AU9464191

(12) PATENT ABRIDGMENT (11) Document No. AU-B-64191/94
(19) AUSTRALIAN PATENT OFFICE (10) Acceptance No. 674215

(54) Title
EVENT QUALIFICATION AND NOTIFICATION

International Patent Classification(s)
(51)[5] **G06F 009/46**

(21) Application No. : **64191/94** (22) Application Date : **31.03.94**

(87) PCT Publication Number : **WO94/23365**

(30) Priority Data

(31) Number	(32) Date	(33) Country
040485	**31.03.93**	**US UNITED STATES OF AMERICA**

(43) Publication Date : **24.10.94**

(44) Publication Date of Accepted Application : **12.12.96**

(71) Applicant(s)
KALIEDA LABS, INC.

(72) Inventor(s)
YIN-SHUR DAVID MA

(74) Attorney or Agent
DAVIES COLLISON CAVE , 1 Little Collins Street, MELBOURNE VIC 3000

(56) Prior Art Documents
EP 297339
WO 92/04682
AU 25303/92 G06F 15/20 9/46

(57) Claim

1. A method for determining, on behalf of an application program, when a specified combination of logically related events has occurred, said method comprising the following steps:

providing an event qualifier network data structure ("EQN") associated with at least one piece of software code, the EQN specifying a logical relationship among the combination of events, the logical relationship comprising a plurality of logical qualifiers, each one of said logical qualifiers qualifying a group of one or more of the events, and wherein at least two of the logical qualifiers correspond to different logic functions;

registering an interest, on behalf of the application program, in being notified when the logical relationship specified by the EQN has been satisfied;

detecting the occurrence of each of the events;

notifying the EQN of each said occurrence by invoking a piece of software code associated with the EQN;

determining when the logical relationship among the combination of events specified by the EQN is satisfied; and

notifying the application program when the specified logical relationship among the combination of events has been satisfied.

GPI DATE 24/10/94 APPLN. ID 64191/94
AOJP DATE 08/12/94 PCT NUMBER PCT/US94/03590

AU9464191

PCT)

(51) International Patent Classification 5 : G06F 9/46

A1

(11) International Publication Number: WO 94/23365

(43) International Publication Date: 13 October 1994 (13.10.94)

(21) International Application Number: PCT/US94/03590

(22) International Filing Date: 31 March 1994 (31.03.94)

(30) Priority Data:
08/040,485 31 March 1993 (31.03.93) US

(71) Applicant: KALIEDA LABS, INC. [US/US]; 1945 Charleston Road, Mountain View, CA 94043 (US).

(72) Inventor: MA, Yin-Shur, David; 4963 Paseo Olivos, San Jose, CA 95130 (US).

(74) Agents: BLATT, Jeffrey, J. et al.; Irell & Manella, 1800 Avenue of the Stars, Suite 900, Los Angeles, CA 90067 (US).

(81) Designated States: AT, AU, BB, BG, BR, BY, CA, CH, CN, CZ, DE, DK, ES, FI, GB, HU, JP, KP, KR, KZ, LK, LU, MG, MN, MW, NL, NO, NZ, PL, PT, RO, RU, SD, SE, SK, UA, UZ, VN, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

674215

(54) Title: EVENT QUALIFICATION AND NOTIFICATION

(57) Abstract

A method and apparatus for event handling in a computer environment is disclosed. The present invention provides applications programs with a mechanism for establishing a template, called an event qualifier network, which monitors the occurrence of an event or specific combination of events, along with virtually any other qualifying condition specified by the application. In response to the occurrence of such events and conditions, the event qualifier network notifies the application that its "desired event" has occurred, whereupon the application may then take the appropriate action. The event qualifier networks of the present invention can be quickly and simply built to monitor arbitrarily complex combinations of events and events in combination with other conditions.

EVENT QUALIFICATION AND NOTIFICATION

Cross References

The present invention is related to Australian Patent Application No. 64189/94 entitled "Synchronize Clocks and Media Players", which is incorporated by reference herein.

Background

The present invention relates to the field of computer operating systems and general application software, and more particularly to methods and apparatus for controlling the communication of events between an operating system and its application programs.

Computer programs are, in essence, a series of instructions that direct the operation of a computer. In general, a program performs the functions either of an operating system, which controls the interaction of the various components of the computer relatively independent of the use to which the computer is being put, or of an application which is designed to accomplish a specific task or tasks (e.g., word processing, spreadsheet, etc.).

Many operating system programs and virtually all application programs provide for, and in fact require, user interaction with the program. The scheme used to facilitate the user's interaction with the computer is referred to as the user interface.

Popular common user interfaces employ windows, which are discrete regions of the screen in which elements are displayed, and user interaction with the computer program may occur, scroll bars for

AUSTRALIAN PATENT OFFICE

scrolling up or down within a window, menus for selecting commands, icons representing applications and documents, etc. In conjunction with a program's user interface, hardware devices are provided for interaction with the programs. Examples of such hardware devices include a keyboard, a mouse, a disk drive, a network interface card, etc. The Macintosh line of computers from Apple Computer Inc. of Cupertino, California, operate with such a user interface and with such hardware.

Many modern application programs operate by looping repeatedly through the program, periodically polling the operating system software for an indication that at least one "event" has occurred. Events generally represent user activity via one or more of the hardware devices, but can also represent actions generated independently by external hardware devices, as well as internal "messages" sent by a system or application program.

When the application learns (by polling the system) that an event has occurred, the application must determine whether the event is of a type in which it is "interested" (i.e., an event in response to which the application program desires to perform some action), and if so, what action needs to be taken in response to that event. Since the application will not do anything until an event is encountered, this type of application is referred to as "event driven."

Examples of user-generated events include: typing on a keyboard (also referred to as "key-down" and "key-up" events), depressing a button on a mouse (also referred to as "mouse-down" and "mouse-up" events), and inserting a disk in a disk

drive. Events may also be generated independently by a hardware device (e.g., a signal from a network interface card indicating that a block of data has arrived over the network), as well as by an application or operating system program (e.g., a message indicating a particular state of the program, such as the completion of a spreadsheet calculation).

These events are typically handled, one at a time, by the application. In certain circumstances, applications are interested in a combination of events. For example, an application may desire to close a window if the user simultaneously depresses the "option" key and the mouse button (within that window). In this case, in deciding what action to take in response to the events generated by depressing these keys, the application program must determine if the key depressions occurred simultaneously.

In the Apple Macintosh, events are posted to an event queue, where they are metered out, one at a time, to the application by an Event Manager. Generally, events are retrieved by the application in a first-in-first-out order. However, some events may have a higher priority than other events, and in addition, an application may ask for certain events before others.

The precise manner by which events are generated, captured, and communicated to an application program varies from system to system, and is beyond the scope of this discussion. For a more detailed discussion of events and event handling, see INSIDE MACINTOSH, Apple Computer, Vol. 1, pp. 1-243 to 1-266, and other sections

thereof (Addison-Wesley, 1985), incorporated herein by reference.

Event handling as presently practiced in the art presents a number of problems and disadvantages to which the present invention is addressed. One problem that arises in event handling, especially where combinations of events are involved, is that a significant amount of application program code is required to check for events, the order in which they occur, the proximity in time to other events, and the state of other system and application resources in which the application is interested. Likewise, the complexity of the application program code increases dramatically as the application is required to monitor for increasingly complex combinations of events and other conditions. The process of checking for the occurrence of these events and conditions requires not only a significant amount of application code, but also a significant amount of application processing time.

Another problem occurs when an application programmer seeks to provide a user with the ability to change the event or combination of events or other conditions which will trigger a certain action by the application program. For example, while an application may open a new window in response to the combination of certain key-down and mouse-down events which occur while the cursor is in a particular region of the screen, it may be desirable to provide the user with the ability to change the specific events or other conditions which trigger the opening of a new window - e.g., the single key-down event caused by depressing a dedicated "function" key. Allowing changes such as these generally requires either recompiling or

"patching" the program, and only exacerbates the problems of code volume, complexity, and processing time. In general it is desirable to simplify the process of writing event driven applications, and as described below this forms the basis for the present invention.

Summary of the Invention

The present invention provides a method and apparatus for event handling which addresses the problems associated with the prior art, including those problems discussed above, by delegating, for example to the system software, the task of determining whether particular combinations of events and other conditions have occurred. The present invention provides applications with a mechanism for establishing a template, called an event qualifier network, which monitors the occurrence of an event or specific combination of events, along with virtually any other qualifying condition specified by the application. In response to the occurrence of such events and conditions, the event qualifier network notifies the application that its "desired event" has occurred, whereupon the application may then take the appropriate action.

Event qualifier networks can be quickly and simply built to monitor for arbitrarily complex combinations of events. Furthermore, an event qualifier network may monitor for events in combination with other conditions, such as a particular state of the system or application program (e.g., a mouse-down event and the cursor being in a particular location of the screen), and notify the application when the desired events

occur while the specified conditions are met. In addition, event qualifier networks may be combined into a single event qualifier network, providing reusability, improved performance and an event qualifier hierarchy.

More specifically the invention provides a method for determining, on behalf of an application program, when a specified combination of logically related events has occurred, said method comprising the following steps:

providing an event qualifier network data structure ("EQN") associated with at least one piece of software code, the EQN specifying a logical relationship among the combination of events, the logical relationship comprising a plurality of logical qualifiers, each one of said logical qualifiers qualifying a group of one or more of the events, and wherein at least two of the logical qualifiers correspond to different logic functions;

registering an interest, on behalf of the application program, in being notified when the logical relationship specified by the EQN has been satisfied;

detecting the occurrence of each of the events;

notifying the EQN of each said occurrence by invoking a piece of software code associated with the EQN;

determining when the logical relationship among the combination of events specified by the EQN is satisfied; and

notifying the application program when the specified logical relationship among the combination of events has been satisfied.

The invention also provides an apparatus for determining, on behalf of an application program, when a specified combination of logically related events has occurred, said apparatus comprising:

an event qualifier network data structure ("EQN") associated with at least one piece of software code, the EQN specifying a logical relationship among the combination of events, the logical relationship comprising a plurality of logical qualifiers, each one of said logical qualifiers qualifying a group of one or more of the events, and wherein at least two of the logical qualifiers correspond to different logic functions;

means for registering an interest with the EQN, on behalf of the application program, in being notified when the logical relationship specified by the EQN has been satisfied;

AUSTRALIAN PATENT OFFICE 13

event manager means for detecting the occurrence of each of the events, and for notifying the EQN of each said occurrence by invoking a piece of software code associated with the EQN;

means for determining when the logical relationship among the combination of events specified by the EQN is satisfied; and

means for notifying the application program when the specified logical relationship among the combination of events has been satisfied.

Brief Description of the Drawings

The present invention will be described in greater detail below, including specific examples thereof, with regard to the Figures, in which:

Fig. 1 is an illustration of the general architecture of a computer system within which the present invention may operate;

Fig. 2 is an illustration of a particular software environment in which the present invention may operate, showing the control and data flow between an application software program and an operating system software program;

Fig. 3 is an illustration of an event qualification network according to an exemplary embodiment of the present invention;

Fig. 4 is an illustration of a data structure passed to an event manager for the purpose of registering an event qualifier with the event manager;

Fig. 5 is an illustration of a portion of a computer software architecture showing an intermediate event manager interposed between an event qualification network and a system event manager;

Fig. 6 is an illustration of an event record containing the required information for enabling a called function to determine, *inter alia*, whether an event is of interest to an event qualifier;

AUSTRALIAN PATENT OFFICE

Fig. 7 is an illustration of another event qualification network according to an exemplary embodiment of the present invention;

Fig. 8 is an illustration of the COS event server hierarchy according to one embodiment of the present invention, as described in Appendix A hereof;

Fig. 9 is an illustration of an interaction between user and computer demonstrating the metaphor of a "hurt" screen, and its response of covering the hurt region with a "bandaid," as described in Appendix A hereof; and

Fig. 10 is a flow diagram for an event referred to as KTOUCH_DOWN_EVNT, also as described in Appendix A hereof.

As between each of these figures, like reference numerals shall denote like elements.

Detailed Description

For purposes of illustration, the present invention will be described by way of selected examples (or applications), with reference to a number of embodiments of the present invention where appropriate. It will be apparent to one skilled in the art that the examples, and the environment in which they are developed, demonstrate the functionality and features of the present invention. It will also be apparent to one skilled in the art that certain variations, additions, deletions, modifications, etc., to the examples are contemplated, and that recitation of these examples does not limit the scope of the invention.

With regard to Fig. 1, there is shown therein the general architecture of a system 10 of the type within which the present invention operates. System 10 comprises a main bus 12, which serves to interconnect various components, including at least some of the following (and possibly other additional components): Central Processing Unit (CPU) 14, Floating Point Unit (FPU) 16, Bus Controller 20, Video RAM 22, Dynamic RAM (DRAM) 24, Static RAM (SRAM) 26, Digital Signal Processor (DSP) 28, Internal Hard Disk 30, External Memory Device 32 (connected for example via a SCSI port 34), External Network Devices 36 (communicating for example over an Ethernet Network 38, and connected via SCSI port 34), Display Device 40 (such as a CRT), Printer 42 (such as a PostScript device connected for example via a serial port 44), Keyboard 46, and Pointing Device 48 (such as a mouse, trackball, etc.) The interconnection and functioning of each of these components in an operating computer system are well known in the art. However, the function and significance of particular components will be discussed in further detail where appropriate.

While the discussion that follows describes the invention in terms of the software programming environment known commonly as object oriented software design, it will be appreciated that the functionality of the present invention is not limited to such environments. However, as support for one possible environment in which the present invention operates, a description of certain relevant details of object oriented programming environments is presented.

In an object oriented programming language, the fundamental data structures of the language are objects, programming elements that contain a specification of both data and actions. Data are specified in local variables, referred to as instance variables, contained within the object. Actions, or methods, are subprograms that operate on the variables. Methods are invoked by messages sent from one object to itself or other objects.

There are two fundamental object constructs: classes and instances. Classes define a set of variables and methods and can be thought of as templates for object types. Instances (or objects) are particular run-time instantiations of a class. An object provides storage for variables defined in its class. Each instance has its own version of those variables. When a message is sent to an object, the object looks to methods defined in its class to determine how the message is to be executed.

Classes are defined in terms of a class hierarchy. A class inherits all variables and methods from classes above them in the hierarchy, referred to as superclasses. The inheriting class may be referred to as a subclass or child of the superclass. The variables and methods inherited from a superclass are available to the inheriting class as if they were defined within the inheriting class. Note that if a subclass has multiple superclasses, all the variables and methods from each superclass are inherited. This is often referred to as multiple inheritance.

There are two types of variables that may be defined within a class. The first are class variables, whose storage is shared by all instances

of the defining class. Class variables are typically used for coordination between classes and instances. The second are instance variables, which are variables specific to each particular instance. That is, separate storage space is allocated for each instance variable of each instance.

Methods specify the action to be performed when a message is sent to an object. When the message is sent to an object, the message name is matched against method names defined for that object. The method actually executed is the method with the same name as the message that is defined lowest in that object's class hierarchy.

Messages can have parameters. When a message is sent, all actual parameters are fully evaluated before a method is located for that message. Methods have formal parameters, corresponding to the actual parameters in the message, and temporary variables. Temporary variables may be created when a message is executed, and are available only with the scope of that method while the method is executing.

Programs perform by sending messages to objects which invoke a method in an object's class or class hierarchy. Often, methods send messages to other objects, invoking other methods. Typically. each method returns a result to the sender. At the heart of many languages, and especially an object oriented language, is a program segment for interpreting and evaluating messages and keeping track of where to return when methods call other methods.

Object oriented programming environments can be implemented with many programming languages.

One common example of an object oriented environment is the C++ environment, described in further detail in Ellis, Margaret and Bjarne Stroustrup, THE ANNOTATED C++ REFERENCE MANUAL (Addison-Wesley 1990) and Coplien, James O., ADVANCED C++ PROGRAMMING STYLES AND IDIOMS (Addison-Wesley 1992).

According to one embodiment of the present invention, a plurality of objects, referred to herein as qualifiers, are interconnected to form a network, referred to herein as an event qualifier network. Each event qualifier is an instance of one of a variety of event qualifier classes. Examples of such classes include KeyQualifier class, MouseQualifier class, RegionQualifier class, ButtonQualifier class, LogicQualifier class, StateCheck class, etc., described in further detail below.

In general, an event qualifier "qualifies" a specific event against a fixed criteria. That is, it represents a test for an occurrence of an event. According to one embodiment, an event qualifier causes an event manager or other event handling mechanism to be polled to determine whether a specified event has occurred. In another embodiment, the event qualifier registers an interest with the event manager, and is thereby alerted upon the occurrence of the event. If the specified event occurs, an action is invoked, principally by way of the event manager calling a function. The action could be a call-back function, a message, etc., and is discussed in more detail below.

Similarly, a state check qualifier "qualifies" a state of some aspect of the system against a

specified state. That is, it represents a test to determine whether a specified aspect of the system is in a desired state - i.e., whether the specified condition is met. Examples of state include time, values of variables, attributes of display objects such as size, color, position, etc. State check qualifiers generally do not check or register with the event manager, but return in a specific fashion an answer to the question: Is the aspect of the system in a specified state? If the state check qualifier returns an indication that the aspect is in the specified state, a function is called, which is described in further detail below.

A logic qualifier "qualifies" a relationship between two other objects. That is, a logic qualifier determines whether a specified logical relationship exists between two objects such as event qualifiers, state check qualifiers, other logical qualifiers or combinations thereof. Logic qualifiers invoke actions in response to satisfaction of the logical relationship they define (e.g., AND, OR, NOT, etc.), as further described below.

The event qualifier network includes at least one event qualifier and at least one other element such as another event qualifier, a state check qualifier or a logic qualifier. The event qualifier network determines whether the specified events have occurred, whether the specified aspects of the system are in their specified states, and/or whether the specified logical relationships among qualifiers exists. For example, an event qualifier network may determine whether one of two specified events has occurred, whether both of two events have occurred, or whether an event has occurred

while an aspect of the system is in a desired state.

With regard to Fig. 2, there is shown therein application software 50 including an embodiment of the present invention, and comprising a plurality of objects 52, 54, and 56, and system software 60 comprising a plurality of objects 62, 64, and 66. Objects 52 and 54 may be of virtually any type, and application software 50 may include an event qualifier object 56. Likewise, objects 62 and 64 may be of virtually any type, and system software 60 may include an event manager object 66 (or similar mechanism for maintaining, prioritizing, and otherwise handling events). While, for purposes of illustration, event qualifier object 56 is shown "contained" within application software 50, it will be appreciated that it is not a physical part of the application, but rather associated with the application, as will be apparent from the following discussion. Objects 52 and 54 of application software 50 may rely on the event qualifier network object 56 for notification of the occurrence of an event or combination of events, and objects 62 and 64 may produce the events that the event qualifier network 56 is checking for. In general, control will be passed from event qualification network 56 to the event manager 66, and data will be passed from the event manager 66 to the event qualifier network 56.

With regard to Fig. 3, there is shown therein an exemplary event qualification network 70 which shall be referred to in the description of class definitions according to one embodiment of the present invention. Event qualification network 70 is comprised of three event qualifiers. Event

qualifier 72 is an instance of the KeyQualifier class, and identifies a function called in response to a key-down event of the letter "A" key. Event qualifier 74 is an instance of the MouseQualifier class, and identifies a function called in response to a mouse-down event. Event qualifier 76 is an instance of the LogicQualifier class, and identifies a function called in response to the "A" key-down event AND the mouse-down event.

In essence, KeyQualifier 72 checks to see whether the key assigned to that instance has been depressed, MouseQualifier 74 checks to see whether the mouse has been depressed while the cursor is in a region of the screen defined for that instance, and LogicQualifier 76 determines whether the defined logical relationship, AND in this case, is satisfied by the instances of KeyQualifier 72 and MouseQualifier 74. That is, in this example event qualifier network 70 tests to see if the letter "A" key and the mouse key have been depressed simultaneously, and if so, calls a certain function identified by LogicQualifier 76.

Definitions of the basic class structures of the qualifiers employed in this example are shown in the following Table 1.

Table 1

1. KeyQualifier
   A. Variable
      i. storage for key definition for key-down
   B. Methods
      i. definition of particular "key" for key-down event
      ii. identification of function (e.g., notification to client) called upon key-down event for particular key
      iii. identification of function (e.g., registering interest with event manager) for determining whether event manager has a key-down event for the particular key

2. MouseQualifier
   A. Variable
      i. storage for region definition for mouse-down
   B. Methods
      i. definition of region of screen in which cursor must be located upon mouse-down event
      ii. identification of function (e.g., notification to client) called upon mouse-down event when cursor is in defined region of screen
      iii. identification of function (e.g., registering interest with event manager) for determining whether event manager has a mouse-down event in defined region

3. LogicQualifier
   A. Variable
      i. storage for qualifier handles
   B. Methods
      i. setting of logical relationship between return of function calls from "connected" event qualifiers (satisfaction of logical relationship results in a LogicQualifier event)
      ii. identification of function called upon LogicQualifier event
      iii. identification of function (e.g., registering interest with event qualifiers) for obtaining return of function calls from connected event qualifiers

It will be appreciated that many additional classes of qualifiers may be similarly defined, for specific events, states of aspects of the system (i.e., any condition that can be evaluated to True of False) or logical relationships required by a particular application. For example, event qualifiers may be established to check for network activity, to distinguish among various keys of a mouse, track ball, remote control device, etc.

In building an application, a programmer establishes the event qualification network appropriate for each desired event, event combination or other condition. Alternatively, event qualification networks may be maintained in a library from which the application developer may draw, incorporating the appropriate network(s) into the application. The network may be built

according to the syntax of the particular development language in which the application is being developed. For example, the network of Fig. 3 may be built in an application developed in the C++ environment as illustrated in Table 2:

Table 2

```
KeyQualifier
     kq = new keyQualifier;
     kq → SetKeyto ('A');
MouseQualifier
     mq = new mouseQualifier;
     mq → SetRegion (region);
LogicQualifier
     lq = new logicQualifier;
     lq → SetLogicQualifier (kq, mq, AND,
timeimportant);
```

In the above example, the SetKeyto and SetRegion methods cause an interest to be registered with the event manager. This means that a data structure 80 representing three parameters is passed to the event manager of the type shown in Fig. 4. Data structure 80 includes three fields, the first field being an event qualifier identification field 82, the second being an event type field 84, and the third being a function address field 86. The event qualification identification field 82 identifies which event qualifier is being passed to the event manager. The event qualifier identification may be a numerical value, or other identifier as appropriate. The event type field 84 identifies what event type the event qualifier is "interested in." That is, the event type field 84 identifies

the event type for which the event manager should call the function identified in the function address field 86. The available event types will be application-specific, but may include key, mouse, button, etc., events as previously discussed. As mentioned, the function address field 86 identifies the address of a function which the event manager will call upon the occurrence of the event specified in field 84. Calling (invoking) this function may result in action within the application, generation of a new event, or other appropriate action.

Integral to the functioning of the present invention is the processing of data structure 80. In order for an event qualifier to indicate whether a specific event has occurred, some action must occur with regard to the event qualifier when the specific event occurs. Thus, the event manager must register the interest of the event qualifier in the specific event, and must call the function identified in field 86 when the event occurs. This functionality may be built into an event manager, and certain event managers have this capability. If a particular event manager is not capable of such registration and notification, this functionality may be introduced by an intermediate event manager 90, in accordance with the architecture 92 illustrated in Fig. 5. As illustrated in Fig. 5, an event qualification network 94 registers its interest in an event by transmitting the data structure 80 of Fig. 4 to the intermediate event manager 90. The field data of data structure 80 is then added to tables which intermediate event manager 90 maintains with regard to interested event qualifiers. Intermediate event

manager 90 then polls event manager 96 to determine the occurrence of events as known in the art. When intermediate event manager 90 detects an event of the type in which a registered event qualifier is interested (that is, of a type specified in the event type field 84 of data structure 80 of Fig. 4), intermediate event manager 90 calls the appropriate function (i.e., the function at the address identified in field 86 of data structure 80).

The architecture of Fig. 5 allows the event qualification network of the present invention to operate in conjunction with event managers of the type commonly provided in certain computer system environments, such as single-task processing environments. For example, in the system environment of the Apple Macintosh line of computers, the intermediate event manager may loop, repeatedly calling WaitNextEvent to retrieve the available event. A CASE statement may then be used to take action based on the event type. Alternatively, it may be desirable to examine the pending event while leaving it available for subsequent retrieval by WaitNextEvent. This may be accomplished by the EventAvail function. An exemplary intermediate event manager routine is shown in table 3.

Table 3

```
KPoint              pt, lastpt;
KEventRecord        event;
char                theChar;

usesClass(CAppl);
usesClass(CMouseServer);
initServerManager();
initEventManager();
new(CAppl);

for (;;)
{
     if (K_WaitNextEvent(KeveryEvent, &event, 0,0))
     {
          switch (event.what)
          {
          case 1:         /* mouse down */
               lastpt.h = 0; lastpt.v = 0;
               while (K_WaitMouseUp())
               {
                    K_GetMousc(&pt);
                    if ((pt.v ! = lastpt.v) || (pt,h
                    ! = lastpt.h))
                    {    lastpt=pt;
                         mouse_task (pt, (long)1);}
               }
               K_GetMouse(&pt);
               mouse_task(pt,  (long)O);
               lastpt = pt;
               break;
          case  2:        /* key down */
          case  3:
               theChar = event.message & Oxff;
               KeyTask(theChar);
               break;
          default:
               K_GetMouse(&pt);
               if ((pt.v ! = lastpt.v) || (pt.h ! =
               lastpt.h))
               {    mouse_task(pt, (long)0);
                    lastpt = pt;)
               if (!pt.h && !pt.v)
               return;
          }
     }
}
```

Alternatively, the event manager 96 may notify the intermediate event manager 90 of the need to be polled. In such a case, intermediate event manager 90 may simply be required to handle the registering of an event qualifier's interest in a specific event, as opposed to polling the event manager. This will be the case in certain multi-task processing environments, such as UNIX. In this type of environment, the above exemplary intermediate event manager of Table 3 may be employed, modified by removal of the looping portion thereof, such as the first "if" statement. In another embodiment of the present invention, the event manager is written specifically with the event qualification network in mind, and thus directly accepts the registration of event qualifiers without the need for an intermediate event manager.

It will be appreciated that the intermediate event manager serves a second, important purpose. In many cases it is desirable to provide the ability to "port" application software written for one computer operating system to a second computer operating system. In so doing, certain portions of the code of the application software must be rewritten in the syntax of the second computer

operating system. While the event qualifier network may be a complex body of code which may be operating system dependent, as illustrated above, the intermediate event qualifier is generally not an overly complex structure. Thus, the intermediate event manager may itself be rewritten to act as an interface, allowing the event qualification network to operate in the second computer operating system environment. This simplifies the porting process, and improves the reliability of a ported application.

Upon the occurrence of the specified event, the event manager (or intermediate event manager) calls a function at the address specified in field 86 of data structure 80 (Fig. 4). Typically, this function will determine whether the event which occurred is of the type in which the event qualifier is interested. For example, an event manager may call an event qualifier's function when a key-down event occurs. The function will then determine whether the key associated with that key-down event was the key in which it was interested, such as the "A" key. In order for the function to make this determination, it requires certain data from the event manager, which is illustrated by event record 100 of Fig. 6. Event record 100

comprises five fields, a What field 102, a Message field 104, a When field 106, a Where field 108, and a More field 110. The individual structures of these fields are specified in Table 4.

Table 4

```
typedef struct KPoint
{
     short v;
     short h;
}
typedef struct K-EventRecord
{
     short          What;      /* type of event */
      long          Message;   /* dependent on event
                               type */
      long          When:      /* time stamp */
      KPoint        Where;     /* pointing
                               coordinates */
      void          More;      /* additional buffer
                               (if any) for the
                               event */
}
K_EventRecord;
```

(See also the discussion of event record in the above-referenced INSIDE MACINTOSH, volume 1, page 1-249.) The What field 102 contains an event code identifying the type of event. Each system will have a predefined set of event types. For example, events on the Macintosh computer are identified as:

```
     nullEvent =0;  {null event}
     mouseDown =1;  {mouse-down}
     mouseUp   =2;  {mouse-up}
     keyDown   =3;  {key-down}
     etc.
```

(See, INSIDE MACINTOSH, Volume 1, page I-249.)

Message field 104 will contain information that is a function of the type of event. The precise structure of the Message field is system dependent, but on the Macintosh computer it is a 32 bit structure, the contents of which are discussed in INSIDE MACINTOSH, volume 1, page I-250. One possible list of the contents for Message field 104 for a system according to one embodiment of the present invention is given in Table 5.

Table 5

Event type	Contents of EventRecord.message
TOUCH_DOWN_EVNT	undefined
TOUCH_UP_EVNT	undefined
FLXBUT_DOWN_EVNT	bit 0-7:which button
FIXBUT_UP_EVNT	bit 0-7:which button
KEYBD_KEY_DOWN_EVNT	bit 0-7:character button; bit 8-15 key code
KEYBD_KEY_UP_EVENT	bit 0-7:character button; bit 8-15 key code
PLAY_KEY_DOWN_EVNT	undefined
PLAY-KEY-UP-EVNT	undefined
STOP_KEY_DOWN_EVNT	undefined
STOP_KEY_UP_EVNT	undefined
POWER_KEY_DOWN_EVNT	undefined
POWER_KEY_UP_EVNT	undefined
IR_TEXT_EVNT	bit 0-7:IR device code; bit 8-15:the text
IR_BUT_DOWN_EVNT	bit 0-7:IR device code; bit 8-15:which button
IR_BUT_UP_EVNT	bit 0-7:IR device code; bit 8-15:which button

When field 106 may contain data about when (e.g., relative to the system clock) the event occurred. Where field 108 may contain data about

where the cursor position was on the screen at the time of the event. Finally, More field 110 is an optional field which may be used to contain data relating to other states of the system at the time the event occurred. The data and use of the data contained in More field 110 is provided as a developer/user convenience. It may but need not be present, and there may be multiple More fields (not shown) as appropriate for a particular application.

Thus, the function may determine if the event is of the type in which the event qualifier is interested by comparing the value of the message field 104 with a predefined value representing the desired particular event type (e.g., a key-down of the letter "A"). The function of the other fields of data structure 100 will become apparent in the following discussion.

In a manner roughly equivalent to the registration of an event qualifier with the event manager (or intermediate event manager), each logic qualifier will register its interest in the outcome of a determination made by the function of an event qualifier or in the state of an aspect of the system returned by a state check qualifier. For example, returning to Fig. 3, logic qualifier 76 will register its interest in the event

determinations of event qualifiers 72 and 74. This registration may be accomplished, for example, by passing the address of a function that is to be called, similar to the procedure described above. When a called function determines that an event it is interested in has occurred, that function calls the registered logic qualifier's function. In most circumstances there will only be one logic qualifier registered with an event qualifier, but the above mechanism may be employed to register multiple logic qualifiers in appropriate circumstances.

The functions of the logic qualifier called by the event qualifiers may be, for example, set flag functions. In the case of the example of Fig. 3, upon a determination that the "A" key has been depressed a first flag may be set. Likewise, upon determination that a mouse-down event has occurred, a second flag may be set. Referring once again to Fig. 6, part of the data structure 100 passed to an event qualifier upon occurrence of an event is the system time at which an event occurred, located in When field 106. Thus, each event qualifier is provided with a relative measure of when the event in which that qualifier is interested occurred. This information may then be used to determine if

the events occurred proximate enough in time to be considered "simultaneous" in order to satisfy an "AND" logic qualifier. For example, the parameter passed to the event qualifier function may, in turn, be passed to the logic qualifier function.

Ultimately, the application must be notified of the occurrence of the events and states of aspects of the system. This may be accomplished in the manner described above. That is, the application has a registered interest in the logic qualifier such that if the logic qualifier is satisfied (i.e., its flags are set and, if important, the events occurred sufficiently proximate to one another in time), an application's function is called by the logic qualifier, and a parameter is passed for execution of that function. To complete the example of Fig. 3, if logic qualifier 76 is satisfied (i.e., the "A" key and the mouse button are depressed simultaneously) the application's function of selecting (or highlighting) all text could be invoked.

Since all checking for events is handled by the event qualification network, including determination of the logical relationship between events, an application need not poll the event manager itself for the occurrence of events, as is

done in the prior art, but may idle, or perform other functions while waiting for the function call from the logic qualifier.

It will be appreciated that since the event record 100 passed by the event manager may contain data representing the time an event occurred, the position of the cursor at the time the event occurred, as well as other states of the system at the time the event occurred, great flexibility is provided in establishing the function invoked upon satisfaction of the logic qualifier. That is, the application may be provided significant data about the event and the state of the system when the event occurred which may be used by the application upon notification that the desired event(s) and conditions occurred.

It will also be appreciated that great flexibility is provided in defining events. That is, events are typically identified by number, and according to one embodiment of the present invention the event identification is a 16 bit number. Therefore, over 65,000 different events may be identified. Again, event definitions may be system-specific. Furthermore, "synthetic" events (i.e., events disconnected from the user) may be evaluated by the network, for example from files

recording user sessions, by means of a journaling mechanism such as that disclosed in INSIDE MACINTOSH, Volume 1, pages I-261-62.

Additional details with regard to events and event handling are discussed in chapter 5 of the Consumer Operating System (COS) Engineering Requirements Specification (ERS), a copy of which is attached hereto as Appendix A, and incorporated herein by reference.

Fig. 7 illustrates another exemplary event qualifier network 150, slightly more complex than that illustrated in Fig. 3. In this example, the event qualifier network 150 may be built into a consumer game device, which includes a remote control pointing and selecting device for interacting with a game displayed on a monitor. Buttons from the remote control device generate events, for example by causing an infra red signal to be emitted by the remote control device which is detected by a base unit containing the operating system and application program. Network 150 is also capable of checking for the position of a pointing object controlled by the remote control and displayed on the screen. Finally, network 150 is capable of checking for the state of certain aspects of the system. Each of these checks is

handled by an event qualifier and/or state check qualifier as follows.

Event qualifier 152 qualifies the button-down event for a button 1 on the remote control device. State check qualifier 154 qualifies the state of an aspect 1 of the system (for example, whether a user-selectable feature is enabled). Logic qualifier 162 is registered with qualifiers 152 and 154, and invokes an action when button 1 is depressed and aspect 1 is in the predetermined state. Likewise, event qualifier 156 qualifies the button-down event for a button 2 on the remote control device. Event qualifier 158 qualifies whether the pointing device is in a particular predefined region of the screen, for example over a target, in a corner, etc. Again, Logic qualifier 164 invokes an action when button 2 is depressed and the pointer is in the predefined region on the screen. Finally, event qualifier 160 qualifies the button-down event for a button 3 on the remote control device. Logic qualifier 166 is registered with event qualifier 160 such that it invokes an action only in the absence of a button-down event for button 3. Logic qualifier 168 qualifies two other logic qualifiers, 164 and 166. It is an AND logic qualifier such that it invokes an action if

button 2 is depressed while the cursor is in region 1 of the display AND while button 3 is not depressed. Logic qualifier 170 is an OR qualifier. It qualifies logic qualifiers 162 and 168 such that it calls a function of the application 172 if button 1 is depressed AND aspect 2 is in a predefined state OR if button 2 is depressed while the cursor is in region 1 of the display AND button 3 is not depressed. Arbitrarily complex logical networks may be built in a similar fashion for the specific needs of the application programmer.

In general, to those skilled in the art to which this invention relates, many changes in construction and widely differing embodiments and applications of the present invention will suggest themselves without departing from its spirit and scope. For example, qualifiers may be created in a graphical user interface environment, in a programming language environment, imported from other applications, etc. Furthermore, event qualification networks may be masked in appropriate circumstances to allow portions of a network to monitor events while segregating other portions from events. Thus, the disclosures and descriptions herein are illustrative, and are not

intended and should not be read to be in any sense limiting.

Appendix A

The Event System

This chapter discusses the concepts of the Event System, then lists the methods and functions for the object in the Event System.

Events

The SweetPea machine provides various ways for end-users to communicate with the machine. For example, it has various buttons that can be pressed up or down (such as the power button), and the screen is touch-sensitive. It also supports infra red remote pointing devices and keypads. The event system in COS is responsible for communicating all hardware input events to the application. For example, when a person touches the screen, uses a remote control or presses a button on Sweetpea, the COS event system sends a software signal that the event has happened.

The response to the software event signal is completely up to the client application. Objects in the application can ask to be notified whenever a particular type of event happens, and can specify what they want to do when they receive the notification. If no objects have expressed an interest in a type of event, then when that type of event happens, the application does not respond in any way. Similarly, if several objects express an interest in a particular type of event, then when that event happens they are all notified and all respond in their own way.

For example, when you press down on the screen, COS generates a touch-down event signal. It continues generating touch-down event signals every 8 milliseconds while you continue touching. When you release your finger from the screen, COS generates a touch-up event signal. If no objects have expressed an interest in the touch down or touch up events, then when someone touches the screen, nothing happens. However, suppose three different DigitalAudioPlayer objects have specified that they will play their sound when they receive notification of a touch down event. When someone touches the screen, the event server notifies each of the three

players in turn, and they each play their sound.

Event Servers

The SweetPea machine supports the following devices for user input:

- Touch-sensitive screen.
- Physical buttons on the hardware for play, stop, and power on/off.
- Virtual buttons at the bottom of the screen. There are five of these, and they are known as "fixed buttons."
- Pointing devices.

COS supports IR and non-IR pointing devices, and absolute and relative pointing devices.

Absolute devices let you point to an absolute place on the screen. Relative devices let you point to a position some distance away from the current position, for example five up and seven to the right from the current position.

- Remote control IR keypad. This enables you to type text for into a remote control keyboard.
- Remote control devices with buttons (like the remote control devices for vcrs and televisions.) These have buttons that each do different things when pressed, such as rewind and fast forward.

COS has software server objects that communicate with the hardware input devices. It has one server for each type of device, as follows:

- CTouch Server. This object communicates with the touch screen.
- CIRServer. This object communicates with infra red remote control devices.
- CSyskeyServer. This object communicates with the "Play", "Stop" and power on/off buttons on the hardware, and with the CD device.

- CRelPtServer. This object communicates with relative pointing devices.
- CAbsPtServer. This object communicates with absolute pointing devices.
- CICServer. This object communicates with the IC card slot.
- CPowerServer. This object communicates with the power supply.

COS creates and initializes the event servers. There is no need for your application to do any event server initialization if it is using the standard set of events.

Types of Events

When someone uses an input device to communicate with Sweetpea, the software server object for that device generates an event signal. Additionally, whenever the battery starts running low, COS generates a low-battery event.

The servers generate the following event signals:

CTouchServer events

KTOUCH_DOWN_EVNT	The touch-sensitive screen is pressed down. A new KTOUCH_DOWN_EVNT is generated every 8 milliseconds while the screen is being touched.
KTOUCH_UP_EVNT	The touch on the screen is released.

CSyskeyServer events

KFIXBUT_DOWN_EVNT	A fixed button is pressed down.
KFIXBUT_UP_EVNT	The touch on a fixed button is released.
KPLAY_KEY_DOWN_EVNT	The "Play" button is pressed down.
KPLAY_KEY_UP_EVNT	The "Play" button is released.
KSTOP_KEY_DOWN_EVNT	The "Stop" key is pressed down.

KSTOP_KEY_UP_EVNT	The "Stop" key is released.
KCD_TOP_OPEN_EVNT	The door for the CD is opened.
KCD_TOP_CLOSE_EVNT	The door for the CD is closed.
KPOWER_KEY_DOWN_EVNT	The power key is pressed down.
KPOWER_KEY_UP_EVNT	The power key is released.
CPowerServer events	
KPOWER_EVNT	The battery power is running low or the IC card has a problem.
CIRServer events	
KIR_TEXT_EVNT	A character is sent from the remote keypad device.
KIR_ABS_DOWN_EVNT	The absolute pointing device is pressed down.
KIR_ABS_UP_EVNT	The absolute pointing device is released.
KIR_BUT_DOWN_EVNT	A button on the remote control device is pressed down.
KIR_BUT_UP_EVNT	A button on the remote control device is released.
KIR_REL_DOWN_EVNT	The relative pointing device is pressed down.
KIR_REL_UP_EVNT	The relative pointing device is released.
CAbsPtServer events	
KABS_POINT_DOWN_EVNT	The absolute pointing device is pressed down.
KABS_POINT_UP_EVNT	The absolute pointing device is released.

CRclPtServer events

KREL_POINT_DOWN_EVNT	The absolute pointing device is pressed.
KREL_POINT_UP_EVNT	The relative pointing device is released.

CICServer events

KICCARD_UNLOCK_EVNT	The IC card housing is unlocked.
KICCARD_LOCK_EVNT	The IC card housing is locked.

Event Handlers

Objects that want to be notified when an event happens must register with the appropriate event server. For example, objects that want to be notified when someone touches the screen must register with the CTouchServer object. Objects that want to be notified when someone uses a relative pointing device must register with the CRelPtServer object, and so on.

The term "client objects" will be used in this chapter to mean objects that have registered with an event server.

When client objects register with a server, they must also specify which of their methods will be called when the event happens.

The term "client method" will be used in this chapter to mean the method on a client object that will be called when the appropriate event happens.

The event Manager object knows about all the event servers in the system, and knows what each one can do. To register a client object with an appropriate server, you call the registerEventHandler routine, and tell it the client object, the client method, and what type of event the client object is interested in. The eventManager then finds the event server that handles that kind of event. Then it creates an eventHandler object and attaches it to the server.

You can think of the event handler as a messenger for the client object. Its duty is to wait at the server until the server signals an event. Then the event handler collects

information about the event (what happened, when and where) and returns to the client object with the information so the client object can act upon the information.

Steps To Make an Application Respond to Events

To make your application do something in response to an event, do the following:

1. Create the client object that is to respond to the event, if it does not already exist.

2. Write the method on the object that does the response action.

   The argument to this method (in addition to self) is the event object. The event object contains an event record. You can get the event record calling the getEventRec method on the event object. Then you can get the value of the what, when, where, message and more fields to get the event type, time, location, and extra information about the event. See page A-9 for a discussion of event records.

If the method is to swallow the event, it must set the value of the event handler's SetSwallow instance variable to true. See page A-7 for a discussion of event swallowing.

3. Register the client object with the appropriate event server by calling the registerEventHandler routine. This routine requires the following arguments:

   - object to be registered
   - event type
   - method to be called on client object when event occurs

That's it. When an appropriate event happens, the client method will be called automatically.

Code Example

For example, suppose every time the screen is touched, you want your application to respond by displaying a picture of a band aid centered on the touched point for five seconds.

You would need the following:

- A routine called ShowBandAid that takes one argument of a point. It displays a picture of a bandaid centered around the input point. After five seconds it erases the bandaid from the screen.

- An object called HurtItem that has a method called Hurt, whose value is the following functions:

```
method void

_Hurt (object self, object evenObj)
{
 K_EventRecord erec;
 kPoint        touchedPoint;

getEventRec (eventObj, &erec);
touchedPoint = erec. where;

ShowBandAid (touchedpoint);
}
```

Now you need to register the HurtItem object with a server that can handle touch down events as follows:

```
registerEventHandler (HurtItem,
KTOUCH_DOWN_EVNT, Hurt);
```

Whenever the screen is touched, the Hurt method is called on the HurtItem object, and the sore spot is temporarily covered by a bandaid, as illustrated in Figure 9.

Low Battery Events

Whenever the battery starts to run low, the power generates an event signal of type KPOWER_EVNT. However, unless you have an object in your application that has registered to be notified when a KPOWER_EVNT signal is triggered, nothing will happen in response to the signal.

It is highly recommended that an application always registers an object to respond to low battery events. For example, it could give a spoken warning or print a message on the screen alerting the user that the power is low.

Swallowing

When a hardware input or a low battery event happens, the relevant event server sends each of its attached event handlers in turn to notify the client objects. However, there may be times when there's no need to notify all the clients objects.

For example, suppose the application has created ten buttons and displayed them on the screen. Each of the displayed buttons corresponds to an object that has registered with the touch server. The desired result when someone touches the screen is that only the touched button responds. The other buttons do nothing. In this case it would be desirable for the event notification process to halt as soon as the object corresponding to the touched button has been notified.

To halt the notification process, a client method can "swallow" an event. When the event is swallowed, the event server does not continue notifying its remaining clients.

To continue with the example of the ten buttons:

When someone touches the screen, the touch server notifies the first of its registered client objects (via the appropriate EventHandler object.) If this object corresponds to the button that was touched, the client method on the button object can swallow the event. If the object does not correspond to the button that was touched, the client method does nothing, and the server goes on to notify its next client. The server continues notifying its clients until either they have all been notified or one of them swallows the event.

If the first button notified is the one that was touched, then the notification process occurs once. If the tenth button checked is the one that was touched, then the notification process occurs ten times.

In this type of situation, the event method must have conditional code that checks if the client object needs to respond to the event or not. If it does need to respond to the event, swallow the event and then do whatever the object needs to do in response to the event.

If it does not need to respond to the event, then do nothing.

In order to swallow an event, the client must pass an argument of Ktrue to the SetSwallow method on the event handler which is attached to it.

In order to be able to call SetSwallow on the event handler later, the client object must know which event handler is attached to it. One way to tell the client object which event handler is attached to it is to create an instance variable called, for example, attachedEH, and put the value returned by registerEventHandler into it. Then when the event happens and the client method is called, it finds the attached event handler by getting the value of that instance variable.

Event Objects and the Event Record Data Structure

When an event occurs, the appropriate event server generates an event object and attaches it to each of its event handlers in turn.

The erec instance variable of the event object holds an event record that contains all the necessary information about the event, such as what the event was, when it happened, and any other extra information. The extra information depends on the type of event. For example, if a fixed button was pressed, the event record records exactly which of the fixed buttons was touched.

When a client method is called as the result of an event happening, it is passed the event object for the event. It can get the event record from the event object by calling the event object's get_EventRec method. It can then extract the values of the fields in the event record data structure. For example, to find when the event happened, it gets the value of the when field.

The definition for the event record data structure is:

```
typedef struct K_EventRecord {

short  what     /* type of event */
long   message; /* varies per event type */
long   when;    /* time stamp */
KPoint where;   /* co-ordinates of touched
                or */
```

```
                    /* pointed at position */
    void    *more;  /* additional buffer (if
                    any) */
                    /* for the event */
    }  K_EventRecord;
```

The value of the message field depends on the type of event, as follows:

Event Type	Contents of message field
TOUCH_DOWN_EVNT	undefined
TOUCH_UP_EVNT	undefined
FIXBUT_DOWN_EVNT	bit 0-7: which button. 1 is button #1, 2 is button #2 and so on. Button #1 is the left-most button
FIXBUT_UP_EVNT	bit 0-7: which button.
PLAY_KEY_DOWN_EVNT	undefined
PLAY_KEY_UP_EVNT	undefined
STOP_KEY_DOWN_EVNT	undefined
STOP_KEY_UP_EVNT	undefined
POWER_KEY_DOWN_EVNT	undefined
POWER_KEY_UP_EVNT	undefined
IR_ABS_DOWN_EVNT	bit 0-7: IR device code
IR_ABS_UP_EVNT	bit 0-7: IR device code
IR_REL_DOWN_EVNT	bit 0-7: IR device code
IR_REL_UP_EVNT	bit 0-7: IR device code
IR_TEXT_EVNT	bit 0-7: IR device code bit 8-15: the text
IR_BUT_UP_EVNT	bit 0-7: IR device code bit 8-15: which button
IR_BUT_DOWN_EVNT	bit 0-7: IR device code bit 8-15: which button
CD_TOP_OPEN_EVNT	undefined
CD_TOP_CLOSE_EVNT	undefined

POWER_EVNT	bit 0-7: MAIN_BATTERY - Main Battery is low. IC_CARD_VCC_FAIL - The IC card cannot operate. IC_CARD_VCC_FAIL - The IC card cannot be programmed.
ICCARD_LOCK_EVNT	undefined
ICCARD_UNLOCK_EVNT	undefined

How the Event Notification System Works

COS takes care of generating event signals whenever user input is received or the battery starts to run low. The client application does not have to worry about generating events, but it must create the objects and methods that respond to the events, and must register those objects with the event manager.

The step-by-step details of the event notification system are explained here, since it may help you to understand how the event notification system works.

In brief: when an event happens, the relevant event server notifies all the objects that have registered with it, and they each do their thing in response.

In more detail: When an event happens (for example, the screen is touched), the event server (for example, the CTouchServer), creates an event object. The event object knows what kind of event happens, and has any other necessary information, such as the point of the screen that was touched or a number that indicates which button was pressed.

If the event server has no attached event handlers, nothing happens and the event is disregarded.

If the event server has one or more attached event handlers, the server attaches the event object to the first event handler that is interested in that type of event.

The event handler knows what method on which object is to be activated by the event. (This

method will be referred to as the client method in this context.)

The event server sends the event handler with the attached event object to the EventManager, where it waits in the dispatch queue. When it gets to the top of the queue, the EventManager dispatches the event, that is, it tells the event handler to execute the client method.

The client method is executed. The client method may use information on the event object, such as when the event happened, what point in the screen was touched or what button was pressed. The client method may or may not set the value of the Set Swallow instance variable on the event handler to true.

Control returns to the event server, which checks if the SetSwallow instance variable on the event handler is set to true. If it is set to true, the event is "swallowed" and nothing more happens. If it is not set to true, the event server gets the next event handler, and the process continues until either the event is swallowed or there are no more event handlers to consider.

The process that occurs when someone touches down on the screen is presented in a flow diagram in Figure 10.

Further Example-Putting Buttons on the Screen

Here is a simple example that shows how to create touch-sensitive buttons on the screen and how to swallow events.

This example shows how to display buttons on the screen that play a sound when touched. The buttons respond only when someone touches down on the button, it does not do anything when the touch on the button is released.

- Create a class called SoundButton.

  This class has the following instance variables:

type	IV name
method	playSound
method	touchDown

```
method          displayButton
Rect            rect
Cstring         label
object          player
object          eventHandler
```

- The new method takes arguments for self, label, left, top, right, bottom and player.

  It makes an instance of the class.

  It creates a rectangle structure from left, top, right and bottom and puts the rectangle in the rect instance variable on the new instance.

  It puts the label argument in the label instance variable.

  It puts the player argument in the player instance variable.

```
method object

_new (  class       self,
        char*       label,
        long        left,
        long        top,
        long        right,
        long        bottom
        object      myplayer)
(
  object button = NULL;
  ivType *iv;
  Rect   buttonRect;

  button = makeInstance (self);
  buttonRect = setRect(left, top, right,
  bottom);

  iv = ivs(button);
  iv->rect = buttonRect;
  iv->player = myplayer;
  strcpy (iv-.label, label);
)
```

- The displayButton method draws the button on the screen and displays the label in the button.

```
method void

-drawButton(object self)
{
  Rect        buttonRect = ivs(self)->
              rect;
  SXFont      myfont;
  KPoint      x, y;

  /*  Draw the button outline */
  PenSize(3.3);
  EraseRoundRect(&buttonRect,16,16);
  FrameRoundRect(&buttonRect,16,16);
  InsetRect(&buttonRect,4,4);
  PenSize(1,1);
  FrameRoundRect(&buttonRect,16,10);

/* Specify the font for the button label
*/
  myfont = SXNewFont(SXMakeFontName
  ("Chicago"), 14, kbold);

  /*  Specify where to start drawing the
  label */
  x = buttonRect.left + 4;
  y = (buttonRect.top +
  buttonRect.bottom) * .5 - 8.0;
  K_MoveTo(x,y);
  /* Draw the label */
  K_DrawText(myfont, invs(self)->label),
  0, 0, khorizontal);
}
```

- The playSound method plays the sound.

```
method void
_playSound(object self)
{
  object myplayer = ivs(self)->player;
  play(myplayer);
}
```

- The registerButton method registers the touchDown method to be called when someone touches down on the screen. It also puts the event handler created by the registration in the Handler instance variable on the button.

```
method void
_registerButton (object self)
{
   object eventH;

   eventH = registerEventHandler(   self, KTOUCH_DOWN_EVNT,
   playSound);
   ivs(self)->eventHolder = eventH;
}
```

- The touchDown method checks if the touched point was in the rectangle, and if it was, then inverts the button (to highlight it); calls the playSound method; inverts the button again (to unhighlight it) and swallows the event.

```
method void

-touchDown (object self, object eventobj)
{
   ivType              *iv;
   Rect                buttonrect;
   K_EventRecord       eventrec = getEventRec(eventobj);
   Point               thispoint;
   object              handler;

   iv  = ivs(self);
   buttonrect=iv->rect;
   handler = iv->eventhandler;

   thispoint = *eventrec.point;
   if (PtInRect(thispoint, &buttonrect)
       {     InvertRect(&buttonrect); playSound(self);
             InvertRect(&buttonrect);
             SetSwallow(handler, 1)
       }
}
```

- Suppose now you want to display a button labelled COW on the screen that plays the sound of a cow mooing when it is touched. The button is to be 100 pixels long and 50 pixels high, and its upper, left corner is at the point (100, 100) on the screen. You would do the following:

1. Create a DigitizedAudioPlayer object called CowPlayer that represents a file containing the digitized data for the sound of a cow mooing.

2. Call the new method on the SoundButton class, passing in the label, left, top, right, bottom and player.

   Call the drawButton method on the button instance.

   Call the registerButton method on the button.

```
{
   object cowbutton=NULL;

   cowButton = new(SoundButton, "COW", 100, 100, 200, 150,
                                CowPlayer);
   drawButton(cowButton);
   registerButton(cowButton);
}
```

3. Wait for someone to touch the button on the screen. The machine will moo.

Methods and Functions

CTouchServer Subclasses

The CAbsPtServer, CICServer, CIRServer, CPowerServer, CRelPtServer, CSyskeyServer and CTouchServer subclasses of CServer inherit the following methods needed for providing event services.

Instance Methods on event server subclasses

Name	Type	Argument Type/Class	Argument	Description
addHandler	object	object object long void	self client eventType *method	Registers the client object with the server. Usually an application does not need to call this method directly. Instead, use the register Event-Handler function, which calls this method as one of its tasks. eventype indicates which type of event the client is interested in. *method is the method on the client that is to be called when the event happens. This method creates an event handler object and attaches it to the server.
removeHandler	void	object object	self eventhandler	Removes the event handler from the server and disposes of the event handler object

EventManager

The EventManager object acts as a central dispatcher for all events. When an event happens, the appropriate event server creates an event object and sends it to the event manager. The event object waits in the event queue at the event manager until it reaches the top of the queue, then the event manager dispatches it.

Routines that use the EventManager

Name	Type	Argument Type/Class	Argument	Description
initEventManager	void		none	Initializes the event manager. COS calls this automatically, so there is usually no need for your application to call it.
registerEventHandler	object	object long void	clientObj eventType *method	Use this routine to register a client object with a server, for a given type of event. See page A3 for a list of event types. *method is the method on the client object that will be called when the event happens.
removeEventHandler	void	object	eventHandler	Removes an event handler from the server to which it is attached.
suspendAllEvent	void	long	timeout	Suspends all events. Events are disregarded by the dispatcher. timeout is the time in milliseconds until dispatching is to resume. A timeout of 0 indicates indefinite suspension.
resumeAllEvent				Resumes the dispatching of events.
hotEvent	void	object	eventhandler	The event for the event handler goes to the head of the event queue at the dispatcher.
dispatchEvent	void	object	eventhandler	Dispatches the event that the eventhandler is handling. The event manager calls this when an eventhandler reaches the head of the dispatch queue. This method should not be called directly by an application.
flushAllEvent	void		none	Flushes all events out of the dispatch queue.

CEvent

Class method on CEvent object

CEvent objects are created by event servers when an event happens. Application should not directly create event objects, but the details of the new method are given here in case they are needed for testing purposes.

Name	Type	Argument Type	Argument	Description
new	object	class K_EventRecord	self eventRecord	Creates a new CEvent object and puts the given event record as the value of its erec instance variable.

Instance Methods on CEvent objects

The creation, use and disposable of CEvent objects is handled internally by the COS event system. Your application should not directly call methods on event objects, but the methods are given here in case you need them for testing purposes.

Name	Type	Argument Type	Argument	Description
setEventRec	void	object K_EventRecord	self *eventRecord	Puts the given event record as the value of the event object's erec instance variable. This method is automatically called when a new event object is created.
getEventRec	void	object K_EventRecord	self *eventRecord	Returns the event record in the event object's erec instance variable.
dispose	void	object	self	Disposes of the event object.

CEventHandler

Class Methods on EventHandlers

CEventHandler objects are created automatically by the registerEventHandler function, which is used to register objects with event servers. Application should not directly create event handler objects, but the details of the new method are given here in case they are needed for testing purposes.

Name	Type	Argument Type	Argument	Description
new	object	class object object void	self server client *method	Creates a new event handler object. server is the server where the event handler is registered. client is the client object that wants to be notified when an event happens. *method is the method on the client that will be activated when an event happens. (This method is called by the addHandle method on a server, which also adds the event type to the event handler.)

Instance Methods on CEventHandlers

The creation, use and disposable of CEventHandler objects is handled internally by the COS event system. Your application should not directly call methods on event handler objects, except for SetSwallowed, which must be called by the client method on the client object represented by the event handler. However, other methods are given here in addition to the SetSwallowed method in case you need them for testing purposes.

Name	Type	Argument Type	Argument	Description
dispose	void	object	self	Disposes of the object.
exec	void	object	self	Executes the client method on the event handler's client object. This is normally called by the event manager's dispatchEvent function.
setEventObj	void	object object	this eventObject	Puts the given event object as the value of the eo instance variable on the event handler.

getEventServer	object	object	self	Returns the event record that is the value of the event handler's eo instance variable.
getEventClient	object	object	self	Returns the client object of the event handler.
swallowed	long	object	self	Returns the value of the event handler's swallowed instance variable, which indicates if the event should be swallowed.
setSwallow	void	object long	self swallow	Sets the value of the swallowed instance variable on the event handler to 0 or 1. If it's 0, it means the event is not swallowed. If it's 1, it means the event is swallowed. This method must be called by the method on the client object that the event handler represents.

THE CLAIMS DEFINING THE INVENTION ARE AS FOLLOWS:

1. A method for determining, on behalf of an application program, when a specified combination of logically related events has occurred, said method comprising the following steps:

providing an event qualifier network data structure ("EQN") associated with at least one piece of software code, the EQN specifying a logical relationship among the combination of events, the logical relationship comprising a plurality of logical qualifiers, each one of said logical qualifiers qualifying a group of one or more of the events, and wherein at least two of the logical qualifiers correspond to different logic functions;

registering an interest, on behalf of the application program, in being notified when the logical relationship specified by the EQN has been satisfied;

detecting the occurrence of each of the events;

notifying the EQN of each said occurrence by invoking a piece of software code associated with the EQN;

determining when the logical relationship among the combination of events specified by the EQN is satisfied; and

notifying the application program when the specified logical relationship among the combination of events has been satisfied.

2. The method of claim 1, wherein the logical qualifiers correspond to logic functions selected from the group comprising logical AND, logical OR, and logical NOT.

3. The method of claim 1, wherein the logical relationship specified by the EQN among the combination of events further comprises a qualifying condition corresponding to a proximity in time between two or more of the events.

AUSTRALIAN PATENT OFFICE

4. The method of claim 1, wherein the logical relationship specified by the EQN among the combination of events further comprises a qualifying condition corresponding to a sequence in which two or more of the events occur.

5. The method of claim 1, wherein the logical relationship specified by the EQN among the combination of events further comprises a qualifying condition corresponding to a location at which one or more of the events occurs.

6. The method of claim 1, further comprising the steps of registering an interest with an event manager on behalf of the EQN in being notified when each event occurs.

7. The method of claim 1, wherein the step of detecting the occurrence of each event, the step of notifying the EQN of each occurrence, and the step of determining when the logical relationship among the combination of events is satisfied, are all performed without the dedication of any application program execution time.

8. The method of claim 1, wherein the method is performed using a computer system, and wherein the logical relationship specified by the EQN among the combination of events further comprises a logical relationship between one or more of the events and a qualifying condition corresponding to the computer system being in a specified state.

9. An apparatus for determining, on behalf of an application program, when a specified combination of logically related events has occurred, said apparatus comprising:

an event qualifier network data structure ("EQN") associated with at least one piece of software code, the EQN specifying a logical relationship among the combination of events,

AUSTRALIAN PATENT OFFICE 13

the logical relationship comprising a plurality of logical qualifiers, each one of said logical qualifiers qualifying a group of one or more of the events, and wherein at least two of the logical qualifiers correspond to different logic functions;

means for registering an interest with the EQN, on behalf of the application program, in being notified when the logical relationship specified by the EQN has been satisfied;

event manager means for detecting the occurrence of each of the events, and for notifying the EQN of each said occurrence by invoking a piece of software code associated with the EQN;

means for determining when the logical relationship among the combination of events specified by the EQN is satisfied; and

means for notifying the application program when the specified logical relationship among the combination of events has been satisfied.

10. The apparatus of claim 9, wherein the logical qualifiers correspond to logic functions selected from the group comprising logical AND, logical OR, and logical NOT.

11. The apparatus of claim 9, wherein the logical relationship specified by the EQN among the combination of events further comprises a qualifying condition corresponding to a proximity in time between two or more of the events.

12. The apparatus of claim 9, wherein the logical relationship specified by the EQN among the combination of events further comprises a qualifying condition corresponding to a sequence in which two or more of the events occur.

13. The apparatus of claim 9, wherein the logical relationship specified by the EQN among the combination of events further comprises a qualifying condition corresponding to a location

AUSTRALIAN PATENT OFFICE 13

at which one or more of the events occurs.

14. The apparatus of claim 9, wherein the means for detecting the occurrence of each event, the means for notifying the EQN of each occurrence, and the means for determining when the logical relationship among the combination of events is satisfied, all operate without the dedication of any application program execution time.

15. A method for determining, on behalf of an application program, when a specified combination of logically related events has occurred, substantially as hereinbefore described with reference to the accompanying drawings.

16. An apparatus for determining, on behalf of an application program, when a specified combination of logically related events has occurred, substantially as hereinbefore described with reference to the accompanying drawings.

DATED this 16th day of October, 1996

KALEIDA LABS, INC.
By its Patent Attorneys
DAVIES COLLISON CAVE

AUSTRALIAN PATENT OFFICE 13

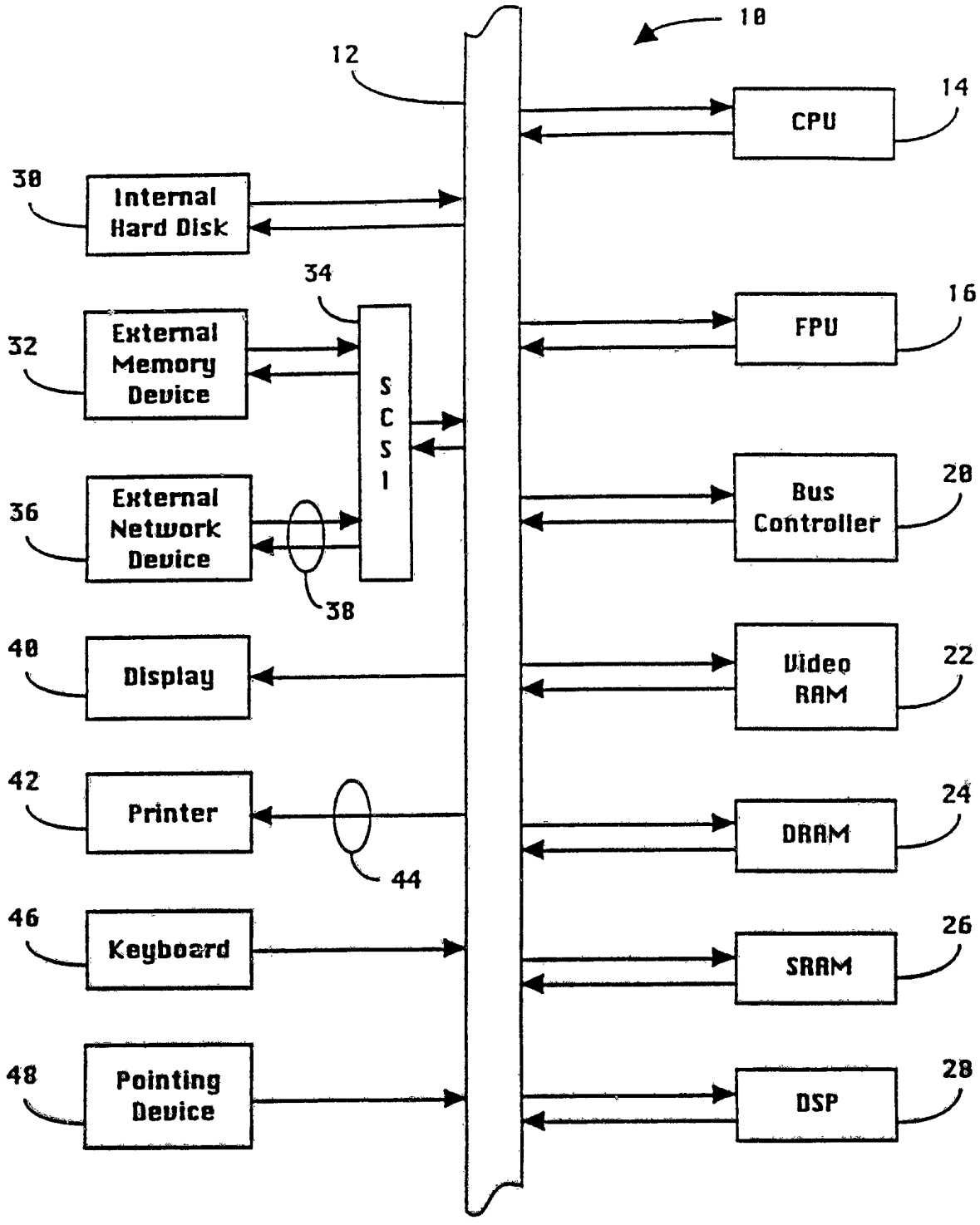

10
12
CPU
14
30
Internal Hard Disk
34
External Memory Device
32
SCSI
FPU
16
External Network Device
36
38
Bus Controller
20
40
Display
Video RAM
22
42
Printer
44
DRAM
24
46
Keyboard
SRAM
26
48
Pointing Device
DSP
28

Fig. 1

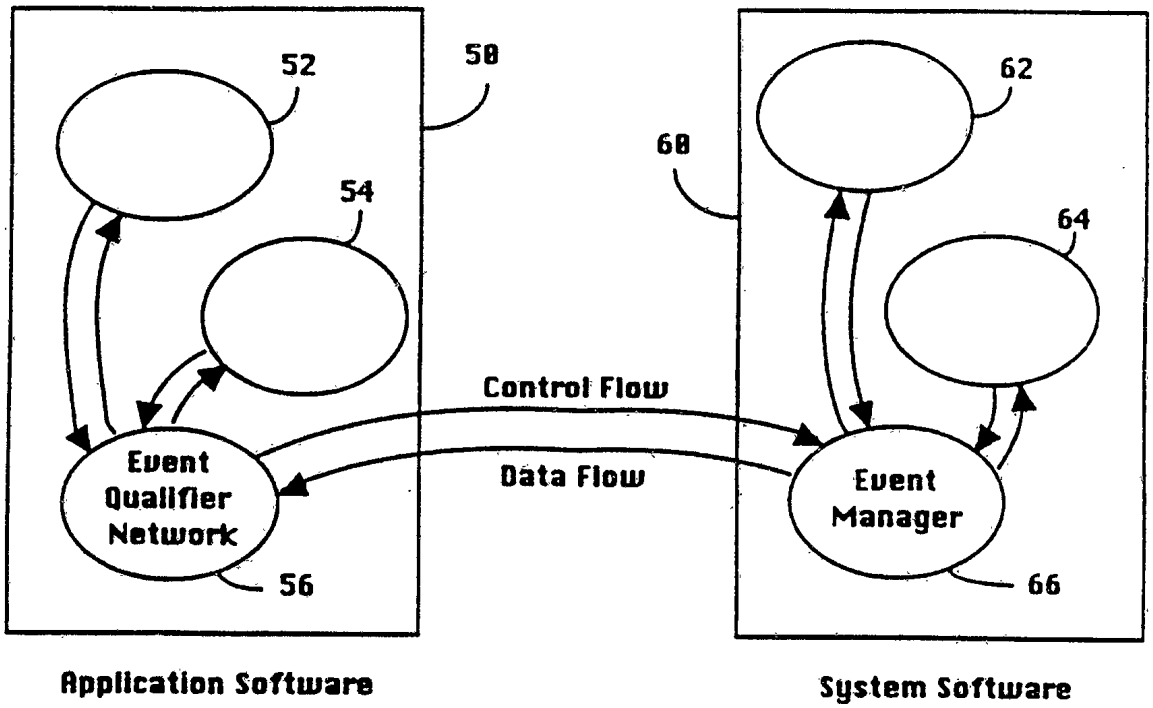

52
58
54
62
60
64
Control Flow
Data Flow
Event Qualifier Network
56
Event Manager
66
Application Software
System Software

Fig. 2

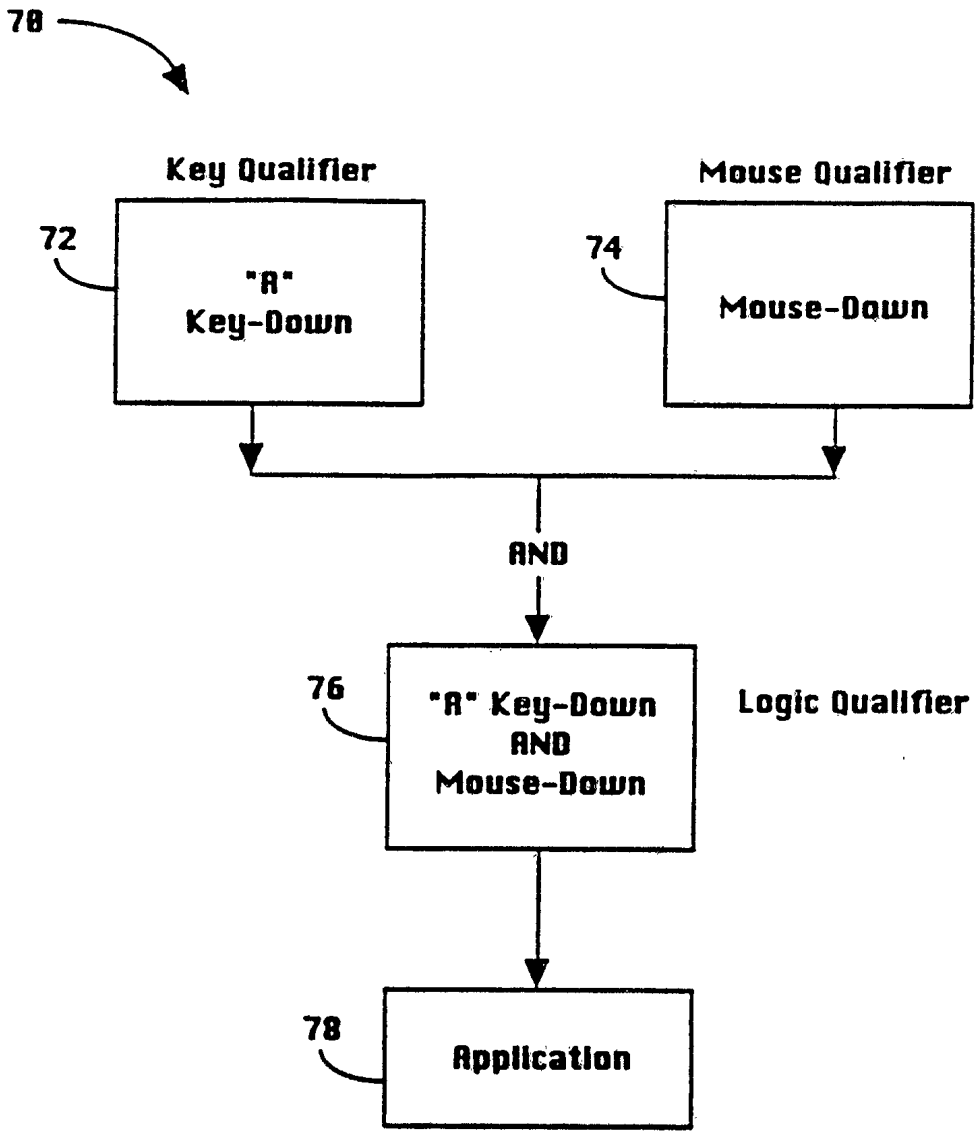

70
Key Qualifier
Mouse Qualifier
72
"A"
Key-Down
74
Mouse-Down
AND
76
"A" Key-Down
AND
Mouse-Down
Logic Qualifier
78
Application

Fig. 3

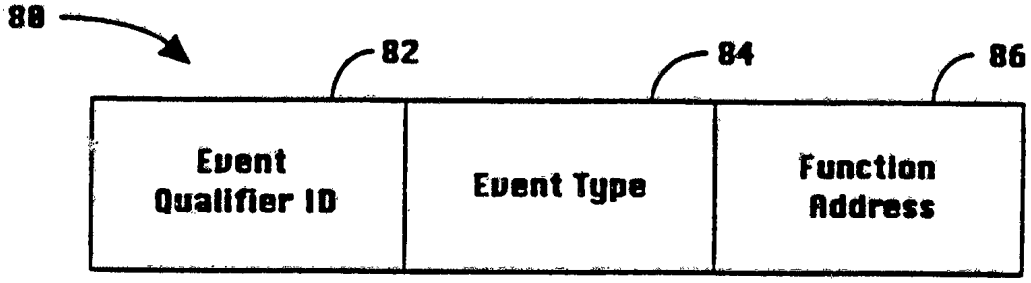

80
82
84
86
Event
Qualifier ID
Event Type
Function
Address

Fig. 4

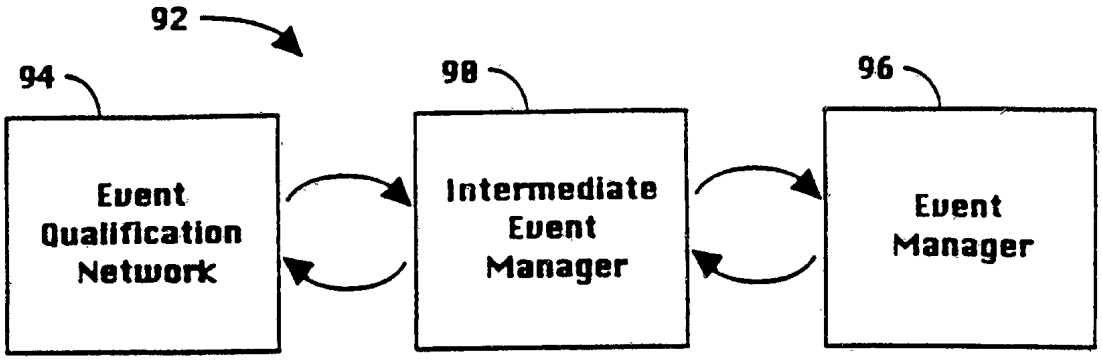

92
94
98
96
Event Qualification Network
Intermediate Event Manager
Event Manager

Fig. 5

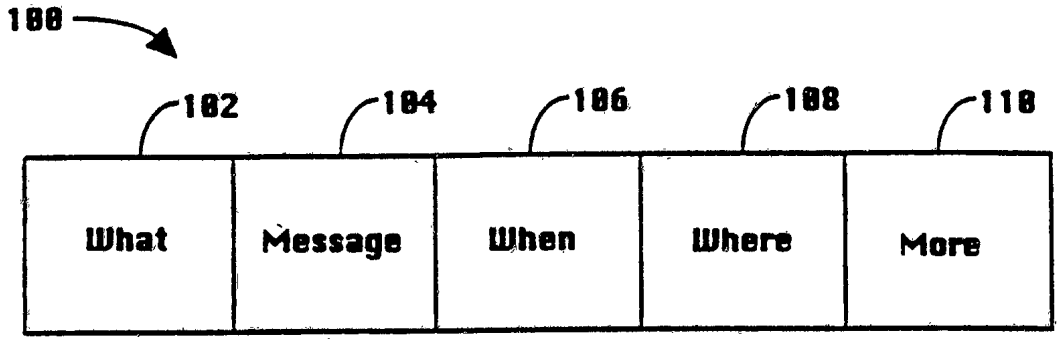

100
102
104
106
108
110
What
Message
When
Where
More

Fig. 6

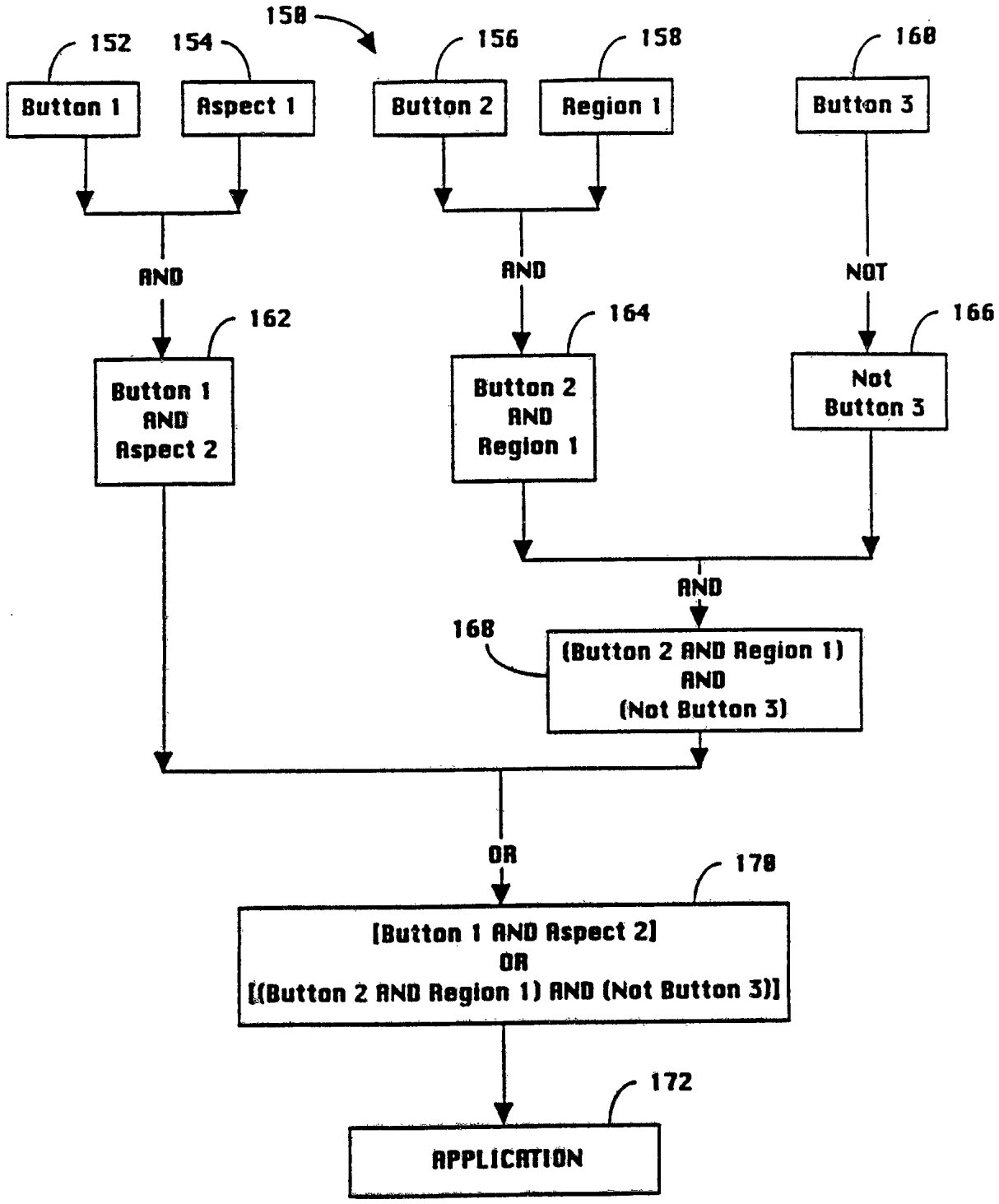

158
152
154
156
158
168
Button 1
Aspect 1
Button 2
Region 1
Button 3
AND
AND
NOT
162
164
166
Button 1 AND Aspect 2
Button 2 AND Region 1
Not Button 3
AND
168
(Button 2 AND Region 1) AND (Not Button 3)
OR
178
[Button 1 AND Aspect 2] OR [(Button 2 AND Region 1) AND (Not Button 3)]
172
APPLICATION

Fig. 7

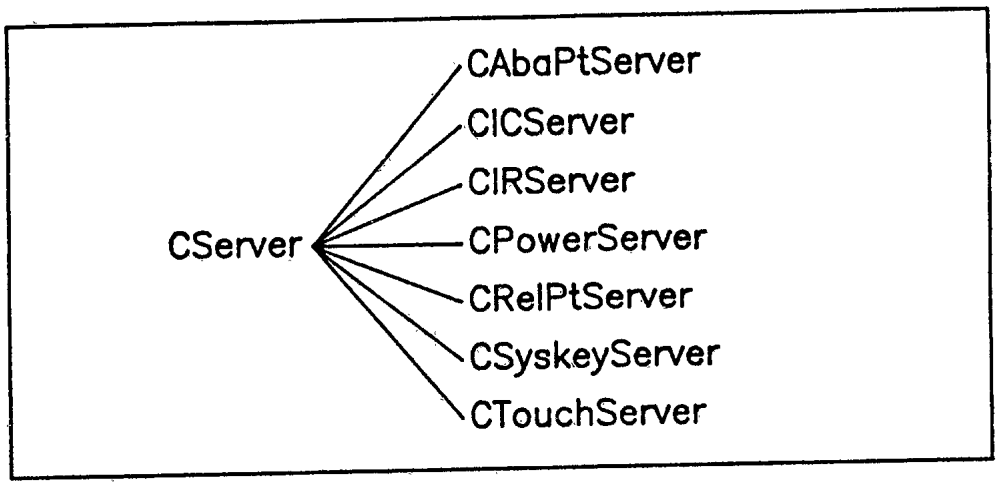

CAbaPtServer
CICServer
CIRServer
CServer
CPowerServer
CRelPtServer
CSyskeyServer
CTouchServer

Fig. 8

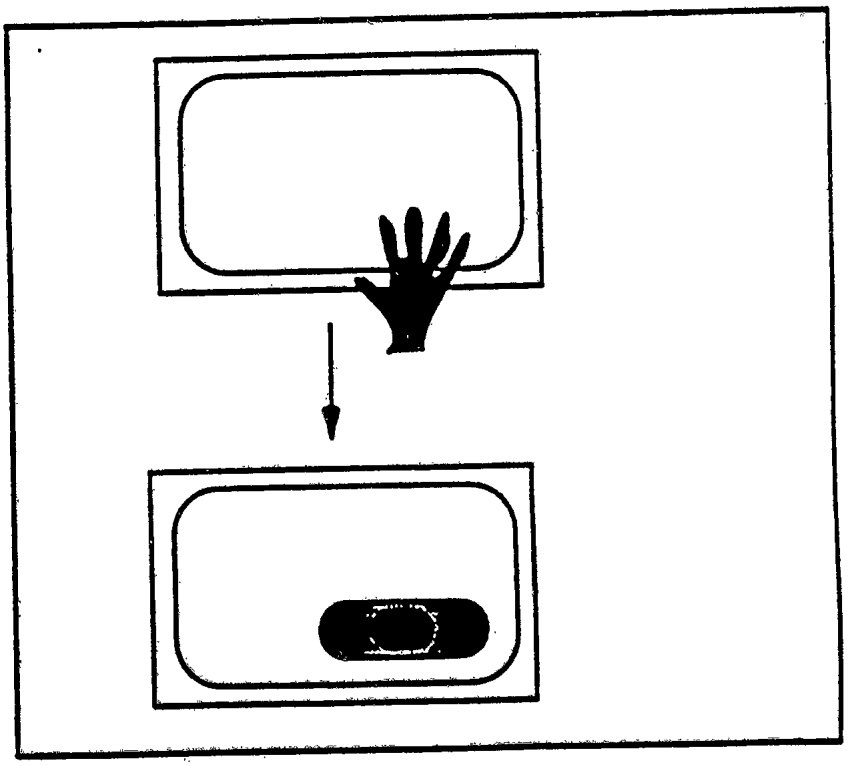
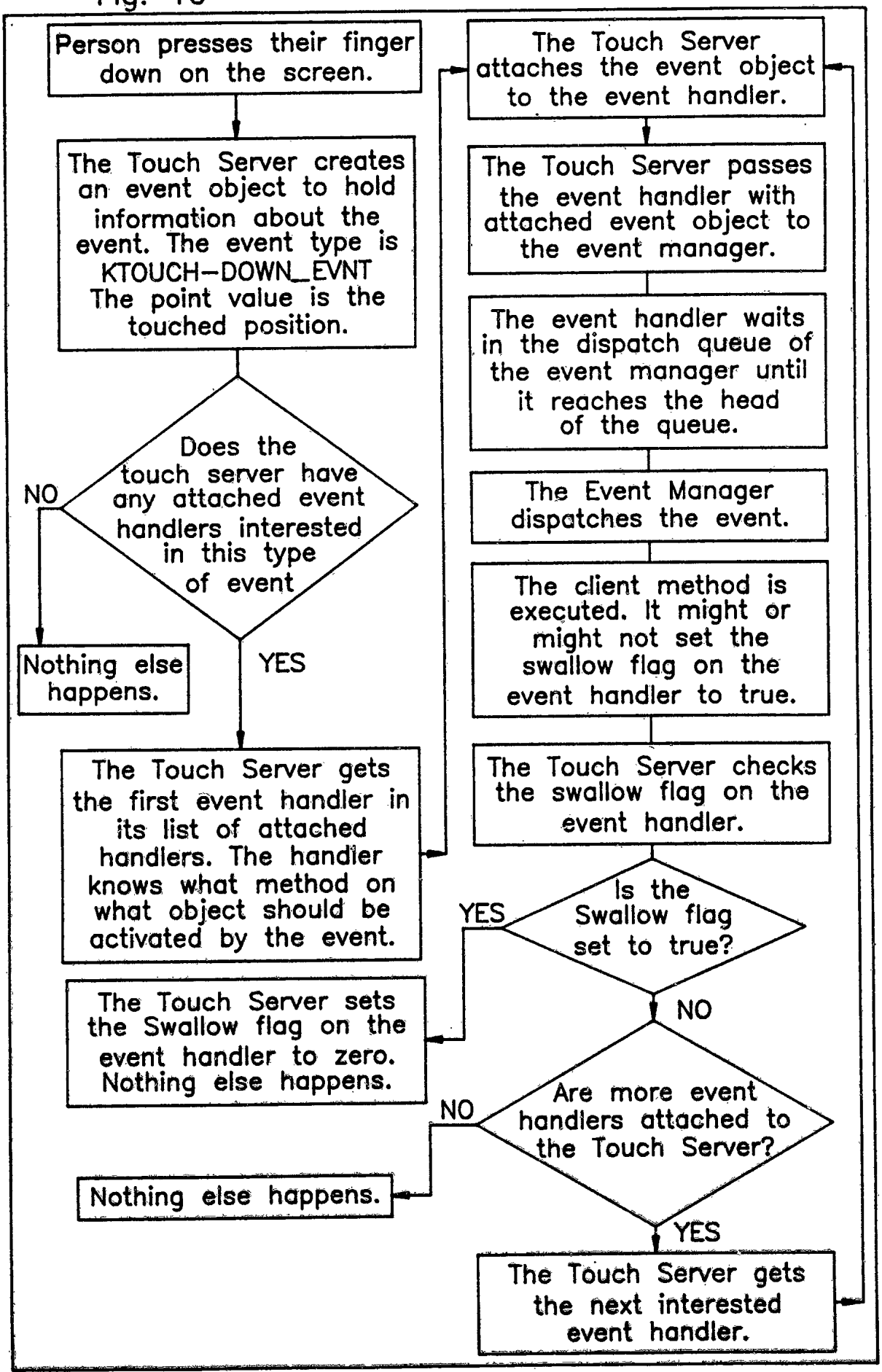


Fig. 9

Fig. 10


Person presses their finger down on the screen.
The Touch Server creates an event object to hold information about the event. The event type is KTOUCH-DOWN_EVNT The point value is the touched position.
Does the touch server have any attached event handlers interested in this type of event
NO
Nothing else happens.
YES
The Touch Server gets the first event handler in its list of attached handlers. The handler knows what method on what object should be activated by the event.
The Touch Server attaches the event object to the event handler.
The Touch Server passes the event handler with attached event object to the event manager.
The event handler waits in the dispatch queue of the event manager until it reaches the head of the queue.
The Event Manager dispatches the event.
The client method is executed. It might or might not set the swallow flag on the event handler to true.
The Touch Server checks the swallow flag on the event handler.
Is the Swallow flag set to true?
YES
NO
The Touch Server sets the Swallow flag on the event handler to zero. Nothing else happens.
Are more event handlers attached to the Touch Server?
NO
Nothing else happens.
YES
The Touch Server gets the next interested event handler.

INTERNATIONAL SEARCH REPORT

Interr. ial Application No
PCT/US 94/03590

A. CLASSIFICATION OF SUBJECT MATTER
IPC 5 G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 5 G06F

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

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

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	EP,A,0 528 222 (IBM) 24 February 1993 see page 2, line 15 - line 23 see page 2, line 54 - page 3, line 10 see page 3, line 11 - line 19 see page 4, line 7 - line 29	1-8
X	EP,A,0 297 339 (BMC SOFTWARE INC) 4 January 1989 see column 3, line 35 - column 4, line 54 see column 8, line 16 - line 30 see column 12, line 12 - line 47 see column 14, line 32 - line 40	1-8

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
27 July 1994	04. 08. 94

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax (+31-70) 340-3016	Brandt, J

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

INTERNATIONAL SEARCH REPORT

Information on patent family members

Intern. nal Application No
PCT/US 94/03590

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP-A-0528222	24-02-93	US-A- 5305454	19-04-94
		JP-A- 5197568	06-08-93
EP-A-0297339	04-01-89	NONE	