

# EXHIBIT A



US005502839A

# United States Patent [19]

[11] Patent Number: 5,502,839

Kolnick

[45] Date of Patent: Mar. 26, 1996

- [54] OBJECT-ORIENTED SOFTWARE ARCHITECTURE SUPPORTING INPUT/OUTPUT DEVICE INDEPENDENCE
- [75] Inventor: Frank C. Kolnick, Willowdale, Canada
- [73] Assignee: Motorola, Inc., Schaumburg, Ill.
- [21] Appl. No.: 361,738
- [22] Filed: Jun. 2, 1989

4,485,439	11/1984	Rothstein	395/500
4,547,628	10/1985	Tamura et al.	340/747
4,555,775	11/1985	Pike	364/900
4,559,614	12/1985	Peek et al.	364/900
4,642,790	2/1987	Minshull et al.	364/900
4,754,395	6/1988	Weissbaar et al.	364/200
4,800,523	1/1989	Gerety et al.	395/500
4,858,114	8/1989	Heath et al.	395/500
5,063,494	11/1991	Davidowski et al.	395/800

### Related U.S. Application Data

- [63] Continuation of Ser. No. 619, Jan. 5, 1987, abandoned.
- [51] Int. Cl.<sup>6</sup> G06F 13/00
- [52] U.S. Cl. 395/800; 364/228.2; 364/237.9; 364/239.9; 364/280; 364/284.2; 364/DIG. 1
- [58] Field of Search 364/200 MS File, 364/900 MS File; 395/500

Primary Examiner—Kevin J. Teska  
 Assistant Examiner—Ayni Mohamed  
 Attorney, Agent, or Firm—Walter W. Nielsen; Harold C. McGurk; S. Kevin Pickens

### [57] ABSTRACT

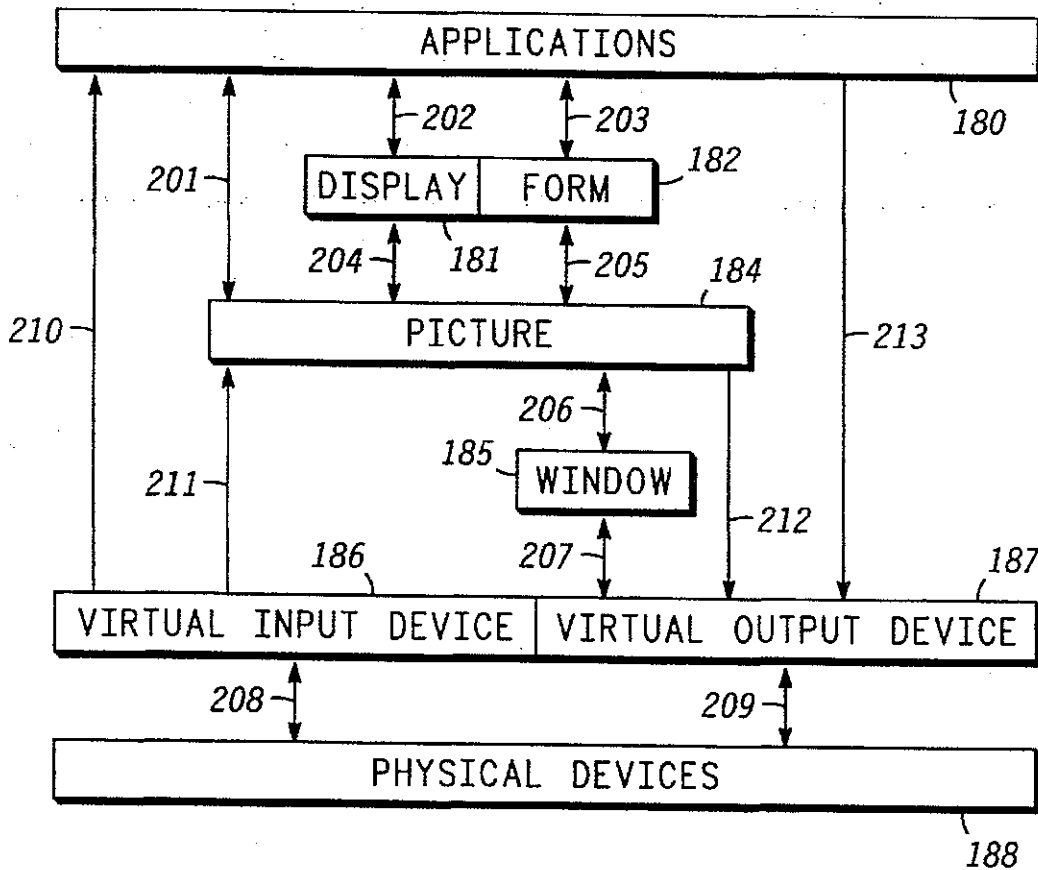
An object-oriented software architecture interacts with "real" input/output devices exclusively through "virtual" input/output devices. Since all human interface with the operating system is performed through such virtual devices, the system can accept any form of real input or output devices. The lowest level of the operating system converts input from any physical device to virtual form and converts virtual output into suitable physical output. Any number of physical devices can be connected to, removed from, or replaced in the system without disrupting the system.

### References Cited

#### U.S. PATENT DOCUMENTS

- 3,930,232 12/1975 Wallach et al. 395/500
- 4,241,341 12/1980 Thorson 340/747
- 4,454,593 6/1984 Fleming 364/900

23 Claims, 9 Drawing Sheets



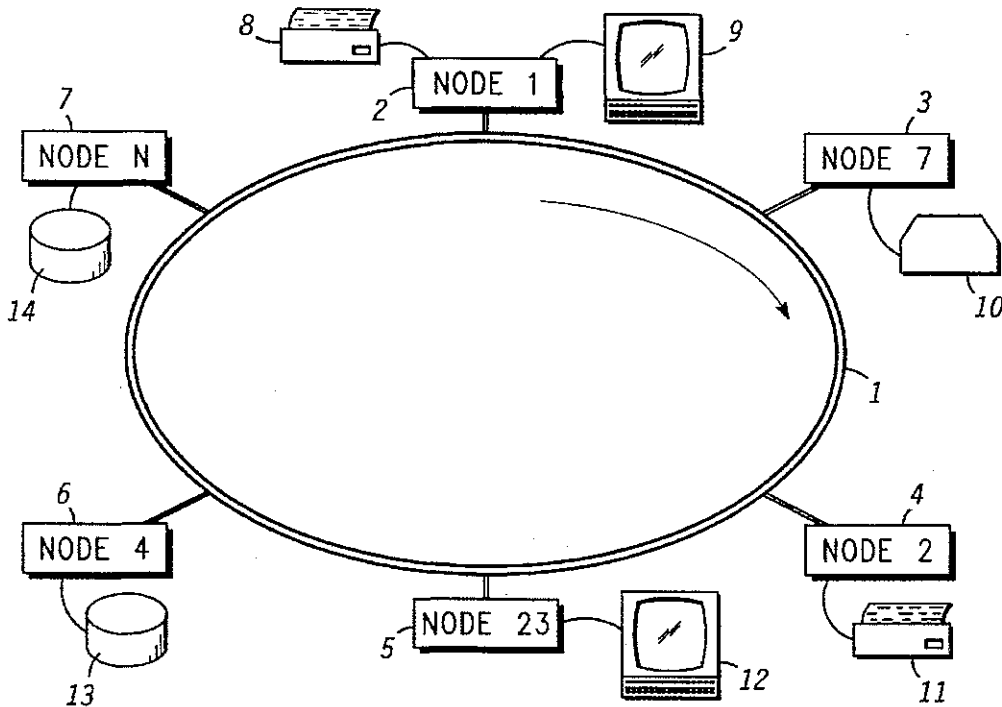


FIG. 1

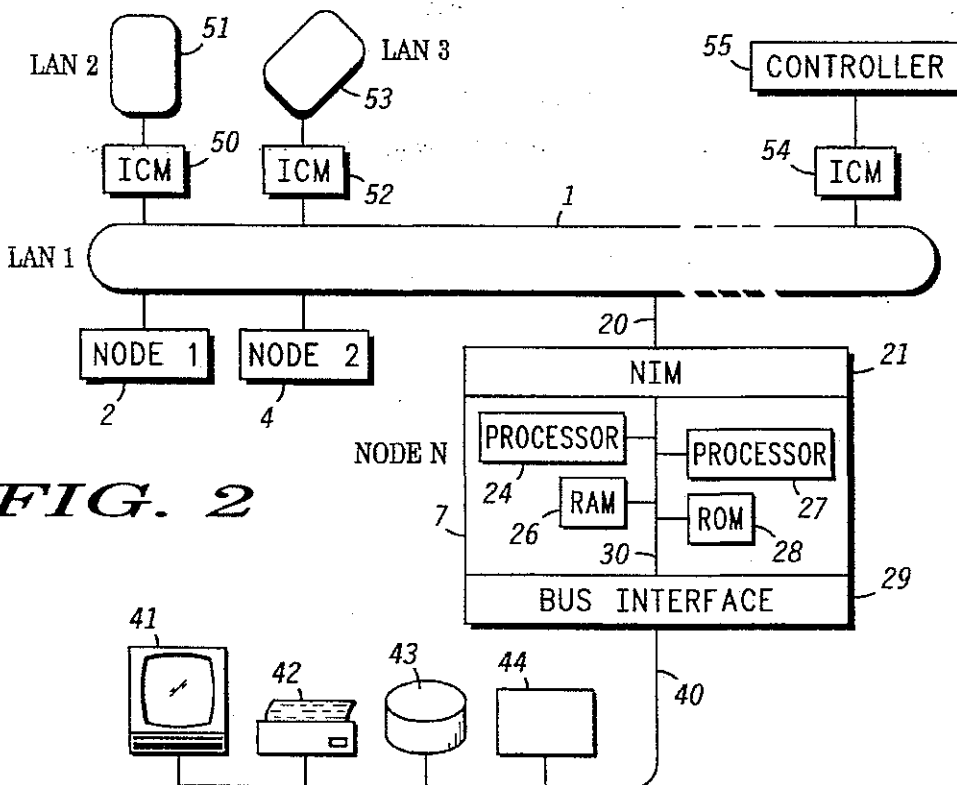


FIG. 2

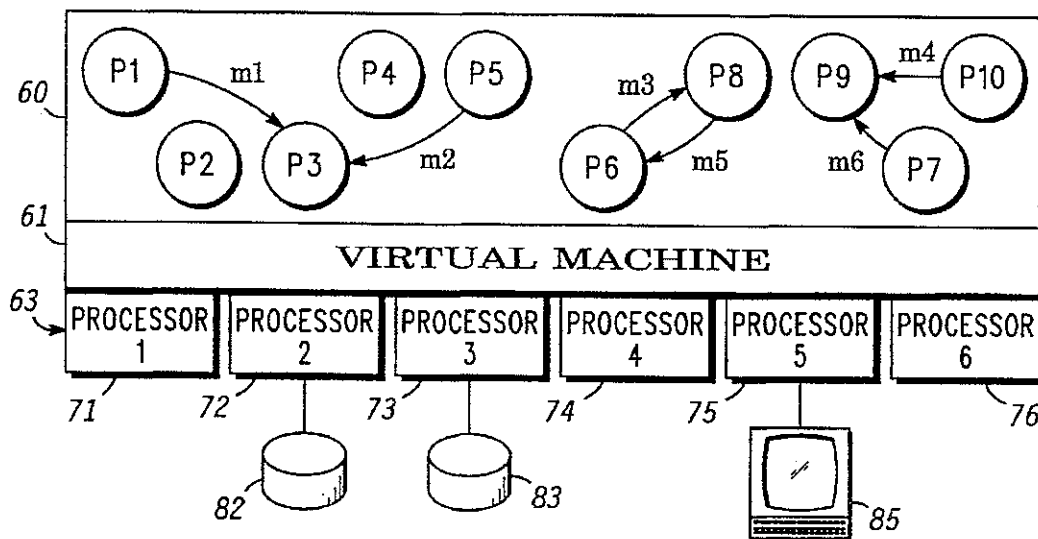
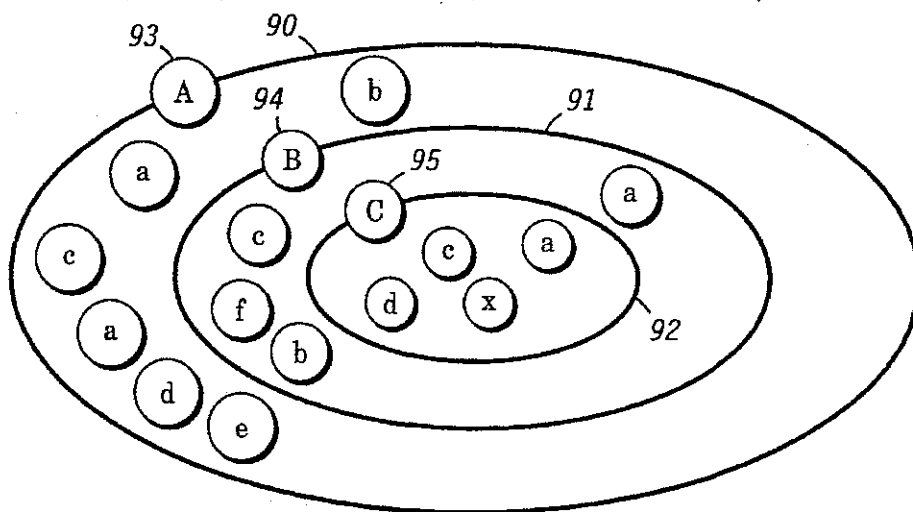
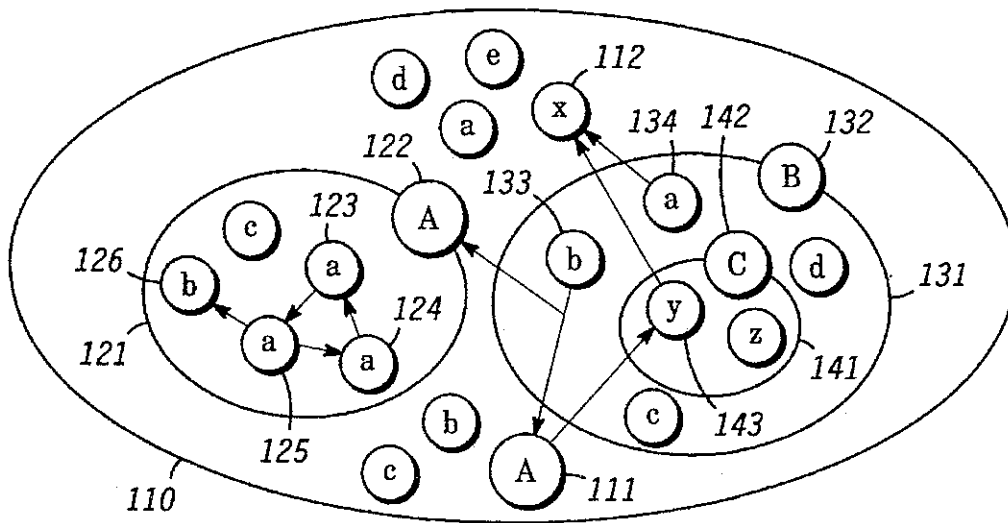


FIG. 3

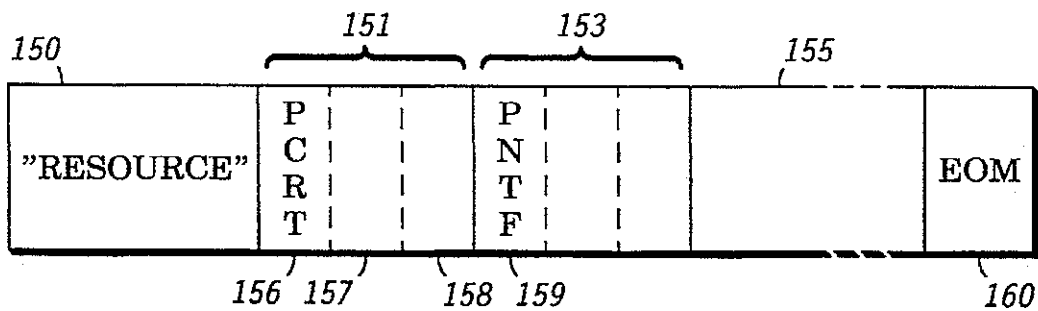
FIG. 4





**FIG. 5**

**FIG. 6**



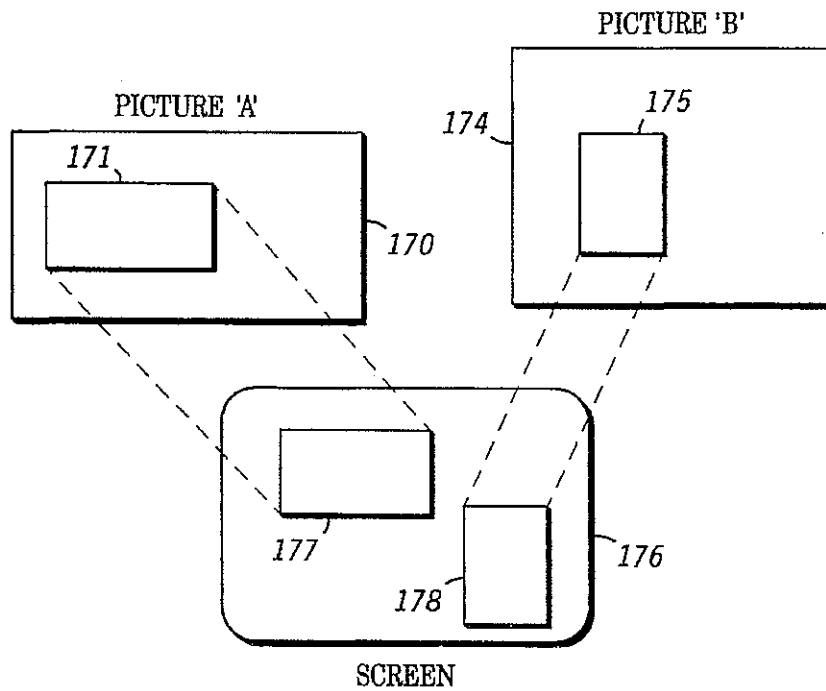
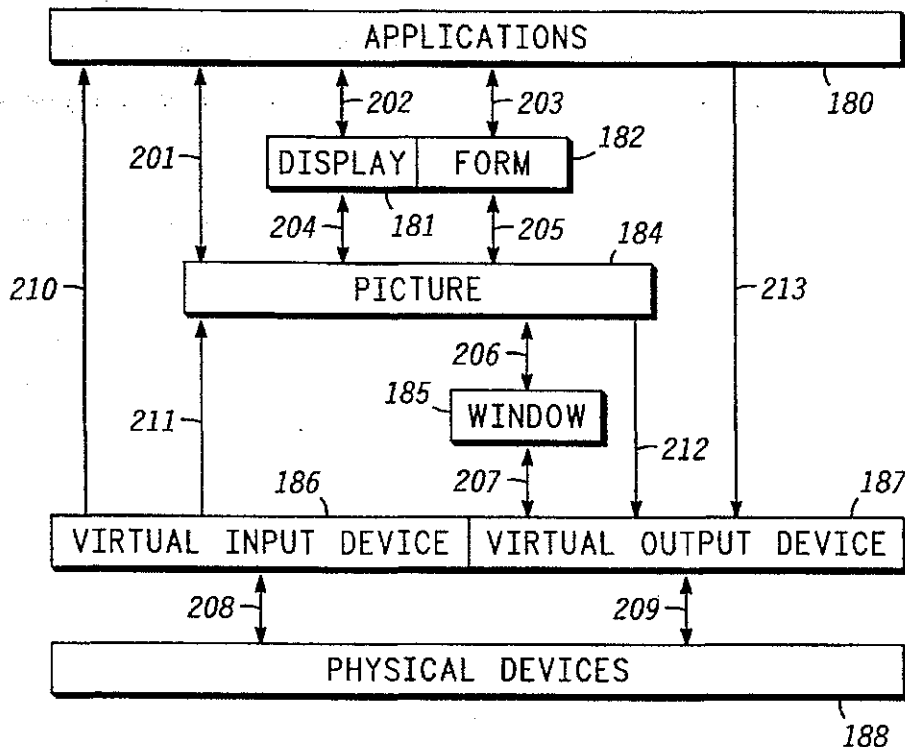


FIG. 7

FIG. 8



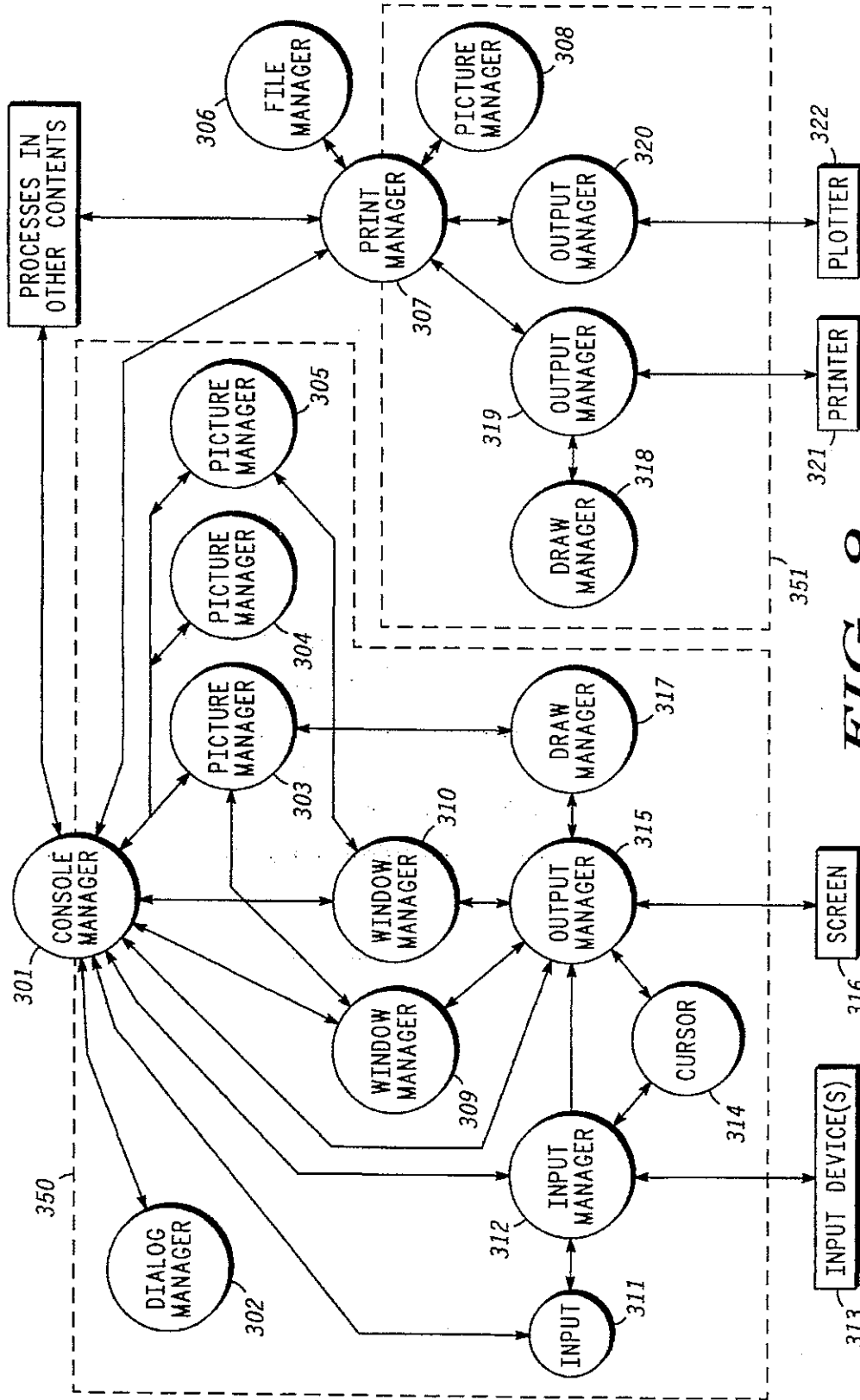


FIG. 9

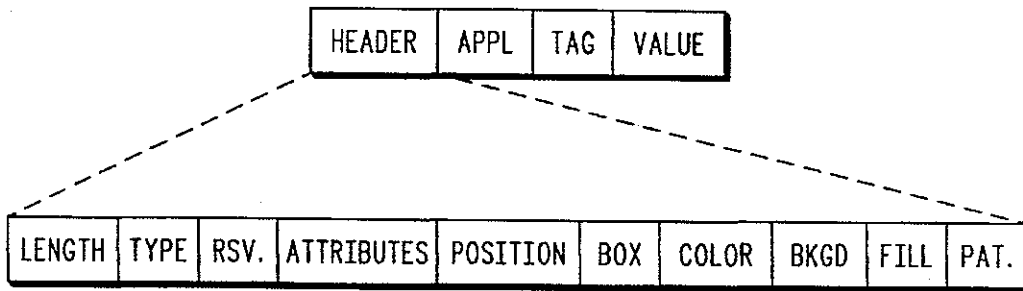
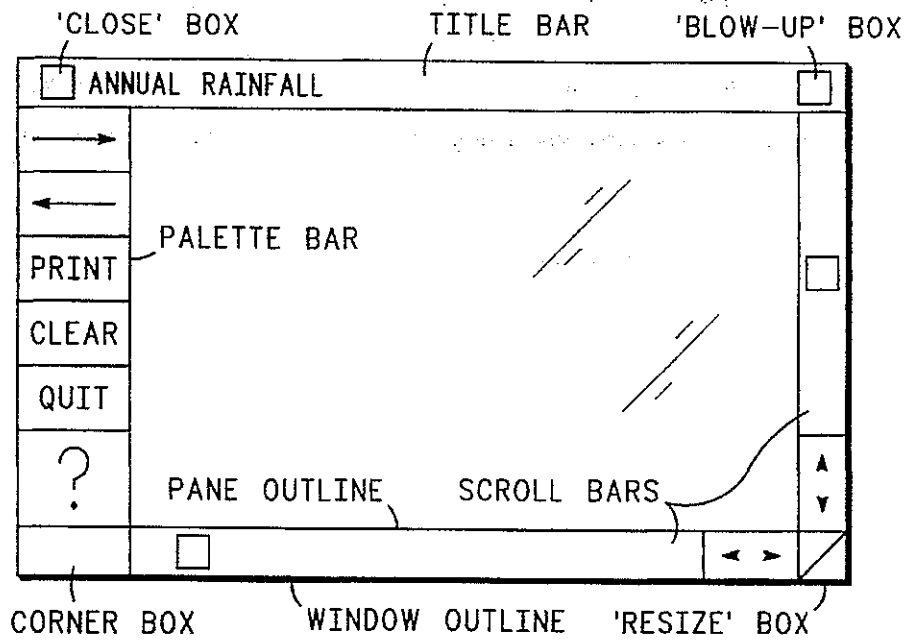
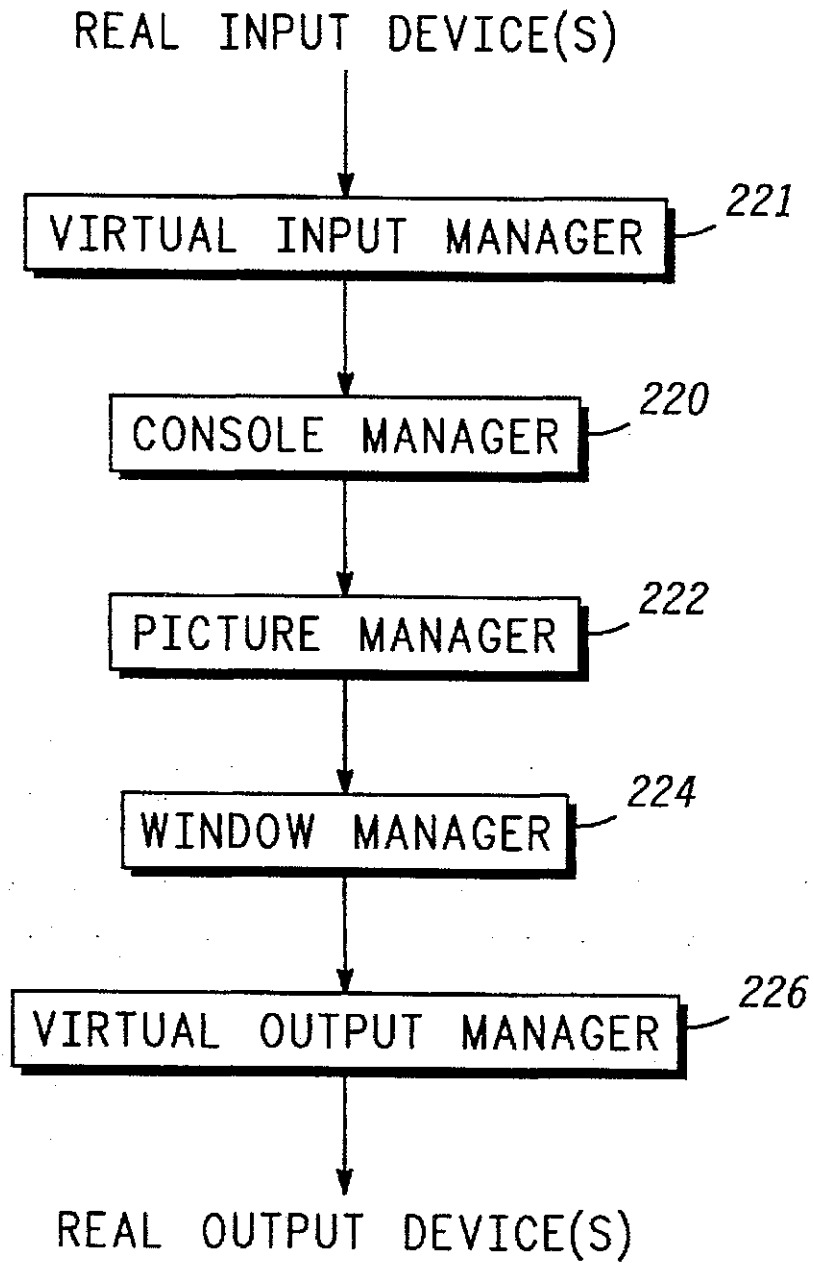


FIG. 10

FIG. 11







**FIG. 12**

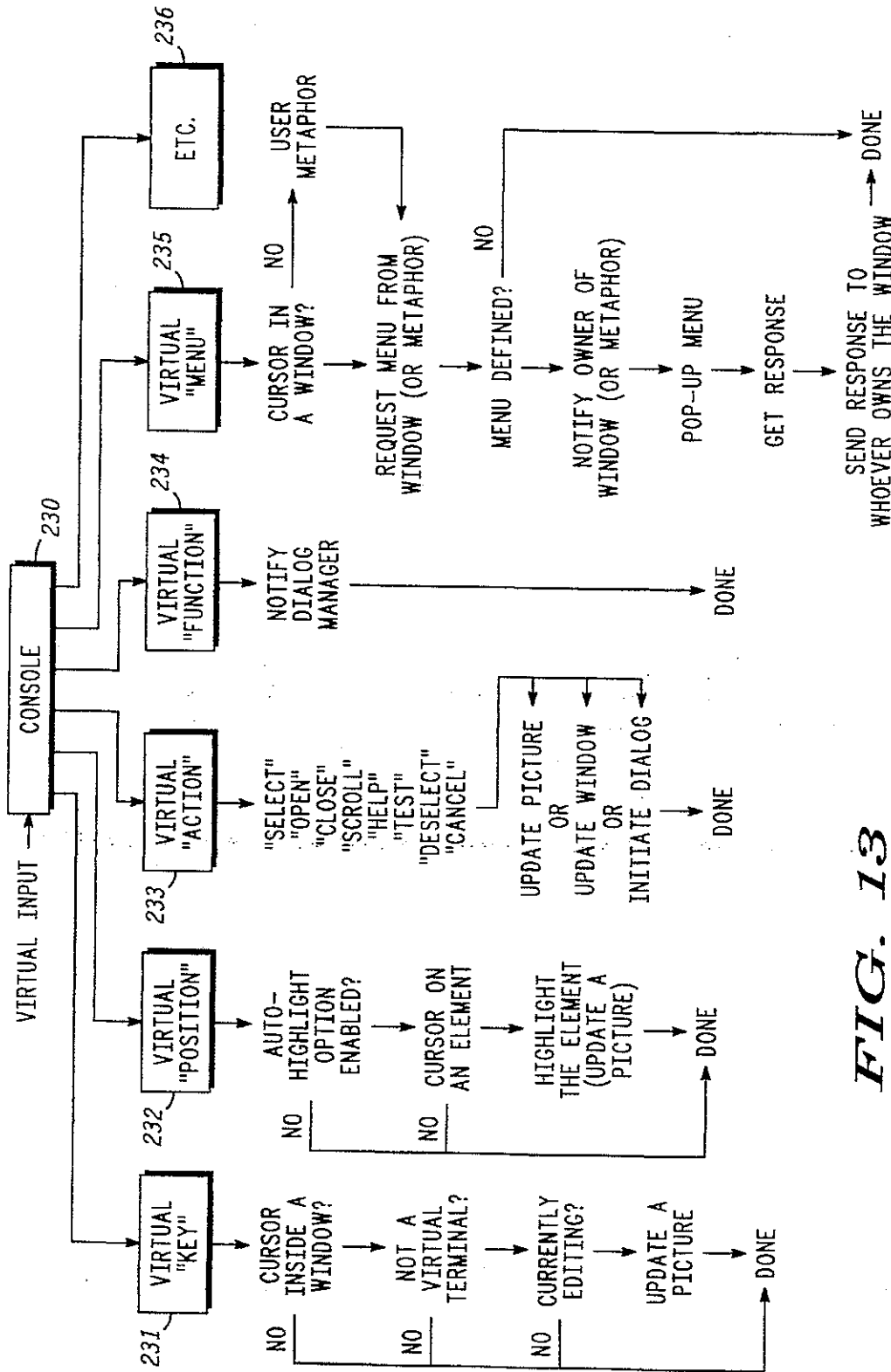


FIG. 13

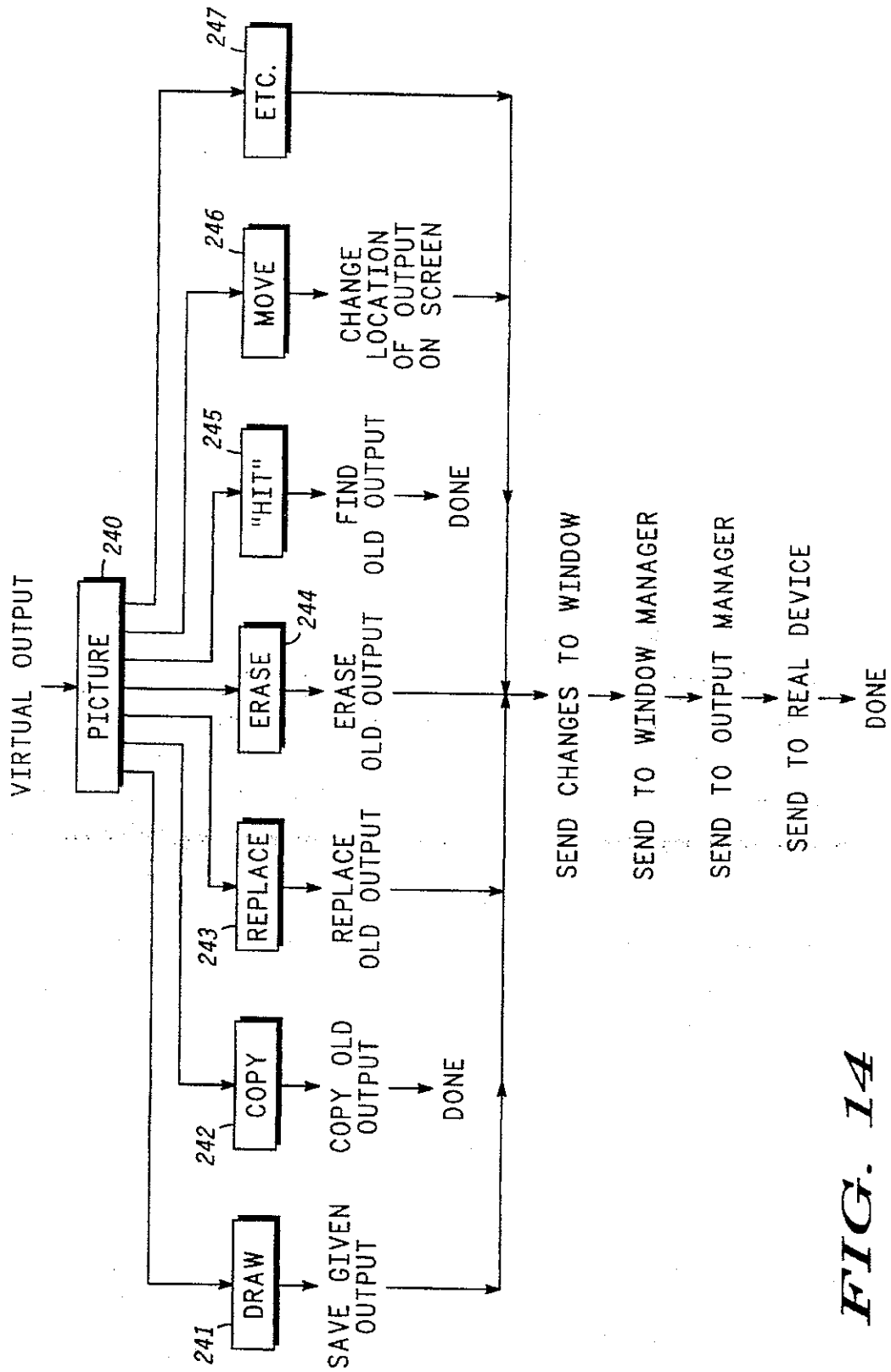


FIG. 14

**OBJECT-ORIENTED SOFTWARE  
ARCHITECTURE SUPPORTING  
INPUT/OUTPUT DEVICE INDEPENDENCE**

This application is a continuation of prior application 5  
Ser. No. 000,619, filed Jan. 5, 1987 now abandoned.

**RELATED INVENTIONS**

The present invention is related to the following inven- 10  
tions, all filed on May 6, 1985, and all assigned to the  
assignee of the present invention:

1. Title: Nested Contexts in a Virtual Single Machine  
Inventors: Andrew Kun, Frank Kolnick, Bruce Mansfield  
Ser. No.: 730,903, now abandoned

2. Title: Computer System With Data Residence Transpar- 15  
ancy and Data Access Transparency

Inventors: Andrew Kun, Frank Kolnick, Bruce Mansfield  
Ser. No.: 730,929 (now abandoned) and Ser. No. 07/110,614  
filed on Oct. 14, 1987 and now abandoned (continuation)

3. Title: Network Interface Module With Minimized Data 20  
Paths

Inventors: Bernhard Weisshaar, Michael Barnea  
Ser. No.: 730,621, now U.S. Pat. No. 4,754,395

4. Title: Method of Inter-Process Communication in a Dis- 25  
tributed Data Processing System

Inventors: Bernhard Weisshaar, Andrew Kun, Frank  
Kolnick, Bruce Mansfield

Ser. No.: 730,892, now U.S. Pat. No. 4,694,396

5. Title: Logical Ring in a Virtual Single Machine 30  
Inventor: Andrew Kun, Frank Kolnick, Bruce Mansfield  
Ser. No.: 730,923 (now abandoned) Ser. No. 183,469 filed  
on Apr. 13, 1988 and now U.S. Pat. No. 5,047,925  
(continuation)

6. Title: Virtual Single Machine With Message-Like Hard- 35  
ware Interrupts and Processor Exceptions

Inventors: Andrew Kun, Frank Kolnick, Bruce Mansfield  
Ser. No.: 730,922 now U.S. Pat. No. 4,835,685

The present invention is also related to the following 40  
inventions, all filed on even date herewith, and all assigned  
to the assignee of the present invention:

7. Title: Computer Human Interface Comprising User-Ad-  
justable Window for Displaying or Printing Information  
Inventor: Frank Kolnick

Ser. No.: 000,625 now abandoned

8. Title: Computer Human Interface With Multi-Application 45  
Display

Inventor: Frank Kolnick

Ser. No.: 000,620 now abandoned

9. Title: Self-Configuration of Nodes in a Distributed Mes- 50  
sage-Based Operating System

Inventor: Gabor Simor

Ser. No.: 000,621 now U.S. Pat. No. 5,165,018

10. Title: Process Traps in a Distributed Message-Based 55  
Operating System

Inventors: Gabor Simor

Ser. No.: 000,624 now abandoned

11. Title: Computer Human Interface With Multiple Inde-  
pendent Active Pictures and Windows

Inventor: Frank Kolnick

Ser. No.: 000,626, now abandoned 60

**TECHNICAL FIELD**

This invention relates generally to digital data processing, 65  
and, in particular, to a human interface system providing  
means for converting "real" input into virtual input, and  
means for converting virtual output into "real" output.

**BACKGROUND OF THE INVENTION**

It is known in the data processing arts to couple a wide  
assortment of input and output devices to a data processing  
system for the purpose of providing an appropriate human  
interface. Such devices may take the form of keyboards of  
varying manufacture, "mice", touch-pads, joy-sticks, light  
pens, video screens, audio-visual signals, printers, etc.

Due to the wide variety of I/O devices which can be  
utilized in the human/computer interface, it would be very  
desirable to isolate the human interface software from  
specific device types. The I/O should be independent of any  
particular "real" devices.

There is thus a need for a computer human interface  
which performs I/O operations in an abstract sense, inde-  
pendent of particular "real" devices.

**BRIEF SUMMARY OF INVENTION**

Accordingly, it is an object of the present invention to  
provide a data processing system having an improved  
human interface.

It is also an object of the present invention to provide an  
improved human interface system which performs input/  
output operations in an abstract sense, independent of any  
particular I/O devices.

It is another object of the present invention to provide an  
improved human interface system in which any type of  
"real" input and output devices may be employed, and in  
which I/O devices may be connected to and disconnected  
from the data processing system without disrupting process-  
ing operations.

These and other objects are achieved in accordance with  
a preferred embodiment of the invention by providing a  
virtual input interface in a data processing system, such  
interface comprising means for accepting input from at least  
one physical device, means for converting the physical  
device input into virtual input, and means responsive to the  
virtual input for performing processing operations upon the  
virtual input.

**BRIEF DESCRIPTION OF THE DRAWINGS**

The invention is pointed out with particularity in the  
appended claims. However, other features of the invention  
will become more apparent and the invention will be best  
understood by referring to the following detailed description  
in conjunction with the accompanying drawings in which:

FIG. 1 shows a representational illustration of a single  
network, distributed message-based data processing system  
of the type incorporating the present invention.

FIG. 2 shows a block diagram illustrating a multiple-  
network, distributed message-based data processing system  
of the type incorporating the present invention.

FIG. 3 shows an architectural model of a data processing  
system of the type incorporating the present invention.

FIG. 4 shows the relationship between software contexts  
and processes as they relate to the present invention.

FIG. 5 shows how messages may be sent between pro-  
cesses within nested contexts.

FIG. 6 shows a standard message format used in the  
distributed data processing system of the present invention.

FIG. 7 shows the relationship between pictures, views,  
and windows in the human interface of a data processing  
system of the type incorporating the present invention.

3

FIG. 8 shows a conceptual view of the different levels of human interface within a data processing system incorporating the present invention.

FIG. 9 illustrates the relationship between the basic human interface components in a typical working environment.

FIG. 10 shows the general structure of a complete picture element.

FIG. 11 shows the components of a typical screen as contained within the human interface system of the present invention.

FIG. 12 shows how the console manager operates upon virtual input to generate virtual output.

FIG. 13 shows how virtual input is handled by the console manager.

FIG. 14 shows how virtual input is handled by the picture manager.

### OVERVIEW OF COMPUTER SYSTEM

The present invention can be implemented either in a single CPU data processing system or in a distributed data processing system that is, two or more data processing systems (each having at least one processor) which are capable of functioning independently but which are so coupled as to send and receive messages to and from one another.

A Local Area Network (LAN) is an example of a distributed data processing system. A typical LAN comprises a number of autonomous data processing "nodes", each comprising at least a processor and memory. Each node is capable of conducting data processing operations independently. In addition, each node is coupled (by appropriate means such as a twisted wire pair, coaxial cable, fiber optic cable, etc.) to a network of other nodes which may be, for example, a loop, star, tree, etc., depending upon the design considerations.

With reference to FIG. 1, a distributed computer configuration is shown comprising multiple nodes 2-7 (nodes) loosely coupled by a local area network (LAN) 1. The number of nodes which may be connected to the network is arbitrary and depends upon the user application. Each node comprises at least a processor and memory, as will be discussed in greater detail with reference to FIG. 2 below. In addition, each node may also include other units, such as a printer 8, operator display module (ODM) 9, mass memory module 13, and other I/O device 10.

With reference now to FIG. 2, a multiple-network distributed computer configuration is shown. A first local area network LAN 1 comprises several nodes 2,4, and 7. LAN 1 is coupled to a second local area network LAN 2 by means of an Intelligent Communications Module (ICM) 50. The Intelligent Communications Module provides a link between the LAN and other networks and/or remote processors (such as programmable controllers).

LAN 2 may comprise several nodes (not shown) and may operate under the same IAN protocol as that of the present invention, or it may operate under any of several commercially available protocols, such as Ethernet; MAP, the Manufacturing Automation Protocol of General Motors Corp.; Systems Network Architecture (SNA) of International Business Machines, Inc.; SECS-II; etc. Each ICM 50 is programmable for carrying out one of the above-mentioned specific protocols. In addition, the basic processing module of the node itself can be used as an intelligent peripheral controller (IPC) for specialized devices.

4

LAN 1 is additionally coupled to a third local area network LAN 3 via ICM 52. A process controller 55 is also coupled to LAN 1 via ICM 54.

A representative node N (7, FIG. 2) comprises a processor 24 which, in a preferred embodiment, is a processor from the Motorola 68000 family of processors. Each node further includes a read only memory (ROM) 28 and a random access memory (RAM) 26. In addition, each node includes a Network Interface Module (NIM) 21, which connects the node to the LAN, and a Bus Interface 29, which couples the node to additional devices within a node. While a minimal node is capable of supporting two peripheral devices, such as an Operator Display Module (ODM) 41 and an I/O Module 44, additional devices (including additional processors, such as processor 27) can be provided within a node. Other additional devices may comprise, for example, a printer 42, and a mass-storage module 43 which supports a hard disk and a back-up device (floppy disk or streaming tape drive).

The Operator Display Module 41 provides a keyboard and screen to enable an operator to input information and receive visual information.

While a single node may comprise all of the above units, in the typical user application individual nodes will normally be dedicated to specialized functions. For example, one or more mass storage nodes may be set up to function as data base servers. There may also be several operator consoles and at least one node for generating hard-copy printed output. Either these same nodes, or separate dedicated nodes, may execute particular application programs.

The system is particularly designed to provide an integrated solution for office or factory automation, data acquisition, and other real-time applications. As such, it includes a full complement of services, such as a graphical output, windows, menus, icons, dynamic displays, electronic mail, event recording, and file management. Software development features include compilers, a window-oriented editor, a debugger, and performance-monitoring tools.

### LOCAL AREA NETWORK

The local area network, as depicted in either FIG. 1 or FIG. 2, ties the entire system together and makes possible the distributed virtual machine model described below. The LAN provides high throughput, guaranteed response, reliability, and low entry cost. The LAN is also autonomous, in the sense that all system and applications software is unaware of its existence. For example, any Network Interface Module (e.g. NIM 21, FIG. 2) could be replaced without rewriting any software other than that which directly drives it.

The LAN interconnection medium may be twisted-pair or coaxial cable. Two channels (logically, two distinct networks) may be provided for reliability and for increased throughput.

The LAN architecture is a logical ring, in which an electronic "token" is constantly passed from node to node at high speed. The current holder of the token may use it to send a "frame" of data or may pass it on to the next node in the ring. The NIM only needs to know the logical address and status of its immediately succeeding neighbor. The NIM's responsibility is limited to detecting the failure of that neighbor or the inclusion of a new neighbor. In general, adjustment to failed or newly added nodes is automatic.

The network interface maps directly into the processor's memory. Data exchange occurs through a dual-ported buffer

5

pool which contains a linked list of pending "frames" Logical messages, which vary in length, are broken into fixed-size frames for transmission and are reassembled by the receiving NIM. Frames are sequence-numbered for this purpose. If a frame is not acknowledged within a short period of time, it is retransmitted a number of times before being treated as a failure.

As described above with reference to FIG. 2, the LAN may be connected to other LAN's operating under the same LAN protocol via so-called "bridgeways", or it may be connected to other types of LAN's via "gateways".

#### SOFTWARE MODEL

The computer operating system of the present invention operates upon processes, messages, and contexts, as such terms are defined herein. Thus this operating system offers the programmer a hardware abstraction, rather than a data or control abstraction.

A "process", as used within the present invention, is defined as a self-contained package of data and executable procedures which operate on that data, comparable to a "task" in other known systems. Within the present invention a process can be thought of as comparable to a subroutine in terms of size, complexity, and the way it is used. The difference between processes and subroutines is that processes can be created and destroyed dynamically and can execute concurrently with their creator and other "subroutines".

Within a process, as used in the present invention, the data is totally private and cannot be accessed from the outside, i.e., by other processes. Processes can therefore be used to implement "objects", "modules", or other higher-level data abstractions. Each process executes sequentially. Concurrency is achieved through multiple processes, possibly executing on multiple processors.

Every process in the distributed data processing system of the present invention has a unique identifier (PID) by which it can be referenced. The PID is assigned by the system when the process is created, and it is used by the system to physically locate the process.

Every process also has a non-unique, symbolic "name", which is a variable-length string of characters. In general, the name of a process is known system-wide. To restrict the scope of names, the present invention utilizes the concept of a "context".

A "context" is simply a collection of related processes whose names are not known outside of the context. Contexts partition the name space into smaller, more manageable subsystems. They also "hide" names, ensuring that processes contained in them do not unintentionally conflict with those in other contexts.

A process in one context cannot explicitly communicate with, and does not know about, processes inside other contexts. All interaction across context boundaries must be through a "context process", thus providing a degree of security. The context process often acts as a switchboard for incoming messages, rerouting them to the appropriate sub-processes in its context.

A context process behaves like any other process and additionally has the property that any processes which it creates are known only to itself and to each other. Creation of the process constitutes definition of a new context with the same name as the process.

Any process can create context processes. Each new context thus defined is completely contained inside the

6

context in which it was created and therefore is shielded from outside reference. This "nesting" allows the name space to be structured hierarchically to any desired depth.

Conceptually, the highest level in the hierarchy is the system itself, which encompasses all contexts. Nesting is used in top-down design to break a system into components or "layers", where each layer is more detailed than the preceding one. This is analogous to breaking a task down into subroutines, and in fact many applications which are single tasks on known systems may translate to multiple processes in nested contexts.

A "message" is a buffer containing data which tells a process what to do and/or supplies it with information it needs to carry out its operation. Each message buffer can have a different length (up to 64 kilobytes). By convention, the first field in the message buffer defines the type of message (e.g., "read", "print", "status", "event", etc.).

Messages are queued from one process to another by name or PID. Queuing avoids potential synchronization problems and is used instead of semaphores, monitors, etc. The sender of a message is free to continue after the message is sent. When the receiver attempts to get a message, it will be suspended until one arrives if none are already waiting in its queue. Optionally, the sender can specify that it wants to wait for a reply and is suspended until that specific message arrives. Messages from any other source are not dequeued until after that happens.

Within the present invention, messages are the only way for two processes to exchange data. There is no concept of a "global variable". Shared memory areas are not allowed, other than through processes which essentially "manage" each area by means of messages. Messages are also the only form of dynamic memory that the system handles. A request to allocate memory therefore returns a block of memory which can be used locally by the process but can also be transmitted to another process.

The context nesting level determines the "scope of reference" when sending messages between processes by name. From a given process, a message may be sent to all processes at its own level (i.e., in the same context) and (optionally) to any arbitrary higher level. The contexts are searched from the current context upward until a match is found. All processes with the given name at that level are then sent a copy of the message. A process may also send a message to itself or to its parent (the context process) without knowing either name explicitly, permitting multiple instances of a process to exist in different contexts, with different names.

Sending messages by PID obviates the need for a name search and ignores context boundaries. This is the most efficient method of communicating.

Processes are referenced without regard to their physical location via a small set of message-passing primitives. As mentioned earlier, every process has both a unique system-generated identifier and a not necessarily unique name assigned by the programmer. The identifier provides quick direct access, while the name has a limited scope and provides symbolic, indirect access.

With reference to FIG. 3, an architectural model of the present invention is shown. The bottom, or hardware, layer 63 comprises a number of processors 71-76, as described above. The processors 71-76 may exist physically within one or more nodes. The top, or software, layer 60 illustrates a number of processes P1-P10 which send messages m1-m6 to each other. The middle layer 61, labelled "virtual machine", isolates the hardware from the software, and it allows programs to be written as if they were going to be

executed on a single processor. Conversely, programs can be distributed across multiple processors without having been explicitly designed for that purpose.

### THE VIRTUAL MACHINE

As discussed earlier, a "process" is a self-contained package of data and executable procedures which operate on that data. The data is totally private and cannot be accessed by other processes. There is no concept of shared memory within the present invention. Execution of a process is strictly sequential. Multiple processes execute concurrently and must be scheduled by the operating system. The processes can be re-entrant, in which case only one copy of the code is loaded even if multiple instances are active.

Every process has a unique "process identifier number" (PID) by which it can be referenced. The PID is assigned by the system when the process is created and remains in effect until the process terminates. The PID assignment contains a randomizing factor which guarantees that the PID will not be re-used in the near future. The contents of the PID are irrelevant to the programmer but are used by the virtual machine to physically locate the process. A PID may be thought of as a "pointer" to a process.

Every process also has a "name" which is a variable-length string of characters assigned by the programmer. A name need not be unique, and this ambiguity may be used to add new services transparently and to aid in fault-tolerance.

FIG. 4 illustrates that the system-wide name space is partitioned into distinct subsets by means of "contexts" identified by reference numerals 90-92. A context is simply a collection of related processes whose names are not known outside of the context. Context 90, for example, contains processes A, a, a, b, c, d, and e. Context 91 contains processes B, a, b, c, and f. And context 92 contains processes C, a, c, d, and x.

One particular process in each context, called the "context process", is known both within the context and within the immediately enclosing one (referred to as its "parent context"). In the example illustrated in FIG. 4, processes A-C are context processes for contexts 90-92, respectively. The parent context of context 91 is context 90, and the parent context of context 92 is context 91. Conceptually, the context process is located on the boundary of the context and acts as a gate into it.

Processes inside context 92 can reference any processes inside contexts 90 and 91 by name. However, processes in context 91 can only access processes in context 92 by going through the context process C. Processes in context 90 can only access processes in context 92 by going through context processes B and C.

The function of the context process is to filter incoming messages and either reject them or reroute them to other processes in its context. Contexts may be nested, allowing a hierarchy of abstractions to be constructed. A context must reside completely on one node. The entire system is treated as an all-encompassing context which is always present and which is the highest level in the hierarchy. In essence, contexts define localized protection domains and greatly reduce the chances of unintentional naming conflicts.

If appropriate, a process inside one context can be "connected" to one inside another context by exchanging PID's, once contact has been established through one or the other of the context processes. Most process servers within the present invention function that way. Initial access is by

name. Once the desired function (such as a window or file) is "opened", the user process and the service communicate directly via PID's.

A "message" is a variable-length buffer (limited only by the processor's physical memory size) which carries information between processes. A header, inaccessible to the programmer, contains the destination name and the sender's PID. By convention, the first field in a message is a null-terminated string which defines the type of message (e.g., "read", "status", etc.) Messages are queued to the receiving process when they are sent. Queuing ensures serial access and is used in preference to semaphores, monitors, etc.

Messages provide the mechanism by which hardware transparency is achieved. A process located anywhere in the system may send a message to any other process anywhere else in the system (even on another processor) if it knows the process name. This means that processes can be dynamically distributed across the system at any time to gain optimal throughput without changing the processes which reference them. Resolution of destinations is done by searching the process name space.

Transparency applies with some restrictions across bridgeways (i.e., the interfaces between LAN's operating under identical network protocols) and, in general, not at all across gateways (i.e., the interfaces between LAN's operating under different network protocols) due to performance degradation. However, they could so operate, depending upon the required level of performance.

### INTER-PROCESS COMMUNICATION

All inter-process communication is via messages. Consequently, most of the virtual machine primitives are concerned with processing messages. The virtual machine kernel primitives are the following:

ALLOC—requests allocation of a (message) buffer of a given size.

FREE—requests deallocation of a given message buffer.

PUT—end a message to a given destination (by name or PID).

GET—wait for and dequeue the next incoming message, optionally from a specific process (by PID).

FORWARD—pass a received message through to another process.

CALL—send a message, then wait for and dequeue the reply.

REPLY—send a message to the originator of a given message.

ANY\_MSG—returns "true" if the receive queue is not empty, else returns "false"; optionally, checks if any messages from a specific PID are queued.

To further describe the function of the kernel primitives, ALLOC handles all memory allocations. It returns a pointer to a buffer which can be used for local storage within the process or which can be sent to another process (via PUT, etc.). ALLOC never "fails", but rather waits until enough memory is freed to satisfy the request.

The PUT primitive queues a message to another process. The sending process resumes execution as soon as the message is queued.

FORWARD is used to quickly reroute a message but maintain information about the original sender (whereas PUT always makes the sending process the originator of the message).

REPLY sends a message to the originator of a previously received message, rather than by name or PID.

CALL essentially implements remote subroutine invocations, causing the caller to suspend until the receiver executes a REPLY. Subsequently, the replied message is dequeued out of sequence, immediately upon arrival, and the caller resumes execution.

The emphasis is on concurrency, so that as many processes as possible are executed in parallel. Hence neither PUT nor FORWARD waits for the message to be delivered. Conversely, GET suspends a process until a message arrives and dequeues it in one operation. The ANY MSG primitive is provided so that a process may determine whether there is anything of interest in the queue before committing itself to a GET.

When a message is sent by name, the destination process must be found in the name space. The search path is determined by the nesting of the contexts in which the sending process resides. From a given process, a message can be sent to all processes in its own context or (optionally) to those in any higher context. Refer to FIG. 5. The contexts are searched from the current one upward until a match is found or until the system context is reached. All processes with the same name in that context are then queued a copy of the message.

For example, with reference to FIG. 5, assume that in context 141 process y sends a message to ALL processes by the name x. Process y first searches within its own context 141 but finds no process x. The process y searches within the next higher context 131 (its parent context) but again finds no process x. Then process y searches within the next higher context 110 and finds a process x, identified by reference numeral 112. Since it is the only process x in context 110, it is the only recipient of the message from process y.

If process a in context 131 sends a message to ALL processes by the name x, it first searches within its own context 131 and, finding no processes x there, it then searches within context 110 and finds process x.

Assume that process b in context 131 sends a message to ALL processes by the name A. It would find process A (111) in context 110, as well as process A (122) which is the context process for context 121.

A process may also send a message to itself or to its context process without knowing either name explicitly.

The concept of a "logical ring" (analogous to a LAN) allows a message to be sent to the NEXT process in the system with a given name. The message goes to exactly one process in the sender's context, if such a process exists. Otherwise the parent context is searched.

The virtual machine guarantees that each NEXT transmission will reach a different process and that eventually a transmission will be sent to the logically "first" process (the one that sent the original message) in the ring, completing the loop. In other words, all processes with the same name at the same level can communicate with each other without knowing how many there are or where they are located. The logical ring is essential for distributing services such as a data base. The ordering of processes in the ring is not predictable.

For example, if process a (125) in context 121 sends a message to process a using the NEXT primitive, the search finds a first process a (124) in the same context 121. Process a (124) is marked as having received the message, and then process a (124) sends the message on to the NEXT process a (123) in context 121. Process a (123) is marked as having received the message, and then it sends the message on to the NEXT process a, which is the original sender process a

(125), which knows not to send it further on, since it's been marked as having already received the message.

Sending messages directly by PID obviates the need for a name search and ignores context boundaries. This is known as the DIRECT mode of transmission and is the most efficient. For example, process A (111) sends a message in the DIRECT mode to process y in context 141.

If a process sends a message in the LOCAL transmission mode, it sends it only to a process having the given name in the sender's own context.

In summary, including the DIRECT transmission mode, there are five transmission modes which can be used with the PUT, FORWARD, and CALL primitives:

ALL—to all processes with the given name in the first context which contains that name, starting with the sender's context and searching upwards through all parent contexts.

LOCAL—to all processes with the given name in the sender's context only.

NEXT—to the next process with the given name in the same context as the sender, if any; otherwise it searches upwards through all parent contexts until the name is found.

LEVEL—sends to "self" (the sending process) or to "context" (the context process corresponding to the sender's context); "self" cannot be used with CALL primitive.

DIRECT—sent by PID.

Messages are usually transmitted by queuing a pointer to the buffer containing the message. A message is only copied when there are multiple destinations or when the destination is on another node.

## OPERATING SYSTEM

The operating system of the present invention consists of a kernel, which implements the primitives described above, plus a set of processes which provide process creation and termination, time management (set time, set alarm, etc.) and which perform node start-up and configuration. Drivers for devices are also implemented as processes (EESP's), as described above. This allows both system services and device drivers to be added or replaced easily. The operating system also supports swapping and paging, although both are invisible to applications software.

Unlike known distributed computer systems, that of the present invention does not use a distinct "name server" process to resolve names. Name searching is confined to the kernel, which has the advantage of being much faster.

A minimal bootstrap program resides permanently (in ROM) on every node, e.g. ROM 28 in node N of FIG. 2. The bootstrap program executes automatically when a node is powered up and begins by performing basic on-board diagnostics. It then attempts to find and start an initial system code module. The module is sought on the first disk drive on the node, if any. If there isn't a disk, and the node is on the LAN, a message will be sent out requesting the module. Failing that, the required software must be resident in ROM. The initialization program of the kernel sets up all of the kernel's internal tables and then calls a predefined entry point of the process.

In general, there exists a template file describing the initial software and hardware for each node in the system. The template defines a set of initial processes (usually one per service) which are scheduled immediately after the node



start-up. These processes then start up their respective subsystems. A node configuration service on each node sends configuration messages to each subsystem when it is being initialized, informing it of the devices it owns. Thereafter, similar messages are sent whenever a new device is added to the node or a device fails or is removed from the node.

Thus there is no well-defined meaning for "system up" or "system down"—as long as any node is active, the system as a whole may be considered to be "up". Nodes can be shut down or started up dynamically without affecting other nodes on the network. The same principle applies, in a limited sense, to peripherals. Devices which can identify themselves with regard to type, model number, etc. can be added or removed without operator intervention.

FIG. 6 shows the standard format of a message in a distributed data processing system of the type incorporating the present invention. The message format comprises a message i.d. portion 150; one or more "triples" 151, 153, and 155; and an end-of-message portion 160. Each "triple" comprises a group of three fields, such as fields 156-158.

The first field 156 of "triple" 151, designated the PCRT field, represents the name of the process to be created. The second field 157 of "triple" 151 gives the size of the data field. The third field 158 is the data field.

The first field 159 of "triple" 153, designated the PNTF field, represents the name of the process to notify when the process specified in the PCRT field has been created.

A message can have any number of "triples", and there can be multiple "triples" in the same message containing PCRT and PNTF fields, since several processes may have to be created (i.e. forming a context, as described hereinabove) for the same resource.

As presently implemented, portion 150 is 16 bytes in length, field 156 is 4 bytes, field 157 is 4 bytes, field 158 is variable in length, and BOM portion 160 is 4 bytes.

#### HUMAN INTERFACE—GENERAL

The Human Interface of the present invention provides a set of tools with which an end user can construct a package specific to his applications requirements. Such a package is referred to as a "metaphor", since it reflects the user's particular view of the system. Multiple metaphors can be supported concurrently. One representative metaphor is, for example, a software development environment.

The purpose of the Human Interface is to allow consistent, integrated access to the data and functions available in the system. Since users' perceptions of the system are based largely on the way they interact with it, it is important to provide an interface with which they feel comfortable. The Human Interface allows a systems designer to create a model consisting of objects that are familiar to the end user and a set of actions that can be applied to them.

The fundamental concept of the Human Interface is that of the "picture". All visually-oriented information, regardless of interpretation, is represented by pictures. A picture (such as a diagram, report, menu, icon, etc.) is defined in a device-independent format which is recognized and manipulated by all programs in the Human Interface and all programs using the Human Interface. It consists of "picture elements", such as "line", "arc", and "text", which can be stored compactly and transferred efficiently between processes. All elements have common attributes like color and fill pattern. Most also have type-specific attributes, such as

typeface and style for text. Pictures are drawn in a large "world" co-ordinate system composed of "virtual pixels".

Because all data is in the form of pictures, segments of data can be freely copied between applications, e.g., from a live display to a word processor. No intermediate format or conversion is required. One consequence of this is that the end user or original equipment manufacturer (OEM) has complete flexibility in defining the formats of windows, menus, icons, error messages, help pages, etc. All such pictures are stored in a library rather than being built into the software and so are changeable at any time without reprogramming. A comprehensive editor is available to define and modify pictures on-line.

All interaction with the user's environment is through either "virtual input" or "virtual output" devices. A virtual input device accepts keyboards, mice, light pens, analog dials, pushbuttons, etc. and translates them into text, cursor-positioning, action, dial, switch, and number messages. All physical input devices must map into this set of standard messages. Only one process, an input manager for the specific device, is responsible for performing the translation. Other processes can then deal with the input without being dependent on its source.

Similarly, a virtual output manager translates standard output messages to the physical representation appropriate to a specific device (screen, printer, plotter, etc.) A picture drawn on any terminal or by a process can be displayed or printed on any device, subject to the physical limitations of that device.

With reference to FIG. 7, two "pictures" are illustrated picture A (170) and picture B (174).

The concept of a "view" is used to map a particular rectangular area of a picture to a particular device. In FIG. 7, picture A is illustrated as containing at least one view 171, and picture B contains at least one view 175. Views can be used, for example, to partition a screen for multiple applications or to extract page-sized subsets of a picture for printing.

If the view appears on a screen it is contained in a "window". With reference again to FIG. 7, view 171 of picture A is mapped to screen 176 as window 177, and view 175 of picture B is mapped as window 178.

The Human Interface allows the user to dynamically change the size of the window, move the window around on the screen, and move the picture under the window to view different parts of it (i.e., scroll in any direction). If a picture which is mapped to one or more windows changes, all affected views of that picture on all screens are automatically updated. There is no logical limit to the number or sizes of windows on a particular screen. Since the system is distributed, it's natural for pictures and windows to be on different nodes. For example, several alarm displays can share a single, common picture.

The primary mechanism for interacting with the Human Interface is to move the cursor to the desired object and "select" it by pressing a key or button. An action may be performed automatically upon selection or by further interaction, often using menus. For example, selecting an icon usually activates the corresponding application immediately. Selecting a piece of text is often followed by selection of a command such as "cut" or "underline". Actions can be dynamically mapped to function keys on a keyboard so that pressing a key is equivalent to selecting an icon or a menu item. A given set of cursors (the cursor changes as it moves from one application picture to another), windows, menus, icons, and function keys define a "metaphor".

The Human Interface builds on the above concepts to provide a set of distributed services. These include electronic mail, which allows two or more users at different terminals to communicate with each other in real time or to queue files for later delivery, and a forms manager for data entry. A subclass of windows called "virtual terminals" provides emulation of standard commercially available terminals.

FIG. 8 shows the different levels of the Human Interface and data flow through them. Arrows 201-209 indicate the most common paths, while arrows 210-213 indicate additional paths. The interface can be configured to leave out unneeded layers for customized applications. The philosophy behind the Human Interface design dictates one process per object. That is, a process is created for each active window, picture, input or output device, etc. As a result, the processes are simplified and can be distributed across nodes almost arbitrarily.

#### MULTIPLE INDEPENDENT PICTURES AND WINDOWS

A picture is not associated with any particular device, and it is of virtually unlimited size. A "window" is used to extract a specified rectangular area—called a "view"—of picture information from a picture and pass this data to a virtual output manager.

The pictures are completely independent of each other. That is, none is aware of the existence of any other, and any picture can be updated without reference to, and without affect upon, any other. The same is true of windows.

Thus the visual entity seen on the screen is really represented by two objects: a window (distinguished by its frame title, scroll bars, etc.), and a picture, which is (partially) visible within the boundaries of the window's frame.

As a consequence of this autonomy, multiple pictures can be updated simultaneously, and windows can be moved around on the screen and their sizes changed without the involvement of other windows and/or pictures.

Also, such operations are done without the involvement of the application which is updating the window. For example, if the size of a window is increased to look at a larger area of the picture, this is handled completely within the human interface.

#### HUMAN INTERFACE—PRIMARY FEATURES

The purpose of the Human Interface is to transform machine-readable data into human-readable data and vice versa. In so doing the Human Interface provides a number of key services which have been integrated to allow users to interact with the system in a natural and consistent manner. These features will now be discussed.

**Device Independence**—The Human Interface treats all devices (screens, printers, etc.) as "virtual devices" None of the text, graphics, etc. in the system are tied to any particular hardware configuration. As a result such representations can be entered from any "input" device and displayed on any "output" device without modification. The details of particular hardware idiosyncracies are hidden in low-level device managers, all of which have the same interface to the Human Interface software.

**Picture Drawing**—The Human Interface can draw "pictures" composed of any number of geometric elements, such as lines, circles, rectangles, etc., as well as any arbitrary shape defined by the user. Each element can have its own

color and line thickness. In addition, closed figures may be filled in with a particular shading pattern in any given color. A picture can be of almost any size. All output from the Human Interface to a user is via pictures, and all input from a user to the Human Interface is stored as pictures, so that there is only one representation of data within the Human Interface.

Text can be freely intermixed with graphical images, so that the user need only learn one "editor" to do his job. Consequently it is not necessary to switch between editors or "cut and paste" between pictures. Text characters can be selected from a large predefined character set, which includes mathematical and Greek symbols, among others, and can be typed in a wide variety of fonts, colors, sizes, and styles (e.g. bold, italic, or underlined). It is also possible for a user to define his own symbols and add them to the character set.

**Windowing**—The Human Interface allows the user to partition a screen into as many "sub-screens" or "windows" as required to view the information he desires. The Human Interface places no restrictions on the contents of such windows, and all windows can be simultaneously updated in real time with data from any number of concurrently executing programs. Any picture can be displayed, created, or modified ("edited") in any window. Also any window can be expanded or contracted, or it can be moved to a new location on the screen at any time.

If the current picture is larger than the current window, the window can be scrolled over the picture, usually in increments of a "line" or a "page". It is also possible to temporarily expand or contract the visible portion of the picture ("zoom in" or "zoom out") without changing the window's dimensions and without changing the actual picture.

**Dialog Management**—The Human Interface is independent of any particular language or visual representation. That is, there are no built-in titles, menus, error messages, help text, icons, etc. for interacting with the system. All such information is stored as pictures which can be modified to suit the end user's requirements, either prior to or after installation. The user can modify the supplied dialog with his own at any time.

**Data Entry**—The Human Interface provides a generalized interface between the user and any program (such as a data base manager) which requires data from the user. The service is called "forms management", because a given data structure is displayed as a fill-in-the-blanks type of "form" consisting of numerous modifiable fields with descriptive labels. The Human Interface form is interactive, so that data can be verified as it is entered, and the system can assist the user by displaying explanatory text when appropriate (on demand or as a result of an error).

**Communication Between Users**—The Human Interface permits two or more users to "converse" with each other in real time or to send "mail" to each other. Conversation is performed through a window on each of the user's screens. Mail is sent by creating a picture (text and/or diagrams) and specifying a destination. The destination may be one particular user, a group of users, or all users in the system (i.e. a "broadcast"). Transmission may be immediate or delayed until a given date and time or until the given user(s) sign onto the system. When mail arrives at the destination, the receiving user is informed and may then read, save, print, or erase the picture.

**Event Management**—The Human Interface can record any arbitrary event for future reference. The Human Interface defines a simple, yet flexible grammar for forming

"sentences" which describe events and which the Human Interface can use to parse in order to manipulate events for specific requests. For example, events can be dynamically displayed on a screen by time and/or priority, or they can be scanned for a particular "subject" or "object" or any other attribute. Each event can be time-stamped by the sender; if not, it is automatically time-stamped upon receipt.

The Human Interface records all of its own actions, such as printing a report or signing-on a user, and it provides this service to any applications program. In addition, the Human Interface can be requested to trigger any given action upon the occurrence of any given event, thus providing a kind of closed-loop control service to applications.

Modularity—The Human Interface comprises a number of separate software components which can be replicated and distributed throughout the hardware configuration to achieve optimal performance. For example, each time a new "console" (for example, keyboard plus screen) is connected to the system, a new "Console" component is created to manage it. There is no logical limit to the number of consoles that the Human Interface can handle. In general the relevant software component is located close to the hardware or other resources on which it most depends.

#### HUMAN INTERFACE—BASIC COMPONENTS

The Human interface comprises the following basic components:

Console Manager—It is the central component of a Console context and consequently is the only manager which knows all about its particular "console" It is therefore aware of all screens and keyboards, all windows, and all pictures. Its primary responsibility is to coordinate the activities of the context. This consists of starting up the console (initializing the device managers, etc.) creating and destroying pictures, and allocating and controlling windows for processes in the Human Interface and elsewhere. Thus all access to a console must be indirect, through the relevant Console Manager.

The Console Manager also implements the first level of Human Interface interaction, via menus, prompts, etc., so that applications processes don't have to. Rather than using built-in text and icons, it depends upon the Dialog Manager to provide it with the visible features of the system. Thus all cultural and user idiosyncracies (such as language) are hidden from the rest of the Human Interface.

A Console Manager knows about the following processes: the Output Manager(s) in its context, the Input Manager in its context, the Window Managers in its context, the Picture Managers in its context, and the Dialog Manager in its context. The following processes know about the Console Manager: any one that wants to.

When a Console Manager is started, it waits for the basic processes needed to communicate with the user to start up and "sign on". If this is successful, it is ready to talk to users and other processes (i.e., accept messages from the Input Manager and other processes). All other permanent processes in the context (Dialog, etc.) are assumed to be activated by the system start-up procedure. The "In" and "Cursor" processes (see "Input Manager" and "Output Manager" below) are created by the Console Manager at this time.

The Console Manager generally clears the entire screen and displays appropriate status text during the course of the start-up (by sending picture elements directly to its Output

Manager(s)). If any part of the start-up fails, it displays appropriate "error" text and possibly waits for corrective action from a user.

The Console Manager views the screen as being composed of blank (unused) space, windows, and icons. Whenever an input character is received, the Console Manager determines how to handle it depending upon the location of the cursor and the type of input, as follows:

A. Requests to create or eliminate a window are handled within the Console Manager. A window may be opened anywhere on the screen, even on top of another window. A new Picture Manager and possibly a Window Manager may be created as a result, and one or more new messages may be generated and sent to them, or the manager(s) may be told to quit.

B. Icons can only be selected, then moved or opened. The Console Manager handles selection and movement directly. It sends notification of an "open" to the Dialog Manager, which sends a notification to the application process associated with the icon and possibly opens a default window for it.

C. For window-dependent actions, if the cursor is outside all windows, the input is illegal, and the Console Manager informs the user; otherwise the input is accepted. Request which affect the window itself (such as "scroll" or "zoom") are handled directly by the Console Manager. A "select" request is pre-checked, the relevant picture elements are selected (by sending a message to the relevant Picture Manager), and the message is passed on to the process currently responsible for the window. All other inputs are passed directly to the responsible process without being pre-checked.

If the cursor is on a window's frame, the only valid actions are to move, close, or change the dimensions of the window, or select an object in the frame (such as a menu or a scroll bar). These are handled directly by the Console Manager.

D. Requests for Human interface services not in the Console context are treated as errors.

A new window is opened by creating a new Window Manager process and telling it its dimensions and the location of its upper left corner on the screen. It must also be given the PID of a Picture Manager and the coordinates of the part of the picture it is to display, along with the dimensions of a "clipping polygon", if that information is available. (It is not possible to create a window without a picture.) The type and contents of the window frame are also specified. Any of these parameters may be changed at any time.

A new instance of a picture is created by creating a new Picture Manager process with the appropriate name and, optionally, telling it the name of a "file" from which to get its picture elements. If a file is not provided, an "empty" picture is created, with the expectation that picture-drawing requests will fill it in.

Menus, prompts, help messages, error text, and icons are simply predefined pictures (provided through the Dialog Manager) which the Console Manager uses to interact with users. They can therefore be created and edited to meet the requirements of any particular system the same way any picture can be created and edited. Menus and help text are usually displayed on request, although they may sometimes be a result of another operation.

Prompts are displayed when the system needs information from the user. Error text is displayed whenever the user tries to do something that is illegal or when the system is having

problems of its own (e.g. "printer out of paper"). Icons are displayed by the Console Manager automatically when a specific frame of reference is requested by the user. The Console Manager may also display informational messages (such as "console starting up") which are automatically 5 erased when the associated action is finished.

**Picture Manager**—It is created when a picture is built, and it exits when the picture is no longer required. There is one Picture Manager per picture. The Picture Manager constructs a device-independent representation of a picture 10 using a small set of elemental "picture elements" and controls modification and retrieval of the elements.

A Picture Manager knows about the following processes: the process which created it, and the Draw Manager. The following processes know about the Picture Manager: the 15 Console Manager in the same context, and Window Managers in the same context.

A Picture Manager is created to handle exactly one picture, and it need only be created when that picture is being accessed. It can be told to quit at any time, deleting its 20 representation of the picture. Some other process must copy the picture to a file if it needs to be saved.

When a Picture Manager first starts up, its internal picture is empty. It must receive a "load file" request, or a series of "draw" requests, before a picture is actually available. Until 25 that is done any requests which refer to specific elements or locations in the picture will receive an appropriate "not found" status message.

A picture is logically composed of device-independent "elements", such as text, line, arc, and symbol. In general, 30 there is a small number of such elements. Each element consists of a common header, which includes the element's position in the picture's coordinate system, its color, size, etc., and a "value" which is unique to the element's type (e.g. a character string, etc.). The header also specifies how 35 the element combines with other elements in the picture (overlays them, merges with them, etc.). A special element type called "null" is also supported to facilitate the removal of picture elements from pictures or other similar large lists without forcing time-consuming compaction procedures. 40 Any element can therefore be redefined to "null", indicating that it should be ignored for all future processing.

The "null" color (zero) is treated as transparent when used in either the foreground or the background. Specifically, if the foreground color is null, the element itself is not drawn, 45 but it may still be filled in. If the background color is null, the element is not filled in. If the shading pattern is null, and the color is not null, the background fill is solid.

A picture is represented in an internal format which may be different from the external representation of picture 50 elements and which is, in any case, hidden from other processes. This representation is designed to optimize retrieval of picture elements, with a secondary emphasis on adding new elements and modifying or erasing old ones. The order in which the elements were originally drawn is preserved (unless explicit "order" requests have been received 55 to re-arrange them).

Requests to "animate" an element result in the creation of a separate, local "animate" process which performs the 60 necessary transformations and sends the appropriate requests (usually "draw" or "erase") back to the Picture Manager periodically.

A Picture Manager processes incoming requests one at a time, as it receives them. Each message can change the state of the picture for later requests. The Picture Manager 65 supports numerous operations, including the following: "draw" new elements; "modify", "overwrite", or "erase"

existing elements; "copy" or "move" elements to another location in the same picture or to any other given process; "group" elements together into one (or "ungroup" them); "scale" them (i.e. expand, stretch, or shrink them); and "rotate" them. It can also be asked to "notify" a particular 5 process if any elements within a given rectangular area (the "viewport") are changed and to determine whether a given location coincides (or come close to) any element in the picture. Any response to a request (e.g., multiple picture 10 elements) is sent in a single message.

When an element is sent as the result of an outstanding "notify" request, all elements which overlap it (and all elements which overlap those elements) are sent as well. These are sent together in one message. The background is 15 displayed by generating a "rectangle" element of the same size as the current viewport with a null foreground color and the appropriate background pattern and color. This element is always the lowest level in the picture; i.e., it is sent before all others. All erasure of elements from a display is accomplished by "draw" requests which redisplay the background 20 and/or elements in the picture, overwriting the "erased" elements. There is no explicit "erase" request to a window (or output) manager.

**Input Manager**—There is one Input Manager per set of "logical input devices" (such as keyboards, mice, light pens, 25 etc.) connected to the system. The Input Manager handles input interrupts and passes them to the console manager. Cursor movement inputs may also be sent to a designated output manager.

The Input Manager knows about the following processes: the process which initialized it, and possibly one particular Output Manager in the same context. The following process 30 knows about the Input Manager: the Console Manager in the same context.

An Input Manager is created (automatically, at system start-up) for each set of "logical input devices" in the 35 system, thus implementing a single "virtual keyboard". There can only be one such set, and therefore one Input Manager, per Console context. The software (message) interface to each manager is identical, although their internal behavior is dependent upon the physical device(s) to which they communicate. All input devices interrupt service routines (including mouse, digitizing pad, etc.) are contained in 40 Input Managers and hidden from other processes. When ready, each Input Manager must send an "I'm here" message to the closest process named "Console".

An Input Manager must be explicitly initialized and told to proceed before it can begin to process input interrupts. Both of these are performed using appropriate messages. 45 Whichever process initializes the manager becomes tightly coupled to it, i.e., they can exchange messages via PID's rather than by name. The Input Manager will send all inputs to this process (usually the Console Manager). This coupling cannot be changed dynamically; the manager would have to be re-initialized. Between the "initialize" and the "proceed" 50 an Input Manager may be sent one or more "set" requests to define its behavior. It does not need to be able to interpret the meaning of any input beyond distinguishing cursor from non-cursor. Device-independent parameters (such as pixel size and density) are not down-loaded but rather are assumed to be built into the software, some part of which, in 55 general, must be unique to each type of Input Manager.

An Input Manager can be dynamically "linked" to a particular Output Manager, if desired. If so, all cursor control input (or any other given subset of the character set) 60 will be sent to that manager, in addition to the initializing process, as it is received. This assignment can be changed or

cut off at any time. (This is generally useful only if the output device is a screen.)

In general, input is sent as single "characters", each in a single "K" (i.e. keyboard string) message (unbuffered) to the specified process(es). Some characters, such as "shift one" or a non-spacing accent, are temporarily buffered until the next character is typed and are then sent as a pair. Redefinable characters, including all displayable text, cursor control commands, "action keys", etc. are sent as triples.

New output devices can be added to the "virtual keyboard" at any time by re-initializing the manager and down-loading the appropriate parameters, followed by a "proceed". All input is suspended while this is being done. Previously down-loaded parameters and the screen assignment are not affected. Similarly, devices can be disconnected by terminating (sending "quit" requests for) them individually. A nonspecific "quit" terminates the entire manager.

Where applicable, an Input Manager will support requests to activate outputs on its device(s), such as lights or sound generators (e.g., a bell).

The Input Process is a distinct process which is created by each Console Manager for its Input Manager to keep track of the current input state. In general, this includes a copy of its last input of each type (text, function key, pointer, number, etc.), the current redefinable character set number, as well as Boolean variables for such conditions as "keyboard locked", "select key depressed" (and being held down), etc. The process is simply named "In". The Input Manager is responsible for keeping this process up-to-date. Any process may examine (but not modify) the contents of "In".

Output Manager—There is one Output Manager per physical output device (screen, printer, plotter, etc.) connected to the system. Each Output Manager converts (and possibly scales) standard "pictures" into the appropriate representation on its particular device.

The Output Manager knows about the following processes: the process which initialized it, and the Draw Manager in the same context. The following processes know about the Output Manager: the Console Manager in the same context, the Input Manager in the same context, and the Window Manager in the same context.

An Output Manager is created (automatically, at system start-up) for each physical output device in the system, thus implementing numerous "virtual screens". There can be any number of such devices per Console context. The software (message) interface to each manager is identical, although their internal behavior is dependent upon the physical device(s) to which they communicate. All output interrupt service routines (if any) are contained in Output Manager and hidden from other processes. Each manager also controls a process called Cursor which holds information concerning its own cursor. When ready, each Output Manager must send an "I'm here" message to the closest process named "Console".

An Output Manager must be explicitly initialized and told to proceed before it can begin to actually write to its device. Both of these are performed using appropriate Human Interface messages. Which process initializes the manager becomes tightly coupled to it; i.e., they can exchange messages via PID's rather than by name. This coupling cannot be changed dynamically; the manager would have to be re-initialized. Between the "initialize" and the "proceed" an Output Manager may be sent one or more "set" requests to define its behavior. Device-independent parameters (such as pixel size and density) are not down-loaded but rather are assumed to be built into the software, some part of which, in

general, must be unique to each type of Output Manager. Things like a screen's background color and pattern are down-loadable at start-up time and at any other time.

In general, an Output Manager is driven by "draw" commands (containing standard picture elements) sent to it by any process (usually a Window Manager). Its primary function then is to translate picture elements, described in terms of virtual pixels, into the appropriate sequences of output to its particular device. It uses the Draw Manager to expand elements into sets of real pixels and keeps the Cursor process informed of any resulting changes in cursor position. It looks up colors and shading patterns in predefined tables. The "null" color (zero) is interpreted as "draw nothing" whenever it is encountered. A "clear" request is also supported. It changes a given polygonal area to the screen's default color and shading pattern.

Any "draw" request can be preceded by a "clip" request. "Clip" means "don't display pixels outside of given polygon", i.e. only the logical AND of the polygonal area and the given picture elements is drawn. The clip request applies only to the next draw request received from the same process and is then discarded.

"Text" elements are displayed by the output device's built-in character generator, if possible. However, most text is created from predefined bit-maps which are stored in a Human Interface library. Different bit-maps exist for various combinations of font and size. Sizes which are not explicitly stored must be calculated from the available bit-maps when required. The style is always generated dynamically, i.e., it is calculated from the basic bit-map.

Output Managers also accept "K" messages (i.e. keyboard strings) containing cursor movement commands. If the associated device is a screen, the manager erases the cursor from its current position (if necessary, i.e. if the cursor is not supported directly by the hardware) and redraws it in its new location. It uses the Cursor Process to get a symbol element representing the cursor's current shape and color, and it tells it the new location after it has redrawn the cursor. (The manager may have to ask its initializing process to redraw the part of the picture which was previously obscured by the cursor after it moves it.) If the associated device is not a real screen, cursor movement commands are simply ignored.

If possible, an Output Manager should be able to save, restore, move, and copy rectangular areas of the virtual screen. These are primarily speed-optimizing operations, and they need not always be supplied. In general, an Output Manager can be queried for its characteristics, e.g., whether it supports the above functions, whether it is bit-mapped or character-oriented, the output dimensions (in pixels or characters, as appropriate), the physical size, etc.

The Cursor Process is a distinct process which is created by each Console Manager in its context to keep track of the cursor. That process, which has the same name as the screen (not the Output Manager), knows the current location of the cursor, all of the symbols which may represent the cursor on the screen, which symbol is currently being used, how many real pixels to move when a cursor movement command is executed, etc. It can, in general, be accessed for any of this information at any time by any process. The associated Output Manager is the prime user of this process and is responsible for keeping it up to date. The associated Input Manager (if any) is the next most common user, requesting the cursor's position every time it processes a "command" input.

Dialog Manager—There is one Dialog Manager per console, and it provides access to a library of "pictures" which define the menus, help texts, prompts, etc. for the Human

Interface (and possibly the rest of the system), and it handles the user interaction with those pictures.

The Dialog Manager knows about the following processes: none. The following processes know about the Dialog Manager: the Console Manager in the same context.

One Dialog Manager is created automatically, at system start-up, in each Console context. Its function is to handle all visual interaction with users through the input and output managers. Its purpose is to separate the external representation of such interaction from its intrinsic meaning. For example, the Console Manager may need to ask the user how many copies of a report he wants. The phrasing of the question and the response are irrelevant—they may be in English, Swahili, or pictographic, so long as the Console Manager ends up with an integral number or perhaps the response “forget it”.

In general, the Dialog Manager can be requested to load (from a file) or dynamically create (from a given specification) a picture which represents a menu, error message, help (informational) text, prompt, a set of icons, etc. This picture is usually displayed until the user responds.

Response to help or error text is simply acknowledgement that the text has been read. The response to a prompt is the requested information. The user can respond to a menu by selecting an item in the menu or by cancelling the menu (and thus cancelling any actions the menu would have caused). Icons can be selected and then moved or “opened”. Opening an icon generally results in an associated application being run.

“Selection” is done through an Input Manager which sends a notification to the Console Manager. The Console Manager filters this response through the Dialog Manager which interprets it and returns the appropriate parameter in a message which is then passed on to the process which requested the service.

All dialog is represented as pictures, mostly in free format. Help and error dialog are the simplest and are unstructured except that one element must be “tagged” to identify it as the “I have read this text” response target symbol. The text is displayed until the user selects this element.

Prompts have three tagged elements: one which defines the response area (i.e., where the user will type the information requested by the prompt), a “cancel” target, and an “enter” target. The prompt is displayed until either one of the latter two elements is selected. The response is returned as a text string, with an indication of which target element was selected. The “response” element may be omitted, in which case the prompt is just a question and the response is a simple yes or no (represented by “enter” and “cancel”).

A menu picture is highly structured. The first element must be a text element which contains the menu’s title for display and for reference by the software. This may be followed by an “explanation” element to describe the menu items. Neither of these elements is selectable.

The menu proper contains a list of “macro” picture elements, one per selectable choice or “item”. Each macro consists of three elements. The first element is mandatory and describes the item (via text or a symbol). It must contain a tag which is what is actually sent back to the requesting process when the item is selected, along with the item’s ordinal number (1 to n, of there are n items). For example, the item element may define an icon, such as a house. The tag might be “H” or “house” or anything else the system designer feels is appropriate. An item number of zero and a tag of “NONE” are sent if the menu is closed without selecting any item. A single character may optionally be

associated with the element. Typing the given character on the keyboard has the same effect as selecting the item from the menu.

The second and third elements in the macro are optional and may be represented by null strings (a single null byte) if not required. The second element describes the “alternate” state of the item. It is displayed when the item is selected and remains in effect until the item is selected again. In other words, the item is toggled between two options. The element must contain a tag (as described for the first element) to identify it. The third element describes the “unavailable” state of the item, and it is displayed when that particular option is marked as not being selectable at the time the menu is requested, as described below.

The last element in the menu picture is a simple text string consisting of a pair of characters for each item in the menu. The list describes whether the item is available (can be selected) or unavailable and which is its current state (normal or alternate). This list can (and should) be changed dynamically by messages to the Dialog Manager to reflect the current options available to the user.

Icons are small pictures which represent applications or services and are organized into sets (or “frames of reference”) of related functions. A set is a picture composed of “macro” elements, one per icon. Each macro comprises a single “symbol” element (which may itself be a macro) and a text element describing the label to be displayed with the symbol. The label element may be null. The macro element must be tagged with the name of the process to which notification is sent when the icon is “opened”, and it must specify whether a window should be opened automatically before sending the notification.

Draw Manager—There is one Draw Manager per console, and it provides access to a library of “pictures” which define the menus, help, prompts, etc., for the Human Interface (and possibly the rest of the system), and it handles the user interaction with those pictures.

The Draw Manager knows about the following processes: none. The following processes know about the Draw Manager: the Picture Managers in the same context, and the Output Managers in the same context.

One Draw Manager is created automatically, at system start-up, in each context that requires expansion of picture elements into bit-maps. Its sole responsibility is to accept one or more picture elements, of any type, in one message and return a list of bit-map (“symbol”) elements corresponding to the figure generated by the elements, also in one message. Various parameters can be applied to each element, most notably scaling factors which can be used to transform an element or to convert virtual pixels to real pixels. The manager must be told to exit when the context is being shut down.

Window Manager—There is one per current instance of a “window” on a particular screen. A Window Manager is created when the window is opened and exits when the window is closed. It maps a given picture (or portion thereof) to a rectangular area of a given size on the given screen; i.e., it logically links a device-independent picture to a device-dependent screen. A “frame” can be drawn around a window, marking its boundaries and containing other information, such as a title or menu. Each manager is also responsible for updating the screen whenever the contents of its window changes.

The Window Manager knows about the following processes: the process that created it; one particular Picture Manager in the same context; and one particular Output Manager in the same context. The following processes know

about the Window Manager: the Console Manager in the same context.

The Window Manager's main job is to copy picture elements from a given rectangular area of a picture to a rectangular area (called a "window") on a particular screen. To do so it interacts with exactly one Picture Manager and one Output Manager. A Window Manager need only be created when a window is "opened" on the screen and can be told to quit when the window is "closed" (without affecting the associated picture). When opened, the Window Manager must draw the outline, frame, and background of the window. When closed, the window and its frame must be erased (i.e. redrawn in the screen's background color and pattern). "Moving" a window (changing its location on the screen) is essentially the same as closing and re-opening it.

A Window Manager can only be created and destroyed by a Console Manager, which is responsible for arranging windows on the screen, resolving overlaps, etc. When a Window Manager is created, it waits for an "initialize" message, initializes itself, returns an "I'm here" message to the process which sent it the "initialize" message, then waits for further messages. It does not send any messages to the Output Manager until it has received all of the following: its dimensions (exclusive of frame), the outline line-type, size and color, background color, location on the screen, a clipping polygon, scaling factors, and framing parameters. A Window Manager also has an "owner", which is a particular process which will handle commands (through the Console Manager, which always has prime control) within the window.

Any of the above parameters can be changed at any time. In general, changing any parameter (other than the owner) causes the window to be redrawn on the screen.

A "frame", which may consist of four components (called "bars"), one along each edge of the window, may be placed around the given window. The bars are designated top, bottom, left, and right. They can be any combination of simple line segment, title bar, scroll bar, menu bar, and palette bar. These are supplied to the message as four separate lists (in four separate messages) of standard picture elements, which can be changed at any time by sending a new message referencing the bar. The origin of each bar is [0,0] relative to the upper left corner of the window.

The Console Manager may query a Window Manager for any of its parameters, to which it responds with messages identical to the ones it originally received. It can also be asked whether a given absolute cursor position is inside its window (i.e. inside the current clipping polygon) or its frame, and for the cursor coordinates relative to the origin of the window or any edge of the frame.

A Window Manager is tightly coupled to its creator (a Console Manager), Picture Manager, and Output Manager; i.e. they communicate with each other using process identifiers (PID's). Consequently, a Window Manager must inform its Picture Manager when it exits, and it expects the Picture Manager to do the same.

Once the Window Manager knows the picture it is accessing and the dimensions of its window (or any time either of these changes), it requests the Picture Manager to send it all picture elements which completely or partially lie within the window. It also asks it to notify it of changes which will affect the displayed portion of the picture. The Picture Manager will send "draw" messages to the Window Manager (at any time) to satisfy these requests.

The Window Manager performs gross clipping on all picture elements it receives, i.e. it just determines whether each element could appear inside the current clipping poly-

gon (which may be smaller than the window at any given moment, if other windows overlap this one).

A Window Manager can be told to "freeze" (stop updating) its display and to "unfreeze" it. It can also be asked to redraw any given rectangular sub-area of the picture it is displaying.

Window Managers deal strictly in virtual pixels and have no knowledge about the physical characteristics of the screen to which they are writing. Consequently, a window's size and location are specified in virtual pixels, implying a conversion from real pixels if these are different.

Print Manager—There is one per "output subsystem", i.e. per pool of output devices. The Print Manager coordinates output to hard-copy devices (i.e. to their Output Managers). It provides a comprehensive queuing service for files that need to be printed. It can also perform some minimal formatting of text (justification, automatic page numbering, headers, footers, etc.)

The Print Manager knows about the following processes: Output Managers in the same context, and a Picture Manager in the same context. The following processes know about the Print Manager: any one that wants to.

One Print Manager is created automatically, at start-up time, in each Print context. It is expected to accept general requests for hard-copy output and pass them on, one message (usually corresponding to one "line" of output) at a time, to the appropriate Output Manager. It can also accept requests which refer to files (i.e. to File Manager processes). Each such message, known as a "spool" request, also contains a priority, the number of copies desired, specific output device requirements (if any) and special form requirements (if any).

Based on these parameters, as well as the size of the file, the amount of time the request has been waiting, and the availability of output devices, the Print Manager maintains an ordered queue of outstanding requests. It dequeues them one at a time, select an Output Manager, and builds a picture (using a Picture Manager). It then requests (from the Picture Manager) and "prints" (plots, etc.) one "page" at a time until the entire file has been printed.

The Print Manager recognizes specially marked ("tagged") picture elements which define headers, footers, foot-notes, and page formatting parameters (such as "page break", "set page number", etc.).

#### HUMAN INTERFACE—RELATIONSHIPS BETWEEN COMPONENTS

The eight Human Interface components together provide all of the services required to support a minimal human interface. The relationships between them are illustrated in FIG. 9, which shows at least one instance of each component. The components represented by circles 301, 302, 307, 312, 315, and 317-320 are generally always present and active, while the other components are created as needed and exit when they have finished their specific functions. FIG. 9 is divided into two main contexts: "Console" 350 and "Print" 351.

Cursor 314 and Input 311 are examples of processes whose primary function is to store data. "Cursor"'s purpose is to keep track of the current cursor position on the screen and all parameters (such as the symbols defining different cursors) pertinent to the cursor. One cursor process is created by the Console Manager for each Output Manager when it is initialized. The Output Manager is responsible for updating the cursor data, although "Cursor" may be queried by anyone. "Input" keeps track of the current input state, such



as "select key is being held down", "keyboard locked", etc. One input process is created by each Console Manager. The console's input message updates the process; any other process may query it.

The Human Interface is structured as a collection of subsystems, implemented as contexts, each of which is responsible for one broad area of the interface. There are two major contexts accessible from outside the Human Interface: "Console" and "Print". They handle all screen/keyboard interaction and all hard-copy output, respectively. These contexts are not necessarily unique. There may be one or more instances of each in the system, with possibly several on the same cell. Within each, there may be several levels of nested contexts.

The possible interaction between various Human Interface components will now be described.

**Console Manager/Other Contexts**—Processes of other contexts may send requests for console services or notification of relevant events directly to the Console Manager(s). The Console Manager routes messages to the appropriate service. It also notifies (via a "status" message) the current owner of a window whenever an object in its window has been selected. Similarly, it sends a message to an application when a user requests that application in a particular window.

**Console Manager/Input Manager**—The Console Manager initializes the Input Manager and usually assigns a particular Output Manager to it. The Input Manager always sends all input (one character, one key, one cursor movement, etc. at a time) directly to the Console Manager. It may also send "status" messages, either in response to a "download", "initialize", or "terminate" request, or any time an anomaly arises.

**Console Manager/Output Manager**—The Console Manager displays information on its "prime" output device during system start-up and shut-down without using pictures and windows. It therefore sends picture elements directly to an Output Manager. The Console Manager is also responsible for moving the cursor on the screen while the system is running, if applicable. The Console Manager (or any other Human Interface manager, such as an "editor") may change the current cursor to any displayable symbol. Output Managers will send "status" messages to the Console Manager any time an anomaly arises.

**Console Manager/Picture Manager**—The Console Manager creates Picture Managers on demand and tells each of them the name of a file which contains picture elements, if applicable. A Picture Manager can also accept requests from the Console Manager (or anyone else) to add elements to a picture individually, delete elements, copy them, move them, modify their attributes, or transform them. It can be queried for the value of an element at (or close to) a given location within its picture. The Console Manager will tell a Picture Manager to erase its picture and exit when it is no longer needed. A Picture Manager usually sends "status" messages to the Console Manager whenever anything unusual (e.g., an error) occurs.

**Console Manager/Window Manager**—The Console Manager creates Window Managers on demand. Each Window Manager is told its size, the PID of an Output Manager, the coordinates (on the screen) of its upper left outside corner, the characteristics of its frame, the PID of a particular Picture Manager, the coordinates of the first element from which to start displaying the picture, and the name of the process which "owns" the window. While a window is active, it can be requested to re-display the same picture starting at a different element or to display a completely different picture.

The coordinates of the window itself may be changed, causing it to move on the screen, or it may be told to change its size, frame, or owner. A Window Manager can be told to "clip" the picture elements in its display along the edges of a given polygon (the default polygon is the inside edge of the window's frame). It can also be queried for the element corresponding to a given coordinate. The Console Manager will tell a Window Manager to "close" (erase) its window and exit when it is no longer needed. A Window Manager sends "status" messages to the Console Manager to indicate success or failure of a request.

**Console Manager/Dialog Manager**—The Dialog Manager accepts requests to load and/or dynamically create "pictures" which represent menus, prompts, error messages, etc. In the case of interactive pictures (such as menus), it also interprets the response for the Console Manager. Other processes may also use the Dialog Manager through the Console Manager.

**Console Manager/Prime Manager**—Console Managers generally send "spool" requests to Print Managers to get hard-copies of screens or pictures. An active picture must first be copied to a file. The Print Manager returns a "status" message when the request is complete or if it fails.

**Window Manager/Picture Manager**—A Window Manager requests lists of one or more picture elements from the relevant Picture Manager, specified by the coordinates of a rectangular "viewport" in the picture. It can also request the Picture Manager to automatically send changes (new, modified, or erased elements), or just notification of changes, to it. The Picture Manager sends "status" messages to notify the Window Manager of changes or errors.

**Window Manager/Output Manager**—A Window Manager sends lists of picture elements to its Output Manager, prefixed by the coordinates of a polygon by which the Output Manager is to "clip" the pixels of the elements as it draws them. A given list of picture elements can also be scaled by a given factor in any of its dimensions. The Output Manager returns a "status" message when a request fails.

**Input Manager/Output Manager**—The Input Manager sends all cursor movement inputs to a pre-assigned Output Manager (if any), as well as to the Console Manager. This assignment can be changed dynamically.

**Print Manager/Other Processes**—The Print Manager accepts requests to "spool" a file or to "print" one or more picture elements. It sends a "status" message at the completion of the request or if the request cannot be carried out. The status of a queued request can also be queried or changed at any time.

**Print Manager/File Manager**—The Print Manager reads picture elements from a File Manager (whose name was sent to it via a "spool" request). It may send a request to "delete" the file back to the File Manager after it has finished printing the picture.

**Print Manager/Picture Manager**—A Print Manager creates a Picture Manager for each spooled picture that it is currently printing, giving it the name of the relevant file. It then requests "pages" of the picture (depending upon the characteristics of the output device) one at a time. Finally, it tells the Picture Manager to go away.

**Print Manager/Output Manager**—The Print Manager sends picture elements to an Output Manager. The Output Manager sends a "status" message when the request completes or fails or when an anomaly arises on the printer.

**Draw Manager/Other Processes**—The Draw Manager accepts lists of elements prefixed by explicit pixel param-



eters (density, scaling factor, etc.). It returns a single message containing a list of bit-map ("symbol") elements of the drawn result for each message it receives.

#### HUMAN INTERFACE—SERVICE

A Human Interface service is accessed by sending a request message to the closest (i.e. the "next") Human Interface manager, or directly to a specific Console Manager. This establishes a "connection" to an existing Human Interface resource or creates a new one. Subsequent requests must be made directly to the resource, using the connector returned from the initial request, until the connection is broken. The Human Interface manager is distributed and thus spans the entire virtual machine. Resources are associated with specific nodes.

A picture may be any size, often larger than any physical screen or window. A window may only be as large as the screen on which it appears. There may be any number of windows simultaneously displaying pictures on a single screen. Updating a picture which is mapped to a window causes the screen display to be updated automatically. Several windows may be mapped to the same picture concurrently—at different coordinates.

The input model provided by the Human Interface consists of two levels of "virtual devices" The lower level supports "position", "character", "action", and "function key" devices associated with a particular window. These are supported consistently regardless of the actual devices connected to the system.

An optional higher level consists of a "dialog service", which adds "icons", "menus", "prompts", "values", and "information boxes" to the repertoire of device-independent interaction. Input is usually event-driven (via messages) but may also be sampled or explicitly requested.

All dimensions are in terms of "virtual pixels" A virtual pixel is a unit of measurement which is symmetrical in both dimensions. It has no particular size. Its sole purpose is to define the spatial relationships between picture elements. Actual sizes are determined by the output device to which the picture is directed, if and when it is displayed. One virtual pixel may translate to any multiple, including fractions, of a real pixel.

Using the core Human Interface services generally involves: creating a picture (or accessing a predefined picture); creating a window on a particular screen and connecting the picture to it; updating the picture (drawing new elements, moving or erasing old ones, etc.) to reflect changes in the application (e.g. new data); if the application is interactive, repeatedly accepting input from the window and acting accordingly; and deleting the picture and/or window when done.

Creating a new resource is done with an appropriate "create" message, directed to the appropriate resource manager (i.e. the Human Interface manager or Console Manager). Numerous options are available when a resource, particularly a window, is created. For example, a typical application may want to be notified when a specific key is pressed. Pop-up and pull-down menus, and function keys, may also be defined for a window.

All input from the Human Interface is sent by means of the "click" message. The intent of this message is to allow the application program to be as independent of the external input as possible. Consequently, a "click" generated by a pop-up menu looks very much like that generated by pressing a function key or selecting an icon. Event-driven input

is initiated by a user interacting with an external device, such as a keyboard or mouse. In this case, the "click" is sent asynchronously, and multiple events are queued.

A program may also explicitly request input, using a menu, prompt, etc., in which case the "click" is sent only when the request is satisfied. A third method of input, which doesn't directly involve the user, is to query the current state of a virtual input device (e.g., the current cursor position).

A "click" message is associated with a particular window (and by implication usually with a particular picture), or with a dialog "metaphor", thus reflecting the two levels of the input model.

Since the visual aspect of the Human Interface is separated from the application aspect, a later redesign of a window, menu, icon, etc. has little or no effect upon existing applications.

#### HUMAN INTERFACE—DETAILED DESCRIPTION

##### Connectors

In general, all interaction with a Human Interface resource (console, window, picture, or virtual terminal) must be through a connector to that resource. Connectors to consoles can only be obtained from the Human Interface manager. Connectors to the other resources are available through the Human Interface manager, or through the Console Manager in which the desired resource resides. Requests must specify the path-name of the resource as follows:

```
[<console_name>][/<screen_name>][/<window_or_picture_name>]
```

That is, the name of the console, optionally followed by a slash and the name of the screen, optionally followed by a slash and the name of a window, picture, or terminal. The console name may be omitted only if the message is sent directly to the desired console manager. If the screen name is omitted, the first screen configured on the given console is assumed. The window name must be specified if one of those resources is being connected.

##### Connection Requests

The "create" and "open" requests can be addressed to the "next" Human Interface context ("HI") or to a specific console connector or to the "next" context named "Console". If sent to "HI", a full path-name (the name parameter) must be given; otherwise, only the name of the desired resource is required (e.g., at a minimum, just the name of the window or picture).

If a picture manager process is created locally by an application, for private use, an "init" message—with the same contents as "create" or "open" must be sent directly to the picture process. The response will be "done" or "failed".

The following are the various Connection Requests and the types of information which may be associated with each:

CREATE is used to create a new picture resource, a new window resource, or a new virtual terminal resource.

When used to create a new picture resource, it may contain information about the resource type (i.e. a "picture"); the path-name of the picture; the size; the background color; the highlighting method; the maximum number of elements; the maximum element size; and the path-name of a library picture from which other elements may be copied.

When used to create a new window resource, it may contain information about the resource type (i.e. a "window"); the path-name of the window; the window's title; the window's position on the screen; the size of the window; the color, width, fill color between the outline and the pane, and the style of the main window outline; the color and width of the pane outline; a mapping of part of a picture into the window; a modification notation; a special character notation; various options; a "when" parameter requesting notification of various specified actions on/within the window; a title bar; a palette bar; vertical and horizontal scroll bars; a general use bar; and a corner box.

When used to create a new virtual terminal, it may contain information about the resource type (i.e. a "terminal"); the path-name of the terminal; the title of the terminal's window; various options; the terminal's position on the screen; the size of the terminal (i.e. number of lines and columns in the window); the maximum height and width of the virtual screen; the color the text inside the window; tab information; emulator process information; connector information to an existing window; window frame color; a list of menu items; and alternative format information.

OPEN is used to connect to a Human Interface service or to an existing Human Interface resource. When used to connect to a Human Interface service, it may contain information about the service type; and the name of the particular instance of the service. This resource must be sent to the Human Interface context.

When used to connect to an existing Human Interface resource, it may contain information about the path-name of the resource; the type of resource (e.g. picture, window, or terminal); and the name of the file (for pictures only) from which to load the picture. This request can be sent to a Human Interface manager or a console manager; alternatively the same message with message I.D. "init" specifying a file can be sent directly to a privately owned picture manager.

DELETE is used to remove an existing Human Interface resource from the system, and it may contain information specifying a connection to the resource; the type of resource; and whether, for a window, the corresponding picture is to be deleted at the same time.

CLOSE is used to break a connection to a Human Interface resource, and it may contain information specifying a connection to the resource; and the type of resource.

WHO? is used to request a list of signed-on users, and it may contain a user identification string.

QUERY is used to get the status of a service or resource, and it may contain information about the resource type; the name of the service or resource; a connector to a resource; and information concerning various options.

The following are the various Connection Responses and the types of information which may be associated with each:

CONNECT provides a connection to a Human Interface resource, and it contains information concerning the originator (i.e. the Human Interface or the console); the resource type; the original request message identifier; the name of the resource; and a connector to the resource.

USER contains the names of zero or more currently signed-on users and their locations, and it contains a connector to a console manager followed by the name of the user signed on at that console.

#### Console Requests

The main purpose of the console is to coordinate the activities of the windows, pictures, and dialog associated

with it. Any of the CREATE, OPEN, DELETE, and CLOSE connection requests listed above, except those relating to the consoles, can be sent directly to a known console manager, rather than to the Human Interface manager (which always searches for the console by name). Subsequently, some characteristics of a window, such as its size, can be changed dynamically through the console manager. The current "user" of the console can be changed. And the console can be queried for its current status (or that of any of its resources).

The following are the various Console Requests and the types of information which may be associated with each:

USER is used to change the currently signed-on user, and it contains a user identification string.

CHANGE is used to change the size and other conditions of a window, and it may contain information about a connector to a window or a terminal; new height and width (in virtual pixels); increment to height and width; row and column position; various options; a connector to a new owner process; and whether the window should be the current active window on the screen.

CURSOR is used to move the screen cursor, and it contains position information as to row and column.

QUERY is used to get the current status of the console or one of its resources, and it contains information in the form of a connector to the resource; and various query options (e.g. list all screens, all pictures, or all windows).

BAND starts/stops the rubber-banding function and dragging function, and it contains information about the position of a point in the picture from which to start the operation; the end point of the figure which is to be dragged; the type of operation (e.g. line, rectangle, circle, or ellipse); the color; and the type of line (e.g. solid). In rubber-banding the drawn figure changes in size as the cursor is moved. In dragging the figure moves with the cursor.

The following are the various Console Responses and the types of information which may be associated with each:

STATUS describes the current state of a console, and it may contain information about a connector to the console; the originator; the name of the console; current cursor position; current metaphor size; scale of virtual pixels per centimeter, vertically and horizontally; number of colors supported; current user i.d. string; screen size and name; window connector and name; and picture connector, screen name, and window name.

#### Picture-Drawing

The picture is the fundamental building block in the Human Interface. It consists of a list of zero or more "picture elements", each of which is a device-independent abstraction of a displayable object (line, text, etc.). Each currently active picture is stored and maintained by a separate picture manager. "Drawing" a picture consists of sending picture manipulation messages to the picture manager.

A picture manager must first be initialized by a CREATE or OPEN request (or INIT, if the picture was created privately). CREATE sets the picture to empty, gives it a name, and defines the background. The OPEN request reads a predefined picture from a file and gives it a name. Either must be sent first before anything else is done. A subsequent OPEN reloads the picture from the file.

The basic request is to WRITE one or more elements. WRITE adds new elements to the end of the current list, thus

reflecting the order. Whenever parts of the picture are copied or displayed, this order is preserved. Once drawn, one or more elements can be moved, erased, copied, or replaced. All or part of the picture can be saved to a given file. In addition, there are requests to quickly change a particular attribute of one or more elements (e.g. select them). Finally, the DELETE request (to the console manager; QUIT, if direct to the picture resource) terminates the picture manager, without saving the picture.

Any single element can be "marked" for later reference. If the element is text, then a particular offset in the string can be marked, and a visible mark symbol displayed at that location.

A picture can be shared among several processes ("applications") by setting the "appl" field in the picture elements. Each application process can treat the picture as if it contains only its own elements. All requests made by each process will only affect elements which contain a matching "appl" field. Participating processes must be identified to the picture manager via an "appl" request.

The following are the various Picture-Drawing Requests and the types of information which may be associated with each:

WRITE is used to add new elements to a picture, and it may contain information providing a list of picture elements; the data type; and an indication to add the new elements after the first element found in a given range (instead of the foreground, at the end of the list).

READ is used to copy elements from a picture, and it may contain information regarding the connection to which to send the elements; an indication to copy background elements; and a range of elements to be copied.

MOVE is used to move elements to another location, and it may contain information indicating a point in the picture to which the elements are to be moved; row and column offsets; to picture foreground; to picture background; fixed size increments; and a range of elements to be moved.

REPLACE is used to replace existing elements with new ones, and it may contain information providing a list of picture elements; and a range of elements to be replaced.

ERASE is used to remove elements from a picture, and it may contain information on the range of elements to be erased.

QUIT is used to erase all elements and terminate, and it has no particular parameters (valid only if the picture is private).

MARK is used to set a "marked" attribute (if text, to display a mark symbol), and it may contain information specifying the element to be marked; and the offset of the character after which to display the mark symbol.

SELECT is used to select an element and mark it, and it may contain information specifying the element(s) to be selected; the offset of the character after which to display the mark symbol; the number of characters to select; and a deselect option.

SAVE is used to copy all or part of a picture to a file, and it may contain information specifying the name of the file; and a subset of a picture.

QUERY is used to get the current status, and it has no particular parameters.

BKGD is used to change a picture's background color, and it may contain information specifying the color.

APPL is used to register a picture as an "application", and it may contain information specifying a name of the

application; a connection to the application process; and a point of origin inside the picture.

NUMBER is used to get ordinal numbers and identifiers of specific elements, and it may contain information specifying the element(s).

HIT is used to find an element at or closest to a given position, and it may contain a position location in a picture; and how far away from the position the element can be.

[.] is used to start/end a batch, and a first symbol causes all updates to be postponed until a second symbol is received (batches may be nested up to 10 deep).

HIGHLIGHT, INVERT, BLINK, HIDE are used to change a specific element attribute, and they may contain information indicating whether the attribute is set or cleared; and a range of elements to be changed.

CHANGE is used to change one or more element fields, and it may contain information specifying the color of the element; the background color; the fill color; and fill pattern; and a range of elements to be changed.

EDIT is used to modify a text element's string, and it may contain information indicating to edit at the current mark and then move the mark; specifying the currently selected substring is to be edited; an offset into the text at which to insert and/or from which to start shifting; to shift the text by the given number of characters to/from the given position; tab spacing; a replacement substring; to blank to the end of the element; and a range of elements to be edited.

In general, when a range of elements is specified, a list of one or more parameters is provided (if omitted, then all elements in the picture are referenced by default) according to the following table:

Keyword	Meaning	Format
@pos	by position (start of range)	row, column
@end	last position of a range	row, column
@num	by relative element number	list of numbers
@tag	search for a tag	pattern
@txt	search for a text element	pattern
@sel	"selected" element(s)	keyword only
@mrk	"marked" element	keyword only
@id	by unique element identifier	list of identifiers
@att	by attributes	attribute structure
@cnt	the number of elements	count

Any range parameters which are given restrict the elements which will be affected by the current request. In general, only the intersection of all of the elements satisfying the given conditions are included in the range. For example, specifying pos, end, tag, txt, and sel together means "use all selected text elements between the given coordinates, containing a particular tag and an particular text string.

The following are the various Picture-Drawing responses and the types of information which may be associated with each:

STATUS describes the current status of the picture, and it may contain information specifying a connector to the picture; an original message identifier, if applicable; the name of the picture; the name of the file last read or written; height and width; lowest and highest row/column in the picture; the number of elements; and the number of currently active viewports.

WRITE contains elements copied from a picture, and it may contain information specifying a connector to the picture; a list of picture elements; and the data type.

NUMBER contains element numbers and identifiers, and it may contain information specifying a list of numbers, and a list of element identifiers.

### Picture Elements

Picture elements are defined by a collection of data structures, comprising one for a common "header", some optional structures, and one for each of the possible element types. The position of an element is always given as a set of absolute coordinates relative to [0,0] in the picture. This defines the upper left corner of the "box" which encloses each element. Points specified within an element (e.g. to define points on a line) are always given as coordinates relative to this position. In a "macro" the starting position of each individual element is considered to be relative to the absolute starting position of the macro element itself, i.e. they're nested.

FIG. 10 shows the general structure of a complete picture element. The "value" part depends upon the element type. The "appl" and "tag" fields are optional, depending upon indicators set in "attr".

The following is a description of the various fields in a picture element:

Length=length of the entire picture element in bytes  
 Type=one of the following: text, line, rectangle, ellipse, circle, symbol, array, discrete, macro, null, meta-element  
 Attr=one of the following: selectable, selected, rectilinear, inverted foreground/background, blink, tagged, application mnemonic, hidden, editable, movable, copyable, erasable, transformed, highlighted, mapped/not mapped, marked, copy  
 Pos=Row/col coordinates of upper left corner of the element's box  
 Box=Height/width of an imaginary box which completely and exactly encloses the element  
 Color=color of the element, consisting of 3 sub-fields: hue, saturation, and value  
 Bkgrnd=background color of the element  
 Fill=the color of the interior of a closed figure  
 Pattern=one of 10 "fill" patterns  
 Appl=a mnemonic referencing a particular application (e.g. forms manager, word-processor, report generator, etc.); allows multiple processes to share a single picture.  
 Tag=a variable-length, null-terminated string, supplied by the user; it can be used by applications to identify particular elements or classes of elements, or to store additional attributes  
 The attributes relating to the "type" field if designated "text" are as follows:  
 Options=wordwrap, bold, underline, italic, border, left-justify, right-justify, centered, top of box, bottom of box, middle of box, indent, tabs, adjust box size, character size, character/line spacing, and typeface  
 Select=indicates a currently selected substring by offset from beginning of string, and length  
 String=any number of bytes containing ASCII codes, followed by a single null byte; the text will be constrained to fit within the element's "box", automatically breaking to a new row when it reaches the right boundary of the area

Indent=two numbers specifying the indentation of the first and subsequent rows of text within the element's "box"  
 Tabs=list of [type, position], where "position" is the number of characters from the left edge of the element's box, and "type" is either Left, Right, or Decimal  
 Grow=maximum number of characters (horizontally) and lines (vertically) by which the element's box may be extended by typed input; limits growth right and downward, respectively  
 Size=height of the characters' extent and relative width  
 Space=spacing between lines of text and between characters  
 Face=name of a particular typeface  
 The attributes relating to the "type" field if designated "line" are as follows:  
 Style=various options such as solid, dashed, dotted, double, dashed-dotted, dash-dot-dot, patterned, etc.  
 Pattern=a pattern number  
 Thick=width of the line in pixels  
 Points=two or more pairs of coordinates (i.e. points) relative to the upper left corner of the box defined in the header  
 The attributes relating to the "type" field if designated "rectangle" are as follows:  
 Style=same as for "line" above, plus solid with a shadow  
 Pattern=same as for "line"  
 Thick=same as for "line"  
 Round=radius of a quarter-circle arc which will be drawn at each corner of the rectangle  
 The attributes relating to the "type" field if designated "ellipse" are as follows:  
 Style=solid, patterned, or double  
 Pattern=same as for "line"  
 Thick=same as for "line"  
 Arc=optional start- and end-angles of an elliptical arc  
 The attributes relating to the "type" field if designated "circle" are as follows:  
 Style=same as for "ellipse"  
 Pattern=same as for "line"  
 Thick=same as for "line"  
 Center=a point specifying the center of the circle, relative to the upper left corner of the element's box  
 Radius=length of the radius of the circle  
 Arc=optional start- and end-angles of a circular arc  
 A "symbol" is a rectangular space containing pixels which are visible (drawn) or invisible (not drawn). It is represented by a two-dimensional array, or "bit-map" of 1's and 0's with its origin in the upper left corner.  
 The attributes relating to the "type" field if designated "symbol" are as follows:  
 Bitmap=a two-dimensional array (in row and column order) containing single bits which are either "1" (draw the pixel in the foreground color) or "0" (draw the pixel in the background color); the origin of the array corresponds to the starting location of the element  
 Alt=A text string which can be displayed on non-bit-mapped devices, in place of the symbol  
 An array element is a rectangular space containing pixels which are drawn in specific colors, similar to a symbol element. It is represented as a two-dimensional array, or "bit-map", of color numbers, with its origin in the upper left corner. The element's "fill" and "pattern" are ignored.

The attributes relating to the "type" field if designated "array" are as follows:

Bitmap—a two-dimensional array (in row and column order) of color numbers; each number either defines a color in which a pixel is to be drawn, or is zero (in which the pixel is drawn in the background color); the origin of the array corresponds to the starting location of the element

Alt—an alternate text string which can be displayed on non-bit-mapped devices in place of the array

A discrete element is used to plot distinct points on the screen, optionally with lines joining them. Each point is specified by its coordinates relative to the element's "box". An explicit element (usually a single-character text element or a symbol element) may be given as the mark to be drawn at each point. If not, an asterisk is used. The resulting figure cannot be filled.

The attributes relating to the "type" field if designated "discrete" are as follows:

Mark—a picture element which defines the "mark" to be drawn at each point; if not applicable, a null-length element (i.e., a single integer containing the value zero) must be given for this field

Style Pat Thick =type, pattern, and thickness of the line (see "line" element above)

Join="Y" or "N" (or null, which is equivalent to "N"); if "Y", lines will be drawn to connect the marks

Points=two or more pairs of coordinates; each point is relative to the upper left corner of the "box" defined in the header

A "macro" element is a composite, made up of the preceding primitive element types ("text", etc.) and/or other macro elements. It can sometimes be thought of as "bracketing" other elements. The coordinates of the contained elements are relative to the absolute coordinates of the macro element. The "length" field of the macro element includes its own header and the "macro" field, plus the sum of the lengths of all of the contained elements. The "text" macro is useful for mixing different fonts and styles in single "unit" (word, etc.) of text.

The attributes relating to the "type" field if designated "macro" are as follows:

Macro=describes the contents of the macro element; may be one of following:

"N"—normal (contained elements are complete)

"Y"—list: same as "N", but only one sub-element at a time can be displayed; the others will be marked "hidden", and only the displayed element will be sent in response to requests ("copy", etc.); the "highlight" request will cycle through the sub-elements in order

"T"—text: same as "N", but the "macro" field is immediately followed by a text "options" field, and a text "select" field; the macro "list" field may be followed by further text parameters (as specified in the options field)

List=any number of picture elements (referred to as sub-elements), formatted as described above; terminated by a null word

A "meta-element" is a pseudo-element generated by the picture manager and which describes the picture itself, whenever the picture is "saved" to a file. Subsequently, meta-elements read from a file are used to set up parameters pertinent to the picture, such as its size and background color. Meta-elements never appear in "write" messages issued by the picture manager (e.g. in response to a "read" request, or as an update to a window manager).

The format of the meta-element includes a length field, a type field, a meta-type field, and a value. The 16-bit length field always specifies a length of 36. The type field is like that for normal picture elements. The meta-element field contains one of the following types:

Name—the value consists of a string which names the picture

Size—the maximum row and column, and the maximum element number and size

Backgnd—the picture's background color

Hight—the picture's highlighting

The format of the value field depends upon the meta-type.

### Windowing

A window maps a particular subset (often called a "view") of a given picture onto a particular screen. Each window on a screen is a single resource which handles the "pane" in which the picture is displayed and up to four "frame bars".

With reference to FIG. 11, a frame bar is used to show ancillary information such as a title. Frame bars can be interactive, displaying the names of "pull-down" menus which, when selected, display a list of options or actions pertinent to the window. A palette bar is like a permanently open menu, with all choices constantly visible.

Scroll bars indicate the relative position of the window's view in the picture and also allow scrolling by means of selectable "scroll buttons". A "resize" box can be selected to expand or shrink the size of the window on the screen while a "close" box can be selected to get rid of the window. Selecting a "blow-up" box expands the window to full screen size; selecting it again reduces it to its original dimensions.

A corner box is available for displaying additional user information, if desired.

The window shown in FIG. 11 comprises a single pane, four frame bars, and a corner box. The rectangular element within each scroll bar indicates the relative position of the window in the picture to which it is mapped (i.e. about a third of the way down and a little to the right).

Performing an action (such as a "select") in any portion of the window will optionally send a "click" message to the owner of the window. For example, selecting an element inside the pane will send "click" with "action"="select" and "area"="inside", as well as the coordinates of the cursor (relative to the top left corner of the picture) and a copy of the element at that position.

Selecting the name of a menu, which may appear in any frame bar, causes the menu to pop-up. It is the response to the menu that is sent in the "click" message, not the selection of the menu bar item. Pop-up menus (activated by menu keys on the keyboard) and function keys can also be associated with a particular window.

All windows are created by sending a "create" request to a Console Manager. As described above, "create" is the most complex of the windowing messages, containing numerous options which specify the size and location of the window, which frame bars to display, what to do when certain actions are performed in the window, and so on.

The process which sent the request is known as the "owner" of the window, although this can be changed dynamically. The most recently opened window usually becomes the current "active" window, although this may be overridden or changed.

A subsequent "map" request is necessary to tell the window which picture to display (if not specified in the "create" request). "Map" can be re-issued any number of times.

Other requests define pop-up menus and soft-keys or change the contents of specific frame bars. A window is always opened on top of any other window(s) it overlaps. Depending upon the background specified for the relevant picture, underlying windows may or may not be visible.

The "delete" request unmaps the window and causes the window manager to exit. The owner of the window (if different from the sender of "delete") is sent a "status" message as a result.

The following are the various Windowing Requests and the types of information which may be associated with each:

MAP is used to map or re-map a picture to the window, and it may contain information specifying a connection to the desired picture; and the coordinates in the picture of the upper left corner of the "viewport", which will become [0,0] in the window's coordinate system.

UNMAP is used to disconnect a window from its picture, and it contains no parameters.

QUERY is used to get a window's status, and it contains no parameters.

[.] is used to start/end a "batch", and the presence of a first symbol causes all updates to be postponed until a second symbol is received (batches may be nested up to 10 deep).

MENU defines a menu which will "pop-up" when a menu key is pressed, and it may contain information specifying which menu key will activate the menu; the name of the menu in the Human Interface library (if omitted, "list" must be given); and a name which is returned in the "click" message.

KEYS defines "pseudo-function" keys for the window, and it may contain information specifying the name of a menu in the Human Interface library; a list of key-names; and a name to be returned in the "click" message.

ADD, COPY, ERASE, REPLACE control elements in a frame bar, and they may contain information specifying the type of bar (e.g. title, palette, general, etc.); a list of picture elements for "add" and "replace" (omitted for "copy" and "erase"); and a tag identifying a particular element (not applicable to "add").

HIGHLIGHT, INVERT, HIDE, BLINK change attributes in a frame bar element, and they may contain information specifying a set/clear attribute; the type of bar; and a tag identifying a particular element in the bar.

The following are the various Windowing responses and the types of information which may be associated with each:

STATUS describes the current status of the window, and it may contain information specifying a connector to the window; specifying the originator (i.e. "window"); an original message identifier, if applicable; the subsystem; the name of the window; a connector to the window's console manager; the position of the window on the screen; the pane size and location; a connector to the picture currently mapped to the window; and the size and position of the view.

BAR represents a request to a "copy" request, and it may contain information specifying the type of bar (e.g. title, palette, general, corner box, etc.); and a list of picture elements.

CLICK describes a user-initiated event on or inside the window, and it may contain information specifying

what event (e.g. inside a pane, frame bar, corner box, pop-up menu, function key, etc.); a connector to the window manager; a connector to the window's Console Manager; the name of the window; a menu or function-key name; a connector to the associated picture manager; a label from a menu or palette bar item or from a function key; the position of the cursor where the action occurred; the action performed by the user; a copy of the elements at the particular position; the first element's number; the first element's identifier; a copy of the character typed or a boundary indicator or the completion character; and other currently selected elements from all other windows, if any.

### Virtual Terminal

In general, a virtual terminal window's behavior emulates that of a particular "real" terminal. If no particular emulation is requested, a simple "generic" terminal is provided.

The virtual terminal resource creates a picture of the given dimensions to represent the virtual "screen". The "screen" is strictly text-oriented and is organized as lines and characters, as reflected in messages. The virtual screen is displayed in a default window created by the terminal manager.

The following are the various Virtual Terminal requests and the types of information which may be associated with each:

WRITE sends the output to a terminal window, and it may contain information specifying a connector to the virtual terminal; the characters to be written; the data type; and the position on the virtual screen.

READ gets input from a terminal window, and it may contain information specifying a connector to the virtual terminal; an optional prompt string; a parameter to protect typed input (i.e. don't "echo"); continuous read (i.e. automatically re-issue the request at the end of every input line); the maximum number of characters to return; and the position on the virtual screen.

CANCEL terminates outstanding requests from processes, it contains no parameters.

SCROLL shifts a subset of lines up or down (inserts blank lines to fill a gap), and it may contain information specifying a starting and ending line number; and the number of lines to shift.

The following are the various Virtual Terminal responses and the types of information which may be associated with each:

STATUS describes the current state of the terminal, and it may contain information specifying a connector to the terminal; specifying the originator (i.e. the "terminal"); an original message identifier, if applicable; the name of the terminal; the height and width in characters; and the name of the emulator (if any).

WRITE is a response from a virtual terminal "read", and it may contain information specifying the name of the terminal; a connector to the terminal; specifying the originator (i.e. the "terminal"); the characters read, followed by a null character; the data type; and the character position within the terminal's "virtual screen"

### Dialog Service

The dialog service provides representation-independent interaction with a user (as compared with device-independence, which is at a lower level). To a large extent programmers can ignore how prompts, error messages, etc. are

displayed, and how prompts are answered or commands are issued. Thus the visual aspect of the interaction can be tailored to specific applications, users, or devices, independently of the software. For example, requesting a report to be printed may be accomplished by selecting an icon on one system, using a menu on another, and pressing a function-key on a third. The report-printing program would be identical on all three systems.

Dialog comprises five primitive components: menus, prompts, icons, values, and informational boxes. Of these, the first four are primarily for entering data and the last is for telling the user something (e.g. "the printer is out of paper"). They are useable at three different levels.

The least complicated (and also least independent) is exemplified by sending a menu directly to the dialog manager. The dialog manager will construct the appropriate display, then return the item selected by the user. Alternatively, the menu could be placed in a file and activated by sending only the file's name to the dialog manager.

The generalized "click" message is used to indicate that an action has been performed (such as selecting an item from a menu, or selecting an icon).

A "metaphor" defines the visual environment in which the user operates on a particular screen. It consists of any combination of pre-defined windows, icons, menus, and soft-keys appropriate to that environment. In general, a metaphor graphically depicts a real user environment. Thus the icons may represent physical objects in the user's frame of reference, such as file folders or diskettes, menus and messages phrased in familiar terminology, and so on.

The dialog service is most useful for low-volume interaction. For large amounts of data display or input, especially if the data is highly structured, other Human Interface services and tools, or specialized applications programs, would be more appropriate.

All dialog requests are sent directly to the desired console. The picture is always displayed on the screen which the user is using at that moment, and at the most appropriate location (usually the current cursor position). In general, dialog can be referenced indirectly (through a predefined picture in the Human Interface library or a unique file) or can be included explicitly in the request. In the latter case, a default display format is used. The "menu", "prompt", "value", and "dialog" (and "info", if "wait" is specified) are generally expected to be used via the CALL primitive, although they may be used otherwise. The "click" is used by the windowing service.

The following are the various Dialog Requests and the types of information which may be associated with each:

**META** displays initial/new icons and windows, and it may contain information specifying the name of a picture file in the Human Interface directory; the color of the metaphor background; data in a picture; and the name of the picture file which contains the icon, menu, prompt, and information picture elements.

**TITLE** is used to replace elements in the metaphor's title, and it may contain information specifying a list of picture elements (existing elements with matching tags are replaced; replacing an element with a null element effectively deletes it; if omitted all tagged elements are deleted).

**ICON** displays a new icon in the current metaphor, and it may contain information specifying the name of a picture element in the metaphor's current icon library; the identity of the icon on the screen; and a single picture element.

**ERASE** is used to remove an icon, and it may contain information specifying a particular icon (default: all icons).

**MENU** is used to create and display a temporary window containing a menu, and it may contain information specifying the absolute position of the dialog window on the screen; a connector to a window within which to display the menu; the relative position of the window (with respect to the given window, if any, otherwise with respect to the screen; any combination of "centered", "upper", "lower", "left", and "right"); the name of a picture element in the metaphor's current library; the number of items to show in the window; specifying that the given items are to be arranged in a given number of evenly-spaced columns; a list of menu items; specifying highlighting; a name returned in the "click" message to help identify the particular menu selected, if more than one is possible; an alternate format; and an optional window title.

**PROMPT** is used to ask a question and return the answer, and it may contain information specifying absolute position of the dialog window on the screen; a connector to a window within which to display the menu; the relative position of the window (with respect to the given window, if any, otherwise with respect to the screen; the name of a picture element in the metaphor's current library; a question string; the maximum length of a typed response; a list of items any of which can be selected by the user as a response; the maximum width of the text box; a name returned in the "click" message to help identify the particular prompt, if more than one is possible; an alternative format; and a default initial response string.

**INFO** is used to display an informative message, and it may contain information specifying absolute position of the dialog window on the screen; a connector to a window within which to display the menu; the relative position of the window (with respect to the given window, if any, otherwise with respect to the screen; the name of a picture element in the metaphor's current library; the name of a file containing a picture; information to be displayed; specifying to wait for a response; specifying to highlight the window to indicate that the picture corresponds to an error condition; and the maximum width of the text box.

**HIGHLIGHT, INVERT, HIDE, BLINK** are used to change an attribute in an icon (etc.) element, and they may contain information specifying whether the attribute is set/cleared; the type of metaphor element (menu, icon, key, title); and identifying the metaphor element (if omitted, all elements of the given type are affected).

**OPEN\_MENU** is used to define or redefine the current "open" key menu, and it has the same format as the MENU request.

**CANCEL** is used to erase any dialog requested by the sending process, and it may contain information specifying what is to be cancelled (any combination of information, menu, prompt, or value).

The following are the various Dialog responses and the types of information which may be associated with each:

**CLICK** indicates that an action has occurred in the metaphor, and it may contain information specifying the name of the currently active metaphor from its "title" element, if given, or else its file name; what event (e.g. menu, icon, title, function key, prompt,

value, etc.); the name of the menu, picture, etc. (if given); the label assigned to the icon, menu item, etc. in its tag field; a numeric input value; a typed response; the point on the screen where the action occurred; a connector to the associated screen; the console and screen names; a connector to the window or terminal manager, if either was opened automatically; the name of a process to initiate; the name of a process to which to send a message; a message identifier; an optional "argument" descriptor string; and a list of currently selected elements (from all windows), if any.

#### Metaphor

A "metaphor" picture comprises more-or-less arbitrary picture elements which model a particular frame of reference for the user. For example, the picture may represent a "desktop", with appropriate elements (typewriter, letter "in" and "out" trays, pads of paper, etc.). The name of the metaphor must be unique among metaphors.

#### ICONS

Selecting an icon causes the metaphor's owner to be notified via a "click" message. Icons are distinguished from other picture elements by tags which contain the following substrings:

Name—a short string which uniquely labels the icon and identifies it to the applications program; the string will be sent (in the "click" message) when the icon is selected.

P=name of the process to activate

M=name of the process to which to send a message

W=position and size of the default window

A—an arbitrary "arguments" string which is passed to the application "as is"

O—a string of single-character options (open a standard window when the icon is opened; open a terminal window when the icon is opened; repeatable)

T=title

Icons must be the last elements in the metaphor picture, following all others. The arguments string ("A" field in the icon's tag) may be arbitrary.

Tagged elements define interactive components of the metaphor, such as icons, menus, etc. The format of the tag contains information which is interpreted dynamically. Untagged elements cannot be selected and are treated as decoration. The formats of all windows are built-in. The owner of an automatically opened window (using the "W" or "T" options) is the dialog manager. An application must issue a "change" request to the console to acquire ownership of the window.

Although a metaphor is usually designed for a particular screen, it will automatically be adjusted to fit any console on which it is displayed.

#### TITLE

An element tagged "TITLE=metaphor-name" may optionally be included in the picture. The element will occupy the entire top line of the screen. If the element is a macro, all sub-elements in the macro are displayed in the line. Sub-elements must be individually tagged if the title line will be dynamically altered via a "title" request.

Sub-elements tagged "DATE" or "TIME" will automatically display the current date or time. The elements must be "text" and must be large enough to contain the dynamic strings. The data minimally consists of the month and day; if the string is 10 characters or longer, the day of the week will also be displayed.

#### POP-UP MENUS

Up to 9 elements in the picture may be tagged "MENU=name; n", where "name" identifies a menu in the Human

Interface library and "n" indicates which menu key on the keyboard can be used to "pop-up" the menu. "n" may also be a name, indicating that the menu can only be referenced indirectly (via a request or through the nested sub-menu option). Both may also be given, as in "MENU— . . . ; 1; edit"

The name is returned in the "click" message to help distinguish the selection. Normally, menu elements are defined as null (type "n") picture elements. If not (i.e. the element is visible on the screen), the menu will also be displayed any time the element is selected

An in-line, predefined menu can be set up by replacing the name with a list of explicit menu items, for example: "MENU=copy, cut, paste; 1". One element tagged "OPEN=name" (or "OPEN=list") may be included in the picture to associate a menu with the Human Interface "open" function-key. If such an element is not defined, pressing "open" will cause an "Open" message (containing a "position" field specifying the cursor row and column) to be sent to the owner of the metaphor.

#### SOFT-KEYS

One element in the picture may be tagged "KEYS=name", where "name" identifies a menu in the Human Interface library. Each item in the menu will be displayed as a "soft key" An in-line, predefined set of keys can be set up by replacing the name with a list of explicit items, for example: "KEYS=open, close, quit". A "name" may be given to the set of keys by appending "; name", e.g. "KEYS= . . . ; name". The name is returned in "click" messages to help identify the response.

The soft-key element is usually a "rectangle" which defines the area of the screen reserved for display of the keys. The element type can also be "n" (null) in which case the keys will not be displayed. The actual number of keys which can be displayed is limited only by the physical size of the screen in use at the time the metaphor is displayed.

The soft-key area is aligned along the appropriate edge of the screen when the metaphor is activated. Selecting a soft-key on the screen is equivalent to selecting the corresponding item from a menu.

#### LIBRARIES

Menus (as well as icons, prompts, and information) can be stored in "libraries" to which the metaphor may be linked when it is built or when it is initiated. A library consists of individual elements, each of which represents one menu, icon, etc. The first substring of the element's "tag" field is the element's name. The "name" is referenced in the corresponding dialog request ("icon", etc.) or response ("click").

An icon is usually a single element. Menus, prompts, and information are generally composites and must each be stored as a distinct macro element in the library picture.

Library references can be built into a metaphor picture (as opposed to being specified in the "meta" message) by including a null picture element tagged "LIB=picture" "Picture" is the name of a file containing the library picture.

#### MENU

A "menu" picture comprises two or more menu "items", each of which is simply a picture element, usually of type "text" although there are no restrictions on pictorial menu.

Each item in the menu is described by a simple element, usually text or a symbol. The element is tagged with a string which is to be sent to the application process when that item is selected from the menu. For example, in a menu of colors, blocks in the actual colors might be displayed but the tags could be "red", "blue", etc.

If the menu item is a text string ending in ". . .", the text (excluding the ellipsis) is assumed to refer to another menu in the Human Interface library. When the item is selected, the referenced menu is automatically brought up. That menu may itself contain further menu references, allowing chaining to any arbitrary depth. Only the final selection is returned to the process.



Preceding an item with "+" indicates that the item is currently "active" and causes a check mark to be displayed beside it whenever the menu is opened. Preceding an item with "-" indicates that the corresponding option is not currently available and cannot be selected.

An "arguments" string can be appended to the tag of an element in the menu. The string is passed "as is" to the application when the item is selected.

**PROMPT**

The greater part of a prompt picture comprises text which asks a question, often with some introductory preamble. One element, located anywhere in the picture, may represent a response area. This is generally a rectangular area into which a user can type the information requested by the prompt. This element must be tagged "RESP".

Two further elements, tagged "ENTER" and "CANCEL", display target text or symbols which are used to complete the prompt. When the "enter" element is selected by the user, the text typed in the response area is returned to the originator of the prompt.

If the "cancel" element is selected instead, the prompt is cancelled with a null response. The response element is optional. If omitted, the "enter" and "cancel" elements effectively correspond to "yes" or "no" responses. Typing a "carriage return" character will have the same effect as selecting "enter". The prompt is erased when any response is given, or by an explicit "cancel" request.

**INFORMATION**

An information picture comprises text (and possibly graphics) which describes something. One element, located anywhere in the picture, is usually tagged "DONE". When this element is selected, the information picture is erased from the display. If no such element is given, the process which requested the information to be displayed must send an explicit "cancel" request when it wants to get rid of it.

**INPUT/OUTPUT DEVICE INDEPENDENCE**

In the present invention all system interaction with the outside world is either through "virtual input" or "virtual output" devices. The system can accept any form of input or output device. The Human Interface is constructed using a well-defined set of "virtual devices". All Human Interface functions (windowing, picture—drawing, dialog management, etc.) use this set of devices exclusively.

These virtual input devices take the form of "keys" (typed textual input); "position" (screen coordinates); "actions" (Human Interface functions such as "open window", etc.); "functions" (user-defined actions); and "means" (pop-up lists of choices).

Virtual output devices produce device-independent output: text, lines, rectangles, polygons, circles, ellipses, discrete points, bit-mapped symbols, and bit-mapped arrays.

FIG. 12 shows how the console manager operates upon virtual input to generate virtual output. The lowest layer of HI software converts input from any "real" physical devices to the generic, virtual form, and it converts Human Interface output (in virtual form) to physical output.

FIG. 12 shows the central process of the Human Interface, the console manager 220, dealing with virtual input and producing virtual output. Virtual input passed through the virtual input manager 221 is processed directly by the console manager 220, while output is passed through two intermediate processes—(1) a picture manager 222, which manipulates device-independent graphical images, and (2) a window manager 224, which presents a subset (called a "view") of the overall picture to the virtual output manager 226.

Any number of physical devices can be connected to the Human Interface and can be removed or replaced dynami-

cally, without disturbing the current state of the Human Interface or of any applications using the Human Interface. In other words, the Human Interface is independent of particular I/O devices, and the idiosyncracies of the devices are hidden from the Human Interface.

FIG. 13 represents a flowchart showing how virtual input is handled by the console manager. The virtual input may take any of several forms, such as a keystroke, cursor position, action, function key, menu, etc.

For example, regarding the operations beneath block 231, if the virtual input to the console manager is a keystroke, then the console manager checks to see whether the cursor is inside a window. If so, it checks to see whether it originated from a virtual terminal, and if not it checks to see whether an edit operation is taking place. If not, it updates the picture.

Regarding the operations beneath block 232, if the virtual input represents a cursor position, then the console manager checks to see whether the auto-highlight option has been enabled. If yes, it checks to see whether the cursor is on an element. If so it highlights that element.

Regarding the operations beneath block 233, the console manager uses any of the indicated actions to update a picture, update a window, or initiate dialog, as appropriate.

Regarding the operations beneath block 234, if the virtual input is from a function key, the console manager notifies the dialog manager.

Regarding the operations beneath block 235, if the virtual input represents a menu choice, the console manager checks to see whether the cursor is in a window. If not, it determines that it is on a user metaphor; if so, it requests a menu from the window. If the menu is defined, it notifies the owner of the window (or metaphor), activates a pop-up menu, gets a response, and sends the response to the window owner.

FIG. 14 represents a flowchart showing how virtual output is handled by the picture manager. The picture manager 240 accepts virtual output from the console manager and then, depending upon the type of operation, performs the requested function. For example, if the operation is a replace operation, the picture manager 240 replaces the old output with the new and sends the change(s) to the window manager. The window manager sends the change to the output manager, which in turn sends it to the real device.

**DESCRIPTION OF SOURCE CODE LISTING**

Program Listings A and B contain a "C" language implementation of the above-described concepts relating to input/output device independence. The following chart indicates where the relevant portions of the listing may be found.

Function	Lines Numbers in Program Listing A	Lines Numbers in Program Listing B
Main-line; initialization; accept input	190-222	125-141
Determine type of input	486-521	161-203
Virtual key	523-631	
Virtual position	633-661	
Virtual action	663-702, 763-1200	
Virtual function	704-723	
Virtual menu	725-761	
Main-line; initialization; start processing		125-141
Accept requests (virtual output); check for changes		161-203
Determine type of request		239-310

45

-continued

Function	
Draw	410-457
Copy	611-632
Replace	537-585
Erase	587-609
Move	634-678
Send changes	1265-1352

46

It will be apparent to those skilled in the art that the herein disclosed invention may be modified in numerous ways and may assume many embodiments other than the preferred form specifically set out and described above. For example, it may be implemented in other than a distributed data processing system.

Accordingly, it is intended by the appended claims to cover all modifications of the invention which fall within the true spirit and scope of the invention.

## PROGRAM LISTING A

```

9  Module submitted      : %M% %I%
10 Date submitted       : %E% %U%
11 Author               : Frank Kolnick
12 Origin                : CX
13 Description          : Console Manager
14 *****
15 #ifndef lint
16 static char srcid[] = "%Z% %M%:%I%";
17 #endif
18 /* Console manager: global data */
19
20 #include <CX.h>
21 #include <HI.h>
22 #include <memory.h>
23 #include <string.h>
24 #include <gen_codes.h>
25 static long none[2] = {0,0};
26
27 #define MIN_HT {1*VCHAR_HT}
28 #define MIN_WD {5*VCHAR_WD}
29 #define POOL_SIZE 10
30 #define activate(node) if (!node->never) map->active = node
31
32 typedef struct names
33 {
34     char console[32];
35     char class[32];
36     char screen[32];
37     char user[64];
38     char metaphor[32];
39     NAME;
40 }
41
42 typedef struct editstat
43 {
44     type of structure[16]:
45     (identifies struct.) */
46     console's name */
47     console's class */
48     screen's name */
49     user's name */
50     preferred metaphor */
51     /* editing status: */

```

```

*****

```

```

/* CX definitions */
/* picture, etc. definitions */

```

```

/* minimum window height */
/* minimum window width */
/* #nodes in local pool */
/* active = node

```

```

/* name of console, etc.: */

```

```

/* editing status: */

```



```

93  ... end of box */
94  ... cursor location */
95  ... new element */
96  auto highlight */
97  edit structure */
98  multi-select */
99  remap at window edge */
100 non-moveable */
101 immovable */
102 user can't close */
103 title (etc) bar... */
104 heights/widths... */
105 ... */
106 move mark on 'select' */
107 special chars. */
108 end-of-input chars. */
109 ->editing descriptor */
110
111 /*
112 /*
113 /*
114 /*
115 /*
116 /*
117 /*
118 /*
119 /*
120 /*
121 /*
122 /*
123 /*
124 /*
125 /*
126 /*
127 /*
128 /*
129 /*
130 /*
131 /*
132 /*
133 /*
134 /*
135 /*
136 /*
137 /*
138 /*
139 /*
140 /*
141 /*
142 /*
143 /*
144 /*
145 /*
146 /*
147 /*
148 /*
149 /*
150 /*
151 /*
152 /*
153 /*
154 /*
155 /*
156 /*
157 /*
158 /*
159 /*
160 /*
161 /*
162 /*
163 /*
164 /*
165 /*
166 /*
167 /*
168 /*
169 /*
170 /*
171 /*
172 /*
173 /*
174 /*
175 /*
176 /*
177 /*
178 /*
179 /*
180 /*
181 /*
182 /*
183 /*
184 /*
185 /*
186 /*
187 /*
188 /*
189 /*
190 /*
191 /*
192 /*
193 /*
194 /*
195 /*
196 /*
197 /*
198 /*
199 /*
200 /*
201 /*
202 /*
203 /*
204 /*
205 /*
206 /*
207 /*
208 /*
209 /*
210 /*
211 /*
212 /*
213 /*
214 /*
215 /*
216 /*
217 /*
218 /*
219 /*
220 /*
221 /*
222 /*
223 /*
224 /*
225 /*
226 /*
227 /*
228 /*
229 /*
230 /*
231 /*
232 /*
233 /*
234 /*
235 /*
236 /*
237 /*
238 /*
239 /*
240 /*
241 /*
242 /*
243 /*
244 /*
245 /*
246 /*
247 /*
248 /*
249 /*
250 /*
251 /*
252 /*
253 /*
254 /*
255 /*
256 /*
257 /*
258 /*
259 /*
260 /*
261 /*
262 /*
263 /*
264 /*
265 /*
266 /*
267 /*
268 /*
269 /*
270 /*
271 /*
272 /*
273 /*
274 /*
275 /*
276 /*
277 /*
278 /*
279 /*
280 /*
281 /*
282 /*
283 /*
284 /*
285 /*
286 /*
287 /*
288 /*
289 /*
290 /*
291 /*
292 /*
293 /*
294 /*
295 /*
296 /*
297 /*
298 /*
299 /*
300 /*
301 /*
302 /*
303 /*
304 /*
305 /*
306 /*
307 /*
308 /*
309 /*
310 /*
311 /*
312 /*
313 /*
314 /*
315 /*
316 /*
317 /*
318 /*
319 /*
320 /*
321 /*
322 /*
323 /*
324 /*
325 /*
326 /*
327 /*
328 /*
329 /*
330 /*
331 /*
332 /*
333 /*
334 /*
335 /*
336 /*
337 /*
338 /*
339 /*
340 /*
341 /*
342 /*
343 /*
344 /*
345 /*
346 /*
347 /*
348 /*
349 /*
350 /*
351 /*
352 /*
353 /*
354 /*
355 /*
356 /*
357 /*
358 /*
359 /*
360 /*
361 /*
362 /*
363 /*
364 /*
365 /*
366 /*
367 /*
368 /*
369 /*
370 /*
371 /*
372 /*
373 /*
374 /*
375 /*
376 /*
377 /*
378 /*
379 /*
380 /*
381 /*
382 /*
383 /*
384 /*
385 /*
386 /*
387 /*
388 /*
389 /*
390 /*
391 /*
392 /*
393 /*
394 /*
395 /*
396 /*
397 /*
398 /*
399 /*
400 /*
401 /*
402 /*
403 /*
404 /*
405 /*
406 /*
407 /*
408 /*
409 /*
410 /*
411 /*
412 /*
413 /*
414 /*
415 /*
416 /*
417 /*
418 /*
419 /*
420 /*
421 /*
422 /*
423 /*
424 /*
425 /*
426 /*
427 /*
428 /*
429 /*
430 /*
431 /*
432 /*
433 /*
434 /*
435 /*
436 /*
437 /*
438 /*
439 /*
440 /*
441 /*
442 /*
443 /*
444 /*
445 /*
446 /*
447 /*
448 /*
449 /*
450 /*
451 /*
452 /*
453 /*
454 /*
455 /*
456 /*
457 /*
458 /*
459 /*
460 /*
461 /*
462 /*
463 /*
464 /*
465 /*
466 /*
467 /*
468 /*
469 /*
470 /*
471 /*
472 /*
473 /*
474 /*
475 /*
476 /*
477 /*
478 /*
479 /*
480 /*
481 /*
482 /*
483 /*
484 /*
485 /*
486 /*
487 /*
488 /*
489 /*
490 /*
491 /*
492 /*
493 /*
494 /*
495 /*
496 /*
497 /*
498 /*
499 /*
500 /*
501 /*
502 /*
503 /*
504 /*
505 /*
506 /*
507 /*
508 /*
509 /*
510 /*
511 /*
512 /*
513 /*
514 /*
515 /*
516 /*
517 /*
518 /*
519 /*
520 /*
521 /*
522 /*
523 /*
524 /*
525 /*
526 /*
527 /*
528 /*
529 /*
530 /*
531 /*
532 /*
533 /*
534 /*
535 /*
536 /*
537 /*
538 /*
539 /*
540 /*
541 /*
542 /*
543 /*
544 /*
545 /*
546 /*
547 /*
548 /*
549 /*
550 /*
551 /*
552 /*
553 /*
554 /*
555 /*
556 /*
557 /*
558 /*
559 /*
560 /*
561 /*
562 /*
563 /*
564 /*
565 /*
566 /*
567 /*
568 /*
569 /*
570 /*
571 /*
572 /*
573 /*
574 /*
575 /*
576 /*
577 /*
578 /*
579 /*
580 /*
581 /*
582 /*
583 /*
584 /*
585 /*
586 /*
587 /*
588 /*
589 /*
590 /*
591 /*
592 /*
593 /*
594 /*
595 /*
596 /*
597 /*
598 /*
599 /*
600 /*
601 /*
602 /*
603 /*
604 /*
605 /*
606 /*
607 /*
608 /*
609 /*
610 /*
611 /*
612 /*
613 /*
614 /*
615 /*
616 /*
617 /*
618 /*
619 /*
620 /*
621 /*
622 /*
623 /*
624 /*
625 /*
626 /*
627 /*
628 /*
629 /*
630 /*
631 /*
632 /*
633 /*
634 /*
635 /*
636 /*
637 /*
638 /*
639 /*
640 /*
641 /*
642 /*
643 /*
644 /*
645 /*
646 /*
647 /*
648 /*
649 /*
650 /*
651 /*
652 /*
653 /*
654 /*
655 /*
656 /*
657 /*
658 /*
659 /*
660 /*
661 /*
662 /*
663 /*
664 /*
665 /*
666 /*
667 /*
668 /*
669 /*
670 /*
671 /*
672 /*
673 /*
674 /*
675 /*
676 /*
677 /*
678 /*
679 /*
680 /*
681 /*
682 /*
683 /*
684 /*
685 /*
686 /*
687 /*
688 /*
689 /*
690 /*
691 /*
692 /*
693 /*
694 /*
695 /*
696 /*
697 /*
698 /*
699 /*
700 /*
701 /*
702 /*
703 /*
704 /*
705 /*
706 /*
707 /*
708 /*
709 /*
710 /*
711 /*
712 /*
713 /*
714 /*
715 /*
716 /*
717 /*
718 /*
719 /*
720 /*
721 /*
722 /*
723 /*
724 /*
725 /*
726 /*
727 /*
728 /*
729 /*
730 /*
731 /*
732 /*
733 /*
734 /*
735 /*
736 /*
737 /*
738 /*
739 /*
740 /*
741 /*
742 /*
743 /*
744 /*
745 /*
746 /*
747 /*
748 /*
749 /*
750 /*
751 /*
752 /*
753 /*
754 /*
755 /*
756 /*
757 /*
758 /*
759 /*
760 /*
761 /*
762 /*
763 /*
764 /*
765 /*
766 /*
767 /*
768 /*
769 /*
770 /*
771 /*
772 /*
773 /*
774 /*
775 /*
776 /*
777 /*
778 /*
779 /*
780 /*
781 /*
782 /*
783 /*
784 /*
785 /*
786 /*
787 /*
788 /*
789 /*
790 /*
791 /*
792 /*
793 /*
794 /*
795 /*
796 /*
797 /*
798 /*
799 /*
800 /*
801 /*
802 /*
803 /*
804 /*
805 /*
806 /*
807 /*
808 /*
809 /*
810 /*
811 /*
812 /*
813 /*
814 /*
815 /*
816 /*
817 /*
818 /*
819 /*
820 /*
821 /*
822 /*
823 /*
824 /*
825 /*
826 /*
827 /*
828 /*
829 /*
830 /*
831 /*
832 /*
833 /*
834 /*
835 /*
836 /*
837 /*
838 /*
839 /*
840 /*
841 /*
842 /*
843 /*
844 /*
845 /*
846 /*
847 /*
848 /*
849 /*
850 /*
851 /*
852 /*
853 /*
854 /*
855 /*
856 /*
857 /*
858 /*
859 /*
860 /*
861 /*
862 /*
863 /*
864 /*
865 /*
866 /*
867 /*
868 /*
869 /*
870 /*
871 /*
872 /*
873 /*
874 /*
875 /*
876 /*
877 /*
878 /*
879 /*
880 /*
881 /*
882 /*
883 /*
884 /*
885 /*
886 /*
887 /*
888 /*
889 /*
890 /*
891 /*
892 /*
893 /*
894 /*
895 /*
896 /*
897 /*
898 /*
899 /*
900 /*
901 /*
902 /*
903 /*
904 /*
905 /*
906 /*
907 /*
908 /*
909 /*
910 /*
911 /*
912 /*
913 /*
914 /*
915 /*
916 /*
917 /*
918 /*
919 /*
920 /*
921 /*
922 /*
923 /*
924 /*
925 /*
926 /*
927 /*
928 /*
929 /*
930 /*
931 /*
932 /*
933 /*
934 /*
935 /*
936 /*
937 /*
938 /*
939 /*
940 /*
941 /*
942 /*
943 /*
944 /*
945 /*
946 /*
947 /*
948 /*
949 /*
950 /*
951 /*
952 /*
953 /*
954 /*
955 /*
956 /*
957 /*
958 /*
959 /*
960 /*
961 /*
962 /*
963 /*
964 /*
965 /*
966 /*
967 /*
968 /*
969 /*
970 /*
971 /*
972 /*
973 /*
974 /*
975 /*
976 /*
977 /*
978 /*
979 /*
980 /*
981 /*
982 /*
983 /*
984 /*
985 /*
986 /*
987 /*
988 /*
989 /*
990 /*
991 /*
992 /*
993 /*
994 /*
995 /*
996 /*
997 /*
998 /*
999 /*
1000 /*

```

```

142 typedef struct selstat
143 {
144     char          unsigned char
145     char          pending;
146     short        area, col;
147     MAPNODE     *map;
148     SELECTION;
149 }
150
151 typedef struct cur_message
152 {
153     type_of_structure[16];
154     *buf;
155     char          sender;
156     long         size;
157     MESSAGE;
158 }
159
160 typedef struct process_ids
161 {
162     type_of_structure[16];
163     output;
164     input;
165     dialogue;
166     self;
167     owner;
168     CONNS;
169 }
170
171 typedef struct lists
172 {
173     char          *pool;
174     MAPNODE     count;
175     MAPNODE     *active;
176     MAPNODE     *first;
177     MAPNODE     *last;
178     MAPNODE     *active;
179     MAPNODE     *metaphor;
180     LIST;
181 }
182
183 /* Local functions: */
184 MAPNODE *find_window(), *create_window(), *create_terminal();
185 long NewProc();
186
187

```

```

/* selection status: */
/* (identifies struct.) */
/* select in progress */
/* original window area */
/* orig. pos'n in window */
/* ->original map node */

/* current message: */
/* (identifies struct.) */
/* ->msg. buffer */
/* conn. to sender */
/* size of msg. */

/* identifies key processes: */
/* (identifies struct.) */
/* Output Manager */
/* Input Manager */
/* Dialog Manager */
/* this process */
/* initializing process */

/* list pointers, etc.: */
/* (identifies struct.) */
/* ->buffer pool */
/* current #window nodes */
/* ->active node, if any */
/* ->start of list */
/* ->end of list */
/* ->prev. active node */
/* ->metaphor node */

```

```

1889 /* Console manager: main-line */
1890 PROCESS(Console)
1891 {
1892     NAME
1893     SCREEN
1894     LIST_SELECTION
1895     WINDOW
1896     MESSAGE
1897     CONNS
1898     register LIST
1899     register MESSAGE
2000     register CONNS
2001     register short
2002     long
2003     list_size = 0, *req = NULL;
2004
2005     set_event_key("Console mgr.");
2006     init_CM(&name, &screen, &map_ptr, &sel, &window, &msg_ptr, &conn_ptr);
2007     map = map_ptr;
2008     msg = msg_ptr;
2009     conn = conn_ptr;
2010     start_up(name, screen, conn);
2011     while(!go)
2012     {
2013         msg->buf = Get(0, &msg->sender, &msg->size);
2014         if (!*(msg->buf+1))
2015             input(screen, map, sel, window, msg, conn, *msg->buf);
2016         else
2017             request(name, screen, map, sel, msg, conn, msg->buf, msg->size);
2018         highlight(map->active_map);
2019         free_requests(msg->buf, msg->size, &req, &list_size);
2020     }
2021     Exit();
2022 }

```

```

223 free requests(msg_size, req, list_size)
224 register Char *msg, **req;
225 register long size, *list_size;
226 {
227     register char *temp, *next;
228     if (msg)
229     {
230         *(char**)msg = *req;
231         *req = msg;
232         *list_size += size;
233         if (!Any msg(NULL) || *list_size > 1000)
234             for (temp = *req, *req = NULL, *list_size = 0; temp; temp = next)
235                 next = *(char**)temp;
236                 Free(temp);
237     }
238 }
239
240
241
242

```



247  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276

```

init CM(name, screen, map, sel, window, msg, conn)
register NAME
register SCREEN
register LIST
register SELECTION
register WINDOW
register MESSAGE
register CONNS
{
    *name = (NAME *) Alloc(sizeof(NAME), YES);
    *screen = (SCREEN *) Alloc(sizeof(SCREEN), YES);
    *map = (LIST *) Alloc(sizeof(LIST), YES);
    *sel = (SELECTION *) Alloc(sizeof(SELECTION), YES);
    *window = (WINDOW *) Alloc(sizeof(WINDOW), YES);
    *msg = (MESSAGE *) Alloc(sizeof(MESSAGE), YES);
    *conn = (CONNS *) Alloc(sizeof(CONNS), YES);
    memset(*name, 0, sizeof(*name));
    memset(*screen, 0, sizeof(*screen));
    memset(*map, 0, sizeof(*map));
    memset(*sel, 0, sizeof(*sel));
    memset(*window, 0, sizeof(*window));
    memset(*msg, 0, sizeof(*msg));
    memset(*conn, 0, sizeof(*conn));
    (*map) -> pool = (MAPNODE *) Alloc(sizeof(MAPNODE), YES);
    memset((*map) -> pool, 0, POOL_SIZE*sizeof(MAPNODE));
}

```

```

277 start up(name, screen, conn)
278 register NAME *name;
279 register SCREEN *screen;
280 register CONNS *conn;
281 {
282     register char *msg;
283     CONNECTOR conf;
284     short *p;
285     long size;
286     while ((msg = Get(0, &conn->owner, &size)) && strcmp(msg, "init"))
287     {
288         reply status(msg, msg, "not ready", 0);
289         Free(msg);
290     }
291     strcpy(name->console, Find_triple(msg, "name", size, none, 2, NULL));
292     conn->self = *(CONNECTOR *) Find_triple(msg, "self", size, none, 4, NULL);
293     Free(msg);
294     if (conf.pid = NewProc("CMconfig", "//processes/CMconfig", YES, -1))
295     {
296         Put(DIRECT, conf.pid, Newmsg(32, "I", NULL));
297         while (!Any_msg(conf.pid))
298             if (Forward(DIRECT, conf.pid, Get(conn->owner.pid, 0, 0)));
299         else Free(Call(NEXT, "Clock",
300                     Newmsg(64, 0, &size), "after=#5s", 0, 0, 5, 0), 0, 0));
301     }
302     msg = Get(conf.pid, 0, &size);
303     conn->input = *(CONNECTOR *) Find_triple(msg, "inp", size, none, 4, NULL);
304     conn->output = *(CONNECTOR *) Find_triple(msg, "outp", size, none, 4, NULL);
305     conn->dialogue = *(CONNECTOR *) Find_triple(msg, "dial", size, none, 4, NULL);
306     Free(msg);
307     if (msg == Call(DIRECT, conn->output.pid, Newmsg(32, "query", NULL), 0, &size))
308     {
309         p = (short *) Find_triple(msg, "scrn", size, none, 4, NULL);
310         screen->meta_ht = screen->height = *p++;
311         screen->meta_wd = screen->width = *p++;
312         screen->char_gen = screen->char_align =
313             (char) Find_triple(msg, "char", size, NO, 0, NULL);
314         screen->colors = *(short *) Find_triple(msg, "clrs", size, none, 2, NULL);
315         screen->bit_map = (char) Find_triple(msg, "bmap", size, NO, 0, NULL);
316         screen->fonts = (char) Find_triple(msg, "font", size, NO, 0, NULL);
317         Free(msg);
318     }
319     else Note("'query' to output mgr. failed", msg);
320     Put(DIRECT, conn->owner.pid,
321         Newmsg(128, "ready", "serv=#S; name=#S", "console", name->console));
322 }
323
324
325
326
327

```

```

3299 request(name, screen, map, sel, msg, conn, buf, size)
3300 REGISTER NAME *name;
3301 SCREEN *screen;
3302 REGISTER LIST *map;
3303 SELECTION MESSAGE *sel;
3304 REGISTER CONNS *msg;
3305 REGISTER LONG buf, size;
3306
3307 if (!strcmp(buf, "create"))
3308     CreateResource(screen, map, buf, size, &conn->output, &msg->sender);
3309 else if (!strcmp(buf, "write"))
3310     WriteElement(selected(map, sel, msg));
3311 else if (!strcmp(buf, "delete"))
3312     DeleteResource(map, msg, conn, sel);
3313 else if (!strcmp(buf, "Meta"))
3314     MetaPhor(screen, map, buf, size, &conn->output, &conn->dialogue);
3315 else if (!strcmp(buf, "user"))
3316     SetUser(name, buf, size);
3317 else if (!strcmp(buf, "resource"))
3318     Query(name, screen, map, msg, conn);
3319 else if (!strcmp(buf, "change"))
3320     Change(screen, map, msg);
3321 else if (!strcmp(buf, "remapped"))
3322     Remap(&msg->sender, NULL, FindTriple(buf, "conn", 0, 0, 8, 0), sel, map);
3323 else if (!strcmp(buf, "failed"))
3324     Status(buf, size);
3325 else if (!strcmp(buf, "done") || !strcmp(buf, "status"))
3326     if (conn->dialogue.pid)
3327     {
3328         buf = (long) Realloc(buf, size+20, YES);
3329         AppendTriple(buf, "Cpos", 4, &screen->row);
3330         Forward(DIRECT, conn->dialogue.pid, buf);
3331         msg->buf = NULL;
3332     }
3333     else reply_status(buf, buf, "unknown msg id", 0);
3334 }

```

```

370 Query(name, screen, map, msg, conn)
371 NAME *name;
372 SCREEN *screen;
373 LIST *map;
374 MESSAGE *msg;
375 CONNS *conn;
376 {
377     static char
378     register char
379     register MAPNODE
380     CONNECTOR
381     {
382         def res[] = "console";
383         *window_name; *resource, *p;
384         *node = -NULL;
385         *res;
386         resource = Find_triple(msg->buf, "res ", msg->size, def_res, 2, NULL);
387         if (!strcmp(resource, "console"))
388             Reply(msg->buf, Newmsg(500, "console",
389                 "name=#S; user=#S; clis=#S; conn=#C; orig=#S"
390                 name->console, name->user, screen->colors, &conn->self, "console"));
391         else
392             if (window_name = Find_triple(msg->buf, "name", msg->size, NULL, 2, NULL))
393             {
394                 if (! (p = strchr(window_name, '/'))
395                     || p == window_name;
396                     for (node = map->first;
397                         node && strcmp(p, node->name); node = node->nxt) ;
398                     )
399                     else if (res = (CONNECTOR*) Find_triple(msg->buf, "conn", 0, NULL, 1, NULL))
400                     for (
401                         && node->map->first; node->node->window.pid != res->pid
402                         && node->picture.pid != res->pid
403                         && node->terminal.pid != res->pid; node = node->nxt) ;
404                     )
405                     else
406                     reply_status(msg->buf, "-query", "missing name/connector", 0);
407                     if (node)
408                     {
409                         if (!strcmp(resource, "window"))
410                             Forward(DIRECT, node->window, pid, msg->buf);
411                         else if (!strcmp(resource, "terminal"))
412                             Forward(DIRECT, node->terminal, pid, msg->buf);
413                         else if (!strcmp(resource, "picture"))
414                             Forward(DIRECT, node->picture, pid, msg->buf);
415                         else
416                             Free(msg->buf);
417                     }
418                 }
419             }
420             }
421             }
422             }
423             }
424             }
425             }
426             }
427             }
428             }
429             }
430             }
431             }
432             }
433             }
434             }
435             }
436             }
437             }
438             }
439             }
440             }
441             }
442             }
443             }
444             }
445             }
446             }
447             }
448             }
449             }
450             }
451             }
452             }
453             }
454             }
455             }
456             }
457             }
458             }
459             }
460             }
461             }
462             }
463             }
464             }
465             }
466             }
467             }
468             }
469             }
470             }
471             }
472             }
473             }
474             }
475             }
476             }
477             }
478             }
479             }
480             }
481             }
482             }
483             }
484             }
485             }
486             }
487             }
488             }
489             }
490             }
491             }
492             }
493             }
494             }
495             }
496             }
497             }
498             }
499             }
500             }
501             }
502             }
503             }
504             }
505             }
506             }
507             }
508             }
509             }
510             }
511             }
512             }
513             }
514             }
515             }
516             }
517             }
518             }
519             }
520             }
521             }
522             }
523             }
524             }
525             }
526             }
527             }
528             }
529             }
530             }
531             }
532             }
533             }
534             }
535             }
536             }
537             }
538             }
539             }
540             }
541             }
542             }
543             }
544             }
545             }
546             }
547             }
548             }
549             }
550             }
551             }
552             }
553             }
554             }
555             }
556             }
557             }
558             }
559             }
560             }
561             }
562             }
563             }
564             }
565             }
566             }
567             }
568             }
569             }
570             }
571             }
572             }
573             }
574             }
575             }
576             }
577             }
578             }
579             }
580             }
581             }
582             }
583             }
584             }
585             }
586             }
587             }
588             }
589             }
590             }
591             }
592             }
593             }
594             }
595             }
596             }
597             }
598             }
599             }
600             }
601             }
602             }
603             }
604             }
605             }
606             }
607             }
608             }
609             }
610             }
611             }
612             }
613             }
614             }
615             }
616             }
617             }
618             }
619             }
620             }
621             }
622             }
623             }
624             }
625             }
626             }
627             }
628             }
629             }
630             }
631             }
632             }
633             }
634             }
635             }
636             }
637             }
638             }
639             }
640             }
641             }
642             }
643             }
644             }
645             }
646             }
647             }
648             }
649             }
650             }
651             }
652             }
653             }
654             }
655             }
656             }
657             }
658             }
659             }
660             }
661             }
662             }
663             }
664             }
665             }
666             }
667             }
668             }
669             }
670             }
671             }
672             }
673             }
674             }
675             }
676             }
677             }
678             }
679             }
680             }
681             }
682             }
683             }
684             }
685             }
686             }
687             }
688             }
689             }
690             }
691             }
692             }
693             }
694             }
695             }
696             }
697             }
698             }
699             }
700             }
701             }
702             }
703             }
704             }
705             }
706             }
707             }
708             }
709             }
710             }
711             }
712             }
713             }
714             }
715             }
716             }
717             }
718             }
719             }
720             }
721             }
722             }
723             }
724             }
725             }
726             }
727             }
728             }
729             }
730             }
731             }
732             }
733             }
734             }
735             }
736             }
737             }
738             }
739             }
740             }
741             }
742             }
743             }
744             }
745             }
746             }
747             }
748             }
749             }
750             }
751             }
752             }
753             }
754             }
755             }
756             }
757             }
758             }
759             }
760             }
761             }
762             }
763             }
764             }
765             }
766             }
767             }
768             }
769             }
770             }
771             }
772             }
773             }
774             }
775             }
776             }
777             }
778             }
779             }
780             }
781             }
782             }
783             }
784             }
785             }
786             }
787             }
788             }
789             }
790             }
791             }
792             }
793             }
794             }
795             }
796             }
797             }
798             }
799             }
800             }
801             }
802             }
803             }
804             }
805             }
806             }
807             }
808             }
809             }
810             }
811             }
812             }
813             }
814             }
815             }
816             }
817             }
818             }
819             }
820             }
821             }
822             }
823             }
824             }
825             }
826             }
827             }
828             }
829             }
830             }
831             }
832             }
833             }
834             }
835             }
836             }
837             }
838             }
839             }
840             }
841             }
842             }
843             }
844             }
845             }
846             }
847             }
848             }
849             }
850             }
851             }
852             }
853             }
854             }
855             }
856             }
857             }
858             }
859             }
860             }
861             }
862             }
863             }
864             }
865             }
866             }
867             }
868             }
869             }
870             }
871             }
872             }
873             }
874             }
875             }
876             }
877             }
878             }
879             }
880             }
881             }
882             }
883             }
884             }
885             }
886             }
887             }
888             }
889             }
890             }
891             }
892             }
893             }
894             }
895             }
896             }
897             }
898             }
899             }
900             }
901             }
902             }
903             }
904             }
905             }
906             }
907             }
908             }
909             }
910             }
911             }
912             }
913             }
914             }
915             }
916             }
917             }
918             }
919             }
920             }
921             }
922             }
923             }
924             }
925             }
926             }
927             }
928             }
929             }
930             }
931             }
932             }
933             }
934             }
935             }
936             }
937             }
938             }
939             }
940             }
941             }
942             }
943             }
944             }
945             }
946             }
947             }
948             }
949             }
950             }
951             }
952             }
953             }
954             }
955             }
956             }
957             }
958             }
959             }
960             }
961             }
962             }
963             }
964             }
965             }
966             }
967             }
968             }
969             }
970             }
971             }
972             }
973             }
974             }
975             }
976             }
977             }
978             }
979             }
980             }
981             }
982             }
983             }
984             }
985             }
986             }
987             }
988             }
989             }
990             }
991             }
992             }
993             }
994             }
995             }
996             }
997             }
998             }
999             }
1000            }

```

```

417 Create_resource(screen,map,buf,size,output,sender)
418 SCREEN
419 *map;
420 CONNECTOR *output,*sender;
421 register long buf,size;
422 {
423     static char def_res[] = "window";
424     register char *resource,*p;
425     register MAPNODE *node = NULL;
426     register CONNECTOR *conn = NULL;
427     CONNECTOR picture;
428
429     resource = Find_triple(buf,"res",size,def_res,2,NULL);
430     if (!strcmp(resource,"window"))
431     {
432         node = create_window(screen,map,output,buf,size);
433         conn = &node->window;
434         node->owner = *sender;
435     }
436     else if (!strcmp(resource,"terminal") && (node =
437         create_terminal(screen,map,output,buf,size,sender)))
438     {
439         conn = &node->terminal;
440     }
441     else if (!strcmp(resource,"picture"))
442     {
443         if (picture.pid = Newproc("picture", //processes/picture",YES,-1))
444         {
445             p = Alloc(size,YES);
446             memcpy(p,buf,size);
447             Free(Call(DIRECT,picture.pid,p,0,0));
448             conn = &picture;
449         }
450     }
451     if (conn)
452     {
453         Reply(buf,Newmsg(200,"connect","conn=#C; orig=#S; res=#S",
454             conn,"console","create",resource));
455         else
456             reply_status(buf,"-create","unknown resource type",0);
457         activate(node);
458     }
459
460     Delete_resource(map,msg,conn,sel)
461     LIST
462     register MESSAGE *map;
463     register CONNS *msg;
464     register CONNECTOR *conn;
465     register MAPNODE *node,*temp;
466     register MESSAGE *resource;

```

```

465 if (resource=(CONNECTOR*))Find triple(msg->buf,"conn",msg->size,NULL,8,NULL)
466 {
467   if (!strcmp(Find_triple(msg->buf,"res",0,NULL,2,NULL),"picture"){
468     Put(DIRECT,resource->pid,Newmsg(32,"quit",NULL));
469     remap(&msg->sender,NULL,NULL,sel,map);
470   }
471   }else
472   {
473     temp = map->active;
474     for (node = map->first;
475          node && node->window.pid != resource->pid
476          && node->picture.pid != resource->pid
477          && node->terminal.pid != resource->pid; node = node->nxt) ;
478     if (node) window(node,map,sel,conn);
479     if (Find triple(msg->buf,"rply",msg->size,NO,0,NULL))
480       if (reply status(msg->buf,"delete","resource deleted",cx_DELETED);
481           map->active = temp;
482     }
483   }
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514

```

```

input(screen,map,sel>window,msg,conn,msgid)
SCREEN *screen;
LIST *map;
register WINDOW *sel;
register MESSAGE *window;
register CONNS *msg;
register char *conn;
register char *msgid;
register char *code;
register short *pos;
register MAPNODE *hnode;
pos = (short *) Find once(msg->buf,"pos",msg->size,none,4,NULL);
code = *Find triple(msg->buf,"0\0\0",msg->size,none,1,NULL);
if (msgid == 'k' && node)
else key input(node,window,msg,code);
else if (msgid == 'f' && hnode)
function_key(node,code,&conn->dialogue);
else
{
node = find window(map>window,*pos,*(pos+1));
if (msgid == 'p')
{
if (node && window->area == 'I')
screen->row = *pos;
screen->col = *(pos+1);
}
}

```

```

516 if (msgid == 'A')
517   action(node,screen,map,sel>window,msg,conn,code,*pos,*pos+1));
518 else if (msgid == 'M')
519   menu (node,&map->metaphor,code,pos,&conn->dialogue);
520 }
521 }
522 key input(node>window,msg,code)
523 register MAPNODE *node;
524 WINDOW *window;
525 register MESSAGE *msg;
526 register char code;
527 {
528   register char *m;
529   register EDIT *edit;
530   if (node->terminal.pid)
531   {
532     Forward(DIRECT node->terminal.pid,msg->buf);
533     msg->buf = NULL;
534   }
535   else if (edit = node->edit)
536   {
537     if (code == 127)
538     {
539       *edit->pos++ = code;
540       if (m = Alloc(edit->msg_size,YES))
541       {
542         memcpy(m,edit->draw msg,edit->msg_size);
543         put(DIRECT,edit->picture.pid,m);
544       }
545     }
546     else if (node->on box)
547     {
548       notify_process(node,
549         edit->row,edit->col,'B','I',edit->hdr,code,NULL);
550     }
551     else if (node->on row)
552     {
553       move mark(edit->row,>pos-edit->text)*VCHAR WD &node->picture);
554       edit->col+(edit->row->pos-edit->special_code)}
555       if (*node->special && strchr(node->special_code)
556         notify_process(node,edit->row,edit->col,'I',NULL,code,node);
557     }
558     else if (node->on anychar)
559     {
560       if ((code > 31 && code < 127) || code == 13 || code == 8)
561       notify_process(node,edit->row,edit->col,'A','I',NULL,code,node);
562     }
563   }
564 }
565 }

```

```

567 edit text(edit, code, node, window)
568 register EDIT *edit;
569 register char code;
570 register MAPNODE *node;
571 register WINDOW *window;
572 {
573     register char *m;
574     if (node->picture.pid)
575     {
576         case 8: if (edit->pos > edit->text)
577             {
578                 edit->pos--;
579                 memcpy(edit->pos, edit->pos+1, strlen(edit->pos+1));
580                 *edit->text_end = '\0';
581                 if (m = Alloc(edit->msg_size, YES))
582                     memcpy(m, edit->draw_msg, edit->msg_size);
583                 Put(DIRTY, edit->picture.pid, m);
584             }
585         else if (node->on_delete)
586             notify_process(node, edit->row, edit->col,
587                 'D+', I, edit->hdr, code, NULL);
588         break;
589     case 9:
590     case 11:
591     case 12:
592     case 10:
593     case 13:
594         if (node->on_modify)
595             end_edit(node, 'M', window->row, window->col, code);
596     }
597 }
598
599
600
601
602
603
604

```



```

605 end edit(node, why, row, col, code)
606 register MAPNODE *node;
607 register char why, code;
608 register short row, col;
609
610 register char *element = NULL, *reply = NULL;
611 register EDIT *edit;
612
613 if (edit = node->edit)
614 {
615     if (why && (why != 'X' || node->on_cancel))
616     {
617         reply = Call(DIRECT, node->picture, pid, Newmsg(64, "hit",
618             "pos=#25", (edit->hdr)->row, (edit->hdr)->col, 0, 0);
619         element = Find triple(reply, "data", 0, NULL, 1, NULL);
620         notify process(node, row, col, why, I, element, code, NULL);
621         Free(reply);
622     }
623     put(DIRECT, node->picture, pid, Newmsg(64, "select",
624         "@pos=#25; ofi#", (edit->hdr)->row, (edit->hdr)->col));
625     Free(edit->draw_msg);
626     edit->draw_msg = NULL;
627     Free(node->edit);
628     node->edit = NULL;
629
630 }
631

```

```

6332 position(node,window)
6333 register MAPNODE *node;
6334 register WINDOW *window;
6335 {
6336     register short *reply;
6337     register P_E_HDR *hdr;
6338     if (node->auto_highlight)
6339     {
6340         if (window->different)
6341             put(DIRECT,node->picture.pid,Newmsg(32,"select","off"));
6342         reply = (short *) Call(DIRECT,node->picture.pid,window->row,window->col),0,0);
6343         if (hdr = Newmsg(64,"hit", "pos=#2s; sel",window->row,window->col),0,0);
6344         {
6345             window->different = {window->node != window->previous
6346                                 |hdr->row != window->prev_row
6347                                 |hdr->col != window->prev_col};
6348             window->prev_row = window->elem_row;
6349             window->prev_col = window->elem_col;
6350             window->elem_row = hdr->row;
6351             window->elem_col = hdr->col;
6352         }
6353         if (Free(reply));
6354     }
6355     if (node->on_location)
6356         notify_process(node,window->row,window->col,'L','I',NULL,NULL,NULL);
6357 }
6358
6359
6360
6361

```

```

662 action(node,screen,map,sel>window,msg,conn,act,row,col)
663 register MAPNODE *node;
664 *screen;
665 *map;
666 register LIST *sel;
667 register SELECTION *window;
668 *msg;
669 MESSAGE *conn;
670 register char act;
671 row, col;
672 register short
673 {
674     switch (act)
675     {
676     case 's': select(node,screen,map,sel>window,msg,conn);
677     break;
678     case 'w': Put(DIRECT,conn->dialogue.pid
679     Newmsg(64,"Open", "pos=#2s",row,col));
680     break;
681     case 'x': if (sel->pending)
682     if (deselect(screen,map,sel,row,col);
683     break;
684     case 'u': case 'l': case 'r':
685     case 'd': case 'L': case 'R':
686     scroll(act,map->active);
687     break;
688     case 'n': next>window(map);
689     break;
690     case 'c': cancel(sel);
691     break;
692     case 'w': close(node,map,sel,conn);
693     break;
694     case 'h': notify_process(node,row,col,'?',NULL,NULL,map->active);
695     break;
696     case 't': NewProc("test", "//processes/test",NO,-1);
697     break;
698     case '-': Put(DIRECT,conn->output.pid,Newmsg(32,"hide",NULL));
699     break;
700     case '+': Put(DIRECT,conn->output.pid,Newmsg(32,"restore",NULL));
701     break;
702     }

```

```

703 function key(node, key_no, dialogue)
704 register MAPNODE *node;
705 char key_no;
706 register CONNECTOR *dialogue;
707 {
708     register char *reply;
709     if (key_no && node)
710         if (!strcmp(reply, "keys"))
711             {
712                 reply = Realloc(reply, 256, YES);
713                 strcpy(reply, "Key");
714                 Append-triple(reply, "num", 1, &key_no);
715                 Append-triple(reply, "owner", 8, &node->owner);
716                 Put(DIRECT, dialogue->pid, reply);
717             }
718             else Free(reply);
719 }
720
721 menu(node, metaphor, key_no, pos, dialogue)
722 register MAPNODE *node *metaphor;
723 register char key_no;
724 short *pos;
725 CONNECTOR *dialogue;
726 {
727     register char *reply; *owner = NULL;
728     register CONNECTOR *owner;
729     if (node)
730         if (owner = &node->owner;
731             else
732                 node = metaphor;
733             if (key_no && node && (reply = Call(DIRECT, node->>window.pid,
734                 Newmsg(64, "menu?", "key=#b", key_no, 0, 0)))
735                 if (!strcmp(reply, "failed"))
736                     {
737                         Free(reply);
738                         reply = NULL;
739                     }
740                     if (Newmsg(64, "menu?", "key=#b", key_no, 0, 0))
741                         if (!strcmp(reply, "failed"))
742                             {
743                                 Free(reply);
744                                 reply = NULL;
745                             }
746                             if (reply)
747                                 {
748                                     Free(reply);
749                                     reply = NULL;
750                                 }
751                             if (reply)
752                                 {
753                                     Free(reply);
754                                     reply = NULL;
755                                 }
756                             if (reply)

```

```

754 reply = Realloc(reply, 256, YES);
755 strcpy(reply, "Menu");
756 AppendTriple(reply, "pos ", 4, pos);
757 if (owner)
758 AppendTriple(reply, "owner", 4, owner);
759 Put(DIRECT, dialogue->pid, reply);
760
761
762 close(node, map, sel, conn)
763 register MAPNODE *node;
764 register LIST *map;
765 register SELECTION *sel;
766 register CONNS *conn;
767
768 {
769 if (node && !node->keep_open)
770 if (node->on_close)
771 notify_process(node, 0, 0, 'C', NULL, NULL, NULL, map->active);
772 else
773 close_window(node, map, sel, conn);
774
775
776 close_window(node, map, sel, conn)
777 register MAPNODE *node;
778 register LIST *map;
779 register SELECTION *sel;
780 register CONNS *conn;
781
782 {
783 end_edit(node, 'X', 0, 0, NULL);
784 Put(DIRECT, node->window.pid, Newmsg(32, "Q", NULL));
785 if (node->terminal.pid)
786 {
787 Put(DIRECT, node->terminal.pid, Newmsg(32, "quit", NULL));
788 Put(DIRECT, node->picture.pid, Newmsg(32, "quit", NULL));
789 }
790 node->window.pid = node->picture.pid = node->terminal.pid = NULL;
791 if (node == map->active)
792 {
793 Put(DIRECT, conn->dialogue.pid, Newmsg(32, "keys", NULL));
794 next_window(map);
795 }
796 if (node == map->active)
797 map->active = NULL;
798 if (node == sel->map)
799 sel->map = NULL;
800 sel->pehding = No;

```

```

801 } if (node->on_quit)
802     notify_process(node, 0, 0, 'Q', NULL, NULL, NULL, map->active);
803 unmap(node, map);
804 free_node(node);
805 clip_window(map->last);
806 }
807
808 next_window(map) *map;
809 register LIST *node;
810 {
811     register MAPNODE *node;
812     if ((node = map->active) && node->nxt)
813         node = node->nxt;
814     while (node && node->never && node != map->active)
815     {
816         node = node->nxt;
817         if (!node)
818             node = map->first;
819     }
820     if (node)
821     {
822         unmap(node, map);
823         map_after(node, NULL, map);
824         activate(node);
825         clip_window(map->last);
826     }
827 }
828
829 select(node, screen, map, sel, window, msg, conn)
830 register MAPNODE *node;
831 register LIST *map;
832 register SELECTION *sel;
833 register WINDOW *window;
834 register MESSAGE *msg;
835 register CONNS *conn;
836 {
837     if (sel->pending)
838         cancel(sel);
839     if (node)
840     {
841         Put(DIRECT, node->picture.pid, Newmsg(32, "select", "off"));
842         sel->row = window->row;
843         sel->col = window->col;
844         sel->area = window->area;
845         sel->map = node;
846         if (sel->area != 'I')
847             if (
848

```

```

850     if (!node->metaphor)
851         sel_window(hode, screen, map, sel, window, conn);
852     }
853     else if (!node->terminal.pid)
854         sel_element(node, map, sel, msg);
855     activate(node);
856 }
857
858 sel_element(node, map, sel, msg)
859     register MAPNODE *node;
860     LIST *map;
861     register SELECTION *sel;
862     register MESSAGE *msg;
863 {
864     register char *reply;
865     long size;
866     if (node->move mark)
867         move mark(sel->row, sel->col, &node->picture);
868     if (reply = Call(DIRECT, node->picture.pid,
869         Newmsg(64, "hit", "pos=#2s; sel" sel->row, sel->col), 0, &size))
870     {
871         if (!strcmp(reply, "write"))
872         {
873             Free(msg->buf);
874             sel->pending = YES;
875             msg->buf = reply;
876             msg->size = size;
877             msg->sender = node->picture;
878             element_selected(map, sel, msg);
879         }
880     }
881     else if (node->on_select)
882     {
883         notify_process(node,
884             sel->row, sel->col, 's', 'I', NULL, NULL, map->active);
885         Free(reply);
886     }
887 }

```

```

888 element_selected(map, sel, msg)
889 LIST *map;
890 register SELECTION *sel;
891 register MESSAGE *msg;
892 {
893     register MAPNODE *node;
894     register P E HDR *hdr;
895     register short row, col;
896
897     node = sel->map;
898     if ( !sel->pending )
899         for ( node = map->first;
900             node && (node->picture.pid != msg->sender.pid); node = node->nxt);
901     if ( node && node->picture.pid == msg->sender.pid )
902     {
903         activate(node);
904         end_edit(node);
905         if (hdr = (P E HDR*) find_triple(msg->buf, "data", msg->size, NULL, 1, NULL))
906         {
907             row = hdr->row;
908             col = hdr->col;
909             if (sel->pending)
910             {
911                 row = sel->row;
912                 col = sel->col;
913                 if (node->on_element)
914                     notify_process(node, row, col, 'S', 'I', hdr, NULL, map->active);
915             }
916             if (hdr->attr.editable && hdr->type == 't')
917                 start_edit(msg, node, hdr, row, col);
918             else
919                 Put(DIRECT, node->picture.pid, Newmsg(32, "select", "off"));
920         }
921     }
922     sel->pending = NO;
923 }
924

```



```

9226 start edit(msg,node,hdr,row,col)
9227 MESSAGE *msg;
9228 register MAPNODE *node;
9229 register P_E_HDR *hdr;
9230 register short row,col;
9231 {
9232     register EDIT *edit;
9233     register short offset;
9234     register char *pos;
9235
9236     node->edit = edit = (EDIT *) Alloc(sizeof(EDIT),YES);
9237     strcpy(edit,"edit:");
9238     edit->draw,msg = msg->buf;
9239     strcpy(edit->draw,msg,"replace");
9240     edit->msg_size = msg->size;
9241     msg->buf = NULL;
9242     offset = ((row - hdr->row) * hdr->width) + (col - hdr->col) / VCHAR_WD;
9243     edit->hdr = hdr;
9244     edit->picture.pid = node->picture.pid;
9245     edit->type = edit->hdr->type;
9246     pos = (char *) hdr + sizeof(P_E_HDR);
9247     if (hdr->attr.appl)
9248         if (pos += 4;
9249             if (hdr->attr.tagged)
9250                 if (pos += strlen(pos) + 1;
9251                 Long align(pos);
9252                 pos += sizeof(long) + 2 * sizeof(short);
9253                 edit->text_end = edit->pos = pos;
9254                 edit->text_end += strlen(pos) - 1;
9255                 edit->pos += offset;
9256                 edit->row = hdr->row;
9257                 edit->col = hdr->col;
9258                 edit->height = hdr->height;
9259                 edit->width = hdr->width;
9260                 move_mark(row,col,&node->picture);
9261             }
9262         }
9263     }

```

```

962 sel_window(node,screen,map,sel,window,conn)
963 register MAPNODE *node;
964 *map;
965 *screen;
966 register SELECTION *sel;
967 register WINDOW *window;
968 *conn;
969 {
970     register char *tag = NULL;
971     sel->pending = NO;
972     if (window->hdr && window->hdr->attr.tagged && window->hdr->attr.selectable)
973     {
974         tag = (char *) window->hdr + sizeof(P_E_HDR);
975         if (window->hdr->attr.appl)
976             tag += 4;
977     }
978     if (tag && strcmp(tag, "RESIZE!"))
979     {
980         if (!strcmp(tag, "CLOSE!"))
981             close(node, map, sel, conn);
982         else if (!strcmp(tag, "FILL!"))
983             fill(screen(node, screen, map));
984         else if (!strcmp(tag, "UP!"))
985             strcmp(tag, "DOWN!");
986         else if (!strcmp(tag, "LEFT!"))
987             strcmp(tag, "RIGHT!");
988         else
989             scroll(*tag-'A'+1, a, node);
990         notify_process(node, window->row, window->col,
991             'S+', window->bar, window->hdr, NULL, node);
992     }
993     else if (sel->pending == !node->nonmod && (window->area == 'r'
994         || window->area == 'c' || !strcmp(tag, "RESIZE!")))
995     {
996         Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", CYAN, 'O'));
997         Put(DIRECT, node->window.pid,
998             Newmsg(64, "c", "colr=#b; bar=#b; tag=#s", RED, 'r', "RESIZE!"));
999     }
1000     else if (sel->pending == !node->fixed)
1001         Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", RED, 'O'));
1002

```

```

1003 fill screen(node, screen, map)
1004 register MAPNODE *node;
1005 register SCREEN *screen;
1006 register LIST *map;
1007 {
1008     register short map_row, map_col, term_adjust, *p;
1009     char
1010     if (!node->fill_ht)
1011     {
1012         Put(DIRECT, node->window.pid, bar=#b; tag=#S" RED, 'T', "FILL!");
1013         Newmsg(64, "c", "colr=#b; tag=#S" RED, 'T', "FILL!");
1014         term_adjust = screen->meta_ht - node->out_ht;
1015         memcpy(&node->fill_row, &node->row, 4*sizeof(short));
1016         node->row = node->col = 0;
1017         node->height = screen->meta_ht - node->top - node->bottom;
1018         node->width = screen->meta_wd - node->left - node->right;
1019     }
1020     else
1021     {
1022         Put(DIRECT, node->window.pid, bar=#b; tag=#S" 0 'T', "FILL!");
1023         Newmsg(64, "c", "colr=#b; tag=#S" 0 'T', "FILL!");
1024         memcpy(&node->row, &node->fill_row, 4*sizeof(short));
1025         term_adjust = node->out_ht - screen->meta_ht;
1026         node->fill_ht = 0;
1027     }
1028     align window(screen, node);
1029     if (reply = Call(DIRECT, node->window.pid, Newmsg(32, "query", NULL), 0, 0))
1030     {
1031         p = (short *) Find_triple(reply, "view", 0, none, 4, NULL);
1032         map_row = *p++;
1033         map_col = *p;
1034         Free(reply);
1035         if (node->terminal.pid)
1036             if ((map_row == term_adjust) < 0)
1037                 map_row = 0;
1038         Put(DIRECT, node->window.pid, Newmsg(128, "set", "pos=#2s; size=#2s; map=#2s",
1039             node->row, node->col, node->height, node->width, map_row, map_col));
1040         activate(node);
1041         clip_window(map->last);
1042     }
1043 }
1044 }
1045 }
1046 }

```

```

1047 cancel(sel)
1048 register SELECTION *sel;
1049 {
1050     register MAPNODE *node;
1051     if ((node = sel->map) && sel->pending)
1052     {
1053         end edit(node, 'X', 0, 0, NULL);
1054         if (node->picture, pid)
1055             Put(DIRECT, node->picture.pid, Newmsg(32, "select", "off"));
1056         if (node->window.pid)
1057         {
1058             Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1059             Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1060             Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1061             Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1062             Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1063             Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1064         }
1065     }
1066     sel->pending = NO;
1067 }
1068 deselect(screen, map, sel, row, col)
1069 register SCREEN *screen;
1070 register LIST *map;
1071 register SELECTION *sel;
1072 register short row, col;
1073 {
1074     register MAPNODE *node;
1075     sel->pending = NO;
1076     node = sel->map;
1077     if (sel->area == 'r' || sel->area == 'c')
1078     {
1079         resize(screen, node, - node->top - node->bottom,
1080             row - node->row, - node->left - node->right);
1081         col = node->col;
1082         Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'r', "RESIZE!"));
1083     }
1084     else
1085     {
1086         node->row = row;
1087         node->col = col;
1088         align_window(screen, node);
1089         Put(DIRECT, node->window.pid, Newmsg(64, "set", "pos=#2s", node->row, node->col));
1090         Put(DIRECT, node->window.pid, Newmsg(64, "set", "pos=#2s", node->row, node->col));
1091         clip_window(map->last);
1092         Put(DIRECT, node->window.pid, Newmsg(64, "c", "colr=#b; bar=#b", 0, 'O'));
1093     }
1094 }
1095 }
1096 }

```

```

1097 resize(screen, node, new ht, new wd)
1098 register SCREEN *screen;
1099 register MAPNODE *node;
1100 register short new ht, new wd;
1101 {
1102     register short map_row, map_col, *p;
1103     register char *reply;
1104     if (new ht < MIN_HT)
1105         new ht = MIN_HT;
1106     if (new wd < MIN_WD)
1107         new wd = MIN_WD;
1108     node->height = new ht;
1109     node->width = new wd;
1110     reply = Call(DIRECT, node->>window.pid, Newmsg(32, "query", NULL), 0, 0);
1111     p = short * Find_triple(reply, "view", 0, none, 4, NULL);
1112     map_row = *p++;
1113     map_col = *p++;
1114     free(reply);
1115     if (node->terminal.pid)
1116     {
1117         map_row = map_row - (new ht - node->out_ht);
1118         map_col = (map_row / VCHAR_HT) * VCHAR_HT;
1119     }
1120     align_window(screen, node);
1121     Put(DIRECT, node->>window.pid, Newmsg(128, "set", "size=#2s",
1122         node->height, node->width, map_row, map_col));
1123     Put(DIRECT, node->>window.pid, Newmsg(64, "C#", "colf=#b; bar=#b", 0, '0'));
1124 }
1125
1126

```

```

1127 scroll(direction,node)
1128 register char direction;
1129 register MAPNODE *node;
1130 {
1131     register char *reply;
1132     register short low_row, low_col;
1133     register short map_row, map_col;
1134     register short pict_ht, pict_wd, *p;
1135     if (node && node->picture.pid && node->window.pid && !node->metaphor)
1136     if (reply = Call(DIRECT,node->window.pid,Newmsg(64,"query",NULL),0,0))
1137     {
1138         if (p = (short *) Find_triple(reply,"view",0,NULL,4,NULL))
1139         {
1140             map_row = *p++;
1141             map_col = *p;
1142             Free(reply);
1143             reply = Call(DIRECT,node->picture.pid,Newmsg(32,"query",NULL),0,0);
1144             p = (short *) Find_triple(reply,"size",0,NULL,4,NULL);
1145             pict_ht = *p++;
1146             pict_wd = *p;
1147             p = (short *) Find_triple(reply,"low",0,NULL,4,NULL);
1148             low_row = *p++;
1149             low_col = *p;
1150             scroll_pos(node,direction,low_row,low_col,pict_ht,pict_wd);
1151             &map_row, &map_col, low_row, low_col, pict_ht, pict_wd);
1152             Put(DIRECT,node->window.pid,Newmsg(64,"map",
1153             "to=#C", at=#2s", &node->picture, map_row, map_col));
1154             Free(reply);
1155         }
1156     }
1157 }
1158

```

```

1160 scroll pos(node,direction,map_row,map_col,low_row,low_col,pict_ht,pict_wd)
1161 register MAPNODE *node;
1162 register char direction;
1163 register short low_row,low_col,pict_ht,pict_wd,*map_row,*map_col;
1164 {
1165     switch (direction)
1166     {
1167     case 'u': if (*map_row - low_row >= VCHAR_HT)
1168               break;
1169               if (*map_row == VCHAR_HT;
1170               if (pict_ht - (*map_row - low_row) - node->height >= VCHAR_HT)
1171               break;
1172               *map_row += VCHAR_HT;
1173               if (*map_col - low_col >= VCHAR_WD)
1174               break;
1175               *map_col == VCHAR_WD;
1176               if (pict_wd - (*map_col - low_col) - node->width >= VCHAR_WD)
1177               break;
1178               *map_col += VCHAR_WD;
1179               if (*map_row - low_row >= node->height)
1180               break;
1181               *map_row == node->height;
1182               else *map_row = low_row;
1183               break;
1184               if (pict_ht - (*map_row - low_row) >= 2 * node->height)
1185               break;
1186               *map_row += node->height;
1187               else *map_row = pict_ht - low_row - node->height;
1188               break;
1189               if (*map_col - low_col >= node->width)
1190               break;
1191               *map_col == node->width;
1192               else *map_col = low_col;
1193               break;
1194               if (pict_wd - (*map_col - low_col) >= 2 * node->width)
1195               break;
1196               *map_col += node->width;
1197               else *map_col = pict_wd - low_col - node->width;
1198               break;
1199               *map_col = pict_wd - low_col - node->width;
1200     }

```

```

1201 notify process(node, row, col, act, area, hdr, indic, active)
1202 register MAPNODE *node;
1203 register P E HDR *hdr; area;
1204 register char act;
1205 register char indic; col;
1206 short row; *active;
1207 MAPNODE
1208 {
1209     register char *p *m;
1210     register int len = 0;
1211     if (hdr) = *(short *) hdr;
1212     m = Newmsg(len+200, "click", #s; actn=#b; what=#b; pos=#2s"
1213         &node->window, &node->picture, node->name, act, area, row, col);
1214     if (hdr)
1215     {
1216         p = Append triple(m, "data", len+6, hdr);
1217         { (P E HDR *p) -> attr.selected = NO;
1218           + = *(short *) p;
1219           Long align(p);
1220           *(short *) p = NULL;
1221         }
1222     }
1223     if (indic)
1224     Append triple(m, "char", 1, &indic);
1225     if (active)
1226     Append triple(m, "acty", 4, &active->owner);
1227     Put(DIRECT, node->owner.pid, m);
1228 }
1229
1230
1231

```



```

12333 Metaphor(screen, map, buf, size, output, dialogue)
12334 register SCREEN *screen;
12335 register LIST *map; size, output;
12336 register long buf, dialogue;
12337 CONNECTOR
12338 {
12339     register short *p;
12340     register MAPNODE *node;
12341
12342     screen->meta_row = screen->meta_col = 0;
12343     screen->meta_ht = screen->height;
12344     screen->meta_wd = screen->width;
12345     if (node = create_window(screen, map, output, "Metaphor", buf, size))
12346     {
12347         map->metaphor = node;
12348         node->owner = *dialogue;
12349         p = (short *) FindTriple(buf, "area", size, none, 8, NULL);
12350         screen->meta_row = *p++;
12351         screen->meta_col = *p++;
12352         screen->meta_ht = *p++;
12353         screen->meta_wd = *p;
12354         node->metaphor = node->never = node->keep_open = YES;
12355         node->fixed = node->nonmod = YES;
12356         Reply(buf, Newmsg(32, "connect", "conn=#C", &node->window));
12357     }
12358     else
12359         reply_status(buf, "-Metaphor", "can't create \window\", 0);
12360 }

```

```

12661 MAPNODE *create_terminal(screen, map, output, buf, size, sender)
12662 SCREEN *screen;
12663 REGISTER LIST *map;
12664 CONNECTOR *output;
12665 REGISTER long buf, size, sender;
12666
12667 {
12668     static char def_type[] = "//processes/terminal";
12669     REGISTER MAPNODE *node;
12670     REGISTER char *p;
12671     CONNECTOR terminal;
12672
12673     if (Find_triple(buf, "name", size, NULL, 1, NULL))
12674     {
12675         if (terminal.pid = NewProc("terminal"
12676             Find_triple(buf, "emul", size, def_type, 1, NULL), YES, -1))
12677         {
12678             p = Alloc(size, YES);
12679             memcpy(p, buf, size);
12680             memcpy(p, sender, sizeof(CONNECTOR));
12681             memcpy(p + sizeof(CONNECTOR), &terminal, sizeof(CONNECTOR));
12682             P = Call(DIRECT, terminal.pid, 0, 0);
12683             if (!strcmp(p, "create"))
12684                 &&(node) = create_window(screen, map, output, "Window", p, size))
12685             {
12686                 node->terminal = node->owner = terminal;
12687                 Free(p);
12688                 return(node);
12689             }
12690             reply_status(buf, "-create", "can't create \'terminal\'", 0);
12691         }
12692     }
12693     else
12694         reply_status(buf, "-create", "(terminal) no name given", 0);
12695     return(NULL);
12696 }

```

```

1297 MAPNODE *create_window(screen, map, output, proc, buf, size)
1298 SCREEN
1299 LIST
1300 CONNECTOR
1301 char
1302 register long
1303 {
1304     static char
1305     register char
1306     register short
1307     register MAPNODE
1308     char
1309     MAPNODE
1310     if ((window name = Find_triple(buf, "name", size, NULL, 1, NULL))
1311         && (node = new_node(map, window name))
1312         && (node->window.pid = NewProc(proc, "//processes/window", YES, -1)))
1313     {
1314         map after (node, NULL, map) / "titl", size, window_name, 1, NULL);
1315         title = Find_triple(buf, "title", size, window_name, 1, NULL);
1316         init_node(node, buf, size);
1317         strcopy(node->device, Find_triple(buf, "from", size, none, 2, NULL));
1318         strncpy(node->term, Find_triple(buf, "mod ", size, none, 1, NULL), sizeof(node->term)-1);
1319         Find_triple(buf, "spec", size, none, 1, NULL), sizeof(node->special)-1);
1320         p = Find_triple(buf, "outl", size, def_outl, 4, NULL);
1321         out_clr = *p++;
1322         node->outline = *p++;
1323         if (! (out_fill = BLACK;
1324             if (! (node->style = 's';
1325                 node->pane = 0;
1326                 pane_clr = out_clr;
1327                 if (p = Find_triple(buf, "pane", size, NULL, 2, NULL))
1328                     pane_clr = *p++;
1329                 node->pane = *p;
1330             }
1331         }
1332     }
1333     else if (node->Hscroll || node->Vscroll)
1334     {
1335         if (p = Find_triple(buf, "map ", size, NULL, 8, NULL))
1336         {
1337             node->picture = *(CONNECTOR *) p;
1338             if (*(long*) (p-4) > sizeof(CONNECTOR))
1339             {
1340
1341
1342
1343
1344

```

```

13456 pict_row = *(short *) (p + sizeof(CONNECTOR)) + sizeof(short);
13457 pict_col = *(short *) (p + sizeof(CONNECTOR)) + sizeof(short);
13458 }
13459 if (init_window(screen,node,output,title,pict_row,pict_col,
13460 out_clr,out_fill,0,pane_clr))
13461 {
13462 activate(node);
13463 clip_window(map->last);
13464 return(node);
13465 }
13466 }
13467 }
13468 }
13469 }
13470 }
13471 }
13472 }
13473 }
13474 }
13475 }
13476 }
13477 }
13478 }
13479 }
13480 }
13481 }
13482 }
13483 }

```

```

13484 }
13485 }
13486 }
13487 }
13488 }
13489 }
13490 }
13491 }
13492 }
13493 }
13494 }
13495 }
13496 }
13497 }
13498 }
13499 }
13500 }
13501 }
13502 }
13503 }
13504 }
13505 }
13506 }
13507 }
13508 }
13509 }
13510 }
13511 }
13512 }
13513 }
13514 }
13515 }
13516 }
13517 }
13518 }
13519 }
13520 }
13521 }
13522 }
13523 }
13524 }
13525 }
13526 }
13527 }
13528 }
13529 }
13530 }
13531 }
13532 }
13533 }
13534 }
13535 }
13536 }
13537 }
13538 }
13539 }
13540 }
13541 }
13542 }
13543 }
13544 }
13545 }
13546 }
13547 }
13548 }
13549 }
13550 }
13551 }
13552 }
13553 }
13554 }
13555 }
13556 }
13557 }
13558 }
13559 }
13560 }
13561 }
13562 }
13563 }
13564 }
13565 }
13566 }
13567 }
13568 }
13569 }
13570 }
13571 }
13572 }
13573 }
13574 }
13575 }
13576 }
13577 }
13578 }
13579 }
13580 }
13581 }
13582 }
13583 }
13584 }
13585 }
13586 }
13587 }
13588 }
13589 }
13590 }
13591 }
13592 }
13593 }
13594 }
13595 }
13596 }
13597 }
13598 }
13599 }
13600 }
13601 }
13602 }
13603 }
13604 }
13605 }
13606 }
13607 }
13608 }
13609 }
13610 }
13611 }
13612 }
13613 }
13614 }
13615 }
13616 }
13617 }
13618 }
13619 }
13620 }
13621 }
13622 }
13623 }
13624 }
13625 }
13626 }
13627 }
13628 }
13629 }
13630 }
13631 }
13632 }
13633 }
13634 }
13635 }
13636 }
13637 }
13638 }
13639 }
13640 }
13641 }
13642 }
13643 }
13644 }
13645 }
13646 }
13647 }
13648 }
13649 }
13650 }
13651 }
13652 }
13653 }
13654 }
13655 }
13656 }
13657 }
13658 }
13659 }
13660 }
13661 }
13662 }
13663 }
13664 }
13665 }
13666 }
13667 }
13668 }
13669 }
13670 }
13671 }
13672 }
13673 }
13674 }
13675 }
13676 }
13677 }
13678 }
13679 }
13680 }
13681 }
13682 }
13683 }
13684 }
13685 }
13686 }
13687 }
13688 }
13689 }
13690 }
13691 }
13692 }
13693 }
13694 }
13695 }
13696 }
13697 }
13698 }
13699 }
13700 }
13701 }
13702 }
13703 }
13704 }
13705 }
13706 }
13707 }
13708 }
13709 }
13710 }
13711 }
13712 }
13713 }
13714 }
13715 }
13716 }
13717 }
13718 }
13719 }
13720 }
13721 }
13722 }
13723 }
13724 }
13725 }
13726 }
13727 }
13728 }
13729 }
13730 }
13731 }
13732 }
13733 }
13734 }
13735 }
13736 }
13737 }
13738 }
13739 }
13740 }
13741 }
13742 }
13743 }
13744 }
13745 }
13746 }
13747 }
13748 }
13749 }
13750 }
13751 }
13752 }
13753 }
13754 }
13755 }
13756 }
13757 }
13758 }
13759 }
13760 }
13761 }
13762 }
13763 }
13764 }
13765 }
13766 }
13767 }
13768 }
13769 }
13770 }
13771 }
13772 }
13773 }
13774 }
13775 }
13776 }
13777 }
13778 }
13779 }
13780 }
13781 }
13782 }
13783 }
13784 }
13785 }
13786 }
13787 }
13788 }
13789 }
13790 }
13791 }
13792 }
13793 }
13794 }
13795 }
13796 }
13797 }
13798 }
13799 }
13800 }
13801 }
13802 }
13803 }
13804 }
13805 }
13806 }
13807 }
13808 }
13809 }
13810 }
13811 }
13812 }
13813 }
13814 }
13815 }
13816 }
13817 }
13818 }
13819 }
13820 }
13821 }
13822 }
13823 }
13824 }
13825 }
13826 }
13827 }
13828 }
13829 }
13830 }
13831 }
13832 }
13833 }
13834 }
13835 }
13836 }
13837 }
13838 }
13839 }
13840 }
13841 }
13842 }
13843 }
13844 }
13845 }
13846 }
13847 }
13848 }
13849 }
13850 }
13851 }
13852 }
13853 }
13854 }
13855 }
13856 }
13857 }
13858 }
13859 }
13860 }
13861 }
13862 }
13863 }
13864 }
13865 }
13866 }
13867 }
13868 }
13869 }
13870 }
13871 }
13872 }
13873 }
13874 }
13875 }
13876 }
13877 }
13878 }
13879 }
13880 }
13881 }
13882 }
13883 }
13884 }
13885 }
13886 }
13887 }
13888 }
13889 }
13890 }
13891 }
13892 }
13893 }
13894 }
13895 }
13896 }
13897 }
13898 }
13899 }
13900 }
13901 }
13902 }
13903 }
13904 }
13905 }
13906 }
13907 }
13908 }
13909 }
13910 }
13911 }
13912 }
13913 }
13914 }
13915 }
13916 }
13917 }
13918 }
13919 }
13920 }
13921 }
13922 }
13923 }
13924 }
13925 }
13926 }
13927 }
13928 }
13929 }
13930 }
13931 }
13932 }
13933 }
13934 }
13935 }
13936 }
13937 }
13938 }
13939 }
13940 }
13941 }
13942 }
13943 }
13944 }
13945 }
13946 }
13947 }
13948 }
13949 }
13950 }
13951 }
13952 }
13953 }
13954 }
13955 }
13956 }
13957 }
13958 }
13959 }
13960 }
13961 }
13962 }
13963 }
13964 }
13965 }
13966 }
13967 }
13968 }
13969 }
13970 }
13971 }
13972 }
13973 }
13974 }
13975 }
13976 }
13977 }
13978 }
13979 }
13980 }
13981 }
13982 }
13983 }
13984 }
13985 }
13986 }
13987 }
13988 }
13989 }
13990 }
13991 }
13992 }
13993 }
13994 }
13995 }
13996 }
13997 }
13998 }
13999 }
14000 }

```

```
1385 check_bar(ptr, keyw, deflt)
1386 register char *ptr, *keyw;
1387 register short deflt;
1388 {
1389     register short *p;
1390     if (!p == (short *) Find_triple(ptr, keyw, 0, NO, 0, NULL))
1391         return(NO);
1392     else if (p == (short *) I)
1393         return(deflt);
1394     else return(*p);
1395 }
1396
1397
1398
```

```

1399 window options(node, buf, size)
1400 register MAPNODE *node;
1401 register long buf, size;
1402 {
1403     register char *options, opt;
1404     options = Find_triple(buf, "when", size, none, 1, NULL);
1405     while (opt = *options++)
1406     {
1407         case 'S': node->on_element = opt; break;
1408         case 'X': node->on_cancel = opt; break;
1409         case 's': node->on_select = opt; break;
1410         case 'O': node->on_open = opt; break;
1411         case 'M': node->on_modify = opt; break;
1412         case 'C': node->on_close = opt; break;
1413         case 'Q': node->on_quit = opt; break;
1414         case 'W': node->on_window_edge = opt; break;
1415         case 'P': node->on_picture_edge = opt; break;
1416         case 'A': node->on_anychar = opt; break;
1417         case 'D': node->on_delete = opt; break;
1418         case 'B': node->on_box = opt; break;
1419         case 'I': node->on_insert = opt; break;
1420         case 'N': node->on_insert = opt; break;
1421     }
1422     options = Find_triple(buf, "opt ", size, none, 1, NULL);
1423     while (opt = *options++)
1424     {
1425         case 'H': node->auto_highlight = opt; break;
1426         case 'E': node->editable = opt; break;
1427         case 'S': node->multi_select = opt; break;
1428         case 'X': node->never = opt; break;
1429         case 'B': node->remap = opt; break;
1430         case 'N': node->nonmod = opt; break;
1431         case 'F': node->fixed = opt; break;
1432         case 'O': node->keep_open = opt; break;
1433         case 'M': node->move_mark = opt; break;
1434         case '+': node->tight = opt; break;
1435         case '-': node->picture.pid = NULL; break;
1436     }
1437 }
1438
1439
1440
1441

```

```

1442  init window(screen,node,output,title,row,col,out_clr,out_fill,out_pat,pane_clr)
1443  register SCREEN *screen;
1444  register MAPNODE *node;
1445  CONNECTOR short row,col;
1446  register char *title;
1447  register char out_clr, out_fill, out_pat, pane_clr;
1448  {
1449  register char *msg;
1450  int result = NO;
1451  if (node->style == 's' && (screen->colors < 7 || !screen->bit_map))
1452  if (node->style == 's');
1453  if (out_line(node));
1454  if (node->palette = node->palette;
1455  if (node->resize_box || node->vscroll)
1456  if (node->right != VCHAR_WD;
1457  if (node->corner && !node->palette)
1458  if (node->left != VCHAR_WD;
1459  if (node->menu || node->general_use)
1460  else if (node->bottom == VCHAR_HT * 2;
1461  align node->bottom = VCHAR_HT;
1462  msg = Newmsg(3000,init);
1463  self = #C; map = #C#2s; name = #S;
1464  node->row,node->col,node->width,pane_clr,node->pane,
1465  node->top,node->bottom,node->left,node->right,screen->height,
1466  screen->width,output,&node->window,&node->picture,row,col,node->name);
1467  init frame(msg,node,title,out_clr;
1468  msg = Call(DIRFCT,node->window.pid,msg,0,0);
1469  result = strcmp(msg,"failed");
1470  return(result);
1471  )

```

```

1482 out_line(node)
1483 register MAPNODE *node;
1484 (
1485     node->outer = node->outline + node->pane + (node->outline && node->pane) *
1486     (node->height/100 + node->width/100 + 2);
1487     if (node->tight)
1488     {
1489         node->top = node->bottom = node->outer;
1490         node->left = node->right = node->outer + node->width/200;
1491     }
1492     else
1493     {
1494         node->top = VCHAR_HT;
1495         node->bottom = node->outer;
1496         node->left = node->right = VCHAR_WD;
1497     }
1498     if (node->style == 's')
1499     {
1500         node->bottom += 5;
1501         node->right += 5;
1502     }
1503 )
1504 )

```



```

1505 init frame(msg,node,title,out_clr)
1506 register MAPNODE *node;
1507 register char *msg,*title,out_clr;
1508 {
1509     char
1510     register char
1511     register PE_HDR
1512     static short
1513     static short
1514     static short
1515     static short
1516     static long
1517     {
1518         *n,*frame_bar({i6})+YELLOW,title_clr=WHITE;
1519         scroll_clr={3,0,6,7,12,7,9,10,9,10,3,7,3,7,0};
1520         *hdr;
1521         up_arrow[] = {3,0,3,0,3,0,9,3,9,3,12,10,6,3,0};
1522         down_arrow[] = {6,0,0,7,3,7,3,6,10,6,3,6,0};
1523         left_arrow[] = {3,0,0,3,7,3,6,10,6,3,9,0,6,0};
1524         right_arrow[] = {3,0,3,6,6,3,6,10,6,3,9,0,3,0};
1525         resize_symbol[] = {0x00007f80,0x7f807f80,0x7f807f80,0x7f807f80};
1526         0x7f807f80,0x00ec007c,0x003c007c,0x00fc0000,0x00000000};
1527     if (node->title)
1528     {
1529         n = frame_bar(msg,"top",400,'T',0,0,node->title,1000,0,out_clr,0,NO);
1530         draw_filled_rect(&n,0,0,node->title,1000,NULL,0,0,out_clr,0,0,6,6,"a");
1531         draw_rect(&n,5,10,10,"CLOSE!");
1532         draw_text(&n,5,10,3*VCHAR_WD,title,"NAME",title_clr,0,NULL,NULL);
1533         if (!node->nonmod)
1534         {
1535             hdr = (PE_HDR *) start_macro(&n,0,1000,
1536             VCHAR_HT-2,2*VCHAR_WD,"N","fill",0,0,"sa");
1537             draw_rect(&n,rect(0,7,2*VCHAR_WD-4,NULL,title_clr,'S',1,NULL));
1538             draw_filled_rect(&n,6,3*VCHAR_HT-14,2*VCHAR_WD-10,
1539             NULL,0,0,title_clr,0,0,NULL);
1540             end_macro(&n,hdr);
1541         }
1542         draw_end(&n);
1543     }
1544     if (node->Vscroll)
1545     {
1546         n = frame_bar(msg,"right",400,'V',node->pane-1,node->pane-1,790,
1547         node->right-node->pane-node->outline+2,out_clr,BLACK,1,NO);
1548         draw_rect(&n,node->pane,node->pane,VCHAR_HT-4,
1549         node->right-(node->pane)-{hnode->outline,
1550         "SCROLL",scroll_clr,"S",1,"Sb"};
1551         draw_poly(&n,875,np!,scroll_clr,0,0,'S',0,1,"Sa");
1552         draw_poly(&n,980,node->pane+1,
1553         g_down_arrow,"DOWN!",scroll_clr,0,0,'S',0,1,"Sa");
1554         draw_end(&n);
1555     }
1556 }

```

```

15510
15511 if (node->Hscroll)
15512 {
15513     n = frame_bar(msg, "bot", 400, 'H', node->pane-1, 0,
15514                 node->bottom-(node->pane) -(node->outline)+2,
15515                 910, out_clr, BLACK, 1, NO);
15516     draw_rect(&n, node->pane, node->pane,
15517             node->bottom-(node->pane)-node->outline,
15518             2*VCHAR_WD-2, "SCROLL", scroll_clr, 'S', 1, "sb");
15519     draw_poly(&n, node->pane, 955,
15520             draw_poly(&n, node->pane, 990,
15521                 draw_poly(&n, node->pane, 990,
15522                     draw_end(&n);
15523
15524     if (node->menu)
15525     frame_bar(msg, "bot", 200, 'M', node->pane-1, 0,
15526             node->bottom-(node->pane) -(node->outline)+2,
15527             1000, out_clr, BLACK, 1, YES);
15528     if (node->general_use)
15529     frame_bar(msg, "bot", 200, 'G', node->pane-1, 0,
15530             node->bottom-(node->pane) -(node->outline)+2,
15531             1000, out_clr, BLACK, 1, YES);
15532     if (node->palette)
15533     frame_bar(msg, "left", 200, 'P', 0, node->pane, 10000,
15534             node->left-(node->pane) -(node->outline)-1, out_clr, BLACK, 1, YES);
15535     if (
15536     {
15537         n = frame_bar(msg, "rbox", 200, NULL, 0, 0, 0, scroll_clr, BLACK, 1, NO);
15538         draw_symbol(&n, 0, 0, 16, 16, resize_symbol, "RESIZE!", scroll_clr, 0, "S");
15539         draw_end(&n);
15540     }
15541     if (node->corner)
15542     frame_bar(msg, "lbox", 200, NULL, 0, 0, 0, out_clr, BLACK, 1, YES);
15543
15544 }

```

```

1585 char *frame bar(msg, keyw, size, type, row, col, height, width, color, fill, thick, end)
1586 register char *msg, *keyw;
1587 char type, color, fill, end;
1588 register short row, col, height, width, size, thick;
1589 {
1590     char *n;
1591     n = Append triple(msg, keyw, size, NULL);
1592     *n++ = type;
1593     draw filled, rect(&n, row, col, height, width);
1594     draw NULL, color, 0, fill, 0, 's', 0, thick, 'a';
1595     if (end)
1596         draw end(&n);
1597     return(n);
1598 }
1599
1600 Set user(name, buf, size)
1601 register NAME *name;
1602 register long buf, size;
1603 {
1604     register char *p;
1605     if (p = Find_triple(buf, "name", size, NULL, 2, NULL))
1606     {
1607         strcpy(name->user, p);
1608         Note unsigned on, p;
1609         Put(ALL, "HI", Newmsg(128, "U", "name=#s", p));
1610     }
1611 }
1612
1613
1614

```

```

16115 change(screen,map,msg)
16116 SCREEN *screen;
16117 LIST *map;
16118 MESSAGE *msg;
16119 (
16120     register CONNECTOR *window, *owner = NULL;
16121     register short *p;
16122     register MAPNODE *node;
16123     if (window = (CONNECTOR*)Find_triple(msg->buf,"conn",msg->size,NULL,8,NULL))
16124     {
16125         for (node = map->first; node && node->window.pid != window->pid
16126             && node->terminal.pid != window->pid; node = node->nxt)
16127         {
16128             if (p = (short*) Find_triple(msg->buf,"size",msg->size,none,4,NULL))
16129             {
16130                 resize(screen,node,*p*(p+1));
16131             }
16132             if (Find_triple(msg->buf,"actv",msg->size,NULL,0,NULL)
16133                 && !node->never)
16134             {
16135                 map->active = node;
16136             }
16137             if (owner
16138                 (CONNECTOR*) Find_triple(msg->buf,"ownr",msg->size,NULL,0,NULL))
16139             {
16140                 if ((long)owner == 1)
16141                     owner = &msg->sender;
16142             }
16143             if (owner)
16144             {
16145                 node->owner = *owner;
16146                 if (node->terminal.pid)
16147                 {
16148                     Forward(DIRECT,node->terminal.pid,msg->buf);
16149                     msg->buf = NULL;
16150                 }
16151             }
16152             clip_window(map->last);
16153         }
16154     )

```

```

16553 highlight(node,map) *node;
16554 register MAPNODE *map;
16555 register LIST
16556 {
16557     if (node && node != map->last_active)
16558     {
16559         if (!node->metaphor)
16560         {
16561             Put(LOCAL,"Window"
16562                Newmsg(64,"highlight", "bar=#b; tag=#S", 'T', "CLOSE!"));
16563             if (node->window.pid && node->title)
16564                 Put(DIRECT,node->window.pid
16565                    Newmsg(128,"highlight", "off; bar=#b; tag=#S", 'T', "CLOSE!"));
16566         }
16567         if (node->window.pid)
16568             Put(DIRECT,node->window.pid,Newmsg(32,"keys?",NULL));
16569         map->last_active = node;
16570     }
16571 }
16572
16573 move mark(row,col,picture)
16574 register short row,col;
16575 register CONNECTOR *picture;
16576 {
16577     Put(DIRECT,picture->pid,Newmsg(32,"mark", "at=#2s",row,col));
16578 }
16579
16580

```

```

1681 clip_window(node)
1682 register MAPNODE *node;
1683 {
1684     register MAPNODE *temp;
1685     register short prio= 127, count, *count_addr, *n;
1686     char *m;
1687     for ( ; node = node->pre)
1688     {
1689         m = Newmsg(1000,"cut" "inHI=#s#A",prio--0,950, NULL);
1690         count_addr = (short *) (Find_triple(m,"inHI",0,NULL,0,NULL) + 2);
1691         n = count_addr + 1;
1692         count = 0;
1693         for (temp = node->pre; temp; temp = temp->pre)
1694         {
1695             *n++ = temp->row;
1696             *n++ = temp->col;
1697             *n++ = temp->out_ht;
1698             *n++ = temp->out_wd;
1699             count++;
1700         }
1701         *count_addr = count;
1702         Put(DIRECT,node->window.pld,m);
1703     }
1704 }
1705
1706 MAPNODE *find_window(map,window,row,col)
1707 register LIST *map;
1708 register WINDOW *window;
1709 register short row, col;
1710 {
1711     register MAPNODE *node;
1712     for (node = map->first; node; node = node->nxt)
1713     {
1714         query_window(window,node->window,row,col);
1715         if (window->area != 'N')
1716             break;
1717     }
1718     window->previous = window->node;
1719     return(window->node == node);
1720 }
1721
1722 }
1723

```

```

17254 query window(window, conn, row, col)
17255 register WINDOW *window;
17256 CONNECTOR conn;
17257 register short row, col;
17258 {
17259     register char *p, *reply;
17260     if (window->hdr)
17261         Free(window->hdr);
17262     window->hdr = NULL;
17263     window->elem row = window->elem col = -1;
17264     reply = CallDIRECT conn.pid, Newmsg(64, "w", "inHI=#2s", row, col), 0, 0);
17265     p = FindTriple(reply, "inHI", 0, none, 1, NULL);
17266     p += 2 * sizeof(short);
17267     window->area = *p++;
17268     window->bar = *p++;
17269     window->row = *p;
17270     window->col = *p;
17271     long align(p);
17272     if (*(short*)p)
17273     {
17274         window->hdr = (P_E_HDR *) Alloc(*(short*)p, YES);
17275         memcpy(window->hdr, p, *(short*)p);
17276     }
17277     Free(reply);
17278 }
17279
17280 MAPNODE *new_node(map, name)
17281 register LIST *map;
17282 register char *name;
17283 {
17284     register MAPNODE *node = NULL;
17285     register short i;
17286     for (i = POOL_SIZE, node = map->pool; node->pool && i; ++node, --i)
17287     {
17288         if (node == (MAPNODE *) Alloc(sizeof(MAPNODE), YES);
17289             node->pool = i;
17290             strcpy(node->name, name);
17291             return(node);
17292         }
17293     }
17294 }

```

```

1767 free node(node) *node;
1768 register MAPNODE
1769 {
1770     if (node->pool) = NULL;
1771     else
1772         Free(node);
1773 }
1774
1775 map after(node, pred, map)
1776 register MAPNODE *node, *pred;
1777 register LIST
1778 {
1779     if (pred)
1780     {
1781         node->nxt = pred->nxt;
1782         node->pre = pred;
1783         if (pred->nxt)
1784             (pred->nxt)->pre = node;
1785         pred->nxt = node;
1786     }
1787     else
1788     {
1789         if (node->nxt == map->first)
1790             (map->first)->pre = node;
1791         node->pre = NULL;
1792     }
1793     if (node->pre)
1794         map->first = node;
1795     if (node->nxt)
1796         map->last = node;
1797     ++map->count;
1798 }
1799
1800 unmap(node, map)
1801 register MAPNODE *node;
1802 register LIST
1803 {
1804     if (node->pre)
1805         (node->pre)->nxt = node->nxt;
1806     else
1807         map->first = node->nxt;
1808     if (node->nxt)
1809         (node->nxt)->pre = node->pre;
1810     else
1811         map->last = node->pre;
1812     --map->count;
1813 }
1814
1815

```



```

1817 remap(window,node,new_picture,map,sel)
1818 register CONNECTOR --*window,*new_picture;
1819 register MAPNODE *node;
1820 register SELECTION *sel;
1821 LIST *map;
1822 {
1823     if (window)
1824         for (node = map->first;
1825              node && window->pid != node->window.pid; node = node->nxt) ;
1826     if (node)
1827     {
1828         end edit(node,'X',0,0,NULL);
1829         if (new_picture && new_picture->pid != node->picture.pid)
1830             if (node == sel->map)
1831             {
1832                 sel->map = NULL;
1833                 sel->pehdng = NO;
1834             }
1835         node->picture = *new_picture;
1836     }
1837 }

```

```

18338 align_window(screen, node)
18339 register SCREEN *screen;
18340 register MAPNODE *node;
18341 {
18342     register short temp;
18343     if (screen->char_align)
18344     {
18345         if (node->tight)
18346         {
18347             temp = ((node->row * VCHAR_HT) | (node->outer ? VCHAR_HT : 0));
18348             node->row = (node->row / VCHAR_HT) * VCHAR_HT + node->outer;
18349             temp = ((node->col * VCHAR_WD) | (node->outer ? VCHAR_WD : 0));
18350             node->col = (node->col / VCHAR_WD) * VCHAR_WD + node->outer;
18351             node->col = (node->col / VCHAR_WD) * VCHAR_WD + temp;
18352             node->col = (node->outer + node->width / 200) + temp;
18353         }
18354     }
18355     else
18356     {
18357         node->row = ((node->row + VCHAR_HT-1) / VCHAR_HT) * VCHAR_HT;
18358         node->col = ((node->col + VCHAR_WD-1) / VCHAR_WD) * VCHAR_WD;
18359     }
18360     if (node->row < screen->meta_row)
18361         node->row += (screen->meta_row - node->row) / VCHAR_HT * VCHAR_HT;
18362     if (node->col < screen->meta_col)
18363         node->col += (screen->meta_col - node->col) / VCHAR_WD * VCHAR_WD;
18364     if (node->out_ht > screen->meta_ht)
18365         node->out_ht = (node->out_ht - (node->top + node->bottom));
18366     if (node->out_wd > screen->meta_wd)
18367         node->out_wd = (node->left + node->right);
18368     if (node->tight)
18369     {
18370         temp = (node->height * VCHAR_HT ? VCHAR_HT : 0) / VCHAR_HT + temp;
18371         node->height = (node->height / VCHAR_HT) * VCHAR_HT + temp;
18372         temp = (node->width * VCHAR_WD ? VCHAR_WD : 0) / VCHAR_WD + temp;
18373         node->width = (node->width / VCHAR_WD) * VCHAR_WD + temp;
18374     }
18375     node->out_ht = node->height + node->top + node->bottom;
18376     node->out_wd = node->width + node->left + node->right;
18377 }

```

```

1879 status(msg, size) *msg;
1880 register char size;
1881 {
1882     register char *m;
1883     *m = Alloc(size, YES) = NULL;
1884     strcat(m, "Find triple(msg, \"orig\", size, none, 1, NULL));
1885     strcat(m, "Find triple(msg, \"stat\", size, none, 1, NULL));
1886     strcat(m, "Find triple(msg, \"req \", size, none, 1, NULL));
1887     Note(m, "ERROR");
1888     Free(m);
1889 }
1890
1891 reply status(req, mid, stat, code)
1892 register char *req, *mid, *stat;
1893 register long *code;
1894 {
1895     register char *type, *msg;
1896     type = "failed";
1897     if (!mid) type = "status";
1898     else if (*mid == '-')
1899         mid++;
1900     else if (*mid == '+')
1901         type = "done";
1902         mid++;
1903     msg = Newmsg(strlen(stat)+100, type,
1904                 "orig=#S; stat=#S; code=#I", "console", stat, code);
1905     if (mid)
1906     {
1907         Append triple(msg, "req ", strlen(mid)+1, mid);
1908         Reply(req, msg);
1909     }
1910     else
1911         Put(DIRECT, (long) req, msg);
1912 }
1913
1914 info(dialogue, string, window)
1915 CONNECTOR dialogue, window;
1916 register char *string;
1917 {
1918     Put(DIRECT dialogue pid, Newmsg(strlen(string)+100, "info",
1919                                     "text=#S; near=#C; wait=#S", string, &window, 5));
1920 }
1921
1922
1923
1924
1925
1926
1927
1928
1929

```

PROGRAM LISTING B

```

9  Module          : %M% %I%
10 Date submitted : %E% %U%
11 Author         : Frank Kolnlick
12 Origin        : CX Picture Manager
13 Description
14
15 *****
16 #ifndef lint
17 #endif
18 #define char SrcId[] = "%Z% %M%: %I%";
19 /* Picture manager: global data */
20
21 #include <CX.h>
22 #include <MEmory.h>
23 #include <string.h>
24 static long none = 0;
25
26 typedef struct element_node
27 {
28     struct element_node *next;
29     struct element_node *pre;
30     unsigned char changed;
31     unsigned char marked;
32     unsigned char pool;
33     unsigned char length;
34     short *NOTE: 'length must start on a long-word boundary
35     } ELEMENT;
36
37 typedef struct current_state
38 {
39     char *msg;
40     long sender;
41     long size;
42     short appl;
43     short row, appl_col;
44     CONNECTOR *marker;
45     char *old mark;
46     char *erase mark;
47     unsigned char display mark;
48     unsigned char check;
49     unsigned char debug;
50     unsigned char height;
51     unsigned char name[32];
52     long file[64];
53     long status;
54     long code;
55     long status_string;
56     } CURRENT;
57
58 *****
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

61 typedef struct view_node
62 {
63     struct view_node *nxt;
64     CONNECTOR owner;
65     short row, col;
66     short height, width;
67 } VIEW;
68
69
70
71 typedef struct appl_node
72 {
73     struct appl_node *nxt;
74     long name;
75     CONNECTOR conn;
76     short row, col;
77 } APPL;
78
79
80 typedef struct anim_node
81 {
82     struct anim_node *nxt;
83     long name;
84     CONNECTOR conn;
85 } ANIM;
86
87 typedef struct affected_area
88 {
89     short r1, c1;
90     short r2, c2;
91     char col_or;
92     short pattern;
93     short max_height;
94     short max_width;
95     short height, width;
96 } AREA;
97
98
99 typedef struct lists
100 {
101     ELEMENT *first;
102     ELEMENT *last;
103     ELEMENT *current;
104     VIEW *views;
105     APPL *appls;
106     ANIM *anims;
107     int changes;
108     int eflags;
109     struct
110     {
111         long n;
112         long size;
113         ELEMENT *ptr;
114     } pool;
115 } LIST;

```

```

/* links viewports: */
/*
/* ->next node */
/* owner of viewport */
/* start of viewport */
/* extent */
/*
/* links applications: */
/*
/* ->next node */
/* name of application */
/* conn to application */
/* origin */
/*
/* links animation processes: */
/*
/* ->next node */
/* name of element */
/* conn. to process */
/*
/* area changed by a request: */
/*
/* upper left front */
/* lower right back */
/* background color */
/* background pattern */
/* max. height */
/* max. width */
/* current size */
/*
/* list pointers, etc.: */
/*
/* ->pict. element list */
/* ->end of p.e. list */
/* ->last p.e. changed */
/* ->viewport list */
/* ->applications list */
/* ->animation list */
/* ->changes in list */
/* ->erasures in list */
/* ->picture elements */
/* element pool descr.: */
/*
/* #elements */
/* size of elements */
/* ->element buffer */

```

```

117 /* local functions */
118 char *value(), *tag();
119 ELEMENT *mark_number(); *mark_area(); *mark_elements(); *new_element();
120 p__E_HDR *first_macro(); *next_macro();
121
122 /* Picture manager: main-line */
123 PROCESS(Picture)
124 {
125     CURRENT cur;
126     AREA area;
127     LIST list;
128     register VIEW *view;
129     register ANIM *anim;
130
131     Set event key("picture mgr.");
132     init_pm(&cur, &area, &list);
133     draw_picture(&cur, &area, &list);
134     for (view = list.views; view; view = view->nxt)
135         put(DIRECT, view->owner.pid, Newmsg(32, "unmap", NULL));
136     for (anim = list.anims; anim; anim = anim->nxt)
137         put(DIRECT, anim->conn.pid, Newmsg(32, "quit", NULL));
138     Exit();
139 }
140
141
142 init_pm(cur, area, list)
143 register CURRENT *cur;
144 register AREA *area;
145 register LIST *list;
146 {
147     area->color = BLACK;
148     area->pattern = 0;
149     *cur->hname = *cur->file = NULL;
150     area->max_height = area->max_width = 0;
151     list->current = list->first = list->last = NULL;
152     list->views = NULL;
153     list->apps = NULL;
154     list->anim = NULL;
155     cur->size = list->pool.n = 0;
156     cur->debug = cur->check = cur->private = cur->display_mark = NO;
157     cur->mark = cur->old_mark = cur->erase_mark = NULL;
158 }

```

```

160 draw_picture(cur, area, list)
161 CURRENT *cur;
162 register AREA *area;
163 register LIST *list;
164 {
165     register char *msg;
166     register short transaction = 0, result = 0, go = YES;
167     register ELEMENT element;
168     long status[list], list_size = 0, *req = NULL;
169     while (go)
170     {
171         cur->msg = msg = Get(0, &cur->sender, &cur->size);
172         if (!transaction)
173         {
174             list->changes = list->erases = area->r2 = area->c2 = 0;
175             area->r1 = area->c1 = 32767;
176             cur->appl = NULL;
177             if (!list->appls)
178                 check_appl(cur, list->appls);
179         }
180         if (*msg == '|' && transaction < 10)
181             status[transaction] = 0;
182         else if (*msg == '|')
183             --transaction;
184         else
185             go = Request(cur, area, list, msg, cur->size, cur->appl);
186         if (!transaction)
187         {
188             if (list->changes)
189                 notify(cur, area, list);
190             for (element = list->first; element; element = element->nxt)
191             {
192                 element->changed = element->marked = NO;
193                 if (element->deleted && !Any_msg(NULL))
194                     delete_element(list, element);
195             }
196             if (Find_triple(msg, "rply", cur->size, NO, 0, NULL) && result >= 0)
197                 reply_status(msg, msg, "completed", result);
198             free_requests(msg, cur->size, &req, &list_size);
199         }
200     }
201 }
202 }
203 }

```

```

204 check appl(cur,appl) *cur;
205 register CURRENT *appl;
206 register APPL
207 {
208     for ( ; appl && (appl->conn.pid != cur->sender.pid); appl = appl->nxt);
209     {
210         if (!(cur->appl = appl->name))
211             cur->appl = -1;
212         cur->appl_row = appl->row;
213         cur->appl_col = appl->col;
214     }
215 }
216
217 free requests(msg, size, req, list_size)
218 register Char *msg, **req;
219 register long size, *list_size;
220 {
221     register char *temp, *next;
222     if (msg)
223     {
224         *(char**)msg = *req;
225         *req = msg;
226         *list_size += size;
227         if (!any(msg(NULL) || *list_size > 1000)
228             for (temp = *req; *req = NULL; *list_size = 0; temp = next)
229                 next = *(char**)temp;
230             Free(temp);
231         )
232     }
233 }
234
235 Request(cur, area, list, msg, size, appl)
236 register CURRENT *cur;
237 register AREA *area;
238 register LIST *list;
239 register long msg, size, appl;

```



```

245 register short go = YES;
246 if (!strcmp(msg, "write"))
247     Draw(!strcmp(msg, "size"), "write");
248 else if (!strcmp(cur, area, list))
249     Edit(!strcmp(msg, "mark"), msg, size, appl);
250 else if (!strcmp(msg, "mark"), list);
251 else if (!strcmp(msg, "up"), list);
252 else if (!strcmp(msg, "down"), list);
253 else if (!strcmp(msg, "move"), appl);
254 else if (!strcmp(msg, "erase"), appl);
255 else if (!strcmp(msg, "size"), appl);
256 else if (!strcmp(msg, "read"), list);
257 else if (!strcmp(msg, "replace"), list);
258 else if (!strcmp(msg, "change"), appl);
259 else if (!strcmp(msg, "animate"), appl);
260 else if (!strcmp(msg, "alter"), list);
261 else if (!strcmp(msg, "cancel"))
262     Alter(!strcmp(msg, "alter"), list);
263 else if (!strcmp(msg, "number"), list);
264 else if (!strcmp(msg, "mark?"), list);
265 else if (!strcmp(msg, "mark?"), list);
266 else if (!strcmp(msg, "save"), list);
267 else if (!strcmp(msg, "set"), list);
268 else if (!strcmp(msg, "restore"), list);
269 else if (!strcmp(msg, "area"), list);
270 else if (!strcmp(msg, "bkgd"), list);
271 else if (!strcmp(msg, "size"), list);
272 else if (!strcmp(msg, "create"), list);
273 else if (!strcmp(msg, "inif"), list);
274 else if (!strcmp(msg, "open"), list);
275 else if (!strcmp(msg, "quit"), list);
276 else if (!strcmp(msg, "quit"), list);
277 else if (!strcmp(msg, "quit"), list);
278 else if (!strcmp(msg, "quit"), list);
279 else if (!strcmp(msg, "quit"), list);
280 else if (!strcmp(msg, "quit"), list);
281 else if (!strcmp(msg, "quit"), list);
282 else if (!strcmp(msg, "quit"), list);
283 else if (!strcmp(msg, "quit"), list);
284 else if (!strcmp(msg, "quit"), list);
285 else if (!strcmp(msg, "quit"), list);
286 else if (!strcmp(msg, "quit"), list);
287 else if (!strcmp(msg, "quit"), list);
288 else if (!strcmp(msg, "quit"), list);
289 else if (!strcmp(msg, "quit"), list);
290 else if (!strcmp(msg, "quit"), list);
291 else if (!strcmp(msg, "quit"), list);
292 else if (!strcmp(msg, "quit"), list);

```

```

294 else if (!strcmp(msg, "query"))
295     Query(cur, list);
296 else if (!strcmp(msg, "failed"))
297     Status(msg, size);
298 else if (!strcmp(msg, "done") || !strcmp(msg, "status"))
299     else if (!Change_attribute(list, msg, size, appl))
300     {
301         if (!strcmp(msg, "view"))
302             Viewport(cur, area, list);
303         else if (!strcmp(msg, "debug"))
304             cur->debug = !cur->debug;
305         else
306             reply_status(msg, "-\unknown\", msg, 0);
307     }
308     return(go);
309 }
310 )

```

```

311 Change attribute(list,msg,size,appl)
312 register LIST *list, size, appl;
313 register long msgids[] = "select\oblink\invert\ohide\ohighlight\0";
314
315 static char
316     *p;
317     new_state, changed, type;
318     *element;
319     *hdr;
320     *p;
321     *p && strcmp(msg,p); p += strlen(p)+1, ++type);
322
323 if (return(NO);
324
325 list->current = element = mark_elements(list,NULL,NULL,msg,size,appl);
326 new_state = (short)Find_triple(msg,"of",size,NO,0,NULL);
327 for (element = element->nxt; element->marked;
328     element = element->nxt)
329     {
330         hdr = (P E HDR *) &element->length;
331         switch (type)
332         {
333             case 0:
334                 changed = hdr->attr.selected || new_state;
335                 if (hdr->attr.selected == new_state)
336                     put("NEXT", "Console", Newmsg(hdr->length+50
337                         "write", "data=fe", type=#C, hdr,NULL, 'P'));
338                 break;
339                 changed = hdr->attr.blink != new_state;
340                 hdr->attr.blink = new_state;
341                 break;
342                 changed = hdr->attr.invert != new_state;
343                 hdr->attr.invert = new_state;
344                 break;
345                 changed = hdr->attr.hidden != new_state;
346                 hdr->attr.hidden = new_state;
347                 break;
348                 changed = hdr->attr.highlight != new_state;
349                 hdr->attr.highlight = new_state;
350
351         if (element->changed == changed)
352             list->changed++;
353         element->marked = NO;
354     }
355     return(YES);
356 }

```

```

357 Query(cur, list) *cur;
358 CURRENT *list;
359 register LIST *list;
360 {
361     unsigned unsigned;
362     register unsigned min_r = 65535;
363     register unsigned max_r = 0;
364     register ELEMENT *element;
365     register P E HDR *hdr;
366     register VIEW *view;
367     n_elem = 0;
368     p_views = 0;
369     min_c = 65535;
370     max_c = 0;
371     pic_ht = 0;
372     pic_wd = 0;
373     for (element = list->first; element = element->nxt;
374         {
375             hdr = (P E HDR *) &element->length;
376             if (hdr->row < min_r) min_r = hdr->row;
377             if (hdr->col < min_c) min_c = hdr->col;
378             if (hdr->row + hdr->height > max_r) max_r = hdr->row + hdr->height;
379             if (hdr->col + hdr->width > max_c) max_c = hdr->col + hdr->width;
380             n_elem++;
381         }
382     if (n_elem)
383     {
384         pic_ht = max_r - min_r;
385         pic_wd = max_c - min_c;
386     }
387     else
388     {
389         pic_ht = pic_wd = max_r = max_c = min_r = min_c = 0;
390         view = list->views;
391         for (view = list->views; view = view->nxt; n_views++;
392             Reply Newmsg(256, "status", "orig=#s; size=#2s; high=#2s; low=#2s; cnt=#s; \
393                 view=#s; name=#s; file=#s", picture
394                 pic_ht, pic_wd, min_r, min_c, max_r, max_c, n_elem, n_views,
395                 cur->name, cur->file));
396     }
397     Query number(list, msg, size, appl)
398     register LIST *list;
399     register long msg, size, appl;
400     register unsigned n = 0;
401     register ELEMENT *element, *temp;
402     if (element = mark_elements(list, NULL, NULL, msg, size, appl))
403     {
404         for (temp = list->first; temp != element; temp = temp->nxt, n++);
405         Reply("msg, Newmsg(element->length+32, "number",
406             n, element->length));
407     }
408     else
409     {
410         reply_status(msg, "-number", "too high", 0);
411     }

```

```

409 Draw(list,msg,size) *list;
410 register long msg, size;
411 {
412     register ELEMENT *after;
413     register long
414     if (p = (long *) Find_triple(msg,"data",size,NULL,4,NULL))
415     {
416         if (Find_triple(msg,"back",size,NO,0,NULL))
417         else
418         else after = list->last;
419         if (!draw_elements(p,*{p-l}list,after))
420         if (reply_status(msg,"-write","bad length/type/macro",0);
421     }
422     else
423     reply_status(msg,"-write","missing \ 'data\'",0);
424 }
425
426 draw_elements(p,list,len,list,after)
427 register char
428 register long list, len;
429 register LIST *list;
430 register ELEMENT *after;
431 {
432     register ELEMENT *element;
433     register short length, number = 0;
434     while ((length = *(short *) p)
435             && (list_len == length) >= 0
436             && strchr("lreadsmn", ((P_E_HDR*)p)->type))
437     {
438         if ((P_E_HDR*)p)->type == 'm' && !check_macro(p))
439         element = new_element(list,length+sizeof(ELEMENT),after);
440         memcpy(&element->length,p,length);
441         if ((P_E_HDR*)p)->height)
442             number++;
443             p += length;
444             Long_align(p);
445     }
446     list->size += number;
447     list->changes += number;
448     list->current = element;
449     return(length ? NO : YES);
450 }
451
452
453
454
455
456
457

```

```

458 define box(hdr)
459 register P_E_HDR *hdr;
460 {
461     register char *val;
462     val = value(hdr);
463     if (hdr->type == 't')
464     {
465         hdr->height = VCHAR_HHT;
466         hdr->width = VCHAR_WD * strlen(val+8);
467     }
468     else if ((hdr->type == 'n') || (hdr->type == 'm'))
469     {
470     }
471 }
472
473 check_macro(hdr)
474 register P_E_HDR *hdr;
475 {
476     register P_E_HDR *temp, *first;
477     short len;
478     char *p, macro_type;
479     for (first = temp = first_macro(hdr, &macro_type, &len, &p); temp;
480         (
481             if (macro_type == 'L')
482             if (temp->attr.hidden == YES;
483             if (temp->height
484                 define_box(temp);
485             if (macro_type == 'L')
486             if (first->attr.hidden == NO;
487             return(p ? YES : NO);
488         )
489     }
490 }
491 }
492

```

```

493 P E HDR *first macro(hdr, type, len, p)
494 register P_E_HDR *hdr;
495 register char *type;
496 register short *len;
497 register char **p;
498 {
499     register P_E_HDR *temp;
500     *p = value(hdr);
501     if (*type == **p)
502         long_align(*p);
503     *temp = (P_E_HDR *) *p;
504     *len = hdr->length && temp->length < *len;
505     if (temp->length && strchr("tlireadsmn", temp->type))
506         *p = NULL;
507     return(NULL);
508 }
509
510 P E HDR *next macro(len, p)
511 register short *len;
512 register char **p;
513 {
514     register P_E_HDR *temp;
515     if (*p)
516     {
517         temp = (P_E_HDR *) *p;
518         *p = temp->length;
519         long_align(*p);
520         *len = (P_E_HDR *) *p;
521         if (temp->length && strchr("tlireadsmn", temp->type))
522             else *p = NULL;
523         return(temp);
524     }
525     return(NULL);
526 }
527
528 }
529
530
531
532
533
534
535

```

```

536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

5337 Replace(area, list, msg, size, appl)
5338 *ayed;
5339 *list;
5340 register long msg, size, appl;
5341 {
5342     register char *p;
5343     register short length = 0;
5344     register ELEMENT *temp;
5345     register P_E_HDR *hdr;
5346     register long list_len;
5347     ELEMENT *after = NULL;
5348
5349     if (Find_triple(msg, "\0\0\0\0", size, NO, 0, NULL))
5350     {
5351         Erase(area, list, msg, size, appl);
5352         after = list->current;
5353     }
5354     if (p = Find_triple(msg, "data", size, NULL, 1, NULL))
5355     {
5356         list_len = *((long *) (p-4));
5357         while ((length = *(short *) p) && (list_len -= length) > 0)
5358         {
5359             hdr = (P_E_HDR *) p;
5360             if (hdr->type == 'M' && !check_macro(hdr))
5361                 break;
5362             temp = list->last;
5363             for (temp = list->length; temp &&
5364                 ((P_E_HDR *) temp->length)->row != hdr->row &&
5365                 ((P_E_HDR *) temp->length)->col != hdr->col; temp = temp->pre);
5366             if (temp)
5367             {
5368                 temp_hdr = (P_E_HDR *) &temp->length;
5369                 temp=>deleted = YES;
5370                 after = temp->pre;
5371             }
5372             draw_elements(hdr, length, list, after);
5373             if (temp_hdr && (hdr->type != 'I' ||
5374                 | | hdr->width != temp_hdr->width))
5375             {
5376                 change_area(area, temp_hdr->row, temp_hdr->col,
5377                     temp_hdr->height, temp_hdr->width);
5378                 list->erases++;
5379             }
5380             p += length;
5381             Long_align(p);
5382         }
5383     }
5384     if (length)
5385         reply_status(msg, "-replace", "bad length/type/macro", 0);

```



```

586 Erase(area, list, msg, size, appl)
587 AREA *area;
588 register LIST *list;
589 register long msg, size, appl;
590 {
591     register ELEMENT *element = NULL;
592     register P_E_HDR *hdr;
593     int number;
594     if (element = mark_elements(list, NULL, &number, msg, size, appl))
595     {
596         list->current = element->pre;
597         for ( ; element = element->nxt)
598             if (element->marked)
599                 {
600                     element->deleted = YES;
601                     hdr = (P_E_HDR*) &element->length;
602                     change_area(area, hdr->row, hdr->col, hdr->height, hdr->width);
603                 }
604             list->erases += number;
605             list->changes += number;
606         }
607     }
608 }
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632

```

```

COPY(cur, area, list, msg, size, appl)
CURRENT *cur;
register AREA *area;
register LIST *list;
register long msg, size, appl;
{
    register ELEMENT *element;
    register short bkgd, *p;
    short *q;
    unsigned int length = 0;
    if (bkgd = (short) Find_triple(msg, "bkgd", size, NO, 0, NULL))
    {
        p = (short *) Find_triple(msg, "qpos", size, &none, 0, NULL);
        q = (short *) Find_triple(msg, "qend", size, &none, 0, NULL);
        change_area(area, *p, *(p+1), *q, *(q+1) - *(p+1));
    }
    if ((element = mark_elements(list, &length, NULL, msg, size, appl)) || bkgd)
    else Reply(msg, Newmsg(64, "write", NULL));
}

```

```

633 Move(area, list, msg, size, appl)
634 AREA *area;
635 LIST *list;
636 long msg, size, appl;
637 {
638     register ELEMENT *element;
639     register P E HDR *hdr;
640     register int_ delta_row, delta_col, by_offset = NO, row = 0, col = 0;
641     register int_ *p;
642     register char n;
643
644     if (p = Find_triple(msg, "by ", size, NULL, 4, NULL))
645     {
646         by_offset = YES;
647         delta_row = *((short *) p)++;
648         delta_col = *((short *) p);
649     }
650     else if (p = Find_triple(msg, "to ", size, NULL, 4, NULL))
651     {
652         row = *((short *) p)++;
653         col = *((short *) p);
654     }
655     if (list->current = element = mark_elements(list, NULL, &n, msg, size, appl))
656     {
657         if (!by_offset)
658         {
659             hdr = (P E HDR *) &element->length;
660             delta_row = row - hdr->row;
661             delta_col = col - hdr->col;
662         }
663         for (; element; element = element->nxt)
664         {
665             if (element->marked)
666             {
667                 hdr = (P E HDR *) &element->length;
668                 change_area(area, hdr->row, hdr->col, hdr->height, hdr->width);
669                 hdr->row += delta_row;
670                 hdr->col += delta_col;
671                 element->changed = YES;
672                 element->marked = NO;
673                 element->deleted = (hdr->row < 0 || hdr->col < 0);
674             }
675             list->changes += n;
676             list->erases += n;
677         }
678     }

```

```

679 Change(area, list, msg, size, appl)
680 register AREA *area;
681 register LIST *list;
682 register long msg, size, appl;
683 {
684     register ELEMENT *element = NULL;
685     register P_E_HDR *hdr;
686     char *color, *bkgd, *fill, *pat;
687
688     color = Find_triple(msg, "color", size, NULL, 1, NULL);
689     bkgd = Find_triple(msg, "bkgd", size, NULL, 1, NULL);
690     fill = Find_triple(msg, "fill", size, NULL, 1, NULL);
691     pat = Find_triple(msg, "pat", size, NULL, 1, NULL);
692     if (list->current == element) = mark_elements(list, NULL, NULL, msg, size, appl)
693     for ( ; element; element = element->nxt)
694     {
695         if (element->marked)
696         {
697             hdr = (P_E_HDR*) &element->length;
698             if (color)
699                 hdr->color = *color;
700             if (bkgd)
701                 hdr->bkgrnd = *bkgd;
702             if (fill)
703                 hdr->fill = *fill;
704             if (pat)
705                 hdr->pattern = *pat;
706             change_area(area, hdr->row, hdr->col, hdr->height, hdr->width);
707             list->changes++;
708         }
709     }
710
711 Background(area, list, msg, size)
712 register AREA *area;
713 register LIST *list;
714 register long msg, size;
715 {
716     area->color = *Find_triple(msg, "color", size, &area->color, 1, NULL);
717     area->pattern = *Find_triple(msg, "pat", size, &area->pat, 1, NULL);
718     change_area(area, 0, 0, MAX_ROW, MAX_COL);
719     list->changes = list->erases = 1;
720 }

```

```

721 New picture(cur_area, list)
722 register CURRENT *cur;
723 register AREA *area;
724 register LIST *llist;
725
726
727 register ELEMENT *element;
728 register long max, maxe;
729 short def_maxe = 100;
730 char def_bkgd = BLACK, -def_pat = 0;
731
732 for (element = list->first; element; element->nxt)
733   element->deleted = YES;
734   list->current = list->last = NULL;
735   list->changes = list->erases = list->size = 0;
736   if (Find_triple(cur->msg, "file", cur->size, NO, 0, NULL))
737     return(Old_picture(cur, llist));
738   else
739   {
740     cur->owner = cur->sender;
741     strcpy(cur->name, Find_triple(cur->msg, "name", cur->size, &none, 1, NULL));
742     area->max_height = Find_triple(cur->msg, "size", cur->size, &none, 4, NULL);
743     area->short * Find_triple(cur->msg, "size", cur->size, &none, 4, NULL);
744     area->max_width = Find_triple(cur->msg, "size", cur->size, &none, 4, NULL);
745     area->color = Find_triple(cur->msg, "bkgd", cur->size, &none, 4, NULL);
746     area->pattern = Find_triple(cur->msg, "pat", cur->size, &def_pat, 1, NULL);
747     cur->highlight = Find_triple(cur->msg, "high", cur->size, &none, 1, NULL);
748     cur->check = (area->max_height != 0);
749     max = (* (short *) Find_triple(cur->msg, "max", cur->size, &def_maxe, 2, NULL)) + 1;
750     maxe = (* (short *) Find_triple(cur->msg, "maxe", cur->size, &def_maxe, 2, NULL));
751     if (maxe & 1)
752       ++maxe;
753     llist->pool.n = max;
754     llist->pool.size = maxe + sizeof(ELEMENT) + 10;
755     llist->pool.ptr = (ELEMENT *) Alloc(max * llist->pool.size, YES);
756     memset(llist->pool.ptr, 0, max * llist->pool.size);
757     change_area(area, 0, 0, MAX_ROW, MAX_COL);
758     llist->changes = llist->erases = 1;
759     reply_status(cur->msg, "+create", "complete", 0);
760     return(YES);
761   }
762 }
763
764

```

```

765 old_picture(cur_list)
766 register CURRENT *cur;
767 register LIST *list;
768
769
770 register char *p = (char*)1;
771 CONNECTOR file;
772
773 strcpy(cur->name, Find_triple(cur->msg, "name", cur->size, &none, 1, NULL));
774 strcpy(cur->file, Find_triple(cur->msg, "file", cur->size, cur->name, 1, NULL));
775 if (*cur->file)
776 {
777     if (Connect to(NEXT "File mgt" Newmsg(64, "open"
778         ("name=#s; omod=#s; amod=#s", cur->file, "R", NULL), &file))
779     {
780         cur->owner = cur->sender;
781         while (p)
782             if (p = Call(DIRECT file.pid
783                 Newmsg(64, "read", "conh=#c; size=#l", &file, -1, 0, 0))
784                 if (p, = Find_triple(p, "data", 0, NULL, 4, NULL, 1, 0))
785                     draw_elements(p, *(long*)(p-4), list, NULL);
786                 Put(DIRECT file.pid, Newmsg(32, "close", "conh=#c", &file));
787                 reply_status(cur->msg, "topen", "complete", 0);
788                 return(YES);
789             }
790         else
791             reply_status(cur->msg, "-open", "can't open file", 0);
792     }
793     else
794         reply_status(cur->msg, "-open", "no file name", 0);
795     return(NO);
796 }

```

```

797 save_picture(cur, list)
798 CURRENT *cur;
799 LIST;
800 {
801     register char *file_name, *m, *p;
802     register ELEMENT file;
803     CONNECTOR length = 0, num;
804     unsigned int
805     if (!(file_name = Find_triple(cur->msg, "file", cur->size, NULL, 1, HULL)))
806     if (!(file_name = cur->file;
807         mark_element =
808         (
809             if (!Connect to(NEXT, "File mgt", Newmsg(64, "open"
810                 "name=#S; omod=#S; amod=#S", file_name, &file))
811                 Connect to(NEXT, "File mgt", Newmsg(64, "create"
812                     "name=#S; omod=#S; amod=#S", file_name, "W", HULL), &file);
813             if (file.pid)
814             {
815                 num = length + 4 * num + 4;
816                 m = Newmsg(num+50, "write", &conn=#C; data=#A", &file, num, HULL);
817                 p = Find_triple(m, "data", 0, NULL, 1, HULL);
818                 for {f; element; element = element->nxt)
819                 {
820                     memcpy(p, element, element->length);
821                     p += element->length;
822                     Long_align(p);
823                 }
824                 *(short *) p = NULL;
825                 Put(DIRECT, file.pid, m);
826                 reply_status(cur->msg, "close", "conn=#C", &file);
827                 reply_status(cur->msg, "save", "picture saved", 0);
828             }
829             else
830                 reply_status(cur->msg, "--save", "can't open/create file", 0);
831         }
832     else
833         reply_status(cur->msg, "--save", "no elements", 0);
834     else
835         reply_status(cur->msg, "--save", "no file name", 0);
836 }
837
838
839
840
841

```

```

8442 APPL(cur, list) *cur;
8443 CURRENT LIST *list;
8444 register APPL *appl;
8445 register long name;
8446 register short *p;
8447
8448 name = *(long *) Find_triple(cur->msg, "name", cur->size, &none, 4, NULL);
8449 for (appl = list->appls; appl && appl->name != name; appl = appl->nxt);
8450 if (!appl)
8451     appl = (APPL *) Alloc(sizeof(APPL), YES);
8452 appl->comp = cur->sender;
8453 p = (short *) Find_triple(cur->msg, "org", cur->size, &none, 2, NULL);
8454 appl->row = *p++;
8455 appl->col = *p;
8456 appl->name = name;
8457 appl->conn =
8458     *(CONNECTOR *) Find_triple(cur->msg, "appl", cur->size, &none, 4, NULL);
8459 appl->nxt = list->appls;
8460 list->appls = appl;
8461
8462 )
8463
8464 Move mark(cur_area, list)
8465 register CURRENT *cur;
8466 register AREA *area;
8467 register LIST *l1st;
8468
8469 {
8470     register p E HDR *bhdr;
8471     register short *pos;
8472     char *q;
8473
8474     if (pos = (short *) Find_triple(cur->msg, "at", cur->size, NULL, 4, NULL))
8475     {
8476         if (cur->mark)
8477             erase_mark(cur_area);
8478         else
8479         {
8480             q = cur->mark = Alloc(sizeof(P E HDR)+30, YES);
8481             draw_line(&q, 0, 0, VCHAR_HT, 0, NULL, YELLOW, 'S', 0, 1, NULL);
8482         }
8483         bhdr = (P E HDR *) cur->mark;
8484         bhdr->row = *pos++;
8485         bhdr->col = *pos;
8486         cur->display mark = YES;
8487         list->changeest++;
8488     }
8489 }

```

```

891 Query mark(CUR) *cur;
892 register CURRENT *cur;
893 {
894     register P_E_HDR *hdr;
895     if (hdr == (P_E_HDR *) cur->mark)
896         Reply(cur->msg, Newmsg(64, "mark", "at=#2s", hdr->row, hdr->col));
897     else
898         reply_status(cur->msg, "--mark?", "no mark defined", 0);
899 }
900
901 Set mark(cur, area, list)
902 {
903     register CURRENT *cur;
904     register AREA *area;
905     register LIST *list;
906     register P_E_HDR *hdr;
907     if ((hdr = (P_E_HDR *) Find_triple(cur->msg, "data", cur->size, NULL, 1, NULL))
908         && hdr->length)
909     {
910         if (cur->mark)
911         {
912             erase mark(cur, area);
913             Free(cur->mark);
914             Free(cur->erase mark);
915             cur->erase mark = NULL;
916         }
917         cur->mark = Alloc(hdr->length, YES);
918         memcpy(cur->mark, hdr, hdr->length);
919         cur->display mark = YES;
920     }
921     else
922     {
923         if (cur->old mark)
924             Free(cur->old mark);
925         cur->old mark = cur->mark;
926         cur->mark = NULL;
927     }
928     list->changes++;
929 }
930
931
932

```



```

9334 Restore mark(cur, area, list)
9335 register CURRENT *cur;
9336 register AREA *area;
9337 register LIST *l1st;
9338 {
9339     if (cur->old_mark)
9340     {
9341         if (cur->mark)
9342         {
9343             erase mark(cur, area);
9344             Free(cur->mark);
9345             Free(cur->erase mark);
9346             cur->erase mark = NULL;
9347         }
9348         cur->mark = cur->old mark;
9349         cur->old mark = NULL;
9350         l1st->changes++;
9351     }
9352 }
9353
9354 erase mark(cur, area) *cur;
9355 register CURRENT *cur;
9356 register AREA *area;
9357 {
9358     if (!cur->erase mark)
9359         cur->erase mark = Alloc(*(short*)cur->mark, YES);
9360     memcpy(cur->erase mark, cur->mark*(short*)cur->mark);
9361     ((P_E_HDR *)cur->erase mark)->color = area->color;
9362 }
9363
9364 Edit text(cur, area, l1st, msg, size, appl)
9365 register CURRENT *cur;
9366 register AREA *area;
9367 register LIST *l1st;
9368 register long msg, size;
9369 register long appl;
9370 {
9371

```

```

972 register char *p, c, *text_start, *new;
973 register short shift, offset, sel_offset, ok = YES;
974 register short sel_length;
975 ELEMENT *element;
976 P_E_HDR *hdr;
977
978 if (list->current = element = mark_elements(list, NULL, NULL, msg, size, appl))
979 {
980     offset = *(short *) Find_once(msg, "offs", size, &none, 2, NULL);
981     hdr = (P_E_HDR *) &element->length;
982     if (hdr->type == 't')
983     {
984         text_start = (p = value(hdr) + sizeof(long)) + 2 * sizeof(short);
985         if (shift = *(short *) Find_once(msg, "shift", size, &none, 2, 0))
986             shift text(p, text_start, shift);
987         if (Find_once(msg, "sel", size, NO, 0, NULL))
988         {
989             sel_offset = *((short *) p)++;
990             sel_length = *(short *) p;
991             ok = (offset < sel_length);
992             offset += sel_offset;
993         }
994     }
995     if (ok = (ok && (offset < strlen(text_start))))
996     {
997         p = text_start + offset;
998         if (new = Find_once(msg, "new", size, NULL, 1, NULL))
999         {
1000             while (c = *new++)
1001                 if (c > 31 && c < 127 && *p)
1002                     *p++ = c;
1003             else if (c == '\t')
1004                 *p++ = ' ';
1005         }
1006         if (Find_once(msg, "blnk", size, NO, 0, NULL))
1007             for (i = *p; *p++ = ' ');
1008         if (Find_triple(msg, "by", size, NO, 0, NULL))
1009             Move(area, list, msg, size, appl);
1010         Draw(list, msg, size);
1011     }
1012     else
1013     {
1014         element->changed = YES;
1015         list->changes++;
1016     }
1017     Move_mark(cur_area, list);
1018     if (Find_once(msg, "fast", size, NO, 0, NULL))
1019         list->erases = 0;

```

```

1020     } else reply_status(msg, "-edit", "outside text string", 0);
1021
1022     } else
1023     {
1024         reply_status(msg, "-edit", "not a text element", 0);
1025     }
1026
1027     } else
1028     {
1029         reply_status(msg, "-edit", "not found", 0);
1030     }
1031
1032     shift text(sel, text, nchars);
1033     register short *sel, nchars;
1034     register char *text;
1035     register short length, n;
1036     if (length == strlen(text))
1037     {
1038         if (nchars < 0 && (n == length + nchars) > 0)
1039         {
1040             memcpy(text, text+n, nchars);
1041             memset(text+nchars, ' ', n);
1042             if (*sel - n >= 0)
1043             {
1044                 else
1045                 {
1046                     *sel = 0;
1047                     *(sel+1) += *sel - n;
1048                 }
1049             }
1050         } else if (nchars > 0 && (n == length - nchars) > 0)
1051         {
1052             memcpy(text+length, text+nchars, nchars);
1053             memset(text, ' ', n);
1054             if (*sel + n < length)
1055             {
1056                 else
1057                 {
1058                     *sel = length - n;
1059                     *(sel+1) -= *sel + n - length;
1060                 }
1061             }
1062         }

```

```

1063 Animate(cur, list)
1064 register CURRENT
1065 register LIST
1066 {
1067     *cur;
1068     *list;
1069     *anim;
1070     *name;
1071     pid;
1072     *m;
1073     if (name = Find triple(cur->msg, "name", cur->size, NULL, 2, NULL))
1074     {
1075         for (anim = list->anims; anim && strcmp(name, anim->name);
1076             if (!anim)
1077                 if (pid = NewProc(name, "//processes/animate", YES, -1))
1078                     anim = (ANIM *) Alloc(sizeof(ANIM), YES);
1079                 strcpy(anim->name, name);
1080                 anim->next = list->anims;
1081                 list->anims = anim;
1082                 m = Alloc(cur->size, YES);
1083                 memcpy(m, cur->msg, cur->size);
1084                 Put(DIRECT, anim->conn.pid, m);
1085     }
1086     else
1087         reply_status(cur->msg, "-animate", "not supported", 0);
1088     else
1089         reply_status(cur->msg, "-animate", "duplicate name", 0);
1090     else
1091         reply_status(cur->msg, "-animate", "name too long", 0);
1092 }
1093
1094
1095
1096
1097

```

```

1098 alter(cur_list)
1099 register CURRENT *cur;
1100 register LIST *list;
1101
1102 register ANIM *anim;
1103 register char *name;
1104 register char conn;
1105 CONNECTOR
1106
1107 if (name = Find_triple(cur->msg,"name",cur->size,NULL,2,NULL))
1108 {
1109     for (anim = list->anims; anim && strcmp(name,anim->name);
1110         if (anim)
1111         {
1112             conn = anim->conn;
1113             if (!strcmp(cur->msg,"cancel"))
1114             {
1115                 list->anims = anim->nxt;
1116                 Free(anim);
1117             }
1118             Forward(DIRECT conn.pid,cur->msg);
1119             cur->msg = NULL;
1120         }
1121     else
1122     reply_status(cur->msg,cur->msg,"not found",0);
1123 }
1124
1125

```

```

1127 int(list,msg,size,appl)
1128 register list,*list;
1129 register long msg, size, appl;
1130 {
1131     register short *p,tolerance;
1132     register ELEMENT *element;
1133     register P_E_HDR *hdr;
1134     ELEMENT *find_box();
1135
1136     tolerance = *(short *) Find_triple(msg,"tolr",size,&none_2,NULL);
1137     if (p = (short *) Find_triple(msg,"pos",size,&none_4,NULL))
1138     {
1139         if (list->current = element = find_box(*p,*(p+1),list,appl))
1140         {
1141             hdr = (P_E_HDR *) &element->length;
1142             if (Find_triple(msg,"sel",size,NO,0,NULL) && hdr->attr.selectable)
1143             {
1144                 hdr->attr.selected = YES;
1145                 if ((hdr->type == 'm') && (*value(hdr) == 'L'))
1146                     set_list(hdr);
1147                 element->changed = YES;
1148                 list->changed++;
1149                 reply(msg,Newmsg(hdr->length+50,"write","data=#e#e",hdr,NULL));
1150             }
1151             else
1152                 reply_status(msg,msg,"not found",0);
1153         }
1154         else
1155             reply_status(msg,msg,"missing \ 'pos\ '",0);
1156     }
1157 }

```

```

1156 ELEMENT *find_box(row,col,list,appl)
1157 register short *row,col;
1158 register list *list;
1159 register long appl;
1160 {
1161     register P_E_HDR *hdr;
1162     register ELEMENT *element;
1163     for (element = list->last; element; element = element->pre)
1164     {
1165         hdr = (P_E_HDR *) &element->length;
1166         if (in_box(hdr->row, hdr->col, hdr->height, hdr->width, row, col)
1167             && !element->deleted)
1168             if (!appl || (appl == -1 && !*(long*)(hdr+1))
1169                 || appl == *(long*)(hdr+1))
1170                 break;
1171     }
1172     return(element);
1173 }
1174
1175 in_box(r,c,h,w,c1,c2)
1176 register short r, c, c1, c2, h, w;
1177 {
1178     if ((c1 < r) || (c2 < c))
1179         return(NO);
1180     if ((c1 > r+h) || (c2 > c+w))
1181         return(NO);
1182     return(YES);
1183 }
1184
1185 sel_list(hdr *hdr;
1186 register P_E_HDR *temp, *first;
1187 short len;
1188 char *p;
1189 for (first = temp = attr.hidden; temp = next_macro(&len, &p);
1190     if (temp)
1191     {
1192         temp->attr.hidden = YES;
1193         if (!temp = next_macro(&len, &p))
1194             temp = first;
1195         temp->attr.hidden = NO;
1196     }
1197 }
1198
1199
1200
1201
1202
1203

```

```

1204 viewport(cur_area, list)
1205 register CURRENT *cur;
1206 register AREA *area;
1207 register LIST *list;
1208 {
1209     register VIEW *view; *prev = NULL;
1210     CONNECTOR *conn;
1211     ELEMENT *element;
1212     unsigned int length = 0;
1213     char *p;
1214     if (p = Find_triple(cur->msg, "area", cur->size, NULL, 8, NULL))
1215     {
1216         for (view = list->views; view && (view->owner.pid != cur->sender.pid);
1217             if (view)
1218                 memcopy(&view->row, p, 4*sizeof(short));
1219         {
1220             view = (VIEW *) Alloc(sizeof(VIEW), YES);
1221             view->nxt = list->views;
1222             view->owner = cur->sender;
1223             memcopy(&view->row, p, 4*sizeof(short));
1224             list->views = view;
1225         }
1226     }
1227     change_area(area, view->row, view->col, view->height, view->width);
1228     element = mark_area(area->f1, area->f2, area->cl, area->cr, list,
1229                     MAX_P, E_NULL, NULL, &length, &length, NULL, cur->appl);
1230     send(cur, area, list, 0, length, element, YES, cur->display_mask, YES);
1231 }
1232 else
1233 {
1234     conn = (CONNECTOR *) Find_triple(cur->msg, "conn", 0, &cur->sender, 8, NULL);
1235     for (view = list->views; view && (view->owner.pid != conn->pid);
1236         if (view)
1237             if (prev)
1238                 if (prev->nxt = view->nxt;
1239                 else list->views = view->nxt;
1240                 Free(view);
1241             )
1242         )
1243     }
1244 }
1245 }
1246 }
1247 }
1248 }
1249 }

```



```

1251 change area(area,row,col,height,width)
1252 register AREA *area;
1253 register short row,col,height,width;
1254 {
1255     if (row < area->r1)
1256         area->r1 = row;
1257     if (col < area->c1)
1258         area->c1 = col;
1259     if (row + height > area->r2)
1260         area->r2 = row + height;
1261     if (col + width > area->c2)
1262         area->c2 = col + width;
1263 }
1264
1265 not_ify(cur_area,list)
1266 register CURRENT *cur;
1267 register AREA *area;
1268 *list;
1269 {
1270     register VIEW *view;
1271     register int length;
1272     for (view = list->views; view; view = view->nxt)
1273     {
1274         length = mark_changes(list->first,
1275             view->row,view->col,view->height,view->width);
1276         send(cur,area,list,&view->owner,length,list->first,
1277             YES,cur->display_mark,list->erases);
1278     }
1279 }
1280
1281 mark_changes(element,r,c,h,w)
1282 register ELEMENT *element;
1283 register short r,c,h,w;
1284 {
1285     register P E HDR *hdr;
1286     register int list_length = 0;
1287     for ( ; element && element->changed; element = element->nxt) ;
1288     for ( ; element; element = element->nxt)
1289     {
1290         hdr = (P E HDR *) &element->length;
1291         if (element->marked = (element->changed && !hdr->attr.hidden &&
1292             (hdr->row + hdr->height >= r) && (hdr->row <= r + h) &&
1293             (hdr->col + hdr->width >= c) && (hdr->col <= c + w)))
1294             list_length += hdr->length + 3;
1295     }
1296     return(list_length);
1297 }
1298
1299

```

```

1300 send(cur, area, list, proc, *cur, length, element, modify, mark, redraw)
1301 register CURRENT
1302 AREA
1303 LIST
1304 register CONNECTOR
1305 register unsigned int
1306 register ELEMENT
1307 register unsigned short modify, mark, redraw;
1308 {
1309     register P E HDR
1310     register short
1311     char *p; *set mark();
1312     ELEMENT
1313     *redraw_bkgd();
1314
1315     if (redraw)
1316         element = redraw_bkgd(area, list, &m, &p);
1317     else
1318         p = (m = Newmsg(length+300
1319             "write", &data=#A; type=#c", length+250, NULL, 'P')) + 24;
1320     if (element)
1321         for ( ; element; element = element->nxt)
1322         {
1323             if (element->marked && lelement->deleted)
1324                 element->marked = NO;
1325             element_length = element->length;
1326             memcpy(p, &element->length, (long)element_length);
1327             hdr = (P E HDR *) p;
1328             if (modify)
1329             {
1330                 if (hdr->attr.selected)
1331                     element_length = set select(hdr, cur->highlight);
1332                 if ((hdr->type == 'm') && (*value(hdr) == 'l'))
1333                     element_length = macro_list(hdr);
1334                 if ((hdr->type == 't'))
1335                     element_length = check_text(hdr, hdr->length);
1336                 if (cur->appl)
1337                     element_length =
1338                         change_origin(hdr, cur->appl_row, cur->appl_col);
1339             }
1340             p += element_length;
1341             Long_align(p);
1342         }
1343     if (mark)
1344         p = set mark(p, cur);
1345     *(short *) p = NULL;
1346     if (proc)
1347         put(DIRECT, proc->pid, m);
1348     else
1349         Reply(cur->msg, m);
1350 }
1351

```

```

1353 change_origin(hdr, row, col)
1354 register p E HDR *hdr;
1355 register short row, col;
1356 {
1357     if ((hdr->row == row) < 0)
1358         return(0);
1359     if ((hdr->col == col) < 0)
1360         return(0);
1361     return(hdr->length);
1362 }
1363
1364 char *set_mark(p, cur)
1365 register char *p;
1366 register CURRENT *cur;
1367 {
1368     if (cur->erase_mark)
1369     {
1370         memcpy(p, cur->erase_mark, *(short*)cur->erase_mark);
1371         p += *(short *) p;
1372     }
1373     if (cur->mark)
1374     {
1375         memcpy(p, cur->mark, *(short*)cur->mark);
1376         p += *(short *) p;
1377     }
1378     return(p);
1379 }
1380
1381 ELEMENT *redraw_bkgd(area, list, buf, ptr)
1382 register AREA *area;
1383 register LIST *list;
1384 register char **buf, **ptr;
1385 {
1386     ELEMENT *element;
1387     int length, num;
1388     element = mark_area(area->r1, area->c1, area->r2, area->c2,
1389         list, MAX_P E NULL, NULL, &length, &num, NULL);
1390     *buf += (4 * num) + 150;
1391     *ptr = Newmsg(length+50, "write", "data=#A; type=#c", length, NULL, 'P');
1392     draw_filled_rect(ptr, area->r1, area->c1, (area->r2)-(area->r1)
1393         *(area->c2)-(area->c1), NULL, 0, 0, area->color, area->pattern, 0, 0, 0, NULL);
1394     return(element);
1395 }

```

```

1399 set select(hdr_high_option)
1400 register p E HDR *hdr;
1401 register char high_option;
1402 {
1403     register short length;
1404     length = hdr->length;
1405     if (!high_option)
1406         if (hdr->attr.invert == !hdr->attr.invert;
1407             else if (high_option == 'i')
1408                 if (hdr->attr.invert == !hdr->attr.invert;
1409             else if (high_option == 'h')
1410                 if (hdr->attr.highlight == !hdr->attr.highlight;
1411             else if (high_option == 'c')
1412                 if (hdr->type != 'm')
1413                     {
1414                         if (hdr->color == (hdr->color + 1) % 7 + 1;
1415                         if (hdr->fill == (hdr->fill + 1) % 7 + 1;
1416                     }
1417                 else macro_color(hdr);
1418             else if (hdr->type == 't')
1419                 set text(hdr_high_option);
1420             return(length);
1421 }
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431 macro list(hdr)
1432 {
1433     register p E HDR *hdr;
1434     register short *temp;
1435     register short row, col;
1436     short len;
1437     char *p;
1438     row = hdr->row;
1439     col = hdr->col;
1440     for (temp = first_macro(hdr, NULL, &len, &p);
1441          temp && temp->attr.hidden; temp = next_macro(&len, &p));
1442     {
1443         memcpy(hdr, temp, temp->length);
1444         hdr->row = row;
1445         hdr->col = col;
1446     }
1447     return(hdr->length);
1448 }

```

```

1449 macro color(hdr)
1450 register P_E_HDR *hdr;
1451 {
1452     register P_E_HDR *temp;
1453     short len;
1454     char *p;
1455     for (temp = first_macro(hdr, NULL, &len, &p); temp; temp = next_macro(&len, &p))
1456     {
1457         temp->color = (temp->color + 1) % 7 + 1;
1458         if (temp->fill)
1459             temp->fill = (temp->fill + 1) % 7 + 1;
1460     }
1461     sel text(hdr, high_option)
1462     register P_E_HDR *hdr;
1463     register char high_option;
1464     register TEXT_OPTIONS *opt;
1465     opt = (TEXT_OPTIONS *) value(hdr);
1466     if (high_option == 'b')
1467         opt->border == YES;
1468     else if (high_option == 'u')
1469         opt->underline == YES;
1470     else if (high_option == 'B')
1471         opt->bold == YES;
1472     check text(hdr, length)
1473     register P_E_HDR *hdr;
1474     register short length;
1475     register char *p;
1476     register TEXT_OPTIONS *opt;
1477     opt = (TEXT_OPTIONS *) value(hdr);
1478     if (opt->border && hdr->fill)
1479     {
1480         opt->border = NO;
1481         p = (char *) hdr + length;
1482         n = p;
1483         draw_rect(&n, hdr->row-3, hdr->col-3, hdr->height+6, hdr->width+6,
1484                 NULL, hdr->fill, 1, NULL);
1485         length = n - (char*)hdr;
1486     }
1487     return(length);
1488 }
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500

```

```

1501 ELEMENT *mark_elements(list, length, num, msg, size, appl)
1502 list
1503 *list
1504 *length, *num,
1505 long
1506 {
1507     register short row = 0, col = 0, number = 0, count = 1;
1508     register short to_row = MAX_ROW, to_col = MAX_COL, *p;
1509     register ELEMENT *element;
1510     register char *tag_pat = NULL;
1511     char *what = NULL, tag_buf[200], *text_pat = NULL, dflt = YES;
1512     long *triple, attr = NULL;
1513     element = NULL;
1514     while (p = (short*)Find_triple(msg, "\0\0\0", size, NULL, 0, &triple))
1515     {
1516         switch (*triple)
1517         {
1518             case Keyback('e', 'c', 'n', 't'):
1519                 count = *p;
1520             case Keyback('e', 'a', 't', 't'):
1521                 attr = *(long *) p;
1522             case Keyback('e', 's', 'e', 'l'):
1523                 break;
1524             case Keyback('e', 'n', 'u', 'm'):
1525                 number = *p;
1526             case Keyback('e', 'p', 'o', 's'):
1527                 what = *p;
1528             case Keyback('e', 'e', 'n', 'd'):
1529                 row = *p;
1530                 to_col = *p;
1531             case Keyback('e', 't', 'x', 't'):
1532                 text_pat = Alloc(500, YES);
1533                 if (!makpat(p, text_pat))
1534                 {
1535                     Free(text_pat);
1536                     text_pat = NULL;
1537                 }
1538             case Keyback('e', 't', 'a', 'g'):
1539                 break;
1540                 if (!makpat(p, tag_pat = tag_buf))
1541                 {
1542                     tag_pat = NULL;
1543                 }
1544         }
1545     }
1546     triple = NULL;
1547     dflt = 'O';
1548     if (dflt)
1549     {
1550         count = MAX_P_E;
1551         if (!what)
1552             element = mark_number(number, tag_pat, text_pat,
1553                                 list_count, attr, length, num, appl);
1554         else if (what == 'A')
1555             element = mark_area(row, col, to_row, to_col, list_count,
1556                               attr, tag_pat, text_pat, length, num, appl);

```

```

15557 if (text_pat)
15558     free(text_pat);
15559     return(element);
15560
15561 ELEMENT *mark_area(row_col, to_row, to_col, list
15562     attr, tag_pat, text_pat, length, num, appl)
15563 register short
15564     row, col, to_row, to_col, count;
15565     *list;
15566     *tag_pat, *text_pat;
15567     *length, *num;
15568     int
15569     register P E HDR
15570     *hdr;
15571     register ELEMENT
15572     *element = NULL, *temp;
15573     register long
15574     total_length = 0;
15575     unsigned int
15576     orig_count;
15577     if (row >= 0 && col >= 0 && to_row >= 0 && to_col >= 0)
15578     {
15579         orig_count = count;
15580         for (temp = list->first; temp && count; temp = temp->nxt)
15581         {
15582             hdr = (P E HDR *) &temp->length;
15583             if (hdr->row <= to_row && hdr->col <= to_col
15584                 && row < hdr->row + hdr->height && col <
15585                 && valid(hdr, tag_pat, text_pat, attr, appl) && !temp->deleted)
15586             {
15587                 total_length += temp->length;
15588                 temp->marked = YES;
15589                 if (!element)
15590                     element = temp;
15591                 count--;
15592             }
15593         }
15594         if (length)
15595             *length = total_length;
15596         if (*num)
15597             *num = orig_count - count;
15598     }
15599     return(element);

```

```

1599 ELEMENT *mark number(n,tag_pat,text_pat,lst,count,attr,length,num,appl)
1600 register short n_count;
1601 register long tag_pat, text_pat, attr;
1602 register list *lst;
1603 register int *length, *num;
1604
1605 register ELEMENT *element = NULL, *temp = NULL;
1606 register long total_length = 0;
1607 register int orig_count;
1608
1609 if (n == -1)
1610     temp = lst->last;
1611 else
1612     for (temp = lst->first; temp && n--; temp = temp->nxt);
1613 for (orig_count = count; temp && count; temp = temp->nxt)
1614     if (valid(&temp->length,tag_pat,text_pat,attr,appl) && !temp->deleted)
1615     {
1616         total_length += temp->length;
1617         temp->marked = YES;
1618         if (!element)
1619             element = temp;
1620         count--;
1621     }
1622 if (*length) = total_length;
1623 if (*num) = orig_count-- count;
1624 return(element);
1625
1626
1627
1628

```



```

1630 valid(hdr, tag_pat, text_pat, attr, appl)
1631 register pE_hdr *hdr;
1632 register char *tag_pat, *text_pat;
1633 register long attr, appl;
1634 {
1635     register char *target, ok = YES;
1636     long temp;
1637     if (tag_pat)
1638         if (target = tag(hdr))
1639             if (ok = amatch(target, tag_pat))
1640                 else ok = NO;
1641     if (text_pat)
1642         if (hdr->type == 't')
1643             if (ok = ok && amatch(value(hdr)+8, text_pat))
1644                 else ok = NO;
1645     if (attr)
1646         if (attr)
1647             memcopy(&temp, &hdr->attr, sizeof(long));
1648             ok = ok && (temp & attr);
1649         }
1650     if (appl)
1651         if (appl == -1)
1652             ok = ok && (!*(long*)(hdr+1));
1653         else
1654             ok = ok && (appl == *(long*)(hdr+1));
1655     }
1656     return(ok);
1657 }
1658
1659 status(msg, size)
1660 register char *msg;
1661 register long size;
1662 {
1663     register char *m;
1664     *m = Alloc(size, YES) = NULL;
1665     strcat(m, "Find triple", size, &none, 1, NULL);
1666     strcat(m, "Find triple", size, &none, 1, NULL);
1667     strcat(m, "Find triple", size, &none, 1, NULL);
1668     strcat(m, "Find triple", size, &none, 1, NULL);
1669     Note(m, "ERROR");
1670     Free(m);
1671 }

```

```

1678 reply status(cur, mid, stat, code)
1679 register char *cur, *mid, *stat;
1680 register long *code;
1681 {
1682     register char *type;
1683     type = "failed";
1684     if (*mid == '-')
1685         mid++;
1686     else if (*mid == '+')
1687         type = "done";
1688     mid++;
1689     reply(cur, Newmsg(strlen(mid)+strlen(stat)+50, type, mid, stat, code));
1690 }
1691
1692 ELEMENT *new element(list, size, after)
1693 register list *list;
1694 register long size;
1695 register ELEMENT *after;
1696 {
1697     register ELEMENT *element;
1698     register long i = 0;
1699     if (size <= list->pool.size)
1700         for (i = list->pool.p, element = list->pool.ptr;
1701             element->pool.&& i && !element->deleted;
1702             (char*)element += list->pool.size, --i);
1703     if (i)
1704     {
1705         if (element->deleted)
1706             delete element(list, element);
1707         element->pool = YES;
1708     }
1709     else
1710     {
1711         element = (ELEMENT *) Alloc(size, YES);
1712         element->pool = NO;
1713     }
1714     element->nxt = NULL;
1715     if (element->pre == list->last)
1716         (list->last)->nxt = element;
1717     else list->first = element;
1718     list->last = element;
1719     element->changed = YES;
1720     element->deleted = element->marked = NO;
1721     return(element);
1722 }
1723
1724
1725
1726
1727
1728
1729

```

```

1730 delete element(list, element)
1731 register ELEMENT *element;
1732 register LIST *list;
1733 {
1734     if (element->pre)
1735         (element->pre) ->nxt = element->nxt;
1736     else
1737         list->first = element->nxt;
1738     if (element->nxt)
1739         (element->nxt) ->pre = element->pre;
1740     else
1741         list->last = element->pre;
1742     if (element->pool)
1743         element->pool = NULL;
1744     else
1745         Free(element);
1746     --list->size;
1747 }
1748
1749 char *value(hdr)
1750 register P_E_HDR *hdr;
1751 {
1752     register char *p;
1753     p = (char *)hdr + sizeof(P_E_HDR);
1754     if (hdr->attr.appl)
1755         if (hdr->attr.tagged)
1756             while (*p++);
1757     long align(p);
1758     return(p);
1759 }
1760
1761 char *tag(hdr)
1762 register P_E_HDR *hdr;
1763 {
1764     register char *p;
1765     p = (char *)hdr + sizeof(P_E_HDR);
1766     if (hdr->attr.appl)
1767         p += 4;
1768     if (hdr->attr.tagged)
1769         return(p);
1770     return(NULL);
1771 }
1772
1773
1774
1775
1776
1777

```

What is claimed is:

1. A virtual input interface in a data processing system, said interface comprising:

means for accepting input from at least one physical device and for converting said physical device input into virtual input, said means comprising a virtual input manager process responsive to said at least one physical input device for generating a picture, said picture comprising one or more picture elements, each picture element comprising a plurality of device-independent data structures in a predetermined, standard data format, at least one of said data structures comprising a plurality of different data fields each containing information describing said picture element; and

means responsive to said virtual input for performing processing operations upon said virtual input, said means comprising a console manager process for performing processing operations on said one or more picture elements.

2. The virtual input interface as recited in claim 1, wherein said input accepting means accepts input in the form of keystrokes.

3. The virtual input interface as recited in claim 1, wherein said input accepting means accepts input in the form of cursor position.

4. The virtual input interface as recited in claim 1, wherein said input accepting means accepts input in the form of system-defined actions.

5. The virtual input interface as recited in claim 1, wherein said input accepting means accepts input in the form of user-defined functions.

6. The virtual input interface as recited in claim 1, wherein said input accepting means accepts input in the form of menu selections.

7. The virtual input interface as recited in claim 1, wherein said at least one physical device can be removed from said system without affecting the operation of the remainder of said system.

8. The virtual input interface as recited in claim 1, wherein at least one additional physical device can be added to said system without affecting the operation of the remainder of said system.

9. A virtual output interface in a data processing system, said interface comprising:

a source of virtual input, said virtual input comprising one or more picture elements, each picture element comprising a plurality of device-independent data structures in a predetermined, standard data format, at least one of said data structures comprising a plurality of different data fields each containing information describing said picture element;

means for performing processing operations on said virtual input and for generating virtual output;

means for accepting said virtual output; and

means for converting said virtual output into at least one physical output suitable for use by at least one physical output device.

10. The virtual output interface as recited in claim 9, wherein said virtual input comprises a plurality of related picture elements and wherein said virtual output accepting means comprises a picture manager process for controlling said plurality of related picture elements.

11. The virtual output interface as recited in claim 10 and further comprising a display device, wherein said virtual output accepting means further comprises a window manager process for controlling the display of said plurality of related picture elements on said display device.

12. The virtual output interface as recited in claim 9, wherein said virtual output converting means comprises a virtual output manager process responsive to said one or more processed picture elements for coupling said one or more processed picture elements to said at least one physical output device.

13. The virtual output interface as recited in claim 9, wherein said at least one physical device can be removed from said system without affecting the operation of the remainder of said system.

14. The virtual output interface as recited in claim 9, wherein at least one additional physical device can be added to said system without affecting the operation of the remainder of said system.

15. In a data processing system, an interface between processes and data in said system and physical input and output devices coupled to said system, said interface comprising:

means responsive to one of said physical input devices for generating a picture, said picture comprising one or more picture elements, each picture element comprising a plurality of device-independent data structures in a predetermined, standard data format, at least one of said data structures comprising a plurality of different data fields each containing information describing said picture element;

means for performing processing operations on said one or more picture elements; and

means responsive to said one or more processed picture elements for coupling said one or more processed picture elements to one of said physical output devices.

16. The data processing system as recited in claim 15, wherein said one or more picture elements define a graphical object and at least one attribute thereof.

17. The data processing system as recited in claim 16, wherein one of said data fields describes the length of the associated picture element.

18. The data processing system as recited in claim 16, wherein one of said data fields identifies the particular type of the associated picture element.

19. The data processing system as recited in claim 16, wherein one of said data fields describes the position of the associated picture element relative to row and column coordinates on a picture of which said picture element forms a part.

20. The data processing system as recited in claim 16, wherein one of said data fields describes the size of the associated picture element.

21. The data processing system as recited in claim 16, wherein one of said data fields describes the color of the associated picture element.

22. The data processing system as recited in claim 15, wherein said means responsive to one of said physical input devices comprises a virtual input manager process.

23. The data processing system as recited in claim 15, wherein said means responsive to said one or more processed picture elements comprises a virtual output manager process.