```
bits 1-7    current header type
Bytes 2-8 Network User Id (ASCII)
Byte 9    External Data Mode
          0000 0000 Reserved
Bytes 10-n
          Correlation Id (binary, max length=8 bytes).
```

Next are FM8 or Type 8 headers. Type 8 headers have been designed to provide secondary routing destinations. Their fields are as follows:

Header Length--length of header data including length field.

Header Type--Bit 0 is header concatenation flag.

Bits 1-7 indicate current header type.

Secondary--a symbolic name representing the ultimate

Destination--destination for the message.

The layout for Type 8 header is:

```
Byte 0          Header Length (hexadecimal)
Byte 1          Header Type
bit 0             no other headers present; or
                  concatenated header present
bits 1-7          current header type
Bytes 2-9       Symbolic Destination Name
```

For FM9 or Type 9 headers, the header has been designed to communicate to a VTAM application which provides various network management support functions. More specifically, the VTAM application has been developed in order to provide a general network management interface which both supports the network (by means of the DIA) and simplifies its maintenance. Additionally, VTAM application provides data transfer and remote command functions, the ability to write to, and read from, a centrally located and maintained database in order to archive statistic and other inter-network messages, and formatting of binary data into Hexadecimal Display.

In the case of Type 9 headers, the fields are:

Header Length--length of header data including length field.

Header Type--Bit 0 is header concatenation flag.

Bits 1-7 indicate current header type.

Function Code--indicates general message type.

Reason Code--indicates message content.

Flags--indicates application action to be performed.

Text Length--indicates length of subsequent text message. (Not including possible concatenated headers)

The layout for type 9 headers is:

```
Byte 0 Header Length (hexadecimal)
Byte 1 Header Type
bit 0    no other headers present; or
         concatenated header present
bits 1-7 current header type
Byte 2 Function Code; e.g.
         Command
         Statistics
         Alert
         Control
Byte 3 Reason Code
       Backbone Alerts Message
       Reception-originated Alerts Message
Byte 4 Flags
bits 0-3 Store by Key - 8 char. name follows;
         Retrieve by Key - 8 char. name follows;
         Data is Binary;
         Data is ASCII;
         Data is EBCDIC
bits 4-7 Reserved
Byte 5 Text Length
       if Flags = 1 . . . or .1 . . . then chars
       0-7 should be the storage key. It is recommended that
       record storage keys initially be the same as the Resource
       Name to which the data pertains.)
```

In the case of FM64 or Type 64 headers, the headers are used to transmit error and status messages between applications. Intermediate nodes need not examine the contents of the FM64 headers except in the case of the CICS switch which must obtain the displacement to the application text. If applications subordinate to an application subsystem are not available, the subsystem would strip the application text from the message, concatenate an FM64 message to any other headers which are present in the inbound message, and return the message to its original source.

Header Type 64 has been designed for the communication of status information between users, and prefaces architected message text. The fields for Type 9 headers are:

Header Length--length of header data including length field.

Header Type--Bit 0 is header concatenation flag.

Bits 1-7 indicate current header type.

Status Type--indicates type of status communicated such as status request or error.

Data Mode--indicates whether message text is ASCII or EBCDIC

Text Length--Length of subsequent message text (Not including possible concatenated headers).

The header Type 64 layout is:

```
Byte 0 Header Length (hexadecimal)
Byte 1 Header Type
bit 0    no other headers present; or concatenated header present
bits 1-7 current header type
Byte 2 Status Type
       Information
       Status Request
       Error
       Terminate
Byte 3 Data Mode; e.g.,
       EBCDIC
       ASCII
       Binary
Bytes 4-5
       Text Length
```

In accordance with the invention, it has been determined that in some cases it is desirable to pre-define certain application level message formats so that they may be consistently used and interpreted. The following discussion is devoted to architected message text formats which are processed at the application level. For FM9 message text, in order to accommodate "Reliability Serviceability Availability" (RSA) functions within network 10, a fixed format for "alets" is defined in the preferred embodiment. Particularly if it is defined as message text following an FM9 header The FM9 Function Code Alerts Message would be as follows:

```
Byte 0          Reserved value
Byte 1          System Origin
Byte 2          Internal/External flag
Byte 3-5        Message Originator
Byte 6-9        Message Number
Byte 10         Severity indicator; e.g.
                Error
                Information
                Severe Error
                Recovery Successful
                Warning
Byte 11         Reserved value
Byte 12-14      Error Threshold.
```

For FM64 message text, the application message text is always prefaced by the appropriate header which indicates whether message text is ASCII or EBCDIC.

The FM64 message text fields are as follows:

Action Field--indicates type of operator or application action to be performed

Module Name--Sending application Id Format of this field is SSSTnnnn where SSS=sender initials T=type 0=Network standard for gateways 1=non-standard, gateway specific nnnn=Sender Site number

Reference Number--Number assigned by sender for reference This number is used to indicate specific error codes if the message is an error message (FM64 stat type 8). This number is used to indicate specific commands if the message is a status request (FM64 stat type 4).

Text--Alphanumeric (Printable) text.

The FM64 Message Text layout is:

Byte 0	Action Field (alphanumeric), e.g.,
	Action
	Decision
	Information
	Wait
Bytes 1-8	Module Name (alphanumeric)
Bytes 9-12	Reference Number (display numeric)
	Default
	request user status
	user active
	user inactive
	user inactive - retry after interval
	store in user mailbox
	cache to server link failure
	request appl status
	server to host failure
	appl active
	appl inactive
	appl inactive - retry after interval
	message was undeliverable
	response was timed out
	syncpoint
	checkpoint
	delay
	appl. error codes
Bytes 13-n	Text (alphanumeric).

Turning next to co called "Backbone States", as will be described below, application sessions may be used as pipes for user transaction traffic. In this regard, it is desirable to establish a set of protocols to be used between originating users and destination users. Further it is important for intermediate nodes to be aware of the status of connectivity with adjacent nodes and specifies some actions to take when messages are known to be undeliverable.

In this context, it is to be noted that the "system up" message is used to signal the start of application traffic between the switch applications. The originating application transmits an FM0 with a system up function code and response expected. The receiving application swaps the SID/DID, sets the Response bit on, and returns the message. If the receiving application is not available no response will be returned and the message will time out.

In the case of "system down" messages, the message is used to prepare the termination of the session between switch applications. The originating application transmits an FM0 with a session down function code and response expected. The originating application sends an FM64 with "status type=terminate", and data mode=EBCDIC. FM64 text follows the header with "action field"=A (Action), "module name"SSSx0nnnn, "reference number"=0, Text=((timestamp=HHMMSS), Number of current users=NNNNN). The intended result is that the originating application will not accept any messages inbound to the utility session. The responding application will then have the opportunity to return outstanding responses across the utility session. The responding application then returns an FM0 with System Down back to the originating application.

For each "echo" messages, the echo message may be used to determine whether a major application is still available. Specifically, the originating application sends an application message to its gatewayed partner using a FM0 with an echo function. The destination application swaps the SID/DID, set the response bit on and returns the message otherwise untouched, thus effecting echo

For "APPL status request messages, the message is used to determine the status of a major application between nodes.

Continuing, for "unsolicited application status posting" messages these messages are used for transmission of application status messages by unsolicited application (No response expected) across a nodes. For the message, the originating application wishes to post an application status to its partner in another node. This message may be on the behalf of the originating application itself or on behalf of another application.

Turning next to user to internal APPL messages, and with regard to "session beginning", it is to be noted these messages normally arise at the start of conversation between a user and an internal application. For them the network user sends an FMO with a "begin session" function code and "response expected". The responding application swaps the SID/DID, supplies a "correlation Id", and returns both the FMO with the response bit set.

In the case of rejection of a conversation initiation requests, the originating application transmits an FMO with a "begin session" function code and "response expected". The responding application swaps the SID/DID, and returns the FMO with the response bit set as well as a function code of "abend" session.

For "applications" messages, these messages normally arise at the middle of conversation between a network user and an internal application. In this care, the originating user transmits an FMO with an "application" message function code, and "response expected". The responding application swaps the SID/DID, sets the response bit on and returns the response. "End session" messages typically arise in connection with unconditional termination of user/internal application sessions. The originating transmits an FMO with an "end sessions" function code. Here however, no response is expected from the corresponding application.

For an "end session abnormal" message, the message unconditionally terminates an application conversation "abend".

Continuing, "request terminate" messages cause conditional termination of session with an internal application.

For messages concerning "rejection of a request due to link failure", in the case of server 205 to host link, the originating application transmits and FMO with "response expected". The message is intercepted by server 205 which recognizes it as undeliverable. A server 205 application returns the message with an FM64 message after stripping the application text.

For messages concerning rejection of request due to link failure, in the case of communication between the cache/concentrator 302 and server 205, the originating application transmits an FMO with Response Expected. The message is intercepted by the cache/concentrator 302 which recognizes it as undeliverable. A cache/concentrator application returns the message with an FM64 message after stripping the application text.

For messages concerning "conditional terminate rejected", the message is issued where a conditional termination of application conversation is not accepted by partner application.

For "user continuity posting" messages, the message is used where the originating application wishes to post the status of a user to its partner application across the gateway 210.

Continuing, for "user continuity requests", the message is used where an external application requests logon status of a particular network user.

In the case of "application error" messages, the messages is used where transmission of application error message by responding application is required.

Still further, for "timeout scenarios", and specifically, "timeout scenario with timeout response required", the originating user sends an application message to an internal application with "data mode"="response expected" and "timeout response" required. The originating switch sets a timer for each "response expected" outbound message. If a response is not received before the switch timeout value is reached the switch 205 sends a message with an FM64 header having a "timeout reference" code to the originating application.

For "response occurs after timeout" messages, the originating user sends an application message to an internal application with "response expected". The originating switch sets a timer for each "response expected" outbound message. If a response is received after the timeout value is exceeded, server 205 switch routes the message to a server 205 application which may log the message as non-deliverable, ship the message to the user, or drop it depending on the FMO class of service option specified on the original request message.

In the case of "maximum resources scenario" messages, the originating user transmits a message to a destined internal application. The destination switch determines that no resources are currently available to support the transmission, and returns the message to the originator, after inserting an FM64 with a "status=error and FM64 text with an "action=wait. The originating user may then retry or take other action.

Finally, the following graphic example illustrates normal message flow. ##STR1##

Turning next to messages passed over gateways 210, the normal exchange of messages between the network and external parties occurs between two applications; i.e., the server 205 network message

handler (NMH). The server Switch 205 is an application which is written and maintained by network 10 and resides on it. The message handler resides on the other side of gateway 210 from network 10 and may be written and maintained by the external party; i.e., suppliers of information to network 10 such as Dow Jones.

The session between the two applications is used as a pipe for the communications between many network users and a variety of applications external to the network. In this design, the switch server 205 has three primary responsibilities. It must pass network originated messages across the gateway to the network message handler. It must distribute messages returning across gateway 210 to the appropriate network applications or users, i.e. RS 400. Additionally, it must manage the continuity of a network user session with the external service provider. Typically, users enter into a conversation with a set of transactions. This set of transactions is referred to as a task. These tasks are called user sessions. The boundaries of these tasks are indicated by begin session/end session flags.

The network message handler also has several responsibilities. It must pass externally originated messages across gateway 210 to the switch server 205 at network 10. It must distributed messages returning across gateway 210 to the appropriate external applications. And, it must be able to communicate the availability of external applications to network switch server 205.

With regard to gateway messages, in the case of "application to application" messages, and for "system up" messages, the system up message is used to signal the start of application traffic between switch 205 and the network message handler. The originating application transmits an FMO with function code "system up", and "response expected". The receiving application swaps the SID/DID, sets the response bit on, and returns the message. If the receiving application is not available no response will be returned and the message will time out.

Continuing for gateway "system down" messages, the system down message is used to prepare the termination of the session between the switch 205 and the NMH. The originating application transmits an FMO with function code "session down" and "response expected. The originating application sends an FM64 with "status type"="terminate","data mode"="EBCDIC". FM64 Text follows the header with "action field"="A" (Action), "module name"="SSSx0nnnn", "reference number"="0", "text"= ((timestamp=HHMMSS), number of current users=NNNNN). The intended result is that the originating application will not accept any messages inbound to the utility session. The responding application will then have the opportunity to return outstanding responses across the utility session. The responding application then returns an FMO with system down back to the originating application.

Further, for "prepare to bring system down" messages, the message is used to prepare the termination of the session between the Switch 205 and the NMH. The originating application transits an FM0 with function code "prepare system down". The responding application transmits an FM0 with function code "session down" and "response expected". The responding application sends an FM64 with "status type"="terminate", "data mode"="EBCDIC". FM64 Text follows the header with "action field"="A" (action), "module Name"="SSSx0nnnn", "reference number"="0", "text"= ((Timestamp=HHMMSS), number of current users=NNNNN). The intended result is that the responding application will not accept any messages inbound to the utility session. The originating application will then have the opportunity to return outstanding responses across the utility session. The originating application then returns an FM0 with "system down" back to the responding application.

For "echo" messages, the message may be used to determine whether a major application is still available. The originating application sends an application message to its gatewayed partner using a FM0 with function echo. The destination application swaps the SID/DID, set the response bit on and returns the message otherwise untouched.

In the case of "APPL status request", the request is used to determine the status of a major application across the gateway.

Continuing, for "unsolicited application status posting messages, the message is used for transmission of application status messages by unsolicited applications no response expected across a gateway. In this case the originating application wishes to post an application status to its partner across the gateway. This message may be on the behalf of the originating application itself or on behalf of another application.

For network to use "external APPL" messages, within the case of "begin session" messages, the message is used for normal start of conversation between a and an external application. The user, i.e. RS 400 sends an FM0 with function "begin session" and "response expected", as well as an FM4 with null value in the "correlation id". The responding application swaps the SID/DID, supplies a Correlation ID, and returns both the FM0 with the response bit set and the FM4.

For rejection of a conversation initiation request, the originating application resident application, transmits an FM0 with function Begin Session and Response Expected as well as an FM4 with NULL value in the Correlation ID. The responding application swaps the SID/DID, and returns the FM0 with the response bit set as well as a function code of ABEND session. The responding application also returns the FM4.

Further, for "applications" message, the message is used for normal middle of conversation between a network user and an external application. The originating user transmits an FM0 with function code "application" message, and "response expected". It also supplies the TTXUID and the correlation id received on the begin session response back to the corresponding application across the gateway. The responding application swaps the SID/DID, sets the response bit on and returns the FM0 and FM4.

For "end session" message, the message is used for unconditional termination of user/external application sessions. The originating user transmits an FM0 with function code "end session", no "response expected". Additionally it sends an FM4 containing the TTXUID and the echoed "correlation id" in an FM4. No response is expected from the corresponding application.

For "end session abnormal" messages, the message is used for unconditional termination ABEND of gatewayed application conversation. In the case of "request terminate", the message is used for conditional termination of user session with an external application. For "conditional terminate rejected" messages, the message is used for a conditional termination of application conversation not accepted by partner application across a gateway.

For "user continuity posting" messages, the message is used where the originating application wishes to post the status of a user to its partner application across the gateway.

In the case of "user continuity" request, external application requests logon status of a particular user, i.e. RS 400. For "application error" messages, the message is used for transmission of application an error message by responding application across a gateway.

In the case of "delayed response" messages, the originating application sends an application message to its gatewayed partner using the minimally a FM0 and a FM4 FM64 may be present. The destination switch signals an application on the originating side that the response may be slow by sending a FM0 with function code "status/return", the response bit is not set. The FM4 is returned, and an FM64 "status", FM64 text "Action"="Information" is also sent. Slow response may be due to a number of

factors such as function shipping requirements or many I/Os. In parallel, the gateway partner application processes the message according to normal flow.

For "timeout scenario", the originating user sends an application message to an external application with "response expected". The switch server sets a timer for each "response expected" outbound message. If a response is received after the timeout value is exceeded, the TPF switch routes the message to a TPF application which may log the message as non-deliverable, ship the message to the user, or drop it depending on the FM0 class of service option specified on the original request message.

For the "maximum resources scenario" messages, the originating user transmits a message to a destined external application. The network message handler determines that no resources are currently available to support this transmission. The network message handler returns the message to the originator, after inserting an FM64 with a "Status"="Error" and FM64 text with an "action=wait". The originating user may then retry or take other action.

Finally, an example illustrates normal message flow. ##STR2##

And, the following is an example that illustrates premature loss of user connectivity due to the loss of connection between the network switch server 205 and a cache/concentrator 302. In this case, an application peripheral to switch 205 posts the user status inactive to the NMH using an 30 FM64 Ref=0008 user inactive. External application reaction to this posting is implementation dependent. In this example, the external application returns outstanding responses using the FM64 "ref"="mailbox option". ##STR3##

OBJECT LANGUAGE

In accordance with the invention, in order to enable the manipulation of the network objects the application programs necessary to support the interactive text/graphic sessions are written in a high-level language referred to as "TBOL", (TRINTEX Basic Object Language, "TRINTEX" being the former company name of one of the assignees of this invention). TBOL is specifically adapted for writing the application programs so that the programs may be compiled into a compact data stream that can be interpreted by the application software operating in the user personal computer, the application software being designed to establish the network Reception System 400 previously noted and described in more detail hereafter.

In accordance with the invention, the Reception System application software supports an interactive text/graphics sessions by managing objects. As explained above, objects specify the format and provide the content; i.e., the text and graphics, displayed on the user's screen so as to make up the pages that constitute the application. As also explained, pages are divided into separate areas called "partitions" by certain objects, while certain other objects describe windows which can be opened on the pages. Further, still other objects contain TBOL application programs which facilitate the data processing necessary to present the pages and their associated text and graphics.

As noted, the object architecture allows logical events to be specified in the object definitions. An example of a logical event is the completion d data entry on a screen; i.e., an application page. Logical events are mapped to physical events such as the user pressing the <ENTER> key on the keyboard. Other logical events might be the initial display of a screen page or the completion of data entry in a field. Logical events specified in page and window object definitions can be associated with the call of TBOL program objects.

RS 400 is aware of the occurrence of all physical events during the interactive text/graphic sessions.

When a physical event such as depression of the forward <TAB> key corresponds to a logical event such as completion of data entry in a field, the appropriate TBOL program is executed if specified in the object definition. Accordingly, the TBOL programs can bethought of as routines which are given control to perform initialization and post-processing application logic associated with the fields, partitions and screens at the text/graphic sessions.

RS 400 run time environment uses the TBOL programs and their high-level key-word commands called verbs to provide all the system services needed to support a text/graphic session particularly, display management, user input, local and remote data access.

In accordance with the invention, the TBOL programs have a structure that includes three sections: a header section in which the program name is specified; a data section in which the data structure the program will use are defined; and a code section in which the program logic is provided composed of one or more procedures. More specifically, the code section procedures are composed of procedure statements, each of which begins with a TBOL key-word called a verb.

In accordance with the invention, the name of a procedure can also be used as the verb in a procedure statement exactly as if it were a TBOL key-word verb. This feature enables a programmer to extend the language vocabulary to include customized application-oriented verb commands.

Continuing, TBOL programs have a program syntax that includes a series of "identifiers" which are the names and labels assigned to programs, procedures, and data structures.

An identifier may be up to 31 characters long; contain only uppercase or lowercase letters A through Z, digits 0 through 9, and/or the special character underscore (.sub.13); and must begin with a letter. Included among the system identifiers are "header section identifiers" used in the header section for the program name; "data section identifiers" used in the data section for data structure names, field names and array names; and finally, "code section identifiers" used in the code section for identification of procedure names and statement labels.

The TBOL statement syntax adheres to the following conventions. Words in uppercase letters are key-words and must be entered exactly as shown in an actual statement. When operand are allowed, descriptive operand names and lowercase letters follow the key word. In this arrangement, operand names or laterals are entered in an actual statement. Operand names enclosed in square brackets (› !) are optional and are not required in an actual statement. Operand names separated by a bar (.vertline.) mean that one, and only one, of the separated operand can be included in an actual statement. Operand names followed by an ellipsis (. . .) can be entered 1 or more times in an actual statement. Model statement words not separated by punctuation must be separated by at least one blank (or space character) in actual statements. Model statement punctuation such as comma (,), semicolon (;), less than sign (<), equal sign (=), greater--than (>), and parentheses (()) must be included where shown in actual statements. Square brackets (), bars (.vertline.), and ellipses (. . .) should not be included in actual statements.

An example of a model statement would be as follows:

GOTO.sub.-- DEPENDING.sub.-- ON index, label (.label . . .).

This model says that a valid GOTO.sub.-- DEPENDING.sub.-- ON statement must begin with the word "GOTO.sub.-- DEPENDING.sub.-- ON" followed by at least one blank. Thereafter, an "index" and a "label" separated by a comma must be included. The index and at least one label are required. Additional labels may also be used, provided each is preceded by a comma. Further, the statement must have a semicolon as the last character.

Comments can be included in a TBOL program on a statement line after the terminating semicolon character or on a separate comment line. Comment text is enclosed in braces ({ }). For example: {comments are enclosed in braces}. Comments can be placed anywhere in the source code stream since, in accordance with the invention they are ignored by the TBOL compiler. Additionally, blanks (or space characters) are ignored in TBOL statement lines except where they function as field separators.

As noted, TBOL programs have a structure that includes a header section, data section and code section. More particularly, every TBOL program must have a header section. The header section contains a PROGRAM statement. The PROGRAM statement contains the key word PROGRAM followed by the name of the program. For example:

PROGRAM program.sub.-- name;

where "program.sub.-- name" is an identifier; i.e., the name of the program.

Accordingly, the header section for a TBOL program called LOGON would look like as follows:

PROGRAM LOGON: {User logon program}

The data section in a TBOL program begins with the key word DATA which is followed by data structure statements. The structure statements contain the data structure definitions used by the program. If the data structure does not have to be defined for the program it can be omitted. However, if a TBOL program does not include a data section, it must use a more restricted structure, more fully explained hereafter. As an example, the data syntax would be:

DATA structure ›structure . . . !;

where "structure" is a data structure statement. The data structure statement contains a definition, which consists of the data structure name followed by an equal sign and then the names of one or more variables. For example:

structure.sub.-- name=variable.sub.-- name ›,variable.sub.-- name . . . !;

where "structure.sub.-- name" is an identifier; i.e., the name of the data structure; and "variable.sub.-- name" is an identifier for the variable; i.e., the name of a variable.

All of the variables in the data structures are defined as string (or character) variables. TBCL string variables are of two kinds, fields and arrays. In the case of filed definitions, a variable field is defined with and identifier; i.e., the name of the field. No data type of length specification is required. An individual field is referenced by using the field name. Further, subsequent fields can be referenced by using a field name followed by a numeric subscript enclosed in parentheses (()). The subscript however, must be an integer number.

A field name followed by a subscript refers to a following field in the data section of a TBOL program. The subscript base is 1. For example, if a field CUST.sub.-- NBR were defined, then CUST.sub.-- NBR refers to the field CUST.sub.-- NBR, CUST.sub.-- NBR(1) also refers to the field CUST.sub.-- NBR and CUST.sub.-- NBR(2) refers to the field following CUST.sub.-- NBR.

In the case of array definitions, the TBOL array is a one-dimensional table (or list) of variable fields, which can be referenced with a single name. Each field in the array is called an element.

An array can be defined with an identifier, particularly, the name of the array, followed by the array's dimension enclosed in parentheses (()). The dimension specifies the number of elements in the array. By way of illustration, if an array is defined with a dimension of 12, it will have 12 elements. An individual element in an array is referenced by using the array name followed by a numeric subscript enclosed in parentheses (()). The subscript indicates the position of the element in the array. The first element in an array is referenced with a subscript of 1. The subscript can be specified as either an integer number or an integer register as described, hereafter.

With regards to variable data, data contained in variables is always left-adjusted. Arithmetic operations can be formed on character strings in variables if they are numbers. A number is a character string that may contain only numeric characters 0 through 9, an optional decimal point, an optional minus sign in the left-most position, commas and the dollar sign ($).

When you perform an arithmetic operation on a character string, leading and trailing zeros are trimmed and fractions are truncated after 13 decimal places. Integer results do not contain a decimal point. Negative results contain a minus sign (-) in the left-most position.

Each field and each array element has a length attribute which is initialized to zero by the Reception System at program start-up. The LENGTH verb, to be described more fully hereafter, can be used to set the current length of a field or array element during program execution. The maximum length of a field or an array element is 65,535.

Further, the maximum number of variables that can be defined in the data section of a TBOL program is 222. This number includes fields and array elements.

The following example data section contains five data structure statements, each defining a data structure. Each structure statement begins with the name of the data structure followed by an equal sign.

Next, are the names of the variables which make up the structure. The variable names are separated by commas. The last variable name in each structure statement is followed by a semicolon which terminates the statement.

The third data structure given, i.e. SALES.sub.-- TABLE, contains two arrays. The others contain fields. The last structure statement, i.e. WK.sub.-- AREA is and example of a single line.

DATA {Key word DATA begins data section}

BILL.sub.-- ADDR={data structure BILL.sub.-- ADDR}

BILL.sub.-- NAME, {field1 BILL.sub.-- NAME}

BILL.sub.-- ADDR1, {field2 BILL.sub.-- ADDR1}

BILL.sub.-- ADDR2, {field3 BILL.sub.-- ADDR2}

BILL.sub.-- ADDR3, {field 4 BILL.sub.-- ADDR3}

SHIP.sub.-- ADDR,={data structure SHIP.sub.-- ADDR})

SHIP.sub.-- NAME, {field1 SHIP.sub.-- NAME}

SHIP.sub.-- ADDR1, {field2 SHIP.sub.-- ADDR1}

SHIP.sub.-- ADDR2, {field3 SHIP.sub.-- ADDR1}

SHIP.sub.-- ADDR3, {field4 SHIP.sub.-- ADDR1}

SALES TABLE={data structure SALES.sub.-- TABLE}

MONTH QUOTA(12), {array1 MONTH.sub.-- QUOTA}

MONTH SALES(12), {array2 MONTH.sub.-- SALES}

MISC.sub.-- DATA={data structure MISC.sub.-- DATA}

SALESPERS.sub.-- NAME, {field1 SALESPERS.sub.-- NAME}

CUST.sub.-- TELNBR; {field2 CUST.sub.-- TELNBR}

WK.sub.-- AREA={data structure WK.sub.-- AREA}

TEMP1,

TEMP1;

Continuing, TBOL contains a number of predefined data structures which can be used in a TBOL program even though they are not defined in the program's data section. There are two kinds of TBOL-defined data structures, these are "system registers" and "external data structures".

In the case of systems registers, tree different types exist. The first type are termed "integer registers", and are used primarily for integer arithmetic. However, these registers are also useful for field or array subscripts. The second type are termed "decimal registers", and are used for decimal arithmetic. The third type are called, "parameter registers" and are used to pass the data contained in procedure statement operand when the name of a procedure is used as the verb in the statement rather than a TBOL keyword.

The variables defined in the data section of a program are string (or character) variables, and the data in them is kept in string format. In most cases there is no need to convert this data to another format, since TBOL allows substantially any kind of operation(including arithmetic) on the data in string form. As will be appreciated by those skilled in the act, this eliminates the clerical chore of keeping track of data types and data conversion.

There are some cases where it is desirable to maintain numeric data in binary integer or internal decimal format. For example, an application involving a great deal of computation will execute more efficiently if the arithmetic is done in binary integer or internal decimal format data rather than string data. In these cases, data conversion can be performed by simple moving the numeric data to the appropriate register. When data is moved from a register to a variable, it is converted to string format.

Integer registers are special-purpose fields for storing and operating on integer numeric data in binary format. The integer registers are named I1 through I8. Numeric data moved to an integer register is converted to an integer number in binary format. Further, an attempt to move non-numeric data to an

integer register will cause an error. The largest negative number an integer register can hold is -32,7767, while the largest positive number than can be held is 32,767. An noted arithmetic operations in integer registers will execute more efficiently than arithmetic operations in string variables.

Decimal registers are special-purpose fields for storing and operating on numeric data in internal decimal format. The decimal registers are named D1 through D8. Numeric data moved to a decimal register is converted to a decimal number in internal decimal format. An attempt to move non-numeric data to a decimal register will cause an error. The largest negative number a decimal register can hold is -9999999999999.9999999999999, while the largest positive number a decimal register can hold is 9999999999999.9999999999999. Additionally, decimal registers can not be used as field or array subscripts. And, again, arithmetic operations in decimal registers will perform better than arithmetic operations in string variables.

As pointed out above, the code section of a TBOL program contains the program logic, which itself is composed of one or more procedures. In the logic, the procedures are expressed as procedure statements. Each procedure statement begins with a TBOL keyword called a verb which is followed by operand, or parameters containing the data on which the verb is to operate. The name of a procedure can be used as the verb in a procedure statement exactly as if it were a TBOL keyword verb. As noted this enables the creator of a TBOL program; i.e. the party creating the text/graphic session, to extend the language vocabulary to include his own application-oriented verb commands.

When a procedure is used as the verb in a procedure statement, TBOL saves the current parameter register values, and the parameter data in the verb operand is moved into the parameter registers where it is available to the "called" procedure. When the "called" procedure returns, TBOL restores the saved parameter register values.

Parameter registers are special-purpose fields for passing parameter data to "called" procedures. The parameter registers are named P0 through P8. When a procedure is "called" by using its name as the verb in a procedure statement, the current contents of P0 through P8 are saved. Further, data from the first operand in the procedure statement is placed in P1; data from the second operand is placed in P2; and so on, up to eight operand. If no operand, or less than eight operand are specified, the parameter registers corresponding to the missing operand are set to null. In accordance with this arrangement, the number of operand is placed in P0, and the "called" procedure is given control.

When control returns to the "calling" procedure from the "called" procedure, the previous contents of P0 through P8 are restored. Following execution of the "called" procedure, execution of the "calling" procedure continues.

The "calling" procedure can pass along its own parameters to the "called" procedure by naming parameter registers as operand. The TBOL internal stack can be used to pass additional data to the "called" procedure, or to pass data back to the "calling" procedure.

There are two kind of TBOL-defined external data structures; they are partition structures and global structures. With regard to partition external data structures, as noted above the screens displayed during a test/graphic session are called pages. As also noted, pages may be divided into separate areas called "partitions". Each page partition has its own predefined partition external data structure. Each partition external data structure can contain up to 256 variables for data pertaining to that partition. A TBOL program associated with a particular partition has access to the partition's external data structure and the variables it contains. However, the program cannot access another partition's external data structure.

The variable in a partition external data structure are character string variables like those defined in the

data section of a program. The variables within each partition external data structure are named &1 through &256. The DEFINE compiler directive enables the program to use meaningful names for these variables in the program source code.

Partition external variables are used to hold screen field data, program flow data and applications data. In the case of screen field data, when page and window objects are defined, the fields in the screen partitions are assigned to partition eternal variables. The TBOL Object Linker resolves these references and at program execution time the Reception System transfers data between the screen fields and their associated partition external variables. The TBOL program has access to the variables, which contain the data entered in the screen fields by the user, and the user has access to the screen fields of which contain the data placed in the variables by the program.

For program flow data, partition external variables are used to hold the object identifiers needed by a TBOL program for transferring control. These may be page object identifiers for transfer to another text/graphic screen page, or window object identifiers needed to open a window on the current page. As in the case of screen field data, flow data values are placed in partition external variable by the TBOL Object Linker.

Finally, for application data, partition external variables can be used to hold partition-specific application data such as tables of information needed by the program to process the expected screen field input.

With regard to the global external data structure, the predefined global external data structure can contain up to 32,000 variables for TBOL system data. All TBOL programs have access to the global external data structure and the variables it contains. The variables in a global external data structure are character string variables like the ones one defines in the data section of a program. The global external variables are named #1 through #32,000. These variables are assigned and controlled by the TBOL database administrator which maintains a file of DEFINE compiler directive statements which assign meaningful names to the global external variables in use. In the preferred embodiment, the MS-DOS file specification for this file can, for example be TBOLLIB.backslash.TBOL.SYS. In this regard, the COPY compiler directive is used to copy TBOL.SYS into a source code input stream. Subsequent statements in the program source code can reference the global external system variables by using the meaningful names assigned by the DEFINE statements in this file.

Examples of global external variables are: SUS.sub.-- RETURN.sub.-- CODE, which is assigned a return code value after the execution of certain TBOL program verb statements; SYS.sub.-- DATE, which contains the current system date; and SYS.sub.-- TIME, which contains the current system time.

With regard to the TBOL program code section, as noted above, every TBOL program must have a code section. The code section contains the program logic which is composed of one or more procedures. In accordance with this arrangement, a procedure begins with the keyword PROC followed by an equal sign (=) and then the name of the procedure. The body of the procedure is composed of procedure statements, ending with the END.sub.-- PROC statement. For example:

PROC=proc.sub.-- name statement ›statement . . . ! END.sub.-- PROC; where "proc.sub.-- name" is an identifier; i.e. the name of the procedure, and "statement" is a TBOL procedure statement as described below.

In accordance with the invention, at program execution time, control is given to the first procedure in the program. This is the mainline procedure. From then on, the flow of procedure execution is controlled by the logic contained in the procedures themselves.

Each procedure statement begins with a TBOL keyword called a verb. However, as noted above, the name of a procedure can also act as the verb in a procedure statement, exactly as if it were a TBOL verb. In such case, the data in any statement operand is moved into parameter registers and control is passed to the other procedure. No special linkage or parameter passing conventions are needed. As will be appreciated by those skilled in the art, this is a powerful feature which enables the application programmer to extend the language vocabulary to include his own library of application-oriented verb commands and commonly used procedures.

When control is transferred to another procedure, as noted, the "called" procedure returns control to the "calling" procedure with a RETURN or END.sub.-- PROC statement, where RETURN and END.sub.-- PROC are TBOL verbs described more fully hereafter. Upon return, the "calling" procedure's parameter data, if any, is restored in the parameter registers, and program execution resumes with the next statement. Recursive logic is possible by using the name of the current procedure as the verb in a procedure statement, thus causing the procedure to "call" itself.

In accordance with the design of TBOL, any procedure statement may be preceded with ore or more identifying labels. A label consists of an Identifier followed by a colon (:). For example:

(stmt.sub.-- label: . . .) statement

where "stmt.sub.-- label" is an Identifier, for the statement, and "statement" is a TBOL procedure statement.

Procedure statement labels are used for transferring control to another statement within the same procedure using a GOTO or GOTO.sub.-- DEPENDING.sub.-- ON statement (TBOL verbs described more fully hereafter).

GOTO and or GOTO.sub.-- DEPENDING.sub.-- ON statement can also be used to transfer control to another procedure. Transfer to another procedure is done by using the target procedure name as the verb in a statement.

Also in accordance with the design of TBOL, all procedural logic is constructed from statements designed to execute in three basic patterns: sequential, conditional, or repetitive. In the case of a sequential pattern, the sequential program logic consists of one or more procedure statements. In the case of a conditional pattern, the conditional programlogic is constructed using IF . . . THEN . . . ELSE and GOTO.sub.-- DEPENDING.sub.-- ON key words, described more fully hereafter. Finally, in the case of a repetitive pattern, the repetitive program logic is constructed using WHILE . . . THEN key words or IF . . . THEN . . . ELSE and GOTO key words also described more fully hereafter.

In accordance with the TBOL design, a procedure statement may contain operand following the verb. In the case of procedure statements, there are five types of procedure statement operand; data names; group data names; system registers, label identifiers, and literals. In this arrangement, data names are the names of variables, and data name operand can be either field names; field numbers with subscripts or array names with subscripts. In the case of filed names, a field name is the identifier used as the name of a variable in a data structure in the data section of the program, or the name of TBOL-defined variable in an external data structure.

For field names with subscripts, a field name followed by a subscript enclosed in parentheses (()) refers to a following field. The subscript must be an integer number expressed as a literal or contained in a variable field. The subscript base is 1. For example: CUST.sub.-- NAME(1) refers to the field

CUST.sub.-- NAME, and CUST.sub.-- NAME(2) refers to the field following CUST.sub.-- NAME.

For array names with subscripts, an array name is the identifier used as the name of an array in a data structure in the data section of the program. An array name followed by a subscript enclosed in parentheses (( )), refers to an individual element in the array. The subscript must be an integer number expressed as a literal or contained in a variable field. The subscript base in 1, so the first element in an array is referenced with a subscript of 1.

In the case of procedure statement group data name operand, the group data names are the names of data structures or arrays. Group data names are used in statements where the verb allows data structures or arrays to be treated as a single unit. For example, the TBOL MOVE verb allows the use of group data name operand. If the names of two arrays as group data operand are used, the contents of each element in the source array is moved to the corresponding element in the destination array. Here the array names are specified without subscripts. However, if the names of two data structures as group data operand are used, the contents of each variable in the source data structure is moved to the corresponding variable in the destination data structure.

With regard to system register operand, they can be either integer registers I1 through I8, or decimal registers D1 through D8, or parameter registers P1 through P8.

In the case of label identifiers, the label identifiers are the identifiers used as procedure statement labels described above.

Continuing, literal operand can be either, integer numbers, decimal numbers or character strings. Where the literal operand are integer numbers, the integer is composed of the digits 0 through 9. Where a negative integer is to be represented, a minus sign (-) is allowed in the left-most position. However, a decimal point is not allowed. Accordingly, the minimum value that can be represented is -32,767, and the maximum value is 32,767. Where the literal operand is a decimal number, the decimal number is composed of the digits 0 through 9 with a decimal point (.) where desired. A minus sign (-) is allowed in the left-most position. Thus the minimum allowable value is 31 9999999999999.99999999999999, and the maximum value is 9999999999999.9999999999999.

Further, where the literal operand is a character string, the character string is composed of any printable characters or control characters. Character strings are enclosed in single quotes (`). To include a single quote character in a character string, it must be preceded with the backslash character (.backslash.). For example: .backslash.`. To include a new line character in a character string, the control character .backslash.n is used. For example; `this causes a new line: .backslash.n`. To include binary data in a character string, the hex representation of the binary data is preceded with the backslash character (.backslash.). For example; `this is binary 01110111:.backslash.77`.

The syntax of a complete TBOL program is illustrated in the following example program.

```
HEADER   PROGRAM program.sub.-- name
SECTION
DATA     DATA
SECTION
:        data.sub.-- Structure.sub.-- name-1={1st data structure}
:        .
:        variable.sub.-- name.sub.-- 1
:        .
:        variable names
:        .
```