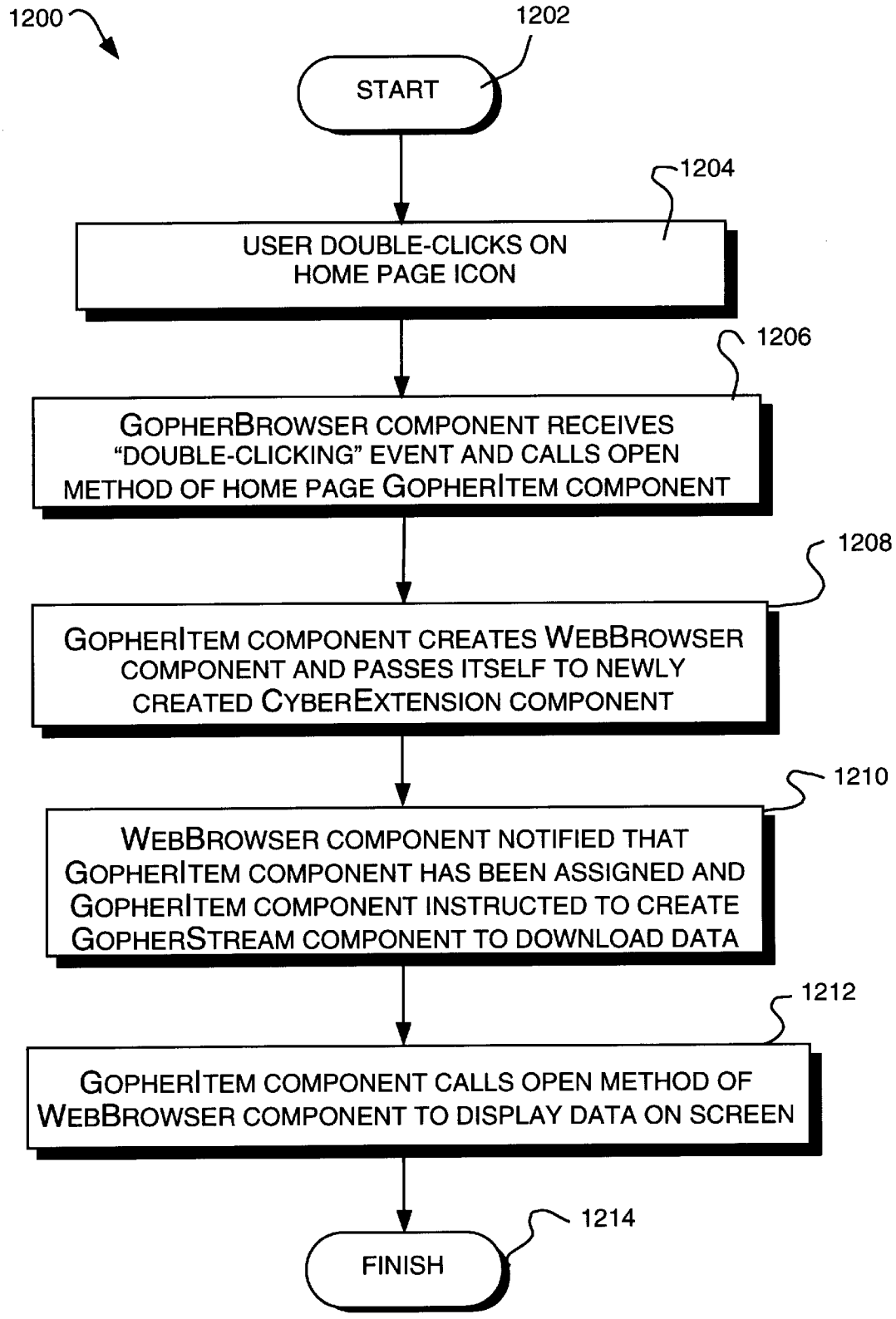

1200
1202
START
1204
USER DOUBLE-CLICKS ON HOME PAGE ICON
1206
GOPHERBROWSER COMPONENT RECEIVES "DOUBLE-CLICKING" EVENT AND CALLS OPEN METHOD OF HOME PAGE GOPHERITEM COMPONENT
1208
GOPHERITEM COMPONENT CREATES WEBBROWSER COMPONENT AND PASSES ITSELF TO NEWLY CREATED CYBEREXTENSION COMPONENT
1210
WEBBROWSER COMPONENT NOTIFIED THAT GOPHERITEM COMPONENT HAS BEEN ASSIGNED AND GOPHERITEM COMPONENT INSTRUCTED TO CREATE GOPHERSTREAM COMPONENT TO DOWNLOAD DATA
1212
GOPHERITEM COMPONENT CALLS OPEN METHOD OF WEBBROWSER COMPONENT TO DISPLAY DATA ON SCREEN
1214
FINISH

FIG. 12

# ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM

This application is a continuation of U.S. patent application Ser. No. 08/435,880, filed May 5, 1995, now abandoned.

## CROSS-REFERENCE TO RELATED APPLICATIONS

This invention is related to the following copending U.S. patent applications:

U.S. patent application Ser. No. 08/435,377, titled EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM;

U.S. Pat. No. 5,784,619 issued Jul. 21, 1998, 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. Pat. No. 5,724,506, issued Mar. 3, 1998, titled REPLACEABLE AND EXTENSIBLE CONNECTION DIALOG COMPONENT OF A NETWORK COMPONENT SYSTEM; and

U.S. Pat. No. 5,781,189 issued Jul. 14, 1998, titled EMBEDDING INTERNET BROWSER/BUTTONS WITHIN COMPONENTS OF A NETWORK COMPONENT SYSTEM, each of which was filed 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 and tools 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 for 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 netnews 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 $\mu$-law sound files.

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 service 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 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 HTTP, 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 application. Thus, if a user operating within the Netscape application "opens" a MPEG movie, that browsing application 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 words, 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 or 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 content 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.

To remotely access information stored on a resource of the network, the user typically invokes a service configured to operate in accordance with a protocol for accessing the resource. In particular, the user types an explicit destination address command that includes a uniform resource locator (URL). The URL is a rather long (approximately 50 character) address pointer that identifies both a network resource and a means for accessing that resource. The following is an example of a hypothetical URL address pointer to a remote resource on a Web server:

http:/aaaa.bbb.cc/hypertext/DdddEeeee/WWW/Ffffffff.html

It is apparent that having to type such long destination address pointers can become quite burdensome for users that frequently access information from remote resources.

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 users with a simple means for remotely accessing information stored on resources connected to computer networks.

## SUMMARY OF THE INVENTION

Briefly, the invention comprises a network-oriented component system for efficiently accessing information from a network resource located on a computer network by creating an encapsulated network entity that contains a reference to that resource. The encapsulated entity is preferably implemented as a network component stored on a computer remotely displaced from the referenced resource. In addition, the encapsulated entity may be manifested as a visual object on a graphical user interface of a computer screen. Such visual manifestation allows a user to easily manipulate the entity in order to display the contents of the resource on the screen or to electronically forward the entity over the network.

In the illustrative embodiment of the invention, the reference to the network resource is preferably a "pointer", such as a uniform resource locator (URL), that identifies the network address of the resource, e.g., a Gopher browser or a Web page. In addition to storing the pointer, the encapsulated entity also contains information for invoking appropriate network components needed to access the resource. Communication among the network components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a cooperating architecture allows the encapsulated entity and network components to "transport" the user to the network location of the remote resource.

Specifically, the encapsulated entity component is an object of the network-oriented component system that is preferably embodied as a customized framework having a set of interconnected abstract classes. A CyberItem class defines the encapsulated entity object which interacts with other objects of the network system to remotely access information from the referenced resource. Since these objects are integral elements of the cooperating component architecture, any type of encapsulated network entity may be developed with consistent behaviors, i.e., these entities may be manifested as visual objects that can be distributed and manipulated iconically.

Advantageously, the inventive encapsulation technique described herein provides a user with a simple means for accessing information on computer networks.

## 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 computer, such as a personal computer, on which the invention may advantageously operate;

FIG. **3** is a block diagram of the 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 software 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 schematic diagram of an illustrative encapsulated network entity object in accordance with the invention;

FIG. **8** is a simplified class heirarchy diagram illustrating a base class CyberItem, and its associated subclasses, used to construct network component objects in accordance with the invention;

FIG. **9** is a simplified class heirarchy diagram illustrating a base class CyberStream, and its associated subclasses, in accordance with the invention;

FIG. **10** is a simplified class hierarchy diagram illustrating a base class CyberExtension, and its associated subclasses, in accordance with the present invention;

FIGS. **11A–11D** are highly schematized diagrams illustrating the interactions between the network component objects, including the encapsulated network entity object of FIG. **7**; and

FIG. **12** is an illustrative flowchart of the sequence of steps involved in invoking, and accessing, information from a referenced network resource.

## 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 networks **250**, via a bidirectional bus **222** and to cursor/pointer control devices, such as a 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. 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 network. 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® 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 component 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 creating encapsulated entity components that contain references to network resources located on computer networks. As described further herein, communication among these components is achieved through novel application programming interfaces (APIs) 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. The network component layer **450** extends the functionality of the underlying component architecture layer **430** by defining network-oriented components **480** that seamlessly integrate with these components **460** to provide basic tools for efficiently accessing information from network resources located on, e.g., servers coupled to the computer networks.

FIG. **4** also illustrates the relationship of applications **490** to the elements of the document 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 and communicate 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 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 create encapsulated entities that contain references to network resources located on computer networks, such as the Internet. The encapsulated entities are manifested as visual objects to a user via a window environment, such as the graphical user interface provided 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 component 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 level 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 component architecture layer **430** to provide a cooperating architecture that allows any encapsulated entity and network component to "transport" the user to the network location of a remote resource; 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 formating 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 illustrated 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 manager itself may be an object which is

created when the operating system is started. Specifically, the component layer creates window objects **630** that cause 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 compound document having mixed data types and formats. Many different viewing editors may work together to display, or modify, the data contents of the document. 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 modification. Consequently, a prior art approach has been to provide a developer with a set of predefined, 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 provides 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 frameworks 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 frameworks. 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 greatly enhancing the ability of a user to efficiently access information from a network resource on computer networks by creating an encapsulated entity that contains a reference to that resource. The encapsulated entity is preferably implemented as a network component of the system and stored as a visual object, e.g., an icon, for display on a graphical user interface. Such visual display allows a user to easily manipulate the entity component to display the contents of the resource on a computer screen or to electronically forward the entity over the networks.

Furthermore, the reference to the network resource is a pointer that identifies the network address of the resource, e.g., a Gopher browser, a Web page or an E-mail message. FIG. **7** is a schematic diagram of an illustrative encapsulated network entity object **700** containing a pointer **710**. In one embodiment of the invention, the pointer may be a uniform resource locator (URL) having a first portion **712** that identifies the particular network resource and a second portion **714** that specifies the means for accessing that

resource. More specifically, the URL is a string of approximately 50 characters that describes the protocol used to address the target resource, the server on which the resource resides, the path to the resource and the resource filename. It is to be understood, however, that other representations of a "pointer" are included within the principles of the invention, e.g., a Post Office Protocol (POP) account and message identification (ID).

In addition to storing the pointer, the encapsulated entity also contains information **720** for invoking appropriate network components needed to access the resource. Communication among these network components is achieved 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. From an implementation viewpoint, the objects can be subclassed and can inherit from base classes to build customized components that 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 the customized network components. These abstract classes include CyberItem, CyberStream and CyberExtension and the objects they define are used to build the novel network components. A description of these abstract classes is provided in copending and commonly assigned U.S. patent application titled Extensible, Replaceable Network Component System, filed May 5, 1995, which application is incorporated by reference as though fully set forth herein.

Specifically, the CyberItem class defines the encapsulated entity object which interacts with objects defined by the other abstract classes of the network system to "transport" the user to the network location, i.e., remotely access information from the referenced resource and display that information to the user at the computer. Since these objects are integral elements of the cooperating component architecture, any type of encapsulated network entity may be developed with consistent behaviors, i.e., these entities may be manifested as visual objects that can be distributed and manipulated iconically.

FIG. **8** illustrates a simplified class hierarchy diagram **800** of the base class CyberItem **802** used to construct the encapsulated network entity component object **602**. In accordance with the illustrative embodiment, 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 subclass GopherItem **804** is derived from the CyberItem class **802** and encapsulates a network entity component object representing a "thing in Gopher space", such as a Gopher directory.

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. Accordingly, methods associated with the CyberItem base class for, e.g., instructing an object to open itself, are assumed by the subclasses to allow the network components to display CyberItem objects in a consistent manner.

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. **9** illustrates a simplified class hierarchy diagram **900** of the base class CyberStream **902** which 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. Specifically, a CyberStream object contains the software commands necessary to create a "data stream" for transfering information from one object to another. According to the invention, a GopherStream subclass **904** is derived from the CyberStream base class **902** and encapsulates a network object that implements the Gopher protocol.

FIG. **10** is a simplified class hierarchy diagram **1000** of the base class CyberExtension **1002** which represents additional behaviors provided to components of the underlying software component architecture. For example, CyberExtension objects add functionality to, and extend the APIs of, existing components so that they may communicate with the novel network components, such as the encapsulated entity objects. As a result, the CyberExtension base class **1002** operates in connection with a Component base class **1006** through their respective subclasses BaseExtension **1004** and BaseComponent **1008**.

CyberExtension objects are used by components that display the contents of CyberItem objects; this includes browser-like components, such as a Gopher browser or Web browser, along with viewer-like components, such as JPEG, MPEG or text viewers. 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 **1010** may be used to construct a Gopher-like network browsing component and the class WebBrowser **1012** may be used to construct a Web-like network browsing component.

FIGS. **11A–11D** are highly schematized diagrams illustrating the interactions between the novel network-oriented components, including the encapsulated (CyberItem) network entity component according to the invention. It is to be understood that the components described herein are objects constructed from the interconnected abstract classes. In general, a user has "double clicked" on an icon of a graphical user interface **1100** displayed on a computer screen. The icon represents, e.g., a Gopher directory displayed in a Gopher browser application. Initially, a GopherBrowser component **1110** displays two icons representing CyberItem components, the icons labeled (GopherItem) Happy Face **1104** and (GopherItem) Home Page **1106**. These latter components represent the contents of a Gopher directory labeled (GopherItem) Apple **1114**.

In FIG. **11A**, the left side of the diagram illustrates a GopherBrowser component **1110** that is displayed on the computer screen, i.e., the right side of the diagram. The GopherBrowser component has a CyberExtension component **1112** which keeps track of the GopherItem components. When the user double clicks on the Home Page GopherItem icon **1106**, the GopherBrowser component **1110** receives this event and issues a call to an "Open" method of a Home Page GopherItem component; this call instructs the GopherItem component **1106** to open itself.

Specifically, and referring to FIG. **11B**, the GopherItem component **1106** creates a component of the appropriate type to display itself. For this example, the GopherItem preferably creates a WebBrowser component **1120**. Once created, the WebBrowser component further creates a CyberExtension component **1122** for storing the Home Page GopherItem component (now shown at **1124**). In accordance with the invention, the Home Page GopherItem component is a network entity containing a pointer that points to the network address of a Gopher server storing the appropriate Web page.

In FIG. **11**C, the CyberExtension component **1122** then notifies the WebBrowser component **1120** that it has been assigned a GopherItem component **1124** to display. The WebBrowser component **1120** calls a method CreateCyberStream of the GopherItem to create a GopherStream component **1130** for downloading the appropriate data. Thereafter, the WebBrowser component **1120** begins asychronously downloading an HTML document from the appropriate Gopher server (not shown).

Control of the execution of this process then returns to the GopherItem component **1124** in FIG. **11**D. This component, in turn, issues a call to an Open method of the WebBrowser component **1120**, which causes the downloaded HTML document to appear on the screen (now shown at **1140**). For a further understanding of the invention, FIG. **12** provides an illustrative flowchart **1200** of the sequence of steps involved in invoking, and accessing, information from a referenced network resource, as described above.

In summary, the network-oriented component system provides a customizable framework that enables a user to create an encapsulated entity containing a reference to a network resource on a computer network. Advantageously, the inventive encapsulation technique allows a user to simply manipulate visual objects when accessing information on the network. Instead of having to type the destination address of a resource, the user can merely "drag and drop" the icon associated with entity anywhere on the graphical user interface. When the user "double clicks" on the icon, the entity opens up in a window and displays the contents of the resource at that network location. Since the address is encapsulated within the network reference entity, the user does not have to labor with typing of the cumbersome character string.

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. Therefore, 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**. A method of efficiently accessing information from a network resource located on a computer network for display on a computer coupled to the network, the network resource having one or more associated data types, each data type being accessible by a corresponding object-oriented software component, the method comprising the steps of:

defining at least one network component that integrates the object-oriented software components needed to access the one or more data types associated with the network resource;

creating an encapsulated entity component containing a reference to a location of the network resource on the computer network, the encapsulated entity component also identifying the at least one network component that was defined for the network resource;

storing the encapsulated entity component as a visual object on the computer;

in response to manipulation of the visual object with a pointing device, displaying the contents of the network resource on a screen of the computer by invoking the object-oriented software components integrated by the at least one identified network component.

**2**. The method of claim **1** wherein the step of displaying comprises the step of invoking a first network component for displaying the contents of the referenced network resource on the screen, the first network component comprising a browsing component.

**3**. The method of claim **2** wherein the step of displaying further comprises the step of invoking a second network component for transfering the contents of the referenced network resource to the first network component, the second network component comprising a data stream component.

**4**. The method of claim **3** further comprising the step of creating objects for communication among the encapsulated entity and network components through application programming interfaces.

**5**. The method of claim **4** wherein the step of creating comprises the step of constructing the encapsulated entity component from an Item object defined by an Item object class.

**6**. The method of claim **5** wherein the step of creating comprises the step of spawning a Stream object from the Item object, the Stream object representing the data stream.

**7**. Apparatus for efficiently accessing information from a network resource located on a computer network for display on a computer coupled to the network, the network resource having one or more associated data types, each data type being accessible by a corresponding object-oriented software component, the apparatus comprising:

an object-oriented software component architecture layer configured to define at least one network component that integrates the object-oriented software components needed to access the one or more data types associated with the network resource; and

an encapsulated network entity component cooperating with the component architecture layer and containing a reference to the network resource and an identifier for the at least one network component that was defined for the network resource wherein, the encapsulated network entity component is manifested as visual object on a display screen of the computer and further wherein, the encapsulated network entity component is adapted for manipulation by a pointing device of the computer to display contents of the network resource on the screen by invoking the object-oriented software components integrated by the at least one identified network component.

**8**. The apparatus of claim **7** further comprising:

an operating system interfacing with the component architecture layer to control the operations of the computer; and

a network component layer coupled to the component architecture layer to form a cooperating component computing arrangement.

9. The apparatus of claim 8 wherein the cooperating component computing arrangement generates the encapsulated network entity.

10. The apparatus of claim 9 wherein the reference to the network resource is a pointer that identifies the address of the network resource on a computer network.

11. The apparatus of claim 10 wherein the pointer is a uniform resource locator.

12. The apparatus of claim 11 wherein the uniform resource locator has a first portion that identifies the network resource and a second portion that specifies a means for accessing that resource.

13. The apparatus of claim 11 wherein the uniform resource locator is a character string that describes a protocol used to address the network resource, a server on which the resource resides, a path to the resource and a resource filename.

14. The apparatus of claim 10 wherein the pointer is a post office protocol account.

15. Apparatus for efficiently accessing information from a network resource located on a computer network for display on a computer coupled to the network, the network resource having one or more associated data types, each data type being accessible by a corresponding object-oriented software component, the apparatus comprising:

means for defining at least one network component that integrates the object-oriented software components needed to access the one or more data types associated with the network resource;

means for creating an encapsulated entity component containing a reference to a location of the network resource on the computer network, the encapsulated entity component also identifying the at least one network component that was defined for the network resource;

means for storing the encapsulated entity component as a visual object on the computer; and

means, responsive to manipulation of the visual object with a pointing device, for displaying contents of the network resource on a screen of the computer by invoking the object-oriented software components integrated by the at least one identified network component.

16. The apparatus of claim 15 wherein the means for displaying comprises means for invoking a first network component for displaying the contents of the referenced network resource on the screen, the first network component comprising a browsing component.

17. The apparatus of claim 16 wherein the means for displaying further comprises means for invoking a second network component for transfering the contents of the referenced network resource to the first network component, the second network component comprising a data stream component.

18. The apparatus of claim 17 further comprising means for creating objects for communication among the encapsulated entity and network components through application programming interfaces.

19. The apparatus of claim 18 wherein the means for creating comprises means for constructing the encapsulated entity component from an Item objected defined by an Item object class.

20. The apparatus of claim 19 wherein the means for creating comprises means for spawning a Stream object from the Item object, the Stream object representing the data stream.

* * * * *

# EXHIBIT 6

US005946647A

# United States Patent [19]

Miller et al.

[11] Patent Number: 5,946,647

[45] Date of Patent: Aug. 31, 1999

[54] **SYSTEM AND METHOD FOR PERFORMING AN ACTION ON A STRUCTURE IN COMPUTER-GENERATED DATA**

[75] Inventors: **James R. Miller**, Mountain View; **Thomas Bonura**, Capitola; **Bonnie Nardi**, Mountain View; **David Wright**, Santa Clara, all of Calif.

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

[21] Appl. No.: **08/595,257**

[22] Filed: **Feb. 1, 1996**

[51] **Int. Cl.**[6] ..... **G06F 17/27**

[52] **U.S. Cl.** ..... **704/9**; 704/1

[58] **Field of Search** ..... 704/1, 7, 9–10, 704/243; 707/513, 101–104

[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 5,115,390 | 5/1992 | Fukuda et al. | 364/146 |
| 5,130,924 | 7/1992 | Barker et al. | 704/1 |
| 5,164,899 | 11/1992 | Sobotka et al. | 704/9 |
| 5,202,828 | 4/1993 | Vertelney et al. | 364/419 |
| 5,247,437 | 9/1993 | Vale et al. | 704/1 |
| 5,369,575 | 11/1994 | Lamberti et al. | 704/1 |
| 5,574,843 | 11/1996 | Gerlach et al. | 395/118 |

OTHER PUBLICATIONS

TerryMorse Software "What is Myrmidon" Downloaded from the Internet at URL http://www.terrymorse.com (Publication Date Unknown), 2 pages.

Shoens, K. et al. "Rufus System: Information Organization for Semi–Structured Data," Proceedings of the 19th VLDB Conference (Dublin, Ireland 1993), pp. 1–12.

Schwarz, Peter and Shoens, Kurt. "Managing Change in the Rufus System," Abstract from the IBM Almaden Research Center, pp. 1–16.

Myers, Brad A. "Tourmaline: Text Formatting by Demonstration," (Chapter 14) in *Watch What I Do: Programming by Demonstration,* edited by Allen Cypher, MIT Press, (Cambridge, MA 1993), pp. 309–321.

Maulsby, David. "Instructible Agents," Dissertation from the Department of Computer Science at The University of Calgary (Calgary, Alberta—Jun. 1994), pp. 178, 181–188, 193–196 (from Chapter 5).

Rus, Daniela and Subramanian, Devika. "Designing Structure–Based Information Agents," AAAI Symposium (Mar. 1994), pp. 79–86.

*Primary Examiner*—Forester W. Isen

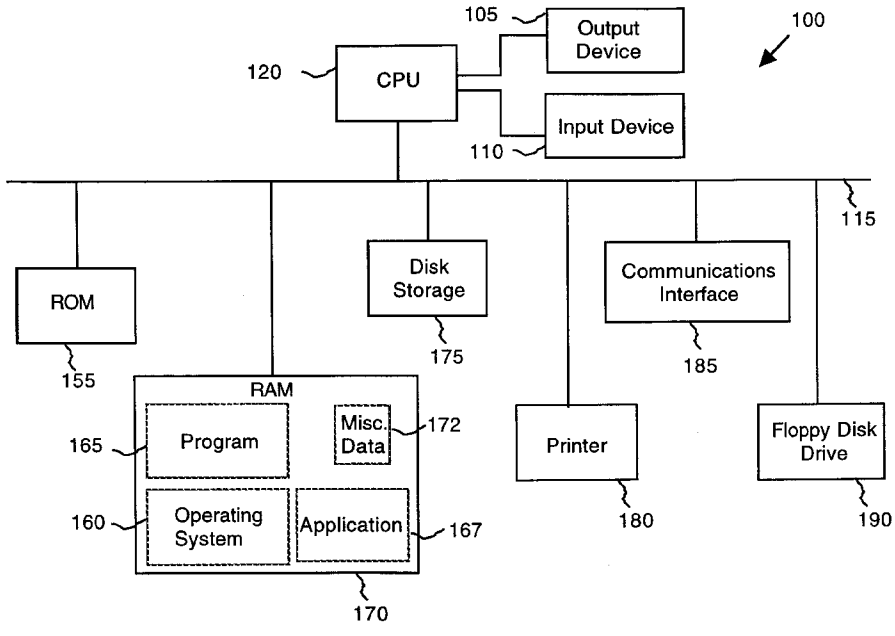
*Assistant Examiner*—Patrick N. Edouard

*Attorney, Agent, or Firm*—Carr & Ferrell LLP

[57] **ABSTRACT**

A system and method causes a computer to detect and perform actions on structures identified in computer data. The system provides an analyzer server, an application program interface, a user interface and an action processor. The analyzer server receives from an application running concurrently data having recognizable structures, uses a pattern analysis unit, such as a parser or fast string search function, to detect structures in the data, and links relevant actions to the detected structures. The application program interface communicates with the application running concurrently, and transmits relevant information to the user interface. Thus, the user interface can present and enable selection of the detected structures, and upon selection of a detected structure, present the linked candidate actions. Upon selection of an action, the action processor performs the action on the detected structure.

**24 Claims, 10 Drawing Sheets**



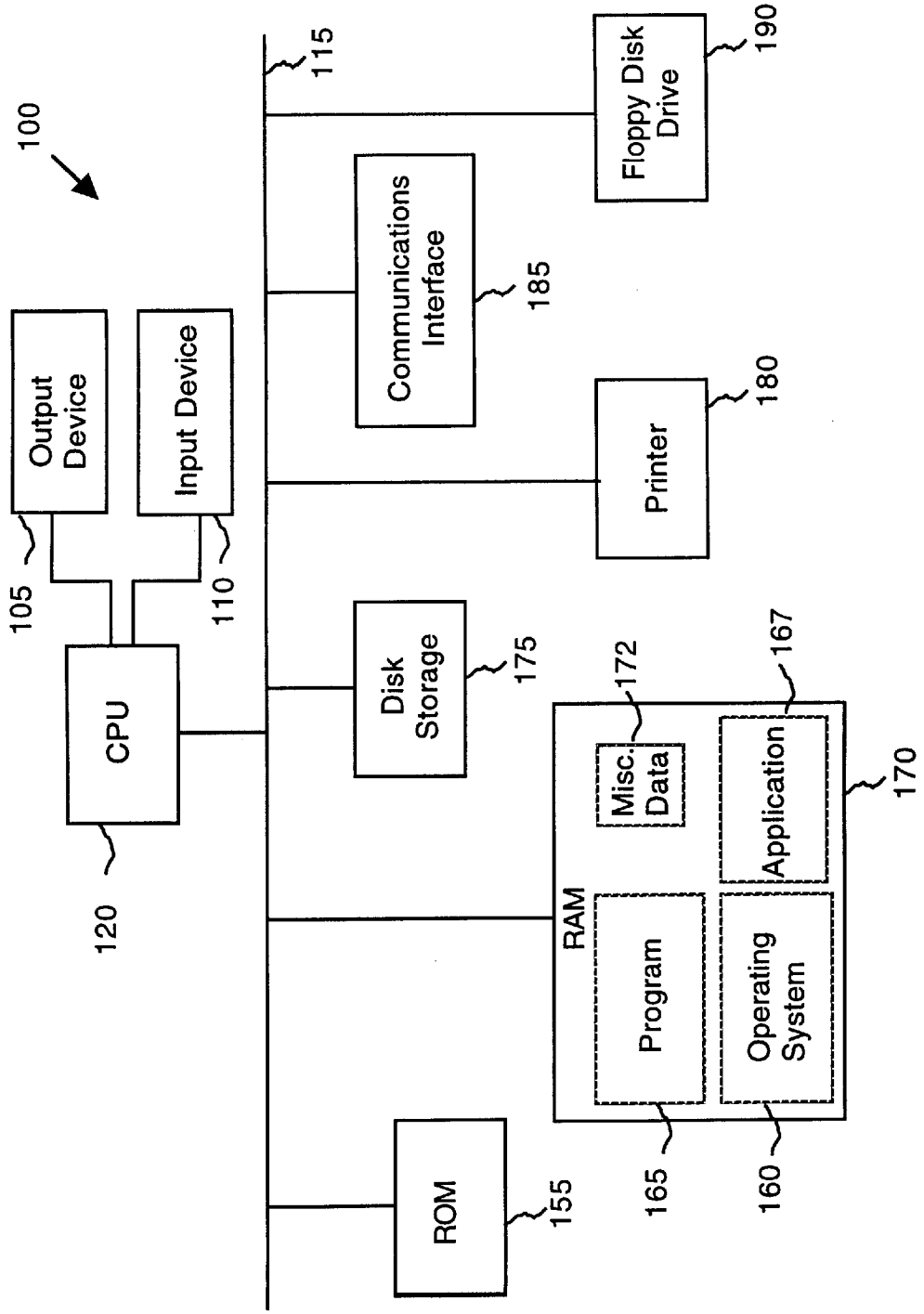

100
115
Floppy Disk Drive
190
Communications Interface
185
Output Device
Input Device
105
110
Printer
180
CPU
120
Disk Storage
175
172
167
Misc. Data
Application
170
RAM
Program
Operating System
ROM
155
165
160

FIG. 1

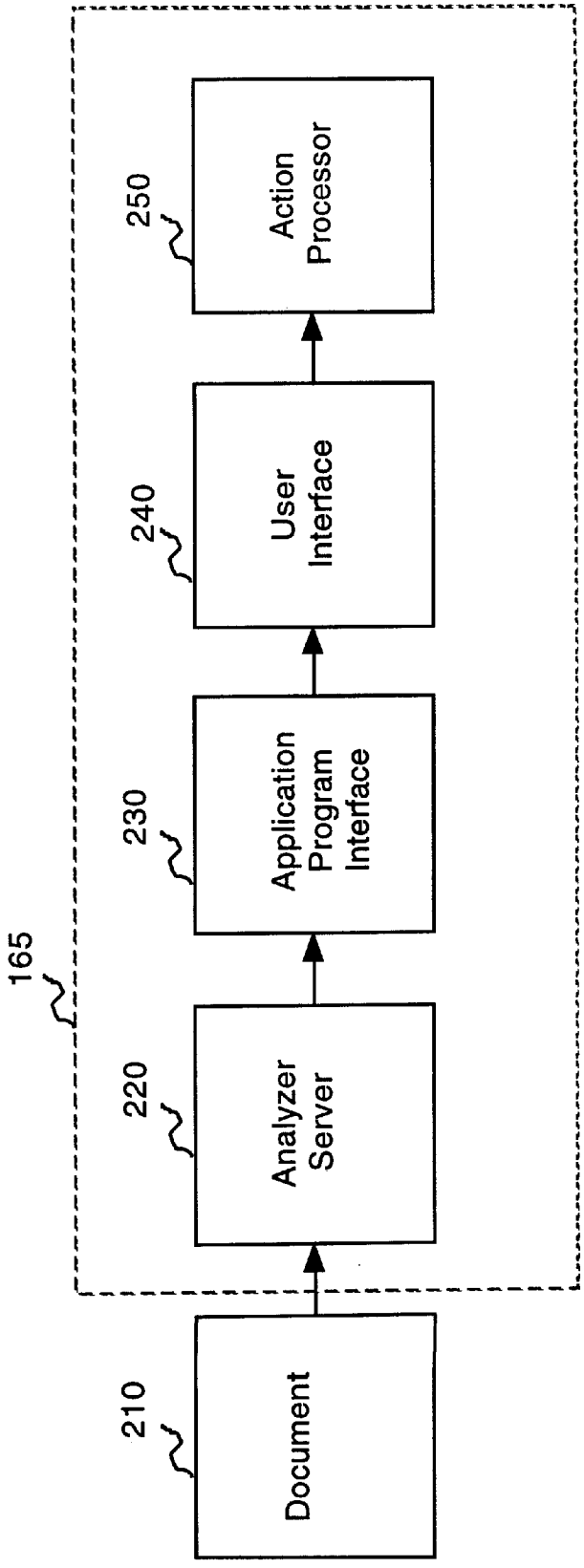

165
210
Document
220
Analyzer Server
230
Application Program Interface
240
User Interface
250
Action Processor

FIG. 2

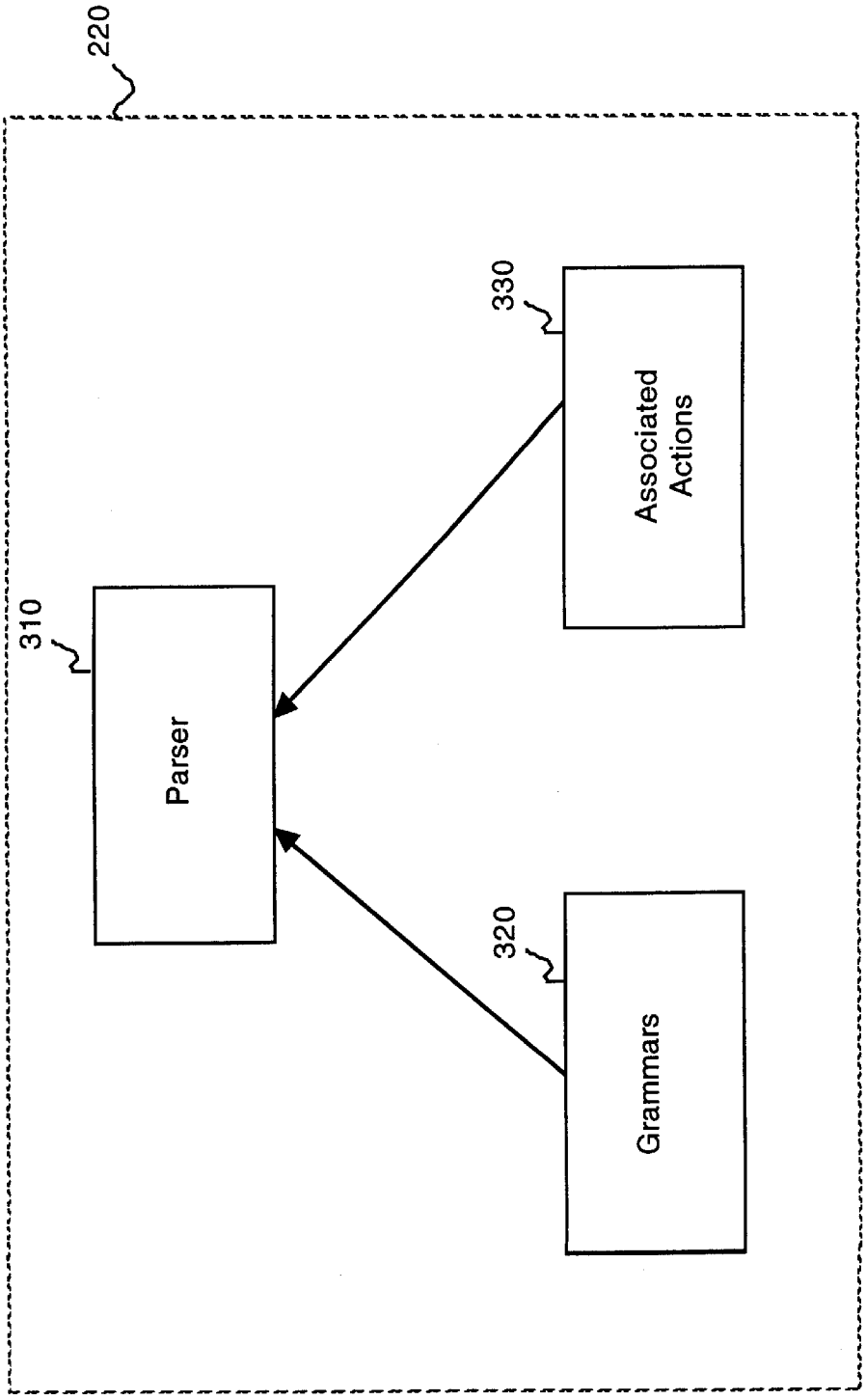



FIG. 3

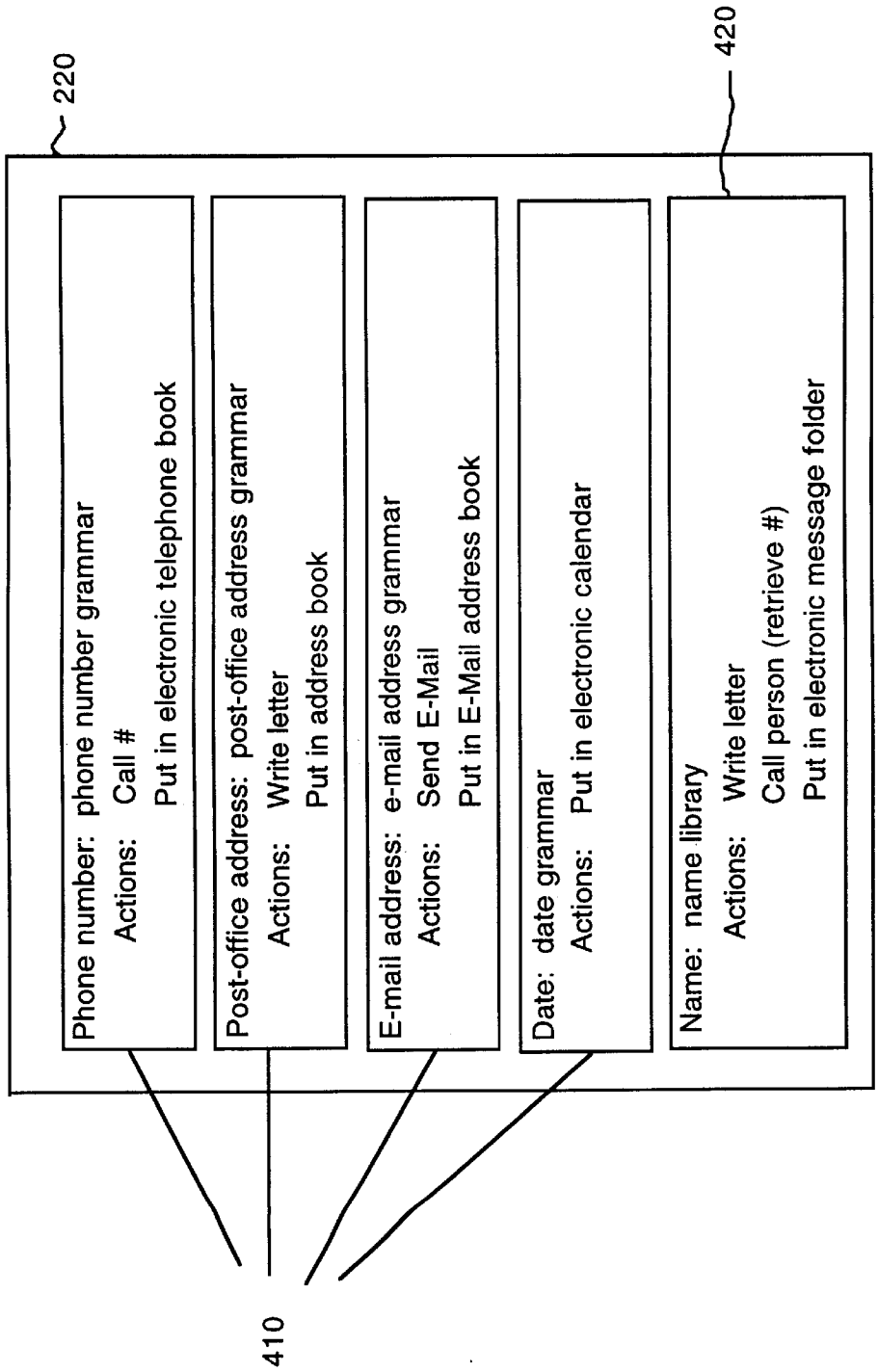

220
420
Phone number: phone number grammar
Actions: Call #
Put in electronic telephone book
Post-office address: post-office address grammar
Actions: Write letter
Put in address book
E-mail address: e-mail address grammar
Actions: Send E-Mail
Put in E-Mail address book
Date: date grammar
Actions: Put in electronic calendar
Name: name library
Actions: Write letter
Call person (retrieve #)
Put in electronic message folder
410

FIG. 4

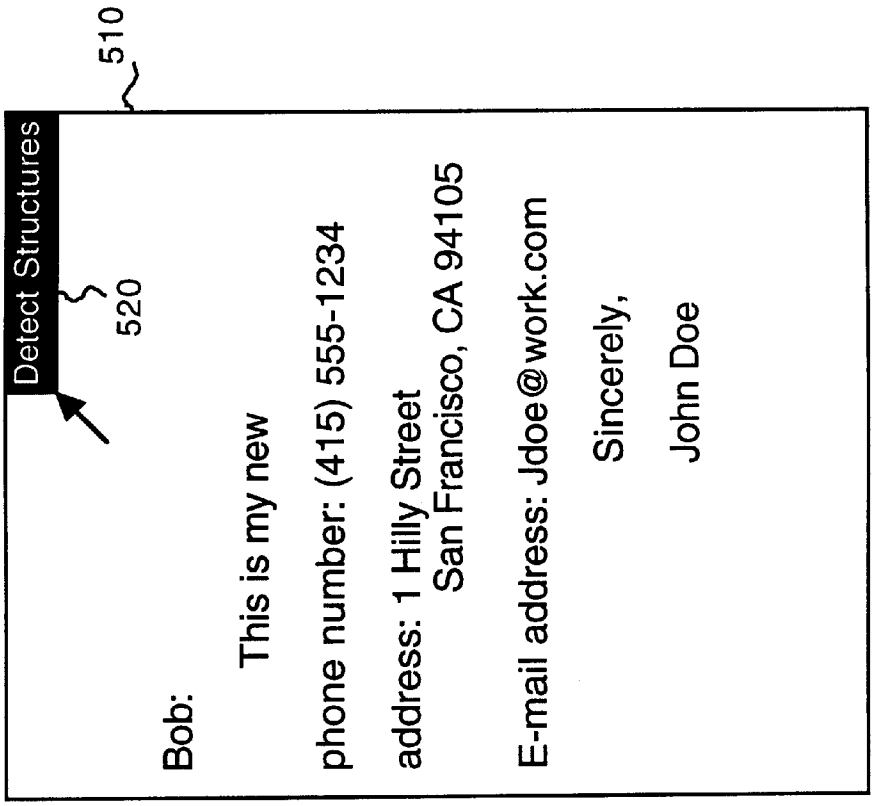

510
Detect Structures
520
Bob:
This is my new
phone number: (415) 555-1234
address: 1 Hilly Street
San Francisco, CA 94105
E-mail address: Jdoe@work.com
Sincerely,
John Doe

FIG. 5

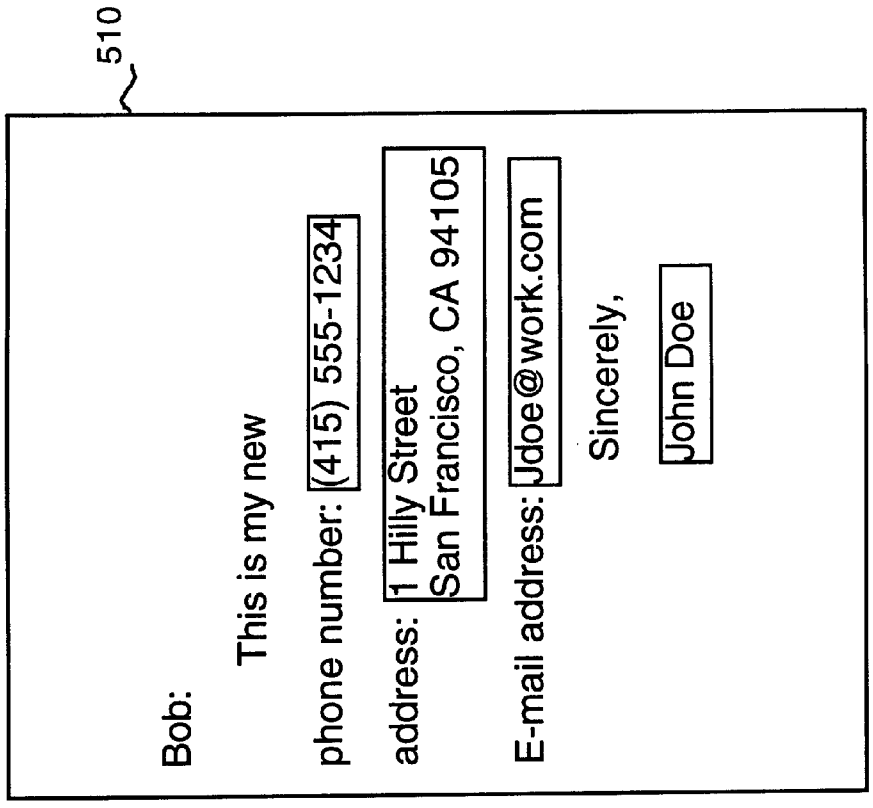

510
Bob:
This is my new
phone number: (415) 555-1234
address: 1 Hilly Street
San Francisco, CA 94105
E-mail address: Jdoe@work.com
Sincerely,
John Doe

FIG. 6

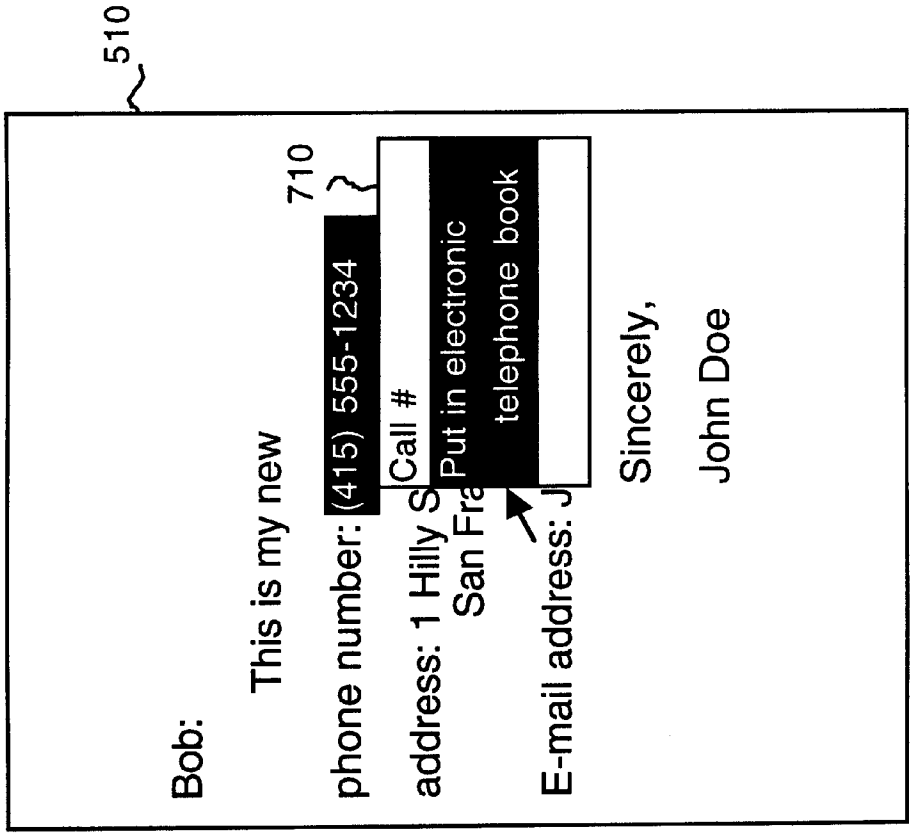

510
710
Bob:
This is my new
phone number: (415) 555-1234
address: 1 Hilly S
San Fra
E-mail address: J
Call #
Put in electronic
telephone book
Sincerely,
John Doe

FIG. 7

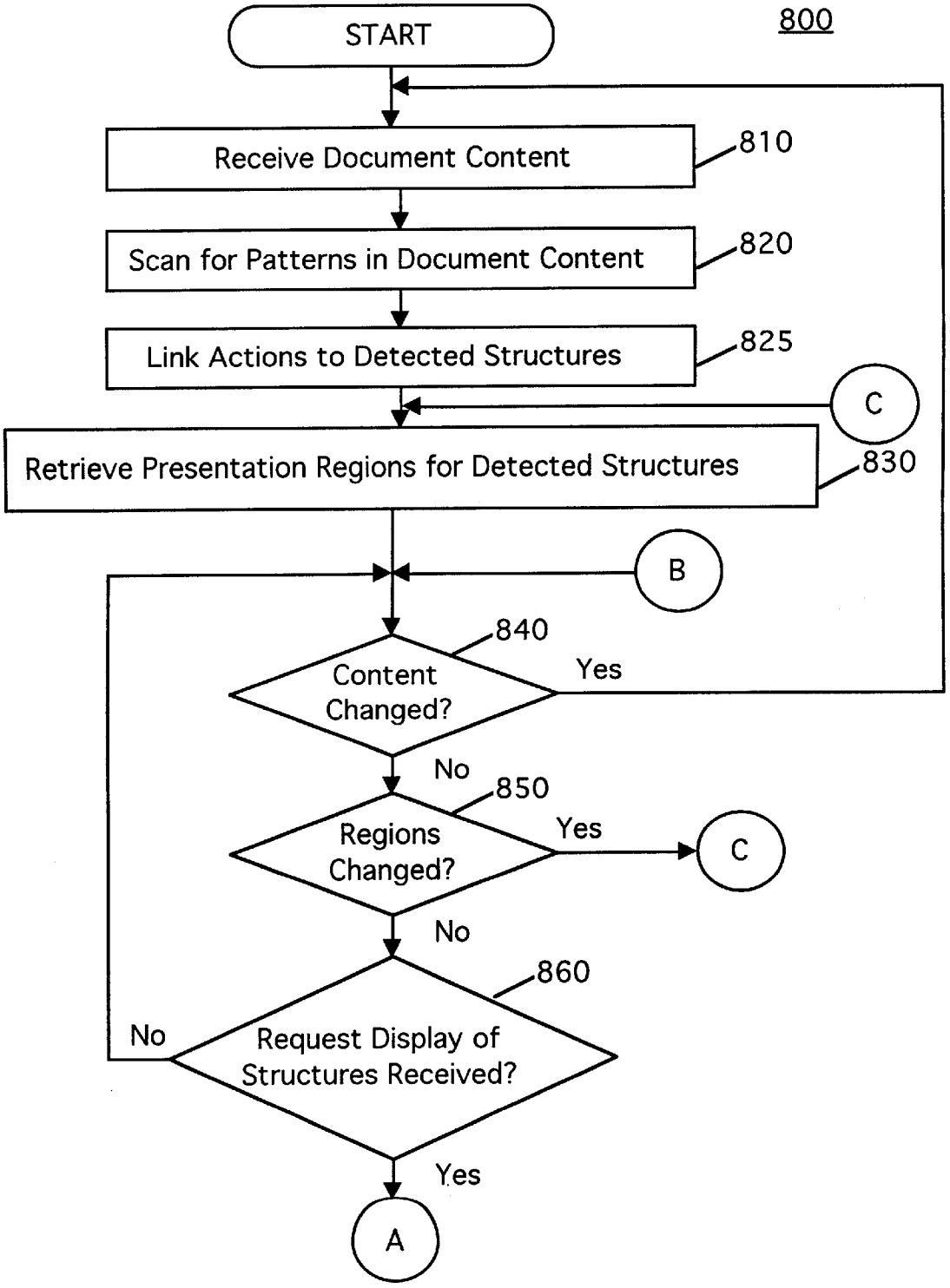

800
START
Receive Document Content
810
Scan for Patterns in Document Content
820
Link Actions to Detected Structures
825
C
Retrieve Presentation Regions for Detected Structures
830
B
840
Content Changed?
Yes
No
850
Regions Changed?
Yes
C
No
860
No
Request Display of Structures Received?
Yes
A

FIG. 8

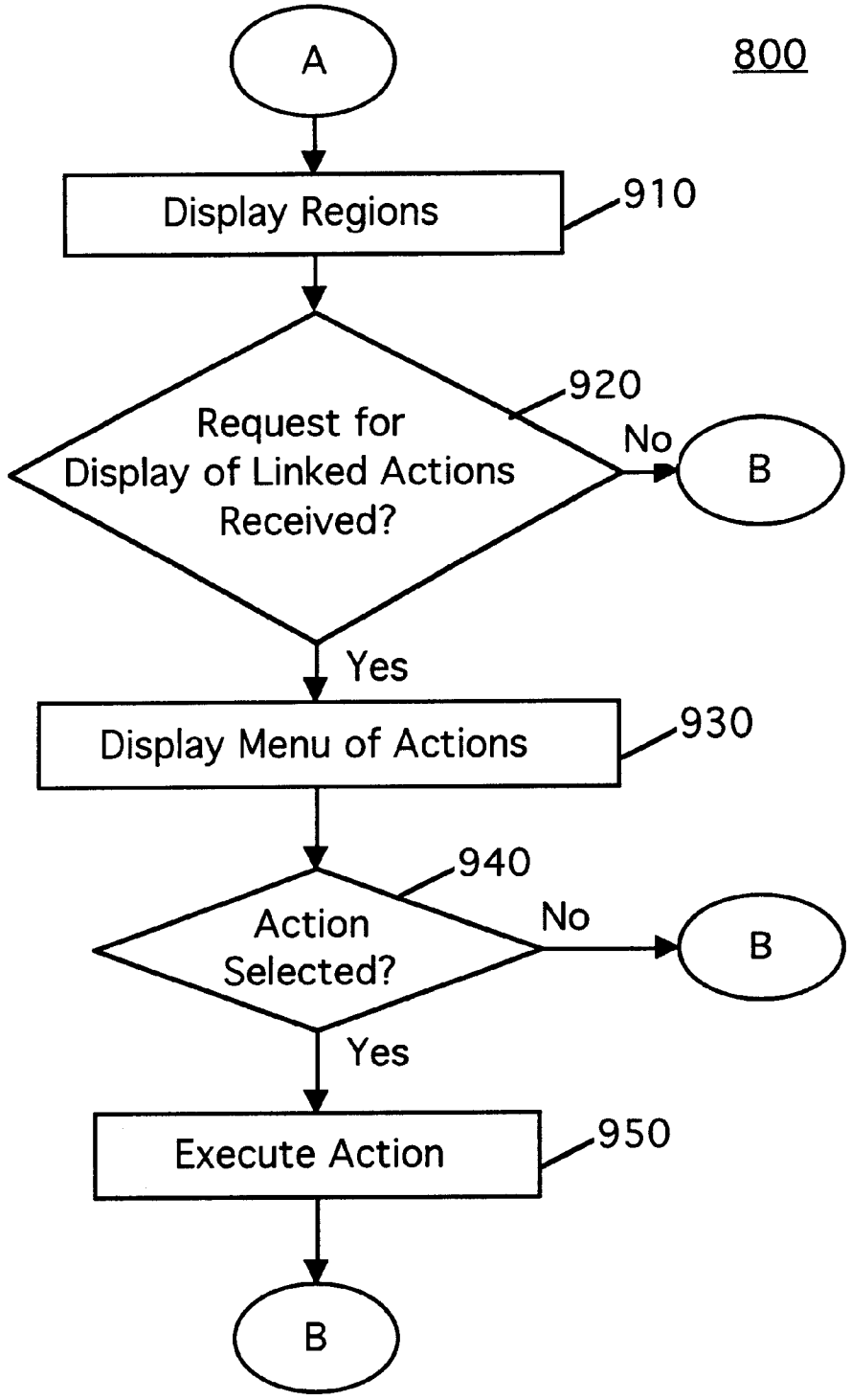

A
800
Display Regions
910
Request for Display of Linked Actions Received?
920
No
B
Yes
Display Menu of Actions
930
Action Selected?
940
No
B
Yes
Execute Action
950
B

FIG. 9

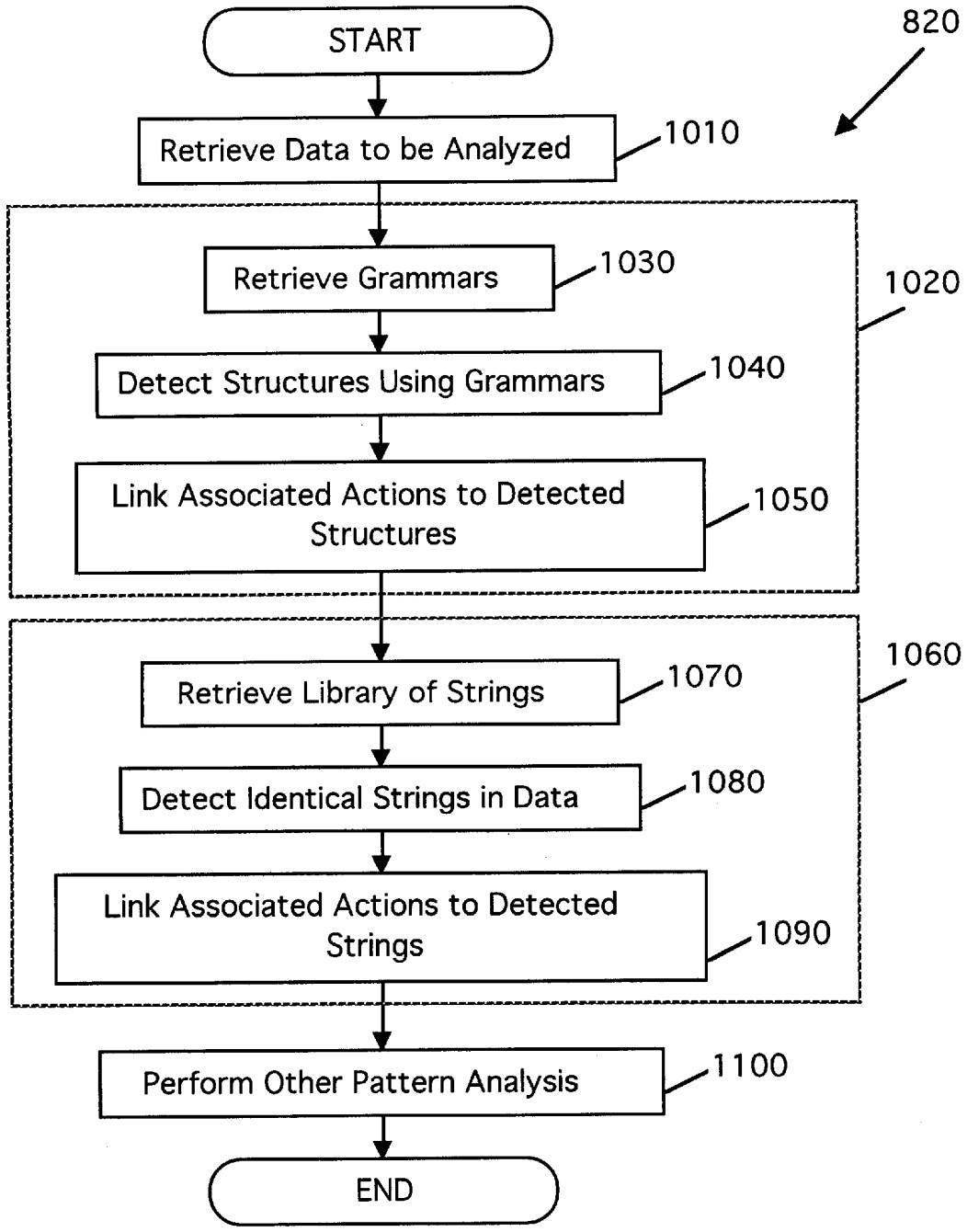

820
START
Retrieve Data to be Analyzed
1010
Retrieve Grammars
1030
1020
Detect Structures Using Grammars
1040
Link Associated Actions to Detected Structures
1050
1060
Retrieve Library of Strings
1070
Detect Identical Strings in Data
1080
Link Associated Actions to Detected Strings
1090
Perform Other Pattern Analysis
1100
END

FIG. 10

# SYSTEM AND METHOD FOR PERFORMING AN ACTION ON A STRUCTURE IN COMPUTER-GENERATED DATA

## BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to manipulation of structures in computer data. More particularly, the invention relates to a system and method for performing computer-based actions on structures identified in computer data.

2. Description of the Background Art

Much data that appears in a computer user's day-to-day activities contains recognizable structures that have semantic significance such as phone numbers, e-mail addresses, post-office addresses, zip codes and dates. In a typical day, for example, a user may receive extensive files from word-processing programs and e-mail that contain several of these structures. However, visually searching data files or documents to find these structures is laborious and cognitively disruptive, especially if the document is lengthy and hard to follow. Furthermore, missing a structure such as a date may lead to missing an important meeting or missing a deadline.

To help facilitate searching a document for these structures, programmers can create or employ pattern analysis units, such as parsers, to automatically identify the structures. For the purposes of the present description, the term "pattern" refers to data, such as a grammar, regular expression, string, etc., used by a pattern analysis unit to recognize information in a document, such as dates, addresses, phone numbers, names, etc. The term "structure" refers to an instantiation of a pattern in the document. That is, a "date" pattern will recognize the structure "Oct. 31, 1995." The application of a pattern to a document is termed "parsing."

Conventional systems that identify structures in computer data do not enable automatic performance of an action on an identified structure. For example, if a long e-mail message is sent to a user, the user may implement a pattern analysis unit to search for particular structures, such as telephone numbers. Upon identification of a structure, the user may want to perform an action on the structure, such as moving the number to an electronic telephone book. This usually involves cutting the structure from the e-mail message, locating and opening the electronic telephone book application program, pasting the structure into the appropriate field, and closing the application program. However, despite the fact that computer systems are getting faster and more efficient, this procedure is still tedious and cognitively disruptive.

One type of system that has addressed this problem involves detecting telephone numbers. Such systems enable a user to select a telephone number and request that the application automatically dial the number. However, these systems do not recognize the selected data as a telephone number, and they generally produce an error message if the user selects invalid characters as a phone number. Also, they do not enable the performance of other candidate actions, such as moving the number to an electronic telephone book. That is, if a user wishes to perform a different action on an identified telephone number, such as storing the number in an address book, the user cannot automatically perform the action but must select and transfer the number to the appropriate data base as described above.

Therefore, a system is needed that identifies structures, associates candidate actions to the structures, enables selection of an action and automatically performs the selected action on the structure.



## SUMMARY OF THE INVENTION

The present invention overcomes the limitations and deficiencies of previous systems with a system that identifies structures in computer data, associates candidate actions with each detected structure, enables the selection of an action, and automatically performs the selected action on the identified structure. It will be appreciated that the system may operate on recognizable patterns for text, pictures, tables, graphs, voice, etc. So long as a pattern is recognizable, the system will operate on it. The present invention has significant advantages over previous systems, in that the present system may incorporate an open-ended number and type of recognizable patterns, an open-ended number and type of pattern analysis units, and further that the system may enable an open-ended number and type (i.e. scripts, macros, code fragments, etc.) of candidate actions to associate with, and thus perform, on each identified structure.

The present invention provides a computer system with a central processing unit (CPU), input/output (I/O) means, and a memory that includes a program to identify structures in a document and perform selected computer-based actions on the identified structures. The program includes program subroutines that include an analyzer server, an application program interface, a user interface and an action processor. The analyzer server receives data from a document having recognizable structures, and uses patterns to detect the structures. Upon detection of a structure, the analyzer server links actions to the detected structure. Each action is a computer subroutine that causes the CPU to perform a sequence of operations on the particular structure to which it is linked. An action may specify opening another application, loading the identified structure into an appropriate field, and closing the application. An action may further include internal actions, such as storing phone numbers in an electronic phone book, addresses in an electronic address book, appointments on an electronic calendar, and external actions such as returning phone calls, drafting letters, sending facsimile copies and e-mail, and the like.

Since the program may be executed during the run-time of another program, i.e. the application which presents the document, such as Microsoft Word, an application program interface provides mechanisms for interprogram communications. The application program interface retrieves and transmits relevant information from the other program to the user interface for identifying, presenting and enabling selection of detected structures. Upon selection of a detected structure, the user interface presents and enables selection of candidate actions. When a candidate action is selected, the action processor performs the selected action on the selected structure.

In addition to the computer system, the present invention also provides methods for performing actions on identified structures in a document. In this method, the document is analyzed using a pattern to identify corresponding structures. Identified structures are stored in memory and presented to the user for selection. Upon selection of an identified structure, a menu of candidate actions is presented, each of which may be selected and performed on the selected structure.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram of a computer system having a program stored in RAM, in accordance with the present invention.

FIG. **2** is a block diagram of the program of FIG. **1**.

FIG. **3** is a block diagram illustrating the analyzer server of FIG. **2**.

FIG. **4** is a block diagram illustrating a particular example of the analyzer server of FIG. **2**.

FIG. **5** illustrates a window presenting an example of a document having recognizable structures.

FIG. **6** illustrates a window with the identified structures in the example document of FIG. **5** highlighted based on the analyzer server of FIG. **4**.

FIG. **7** illustrates a window showing the display of a pop-up menu for selecting an action.

FIGS. **8** and **9** together are a flowchart depicting the preferred method for selecting and performing an action on an identified structure.

FIG. **10** is a flowchart depicting the preferred method for identifying a structure in a data sample.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring now to FIG. **1**, a block diagram is shown of a computer system **100** including a CPU **120**. Computer system **100** is preferably a microprocessor-based computer, such as a Power Macintosh manufactured by Apple Computer, Inc. of Cupertino, Calif. An input device **110**, such as a keyboard and mouse, and an output device **105**, such as a CRT or voice module, are coupled to CPU **120**. ROM **155**, RAM **170** and disk storage **175** are coupled to CPU **120** via signal bus **115**. Computer system **100** optionally further comprises a printer **180**, a communications interface **185**, and a floppy disk drive **190**, each coupled to CPU **120** via signal bus **115**.

Operating system **160** is a program that controls and facilitates the processing carried out by CPU **120**, and is typically stored in RAM **170**. Application **167** is a program, such as a word-processor or e-mail program, that presents data on output device **105** to a user. The program **165** of the present invention is stored in RAM **170** and causes CPU **120** to identify structures in the data presented by application **167**, to associate actions with the structures identified in the data, to enable the user to select a structure and an action, and to automatically perform the selected action on the identified structure. This program **165** may be stored in disk storage **175** and loaded into an allocated section of RAM **170** prior to execution by CPU **120**. Another section of RAM **170** is used for storing intermediate results and miscellaneous data **172**. Floppy disk drive **190** enables the storage of the present program **165** onto a removable storage medium which may be used to initially load program **165** into computer system **100**.

Referring now to FIG. **2**, a schematic block diagram of program **165** is shown together with its interaction with a document **210**. Program **165** contains program subroutines including an analyzer server **220**, an application program interface **230**, a user interface **240** and an action processor **250**. Analyzer server **220** receives data having recognizable patterns from a document **210**, which may be retrieved from a storage medium such as RAM **170**, ROM **155**, disk storage **175**, or the like, and presented on output device **105** by application **167**. Analyzer server **220** comprises one or more pattern analysis units, such as a parser and grammars or a fast string search function and dictionaries, which uses patterns to parse document **210** for recognizable structures. Upon detection of a structure, analyzer server **220** links actions associated with the responsible pattern to the detected structure, using conventional pointers.

After identifying structures and linking actions, application program interface **230** communicates with application **167** to obtain information on the identified structures so that user interface **240** can successfully present and enable selection of the actions. In a display-type environment, application program interface **230** retrieves the locations in document **210** of the presentation regions for the detected structures from application **167**. Application program interface **230** then transmits this location information to user interface **240**, which highlights the detected structures, although other presentation mechanisms can be used. User interface **240** enables selection of an identified structure by making the presentation regions mouse-sensitive, i.e. aware when a mouse event such as a mouse-down operation is performed while the cursor is over the region. Alternative selection mechanisms can be used such as touch sensitive screens and dialog boxes. It will be appreciated that detected structures can be hierarchical, i.e. that a sub-structure can itself be selected and have actions associated with it. For example, a user may be able to select the year portion of an identified date, and select actions specific to the year rather than to the entire date.

User interface **240** communicates with application **167** through application program interface **230** to determine if a user has performed a mouse-down operation in a particular mouse-sensitive presentation region, thereby selecting the structure presented at those coordinates. Upon selection of this structure, user interface **240** presents and enables selection of the linked candidate actions using any selection mechanism, such as a conventional pull-down or pop-up menu.

The above description of the user interface is cast in terms of a purely visual environment. However, the invention is not limited to visual interface means. For example, in an audio environment, user interface **240** may present the structures and associated actions to the user using voice synthesis and may enable selection of a pattern and action using voice or sound activation. In this type of embodiment, analyzer server **220** may be used in conjunction with a text-to-speech synthesis application **167** that reads documents to users over a telephone. Analyzer server **220** scans document **210** to recognize patterns and link actions to the recognized patterns in the same manner as described above. In the audio environment, user interface **240** may provide a special sound after application **167** reads a recognized pattern, and enable selection of the pattern through the use of an audio interface action, such as a voice command or the pressing of a button on the touch-tone telephone keypad as before. Thus, user interface **240** may present the linked actions via voice synthesis. One can create various environments having a combination of sensory mechanisms.

Upon selection of a candidate action, user interface **240** transmits the selected structure and the selected action to action processor **250**. Action processor **250** retrieves the sequence of operations that constitute the selected action, and performs the sequence using the selected structure as the object of the selected action.

Referring now to FIG. **3**, a block diagram illustrating an analyzer server **220** is shown. In this figure, analyzer server **220** is described as having a parser **310** and a grammar file **320**, although alternatively or additionally a fast string search function or other function can be used. Parser **310** retrieves a grammar from grammar file **320** and parses text using the retrieved grammar. Upon identification of a structure in the text, parser **310** links the actions associated with the grammar to the identified structure. More particularly, parser **310** retrieves from grammar file **320** pointers attached

to the grammar and attaches the same pointers to the identified structure. These pointers direct the system to the associated actions contained in associated actions file **330**. Thus, upon selection of the identified structure, user interface **240** can locate the linked actions.

FIG. **4** illustrates an example of an analyzer server **220**, which includes grammars **410** and a string library **420** such as a dictionary, each with associated actions. One of the grammars **410** is a telephone number grammar with associated actions for dialing a number identified by the telephone number grammar or placing the number in an electronic telephone book. Analyzer server **220** also includes grammars for post-office addresses, e-mail addresses and dates, and a string library **420** containing important names. When analyzer server **220** identifies an address using the "e-mail address" grammar, actions for sending e-mail to the identified address and putting the identified address in an e-mail address book are linked to the address.

FIG. **5** shows a window **510** presenting an exemplary document **210** having data containing recognizable structures, including a phone number, post-office address, e-mail address, and name. Window **510** includes a button **520** for initiating program **165**, although alternative mechanisms such as depressing the "option" key may be used. Upon initiation of program **165**, system **100** transmits the contents of document **210** to analyzer server **220**, which parses the contents based on grammars **410** and strings **420** (FIG. **4**). This parsing process produces the window shown in FIG. **6**. As illustrated in FIG. **6**, analyzer server **220** identifies the phone number, post-office address, e-mail address and name. Although not shown in FIG. **6**, analyzer server **220** links the actions associated with grammars **410** and strings **420** to these identified structures, and application program interface **230** retrieves information on the location of these structures from application **167**. User interface **240** then highlights the identified structures in document **210**, and makes the identified structures mouse-sensitive.

As shown in FIG. **7**, upon recognition of a mouse-down operation over a structure, user interface **240** presents a pop-up menu **710**. In this example, pop-up menu **710** displays the candidate actions linked to the selected telephone number grammar **410**, including dialing the number and putting the number into an electronic telephone book. Upon selection of the action for putting the number in an electronic telephone book, user interface **240** transmits the corresponding telephone number and selected action to action processor **250**. Action processor **250** locates and opens the electronic telephone book, places the telephone number in the appropriate field and allows the user to input any additional information into the file.

FIGS. **8** and **9** display a flowchart illustrating preferred method **800** for recognizing patterns in documents and performing actions. This method is carried out during the run-time of application **167**. Referring first to FIG. **8**, method **800** starts by receiving **810** the content, or a portion of the content, from document **210**. Assuming program **165** initiates with the receipt of any text, the received content or portion is scanned **820** for identifiable structures using the patterns in analyzer server **220**. Upon detection of a structure based on a particular pattern, actions associated with the particular pattern are linked **825** to the detected structure. Assuming a display-type environment, the presentation region location for a detected structure is retrieved **830** from application **167**. If the document content being displayed on output device **105** is changed **840**, for example by the user adding or modifying text, method **800** restarts. Otherwise, method **800** continues with block **850**. If the presentation regions change **850**, for example by the a user scrolling document **210**, then new presentation regions from application **167** are again retrieved **830**. Otherwise, method **800** continues to block **860**. As illustrated by block **860**, method **800** loops between blocks **840** and **860** until a request for display of identified structures is received **860**. It will be appreciated that the steps of the loop (blocks **840**, **850** and **860**) can be performed by application **167**.

Referring also to FIG. **9**, when a request for the display of detected structures is received **860**, the regions are displayed **910** using presentation mechanisms such as highlighting the presentation region around each detected structure, although alternative presentation mechanisms can be used. If a request for the display of candidate actions linked to a detected structure is not received **920**, method **800** returns to block **840**. However, if a request is received **920**, the actions linked in block **825** are displayed **930**. This request for display of candidate actions can be performed using a selection mechanism, such as a mouse-down operation over a detected structure, which causes the candidate actions linked to the structure to be displayed **930**. Display **930** of candidate actions may be implemented using a pop-up menu, although alternative presentation mechanisms can be used such as pull-down menus, dialog boxes and voice synthesizers.

As illustrated in block **940**, if an action from the displayed candidate actions is not selected **940**, method **800** returns to block **840**. However, if an action is selected **940**, the action is executed **950** on the structure selected in block **920**. After execution **950** of an action, method **800** returns to block **840**. Method **800** ends when the user exits application **167**, although other steps for ending method **800** can alternatively be used.

Referring now to FIG. **10**, a flowchart illustrating the preferred method **820** for scanning and detecting patterns in a document is shown. Method **820** starts by retrieving **1010** data to be analyzed. After the data is retrieved, several pattern analysis processes may be performed on the data. As illustrated in block **1020**, a parsing process retrieves **1030** grammars, detects **1040** structures in the data based on the retrieved grammars, and links **1050** actions associated with each grammar to each structure detected by that grammar. As illustrated in block **1060**, a fast string search function retrieves **1070** the contents of string library **420**, detects **1080** the strings in the data identical to those in the string library **420**, and links **1090** actions associated with the library string to the detected string. As illustrated in block **1100**, additional pattern analysis processes, such as a neural net scan, can be performed **1100** to detect in the data other patterns, such as pictures, graphs, sound, etc. Method **820** then ends. Alternatively, the pattern analysis processes can be performed in parallel using a multiprocessor multitasking system, or using a uniprocessor multithreaded multitasking system where a thread is allocated to execute each pattern detection scheme.

These and other variations of the preferred and alternate embodiments and methods are provided by the present invention. For example, program **165** in FIG. **1** can be stored in ROM, disk, or in dedicated hardware. In fact, it may be realized as a separate electronic circuit. Other components of this invention may be implemented using a programmed general purpose digital computer, using application specific integrated circuits, or using a network of interconnected conventional components and circuits. The analyzer server **220** of FIG. **2** may use a neural net for searching a graphical document **210** for faces, or a musical library for searching a stored musical piece **210** for sounds. The user interface **240**

may present structures and actions via voice synthesis over a telephone line connection to system **100**. The embodiments described have been presented for purposes of illustration and are not intended to be exhaustive or limiting, and many variations and modifications are possible in light of the foregoing teaching. The system is limited only by the following claims.

What is claimed is:

**1**. A computer-based system for detecting structures in data and performing actions on detected structures, comprising:

an input device for receiving data;

an output device for presenting the data;

a memory storing information including program routines including

an analyzer server for detecting structures in the data, and for linking actions to the detected structures;

a user interface enabling the selection of a detected structure and a linked action; and

an action processor for performing the selected action linked to the selected structure; and

a processing unit coupled to the input device, the output device, and the memory for controlling the execution of the program routines.

**2**. The system recited in claim **1**, wherein the analyzer server stores detected structures in the memory.

**3**. The system recited in claim **1**, wherein the input device receives the data from an application running concurrently, and wherein the program routines stored in memory further comprise an application program interface for communicating with the application.

**4**. The system recited in claim **1**, wherein the analyzer server includes grammars and a parser for detecting structures in the data.

**5**. The system recited in claim **4**, wherein the analyzer server includes actions associated with each of the grammars, and wherein the analyzer server links to a detected structure the actions associated with the grammar which detects that structure.

**6**. The system recited in claim **1**, wherein the analyzer server includes a string library and a fast string search function for detecting string structures in the data.

**7**. The system recited in claim **6**, wherein the analyzer server includes actions associated with each of the strings, and wherein the analyzer server links to a detected structure the actions associated with the grammar which detects that string structure.

**8**. The system recited in claim **1**, wherein the user interface highlights detected structures.

**9**. The system recited in claim **1**, wherein the user interface enables selection of an action by causing the output device to display a pop-up menu of the linked actions.

**10**. The system recited in claim **1**, wherein the programs stored in the memory further comprise an application running concurrently that causes the output device to present the data received by the input device, and an application program interface that provides interrupts and communicates with the application.

**11**. The system recited in claim **1**, wherein the user interface enables the selection of a detected structure and a linked action using sound activation.

**12**. The system recited in claim **1**, wherein a first one of the actions may invoke a second one of the actions.

**13**. A program storage medium storing a computer program for causing a computer to perform the steps of:

receiving computer data;

detecting a structure in the data;

linking at least one action to the detected structure;

enabling selection of the structure and a linked action; and

executing the selected action linked to the selected structure.

**14**. In a computer having a memory storing actions, a system for causing the computer to perform an action on a structure identified in computer data, comprising:

means for receiving computer data;

means for detecting a structure in the data;

means for linking at least one action to the detected structure;

means for selecting the structure and a linked action; and

means for executing the selected action linked to the selected structure.

**15**. In a computer having a memory storing actions, a method for causing the computer to perform an action on a structure identified in computer data, comprising the steps of:

receiving computer data;

detecting a structure in the data;

linking at least one action to the detected structure;

enabling selection of the structure and a linked action; and

executing the selected action linked to the selected structure.

**16**. The method recited in claim **15**, wherein the computer data is received from the application running concurrently.

**17**. The method recited in claim **15**, wherein the memory contains grammars, and wherein the step of detecting a structure further comprises the steps of retrieving a grammar and parsing the data based on the grammar.

**18**. The method recited in claim **17**, wherein the grammar is associated with a particular action, and wherein the step of linking at least one action to the detected structure includes the step of linking the particular action to the detected structure.

**19**. The method recited in claim **15**, wherein the memory contains strings, and wherein the step of detecting a structure further comprises the steps of retrieving a string from the memory and scanning the data to identify the string.

**20**. The method recited in claim **15**, further comprising after the step of detecting a structure, the step of highlighting the detected structure.

**21**. The method recited in claim **15**, further comprising, after the step of linking at least one action to the detected structure, the step of displaying and enabling selection of an action for performance on the detected structure.

**22**. A computer-based method for causing a computer to identify, select and perform an action on a structure in computer data received from a concurrently running application, said application presenting the computer data to the user, the method comprising the steps of:

receiving computer data from the application;

detecting a structure in the computer data;

linking at least one action to the detected structure;

communicating with the application to determine the location of the detected structure as presented by the application, to enable selection of the detected structure and a linked action, and to determine if the detected structure and a linked action have been selected; and

performing a selected action linked to the detected pattern.

**23**. The method recited in claim **15**, wherein the step of enabling uses sound activation.

**24**. The method recited in claim **15**, wherein a first one of the actions may invoke a second one of the actions.

* * * * *