



(72) LAING, Michael P., US

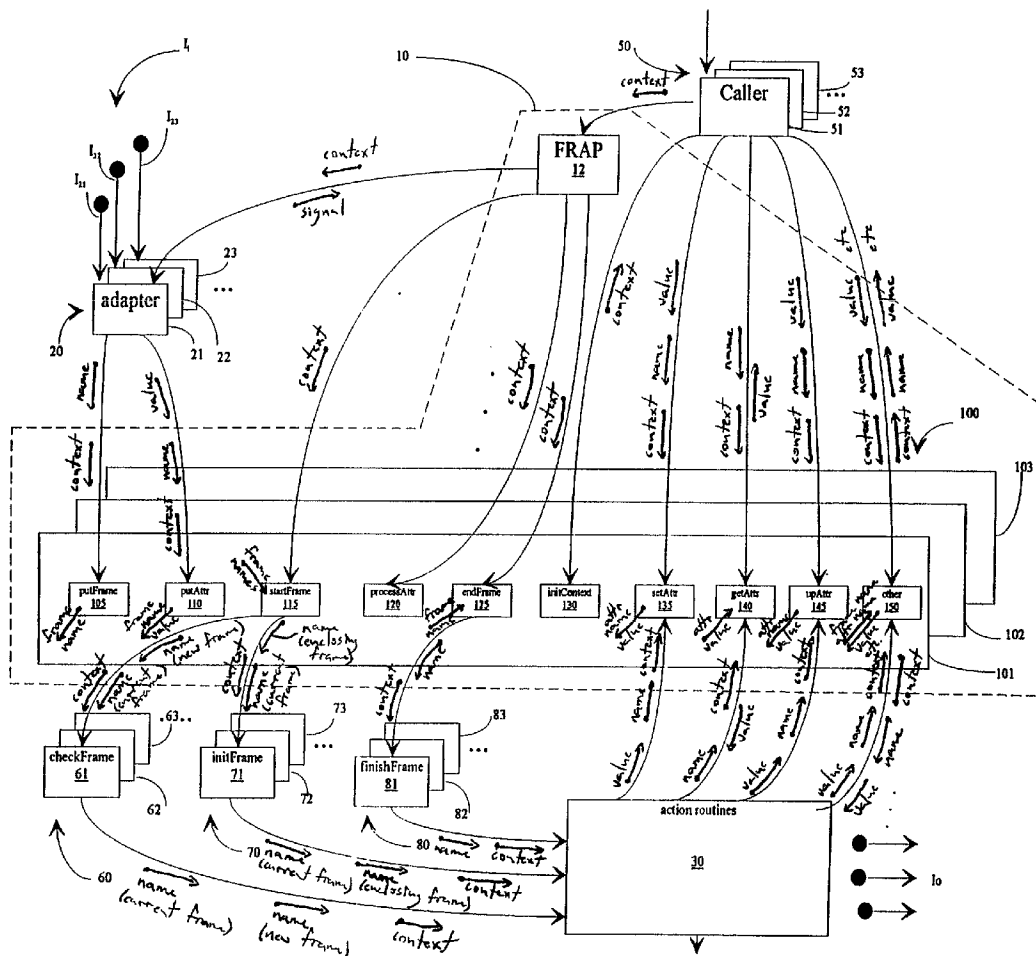
(71) PRESIDENT AND FELLOWS OF HARVARD COLLEGE, US

(51) Int.Cl.⁶ H04L 29/06, G06F 19/00

(30) 1995/09/15 (60/003,786) US

(54) **SYSTEME INFORMATIQUE DE TRAITEMENT DE FLUX DE
DONNEES**

(54) **COMPUTING SYSTEM FOR PROCESSING INFORMATION
FLOWS**



(57) Procédé polyvalent permettant de traiter des flux de données représentés par une séquence de trames. Toute trame peut avoir n'importe quel nombre d'attributs caractérisé chacun par un type et une valeur. Un attribut peut également être une trame, ce qui permet le traitement récursif des flux de données. Le traitement

(57) A general purpose procedure can process information flows which are represented by a sequence of frames. Any frame may have any number of attributes, each of which has a type and a value. An attribute can also be a frame, thereby permitting recursive processing of information flows. Common processing of frames is





(21) (A1) **2,228,593**
(86) 1996/09/13
(87) 1997/04/03

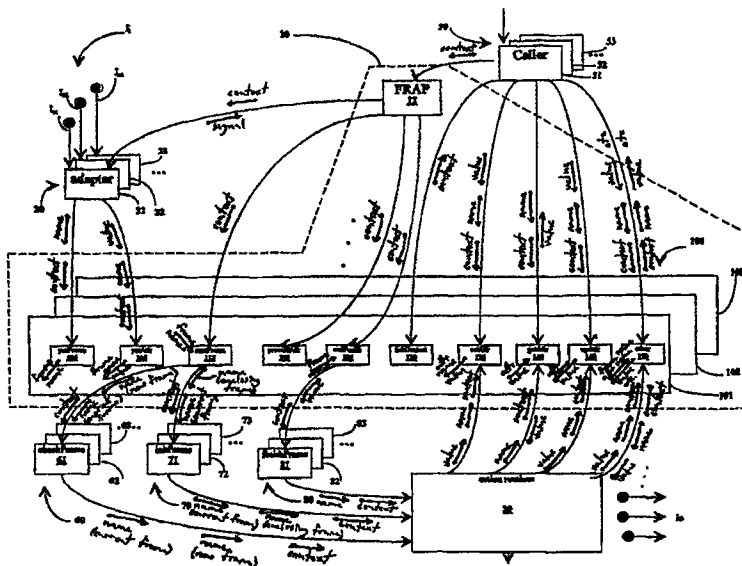
commun des trames est contenu dans une seule procédure capable de traiter les flux de données d'une grande variété de caractéristiques syntaxiques et sémantiques. La procédure unique peut traiter en même temps des protocoles de communication, des communications intra-ordinateur et des protocoles codage de sécurité. Cette procédure peut également être réalisée sous la forme de matériel ou de microprogrammes.

encapsulated into a single procedure that can handle information flows having a wide variety of syntactic and semantic characteristics. The single procedure can process communication protocols, intracomputer communications and security encryption protocols at the same time. The procedure can also be embodied in hardware or firmware.

**PCT**WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁶ : H04L 29/06	A3	(11) International Publication Number: WO 97/12339 (43) International Publication Date: 3 April 1997 (03.04.97)
(21) International Application Number: PCT/US96/14663 (22) International Filing Date: 13 September 1996 (13.09.96) (30) Priority Data: 60/003,786 15 September 1995 (15.09.95) US (71) Applicant (for all designated States except US): PRESIDENT AND FELLOWS OF HARVARD COLLEGE [US/US]; 1350 Massachusetts Avenue, Cambridge, MA 02138 (US). (72) Inventor; and (75) Inventor/Applicant (for US only): LAING, Michael, P. [US/US]; 559 Friendship Road, Waldobor, ME 04572 (US). (74) Agents: SMITH, James, M. et al.; Hamilton, Brook, Smith & Reynolds, Two Militia Drive, Lexington, MA 02173 (US).		(81) Designated States: CA, JP, US, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE). Published <i>With international search report.</i> <i>Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.</i> (88) Date of publication of the international search report: 31 July 1997 (31.07.97)

(54) Title: COMPUTING SYSTEM FOR PROCESSING INFORMATION FLOWS**(57) Abstract**

A general purpose procedure can process information flows which are represented by a sequence of frames. Any frame may have any number of attributes, each of which has a type and a value. An attribute can also be a frame, thereby permitting recursive processing of information flows. Common processing of frames is encapsulated into a single procedure that can handle information flows having a wide variety of syntactic and semantic characteristics. The single procedure can process communication protocols, intracomputer communications and security encryption protocols at the same time. The procedure can also be embodied in hardware or firmware.

-1-

COMPUTING SYSTEM FOR PROCESSING INFORMATION FLOWSRELATED APPLICATIONS

This application claims priority to U.S. Provisional Application Serial No. 60/003,786 filed on September 15, 1995 incorporated by reference.

5 BACKGROUND OF THE INVENTION

Information flows within or between computers must have a structure that allows meaning to be extracted in order to be useful. The encoding of the information is generally termed its syntax and the term semantics is used for the meaning of the information, which may imply actions to be taken. The information flows are many
10 and various; usually specific non-general methods are associated with the processing of each type and level of such flows within the computer.

To generalize the protocols, there is needed a general way to hook the syntax of a protocol (what it looks like "on the wire") to the semantics of the protocol (how it is processed). This can be done by creating recursive routines that invoke each other
15 to do the processing. That prior art technique is difficult to program, expensive of computer resources, and difficult to generalize.

One example of an explicit architecture for constructing and composing network protocols is discussed by N.C. Hutchinson and Larry L. Peterson in "The x-Kernel: An Architecture for Implementing Network Protocols" (IEEE Trans. Software
20 Eng., Vol. 17:1 (1991)). This architecture defines protocols, sessions and messages as objects. The kernel shepards messages through a network of protocol and session objects. Through this process, message headers are added or stripped from the data and messages can be manipulated. The x-kernel architecture requires that supported protocols be created and initialized in a hierarchical order.

-1.1-

SUMMARY OF THE INVENTION

Frames are a general way of representing information and its structure in artificial intelligence systems. Essentially, a frame is a set of attributes and each
5 attribute in a frame has a type and a value. Furthermore, any attribute can itself be a frame. Many of the information flows within or between computers can be represented as sequences of frames.

- 2 -

A general apparatus and method in accordance with the invention processes information flows which can be represented as sequences of frames. Preferred embodiments of the invention encapsulate the common processing of frames in a single processing engine that can efficiently handle information flows possessing a wide variety of syntactic and semantic characteristics. Preferred embodiments deal abstractly with the information flows, relying on one class of lower level routines (termed 'adapter routines') to provide inputs and on other classes of lower level routines (termed 'action routines') to implement any concrete actions necessary to accomplish the processing task. By performing the difficult iterations and recursions in the general engine, the adapter and action routines can be made simpler and smaller and thus easier to program.

Preferably, the processing engine includes a common core engine and a plurality of support routines. The support routines operate on at least one hierarchical memory store, such as a stack of attributes and a stack of frames. The support routines can also generate synthetic attributes for storage in the stack of attributes. In addition, the processing engine is preferably reentrant and capable of multi-threading.

The information flows can represent various kinds of information. The information flow can be a protocol-based stream of data between computers for intercomputer communications. The information flow can also represent intracomputer communications, such as configuration and environment information. The processing engine can also be viewed as a language processor.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features and advantages of the invention, including various novel details of construction and combination of parts will be apparent from the following more particular drawings and description of preferred embodiments of the computing system for processing information flows in which like reference characters refer to the same parts throughout the different views. It will be understood that the particular apparatus and methods embodying the

- 3 -

invention are shown by way of illustration only and not as a limitation of the invention, emphasis instead being placed upon illustrating the principles of the invention. The principles and features of this invention may be employed in various and numerous embodiments without departing from the scope of the invention.

5 FIG. 1A is a block diagram illustrating a prior art system for interchanging information over a computer network.

FIG. 1B is a representative electronic mail message processable by the prior art system of FIG. 1A.

10 FIG. 2 is a simplified block diagram illustrating a basic embodiment of the invention.

FIG. 3 is a more detailed block diagram of a preferred embodiment of the invention.

FIG. 4 is a schematic diagram of a preferred embodiment of an initialized memory store.

15 FIG. 5 is a schematic diagram of a representative memory store of FIG. 4 with data stored therein.

FIG. 6 is a schematic diagram of the memory store of FIG. 5 after invocation of the 'endFrame' routine 125 of FIG. 3.

20 FIG. 7 is a schematic diagram of the memory store of FIG. 6 after invocation of the 'setAttribute' routine 135 of FIG. 3.

FIG. 8 is a schematic diagram of the memory store of FIG. 7 after invocation of the 'upAttribute' routine 145 of FIG. 3.

FIG. 9 is a block diagram of a preferred embodiment of the invention employing parallel processing and pipelining techniques.

25 FIG. 10 is a block diagram of a memory construct for the system of FIG. 9.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS OF THE INVENTION

FIG. 1A is a block diagram illustrating a prior art system for interchanging information over a computer network. As illustrated, two computers 1, 2 are

- 4 -

exchanging information over a computer network 3, such as Ethernet, which has its own defined communication protocols. The first computer 1 includes layers of procedures to translate the data from the network protocol into information which can be presented to a user. As illustrated the computers 1, 2 have identical internal
5 procedural layers. However, the two computers 1, 2 may typically translate through different layers of protocols. FIG. 2 is a representative electronic mail message processable by the prior art system of FIG. 1A.

Briefly, a network interface routine (NIF) receives the information flows from the network and processes the network protocol layer. A Transmission
10 Control Protocol (TCP) procedure processes the TCP layer. A Simple Mail Transport Protocol (SMTP) procedure processes the SMTP layer. An RFC822 procedure processes the mail protocol layer. Finally, a user procedure then displays the information to the user. Messages created by the user are converted by the respective computer procedures into the network protocol for transmission across the
15 network 3 to the destination computer.

Preferred embodiments of the computing systems in accordance with the invention facilitate the separation of the primary task of processing an appropriate information flow into three distinct subtasks, one of which is a processing engine that is universally applicable to all such information flows.

20 The information flows can be viewed as frames as defined by Marvin Minsky, "A Framework for Representing Knowledge," in *The Psychology of Computer Vision* pp. 211-277 (P.H. Winston ed., McGraw-Hill) (1975). As used herein, a frame is a collection of attributes and an attribute has a name and a value. An attribute can also be a frame, which creates recursion.

25 FIG. 2 is a schematic block diagram of a basic embodiment of the invention. A processing engine 10 has as its responsibilities: 1) validation that the information flow is a sequence of frames, and 2) the primary control logic including iteration and recursion to process the sequence. At least one adapter routine 20 is responsible for scanning an incoming information flow I_i , recognizing frame boundaries and
30 attributes, and signalling the processing engine 10 as they are encountered. A

- 5 -

plurality of action routines 30 are responsible for implementing the actual processing of the components of the information flow I_i when activated by the processing engine 10. The action routines 30 can also generate outgoing information flows I_o , which can be recursively processed through an adapter routine 20.

5 While the processing engine 10 is a general purpose procedure which is transportable to different computers, the adapter routine 20 and the action routines 30 are specifically written by a programmer for the target computer. Preferably, there are multiple adapter routines 20, each specifically written for a particular protocol. As will be shown below, the processing engine 10, the adapter routines
10 20 and the action routines 30 are very simple to program because the incoming information flows I_i are treated as sequences of frames of data.

FIG. 3 is a schematic block diagram of a preferred embodiment of the invention. In the figure, the larger arrows represent control signals and the smaller arrows extending from dots represent data flow. General purpose routines of the
15 processing engine are shown unshaded, whereas non-general purpose routines (specifically programmed for the environment) are shown shaded in the figure.

The processing engine 10 is illustrated as having two major components, a reentrant frame processor (FRAP) 12 which acts as the core engine and multiple memory contexts 100. The processing engine 10 communicates with multiple
20 context-linked adapter routines 20, which receive multiple incoming information flows I_i , at least one information flow per adapter 20. The processing engine 10 is triggered by multiple potential caller routines 50. Typically, each caller routine 50 acquires at least one context to avoid conflicts. The processing engine 10 also communicates with general purpose action routines 30 and special context-linked
25 action routines 60, 70, 80.

Briefly, each memory context 100 is maintained by a plurality of support routines. The support routines include a 'putFrame' routine 5, a 'putAttribute' routine 110, a 'startFrame' routine 115, a 'processAttribute' routine 120, an 'endFrame' routine 125, an 'initContext' routine 130, a 'setAttribute' routine 135, a
30 'getAttribute' routine 140, an 'upAttribute' routine 145 and possibly other support

- 6 -

5 routines shown generally as 150. Each adapter routine 20 is associated with a respective memory context. As illustrated, the first adapter 21 receives an information flow I_{21} and is associated with a respective memory context 101. Likewise, a second adapter 22 receives an information flow I_{22} and is associated with a respective memory context 102.

Also illustrated are 'checkFrame' routines 60, 'initFrame' routines 70 and 'finishFrame' routines 80. Each instance of these special action routines are associated with a respective memory context and adapter pair. For example, the first adapter routine 21 is associated with a first memory context 101, a first 'checkFrame' routine 61, a first 'initFrame' routine 71, and a first 'finishFrame' routine 81. When routines are invoked, the specific routine called depends upon the context. In the case of memory, the hierarchal memory store used also depends on the context.

15 The general purpose action routines 30 can also create outgoing information flows I_0 which may then wrap around and be processed through another invocation of the frame processor 12. This is typically accomplished by an action routine 30 invoking the frame processor 12 again in the role of a caller routine.

Processing begins from a caller routine 51. The caller routine 51 first invokes an 'initContext' routine 130 to create a new context and associate the various routines, information flows, and memory with that context. A unique identifier for the context is returned to the caller 51. The caller 51 then triggers the frame processor 12 and passes the context. The frame processor 12 then signals a respective adapter routine 21 which is monitoring an incoming information flow I_{21} . The adapter routine 21 signals the frame processor 12 with a START_OF_FRAME or an END_OF_FRAME or an ATTRIBUTE signal.

25 Upon receiving a START_OF_FRAME signal, the frame processor 12 signals the respective 'startFrame' routine 115 passing the context. The 'startFrame' routine 115 retrieves the current and new frame names from memory. The 'startFrame' routine 115 then invokes a respective 'checkFrame' routine 61 passing the context, the current frame name and the new frame name. The 'checkFrame'

30

- 7 -

routine 61 may also call other action routines 30 passing the context, current frame name and new frame name.

The 'startFrame' routine 115 then makes the new frame the current frame and invokes a respective 'initFrame' routine 71 passing the context, the current
5 frame name and enclosing frame name. The 'initFrame' routine 71 may also call other action routines 30, passing the context, current frame name and enclosing frame name.

The frame processor 12 may then receive an ATTRIBUTE signal from the adapter routine 21. In response to the ATTRIBUTE signal, the frame processor 12
10 invokes the 'processAttribute' routine 120, passing the context, to store the attribute in memory.

The frame processor 12 may next receive an END_OF_FRAME signal from the adapter routine 21. In response, the frame processor 12 invokes the 'endFrame' routine 125 with the context. The 'endFrame' routine 125 retrieves the frame name
15 from the memory context and invokes the 'finishFrame' routine 81. The finish frame routine 81 is passed the context and frame name. The 'finishFrame' routine 81 also invokes action routines 30, passing the name and context.

As can be seen, the frame processor 12 keeps track of signals. Indeed, the frame processor 12 does not receive any values, only signals. By using context
20 memory the frame processor 12 can be made reentrant and multi-threaded. These and other routines will be described in more detail below.

The memory context is a hierarchical memory store having a unique identifier. Each memory context matches two stacks: a frame stack and an attribute stack. Each attribute is associated with a frame. In the stacks, the most recently
25 stored and therefore most likely to be needed attributes are at the top of the stack, where they are readily available for processing. If further attributes are needed and they are not supplied with the top frame, the stacks can be scanned from top to bottom to inherit the attribute from below (i.e., an outer frame layer). A context has three associated information streams: an input stream, an output stream and an

- 8 -

error output stream. It should be noted that context memory 100 is not necessary for the invention, however the use of context memory 100 offers certain advantages.

Basic Grammar

5 This instance of the processing system comprises a formal grammar which, although expressed here in English, can be translated directly into an executable computer procedure by any of several grammar translators such as YACC or BISON. This is one of many equivalent ways of creating a computer procedure that implements the processing engine 10. Presented are the elements of the grammar together with explanatory comments that are not a part of the grammar.

10 In the grammar, words in quotes represent essential elements of the grammar with particular meanings: 1) words that are all capitals represent input signals to the processing engine from the adapter routine; 2) words that are mixed case and begin with a capital are used internally within the processing engine and are derived from the inputs through the rules of the grammar; and 3) words that are mixed case and
15 begin with a lower case letter are routines that are invoked by the frame processor 12.

1. A 'FrameList' is:
 - a. a 'Frame'
 - or
 - 20 b. a 'Frame' list followed by a 'Frame'.

A 'FrameList' is a sequence of frames which may consist of just one 'Frame'. Because this is the first rule, the resulting computer procedure will expect that its entire input information flow can be categorized as a sequence of frames and will return an error if this is not so. This rule provides the first level of iteration in
25 the control structure of the processing engine 10.

- 9 -

2. A 'Frame' is:

a. a 'START_OF_FRAME' (execute the 'startFrame' routine) followed by a 'SubFrameList' followed by an 'END_OF_FRAME' (execute the 'endFrame' routine)

5 or

b. a 'START_OF_FRAME' followed by an 'END_OF_FRAME' (execute the 'startFrame' routine then execute the 'endFrame' routine).

A 'Frame' has a start ('START_OF_FRAME') and an end ('END_OF_FRAME') and optionally contains a 'SubFrameList'; routines ('startFrame' and 'endFrame') are executed at the appropriate times as directed by the grammar.

3. A 'SubFrameList' is:

a. a 'SubFrame'

15 or

b. a 'SubFrameList' followed by a 'SubFrame'.

A 'SubFrameList' is a sequence of SubFrames which may consist of just one 'SubFrame'. This rule provides the second level of iteration in the control structure of the processing engine 10.

20 4. A SubFrame' is:

a. a 'Frame'

or

b. an 'ATTRIBUTE' (execute the 'processAttribute' routine).

-10-

A 'SubFrame' is either itself a 'Frame' or it is an 'Attribute', in which case the action routine 'processAttribute' is invoked. This rule provides the recursion in the control structure of the processing engine 10.

The above grammar is now written below in a language that can be processed into an executable program. The vertical bar (|) represents "or" in the language and routines being invoked are placed inside braces ({}).

1. FrameList: Frame | FrameList frame
2. Frame: START_OF_FRAME {startFrame();} SubFrameList
END_OF_FRAME {endFrame();}
10 | START_OF_FRAME END_OF_FRAME {startFrame ();
endFrame();}
3. SubFrameList: SubFrame | SubFrameList SubFrame
4. SubFrame: ATTRIBUTE {processAttribute ();} | frame

Because an attribute is really a type and a value, the rules can be extended to so define an attribute. However, expanding the rules to include types and values can complicate the task of processing the information flows. For that reason, the rules are not extended in this embodiment.

Primary Supplements to the Basic Embodiment

As briefly described above, the basic embodiment described with reference to FIG. 2 can be supplemented with general purpose action routines that facilitate the processing of frames and their attributes. Turning to FIG. 3, most of these supplemental routines are private, which means they can only be invoked by the processing engine itself. One routine is public, which means that non-general action routines may invoke it.

In some cases described below, the invocation of an action routine associated with a frame is referenced. There are many ways to perform such an association including: associating all frames with one such routine, using the name of the frame

-11-

to determine a routine to invoke, using the available attributes of the frame to determine the routine, etc. Making a correspondence between frames and object classes in an object-oriented programming environment is another way to provide the association. These routines work with frame names and with attribute names and
5 values; any or all of these items may be complex, *i.e.*, consist of an associated collection of data items, any of which may also be complex.

The private 'startFrame' support routine 115 first calls a non-general action routine (*e.g.* 'checkFrame' 61) and passes it two values: the name (if it has one) of current frame and the name (if it has one) of the new frame being started. The
10 name of the current frame is derived from the hierarchical memory store of frame names described below. The name of the new frame may have been passed from the adapter routine 21 to the processing engine 10 and then to the 'startFrame' routine 115 or it may have been made available in some other way to the routine, but in any case it is provisionally stored in memory at this point. The purpose of
15 calling the 'checkFrame' routine 61 is to allow a routine associated with the current frame to be executed prior to the full initialization of the new frame. Such a routine can perform any sort of processing and might typically check the validity of processing the new frame within the current frame. If the 'checkFrame' routine 61 returns an error, then the 'startFrame' routine 115 may cause the new frame to be
20 skipped, including its attributes and any contained frames to any depth.

If warranted by the degree of success of the preceding steps, the 'startFrame' routine 115 next triggers the permanent storage of the name of the new frame, which then becomes the current frame. Frame names are used to dynamically construct a computer memory space that contains the name of the active frame at
25 each hierarchical level. The current frame is at the highest level and the lowest level is always occupied by an implicit frame, typically named 'ROOT', which logically contains all the other frames and is present upon initialization of the system and support routines.

FIG. 4 is a schematic diagram of a preferred embodiment of an initialized
30 memory store. The computer memory space comprises a hierarchical memory store

-12-

of frame names (stack FRAME) privately kept by the general purpose action routines and directly accessible only by them. The hierarchical memory store of frame names is associated with a corresponding store of attribute names (stack ATTR) and values as described below. By default, the 'ROOT' frame is associated with a null attribute 'NULL', but particular attributes and values may be stored on the ATTR stack and associated with the 'ROOT' frame to be inherited by later frames, as will be described below.

FIG. 5 is a schematic diagram of a representative memory store of FIG. 4 with data stored therein. The memory store of attribute names and values is logically organized as a last-in-first-out stack in such a way that attribute names and values can be accessed sequentially in reverse order (from last in to first out) and such that all the attribute names and values associated with the current frame can be removed from the stack. Like the memory store of frame names, the memory store of attribute names and associated values is privately kept by the non-general purpose action routines and is directly accessible only by them.

Returning to FIG. 3, the 'startFrame' support routine 115 then calls a non-general action routine (e.g. 'initFrame' 71) and passes it two values: the name (if it has one) of the new frame, which is now the current frame, and the name (if it has one) of the parent frame. The frame names are derived from the hierarchical memory store of frame names described above. The purpose of calling the 'initFrame' routine 71 is to allow a routine associated with the current (new) frame to be executed prior to the processing of attributes and/or contained frames. Such a routine can perform any sort of processing and may typically check the validity of processing the current (new) frame within the parent frame. If the 'initFrame' routine 71 returns an error then the 'startFrame' routine 115 may cause the current (new) frame to be skipped, including its attributes and any contained frames to any depth.

The combination of the 'initFrame' routine 71 invocation and the 'checkFrame' routine 61 invocation described above provides the opportunity for non-general action routines associated with frames to easily enforce a hierarchy of

-13-

processing wherein each frame specifies both what frames can contain it and what frames it can contain, and the enclosing frame always takes precedence over the enclosed frame.

The private 'processAttribute' support routine 120 triggers the storage of the attribute name and associated value in memory associated with the hierarchical memory store of frame names. The attribute name and value may have been passed from the adapter routine to the processing engine 10 and then to the 'processAttribute' routine 120 or they may have been made available in some other way to the routine.

10 The private 'endFrame' support routine 125 calls a non-general action routine (e.g. 'finishFrame' 81) and passes it the name (if it has one) of the current frame derived from the hierarchical memory store of frame names described above. The purpose of calling the 'finishFrame' routine 81 is to allow a routine associated with the frame to be invoked that will complete the processing of the frame, as all
15 contained attributes and frames will have been completely processed at this point. Upon completion of the 'finishFrame' routine, all attribute names and values associated with the current frame are logically removed from the memory store, the current frame name is also removed, and the enclosing frame is made the current frame.

20 FIG. 6 is a schematic diagram of the memory store of FIG. 5 after invocation of the 'endFrame' routine 125. Note that frame F3 and its associate attributes A4, A5 have been popped from the stacks and the current frame is now F2.

Returning to FIG. 3, the public 'getAttribute' support routine 140 is typically
25 invoked from a non-general action routine associated with a frame. It expects to be passed an attribute name, then logically scans the stack of attribute names and values in reverse order (from last in to first out) until a matching name is found or until all candidate attributes in the stack have been scanned. If a matching name is found, it returns the associated value to the caller, otherwise it notifies the caller that the
30 attribute was not found.

Via the above mechanism, the attribute values are logically presented in a hierarchical manner wherein the latest value stored with a particular name is always the one returned, overriding any previous values associated with that name and set at the same frame level or at higher levels. This scheme provides for inheritance of attributes by enclosed frames wherein a frame for which the adapter routine did not provide an attribute from the information flow can inherit that attribute from an enclosing frame higher in the hierarchy and the attribute value provided is always the closest one that is associated with an active frame. Referring again to FIGS. 5 and 6, the attribute A2 associated with frame F1 is overridden by the attribute A2 associated with the more current frame F2. These attributes can have different values.

Secondary Supplements to the Basic Embodiment

Additional support routines also add useful extensions to the basic embodiment described above, enabling the definition of sequences of frames and their processing that can be sufficiently complex as to comprise a family of computer languages.

Returning to FIG. 3, the public 'setAttribute' support routine 135 is typically invoked from a non-general action routine associated with a frame. It expects to be passed an attribute name and value and places them in the private hierarchical memory stores shared with the other support routines. Attributes set this way are termed synthetic attributes as they are created by action routines rather than being received in the information flow itself. The 'setAttribute' routine 135 stores the attribute in association with the current frame as if it were received from the information flow. The effect of this facility is to provide a way for action routines associated with the current frame to pass information to each other and to enclosed frames via inheritance.

FIG. 7 is a schematic diagram of the memory store of FIG. 6 after invocation of the 'setAttribute' routine 135. Attribute A6 is placed on the attribute stack (ATTR) and associated with the current frame F2 (determined from FIG. 6).

-15-

Returning to FIG. 3, the public 'upAttribute' support routine 145 is also typically invoked from a non-general action routine associated with a frame. It too expects to be passed an attribute name and value and places them in the private hierarchical memory stores shared with the other support routines. However, the synthetic attribute thus created is associated with its immediate parent frame, not with the current frame. The hierarchical memory store is adjusted so that it appears as if the synthetic attribute was the last attribute received from the information flow for the parent frame and was then followed by the attributes of the current frame present prior to the adjustment. Logically, the set of attributes associated with the current frame are removed from the hierarchical memory store and saved, the current frame is then removed from the memory store and saved, the enclosing frame is made the current frame, the new synthetic attribute is stored, the saved frame is stored (becoming the current frame), and finally each attribute in the saved set of attributes is stored in its original order.

FIG. 8 is a schematic diagram of the memory store of FIG. 7 after invocation of the 'upAttribute' routine 145. Attribute A7 has been placed on the attribute stack (ATTR) and associated with the parent frame F1. The current frame F2 retains an association with its attributes A2, A6.

One effect of this facility is the opposite of inheritance: it provides a way for action routines to return values up the hierarchy so that they can be retrieved and acted upon by action routines associated with frames at a higher level. Additionally the synthetic attribute can be inherited by downstream frames (frames encountered subsequently in the information flow) outside of the hierarchical sub-tree headed by the current frame. Through the action stated above, this routine provides a general facility for resolving frames into attributes. That is, the frame, and all of its attributes and enclosed frames if any, can be viewed as being replaced in the information flow by synthetic attributes which are set by the action of this routine.

This general facility for reducing complex information objects (frames) into simpler ones (attributes) and returning results up the processing hierarchy together with the other support routines and the basic embodiment itself enables the general

-16-

processing of information flows sufficiently complex that they can be considered computer languages. The languages must, however, be expressible as sequences of frames.

Returning to FIG. 3, other useful public support routines 150 or special purpose attributes can make additional facilities available to action routines. For example, one class of routines can provide more detailed and/or complete access to the hierarchical memory store, enabling an action routine to walk through the active frames and attributes, possibly retrieving more detailed information about each element. Another class of routines or synthetic attributes can provide ways of affecting the processing of frames, *e.g.* skipping the remaining processing of a frame and its attributes and enclosed frames. Another class of routines or attributes can allow a frame or an action routine associated with a frame to disable or modify access to the public support routines by the action routines associated with enclosed frames; this could enable a hierarchically based security scheme in which enclosing frames and their action routines control the privileges of the action routines of enclosed frames. An additional class of synthetic attributes can modify the normal retrieval of attribute values, for example a special type of synthetic attribute could nullify or mask the presence of an attribute from any downstream action routine within the scope of the synthetic attribute's associated frame.

20 *Tertiary Supplements to the Basic Embodiment*

Additional support routines and minor modifications to the other routines further extend the basic embodiment described above such that it can be more compactly implemented, lessens the burden on programmers of action routines, and is fully reentrant and capable of multi-threading, *i.e.*, capable of processing multiple information flows at one time without collision.

The public 'initContext' routine 130 takes as arguments the information necessary to invoke four non-general routines: an 'adapter' routine 21, a 'checkFrame' routine 61 an 'initFrame' routine 71, and a 'finishFrame' routine 81; and the information necessary to invoke three information flows: 'input', 'output',

-17-

and 'error output'. It initializes a new hierarchical memory store (a memory context) distinct from any others and returns a unique item called a context that links together the four routines, the information flows, and the new memory context such that they can be used by other routines.

5 Multiple contexts can be active at once. Each general routine and each non-general routine invoked by a general routine receives a context as an argument in addition to any other arguments. A new context can be grafted onto an existing context such that the memory store of the existing context appears as an enclosing or a superior memory hierarchy of the new context. Any number of these contexts can
10 exist in parallel, defining a tree-structured memory construct. Each thread of execution establishes at least one memory context. The core engine manages the trees of memory context in such a way that memory can be safely shared among the executing threads.

The public 'putFrame' support routine 105 is typically invoked from an
15 adapter routine. It is passed the frame name and prepares the memory context for storage of the name. Similarly, the public 'putAttribute' support routine 110 is passed an attribute name and value and prepares the memory context for their storage.

By providing direct movement of the names and values from adapter
20 routines, the core implementation of the processing engine 10 is simplified because it does not need to receive the names and values, store them internally, and pass them to other routines. This speeds up the execution of the system and reduces its internal memory requirements substantially.

Other useful routines could manipulate contexts. A routine could make a
25 copy of the contents of a context, cloning the state of the hierarchical memory store for use in a new context. Other routines could be devised to merge memory contexts.

Other supplements to the system in accordance with the invention address

multi-threading. Based upon effective values of attributes, the system automatically creates parallel threads of execution to process each frame in a sequence.

Additionally, routines allow the creation of threaded pipes to connect the output of one thread (as a frame sequence) to the input of another thread.

- 5 A preferred processing engine in accordance with the invention provides at least two services based upon multi-threading: automatic "parallel processing" of serial frame sequences and pipelines of frame sequences between threads ("threaded pipes"). Parallel processing occurs automatically when an adapter routine receives a sequence of frames; as each complete frame is received, the general engine
- 10 dispatches a thread of execution to process the frame and then immediately invokes its adapter routine to obtain the next frame. Due to the hierarchal nature of frames and the reentrant and recursive attributes of the general engine, this parallel processing feature can result in many simultaneously executed parallel threads of execution which execute in an orderly manner to receive and process a sequence of
- 15 frames. Threaded pipes dynamically connect the output (as a frame sequence) from the action routines in one thread of execution to the input of the adapter routine in another thread. This preferred feature allows for dynamic parallelization of the processing of successive levels in a complex information flow. Additionally, processing routines for each level can be kept relatively simple and distinct.
- 20 FIG. 9 is a block diagram of a preferred embodiment of the invention employing parallel processing and pipelining techniques. Illustrated in a calling application 100 for processing information flows from a source 105. The application 100 invokes the frame processor 110. The frame processor 110 can execute a first TCP adapter 120 and a plurality of TCP action routines 130a, 130b,
- 25 130c in parallel. The TCP adapter 120 recognizes a TCP request stream from the source 105. The actual protocol is processed by the action routines 130. Each action routine 130a can execute a plurality of frame processors 140a, 150a, 160a in parallel to process the frame sequence from the source 105. Initial frame processor 140a invokes a second TCP adapter 142a which recognizes the frames from the TCP
- 30 port. Attributes from the second TCP adapter 142a are fed through a TCP frame

-19-

processor 140a to a second TCP action routine 144a. The second TCP action routine 144a places a sequence of parsed frames in a pipe output 146a. The encompassing TCP action routine 130a has, for example, invoked an SMTP frame processor 150a which removes the sequence of frames from the TCP pipe output 146a into an SMTP pipe input 152a. The sequence of frames are processed by the SMTP frame processor 150a using an SMTP action routine 154a which places a parsed sequence of frames into an SMTP pipe output 156a. In addition, a MIME frame processor 160a receives the SMTP output in its pipe input 162a. These sequence of frames are processed by the MIME frame processor 160a using a MIME action routine 164a. Using a MIME pipe output 166a, additional frame processors can further process the sequence of frames.

FIG. 10 is a block diagram of a memory construct for the system of FIG. 9. A memory store for the application route 210 and the memory store for the first TCP adapter 220 are encompassed by a common memory context 215. Each of the TCP action routines 130a, 130b, 130c of FIG. 9 have a respective memory store 230a, 230b, 230c which each have a respective memory context 235a, 235b, 235c as illustrated. In addition, the TCP frame processor 140a, SMTP frame processor 150a, and the MIME frame processor 160a have a respective memory store 240a, 250a, 260a which are encompassed by a respective memory context 245a, 255a, 265a.

Benefits from the System

Control structures for recursion and iteration are some of the most difficult and error prone aspects of computer programming. Multi-threading is likewise very difficult to incorporate into programs. By encapsulating these aspects, the processing engine 10 makes the remaining tasks of developing a computer procedure for processing an information flow much simpler than current practice normally allows. Additionally, by facilitating a logical separation of tasks, the use of the processing engine 10 can result in computer programs with highly independent,

- 20 -

reusable components. Hence, the processing engine 10 can make it easier to process information flows.

Because the processing engine 10 is very regular and compact, it can be implemented in a small and fast computational routine. In preferred embodiments, the processing engine 10 may be significantly faster than the techniques used in the prior art.

The processing engine 10 is also universal. Typically, several layers of information flows must be processed within a computer, wherein upper layers are logically or physically encapsulated within the layer below them. Similarly, a computer may need to process many information flows in parallel at the same time. In each of these cases or even in the combined case, a single instance of the frame processor 12 can be used as a core process, given appropriate adapter and action routines and utilizing standard techniques for task and memory management. Hence use of the processing engine greatly reduce the complexity of information flow processing in computing by decreasing the number of computational components involved, providing a common architecture and single core process for the remaining components, and simplifying each remaining component.

The processing engine 10 makes it possible to efficiently harness more complex forms of information flow than current practice allows. Because the method encapsulates recursion, it makes it easy for the full recursive power of frames to be used in information flows. This in turn adds another dimension to the design of such flows without increasing the difficulty of creating processing routines to handle them. Therefore designers of information flows can put more complexity into their designs because the system can easily be used to process them. For example, the International Standards Organization (ISO) specification of Abstract Syntax Notation 1 (ASN.1) and the Basic Encoding Rules is an extremely powerful and efficient protocol which has languished because the protocol is difficult to understand and process. The processing engine 10 makes it easier to use this protocol.

Example Uses for the System

When a computer program executes in a computer, information is initially passed to it in various forms from the computer operating system and other sources. These sources may include, but are not limited to, command line arguments, environment variables, configuration files and the results of system calls. By treating these sources as frames, possibly nested, using adapter routines specific to each source, the main portion of the computer program can be insulated from direct contact with its operating environment and therefore be more easily ported between environments. In particular, values can be placed on the attribute stack at start-up by associating the values with the 'ROOT' frame.

In a complex computer operating system environment, there are normally many computer processes running programs that are engaged in the task of communicating with other computers across a network. Each program usually deals with a single type of information flow and has unique, often large and complex, processing routines. An implementation done using the processing engine can combine those programs into a single program run in one process with a single instance of the processing engine and multiple sets of (generally simpler and smaller) adapter and action routines. The resulting process executes much faster and the program is smaller and easier to maintain than the programs it replaces.

In a networked computing environment, information flows arrive off the physical network in low-level encodings of bits which are progressively processed by different layers of computer programs and/or routines. By using the processing engine, the layers can be kept distinct (through different adapter and action routines) but the overall processing can be simplified, made more compact, and speeded up by incorporating the invention in the processing core. In preferred embodiments, the computer system operates 2-3 times faster than directly programmed approaches used in the prior art and is about 20-30 times smaller when combined with an embodiment which operates with the operating system.

Some information flows represent information to be displayed on a screen or to be printed. Special purpose interpreters are typically used to accomplish this

- 22 -

rendering. The invention can be used to develop more general purpose interpreters with a common core engine if the information flows can be represented as sequences of frames.

Another example is security encryption, where the technique is naturally
5 processed in layers. Those layers can be expressed as frames which are easily handled by the processing engine.

The processing engine 10 is sufficiently compact that it can be easily implemented in hardware or firmware. This can make the processing engine very useful for inclusion in any special purpose device that needs to communicate with
10 other devices, including, but not limited to, electronic appliances, machine tools and medical instruments. Additionally, such devices can use the processing engine for processing internal information flows. For example, a printer can use the technique described in the preceding paragraph with a ROM-based implementation of the invention.

15 *Detailed Protocol Examples*

Appendix A attached hereto is a complete frame processor implementation for a simple sample protocol. A lexical scanner (lexer) breaks the input protocol into tokens. In this example, the lexer understands a parentheses-based protocol, an example of which might look like this: (F1(a1 v1)(a2 v2)). This can be read as
20 'Frame' instance named 'F1' that has two 'Attributes', 'a1' and 'a2', whose values are 'v1' and 'v2' respectively. Frames can nest, so (F1(a1 v1)(F2(a2 vs))(a3 vd)) is valid (also attributes and frames can be mixed in any order at any level). However, the lexer does not need to handle hierarchy, it only needs to recognize the tokens and pass them on to the next procedure.

25 This example core engine only deals with symbolic tokens passed by the lexer and not with the protocol itself. Hence the core engine can handle any protocol that conforms to its semantic model, given a suitable lexer. The only tokens acceptable to the frame processor are: FRAME_TYPE, END_OF_FRAME, and ATTRIBUTE. The frame processor forms a hierarchical structure based upon

- 23 -

the input from the lexer and maintains internal data structures (stack and heap) that comprise a scoped symbol table of attribute-value pairs. This enables scoped inheritance of attributes by nested frames.

The 'action' routines consist of 'main', which calls the frame processor (FREDI) after performing any necessary initialization, *e.g.*, setting STDIN and STDOUT, and has 'callback' routines which the frame processor calls at defined points in protocol processing: when a frame is initialized, when a sub-frame is encountered, and when the end of a frame is encountered. There is a sample dispatch table in the code. The callback routines can access attribute values with a 'get' routine that returns the properly scoped value and then do whatever. The action routines are completely independent of the details of the input protocol, insulated by the frame processor.

The following files are provided in Appendix A:

- i) makefile;
- 15 ii) fred.i.y - bison grammar;
- iii) fred.i.h;
- iv) paren.l - parentheses protocol lexer;
- v) frame.c - sample main, dispatch table, and action routines
(Two frame types are defined, and they each have three action
20 routines. The action routines print out the current values of
some attributes); and
- vi) test.in - a sample input file (the lexer ignores extra white
space)

Appendix B attached hereto is a programming example of what is needed to
25 attach the frame processor (FREDI) to a real-world protocol. Note that the file
fred.i.h provides documentation on how to use the frame processor.

Appendix C attached hereto is the source version of a binary library that
provides the core frame processing engine for the example of Appendix B. The
attached version of the source code handles synthetic attributes including the
30 'upAttribute' facility. As provided, the engine is nearly reentrant because the static

- 24 -

storage requirements have been reduced to about 100 bytes. These 100 bytes can be made part of each context to make the routines reentrant.

Equivalents

While this invention has been particularly shown and described with
5 referenced to preferred embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the spirit and scope of the invention as defined by the appended claims.

These and all other equivalents are intended to be encompassed by the
10 following claims.

-25-

APPENDIX A

```
# makefile for fred i - presumes GNU tools: make,gcc,bison,flex
```

```
YACC=bison -y
```

```
YFLAGS=-d -v
```

```
LEX=flex
```

```
LOADLIBES= -lfl
```

```
fred i: paren.o frame.o
```

```
frame.o fred i.o: fred i.h
```

```
paren.l: y.tab.h
```

```
y.tab.h: fred i.o
```

```
-----
```

```
%{
```

```
/* fred i.y - FFrame EDI core engine version 0.1
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include "fred i.h"
```

-26-

```
static void doInitFrame(char *pcz);
static void doFinisFrame(char *pcz);
static char *getCurrentFrame(void);
static void pushFrame(char *pcz);
static void popFrame(void);
static void processAttribute(char *pc);

static char acAttributes[128*1024];
static char *pcAttributesEnd = acAttributes + sizeof acAttributes;

static struct sFrameEntry {
    char *pcAttributes;
    char aczFrameType[60];
} asFrameEntry[32];

static int iFrame = 0;

%}

%union {
    char *string;
}

%token <string> FRAME_TYPE
%token <string> ATTRIBUTE
%token END_OF_FRAME

%type <string> frames frame subframes subframe
```

-27-

%%

frames:frame

| frames frame

;

frame:FRAME_TYPE {doInitFrame(\$1);} subframes END_OF_FRAME

{doFinisFrame(\$1);}

| FRAME_TYPE END_OF_FRAME {initFrame(\$1);finisFrame(\$1);free(\$1);}

;

subframes:subframe

| subframes subframe

;

subframe:ATTRIBUTE {processAttribute(\$1);}

| frame

;

%%

yyerror(char *pcz)

{

int iError = 0;

frameParseError(iError, pcz);

}

char *getAttributeValue(char *pcz)

{

char *pczCurrent = asFrameEntry[iFrame].pcAttributes;

-28-

```
while (pczCurrent < pcAttributesEnd) {
    if (strcmp(pczCurrent, pcz)) {
        pczCurrent += strlen(pczCurrent) + 1;
        pczCurrent += strlen(pczCurrent) + 1;
    }
    else {
        return pczCurrent + strlen(pczCurrent) + 1;
    }
}

return NULL;
}

static void processAttribute(char *pc)
{
    int iAttrNameLen = strlen(pc);
    int iAttrLen = iAttrNameLen + strlen(pc+iAttrNameLen+1) + 2;

    asFrameEntry[iFrame].pcAttributes -= iAttrLen;
    memcpy(asFrameEntry[iFrame].pcAttributes, pc, iAttrLen);
    free(pc);
}

static void doInitFrame(char *pcz)
{
    char *pczCurrent;

    if (pczCurrent = getCurrentFrame()) checkFrame(pczCurrent);
    pushFrame(pcz);
}
```

-29-

```
    initFrame(pcz);
}

static void doFinisFrame(char *pcz)
{
    finisFrame(pcz);
    popFrame();
    free(pcz);
}

static char *getCurrentFrame(void)
{
    if (iFrame) {
        return asFrameEntry[iFrame].aczFrameType;
    }

    asFrameEntry[0].pcAttributes = pcAttributesEnd;
    return NULL;
}

static void pushFrame(char *pcz)
{
    iFrame++;
    asFrameEntry[iFrame].pcAttributes = asFrameEntry[iFrame-1].pcAttributes;
    strcpy(asFrameEntry[iFrame].aczFrameType, pcz);
}

static void popFrame(void)
{

```

WO 97/12339

PCT/US96/14663

-30-

```
    if (iFrame) iFrame--;  
}
```

```
/* fred.i.h */
```

```
char *getAttributeValue(char *pcz);
```

```
void frameParseError(int iError, char *pczErrorMsg);
```

```
void initFrame(char *pcz);
```

```
void checkFrame(char *pcz);
```

```
void finisFrame(char *pcz);
```

```
%{
```

```
/* paren.l - lexical scanner for a simple parentheses based protocol */
```

```
#include "y.tab.h"
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int iParenCount, iWordCount;
```

```
char aczAttrName[256];
```

WO 97/12339

PCT/US96/14663

-31-

%}

%s PARSE_SIMPLE_FRAME_TYPE PARSE_FRAME_TYPE PARSE_ATTRIBUTE

WORD [A-Za-z0-9]+

OPEN_PAREN "("

CLOSE_PAREN ")"

WS [\n\t]

%%

{WS} {

;

}

<INITIAL>{OPEN_PAREN}{WS}*{WORD}{WS}*{CLOSE_PAREN} {

BEGIN(PARSE_SIMPLE_FRAME_TYPE);

yyless(0);

}

<INITIAL>{OPEN_PAREN}{WS}*{WORD}{WS}*{OPEN_PAREN} {

iParenCount = 0;

BEGIN(PARSE_FRAME_TYPE);

yyless(0);

}

<PARSE_FRAME_TYPE>{OPEN_PAREN} {

if (iParenCount) {

yyless(0);

-32-

```
BEGIN(INITIAL);
}

iParenCount++;
}

<PARSE_FRAME_TYPE>{WORD} {
    yylval.string = strdup(yytext);
    return (FRAME_TYPE);
}

<PARSE_SIMPLE_FRAME_TYPE>{WORD} {
    yylval.string = strdup(yytext);
    BEGIN(INITIAL);
    return (FRAME_TYPE);
}

<INITIAL>{OPEN_PAREN}{WS}*{WORD}{WS}+{WORD}{WS}*{CLOSE_PAREN} {
    iWordCount = 0;
    BEGIN(PARSE_ATTRIBUTE);
    yyless(0);
}

<PARSE_SIMPLE_FRAME_TYPE,PARSE_ATTRIBUTE>{OPEN_PAREN} {
    ;
}

<PARSE_ATTRIBUTE>{CLOSE_PAREN} {
    BEGIN(INITIAL);
```

-33-

}

<PARSE_ATTRIBUTE>{WORD} {

if (iWordCount) {

int iAttrLen = strlen(aczAttrName);

yylval.string = malloc(yyleng+iAttrLen+2);

strcpy(yylval.string, aczAttrName);

strcpy(yylval.string+iAttrLen+1, yytext);

yyleng = yyleng + iAttrLen + 1;

return (ATTRIBUTE);

}

else {

iWordCount++;

strcpy(aczAttrName, yytext);

}

}

<INITIAL>{OPEN_PAREN}{WS}*{CLOSE_PAREN} {

;

}

<INITIAL>{CLOSE_PAREN} {

return (END_OF_FRAME);

}

%%

-34-

```
/* frame.c - example action routines for fred-i-based protocol processing*/
```

```
#include <string.h>
```

```
#include <stdio.h>
```

```
#include "fredi.h"
```

```
int F1_init(char*pcz);
```

```
int F1_check(char*pcz);
```

```
int F1_finis(char*pcz);
```

```
int F2_init(char*pcz);
```

```
int F2_check(char*pcz);
```

```
int F2_finis(char*pcz);
```

```
struct s_fte{
```

```
    char*aczFrameType;
```

```
    int (*pfinit)(char*pcz);
```

```
    int (*pfCheck)(char*pcz);
```

```
    int (*pfFinis)(char*pcz);
```

```
} as_fte[]={
```

```
    {"F1", F1_init, F1_check, F1_finis},
```

```
    {"F2", F2_init, F2_check, F2_finis}
```

```
};
```

-35-

```
#define AS_FTE_NUM (sizeof as_fte / sizeof as_fte[0])
```

```
void main()
```

```
{  
    yyparse();  
    printf("Done!\n");  
}
```

```
void frameParseError(int iError, char *pczErrorMsg)
```

```
{  
    printf("Parse error: %d %s\n", iError, pczErrorMsg);  
}
```

```
void initFrame(char *pcz)
```

```
{  
    int iFrame;  
  
    if ((iFrame = getFrameType(pcz)) >= 0) (*(as_fte[iFrame].pfInit))(pcz);  
}
```

```
void checkFrame(char *pcz)
```

```
{  
    int iFrame;  
  
    if ((iFrame = getFrameType(pcz)) >= 0) (*(as_fte[iFrame].pfCheck))(pcz);  
}
```

```
void finisFrame(char *pcz)
```

```
{
```

-36-

```
int iFrame;

if ((iFrame = getFrameType(pcz)) >= 0) (*(as_fte[iFrame].pfFinis))(pcz);
}

int F1_init(char *pcz)
{
    printf("F1_init\n");
    return 0;
}

int F1_check(char *pcz)
{
    char *pczAttrValue;

    printf("F1_check\n");
    if (pczAttrValue = getAttributeValue("a1")) printf("a1=%s\n",
pczAttrValue);
    if (pczAttrValue = getAttributeValue("a2")) printf("a2=%s\n",
pczAttrValue);
    return 0;
}

int F1_finis(char *pcz)
{
    char *pczAttrValue;

    printf("F1_finis\n");
    if (pczAttrValue = getAttributeValue("a1")) printf("a1=%s\n",
```

-37-

```
pczAttrValue);
    if (pczAttrValue = getAttributeValue("a2")) printf("a2=%s\n",
pczAttrValue);
    return 0;
}
```

```
int F2_init(char *pcz)
{
    printf("F2_init\n");
    return 0;
}
```

```
int F2_check(char *pcz)
{
    char *pczAttrValue;

    printf("F2_check\n");
    if (pczAttrValue = getAttributeValue("a1")) printf("a1=%s\n",
pczAttrValue);
    if (pczAttrValue = getAttributeValue("a3")) printf("a3=%s\n",
pczAttrValue);
    if (pczAttrValue = getAttributeValue("a4")) printf("a4=%s\n",
pczAttrValue);
    return 0;
}
```

```
int F2_finis(char *pcz)
{
    char *pczAttrValue;
```

-38-

```

    printf("F2_finis\n");
    if (pczAttrValue = getAttributeValue("a1")) printf("a1=%s\n",
pczAttrValue);
    if (pczAttrValue = getAttributeValue("a3")) printf("a3=%s\n",
pczAttrValue);
    if (pczAttrValue = getAttributeValue("a4")) printf("a4=%s\n",
pczAttrValue);
    return 0;
}

```

```

int getFrameType(char *pcz)
{
    int i;

    for (i=0;i<=AS_FTE_NUM;i++) {
        if (strcmp(pcz,as_fte[i].aczFrameType) == 0) return i;
    }

    return -1;
}

```

```

(F1
(a1 v1)
(a2 v2)
(F2
(a3 v3)
(a4 v4)

```

WO 97/12339

PCT/US96/14663

-39-

)

(F2

(a1 v11)

(a4 v44)

)

(a2 v22)

)

-40-

APPENDIX B

```
# makefile for Sun using libfredi.a
```

```
# any ANSI C compiler should do
```

```
CC= gcc
```

```
peb_act: peb_act.o peb_act1.o peb_act2.o peb_gen.o peb_lex.o libfredi.a
        ${CC} -o peb_act peb_act.o peb_act1.o peb_act2.o peb_gen.o peb_lex.o -L.
        -lfredi -ll
```

```
peb_act.o peb_act1.o peb_act2.o peb_lex.o: fred_i.h
```

```
peb_act.o peb_act1.o peb_act2.o peb_gen.o: peb_act.h
```

```
clean:
```

```
        rm -f peb_act *.o core
```

```
***** separator *****
```

```
/*
```

Intro

Fredi-based processors have a simple structure that facilitates the partition of functions into modules that are themselves relatively simple, hopefully decreasing

-41-

the overall complexity of building and maintaining complex protocol processing programs.

Fredi views all input as a balanced recursive set of 'frames' and maintains a scoped symbol table to support processing them. Frames have any number of attributes which themselves can be frames. Fredi only deals in 'abstract' syntax; the programmer provides the translation to and from the 'transfer' syntax used 'on the wire'. The programmer also provides the semantic processing.

The standard fredi routines are encapsulated as a library of functions that require certain 'callback' routines to be provided by the programmer. Thus the fredi routines can be viewed as 'middleware'.

Specifically, the programmer must create:

- . A main routine that sets stdin, stdout & stderr and calls entry point yyparse()
- . A 'lexer' routine with entry point yylex() that tokenizes the input into:
 - FRAME_TYPE (start of frame)
 - ATTRIBUTE (a scalar attribute/value pair)
 - END_OF_FRAME
 - CHECKPOINT (used to establish a known and recoverable fredi symbol table state)

-42-

. Action routines to be called by fredf when frames start and finish processing and

when parse errors occur. Typically the programmer also creates a 'generator' routine

as well to output the transfer syntax based upon the abstract syntax and semantics.

The programmer gets from fredf:

- . Routines to 'put' frames and attributes from the lexer.
- . Routines to 'set' attributes from action routines.
- . A Routine to 'get' the current frame from action routines.
- . Routines to 'get' attributes from action routines.
- . Miscellaneous other useful routines.
- . Define's for fredf data types - only a few currently...

*/

/* API's implemented by action routines for the FREDf support routines */

void frParseErr(int iErr, char *pczErrMsg);

/* iErr is always 0 currently, pczErrMsg is whatever comes from yyparse()

*/

void frChkFrm(char *pczFrmType, char *pczSubFrmType);

/*

Called when a new frame is encountered; pczFrameType is the current (enclosing)

-43-

frame type, pczSubFrmType is the new frame type, frame level is at the
current

(enclosing) frame.

*/

void frInitFrm(char *pczSubFrmType, char *pczFrmType);

/*

Called after frChkFrm when a new frame is encountered; pczSubFrmType is
the

current (new) frame type, pczFrmType is the enclosing frame type, frame
level

is at the current (new) frame - no attributes will have been set.

*/

void frFinFrm(char *pczFrmType);

/*

Called when the end of a frame is encountered, pczFrmType is the current
frame

type, frame level is at the current frame. All frame attributes are
available

and will be 'popped' out of fred's symbol table when this call returns.

*/

/* API's implemented by the FREDI support routines for action routines */

/*

Note: fred implements simple versions of some routines when only 'string'
versions of attributes are of interest. However, fred can handle
arbitrary

-44-

values up to about 8K (currently). fredri can handle a pointer as the value of

an attribute, so BLOBs should be passed as pointers. Future versions of fredri will do 'garbage collection' on such pointers, but currently the programmer is responsible for deallocating memory.

*/

/*

Get: fredri has scoped memory and will 'get' the most tightly scoped attribute

value. The 'get next' routines walk through all attribute values in scoped order.

*/

char *frGetAttrVal(char *pczAttrNm);

/*

Returns a pointer to a copy of the string value of the attribute whose name is pointed to by pczAttrNm.

*/

char *frGetAttrValBin (

char *pczAttrNm, /*in*/

void *pvAttrValBin, /*out - pointer to lvalue allocated by caller */

int *piAttrValType,

int *piAttrValBinLen,

int *piAttrStat

);

/*

Returns a pointer to a copy of the string value of the attribute whose

-45-

name is pointed to by pczAttrNm. Also copies the binary value of the attribute

(if present) to *pvAttrValBin if pvAttrValBin is not NULL.

*/

```
int frSetAttr(char *pczAttrNm, char *pczAttrValStr);
```

/*

Copies the value pointed to by pczAttrValStr to the string value of the attribute whose name is pointed to by pczAttrNm. Returns 0 always (for now).

*/

```
int frSetAttrBin (
```

```
char *pczAttrNm,
```

```
char *pczAttrValStr,
```

```
void *pvAttrValBin,
```

```
int iAttrValType,
```

```
int iAttrValBinLen,
```

```
int iAttrStat
```

```
);
```

/*

Copies the value pointed to by pczAttrValStr to the string value of the attribute whose name is pointed to by pczAttrNm. If pvAttrValBin is not NULL,

copies iAttrValBinLen bytes of the value pointed to by pvAttrValBin to the

binary value of the attribute. Returns 0 always (for now).

*/

-46-

```
int frUpAttr(char *pczAttrNm, char *pczAttrValStr);
/*
```

Identical to the 'set' routine but positions the attribute in the scope of the

'enclosing' frame. This a very useful and general way of 'returning' values

from a frame, i.e. resolving frames into scalars. The returned values can be

used to control processing in the enclosing frame or propagated to higher levels.

```
*/
```

```
int frUpAttrBin (
    char *pczAttrNm,
    char *pczAttrValStr,
    void *pvAttrValBin,
    int iAttrValType,
    int iAttrValBinLen,
    int iAttrStat
);
```

```
/* See above */
```

```
int frGetNxtAttr (
    int *piFrmLvl,
    char **ppczFrmType,
    char **ppczAttrNm,
    char **ppczAttrValStr
);
/*
```

-47-

Returns the 'next' attribute and information about the frame to which it belongs.

Sets pointers to copies of the relevant strings. Returns 0 normally;
returns -1

on the call subsequent to the call in which the last attribute was
returned.

If any input parameter is NULL, no attempt is made to return a value
for that parameter.

*/

```
int frGetNxtAttrBin (
    int *piFrmLvl,
    char **ppczFrmType,
    char **ppczAttrNm,
    char **ppczAttrValStr,
    void **ppvAttrValBin,
    int *piAttrValType,
    int *piAttrValBinLen,
    int *piAttrStat
);
/*
```

Returns the 'next' attribute and information about the frame to which it belongs.

Sets pointers to copies of the relevant values and updates integers.
Returns 0

normally; returns -1 on the call subsequent to the call in which the
last attribute

was returned. If any input parameter is NULL, no attempt is made to
return a value

-48-

for that parameter.

*/

int frGetNxtAttrReset(void);

/*

Resets the 'attribute walk' such that the next call to a 'get next' routine

retrieves the first one. Called automatically when a 'get next' routine returns -1

or when the end of a frame is reached. Always returns 0 (currently).

*/

char *frGetFrm (

void **ppvUsrHndl,

int *piOffStrt,

int *piOffNxt

);

/* Returns the current frame type and its information - see below. */

/*

Note: there may be a 'walk' function added for frames in the future analogous to

the attribute walk and possibly a 'set' function.

*/

int frSkipFrm(void);

/*

Causes fredt to skip to the end of the frame, including skipping all

-49-

contained

frames. The end of frame routine will be called for the frame. Allows the programmer

to 'bail out' if things go bad, e.g. database failure, and may have other uses...

*/

/* API's implemented by the FREDI support routines for lexer */

```
int frPutFrm (
    char *pczFrmType,
    void *pvUsrHndl,
    int iOffStrt,
    int iOffNxt
);
/*
```

Prepares a new frame. Copies the frame type from the string pointed to by pczFrmType.

Stores the other values, regardless of contents. Contemplated uses include identifying

the starting and next offsets of the raw input in a file or buffer such that a

raw frame including its contents can be sent transparently somewhere else. Always

returns 0 (currently). The frame is not 'committed' until the lexer returns with

FRAME_TYPE.

*/

-50-

```
int frUpdtFrm (  
    void *pvUsrHndl,  
    int iOffStrt,  
    int iOffNxt  
);  
/*
```

Updates values in the current frame. Contemplated to be called when end of frame is

reached in order to update values. If any of the integer arguments is set to -1 before

the call, or if pvsrHndl is set to NULL, then that value is not updated.
*/

```
int frPutAttr(char *pczAttrNm, char *pczAttrValStr);
```

/* Prepares a new attribute in the current frame. Analogous to 'set attribute' above. */

```
int frPutAttrBin (  
    char *pczAttrNm,  
    char *pczAttrValStr,  
    void *pvAttrValBin,  
    int iAttrValType,  
    int iAttrValBinLen,  
    int iAttrStat
```

```
);  
/* see above */
```

/* values shared by all */

-51-

```
enum eAttrValType {FR_STR_TYPE, FR_BIN_TYPE};
```

```
/* values for communicating from the lexer to the fred core engine */
```

```
#ifndef YYSTYPE
```

```
#define YYSTYPE int
```

```
#endif
```

```
#define FRAME_TYPE 258
```

```
#define ATTRIBUTE 259
```

```
#define END_OF_FRAME 260
```

```
#define CHECKPOINT 261
```

```
extern YYSTYPE yylval;
```

```
***** separator *****
```

```
/* peb_act.h - prototypes for action routines */
```

```
int root_check(char *pczFrameType, char *pczSubFrameType);
```

```
int default_finis(char *pczSubFrameType);
```

```
int REQ_AU1_init(char *pczSubFrameType, char *pczFrameType);
```

```
int REQ_AU1_check(char *pczFrameType, char *pczSubFrameType);
```

```
int REQ_AU1_finis(char *pczSubFrameType);
```

```
void sendFrameType(char *pczFrameType);
```

```
void sendAttribute(char *pczAttrName, char *pczAttrValue);
```

```
void sendEndOfFrame(void);
```

```
void sendEndOfContent(void);
```

```
***** separator *****
```

-52-

```
/* peb_act.c - example action routines for fred-i-based protocol processing
*/
/*      using the Pebble protocol
*/

#include "peb_act.h"

int yyparse(void);

void main()
{
    yyparse();
    sendEndOfContent();
}

***** separator *****
/* peb_act1.c - example action routines for fred-i-based protocol processing
*/
/*      using the Pebble protocol
*/

#include <string.h>
#include <stdio.h>

#include "fredi.h"
#include "peb_act.h"

static int getFrameType(char *pczFrameType);

static struct s_fte {
```

-53-

```

char *aczFrameType;
int (*pfInit)(char *pczSubFrameType, char *pczFrameType);
int (*pfCheck)(char *pczFrameType, char *pczSubFrameType);
int (*pfFinis)(char *pczSubFrameType);
} as_fte[] = {
    {"default", NULL, NULL, default_finis},
    {"root", NULL, root_check, NULL},
    {"REQ/AU1", REQ_AU1_init, REQ_AU1_check, REQ_AU1_finis}
};

```

```

#define AS_FTE_NUM (sizeof as_fte / sizeof as_fte[0])

```

```

void frParseErr(int iErr, char *pczErrMsg)
{
    fprintf(stderr, "Parse error: %d %s\n", iErr, pczErrMsg);
}

```

```

void frInitFrm(char *pczSubFrameType, char *pczFrameType)
{
    int iFrame;

    if (as_fte[iFrame = getFrameType(pczSubFrameType)].pfInit)
        (*(as_fte[iFrame].pfInit))(pczSubFrameType, pczFrameType);
}

```

```

void frChkFrm(char *pczFrameType, char *pczSubFrameType)
{
    int iFrame;

```

-54-

```

if (as_fte[iFrame = getFrameType(pczFrameType)].pfCheck)
    (*(as_fte[iFrame].pfCheck))(pczFrameType, pczSubFrameType);
}

```

```

void frFinFrm(char *pczFrameType)

```

```

{
    int iFrame;

    if (as_fte[iFrame = getFrameType(pczFrameType)].pfFinis)
        (*(as_fte[iFrame].pfFinis))(pczFrameType);
}

```

```

static int getFrameType(char *pczFrameType)

```

```

{
    int i;

    for (i=1;i<AS_FTE_NUM;i++) {
        if (strcmp(pczFrameType,as_fte[i].aczFrameType) == 0) return i;
    }

```

```

    return 0;
}

```

```

***** separator *****

```

```

/* peb_act2.c - example action routines for fredt-based protocol processing

```

```

*/

```

```

/*      using the Pebble protocol

```

```

*/

```

-55-

```
#include <string.h>
```

```
#include <stdio.h>
```

```
#include "fredi.h"
```

```
#include "peb_act.h"
```

```
int root_check(char *pczFrameType, char *pczSubFrameType)
```

```
{
```

```
    if (pczSubFrameType[0] != 'R') {
```

```
        frSkipFrm();
```

```
        fprintf(stderr, "%s cannot contain %s\n", pczFrameType, pczSubFrameType);
```

```
    }
```

```
    return 0;
```

```
}
```

```
int default_finis(char *pczFrameType)
```

```
{
```

```
    char *pczAttrValue;
```

```
    char *pczFrameTypeTemp;
```

```
    int iFrameLevel;
```

```
    char *pczAttrName;
```

```
    fprintf(stderr, "Unrecognized Frame Type:%s\n", pczFrameType);
```

```
    while(!frGetNxtAttr(&iFrameLevel,&pczFrameTypeTemp,&pczAttrName,&pczAttrValue))
```

```
    {
```

```
        fprintf (
```

-56-

```
    stderr,
    " FrameLevel=%d; FrameType=%s; AttrName=%s; AttrValue=%s\n",
    iFrameLevel,
    pczFrameTypeTemp,
    pczAttrName,
    pczAttrValue
    );
}

return 0;
}

int REQ_AU1_init(char *pczSubFrameType, char *pczFrameType)
{
    if (strcmp(pczFrameType, "root")) {
        frSkipFrm();
        fprintf(stderr, "%s cannot be contained in %s\n", pczSubFrameType,
pczFrameType);
        return 0;
    }

    fprintf(stderr, "Frame Type:%s\n", pczSubFrameType);
    return 0;
}

int REQ_AU1_check(char *pczFrameType, char *pczSubFrameType)
{
    frSkipFrm();
    fprintf(stderr, "%s cannot contain %s\n", pczFrameType, pczSubFrameType);
```

-57-

```

    return 0;
}

int REQ_AU1_finis(char *pczFrameType)
{
    char *apczAttrName[] = {
        "AID", "ATP", "CID", "TDT", "TID", "TOT", "VID", "AXD", "CDM", "TTM"
    };

    int iAttrNames = sizeof(apczAttrName) / sizeof(apczAttrName[0]);
    int i;
    char *pczAttrValue;

    for (i=0; i<iAttrNames; i++) {
        if ((pczAttrValue = frGetAttrVal(apczAttrName[i])))
            fprintf(stderr, " %s:%s\n", apczAttrName[i], pczAttrValue);
    }

    sendFrameType("RSP/RSP");
    sendAttribute("RES", "1");
    sendAttribute("REF", "2");
    sendAttribute("TID", "3");
    sendAttribute("MSG", "Simulated Failure...");
    sendEndOfFrame();

    return 0;
}

***** separator *****
/* peb_gen.c - example generator routines for freda-based protocol processing

```

-58-

```
*/
/*      using the Pebble protocol
*/

#include <string.h>
#include <stdio.h>

#include "peb_act.h"

void sendFrameType(char *pczFrameType)
{
    if (strcmp(pczFrameType, "RSP/RSP")) {
        pczFrameType[3] = ':';
        printf("%s\t", pczFrameType);
    }
    else {
        printf("RSP:");
    }
}

void sendAttribute(char *pczAttrName, char *pczAttrValue)
{
    if (strcmp(pczAttrName, "RES")) {
        printf("%s:%s\t", pczAttrName, pczAttrValue);
    }
    else {
        printf("%s\t", pczAttrValue);
    }
}
```

-58/1-

```
void sendEndOfFrame(void)
```

```
{  
    printf("\n");  
}
```

```
void sendEndOfContent(void)
```

```
{  
    printf("\f");  
}
```

```
***** separator *****
```

```
%{
```

```
/* simple pebble lexer */
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include "fredi.h"
```

```
char aczAttrName[128];
```

```
char aczFrameType[128];
```

```
char aczAttr[128];
```

```
%}
```

```
%s SCAN_ALL GET_NAME DO_RSP_FRAME_TYPE DO_FRAME_TYPE  
DO_ATTRIBUTE CLEANUP
```

```
WS [ ]
```

-58/2-

ATTR_SEP [:]

ATTR_END \t

LINE_END \n

CONTENT_END \f

NAME [A-Z]+

VALUE [^\t\n\f]+([]+[^ \t\n\f]+)*

%%

{WS} {

;

}

<INITIAL>. {

yyless(0);

BEGIN(SCAN_ALL);

}

<SCAN_ALL>{CONTENT_END} {

return(0);

}

<SCAN_ALL>{LINE_END} {

return(END_OF_FRAME);

}

<SCAN_ALL>{NAME}{ATTR_SEP}{WS}*{VALUE}{WS}*{ATTR_END} {

yyless(0);

BEGIN (GET_NAME);

-58/3-

}

<GET_NAME>{NAME}{ATTR_SEP} {

if ((strcmp("RSP:", yytext) == 0)) {

BEGIN(DO_RSP_FRAME_TYPE);

}

else if (

(strcmp("REQ:", yytext) == 0)

|| (strcmp("CTX:", yytext) == 0)

) {

strcpy(aczFrameType, yytext);

aczFrameType[yyvaleng-1] = 0;

BEGIN(DO_FRAME_TYPE);

}

else {

strcpy(aczAttrName, yytext);

aczAttrName[yyvaleng-1] = 0;

BEGIN(DO_ATTRIBUTE);

}

}

<DO_ATTRIBUTE>{VALUE} {

frPutAttr(aczAttrName, yytext);

BEGIN(CLEANUP);

return(ATTRIBUTE);

}

<DO_RSP_FRAME_TYPE>{VALUE} {

int i;

-58/4-

```

frPutFrm("RSP/RSP", NULL, 0, 0);
strcpy(aczAttr, "RES:");
strcat(aczAttr, yytext);
for (i=strlen(aczAttr)-1;i>=0;i--) unput(aczAttr[i]);
BEGIN(SCAN_ALL);
return(FRAME_TYPE);
}

```

```

<DO_FRAME_TYPE>{VALUE} {
    strcat(aczFrameType,"/");
    frPutFrm(strcat(aczFrameType,yytext), NULL, 0, 0);
    BEGIN(CLEANUP);
    return(FRAME_TYPE);
}

```

```

<CLEANUP>{ATTR_END} {
    BEGIN(SCAN_ALL);
}

```

%%

```

int yywrap(void)
{
    return 1;
}

```

***** separator *****

The following lines are sample input of the Pebble protocol:

REQ:AUI AID:aid_string ATP:atp_const

- WO 97/12339

PCT/US96/14663

-58/5-

CID:cid_string TDT:tdt_date TID:tid_num TOT:tot_money
VID:vid_string AXD:axd_date CDM:cdm_string TTM:ttm_time
REQ:AUI AID:2aid_string ATP:atp_const
CID:cid_string TDT:tdt_date TID:tid_num TOT:tot_money
VID:vid_string AXD:axd_date CDM:cdm_string TTM:ttm_time

-58/6-

***** separator *****

The following lines are sample output of the Pebble protocol,
these lines are generated by the action routines and are
themselves processable as input:

RSP:1 REF:2 TID:3 MSG:Simulated Failure...

RSP:1 REF:2 TID:3 MSG:Simulated Failure...

-58/7-

APPENDIX C

```
/* fr_spt.h - FFrame EDI core engine version 0.3 */
```

```
/* API's implemented by FREDI support routines (fr_spt.c) for the FREDI core
*/
```

```
void frDoAttr(void);
```

```
void frDoInitFrm(void);
```

```
void frDoFinFrm(void);
```

```
void frDoChkPt(void);
```

```
***** separator *****
```

```
%{
```

```
/* fr_gram.y - FFrame EDI core engine version 0.3 */
```

```
#include "fr_spt.h"
```

```
%}
```

```
%token FRAME_TYPE ATTRIBUTE END_OF_FRAME CHECKPOINT
```

```
%%
```

```
frameList: frameList frame | frame;
```

```
frame: FRAME_TYPE {frDoInitFrm();} subFrameList END_OF_FRAME {frDoFinFrm();}
      | FRAME_TYPE END_OF_FRAME {frDoInitFrm();frDoFinFrm();}
```

-58/8-

```
| CHECKPOINT {frDoChkPt();}
;
```

```
subFrameList: subFrameList subFrame | subFrame
```

```
subFrame: ATTRIBUTE {frDoAttr();} | frame;
```

```
%%
```

```
***** separator *****
```

```
/* fr_spt.c - FFrame EDI core engine version 0.3 */
```

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include "fredi.h"
```

```
#include "fr_spt.h"
```

```
typedef struct sAttrEntStruct *psAttrEnt;
```

```
typedef struct sFrmEntStruct *psFrmEnt;
```

```
typedef struct sAttrEntStruct {
```

```
    psAttrEnt psAttrEntNxt;
```

```
    psFrmEnt psFrmEntOwn;
```

```
    char *pczAttrValStr;
```

```
    char *pczAttrValStrCpy;
```

```
    void *pvAttrValBin;
```

```
    void *pvAttrValBinCpy;
```

```
    int iAttrValType;
```

-58/9-

```
int iAttrValBinLen;  
int iAttrStat;  
} sAttrEnt;
```

```
static psAttrEnt psAttrEntTail = NULL;  
static psAttrEnt psAttrEntNew = NULL;  
static psAttrEnt psAttrEntWalk = NULL;
```

```
typedef struct sFrmEntStruct {  
    psFrmEnt psFrmEntNxt;  
    psAttrEnt psAttrEntHead;  
    psAttrEnt psAttrEntTail;  
    int iFrmLvl;  
    void *pvUsrHndl;  
    int iOffStrt;  
    int iOffNxt;  
} sFrmEnt;
```

```
static struct sFrmRootStruct {  
    sFrmEnt sFrmEntRoot;  
    char aczRoot[(64 - sizeof(sFrmEnt))];  
} sFrmRoot = {{NULL, NULL, NULL, 0, NULL, 0, 0}, "root"};
```

```
static psFrmEnt psFrmEntTail = &sFrmRoot.sFrmEntRoot;  
static psFrmEnt psFrmEntNew = NULL;  
static psFrmEnt psFrmEntFree = NULL;  
static int iSkip = 0;  
static int iChkPtNum = 0;
```

-58/10-

```
static int frDoPutAttrBin (  
    char *pczAttrNm,  
    char *pczAttrValStr,  
    void *pvAttrValBin,  
    int iAttrValType,  
    int iAttrValBinLen,  
    int iAttrStat,  
    psAttrEnt *ppsAttrEnt  
);  
  
static char *frGetFrmType(psFrmEnt psFrmEntTmp);  
static char *frGetAttrNm(psAttrEnt psAttrEntTmp);  
  
int yyerror(char *pcz)  
{  
    int iErr = 0;  
  
    frParseErr(iErr, pcz);  
    return 0;  
}  
  
int frSkipFrm(void)  
{  
    if (!iSkip) iSkip = 1;  
    return 0;  
}  
  
int frUpAttr(char *pczAttrNm, char *pczAttrValStr)  
{
```

-58/11-

```
void *pvAttrValBin = NULL;
int iAttrValType = FR_STR_TYPE;
int iAttrValBinLen = 0;
int iAttrStat = 0;
```

```
return frUpAttrBin (
    pczAttrNm,
    pczAttrValStr,
    pvAttrValBin,
    iAttrValType,
    iAttrValBinLen,
    iAttrStat
);
}
```

```
int frUpAttrBin (
    char *pczAttrNm,
    char *pczAttrValStr,
    void *pvAttrValBin,
    int iAttrValType,
    int iAttrValBinLen,
    int iAttrStat
)
{
    psAttrEnt psAttrEntUpNew;
    int iReturn;

    if ((iReturn = frDoPutAttrBin (
        pczAttrNm,
```

-58/12-

```

        pczAttrValStr,
        pvAttrValBin,
        iAttrValType,
        iAttrValBinLen,
        iAttrStat,
        &psAttrEntUpNew
    )
)) {
    return iReturn;
}

psAttrEntUpNew->psAttrEntNxt = psFrmEntTail->psFrmEntNxt->psAttrEntTail;
psAttrEntUpNew->psFrmEntOwn = psFrmEntTail->psFrmEntNxt;
psFrmEntTail->psFrmEntNxt->psAttrEntTail = psAttrEntUpNew;

if (psFrmEntTail->psAttrEntHead) {
    psFrmEntTail->psAttrEntHead->psAttrEntNxt = psAttrEntUpNew;
}
else {
    psAttrEntTail = psAttrEntUpNew;
}

if (!psFrmEntTail->psFrmEntNxt->psAttrEntHead)
    psFrmEntTail->psFrmEntNxt->psAttrEntHead = psAttrEntUpNew;

return 0;
}

int frSetAttr(char *pczAttrNm, char *pczAttrValStr)

```

-58/13-

```
{  
    void *pvAttrValBin = NULL;  
    int iAttrValType = FR_STR_TYPE;  
    int iAttrValBinLen = 0;  
    int iAttrStat = 0;
```

```
    return frSetAttrBin (  
        pczAttrNm,  
        pczAttrValStr,  
        pvAttrValBin,  
        iAttrValType,  
        iAttrValBinLen,  
        iAttrStat  
    );  
}
```

```
int frSetAttrBin (  
    char *pczAttrNm,  
    char *pczAttrValStr,  
    void *pvAttrValBin,  
    int iAttrValType,  
    int iAttrValBinLen,  
    int iAttrStat  
)  
{  
    psAttrEnt psAttrEntTmp;  
    int iReturn;  
  
    if ((iReturn = frDoPutAttrBin (
```

-58/14-

```
        pczAttrNm,
        pczAttrValStr,
        pvAttrValBin,
        iAttrValType,
        iAttrValBinLen,
        iAttrStat,
        &psAttrEntTmp
    )
)) {
    return iReturn;
}

psFrmEntTail->psAttrEntTail = psAttrEntTmp;
psAttrEntTmp->psFrmEntOwn = psFrmEntTail;
psAttrEntTmp->psAttrEntNxt = psAttrEntTail;
psAttrEntTail = psAttrEntTmp;
return 0;
}

int frPutAttr(char *pczAttrNm, char *pczAttrValStr)
{
    void *pvAttrValBin = NULL;
    int iAttrValType = FR_STR_TYPE;
    int iAttrValBinLen = 0;
    int iAttrStat = 0;

    return frPutAttrBin (
        pczAttrNm,
        pczAttrValStr,
```

WO 97/12339

PCT/US96/14663

-58/15-

```
        pvAttrValBin,
        iAttrValType,
        iAttrValBinLen,
        iAttrStat
    );
}

int frPutAttrBin (
    char *pczAttrNm,
    char *pczAttrValStr,
    void *pvAttrValBin,
    int iAttrValType,
    int iAttrValBinLen,
    int iAttrStat
)
{
    if (iSkip) return 0;
    if (psAttrEntNew) free(psAttrEntNew);

    return frDoPutAttrBin (
        pczAttrNm,
        pczAttrValStr,
        pvAttrValBin,
        iAttrValType,
        iAttrValBinLen,
        iAttrStat,
        &psAttrEntNew
    );
}
```

-58/16-

```
int frDoPutAttrBin (
    char *pczAttrNm,
    char *pczAttrValStr,
    void *pvAttrValBin,
    int iAttrValType,
    int iAttrValBinLen,
    int iAttrStat,
    psAttrEnt *ppsAttrEnt
)
{
    psAttrEnt psAttrEntTmp;
    int iAttrNmLen = strlen(pczAttrNm);
    int iAttrValStrLen = strlen(pczAttrValStr);
    int iAttrValStrOff;
    int iMalloc;

    iAttrValStrOff = iMalloc = sizeof(sAttrEnt) + 2*(iAttrNmLen+1);
    if (pczAttrValStr) iMalloc += 2*(iAttrValStrLen+1);
    psAttrEntTmp = malloc(iMalloc);
    memset(psAttrEntTmp, 0, iMalloc);
    strcpy((char *)psAttrEntTmp + sizeof(sAttrEnt), pczAttrNm);

    if (pczAttrValStr) {
        psAttrEntTmp->pczAttrValStr = (char *)psAttrEntTmp + iAttrValStrOff;
        strcpy(psAttrEntTmp->pczAttrValStr, pczAttrValStr);
        psAttrEntTmp->pczAttrValStrCpy = psAttrEntTmp->pczAttrValStr +
iAttrValStrLen + 1;
    }
}
```

-58/17-

```

if (pvAttrValBin && (iAttrValBinLen >= 0)) {
    psAttrEntTmp->pvAttrValBin = malloc(iAttrValBinLen + 1);
    memcpy(psAttrEntTmp->pvAttrValBin, pvAttrValBin, iAttrValBinLen);
    psAttrEntTmp->iAttrValBinLen = iAttrValBinLen;
    *((char *)psAttrEntTmp->pvAttrValBin + iAttrValBinLen) = 0;
    psAttrEntTmp->pvAttrValBinCpy = malloc(iAttrValBinLen + 1);
}

psAttrEntTmp->iAttrValType = iAttrValType;
psAttrEntTmp->iAttrStat = iAttrStat;
*ppsAttrEnt = psAttrEntTmp;
return 0;
}

void frDoAttr(void)
{
    if (!psAttrEntNew) return;
    if (iSkip) return;

    if (!psFrmEntTail->psAttrEntHead)
        psFrmEntTail->psAttrEntHead = psAttrEntNew;

    psAttrEntNew->psAttrEntNxt = psAttrEntTail;
    psFrmEntTail->psAttrEntTail = psAttrEntNew;
    psAttrEntNew->psFrmEntOwn = psFrmEntTail;
    psAttrEntTail = psAttrEntNew;
    psAttrEntNew = NULL;
}

```

-58/18-

```

char *frGetAttrVal(char *pczAttrNm)
{
    return frGetAttrValBin (
        pczAttrNm,
        NULL,
        NULL,
        NULL,
        NULL
    );
}

```

```

char *frGetAttrValBin (
    char *pczAttrNm,    /*in*/
    void *pvAttrValBin, /*out - pointer to lvalue allocated by caller or NULL
    */
    int *piAttrValType,
    int *piAttrValBinLen,
    int *piAttrStat
)
{
    psAttrEnt psAttrEntNxt = psAttrEntTail;
    char *pczAttrValStrCpy = NULL;

    while (psAttrEntNxt) {
        if (strcmp(pczAttrNm, (char *)psAttrEntNxt + sizeof(sAttrEnt))) {
            psAttrEntNxt = psAttrEntNxt->psAttrEntNxt;
        }
        else {
            if (piAttrValType) *piAttrValType = psAttrEntNxt->iAttrValType;

```

-58/19-

```

    if (piAttrValBinLen) *piAttrValBinLen = psAttrEntNxt->iAttrValBinLen;
    if (piAttrStat) *piAttrStat = psAttrEntNxt->iAttrStat;

    if (psAttrEntNxt->pczAttrValStr)
        pczAttrValStrCpy = strcpy(psAttrEntNxt->pczAttrValStrCpy,
psAttrEntNxt->pczAttrValStr);

    if (pvAttrValBin && psAttrEntNxt->pvAttrValBin)
        memcpy(pvAttrValBin, psAttrEntNxt->pvAttrValBin,
psAttrEntNxt->iAttrValBinLen + 1);

    return pczAttrValStrCpy;
}
};

*piAttrValType = 0;
*piAttrValBinLen = 0;
*piAttrStat = 0;
return NULL;
}

int frGetNxtAttr (
    int *piFrmLvl,
    char **ppczFrmType,
    char **ppczAttrNm,
    char **ppczAttrValStr
)
{
    return frGetNxtAttrBin (

```

-58/20-

```

        piFrmLvl,
        ppczFrmType,
        ppczAttrNm,
        ppczAttrValStr,
        NULL,
        NULL,
        NULL,
        NULL
    );
}

```

```

int frGetNxtAttrBin (
    int *piFrmLvl,
    char **ppczFrmType,
    char **ppczAttrNm,
    char **ppczAttrValStr,
    void **ppvAttrValBin,
    int *piAttrValType,
    int *piAttrValBinLen,
    int *piAttrStat
)
{
    psAttrEnt psAttrEntTmp;

    if (psAttrEntWalk == (psAttrEnt)(-1)) {
        psAttrEntWalk = NULL;
        if (piFrmLvl) *piFrmLvl = 0;
        if (ppczFrmType) *ppczFrmType = NULL;
        if (ppczAttrNm) *ppczAttrNm = NULL;
    }
}

```

-58/21-

```

if (ppczAttrValStr) *ppczAttrValStr = NULL;
if (ppvAttrValBin) *ppvAttrValBin = NULL;
if (piAttrValType) *piAttrValType = 0;
if (piAttrValBinLen) *piAttrValBinLen = 0;
if (piAttrStat) *piAttrStat = 0;
return -1;
}

```

```

if (!psAttrEntWalk) psAttrEntWalk = psAttrEntTail;
if (piFrmLvl) *piFrmLvl = psAttrEntWalk->psFrmEntOwn->iFrmLvl;
if (ppczFrmType) *ppczFrmType = frGetFrmType(psAttrEntWalk->psFrmEntOwn);
if (ppczAttrNm) *ppczAttrNm = frGetAttrNm(psAttrEntWalk);

```

```

if (ppczAttrValStr && psAttrEntWalk->pczAttrValStr)
    *ppczAttrValStr = strcpy(psAttrEntWalk->pczAttrValStrCpy,
psAttrEntWalk->pczAttrValStr);

```

```

if (ppvAttrValBin && psAttrEntWalk->pvAttrValBin) {
    *ppvAttrValBin = memcpy (
        psAttrEntWalk->pvAttrValBinCpy,
        psAttrEntWalk->pvAttrValBin,
        psAttrEntWalk->iAttrValBinLen
    );
}

```

```

if (piAttrValType) *piAttrValType = psAttrEntWalk->iAttrValType;
if (piAttrValBinLen) *piAttrValBinLen = psAttrEntWalk->iAttrValBinLen;
if (piAttrStat) *piAttrStat = psAttrEntWalk->iAttrStat;

```

-58/22-

```
psAttrEntTmp = psAttrEntWalk;
psAttrEntWalk = psAttrEntWalk->psAttrEntNxt;
if (!psAttrEntWalk) psAttrEntWalk = (psAttrEnt)(-1);
return 0;
}
```

```
int frGetNxtAttrReset(void)
{
    psAttrEntWalk = NULL;
    return 0;
}
```

```
int frPutFrm (
    char *pczFrmType,
    void *pvUsrHndl,
    int iOffStrt,
    int iOffNxt
)
{
    int iFrmTypeLen;

    if (iSkip) return 0;
    if (psFrmEntNew) free(psFrmEntNew);
    iFrmTypeLen = strlen(pczFrmType);

    if (psFrmEntFree) {
        if (strcmp((char *)psFrmEntFree + sizeof(sFrmEnt), pczFrmType)) {
            psFrmEntNew = realloc(psFrmEntFree, sizeof(sFrmEnt) +
2*(iFrmTypeLen+1));
```

-58/23-

```

        strcpy((char *)psFrmEntNew + sizeof(sFrmEnt), pczFrmType);
    }
    else {
        psFrmEntNew = psFrmEntFree;
    }

    psFrmEntFree = NULL;
}
else {
    psFrmEntNew = malloc(sizeof(sFrmEnt) + 2*(iFrmTypeLen+1));
    psFrmEntNew->psFrmEntNxt = NULL;
    psFrmEntNew->iFrmLvl = -1;
    psFrmEntNew->psAttrEntHead = NULL;
    psFrmEntNew->psAttrEntTail = NULL;
    psFrmEntNew->pvUsrHndl = pvUsrHndl;
    psFrmEntNew->iOffStrt = iOffStrt;
    psFrmEntNew->iOffNxt = iOffNxt;
    strcpy((char *)psFrmEntNew + sizeof(sFrmEnt), pczFrmType);
}

return 0;
}

int frUpdtFrm (
    void *pvUsrHndl,
    int iOffStrt,
    int iOffNxt
)
{

```

-58/24-

```

if (iSkip) return 0;
if (pvUsrHndl) psFrmEntTail->pvUsrHndl = pvUsrHndl;
if (iOffStrt != -1) psFrmEntNew->iOffStrt = iOffStrt;
if (iOffNxt != -1) psFrmEntNew->iOffNxt = iOffNxt;
return 0;
}

```

```

char *frGetFrm (
    void **ppvUsrHndl,
    int *piOffStrt,
    int *piOffNxt
)
{
    if (ppvUsrHndl) *ppvUsrHndl = psFrmEntTail->pvUsrHndl;
    if (piOffStrt) *piOffStrt = psFrmEntTail->iOffStrt;
    if (piOffNxt) *piOffNxt = psFrmEntTail->iOffNxt;
    return frGetFrmType(psFrmEntTail);
}

```

```

void frDoInitFrm(void)

```

```

{
    if (iSkip) {
        iSkip++;
        return;
    }

```

```

if (!psFrmEntNew) frPutFrm("default", NULL, 0, 0);
frChkFrm(frGetFrmType(psFrmEntTail), frGetFrmType(psFrmEntNew));
psFrmEntNew->psFrmEntNxt = psFrmEntTail;

```

-58/25-

```
psFrmEntNew->iFrmLvl = psFrmEntTail->iFrmLvl + 1;
```

```
psFrmEntTail = psFrmEntNew;
```

```
psFrmEntNew = NULL;
```

```
frInitFrm (
```

```
    frGetFrmType(psFrmEntTail),
```

```
    frGetFrmType(psFrmEntTail->psFrmEntNxt)
```

```
);
```

```
}
```

```
void frDoFinFrm(void)
```

```
{
```

```
    psAttrEnt psAttrEntNxt;
```

```
    psAttrEnt psAttrEntFree;
```

```
    if (iSkip > 1) {
```

```
        iSkip--;
```

```
        return;
```

```
    }
```

```
    else if (iSkip) {
```

```
        iSkip = 0;
```

```
    }
```

```
    else {
```

```
        frFinFrm(frGetFrmType(psFrmEntTail));
```

```
    }
```

```
    if (psFrmEntFree) free(psFrmEntFree);
```

```
    psFrmEntFree = psFrmEntTail;
```

```
    psFrmEntTail = psFrmEntTail->psFrmEntNxt;
```

-58/26-

```
psFrmEntFree->psAttrEntHead = NULL;
psFrmEntFree->psAttrEntTail = NULL;
psAttrEntNxt = psAttrEntTail;
psAttrEntTail = psFrmEntTail->psAttrEntTail;
psAttrEntWalk = NULL;
```

```
while(psAttrEntNxt != psAttrEntTail) {
    psAttrEntFree = psAttrEntNxt;
    psAttrEntNxt = psAttrEntNxt->psAttrEntNxt;
```

```
    if (psAttrEntFree->pvAttrValBin) {
        free(psAttrEntFree->pvAttrValBin);
        free(psAttrEntFree->pvAttrValBinCpy);
    }
```

```
    free(psAttrEntFree);
}
}
```

```
void frDoChkPt(void)
```

```
{
    char aczAttrNm[] = "checkpoint";
    char aczAttrValStr[32];
    void *pvAttrValBin;
    int iAttrValType = FR_BIN_TYPE;
    int iAttrValBinLen = sizeof(int);
    int iAttrStat = 0;
```

```
    sprintf(aczAttrValStr, "%d", iChkPtNum);
```

-58/27-

```
pvAttrValBin = &iChkPtNum;
```

```
frSetAttrBin (  
    aczAttrNm,  
    aczAttrValStr,  
    pvAttrValBin,  
    iAttrValType,  
    iAttrValBinLen,  
    iAttrStat  
);
```

```
    iChkPtNum++;  
}
```

```
static char *frGetFrmType(psFrmEnt psFrmEntTmp)  
{  
    char *pczFrmType = (char *)psFrmEntTmp + sizeof(sFrmEnt);  
  
    return strcpy(pczFrmType + strlen(pczFrmType) + 1, pczFrmType);  
}
```

```
static char *frGetAttrNm(psAttrEnt psAttrEntTmp)  
{  
    char *pczAttrNm = (char *)psAttrEntTmp + sizeof(sAttrEnt);  
  
    return strcpy(pczAttrNm + strlen(pczAttrNm) + 1, pczAttrNm);  
}
```

-59-

CLAIMS

The invention claimed is:

1. A computing system comprising:
 - a) an information flow represented as sequential frames to be processed
5 by the computing system, each frame being a set of attributes and each attribute having a type and a value;
 - b) a plurality of action routines for processing the values; and
 - c) a general purpose processing engine responsive to the frames, the
10 processing engine invoking select action routines in response to the frames.
2. The computing system of Claim 1 wherein the information flow is a protocol-based interprocessor communication.
3. The computing system of Claim 1 wherein the processing engine includes a common core engine and a plurality of support routines.
- 15 4. The computing system of Claim 3 wherein the support routines operate on at least one hierarchical memory store.
5. The computing system of Claim 4 wherein the at least one hierarchical memory store includes a stack of attributes.
- 20 6. The computing system of Claim 5 wherein the at least one support routine generates synthetic attributes for storage in the stack of attributes.

- 60 -

7. The computing system of Claim 1 further comprising at least one adapter routine in communication with the processing engine, each adapter routine parsing a select information flow into frames.
8. The computing system of Claim 1 wherein the information flow is processed in a multi-threading architecture.
9. The computing system of Claim 8 wherein serial frame sequences are processed in parallel.
10. The computing system of Claim 8 wherein frame sequences are pipelined between threads.
- 10 11. In a computing system, a method of processing an information flow, comprising the steps of:
 - representing an information flow as sequential frames to be processed by the computing system, each frame being a set of attributes and each attribute having a type and a value;
 - 15 in a plurality of action routines, processing the values; and
 - in a general purpose processing engine responsive to the frames, invoking select action routines in response to the frames.
12. The method of Claim 11 wherein the information flow is a protocol-based interprocessor communication.
- 20 13. The method of Claim 11 wherein the processing engine includes a common core engine and a plurality of support routines.
14. The method of Claim 13 wherein the support routines operate on at least one hierarchical memory store.

- 61 -

15. The method of Claim 14 wherein the at least one hierarchical memory store includes a stack of attributes.
16. The method of Claim 15 wherein the at least one support routine generates synthetic attributes for storage in the stack of attributes.
- 5 17. The method of Claim 11 further comprising the steps of:
providing at least one adapter routine in communication with the processing engine; and
in each adapter routine, parsing a select information flow into frames.
- 10 18. The method of Claim 11 wherein the information flow is processed in a multi-threading architecture.
19. The method of Claim 18 wherein serial frame sequences are processing in parallel.
20. The method of Claim 18 wherein frame sequences are pipelined between threads.

1/5

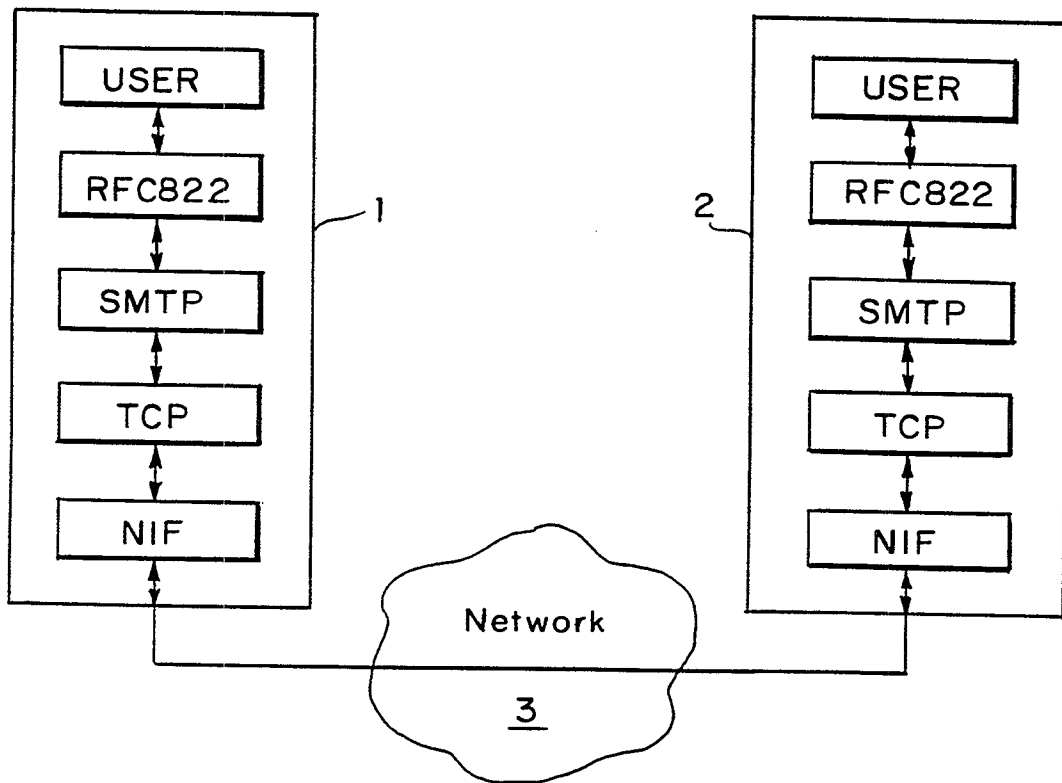


FIG. 1A (Prior Art)

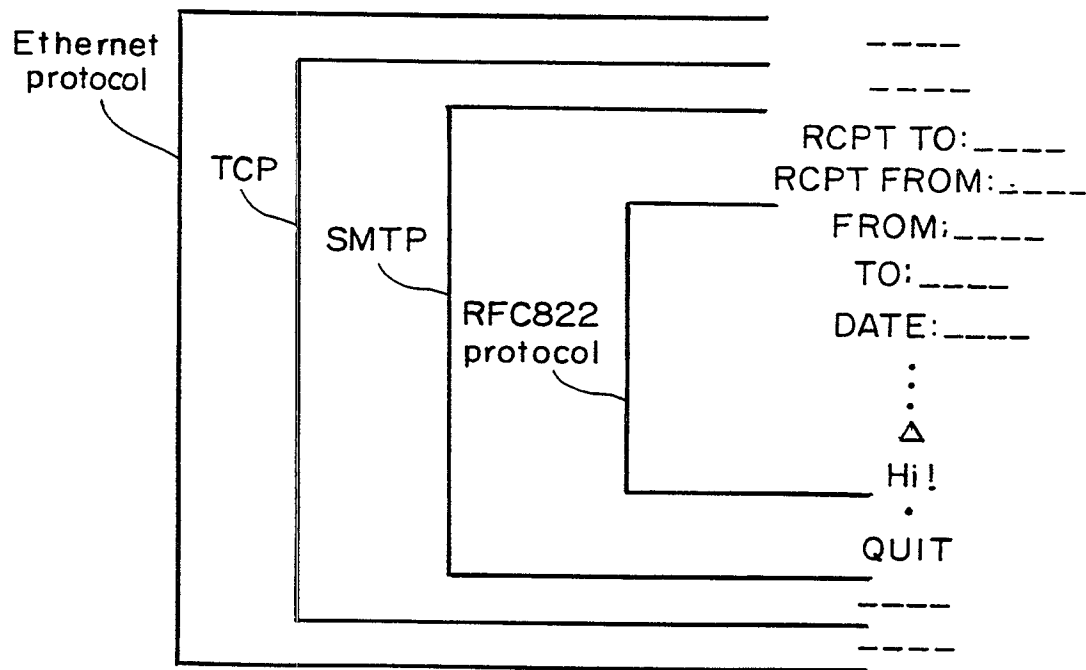


FIG. 1B (Prior Art)

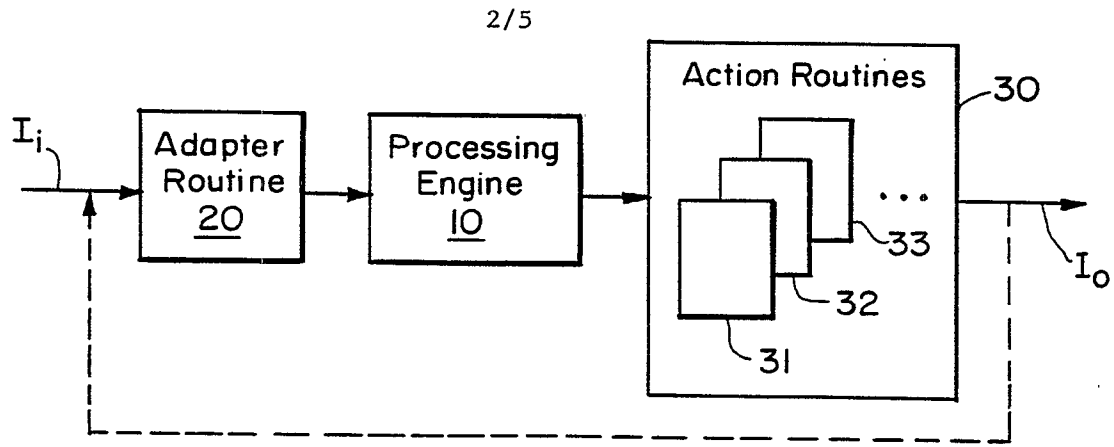


FIG. 2 (Prior Art)

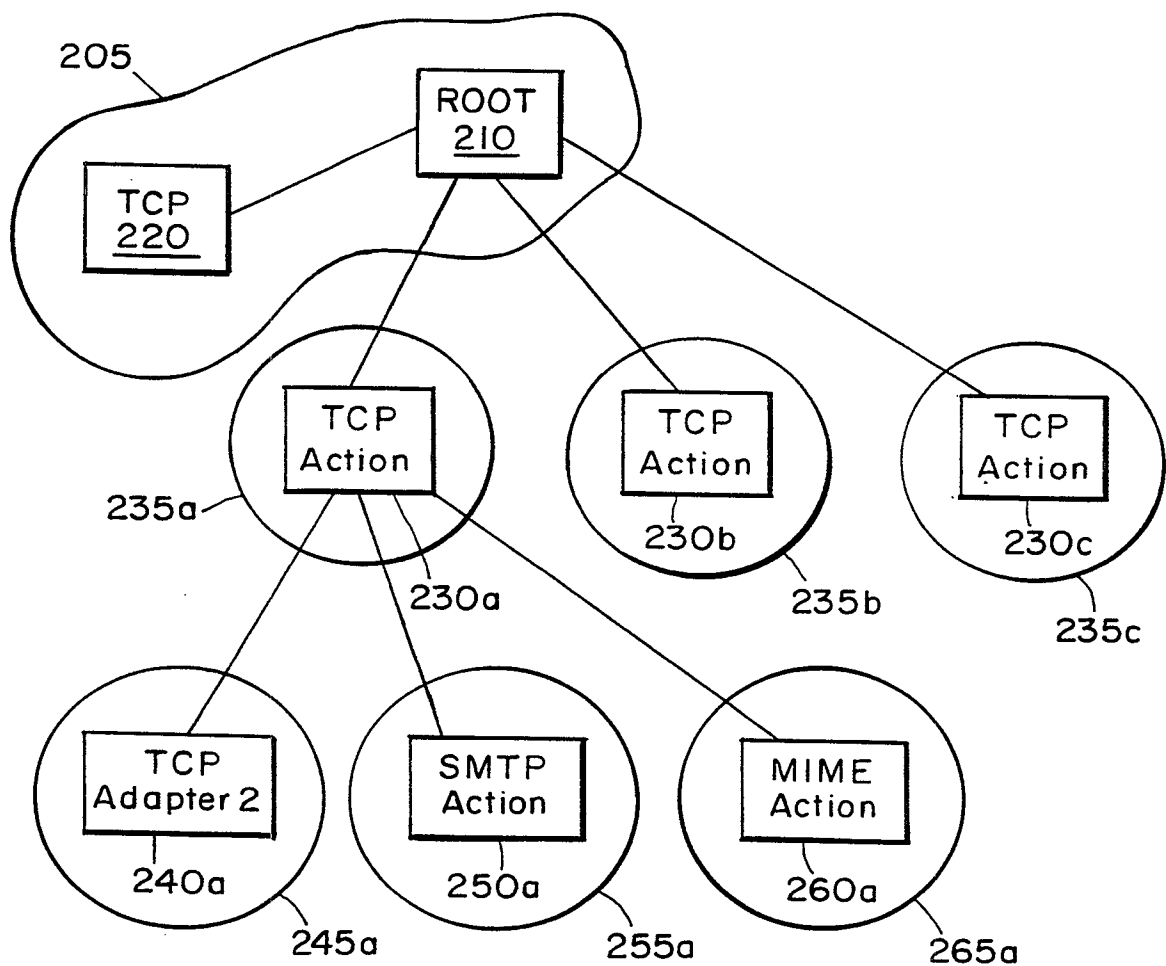
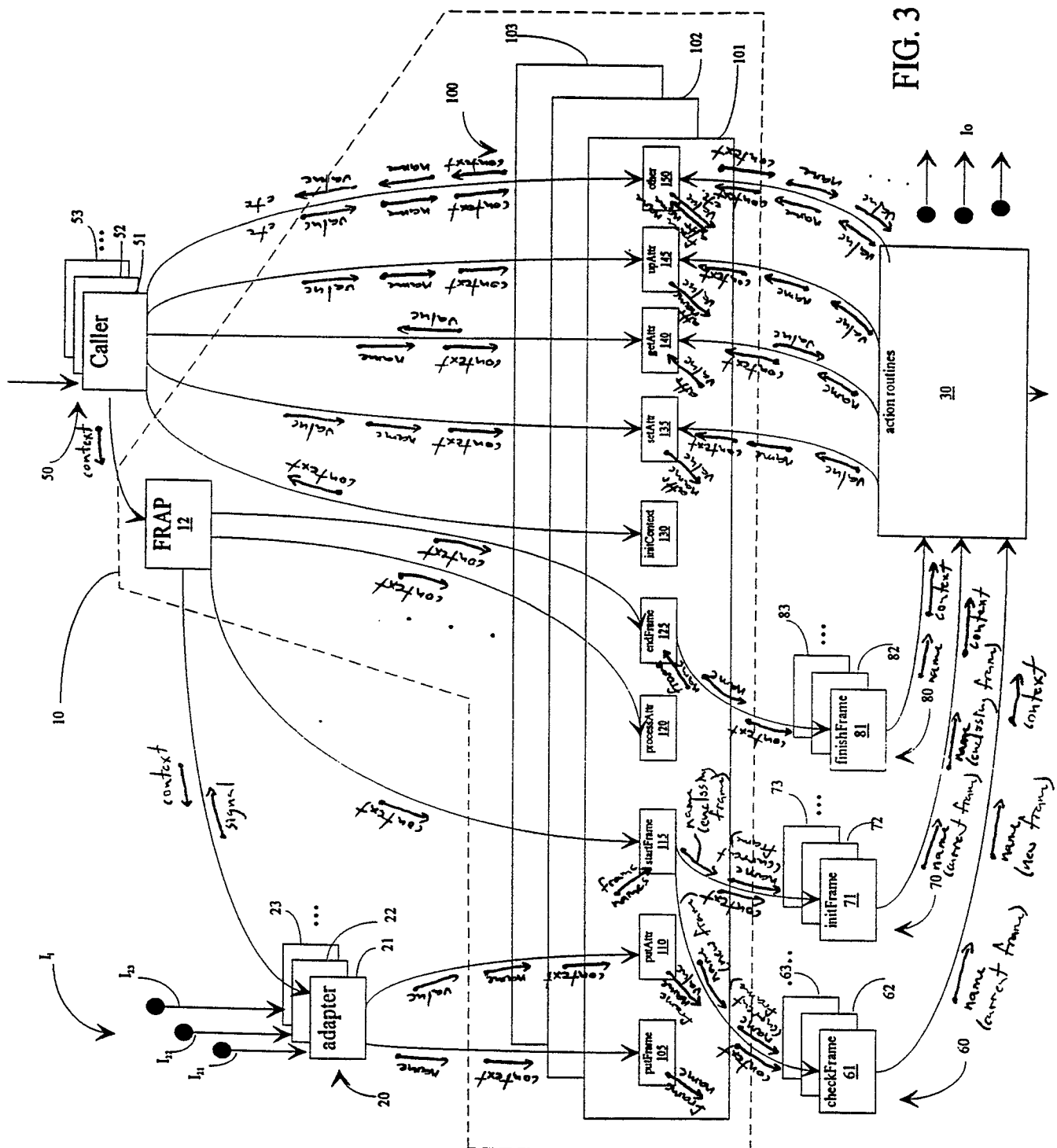


FIG. 10



4/5

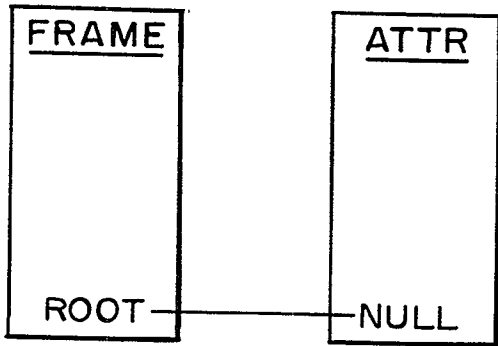


FIG. 4

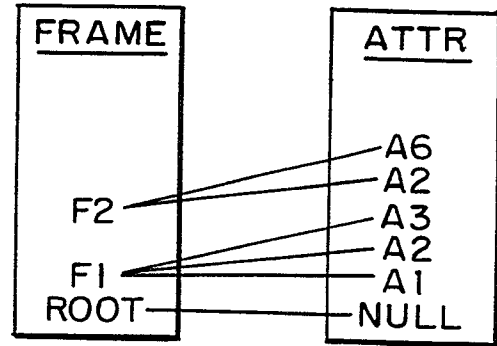


FIG. 7

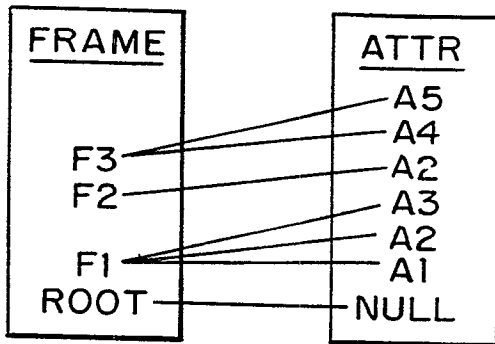


FIG. 5

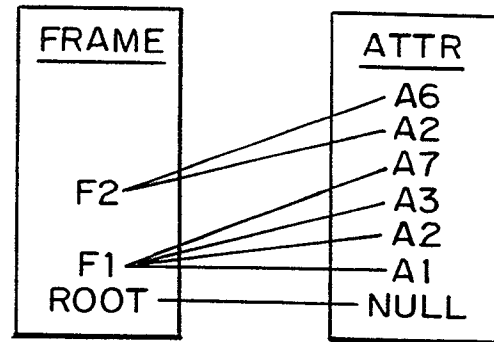


FIG. 8

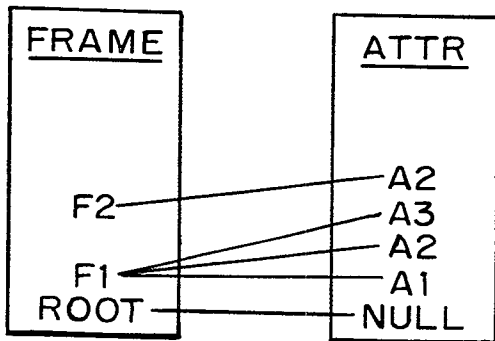


FIG. 6

5/5

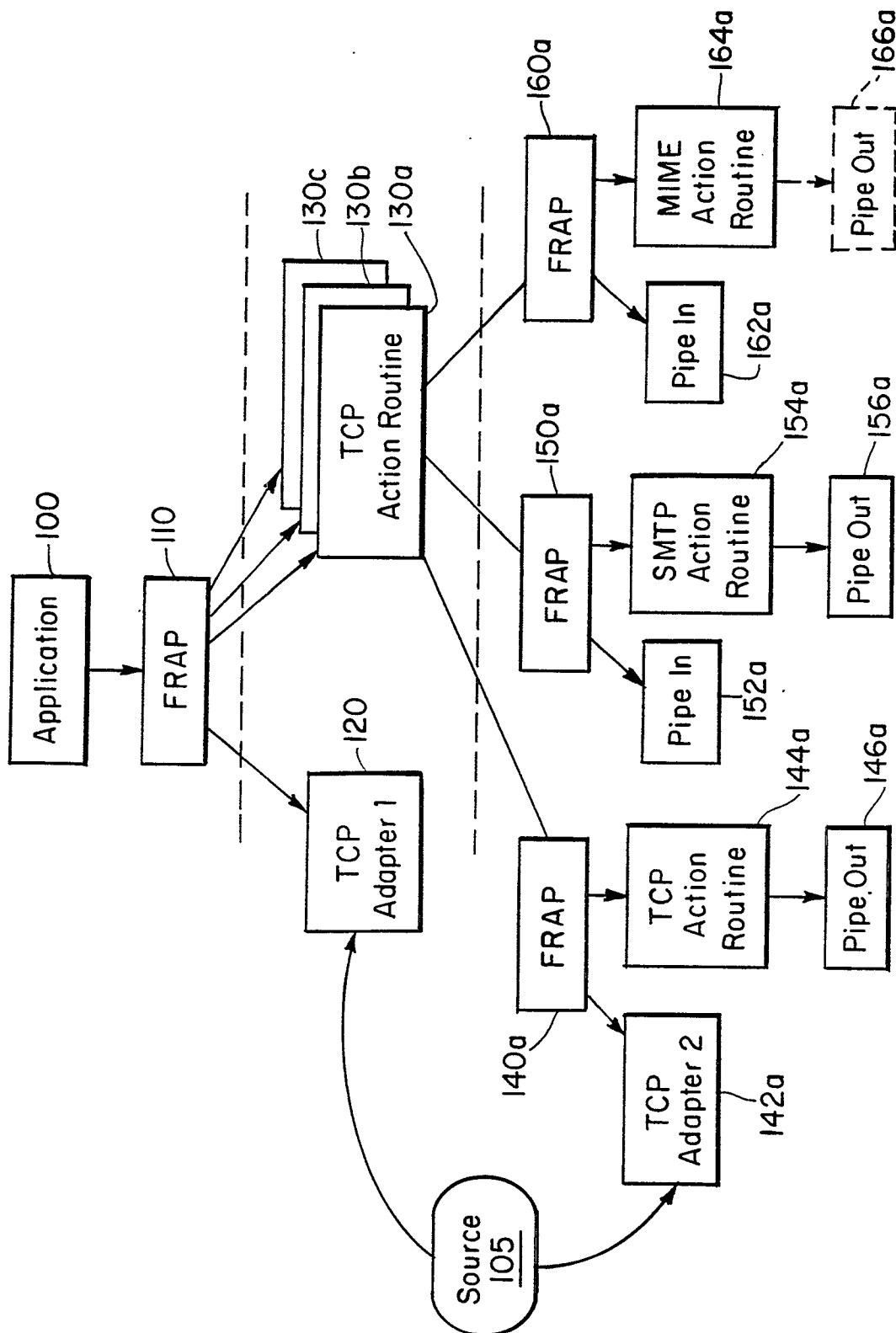


FIG. 9

