```
:       variable.sub.-- name.sub.-- n
:       .
:       data structures
:       .
:       data.sub.-- structure.sub.-- name.sub.-- n={nth data structure}
:       .
:       variable.sub.-- name.sub.-- 1,
:       .
:       variable names followed by commas
:       .
:       variable.sub.-- name.sub.-- n;
CODE    PROC proc.sub.-- name.sub.-- 1={mainline procedure}
SECTION
:       .
:       procedure statements
:       .
:       IF x = x THEN EXIT: {if done,ret to:RS Sys}
:       procedure statements
:       .
:       END.sub.-- PROC; {end of mainline procedure}
:       .
:       procedures
:       .
:       PROC proc name.sub.-- n={nth procedures}
:       .
:       procedure statements
:       .
:       IF x = x THEN RETURN; {if done,ret to: "calling"proc}
:       procedure statements
:       END--PROC; {end of nth procedure}
        {end of program}
```

In accordance with the invention, the TBOL compilerenables portability of TBOL programs. Specifically, the TBOL compiler is capable of generating compact data streams from the TBOL source code that can be interpreted by any reception system configured in accordance with the invention, i.e., a personal computer running the reception system application software. For this arrangement, the compiler input file containing the TBOL source code may have any name. For example, the extension .SRC can be used.

During the compilation, three files are generated. Their names are the same as the source code file; their extensions identify their contents. For example, when the file names INPUT.SRC is compiled the following files are generated by the compiler: INPUT.SYM which contains a symTBOL directory; INPUT.COD which contains the compiled code; and INPUT.LST which contains the listing.

In order to resolve an undefined procedure, the TBOL compiler automatically search the local MS-DOS directory TBOLLIB for a file named procname.LIB, where procname is the name of the unresolved procedure. IF procname.LIB is found, the compiler will automatically copy it into the source code stream after the program source text has ended.

In addition to the undefined procedures facility above noted, the TBOL compiler also may be caused to substitute one text string for another. This accomplished by a DEFINE directive.

Wherever the text pattern specified in operand 1 is found in the source code stream, it is replaced by the

compiler with the text pattern specified in operand 2. The syntax for the procedure is:

DEFINE source.sub.-- pattern,replacement.sub.-- -pattern;

where "source-pattern" is the text in the source code which the compiler is to replace, and "replacement.sub.-- pattern" is the text the compiler will use to replace source.sub.-- pattern.

If source.sub.-- pattern or replacement.sub.-- pattern contain any blank (space) characters, the text must be enclosed in single quotes (`). Further, the compiler can be made to eliminate certain text from the input source stream by using a null text string for the replacement.sub.-- pattern (").

It is to be noted that while DEFINE directives are normally placed in the data section, they can also be placed anywhere in the source code stream. For example, if the name CUST.sub.-- NUMBER has been used in a TBOL application program to refer to a partition external variable named &6. The DEFINE statement DEFINE CUST.sub.-- NUMBER,.&6 would cause the compiler to substitute &6 whenever it encounters CUST.sub.-- NUMBER in subsequent statements.

As a further illustration, if the words MAX and MIN are defined with numeric values, DEFINE MAX,1279; and DEFINE MIN,500; MAX and MIN can be used throughout the program source code rather than the actual numeric values. If the values of MAX and MIN change in the future, only the DEFINE statements will need to be changed.

Still further, the compiler can also be caused to copy source code from some other file into the compiler input source code stream. This can be accomplished with a directive entitled COPY. With the use of the COPY directive, the source code contained in the file specified in operand 1 is copied into the source code stream at the point where the COPY statement is read by the compiler. For example, the syntax would be:

COPY `file.sub.-- name`;

where "file.sub.-- name" is the name of the file containing source code to be inserted in the source code steam at the point of the COPY statement. In this arrangement, file.sub.-- name must be enclosed in single quotes (`), and file.sub.-- name must conform to the operating system file naming rules (in the current preferred embodiment, those of MS-DOS). Further, the file referenced in a COPY statement must reside in the TBOLLIB directory on the compilation machine. In accordance with the invention the COPY statement can be placed anywhere in the source code stream.

By way of illustration, the COPY statement COPY `TBOL.SYS`; causes the compiler to insert source text from the file TBOL.SYS. This file is maintained by the TBOL Database Administrator, and contains DEFINE statements which assign meaningful names to the TBOL system variables in the global external data structure.

As shown in Table 2, 25 verbs are associated with data processing; 15 with program flow; 5 with communications; 6 with file management, 5 with screen management; 1 with object management and 2 with program structure for a total of 59. Following is a alphabetical listing of the TBOL verbs, together with a description of its function and illustration of its syntax.

ADD

The ADD verb adds two numbers. Specifically, the number in operand 1 is added to the number in operand 2. Thus, the number in operand 1 is uncharged, while the number in operand 2 is replaced by

the sum of the two numbers. The syntax for ADD is:

ADD number1,number2;,

where number1 contains the number to be added to number2. In this arrangement, number1 can be a data name; system register or literal number. As is apparent, number2 contains the second number, and is overlaid with the resulting sum. Number2 can be a data name or system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the add operation. In accordance with this embodiment, fractions will be truncated after 13 decimal places, and whole numbers will not contain a decimal point. Negative results contain a minus sign (-) in the left-most position.

AND

The AND verb performs a logical AND function on the bits of two data fields. The logical product (AND) of the bits of operand 1 and operand 2 is placed in operand 2. Moving from left to right, the AND is applied to the corresponding bits of each field, bit by bit, ending with the last bit of the shorter field. If the corresponding bits are 1 and 1, then the result bit is 1. If the corresponding bits are 1 and 0, or 0 and 1, or 0 and 0, then the result bit is 0. In this arrangement, the data in operand 1 is left unchanged, and the data in operand 2 is replaced by the result.

The AND syntax is:

AND field1,field2;

where "field1" contains the first data field, which can be a data name, system register, I1-I8 or P1-P8 only, or a literal. Continuing, "field2" contains the second data fields, and the contents of field2 are overlaid by the result of the AND operation. Field2 can be a data name, a system register: I1-I8 or P1-P8 only.

As will be appreciated, the AND verb can be used to set a bit to 0.

CLEAR

The CLEAR verb sets one or more variables to null. The CLEAR statement may have either one or two operand. If only one operand is specified, it may contain the name of a field, an array or a data structure. If the operand contains a field name, then that field is set to null. If the operand contains an array name, then all elements of the array are set to null. If the operand contains the name of a data structure, then all fields and array elements in the data structure are set to null. If two operand are specified, then each operand must contain the name of a field. In this case, all fields, beginning with the field in operand 1 and ending with the field in operand 2, are set to null.

The syntax for CLEAR is:

CLEAR name1 ›,name2!;

where "name1" contains the name of a field, array, or data structure to be set to null. If "name2" is specified, name1 must contain a field name. Name1 can be a data name, group data name, or system register P1-P8 only. Further, name2 contains the last field name of a range of fields to be set to null, and can be a data name, group data name, or system register P1-P8 only.

CLOSE

The CLOSE verb is used to close a reception system file after file processing has been completed. By using CLOSE, the file named in operand 1 is closed. If no operand is specified, then all open files are closed. The CLOSE syntax is:

CLOSE ›filename!;

where, "filename" contains the name of the reception system file to be closed. The file name "PRINTER" specifies the system printer. Otherwise, the name of the file must be a valid MS-DOS file specification; e.g., ›drive:!›.backslash.path.backslash.!name›.extension! File name can be a data name, or system register P1-P8 only. When file processing is complete, the file must be closed.

CLOSE.sub.-- WINDOW

The CLOSE.sub.-- WINDOW verb is used to close the open window on the base screen and, optionally, open a new window by appending the partial operator.sub.-- OPEN to the middle of the verb (as shown below). Specifically, by using CLOSE.sub.-- WINDOW, the open window on the base screen is closed. If no operand is specified, program execution continues with the next statement in the program which last performed an OPEN.sub.-- WINDOW. If operand 1 is specified, the window whose object ID is contained in operand 1 is opened, and program execution continues with the first statement of the program associated with the newly opened window object.

The CLOSE.sub.-- WINDOW syntax is:

CLOSE.sub.-- WINDOW ›window-id!;

where, "window-id" contains the object ID of a new window to be opened after closing the currently open window. A window-id can be a data name, system register P1-P8 only, or a literal. The CLOSE.sub.-- WINDOW verb can only be performed by a window program; i.e., a program associated with a window object. CLOSE.sub.-- WINDOW is the method by which a window program relinquishes control. A window program can also close itself by performing one of the following verbs: NAVIGATE, TRIGGER.sub.-- FUNCTION. Although a window program cannot perform a OPEN.sub.-- WINDOW operation, it can use CLOSE.sub.-- WINDOW to close itself and open another window This process can continue through several windows. Finally, when a window program performs a CLOSE.sub.-- WINDOW without opening a new window, program control does not work its way back through all the window programs. Instead, control returns to the non-window program which opened the first window. Program execution continues in that program with the statement following the OPEN.sub.-- WINDOW statement.

CONNECT

The CONNECT verb dials a telephone number. The telephone number contained in operand 1 is dialed. The telephone line status is returned in the system variable SYS.sub.-- CONNECT.sub.-- STATUS. The syntax for CONNECT is:

CONNECT phone.sub.-- number;

where "phone.sub.-- number" contains the telephone number to be dialed. Phone.sub.-- number can be a data name, system register P1-P8 only, or a literal.

DEFINE.sub.-- FIELD

The DEFINE.sub.-- FIELD verb is used to define a screen field at program execution time. From five to seven operand specify a single-line or multiple-line field within the currently active screen partition; i.e. the partition associated with the running program. The field is dynamically defined on the current screen partition.

The syntax for DEFINE.sub.-- FIELD is:

DEFINE.sub.-- FIELD name,row,coln,width,height ›,object.sub.-- id ›,state!!;

where "name" is the field to receive the name of a partition external variable. When this statement is performed, a screen field is defined and it is assigned to a partition external variable. The partition external variable name is placed in the name operand. Name may be a data name, or system register P1-P8 only.

Continuing "row" in the DEFINE.sub.-- FIELD syntax contains the row number where the field starts. The top row on the screen is row number 1. Row can be a data name, system register P1-P8, or a literal. "Column" contains the column number where the field starts. The left-most column on the screen is column number 1. Column can be a data name, system register P1-P8 only, or a literal. In the DEFINE.sub.-- FIELD syntax, "width" contains a number specifying how many characters each line the field will hold. Width can be a data name, system register P1-P8 only, or a literal. Further, "height" contains a number specifying how many lines the field will have. For multiple-line fields, each field line will begin in the column number specified in the column operand. Height can be a data name, system register P1-P8 only, or a literal.

Yet further, in the DEFINE.sub.-- FIELD syntax, "object.sub.-- id" contains the object ID of a field post processor program that is to be associated with this field. Object.sub.-- id can be a data name, system register P1-P8 only, or a literal. Finally, for the DEFINE.sub.-- FIELD syntax "state" contains a character string which is to be placed in parameter register P1 when the program specified in the object.sub.-- id operand is given control. State can be a data name, system register P1-P8 only, or a literal.

In the case of the DEFINE.sub.-- FIELD verb, if the object-id operand is specified, then the post processor program object is obtained only on a "commit" event; avoiding the need for a synchronous FETCH. Since DEFINE.sub.-- FIELD defines a field only in the screen partition associated with the running program, a program can not define a field in some other screen partition with which it is not associated. Additionally, page-level processor programs which are not associated with a particular screen partition can not use this verb.

DELETE

DELETE is used to delete a reception system file for file processing. the file named in operand 1 is deleted. The syntax for DELETE is:

DELETE ›filename!;

where "filename" contains the name of the reception system file to be deleted. Filename can be a data name or system register P1-P8. Filename must be a valid operating specification.

DISCONNECT

The DISCONNECT verb "hangs up the telephone", thus, terminating the telephone connection. The syntax for DISCONNECT is simply:

DISCONNECT.

DIVIDE

The DIVIDE verb divides one number by another. The number in operand 2 is divided by the number in operand 1. The number in operand 1 is unchanged, however, the number in operand 2 is replaced by the quotient. If operand 3 is specified, the remainder is placed in operand 3. The syntax for DIVIDE is:

DIVIDE number1,number2 ›,remainder!;

where "number1" contains the divisor, i.e. the number to be divided into number2. Number1 can be a data name, system register, or literal number. Continuing, "number2" contains the dividend; i.e., the number to be divided by number1. The contents of number2 are overlaid by the resulting quotient. Number2 can be a data name, or a system register. And, "remainder" is a variable or system register designated to hold the remainder of the divide operation. Remainder can be a data name, or a system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the divide operation. Fractions will be truncated after 13 decimal places, while whole number will not contain a decimal point. Negative results will contain a minus sign (-) in the left-most position.

DO . . . END

The keyword DO specifies the beginning of a block of statements; the keyword END specifies the end of the block. A block of statements, bracketed by DO and END can be used as a clause in an IF or WHILE statement. In an IF statement, either the THEN clause or an optional ELSE clause can be executed, based upon the evaluation of a boolean expression. In a WHILE statement, the THEN clause is executed repetitively until a boolean expression is false.

The syntax for DO . . . END is:

DO . . . block . . . END;

where "Block" is any number of TBOL statements. As shown, the keyword DO is not followed by a semicolon, and the END statement requires a terminating semicolon.

EDIT

The EDIT verb gathers and edits data from multiple sources, then joins it together and places it in the specified destination field. Data from one to six sources, beginning with operand 3, is edited in accordance with the mask contained in operand 2. The edited data, joined together as a single character string is places in the output destination field specified in operand 1.

The EDIT syntax is EDIT output,mask,source ›,source . . . !;, where "output" contains the name of the destination field for the edited data. After performance of the EDIT statement, the destination field will

contain "sub-fields" of data; one for each source operand. Output can be a data name, or a register P1-P8 only.

Continuing, "mask" contains a character string consisting of one edit specification for each source operand. Edit specifications are in the form: %›-!›min.max!x, where "%" indicates the beginning of an edit specification; "-" indicates left-adjustment of the source data in the destination sub-field, and "min.max" are two numbers, separated by a decimal point, which specify the minimum and maximum width of the edited data in the destination sub-field, and "x" is an alpha character which controls the retrieval of data from the corresponding source operand. Further, "x" can be a "d" to indicate a digit, characters retrieved from the corresponding source operand are converted to integer format; or "x" can be an "f" to indicate floating point, characters retrieved from the corresponding source operand are converted to a decimal format; or an "x" can be an "s" to indicate a string, characters retrieved from the corresponding source operand are converted to character format; or an "x" can be a "c" to indicate a character, only one character is retrieved from the corresponding source operand, and is converted to character format.

Characters in mask which are not part of edit specifications are placed in output as laterals. Mask can be a data name, or system register P1-P8 only.

Continuous source contains the source data to be edited. The EDIT statement may contain up to six source operand. Mask must contain an edit specification for each source operand specified. Source can be a data name, a system register, or a literal.

END.sub.-- PROC

The END.sub.-- PROC verb identifies the last physical statement in a procedure definition. Control returns to the "calling" procedure and program execution continues with the statement following the "call" statement. The syntax for END.sub.-- PROC is:

END.sub.-- PROC;

An END.sub.-- PROC statement is required as the last physical statement in every procedure. Accordingly, a procedure may contain only one END.sub.-- PROC statement.

An END.sub.-- PROC statement in a "called" procedure is equivalent to a RETURN statement. Further, an END.sub.-- PROC statement in the highest level procedure of a program is equivalent to an EXIT statement.

ERROR

The ERROR verb causes the Reception System to reset. Processing resumes with a new page template object. Execution of the currently running program is terminated and control returns to the Reception System. ,The reception System resets itself. Program execution then resumes with the first statement in the program associated with the page template object specified in operand 1.

The ERROR syntax is:

ERROR object.sub.-- id;

where "object.sub.-- id" contains the object ID of a page template object. After the Reception System reset control is transferred to the program associated with the page template object. Object.sub.-- id can

be a data name, a system register P1-P8 only, or a literal.

The ERROR verb is used to continue a text/graphic session when the currently running program encounters a condition which can only be resolved by a reset of the Reception System.

EXIT

The EXIT verb is used to transfer program control to the Reception System. When EXIT executes, the currently running program is ended. The data in operand 1 is moved to SYS.sub.-- RETURN.sub.-- CODE, and control is returned to the Reception System. The syntax for EXIT is:

EXIT return code;

where "return-code" contains data to be moved to SYS.sub.-- RETURN.sub.-- CODE prior to transfer of control to the Reception System. A value of 0 indicates a normal return. A non-zero value indicates an error condition. Return.sub.-- code can be a data name, system register, or a literal.

The EXIT verb is the normal way to end processing in a TBOL program. In the highest level procedure of a program a RETURN or an END.sub.-- PROC is equivalent to an EXIT.

FETCH

The FETCH verb is used to retrieve an object from a host system or from the Reception System storage device stage. The object specified in operand 1 is retrieved from its present location and made available in the Reception System. If operand2 is specified, the object's data segment is placed in the operand 2 field.

The syntax for FETCH is:

FETCH object.sub.-- id ›,field!;

where "object.sub.-- id" contains the object ID of the object to be located and retrieved. Object.sub.-- id can be a data name, system register P1-P8 only, or a literal.

In the FETCH syntax "field" contains the name of a field to hold the retrieved object's data segment. Field can be a data name, or a system register P1-P8 only.

When an object might be required for subsequent processing, the field operand should not be specified in the FETCH statement. In that case, the FETCH will be an asynchronous task and the program will not experience a wait. The object is placed in the Reception System ready for use. The field operand is specified when an object is required to immediate use. Here, the FETCH is a synchronous task and the program may experience a wait When the FETCH is completed, the program has access to the FETCHed object's data segment in the field operand.

FILL

The FILL verb is used to duplicate a string of characters repeatedly within a field. The character string pattern contained in operand 2 is duplicated repeatedly in operand 1 until the length of operand 1 is equal to the number specified in operand 3. The syntax for FILL is:

FILL output,pattern,length;

where "output" is the name of the field to be filled with the character string specified in "pattern". Output can be a data name or a system register P1-P8 only, or a literal. Finally, "length" contains an integer number specifying the final length of output. Length can be a data name, system register or a literal.

FORMAT

The FORMAT verb is used to transfer a string of character data into variables defined in the DATA section of the program. The string of character data contained in operand 1 is transferred to DATA section variables using destination and length specification in the format map contained in operand 2. The FORMAT syntax is:

FORMAT source,map;

where "source" contains a string of character data to be transferred to DATA section variables, and can be a data name or system register P1-P8 only.

Continuing, "map", on the other hand, contains a format map consisting of a destination/length specification for each field of data to be transferred. Map is created with the MAKE.sub.-- FORMAT verb prior to execution of the statement.

GOTO

The GOTO verb transfers control to another statement within the currently running procedure. Program execution continues at the statement with the label identifier specified as operand 1. The syntax for GOTO is:

GOTO label.sub.-- id;

where "label.sub.-- id" is a label identifier directly preceding a statement within the currently running procedure. A GOTO statement can be used to transfer control to another procedure. Transfer to another procedure is accomplished by using the target procedure name as the verb in a statement.

GOTO.sub.-- DEPENDING.sub.-- ON

The GOT.sub.-- DEPENDING.sub.-- ON verb transfers control to one of several other statements within the currently running procedure. Operand 1 contains a number, and is used as an index to select one of the label identifiers beginning with operand2 in the statement. Program execution continues at the statement with the selected label identifier.

The syntax for GOTO.sub.-- DEPENDING.sub.-- ON is:

GOTO.sub.-- DEPENDING.sub.-- ON index,label.sub.-- id ›,label.sub.-- id . . . !;

where "index" is an integer number used to select one of the label identifiers in the statement as the point where program execution will continue. If index contains a 1, then program execution continues at the statement with the label identifier specified as operand2. If index contains a 2, then program execution continues at the statement with the label identifier specified as operand 3. And so on. If there is no label.sub.-- id operand corresponding to the value in index, then program execution continues with the statement following the GOTO.sub.-- DEPENDING.sub.-- ON statement. Index can be a data name

or system register. Continuing, "label.sub.-- id" is a label identifier directly preceding a statement within the currently running procedure. Up to 147 label.sub.-- id operands may be specified in a GOTO.sub.-- DEPENDING.sub.-- ON statement.

A GOTO.sub.-- DEPENDING.sub.-- ON statement, however, cannot 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.

IF . . . THEN . . . ELSE

In this verb, the keyword IF directs the flow of program execution to one of two possible paths depending upon the evaluation of a boolean expression. The keyword IF is followed by a boolean expression. The boolean expression is always followed by a THEN clause. The THEN clause may be followed by an ELSE clause. The boolean expression is evaluated to determine whether it is "true" or "false". If the expression is true, program execution continues with the THEN clause the ELSE clause, if present, is skipped. If the expression is false, the THEN clause is skipped; program execution continues with the statement following the clause or clauses.

The syntax for IF . . . THEN . . . ELSE is:

IF boolean THEN clause ›ELSE clause!;

where "boolean" is a boolean expression. Boolean can be a single relational expression or two or more relational expressions separated by the key words AND and OR. These relational expressions can be enclosed with parentheses, and then treated as a single relational expression separated from others with AND or OR. They are evaluated from left to right.

In the syntax, "clause" can be: a single statement, or a block of statements. Where clause is a block of statements, the block begins with the keyword DO and ends with the END verb. Further Clause is always preceded by the keyword THEN or ELSE.

INSTR

The INSTR verb searches a character string to determine if a specific substring of characters is contained within it. The character string in operand 1 is searched for the first occurrence of the character string in operand 2. If a matching string is found in operand 1, an integer number specifying its starting position is placed in operand 3. If a matching string is not found, 0 is placed in operand 3.

The syntax for INSTR is:

INSTR string,pattern,strt.sub.-- pos;

where "string" contains the character string to be searched. String can be a data name, system register P1-P8 only, or a literal.

Continuing, "pattern" contains the character string pattern which may occur within the string operand, and can be a data name, system register P1-P8 only, or a literal.

Finally, "strt.sub.-- pos" is the name of the variable where the starting position (or o) is to be stored. Strt.sub.-- pos can be a data name, or system register P1-P8 only.

LENGTH

The LENGTH verb is used to determine the length of a specified variable. An integer number specifying the number of characters in operand 1 is placed in operand 2. The syntax for LENGTH is:

LENGTH field,length;

where "field" contains the data whose length is to be determined. Field can be a data name, system register P1-P8 only, or a literal.

Continuing, on the other hand, "length" is the name of the variable which is to contains the length of the field operand, and can be a data name, or a system register P1-P8 only.

LINK

The LINK verb transfers control to another TBOL program. Program execution continues at the first statement in the program whose object ID is contained in operand 1. Up to eight parameters may be passed to the "called" program in operands 2-9. Control returns to the statement following the LINK statement when the "called" program performs an EXIT.

The syntax for LINK is:

LINK object.sub.-- id ›,parameter . . . !;

where "object.sub.-- id" contains the object ID of a TBOL program, and can be data name, system register P1-P8, only or a literal. Further, "parameter" contains parameter data for the program whose object ID is contained in operand 1. The contents of the parameter operand 2 through 9, if present, are placed in parameter registers P1 through P8. The number of parameter operand is placed in P0. P0 through P8 are accessible to the "called" program. Parameter can be a data name, system register, or a literal.

LOOKUP

The LOOKUP verb issued to search for an entry in a table of data contained in a character string. Operand 2 contains a single character string consisting of a number of logical records of equal length. Each record consists of a fixed-length keyfield and a fixed-length data field. Operand 3 contains the record length.

Operand 1 contains a search key equal in length to the length of the key field. Operand 2 in searched for a record with a key field equal to operand 1. If a record with a matching key is found, an integer number specifying its starting position is placed in operand 4. If a matching record is not found, 0 is placed in operand 4.

The syntax for LOOKUP is:

LOOKUP schkey,table,rcd.sub.-- lth,result;

where "schkey" contains the key data of the desired record and can be a data name, system register or a litera. Further, "table" contains a character string consisting of a number of equal length logical records, and be a data name or system register P1-P8 only. Yet further, "rcd)lth" contains an integer number equal to the length of a record in a table, and can be a data name, system register, or a literal. Finally,

"result" is the name of the field to receive the result of the search. Result can be a data name, or a system register.

MAKE.sub.-- FORMAT

The MAKE.sub.-- FORMAT verb is used to create a format map for use with the FORMAT verb. From 1 to 255 destination/length specifications contained in operand (beginning in operand 2) are used to create a format map which is stored in operand 1. Operand 1 can then be specified as the map operand in a FORMAT statement.

The MAKE.sub.-- FORMAT syntax is:

MAKE.sub.-- FORMAT map,format›,format . . . !;

where "map" is the name of the variable which is to contain the format map created with this statement. Map will be specified as an operand in a subsequent FORMAT statement to control the transfer of a string of character data to variables. Map can be either a data name or system register P1-P8 only. Continuing, "format" contains a destination/length specification for one logical field of a string of character data. From 1 to 255 format operand can be specified in this statement to create a format map. Each format operand controls the transfer of one logical field of data from a character string when the format map created in this statement is used in a subsequent FORMAT statement. In this arrangement, format can be a data name or a system register P1-P8 only.

A destination/length specification in a format operand always contains a destination field name. The field name is followed by either one or two integer numbers controlling the lengths of the designation field data. The field name and numbers are separated by the colon character, e.g., destination:fix.sub.-- lth:imbed.sub.-- lth, or destination:fix.sub.-- lth, or as destination::imbed.sub.-- lth.

For this approach, "destination is a variable field name which will contain the logical field of data from the character string after the subsequent performance of the FORMAT verb. And, "fix.sub.-- lth" is an integer number between 1 and 33767 specifying a fixed field length for destination. If fix.sub.-- lth is not specified then 2 colon characters are used to separate destination from imbed.sub.-- lth, showing that fix.sub.-- lth has been omitted. In this case, the destination field length is controlled entirely by imbed.sub.-- lth, which must be specified. If fix.sub.-- lth is specified and imbed.sub.-- lth is not, then fix.sub.-- lth characters will be transferred to destination during the subsequent performance of the FORMAT verb. Finally, if fix.sub.-- lth is specified with imbed.sub.-- lth, then destination will have a length of fix.sub.-- lth after the transfer of data by the FORMAT verb.

Continuing, "imbed.sub.-- lth" is an integer number, either 1 or 2 which specifies length of an imbedded length field that immediately precedes the logical field of data in the character string. The imbedded length field contains the length of the logical field of data immediately following. For example, 1 specifies a 1-character length field and 2 specified a 2-character length field.

If imbed.sub.-- lth is not specified then the designation field length is controlled entirely by fix.sub.-- lth which must be specified. If imbed.sub.-- lth is specified and fix.sub.-- lth is not, then the number of characters transferred to destination from the character string is controlled by the number in the one or two-character length field which precedes the logical field of data. If imbed.sub.-- lth is specified with fix.sub.-- lth, then the number of characters transferred to destination from the character string is controlled by the number in the one or two-character length field which precedes the logical field of data. After the transfer of data, if the length of destination is not equal to fix.sub.-- lth, then it is either truncated, or extended with blank characters as necessary.

MOVE

The move verb copies data from one or more source fields into an equal number of destination fields. The data contained in the operand 1 data structure field (or fields) replaces the contents of the operand 2 data structure field (or fields). Operand 1 data remains unchanged. Normally, the moved data is converted to the data type of the destination. If the key word ABS is included as operand 3, then data conversion does not take place.

The syntax for MOVE is:

MOVE source,destination›, ABS!;

where "source" is the name of the data structure containing the data to be moved, and can be a data name, or a group data name, or system register, or a literal. Further "destination" is the name of the data structure field (or fields) to receive the source data, and can be a data name, or group data name, or a system register. Finally, "ABS" is a keyword specifying an absolute move; i.e., no data conversion takes place. However, data residing in an integer register will always be in binary integer; and data residing in a decimal register will always be in internal decimal format.

If the source operand is a group data name, then the destination operand must be a group data name. Further, data in all of the fields contained in the source data structure or array are moved to the corresponding fields in the destination data structure or array.

MULTIPLY

The MULTIPLY verb multiplies two numbers. The number in operand 2 is multiplied by the number in operand 1. The number in operand 1 is unchanged. The number in operand 2 is replaced with the product of the two numbers. The syntax for MULTIPLY is:

MULTIPLY number1,number2;

where "number1" contains the first number factor for the multiply operation, and can be a data name, system register or literal; and "number2" contains the second numberfactor for the multiply operation. Following execution, the contents of number2 are overlaid with the resulting of the product. Number2 can be a data name, or a system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number 2 having a different data type after the add operation. Fractions will be truncated after 13 decimal places, and whole numbers will not contain a decimal point. Negative results will contain a minus sign (-) in the left-most position.

NAVIGATE

The NAVIGATE verb is used to transfer control to the TBOL program logic associated with different page template objects. The external effect is the display of a new screen page. Operand 1 contains either a page template object ID, or a keyword representing a navigation target page. Control is returned to the Reception System where the necessary objects are acquired and made ready to continue the videotext session at the specified new page. The syntax for NAVIGATE is:

NAVIGATE object.sub.-- d;

where "object.sub.-- id" contains the object ID of a target page template object, and can be a data name, register P1-P8 only, or a literal.

NOTE

The NOTE verb returns the current position of the file pointer in a reception system file. Operand 1 contains the name of a file. An integer number specifying the current position of the file's pointer is returned in operand 2. The NOTE syntax is:

NOTE filename,position;

where "filename" contains the name of a reception system file. The name of the file must be a valid MS-DOS file specification; e.g., ›drive:!›.backslash.path.backslash.!name›.extension!. Filename can be a data name, or a system register P1-P8 only. Continuing, "position" is the name of the field to receive the current position of the file pointer for the file specified in filename. This will be an integer number equal to the numeric offset from the beginning of the file; a 10 in position means the file pointer is positioned at the 10th character position in the file. Position can be a data name, or system register

OPEN

The OPEN verb is used to open a reception system file for file processing. The file named in operand 1 is opened for processing in the mode specified an operand 2. The syntax for OPEN is:

OPEN filename, INPUT:OUTPUT:I/O:APPEND:BINARY;

where "filename" contains the name of the reception system file to be opened. As will be appreciated with this convention,the file name PRINTER specified the system printer. Otherwise, the name of the file must be a valid MS-DOS file specification; e.g.›drive:!›.backslash.path.backslash.!name›.extension!. Filename can be a data name, or system register P1-P8 only.

Further, "INPUT" is a keyword specifying that the file is to be opened for reading only; "OUTPUT" is a keyword specifying that the file is to be opened for writing only; "I/O" is a key word specifying that the file is to be opened for both reading and writing; "APPEND" is a keyword specifying that the file is to be opened for writing, where new data is appended to existing data; and "BINARY" is a keyword specifying that the file is to be opened for both reading and writing. Where all file data is in binary format.

OPEN.sub.-- WINDOW

The OPEN.sub.-- WINDOW verb is used to open a window on the base screen. The window whose object ID is contained in operand 1 is opened. Program execution continues with the first statement of the program associated with the newly opened window object. The syntax for OPEN.sub.-- WINDOW is:

OPEN.sub.-- WINDOW window.sub.-- id;

where "window.sub.-- id" contains the object ID of the window to be opened on the basescreen, and can be a data name, or system register P1-P8 only or a literal.

After performance of the OPEN.sub.-- WINDOW statement, program execution continues with the first

statement of the window program; i.e., the program associated with the newly opened window object. A window program relinquishes control by performing a CLOSE.sub.-- WINDOW. Although a window program cannot perform an OPEN.sub.-- WINDOW, it can use CLOSE.sub.-- WINDOW to close itself and open another window. This process can continue through several windows. Finally, when a window program performs a CLOSE.sub.-- WINDOW without opening a new window, program control does not work its way back through all the window programs. Instead, control returns to the non-window program which opened the first window. Program execution continue in that program with the statement following the OPEN.sub.-- WINDOW statement. A window program can also close itself by performing one of the following verbs: NAVIGATE; or TRIGGER.sub.-- FUNCTION. In such cases, control does not return to the program which opened the window.

OR

The OR verb performs a logical OR function on the bits of two data fields. The logical sum (OR) of the bits of operand 1 and operand 2 is placed in operand 2. Moving from left to right, the OR is applied to the corresponding bits of each field, bit by bit, ending with the last bit of the shorter field.

If the corresponding bits are 1 and 1, then the result bit is 1. If the corresponding bits are 1 and 0, or 0 and 1, then the result bit is 1. If the corresponding bits are 0 and 0, then the result bit is 0.

The data in operand 1 is left unchanged. The data in operand 2 is replaced by the result.

The syntax for OR is:

OR field1,field2;

where "field1" contains the first data field, and can be a data name, or system register I1-P8 or P1-P8 only, or a literal. Further, "field2" contains the second data field. The contents of field2 are overlaid by the result of the OR operation. Field2 can be a data name, or system register I1-I8 or P1-P8 only. As will be appreciated by those skilled in the art, the OR verb can be used to set a bit to 1.

POINT

The POINT verb is used to set the file pointer to a specified position in a reception system file. Operand 1 contains the name of a file. The file's pointer is set to the position specified by the integer number in operand 2. The POINT syntax is:

POINT filename,position;

where "filename" contains the name of a reception system file. The name of the file must be a valid MS-DOS file specification; e.g. ›drive:!›.backslash.path.backslash.!name›.extension!. Filename can be a data name, or system register P1-P8 only. Further, "position" contains an integer number equal to the desired position of the file pointer for the file specified n filename. A 10 in position means the file pointer will be positioned at the 10th character position in the file. Position can be a data name, or system register or literal.

POP

The POP verb transfers data from the top of the system stack to a variable field. The contents of operand 1 are replaced with data removed from the top of the system stack. The POP syntax is:

POP field;

where "field" is the name of the variable field to receive data from the stack, and can be a data name or a system register.

PUSH

The PUSH verb transfers data from a variable field to the top of the system stack. The data contained in operand 1 is placed on the top of the system stack, "pushing down" the current contents of the stack. The contents of operand 1 remain unchanged. The PUSH syntax is:

PUSH field;

where "field" is the name of the variable field containing data to be "pushed" on the stack, and can be a data name, or a system register, or a literal.

READ

The READ verb is used to read data from a reception system file into a variable field. Operand 1 contains the name of a file. Data is read from the file, beginning with the character position specified by the current contents of the file's pointer. Data read from the file replaces the contents of operand 2. Operand 3 may be present, containing an integer number specifying the number of characters to be read. For ASCII files, data is read from the file until the first end-of-line character (ASCII 13) is encountered. Or, if operand 3 is present, until the number of characters specified in operand 3 is read. For binary files, operand 3 is required to specify the length of the data to be read from the file.

The syntax for READ is:

READ filename,input ›,length!;

where "filename" contains the name of a reception system file, which must be a valid MS-DOS file specification, e.g. ›drive:!›path.backslash.!name›.extension!. Filename can be a data name, or system register P1-P8 only. Continuing, "input" is the name of the variable field to receive data read from the file, and can be a data name, or a system register P1--P8 only. Finally, "length" contains an integer number. For ASCII files, length specifies the maximum number of characters to be read. For binary files, length specifies the length of the data to be read.

As will be appreciated by those skilled in the art, in order to perform a READ operation, a file must first be opened as INPUT or I/O before the READ operation can take place.

RECEIVE

The RECEIVE verb is used to access the expected reply to a message sent previously to a host system. Operand 1 contains the message ID of a message sent previously to a host system. The message reply from the host replaces the contents of operand 2. The RECEIVE syntax is:

RECEIVE msg.sub.-- is,message;

where "msg.sub.-- id" contains the ;message ID of a message sent previously to a host system, and can be a data name, or a system register P1-P8 only. Further, "message" is the name of the variable field to receive the incoming message reply, and can be a data name, or a system register P1-P8 only.

RELEASE

The RELEASE verb reclaims memory space in the reception system by deleting a block of data saved previously with the SAVE verb. The block of data named in operand 1 is deleted from memory.

The syntax for RELEASE is:

RELEASE block.sub.-- name;

where "block.sub.-- name" contains a block name used in some previously performed SAVE statement,and can be a literal.

REFRESH

The REFRESH verb causes the current screen fields to receive the contents of the associated partition external variables. The contents of all fields on the current screen are replaced with the contents of their corresponding partition external variables. The REFRESH syntax is:

REFRESH.

The REFRESH operation occurs automatically whenever all programs for a given event (for example, commit; field end; or initial display) have finished execution. Therefore a program should only perform a REFRESH statement if fields are updated during an event.

RESTORE

The RESTORE verb is used to restore the previously saved contents of a block of variables. The block of data named in operand 1 replaces the contents of a block of variables, beginning with the variable named in operand 2. The RESTORE syntax is:

RESTORE block.sub.-- name,field1;

where "block.sub.-- name" contains a block name used in some previously performed SAVE statement, and can be a literal. Further, "field1" is the name of the first field or a data structure to receive data from the block specified in block.sub.-- name. Field1 can be a data name, or a group data name.

RETURN

The RETURN verb is used to return control to the procedure which "called" the currently running procedure. Execution of the currently running procedure is ended. The data in operand 1 is moved to SYS.sub.-- RETURN.sub.-- CODE, and control is returned to the procedure which "called" the currently running procedure.

The RETURN syntax is:

RETURN return-code;

where "return-code" contains data to be moved to SYS.sub.-- RETURN.sub.-- CODE prior to transfer of control to the "calling" procedure, and can be a data name, or system register, or a literal. It should be noted that in the highest level procedure of a program, a RETURN or an END.sub.-- PROC is

equivalent to an EXIT.

SAVE

The SAVE verb is used to save the contents of a block of variables. Operand 1 contains a name to be assigned to the block of saved data. This name will be used later to restore the data. If operand 2 is specified without operand 3, then operand 2 may contain the name of a field, an array, or a data structure. In this case, the contents of the field; or the contents of all the elements in the array; or the contents of all the fields in the data structure are saved under the name specified in operand 1. If operand 2 and operand 3 are specified, then they both must contain a field name. In this case, the contents of all the fields, beginning with the field in operand 1 and ending with the field in operand 2, are saved under the name specified in operand 1.

The syntax for SAVE is:

SAVE block.sub.-- name,name1 ›,name2!;

where "block.sub.-- name" contains a block name to be assigned to the saved data, and will be used subsequently to restore the saved contents of the fields. Block.sub.-- name can be a data name, system register P1-P8 only, or a literal. Continuing, "name1" contains the name of a field, array, or data structure to be saved. If name2 is specified, name1 must contain a field name. Name1 can be a data name. Further, "name2" contains the last field name of a range of fields to be saved, and it can be a data name.

SEND

The SEND verb is used to transmit a message to a host system. The message text contained in operand 2 is transmitted from the reception system using a message header constructed from the data contained in operand 2. Operand 3, if present, indicates that an incoming response to the message is expected. The syntax for SEND is:

SEND message ›,RESPONSE:TIMEOUT!; ps where "message" contains the outgoing message text (the header data for which has been placed in GEVs before SEND), and can be a data name, or a system register, or a literal. "RESPONSE" is a keyword indicating that a response to the message is expected. "TIMEOUT" is a parameter that sets the number of seconds for message time-out.

After performance of the SEND statement, the global external system variable SYS.sub.-- LAST.sub.-- MSG.sub.-- ID contains a message ID number assigned to the outgoing message by the Reception System. This message ID number can be used later in a RECEIVE statement.

SET.sub.-- ATTRIBUTE

The SET.sub.-- ATTRIBUTE verb is used to set or change the color and input format attributes of a screen field. The characteristics of the screen field expressed as operand 1 are set or changed according to the specifications contained in operand 2. The syntax for SET.sub.-- ATTRIBUTE is:

SET.sub.-- ATTRIBUTE name, attr.sub.-- list;

where "name" expresses the name of the field whose characteristics are to be set or changed. This is a partition external variable name, and if the name is expressed as a literal; e.g., "SET.sub.-- ATTRIBUTE 1, . . . ", then this is taken to mean that the attributes of the partition external variable &1 contains the

name of the partition external variable whose attributes are to be set by this statement.

Further, "attr.sub.-- list" is a literal character string containing a list of key words and values describing the desired attributes to be assigned to the field expressed in operand 1.

When SET.sub.-- ATTRIBUTE is performed, existing field attributes remain in effect unless superseded by the attribute list contained in operand 2. The attribute list operand literal is in the form:

keyword›(values)!›,keyword›(values)! . . . !.

It should also be noted that where key words and their associated values are: "DISPLAY", not user input data can be entered in a field with this attribute; "INPUT", a field with this attribute can receive user input data; "ALPHABETIC", an INPUT field with this attribute can receive any alphabetic character: A through A, and blank; "ALPHANUMERIC", an "INPUT", field with this attribute can receive any displayable character; "NUMERIC", an INPUT field with this attribute can receive any numeric character: 0 through 9, ($), (,), (.), and (-); "PASSWORD", an INPUT field with this attribute is intended for use as a password field. Any character entered by the user is displayed in the field as an asterisk (*); "ACTION", a field with this attribute is a TBOL "action" field; "COLOR-(fg,bg)", where fg and bg are numeric values specifying the foreground and background colors of the field; "FORM(pattern)", where pattern specifies the input data format for this field. Pattern may contain "A", an alphabetic character of A through Z, which must be in this position; "a", an alphabetic character of A through Z, or a blank, which must be in this position; "N" a number character of 0 through 9, or ($), (,), (.), or (-) which must be in this position; "n", a numeric character of 0 through 9, or ($), (,), (.), (-), or a blank may occupy this position; "X", any displayable character which must be in this position; and "x", any displayable character or a blank which must be in this position.

Any other character in the pattern is displayed in the field as a literal, and acts as an autoskip character at user input time. To include any of the pattern characters as literals in the pattern, they must be preceded by the backslash character. For example, to include the character "A: as a literal in a pattern it would code as ".backslash.A". To include the backslash character as a litera, it would code as ".backslash. .backslash.".

SET.sub.-- CURSOR

The SET.sub.-- CURSOR verb moves the cursor to the field specified as operand 1, itself specified as a field number. The syntax for the SET.sub.-- CURSOR verb is:

SET.sub.-- CURSOR ›field number!

SET.sub.-- FUNCTION

The SET.sub.-- FUNCTION verb changes and/or filters a "logical function" process program. The syntax for SET.sub.-- FUNCTION is: SET-FUNCTION function-id, status›,program-object-id ›,state!!; where "function.sub.-- id.sub.-- is the logical function" identifier; "status" is one of the following key words: "DISABLE"; "FILTER"; or "ENABLE". DISABLE is used to deactivate "logical function". FILTER is used to execute the logic contained in program.sub.-- object.sub.-- id prior to executing the normal "logical function" process. It the logic contained in program.sub.-- object.sub.-- id returns a non-zero SYS.sub.-- RETURN.sub.-- CODE< the normal "logical function" process will not execute, otherwise, it begins. ENABLE is used to set "logical function" to normal default process.

Continuing, in the SET.sub.-- FUNCTION syntax, "program.sub.-- object.sub.-- id" is the 13 byte

object.sub.-- id of the TBOL program, (conditional); and "state" is data to be passed to the "logical function" program. The data will reside in the P1 register when logic is executed, (optional).

SORT

The SORT verb is used to sort a range of variable fields into the sequence of the key contained in each field. Each variable field contains a record consisting of a fixed-length key field followed by a data field. The key field is the same length is each record. Operand 1 contains the name of the first field in the range of fields to be sorted; operand 2 contains the name of the last field. Operand 3 contains an integer number specifying the length of the key field contained in the beginning of each field. The fields in the range specified by operand 1 and operand 2 are sorted into the sequence of the key field.

The syntax for SORT is:

SORT field1,field2,key.sub.-- lath;

where "field1" contains the first field name of the range of fields to be sorter, and can be a data name, or system register P1-P8 only; "field2" contains the last field name of the range of fields to be sorted and can be a data name; or system register P1-P8 only; and "key.sub.-- lath" contains an integer number equal to the length of the key field contained in each field in the range. Key.sub.-- lath can be a data name, or system register P1-P8 only or a literal.

SOUND

The SOUND verb is used to produce a sound through the reception system speaker. A sound is produced of the pitch specified by operand 1, for the duration specified by operand 2, If operand 1 and operand 2 are not present, values from the most recently performed SOUND statement are used. The SOUND syntax is:

SOUND ›pitch,duration!;

where "pitch" is a numeric value in the range of 0 to 20,000 specifying the desired pitch of the sound. Pitch can be a data name, system register P1-P8, or a literal; and "duration" is a numeric value in the range of 0 to 65,535 specifying the desired duration of the sound in increments of .1 seconds. Duration can be a data name, or system register P1-P8 only or literal.

STRING

The STRING verb joins multiple character strings together with into one character string. Up to eight character strings, beginning with the character string contained in operand 1, are joined together sequentially. The resulting new character string replaces the contents of operand 1. The STRING syntax is:

STRING string1, ›,string . . . !;

where "string1" is empty, or contains the character string which will become the left-most portion of the new character string, and a data name, or a system register P1-P8 only; "string" is empty, or contains the character string to be joined behind the character strings in preceding operand, and can be a data name, or system register P1-P8 only or a literal.

SUBSTR

The SUBSTR verb is used to copy a substring of characters from a character string into a designated variable field. The character string containing the substring is in operand 1. Operand 3 contains an integer number equal to the position of the first character to be copied. Operand 4 contains an integer number equal to the number of characters to be copied. The specified substring is copied from the character string in operand 1 and replaces the contents of operand 2.

The syntax for SUBSTR is:

SUBSTR string,destination,strt.sub.-- pos,length;

where "string" contains a character string, and can be a data name or system register P1-P8 only, or a literal; "destination" is the name of the variable field to receive the substring copied from the string operand, and can be a data name, or system register P1-P8 only, "strt,pos" contains an integer number specifying the position of the first character to be copied into the destination operand, and can be a data name, or system register or a literal; and "length" contains an integer number specifying the number of characters to be copied into the destination operand, and can be a data name, or system register or a literal.

In accordance with this arrangement, the SUBSTR operation does not take place if: if the length operand is 0, or if the strt.sub.-- pos operand is greater than the length of the string operand.

SUBTRACT

The SUBTRACT verb subtracts one number from another. The number in operand 1 is subtracted from the number in operand 2. The number in operand 1 is unchanged. The number in operand 2 is replaced by the arithmetic difference between the two numbers. The syntax for SUBTRACT is:

SUBTRACT number1,number2;

where "number1" contains the number to be subtracted from number2, and can be a data name, or system register, or a literal; "number2" contains the second number. As noted, the contents of number2 are overlaid with the resulting difference. Number2 can be a data name, or system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the subtract operation. Fractions will be truncated after 13 decimal places, and whole numbers will not contain a decimal point. Further, negative results will contain a minus sign (-) in the left-most position.

TRANSFER

The TRANSFER verb transfers control to another TBOL program. Control however, does not return to the original program. Rather, program execution continues at the first statement in the program whose object ID is contained in operand 1. Up to eight parameters may be passed to the "called" program in operand 2-9. Control is transferred to the Reception System when the "called" program performs an EXIT.

The syntax for TRANSFER is:

TRANSFER object.sub.-- id ›,parameter . . . !;

where "object.sub.-- id" contains the object ID of a TBOL program, and can be a data name, or system register P1-P8 only, or a literal; "parameter" contains parameter data for the program whose object ID is contained in operand 1. The contents of the parameter operand 2 through 9, if present, are placed in parameter registers P1 through P8. The number of parameter operand is placed in P0. P0 through P8 are accessible to the "called" program. Parameter can be a data name, or system register, or a literal.

TRIGGER.sub.-- FUNCTION

The TRIGGER-FUNCTION verb is designed to execute a "logical function". Its syntax is:

TRIGGER.sub.-- FUNCTION function id;

where "function.sub.-- id" is the logical function" identifier. In accordance with the design of TRIGGER.FUNCTION, control may or may not be returned depending on the function requested.

UPPERCASE

The UPPERCASE verb converts lowercase alphabetic characters to uppercase alphabetic characters. Lowercase alphabetic characters (a-z) in the character string contained in operand 1 are converted to uppercase alphabetic characters (A-Z). The syntax for UPPERCASE is:

UPPERCASE string;

where "string" contains a character string, and can be a data name, or a system register P1-P8 only.

WAIT

The WAIT verb causes program control to be given to the Reception System for the number of seconds defined in the parameter head. Control is given to the Reception System for one "time slice" and then returned to the currently running program.

The WAIT syntax is simply:

```
WAIT;seconds
WHILE . . . THEN
```

The key word WHEN causes a single statement or a block of statements to be executed repetitively while a specified boolean expression is true. The key word WHILE is followed by a boolean expression. The boolean expression is always followed by a THEN clause. The boolean expression is evaluated to determine whether it is "true" or "false". If the expression is true, the THEN clause is executed and the expression is evaluated again. If the expression is false, program execution continues with the statement following the THEN clause.

The syntax for WHILE . . . THEN is:

WHILE boolean THEN clause;

where "boolean is a boolean expression, which can be a single relational expression, where a relational expression consists of two operands separated by a relational operator such as (=), (<>), (<), (>), (<=), or (=>), or two or more relational expressions separated by the key words AND or OR These relational expressions can be enclosed with parentheses, and then treated as a single relational expression separated from others with and or OR. Further, they are evaluated from left to right. Continuing, with the syntax for WHILE . . . THEN, "clause" can be either a single statement, a block of statements, where the block begins with the key word GO and ends with the END verb.

When character strings of unequal length are compared lexicographically, the longer string is truncated to the length of the shorter string before the comparison. If the shorter string compares "high", then the longer string is "lower". For example: When comparing "GG" to "H", "GG" is valued as less than "H". If the shorter string compares "low" or "equal", then the longer string is "high". For example: When comparing "TO" to "TOO", "TO" is less than "TOO".

In this regard, truncation is done outside of the operand, which the operand remaining the same length after the evaluation.

WRITE

WRITE is the verb used to write records to a file. The syntax for WRITE is:

WRITE filename, output.sub.-- area ›, key!;

where "filename" is the name of the file that the record is to be written to, and can be a field.sub.-- id, array.sub.-- id(subscript), partition.sub.-- external.sub.-- id, global.sub.-- external.sub.-- id, or a literal; "output .sub.-- area" is the name of the area from which the record will be created, and can be a field.sub.-- id, array.sub.-- id(subscript), partition.sub.-- external.sub.-- id or a global.sub.-- external.sub.-- id; and "length" specifies either the maximum number of characters to be read from an ASCII file, or the length of data to be read from a binary file. The file must have been previously opened as OUTPUT, APPEND, or I/O.

XOR

The XOR verb performs a logical XOR function on the bits of two data fields. The modula-two sum (exclusive OR) or the bits of operand 1 and operand 2 is placed in operand 2. Moving from left to right, the XOR is applied to the corresponding bits of each field, bit by bit, ending with the last bit of the shorter field. If the corresponding bits are 1 and 0, or 0 and 1, then the result bit is 1. If the corresponding bits are 1 and 1, or 0 and 0, then the result bit is 0. The data in operand 1 is left unchanged. The data in operand 2 is replaced by the result.

The syntax for XOR is:

XOR field1,field2;

where "field1" contains the first data field, and can be a data name, a system register I1-18 or P1-P8 only, or a literal; and "field2" contains the second data field. As in other logic operations, the contents of field2 are overlaid by the result of the XOR operation. Field2 can be a data name, system register I1-18 or P1-P8 only.

As will be appreciated by those skilled in the art, the XOR verb can be used to invert a bit. Further, any field XOR'ed with itself becomes all zeros, and, the sequence: XOR A.B; XOR B.A; XCR A.B; causes

the contents of A and B to be exchanged.

GLOBAL EXTERNAL SYSTEM VARIABLES

In accordance with the design of TBOL, names have been assigned to the TBOL system variables in the global external variable (GEV) data structure. The names of GEVs are assigned in DEFINE statements as described above and in the file TBOL.SYS. There are a total of 32,000 GEVs. The first 256 GEVs are reserved for the system, and the remaining 31,744 are assigned as application variables, and are application specific. Since system variables referenced by TBOL interpreter as global variables and are ASCII strings, a system variable table is constructed so that reception system native code can access them as binary integer. An adaptation of this table from the source code file "rs.backslash.rsk.backslash.c.backslash.sysvar.c", presented in more detail hereafter, is shown in Table 1.

TABLE 1

SYSTEM GLOBAL EXTERNAL VARIABLES System Variable Name	GEV # Description
Sys.sub.-- rtn.sub.-- code;	0001 API instr. return code.
Sys.sub.-- api.sub.-- event;	0002 API event: post, pre, init or sel
Sys.sub.-- logical.sub.-- key;	0003 Current logical key.
Sys.sub.-- last.sub.-- msg.sub.-- id;	0004 Last message id.
Sys.sub.-- tone.sub.-- pulse;	0005 Phone type pulse/tone.
Sys.sub.-- line.sub.-- status;	0006 Line connection status.
Sys.sub.-- keyword;	0007 Keyword flag.
Sys.sub.-- automatic.sub.-- uppercase;	0008 Auto uppercase.
Sys.sub.-- scroll.sub.-- increment;	0009 Scroll increment.
Sys.sub.-- current.sub.-- field;	0010 Current field.
Sys.sub.-- date;	0011 system date.
Sys.sub.-- time;	0012 system time.
Sys.sub.-- current.sub.-- page;	0013 current page.
Sys.sub.-- selected.sub.-- obj.sub.-- id	0014 sel object id.
Sys.sub.-- navigate.sub.-- obj.sub.-- id;	0015 nav object id.
Sys.sub.-- cursor.sub.-- row;	0016 cursor row position.
Sys.sub.-- cursor.sub.-- col;	0017 cursor col position.
Sys.sub.-- path;	0018 user personal path table.
Sys.sub.-- ttx.sub.-- phone;	0019 dial trintex phone #.
Sys.sub.-- total.sub.-- pages;	0020 total pages in page set.

```
Sys.sub.-- page.sub.-- number;
                    0021 curr. page of n pages.
Sys.sub.-- base.sub.-- obj.sub.-- id;
                    0022 curr. base page
                       object--id.
Sys.sub.-- window.sub.-- id;
                    0023 curr. window
                       object--id.
Sys.sub.-- path.sub.-- ptr;
                    0024 curr. path location.
Sys.sub.-- keywords;
                    0025 keyword list.
Sys.sub.-- current.sub.-- cursor.sub.-- pos;
                    0026 curr. cursor position.
Sys.sub.-- current.sub.-- background.sub.-- color;
                    0027 curr background color.
Sys.sub.-- current.sub.-- foreground.sub.-- color;
                    0028 curr foreground color.
Sys.sub.-- hardware.sub.-- status;
                    0029 nature of hard error.
Sys.sub.-- nocomm;  0030 send:don't send to S1.
Sys.sub.-- um.sub.-- dia.sub.-- header;
                    0031 header unsolicited msg.
Sys.sub.-- um.sub.-- message.sub.-- text;
                    0032 text unsolicited msg.
Sys.sub.-- ca.sub.-- error.sub.-- track.sub.-- info;
                    0033 error tracking data.
Sys.sub.-- assisant.sub.-- current.sub.-- info;
                    0034 curr. context info.
Sys.sub.-- screen.sub.-- data.sub.-- table;
                    0035 data table copy & file.
Sys.sub.-- ad.sub.-- list;
                    0036 pointer to AD list.
Sys.sub.-- current.sub.-- keyword;
                    0037 pointer to cur. keyword.
Sys.sub.-- previous.sub.-- keyword;
                    0038 pointer to prev.
                       keyword.
Sys.sub.-- guide;   0039 guide.
Sys.sub.-- previous.sub.-- menu;
                    0040 prev menu object--id.
Sys.sub.-- previous.sub.-- seen.sub.-- menu;
                    0041 prev seen menu obj--id.
Sys.sub.-- scan.sub.-- list;
                    0042 pointer to scan list.
Sys.sub.-- scan.sub.-- list.sub.-- pointer;
                    0043 user scan list pointer.
Sys.sub.-- path.sub.-- name;
                    0044 Pointer to path name.
Sys.sub.-- navigate.sub.-- keyword;
                    0045 Navigate to keyword.
Sys.sub.-- keyword.sub.-- table;
                    0046
Sys.sub.-- keyword.sub.-- disp;
                    0047
Sys.sub.-- keyword.sub.-- table.sub.-- entry.sub.-- length;
                    0048
Sys.sub.-- keyword.sub.-- length;
                    0049
Sys.sub.-- ext.sub.-- table;
```

```
                    0050
Sys.sub.-- data.sub.-- collect;
                    0051 Indicates Tracking
                       status.
Sys.sub.-- fm0.sub.-- txhdr;
                    0052 DIA message header
Sys.sub.-- fm0.sub.-- txdid;
                    0053
Sys.sub.-- fm0.sub.-- txrid;
                    0054
Sys.sub.-- fm4.sub.-- txhdr;
                    0055
Sys.sub.-- fm4.sub.-- txuseid;
                    0056
Sys.sub.-- fm4.sub.-- txcorid;
                    0057
Sys.sub.-- fm64.sub.-- txhdr;
                    0058
Sys.sub.-- fm64.sub.-- txdata;
                    0059
Sys.sub.-- fm0.sub.-- rxhdr;
                    0060
Sys.sub.-- fm4.sub.-- rxhdr;
                    0061
Sys.sub.-- fm4.sub.-- rxuseid;
                    0062
Sys.sub.-- fm4.sub.-- rxcorid;
                    0063
Sys.sub.-- fm64.sub.-- rxhdr;
                    0064
Sys.sub.-- fm64.sub.-- rxdata;
                    0065
Sys.sub.-- surrogate;
                    0066 md
Sys.sub.-- leave;   0067 md
Sys.sub.-- return;  0068 md
Sys.sub.-- int.sub.-- regs;
                    0069 md, area for int save
                       stack
Sys.sub.-- ttx.sub.-- help.sub.-- id;
                    0070 md, id of sys help
                    window/
Sys.sub.-- selector.sub.-- data;
                    0071 md
Sys.sub.-- selector.sub.-- path;
                    0072 md
Sys.sub.-- logical.sub.-- event;
                    0073 am
Sys.sub.-- user.sub.-- id;
                    0074 mg/
Sys.sub.-- help.sub.-- appl;
                    0075 md/
Sys.sub.-- help.sub.-- hub.sub.-- appl.sub.-- pto;
                    0076 md/
Sys.sub.-- access.sub.-- key.sub.-- obj.sub.-- id;
                    0077 lw, bi/
Sys.sub.-- word.sub.-- wrap=1;
                    0078
Sys.sub.-- messaging.sub.-- status;
                    0079
```

```
Sys.sub.-- version; 0080
Sys.sub.-- leader.sub.-- ad.sub.-- id;
                    0081
Sys.sub.-- baud.sub.-- rate;
                    0082
Sys.sub.-- com.sub.-- port;
                    0083
Sys.sub.-- obj.sub.-- header;
                    0084
Sys.sub.-- session.sub.-- status;
                    0085
Systbl sys.sub.-- var.sub.-- table >! =
                    NA Define system var table.
&Sys.sub.-- rtn.sub.-- code,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- api.sub.-- event,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- logical.sub.-- key,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- last.sub.-- msg.sub.-- id,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- tone.sub.-- pulse,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- line.sub.-- status,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- keyword,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- automatic.sub.-- uppercase,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- scroll.sub.-- increment,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- current.sub.-- field,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE
&(unsigned int)Sys.sub.-- date,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- time,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- current.sub.-- page,
                    0, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- selected.sub.-- obj.sub.-- id,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- navigate.sub.-- obj.sub.-- id,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- cursor.sub.-- row,
                    0, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- cursor.sub.-- col,
                    0, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- path,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- ttx.sub.-- phone,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- total.sub.-- pages,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- page.sub.-- number,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- base.sub.-- obj.sub.-- id,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- window.sub.-- id
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- path.sub.-- ptr,
```

```
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- keywords,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- current.sub.-- cursor.sub.-- pos,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- current.sub.-- background.sub.-- color,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- current.sub.-- foreground.sub.-- color,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- hardware.sub.-- status,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE
&Sys.sub.-- nocomm, INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- um.sub.-- dia.sub.-- header,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- um.sub.-- message.sub.-- text,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)     0, SYS.sub.-- STR.sub.-- TYPE,
 Sys.sub.-- ca.sub.-- error track.sub.-- info,
&(unsigned int)     0, SYS.sub.-- STR.sub.-- TYPE,
 Sys.sub.-- assisant.sub.-- current.sub.-- info
&(unsigned int)Sys.sub.-- screen.sub.-- data.sub.-- table,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- ad.sub.-- list,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- current.sub.-- keyword,

                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- previous.sub.-- keyword,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- guide,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- previous.sub.-- menu,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)     0, SYS.sub.-- STR.sub.-- TYPE
 Sys.sub.-- previous.sub.-- seen.sub.-- menu
&(unsigned int)Sys.sub.-- scan.sub.-- list,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- scan.sub.-- list.sub.-- pointer,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- path.sub.-- name,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- navigate.sub.-- keyword,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- keyword.sub.-- table,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- keyword.sub.-- disp,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- keyword.sub.-- table.sub.-- entry.sub.-- length,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- keyword.sub.-- length,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- exl.sub.-- table,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&()Sys.sub.-- data.sub.-- collect,
&(unsigned int)Sys.sub.-- fm0.sub.-- txhdr,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm0.sub.-- txdid,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm0.sub.-- txrid,
                    0, SYS.sub.-- STR.sub.-- TYPE,
```

```
&(unsigned int)Sys.sub.-- fm4.sub.-- txhdr,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm4.sub.-- txuseid,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm4.sub.-- txcorid,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm64.sub.-- txhdr,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm64.sub.-- txdata,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm0.sub.-- rxhdr,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm4.sub.-- rxhdr,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm4.sub.-- rxuseid,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm4.sub.-- rxcorid,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm64.sub.-- rxhdr,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- fm64.sub.-- rxdata,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- surrogate,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- leave,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- return,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- in.sub.-- regs,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- ttx.sub.-- help.sub.-- id,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- selector.sub.-- data,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- selector.sub.-- path,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- logical.sub.-- event,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE
&(unsigned int)Sys.sub.-- user.sub.-- id,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- help.sub.-- appl,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)     0, SYS.sub.-- STR.sub.-- TYPE,
 Sys.sub.-- help.sub.-- hub.sub.-- appl.sub.-- pto,
&(unsigned int)     0, SYS.sub.-- STR.sub.-- TYPE,
 Sys.sub.-- access.sub.-- key.sub.-- obj.sub.-- id,
&Sys.sub.-- word.sub.-- wrap,
                    1, SYS.sub.-- INT.sub.-- TYPE,
&(unsigned int)Sys.sub.-- messaging.sub.-- status,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- version,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&(unsigned int)Sys.sub.-- leader.sub.-- ad.sub.-- id,
                    0, SYS.sub.-- STR.sub.-- TYPE,
&Sys.sub.-- baud.sub.-- rate,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- com.sub.-- port,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
&Sys.sub.-- obj.sub.-- header,
                    0, SYS.sub.-- STR.sub.-- TYPE,/RDC
```

```
&Sys.sub.-- session.sub.-- status,
                    INTLEN, SYS.sub.-- INT.sub.-- TYPE,
```

TABLE 2

TBOL VERBS BY FUNCTIONAL CATEGORY

DATA PROCESSING		
ADD	LOOKUP	SAVE
AND	MAKE.sub.-- FORMAT	
		SORT
CLEAR	MOVE	STRING
DIVIDE	MULTIPLY	SUBSTR
EDIT	OR	SUBTRACT
FILL	POP	UPPERCASE
FORMAT	PUSH	XOR
INSTR	RELEASE	
LENGTH	RESTORE	
PROGRAM FLOW		
CLOSE.sub.-- WINDOW		
	LINK	TRANSFER
EXIT	NAVIGATE	TRIGGER.sub.-- FUNCTION
GOTO	OPEN.sub.-- WINDOW	
		WAIT
GOTO.sub.-- DEPENDING.sub.-- ON		
	RETURN	WHILE . . . THEN
IF . . . THEN . . . ELSE		
	SET.sub.-- FUNCTION	
		SYNC.sub.-- RELEASE
COMMUNICATIONS		
CONNECT	RECEIVE	
DELETE	SEND	
DISCONNECT		
FILE MANAGEMENT		
CLOSE	OPEN	READ
NOTE	POINT	WRITE
SCREEN MANAGEMENT		
DEFINE.sub.-- FIELD		
	SOUND	
SET.sub.-- ATTRIBUTE		
	REFRESH	
SET.sub.-- CURSOR		
OBJECT MANAGEMENT		
FETCH		
PROGRAM STRUCTURE		
DO . . . END	END.sub.-- PROC	

RECEPTION SYSTEM OPERATION

RS 400 of computer system network 10 uses software called native code modules (to be described below) to enable the user to select options and functions presented on the monitor screen 414 of personal computer 405, to execute partitioned applications and to process user created events, enabling the partitioned application to interact with interactive system 10. Through this interaction, the user is able to input data into fields provided as part of the display, or may individually select choices causing a standard or personalized page to be built (as explained below for display on the monitor of personal

computer 405. Such inputs will cause RS 400 to interpret events and trigger pre-processors or post-processors, retrieve specified objects, communicate with system components, control user options, cause the display of advertising on a page, open or close window partitions to provide additional navigation possibilities, and collect and report data about events, including certain types of objects processed. For example, the user may select a particular option, such as opening or closing window partition 275, which is present on the monitor and follow the selection with a completion key stroke, such as ENTER. When the completion keystroke is made, the selection is translated into a logical event that triggers the execution of a post-processor, (i.e., a partitioned application program object) to process the contents of the field.

Functions supporting the user-partitioned application interface can be performed using the command bar 290, or its equivalent using pull down windows or an overlapping cascade of windows. These functions can be implemented as part of the RS native functions or can be treated as another partition(s) defined for every page for which an appropriate set of supporting objects exist and remain resident at RS 400. If the functions are part of RS 400, they can be altered or extended by verbs defined in the RS virtual machine that permit the execution of program objects to be triggered when certain functions are called, providing maximum flexibility.

To explain the functions the use of a command bar is assumed. Command bar 290 is shown in FIGS. 3a and 3b and includes a NEXT command 291, a BACK command 292, a PATH command 293, a MENU command 294, an ACTION command 295, a JUMP command 296, a HELP command 297, and an EXIT command 298.

NEXT command 291 causes the next page in the current page set to be built. If the last page of a page set has already been reached, NEXT command 291 is disabled by RS 400, avoiding the presentation of an invalid option.

BACK command 292 causes the previous page of the current page set to be built. If the present page is the first in the page set, BACK command 292 is disabled, since it is not a valid option.

A filter program can be attached to both the NEXT or BACK functions to modify their implicit sequential nature based upon the value of the occurrence in the object set id.

PATH command 293 causes the next page to be built and displayed from a list of pages that the user has entered, starting from the first entry for every new session.

MENU command 294 causes the page presenting the previous set of choices to be rebuilt.

ACTION command 295 initiates an application dependent operation such as causing a new application partition to be interpreted, a window partition 275 to be opened and enables the user to input any information required which may result in a transaction or selection of another window or page.

JUMP command 296 causes window partition 275 to be opened, allowing the user to input a keyword or to specify one from an index that may be selected for display.

HELP command 297 causes a new application partition to be interpreted such as a HELP window pertaining to where the cursor is positioned to be displayed in order to assist the user regarding the present page, a particular partition, or a field in a page element.

EXIT command 298 causes a LOGOFF page template object (PTO) to be built, and a page logoff sequence to be presented at RS 400 monitor screen 414.

NAVIGATION INTERFACE

Continuing, as a further feature, the method aspect of the invention includes an improved procedure for searching and retrieving applications from the store of applications distributed throughout network 10; e.g., server 205, cache/concentrator 302 and RS 400. More specifically, the procedure features use of pre-created search tables which represent subsets of the information on the network arranged with reference to the page template objects (PTO) and object-ids of the available applications so that in accordance with the procedure, the relevant tables and associated objects can be provided to and searched at the requesting RS 400 without need to search the entire store of applications on the network. As will be appreciated, this reduces the demand on the server 205 for locating and retrieving applications for display at monitor 412.

In conventional time-sharing networks that support large conventional databases, the host receives user requests for data records; locates them; and transmits them back to the users. Accordingly, the host is obliged to undertake the data processing necessary to isolate and supply the requested information. And, as noted earlier, where large numbers of users are to be served, the many user requests can bottleneck at the host, taxing resources and leading to response slowdown.

Further, users have experienced difficulty in searching data bases maintained on conventional time-sharing networks. For example, difficulties have resulted from the complex and varied way previously known database suppliers have organized and presented their information. Particularly, some database providers require searching be done only in selected fields of the data base, thus requiring the user to be fully familiar with the record structure. Others have organized their databases on hierarchical structures which require the user understand the way the records are grouped. Still further, yet other database suppliers rely upon keyword indices to facilitate searching of their records, thus requiring the user to be knowledgeable regarding the particular keywords used by the database provider.

The method aspect of the present invention, however, serves to avoid such difficulties. In the preferred embodiment, the invention includes procedures for creating preliminary searches which represent subsets of the network applications users are believed likely to investigate. Particularly, in accordance with these procedures, for the active applications available on network 10, a library of tables is prepared, and maintained within each of which a plurality of so called "keywords" are provided that are correlated with page template objects and object-ids of the entry screen (typically the first screen) for the respective application. In the preferred embodiment, approximately 1,000 tables are used, each having approximately 10 to 20 keywords arranged in alphabetical order to abstract the applications on the network. Further, the object-id for each table is associated with a code in the form of a character string mnemonic which is arranged in a set of alphabetically sequenced mnemonics termed the sequence set so that on entry of a character string at an RS 400, the object-id for the relevant keyword table can be obtained from the sequence set. Once the table object-id is identified, the keyword table corresponding to the desired subset of the objects and associated applications can then be obtained from network 10. Subsequently the table can be presented to the user's RS 400, where the RS 400 can provide the data processing required to present the potentially relevant keywords, objects and associated applications to the user for further review and determination as to whether more searching is required. As will be appreciated, this procedure reduces demand on server 205 and thereby permits it to be less complex and costly, and further, reduces the likelihood of host overtaxing that may cause network response slowdown.

As a further feature of this procedure, the library of keywords and their associated PTOs and objects may be generated by a plurality of operations which appear at the user's screen as different search techniques. This permits the user to select a search technique he is most comfortable with, thus

expediting his inquiry.

More particularly, in accordance with the invention, the user is allowed to invoke the procedure by calling up a variety of operations The various operations have different names and seemingly present different search strategies. Specifically, the user may invoke the procedure by initiating a "Jump" command at RS 400. Thereafter, in connection with the Jump operation, the user when prompted, may enter a word of the user's choosing at monitor screen 414 relating to the matter he is interested in locating; i.e., a subject matter search of the network applications. Additionally, the users may invoke the procedure by alternatively calling up an operation termed "Index" with selection of the Index command. When selected, the Index command presents the user with an alphabetical listing of keywords from the tables noted above which the user can select from i.e., an alphabetical search of the network applications. Further, the user may evoke the procedure by initiating an operation termed "Guide." By selecting the Guide command, the user is provided with a series of graphic displays that presents a physical description of the network applications; e.g., department floor plan for a store the user may be electronically shopping in. Still further, the user may invoke the procedures by initiating an operation termed "Directory." By selecting the Directory command, the user is presented with the applications available on the network as a series of hierarchical menus which present the content of the network information in commonly understood categories. Finally, the user may invoke the procedure by selecting the "Path" command, which accesses a list of keywords the user has previously selected; i.e., a personally tailored form of the Index command described above. As described hereafter, Path further includes a Viewpath operation which permits the user to visually access and manage the Path list of keywords. In preferred form, where the user has not selected a list of personalized keywords,a default set is provided which includes a predetermined list and associated applications deemed by network 10 as likely to be of interest to the user.

In accordance with the invention, this ability to convert these apparently different search strategies in a single procedure for accessing pre-created library tables is accomplished by translating the procedural elements of the different search techniques into a single set of procedures that will produce a mnemonic; i.e., code word, which can first be searched at the sequence set, described above to identify the object-id for the appropriate library table and, thereafter, enable access of the appropriate table to permit selection of the desired keyword and associated PTO and object-ids. That is to say, the reception system native code simply relates the user-entered character string, alphabetical range, category, or list item of respectively, "Jump", "Index", "Directory", or "Path" to the table codes through the sequence set, so that the appropriate table can be provided to the reception system and application keyword selected. Thus, while the search techniques may appear different to the user, and in fact accommodate the user's preferences and sophistication level, they nonetheless invoke the same efficient procedure of relying upon pre-created searches which identify related application PTOs and object-ids so that the table and objects may be collected and presented at the user's RS 400 where they can be processed, thereby relieving server 205.

In preferred form, however, in order to enhance presentation speed the Guide operation is specially configured. Rather than relating the keyword mnemonic to a sequence set to identify the table object-id and range of keywords corresponding to the entry PTO and associated object-ids, the Guide operation presents a series of overlapping windows that physically describe the "store" in which shopping is being conducted or the "building" from which information is being provided. The successive windows increase in degree of detail, with the final window presenting a listing of relevant keywords. Further, the PTO and object-ids for the application entry screen are directly related to the graphic presentation of the keywords. This eliminates the need to provide variable fields in the windows for each of the keywords and enables the entry screen to be correlated directly with the window graphic. As will be appreciated, this reduces the number of objects that would otherwise be required to be staged at RS 400 to support pretention of the keyword listing at monitor screen 414, and thus speeds network response.

A more detailed understanding of the procedure may be had upon a reading of the following description and review of accompanying FIGS. 2, 3a and particularly FIG. 11 which presents a flow diagram for the Jump sequence of the search procedure.

To select a particular partitioned application from among thousands of such applications residing either at the RS 400 or within delivery system 20, the present invention avoids the need for a user to know or understand, prior to a search, the organization of such partitioned applications and the query techniques necessary to access them. This is accomplished using a collection of related commands, as described below.

The Jump command 296 as seen in FIG. 3a, can be selected, by the user from command bar 290. When Jump command 296 is selected, a window partition 275 is opened. In window 275, the user is presented and may select from a variety of displayed options that include among others, the Directory command, the Index command, and the Guide command, which when selected, have the effect noted above. Additionally, the user can select a command termed Viewpath which will presents the keywords that currently make up the list of keywords associated with the user's Path command, and from which list the user can select a desired keyword. Still further, and with reference FIG. 11, which shows the sequence where a user offers a term to identify a subject of interest, the user may enter a keyword at display field 270 within window partition 275 as a "best guess" of the mnemonic character string that is assigned to a partitioned application the user desires (e.g., the user may input such english words as "news," "pet food," "games," etcetera). Where the user enters a character string it is displayed in field 270, and then searched by RS 400 native code (discussed below) against the sequence sets above noted to identify the object-id for the appropriate table of keywords (not shown) that RS 400 may request from host 205. While as noted above, a table may include 10 to 20 keywords, in the preferred embodiment, for the sake of speed and convenience, a typical keyword table includes approximately 12 keywords.

If the string entered by the user matches a keyword existing on one of the keyword tables, and is thus associated with a specific PTO, RS 400 fetches and displays associated objects of the partitioned applications and builds the entry page in accordance with the page composition dictated by the target PTO.

If the string entered by the user does not match a specific keyword, RS 400 presents the user with the option of displaying the table of keywords approximating the specific keyword. The approximate keywords are presented as initialized, cursorable selector fields of the type provided in connection with a Index command. The user may then move the cursor to the nearest approximation of the mnemonic he originally selected, and trigger navigation to the PTO associated with that keyword, navigation being as described hereafter in connection with the RS 400 native code.

If, after selecting the Jump command, the user selects the Index command, RS 400 will retrieve the keyword table residing at RS 400, and will again build a page with initialized, cursorable fields of keywords. The table fetched upon invoking the Index command will be comprised of alphabetic keywords that occur within the range of the keywords associated with the page template object (PTO) from which the user invoked the Indexcommand. As discussed above, the user may select to navigate to any of this range of PTOs by selecting the relevant keyword from the display. Alternatively, the user can, thereafter, select another range of alphabetical keywords by entering an appropriate character string in a screen field provided or move forward or backward in the collection by selecting the corresponding option.

By selecting the Directory command, RS 400 can be caused to fetch a table of keywords, grouped by categories, to which the PTO of the current partitioned application (as specified by the object set field

630 of the current PEO) belongs. Particularly, by selecting the Directory command, RS 400, is causes to displays a series of screens each of which contains alphabetically arranged general subject categories from which the user may select. Following selection of a category, a series of keywords associated with the specified category are displayed in further screens together with descriptive statements about the application associated with the keywords. Thereafter, the user can, in the manner previously discussed with regard to the Index command, select from and navigate to the PTOs of keywords which are related to the present page set by subject.

The Guide command provides a navigation method related to a hierarchical organization of applications provided on network 10, and are described by a series of sequentially presented overlaying windows of a type known in the art, each of which presents an increasing degree of detail for a particular subject area, terminating in a final window that gives keywords associated with the relevant applications. The Guide command makes use of the keyword segment which describes the location of the PTO in a hierarchy (referred to, in the preferred embodiment, as the "BFD," or Building-Floor-Department) as well as an associated keyword character string. The BFD describes the set of menus that are to be displayed on the screen as the sequence of pop-up windows. The Guide command may be invoked by requesting it from the Jump window described above, or by selecting the Menu command on Command Bar 290. As noted above, in the case of the Guide command, the PTO and object-ids for the application entry screen are directly associated with the graphic of the keyword presented in the final pop-up window. This enables direct access of the application entry screen without need to access the sequence set and keyword table, and thus, reduces response time by reducing the number of objects that must be processed at RS 400.

Activation of the Path command accesses the user's list of pre-selected keywords without their display, and permits the user to step through the list viewing the respective applications by repeatedly invoking the Path command. As will be appreciated, the user can set a priority for selecting keywords and viewing their associated applications by virtue of where on the list the user places the keywords. More specifically, if the user has several application of particular interest; e.g, news, weather, etc., the user can place them at the top of the list, and quickly step through them with the Path command. Further, the user can view and randomly access the keywords of his list with the Viewpath operation noted above. On activation of Viewpath, the user's Path keywords are displayed and the user can cursor through them in a conventional manner to select a desired adjusting their relative position. This is readily accomplished by entering the amendments to the list presented at the screen 414 with a series of amendment options presented in a conventional fashion with the list. As noted, the list may be personally selected by the user in the manner described, or created as a default by network 10.

Collectively, the Jump command, Index command, Directory command, Guide command, and Path command as described enable the user to quickly and easily ascertain the "location" of either the partitioned application presently displayed or the "location" of a desired partitioned application. "Location," as used in reference t the preferred embodiment of the invention, means the specific relationships that a particular partitioned application bears to other such applications, and the method for selecting particular partitioned applications from such relationships. The techniques for querying a database of objects, embodied in the present invention, is an advance over the prior art, insofar as no foreknowledge of either database structure or query technique or syntax is necessary, the structure and search techniques being made manifest to the user in the course of use of the commands.

RS APPLICATION PROTOCOL

RS protocol defines the way the RS supports user application conversation (input and output) and the way RS 400 processes a partitioned application. Partitioned applications are constructed knowing that this protocol will be supported unless modified by the application. The protocol is illustrated FIG. 6. The boxes in FIG. 6 identify processing states that the RS 400 passes through and the arrows indicate the

transitions permitted between the various states and are annotated with the reason for the transition.

The various states are: (A) Initialize RS, (B) Process Objects, (C) Interpretively Execute Preprocessors, (D) Wait for Event, (E) Process Event, and (F) Interpretively Execute Function Extension and/or Post-processors.

The transitions between states are: (1a) Logon Page Template Object Identification (PTO-id) (1b) Object Identification, (2) Trigger Program Object identification (PO-id)& return (3) Page Partition Template (PPT) or Window Stack Processing complete, (4) Event Occurrence, and (5) Trigger PO-id and Return.

Transition (1a) from Initialize RS (A) to Process Objects (B) occurs when an initialization routine passes the object-id of the logon PTO to object interpreter 435, when the service is first invoked. Transition (1b) from Process Event (E) to Process Objects (B) occurs whenever a navigation event causes a new page template object identification (PTO-id) to be passed to object interpreter 435; or when a open window event (verb or function key) occurs passing a window object-id to the object interpreter 435; or a close window event (verb or function key) occurs causing the current top-most window to be closed.

While in the process object state, object interpreter 435 will request any objects that are identified by external references in call segments. Objects are processed byparsing and interpreting the object and its segments according to the specific object architecture. As object interpreter 435 processes objects, it builds a linked list structure called a page processing table (PPT), shown in FIG. 10, to reflect the structure of the page, each page partition, Page Element Objects (PEOs) required, program objects (POs) required and each window object (WO) that could be called. Object interpreter 435 requests all objects required to build a page except objects that could be called as the result of some event, such as a HELP window object.

Transition (2) from Process Objects (B) to Interpretively Execute Pre-processors (C) occurs when the object interpreter 435 determines that a pre-processor is to be triggered. Object processor 436 then passes the object-id of the program object to the TBOL interpreter 438. TBOL interpreter 438 uses the RS virtual machine to interpretively execute the program object. The PO can represent either a selector or an initializer. When execution is complete, a transition automatically occurs back to Process Objects (B).

Selectors are used to dynamically link and load other objects such as PEOs or other PDOs based upon parameters that they are passed when they are called. Such parameters are specified in call segments or selector segments. This feature enables RS 400 to conditionally deliver information to the user base upon predetermined parameters, such as his personal demographics or locale. For example, the parameters specified may be the transaction codes required to retrieve the user's age, sex, and personal interest codes from records contained in user profiles stored at the switch/file server layer 200.

Initializers are used to set up the application processing environment for a partitioned application and determine what events RS 400 may respond to and what the action will be.

Transition (3) from Process Objects (B) to Wait for Event (D)occurs when object interpreter 435 is finished processing objects associated with the page currently being built or opening or closing a window on a page. In the Wait for Event state (D), an input manager, which in the preferred form shown includes keyboard manager 434 seen in FIG. 8, accepts user inputs. All keystrokes are mapped from their physical codes to logical keystrokes by the Keyboard Manager 434, representing keystrokes recognized by the RS virtual machine.

When the cursor is located in a field of a page element, keystrokes are mapped to the field and the partitioned external variable (PEV) specified in the page element object (PEO) field definition segment by the cooperative action of keyboard manager, 434 and display manager 461. Certain inputs, such as RETURN or mouse clicks in particular fields, are mapped to logical events by keyboard manager 434, which are called completion (or commit) events. Completion events signify the completion of some selection or specification process associated with the partitioned application and trigger a partition level and/or page level post-processor to process the "action" parameters associated with the user's selection and commit event.

Such parameters are associated with each possible choice or input, and are set up by the earlier interpretive execution of an initializer pre-processor in state (C). Parameters usually specify actions to perform a calculation such as the balance due on an order of several items with various prices using sales tax for the user's location, navigate to PTO-id, open window WO-id or close window. Actions parameters that involve the specification of a page or window object will result in transition (1b) to the Process Objects (B) state after the post-processor is invoked as explained below.

Function keys are used to specify one or more functions which are called when the user strikes these keys. Function keys can include the occurrence of logical events, as explained above. Additionally, certain functions may be "filtered", that is, extended or altered by SET.sub.-- FUNCTION or TRIGGER.sub.-- FUNCTION verbs recognized by the RS virtual machine. Function keys cause the PO specified as a parameter of the verb to be interpretively executed whenever that function is called. Applications use this technique to modify or extend the functions provided by the RS.

Transition (5) from Process Event (E) to Interpretively Execute Pre-processors (F) occurs when Process Event State determines that a post-processor or function extension PDO is to be triggered. The id of the program object is then passed to the TBOL interpreter 438. The TBOL interpreter 438 uses the RS virtual machine to interpretively execute the PO. When execution is complete a transition automatically occurs back to Process Event (E).

RECEPTION SYSTEM SOFTWARE

The reception system 400 software is the interface between the user of personal computer 405 and interactive network 10. The object of reception system software is to minimize mainframe processing, minimize transmission across the network, and support application extendibility and portability.

RS 400 software is composed of several layers, as shown in FIG. 7. It includes external software 451, which is composed of elements well known to the art such as device drivers, the native operating systems; i.e., MS-DOS, machine-specific assembler functions (in the preferred embodiment; e.g., CRC error checking), and "C" runtime library functions; native software 420; and partitioned applications 410.

Again with reference to FIG. 7, native software 420 is compiled from the "C" language into a target machine-specific executable, and is composed of two components: the service software 430 and the operating environment 450. Operating environment 450 is comprised of the Logical Operating System 432, or LOS; and a multitasker 433. Service software 430 provides functions specific to providing interaction between the user and interactive network 10, while the operating environment 450 provides pseudo multitasking and access to local physical resources in support of service software 430. Both layers of native software 420 contain kernel, or device independent functions 430 and 432, and machine-specific or device dependent functions 433. All device dependencies are in code resident at RS 400, and are limited to implementing only those functions that are not common across machine types, to enable interactive network 10 to provide a single data stream to all makes of personal computer which are of

the IBM or IBM compatible type. Source code for the native software 420 is included in parent application Ser. No. 388,156 now issued as U.S. patent , the contents of which patent are incorporated herein by reference. Those interested in a more detailed description of the reception system software may refer to the source code provided in the referenced patent.

Service software 430 is comprised of modules, which are device-independent software components that together obtain, interpret and store partitioned applications existing as a collection of objects. The functions performed by, and the relationship between, the service software 430 module is shown in FIG. 8 and discussed further below.

Through facilities provided by LOS 432 and multitasker 433, here called collectively operating environment 450, device-independent multitasking and access to local machine resources such as multitasking, timers, buffer management, dynamic memory management, file storage and access, keyboard and mouse input, and printer output are provided. The operating environment 450 manages communication and synchronization of service software 430, by supporting a request/response protocol and managing the interface between the native software 420 and external software 437.

Applications software layer 410 consists of programs and data written in an interpretive language, "TRINTEX Basic Object Language" or "TBOL," described above. TBOL was written specifically for use in RS 400 and interactive network 10 to facilitate videotext-specific commands and achieve machine-independent compiling. TBOL is constructed as objects, which in interaction with one another comprise partitioned applications.

RS native software 420 provides a virtual machine interface for partitioned applications, such that all objects comprising partitioned applications "see" the same machine. RS native software provides support for the following functions: (1) keyboard and mouse input; (2) text and graphics display; (3) application interpretation; (4) application database management; (5) local application storage; (6) network and link level communications; (7) user activity data collection; and (8) advertising management.

With reference to FIG. 8, service software 430 is comprised of the following modules: startup (not shown); keyboard manger 434; object interpreter 435; TBOL interpreter 438; object storage facility 439; display manager 461; data collection manager 441; ad manager 442; object/communications manager interface 443; link communications manager 444; and fatal error manager 469. Each of these modules has responsibility for managing a different aspect of RS 400.

Startup reads RS 400 customization options into RAM, including modem, device driver and telephone number options, from the file CONFIG.SM.Startup invokes all RS 400 component startup functions, including navigation to the first page, a logon screen display containing fields initialized to accept the user's id and password. Since Startup is invoked only at initialization, for simplicity, it has not been shown in FIG. 8.

The principal function of keyboard manger 434 is to translate personal computer dependent physical input into a consistent set of logical keys and to invoke processors associated with these keys. Depending on the LOS key, and the associated function attached to it, navigation, opening of windows, and initiation of filter or post-processor TBOL programs may occur as the result input events handled by the keyboard manger 434. In addition, keyboard manger 434 determines inter and intra field cursor movement, and coordinates the display of field text and cursor entered by the user with display manager 461, and sends information regarding such inputs to data collection manager 441.

Object interpreter 435 is responsible for building and recursively processing a table called the "Page

Processing Table," or PPT. Object interpreter 435 also manages the opening and closing of windows at the current page. Object interpreter 435 is implemented as two sub-components: the object processor 436 and object scanner 437.

Object processor 436 provides an interface to keyboard manger 434 for navigation to new pages, and for opening and closing windows in the current page. Object processor 436 makes a request to object storage facility 439 for a page template object (PTO) or window object (WO), as requested by keyboard manger 434, and for objects and their segments which comprise the PTO or WO returned by object storage facility 439 to object processor 436. Based on the particular segments comprising the object(s) making up the new PTO or WO, object processor 436 builds or adds to the page processing table (PPT), which is an internal, linked list, global data structure reflecting the structure of the page or page format object (PFO), each page partition or page element object (PEO) and program objects (POs) required and each window object(WO) that could be called. Objects are processed by parsing and interpreting each object and its segment(s) according to their particular structure as formalized in the data object architecture (DOA). While in the process object state, (state "B" of FIG. 6), object processor 436 will request any objects specified by the PTO that are identified by external references in call segments (e.g. field level program call 518, page element selector call 524, page format call 526 program call 532, page element call 522 segments) of such objects, and will, through a request to TBOL interpreter 438, fire initializers and selectors contained in program data segments of all PTO constituent program objects, at the page, element, and field levels. Object processor 436 requests all objects required to build a page, except objects that could only be called as the result of some event external to the current partitioned application, such as a HELP window object. When in the course of building or adding to the PPT and opening/closing WOs, object processor encounters a call to an "ADSLOT" object id, the next advertising object id at ad manager 442 is fetched, and the identified advertising object is retrieved either locally, if available, or otherwise from the network, so that the presentation data for the advertising can be sent to display manager 461 along with the rest of the presentation data for the other objects to enable display to the user. Object processor 436 also passes to data collection manager 441 all object ids that were requested and object ids that were viewed. Upon completion of page or window processing, object processor 436 enters the wait for event state, and control is returned to keyboard manger 434.

The second component of object interpreter 435, object scanner 437, provides a file-like interface, shared with object storage facility 439, to objects currently in use at RS 400, to enable object processor 436 to maintain and update the PPT. Through facilities provided by object scanner 437, object processor recursively constructs a page or window in the requested or current partitioned application, respectively.

Object storage facility 439 provides an interface through which object interpreter 435 and TBOL interpreter 438 either synchronously request (using the TBOL verb operator "GET") objects without which processing in either module cannot continue, or asynchronously request (using the TBOL verb operator "FETCH") objects in anticipation of later use. Object storage facility 439 returns the requested objects to the requesting module once retrieved from either local store 440 or interactive network 10. Through control structures shared with the object scanner 437, object storage facility determines whether the requested object resides locally, and if not, makes an attempt to obtain it from interactive network 10 through interaction with link communications manager 444 via object/communications manager interface 443.

When objects are requested from object storage facility 439, only the latest version of the object will be provided to guarantee currency of information to the user. Object storage facility 439 assures currency by requesting version verification from network 10 for those objects which are available locally and by requesting objects which are not locally available from delivery system 20 where currency is maintained.

Version verification increases response time. Therefore, not all objects locally available are version checked each time they are requested. Typically, objects are checked only the first time they are requested during a user session. However, there are occasions, as for example in the case of objects relating to news applications, where currency is always checked to assure integrity of the information.

The frequency with which the currency of objects is checked depends on factors such as the frequency of updating of the objects. For example, objects that are designated as ultrastable in a storage control parameter in the header of the object are never version checked unless a special version control object sent to the RS as part of logon indicates that all such objects must be version checked. Object storage facility 439 marks all object entries with such a stability category in all directories indicating that they must be version checked the next time they are requested.

Object storage facility 439 manages objects locally in local store 440, comprised of a cache (segmented between available RAM and a fixed size disk file), and stage (fixed size disk file). Ram and disk cached objects are retained only during user sessions, while objects stored in the stage file are retained between sessions. The storage control field, located in the header portion of an object, described more fully hereafter as the object "storage candidacy", indicates whether the object is stageable, cacheable or trashable.

Stageable objects must not be subject to frequent change or update. They are retained between user sessions on the system, provided storage space is available and the object has not discarded by a least-recently-used (LRU) algorithm of a conventional type; e.g., see Operating System Theory, by Coffman, Jr. and Denning, Prentice Hall Publishers, New York, 1973, which in accordance with the invention, operates in combination with the storage candidacy value to determine the object storage priority, thus rendering the stage self-configuring as described more fully hereafter. Over time, the self-configuring stage will have the effect of retaining within local disk storage those objects which the user has accessed most often. The objects retained locally are thus optimized to each individual user's usage of the applicators in the system. Response time to such objects is optimized since they need not be retrieved from the interactive computer system.

Cacheable objects can be retained during the current user session, but cannot be retained between sessions. These objects usually have a moderate update frequency. Object storage facility 439 retains objects in the cache according to the LRU storage retention algorithm. Object storage facility 439 uses the LRU algorithm to ensure that objects that are least frequently used forfeit their storage to objects that are more frequently used.

Trashable objects can be retained only while the user is in the context of the partitioned application in which the object was requested. Trashable objects usually have a very high update frequency and must not be retained to ensure that the user has access to the most current data.

More particularly and, as noted above, in order to render a public informational and transactional network of the type considered here attractive, the network must be both economical to use and fast. That is to say, the network must supply information and transactional support to the user at minimal costs and with a minimal response time. In accordance with the present invention, these objectives are sought to be achieved by locating as many information and transactional support objects which the user is likely to request, as close to the user as possible; i.e., primarily at the user's RS 400 and secondarily at delivery system 20. In this way, the user will be able to access objects required to support a desired application with minimal intervention of delivery system 20, thus reducing the cost of the session and speeding the response time.

However, the number of objects that can be maintained at RS 400 is restricted by at least two factors: the RS 400 storage capacity; i.e., RAM and disk sizes, and the need to maintain the stored objects current.

In accordance with the method aspect of the invention, in order to optimize the effectiveness of the limited storage space at RS 400, the collection of objects is restricted to those likely to be requested by the user; i.e., tailored to the user's tastes--and to those least likely to be time sensitive; i.e., objects which are stable. To accomplish this, objects are coded for storage candidacy to identify when they will be permitted at RS 400, and subject to the LRU algorithm to maintain presence at RS 400. Additionally, to assure currency of the information and transaction support provided at RS 400, objects are further coded for version identification and checking in accordance with a system of priorities that are reflected in the storage candidacy coding.

Specifically, to effect object storage management, objects are provided with a coded version id made up of the storage control byte and version control bytes identified above as elements of tie object header, specifically, bytes 16 and 18 shown in FIG. 4b. In preferred form, the version id is comprised of bytes 16 and 18 to define two fields, a first 13 bit field to identify the object version and a second three bite field to identify the object storage candidacy.

In this arrangement, the storage candidacy value of the object is addressed to not only the question of storage preference but also object currency. Specifically, the storage candidacy value establishes the basis upon which the object will be maintained at RS 400 and also identifies the susceptibility of the object to becoming stale by dictating when the object will be version checked to determine currency.

The version value of the object on the other hand, provides a parameter that can be checked against predetermined values available from delivery system 20 to determine whether an object stored at RS 400 is sufficiently current to permit its continued use, or whether the object has become stale and needs to be replaced with a current object from delivery system 20.

Still further, in accordance with the invention, object storage management procedure further includes use of the LRU algorithm, for combination with the storage and version coding to enable discarding of objects which are not sufficiently used to warrant retention, thus personalizing the store of objects at RS 400 to the user's tastes. Particularly, object storage facility 439, in accordance with the LRU algorithm maintains a usage list for objects. As objects are called to support the user's applications requests, the objects are moved to the top of a usage list. As other objects are called, they push previously called objects down in the list. If an object is pushed to the bottom of the list before being recalled, it will be forfeited from the list if necessary to make room for the next called object. As will be appreciated, should a previously called object be again called before it is displaced from the list, it will be promoted to the top of the list, and once more be subject to depression in the list and possible forfeiture as other objects are called.

As pointed out above, in the course of building the screens presented to the user, objects will reside at various locations in RS 400. For example, objects may reside in the RS 400 RAM where the object is supporting a particular application screen then running or in a cache maintained at either RAM or disk 424 where the object is being held for an executing application or staged on the fixed size file on disk 424 noted above where the object is being held for use in application likely to be called by the user in the future.

In operation, the LRU algorithm is applied to all these regions and serves to move an object from RAM cache to disk cache to disk file, and potentially off RS 400 depending on object usage.

With regard to the storage candidacy value, in this arrangement, the objects stored at RS 400 include a

limited set of permanent objects; e.g., those supporting logon and logoff, and other non-permanent objects which are subject to the LRU algorithm to determine whether the objects should be forfeited from RS 400 as other objects are added. Thus, in time, and based on the operation of the LRU algorithm and the storage candidacy value, the collection of objects at RS 400 will be tailored to the usage characteristics of the subscriber; i.e., self-configuring.

More particularly, the 3-bit field of the version id that contains the storage candidacy parameter can have 8 different values. A first candidacy value is applied where the object is very sensitive to time; e.g., news items, volatile pricing information such as might apply to stock quotes, etc. In accordance with this first value, the object will not be permitted to be stored on RS 400, and RS 400 will have to request such objects from delivery system 20 each time it is accessed, thus, assuring currency. A second value is applied where the object is sensitive to time but less so than the first case; e.g., the price of apples in a grocery shopping application. Here, while the price might change from day to day, it is unlikely to change during a session. Accordingly the object will be permitted to persist in RAM or at the disk cache during a session, but will not be permitted to be maintained at RS 400 between sessions.

Continuing down the hierarchy of time sensitivity, where the object concerns information sufficiently stable to be maintained between sessions, a third storage candidacy value is set to permit the object to be stored at RS 400 between sessions, on condition that the object will be version check the first time it is accessed in a subsequent session. As will be appreciated, during a session, and under the effect of the LRU algorithm, lack of use at RS 400 of the object may result in it being forfeited entirely to accommodate new objects called for execution at RS 400.

Still further, a fourth value of storage candidacy is applied where the object is considered sufficiently stable as not to require version checking between sessions; e.g., objects concerning page layouts not anticipated to change. In this case, the storage candidacy value may be encoded to permit the object to be retained from session to session without version checking. Here again, however, the LRU algorithm may cause the object to forfeit its storage for lack of use.

Where the object is of a type required to be stored at RS 400, as for example, objects needed to support standard screens, it is coded for storage between sessions and not subject to the LRU algorithm forfeiture. However, where such objects are likely to change in the future they may be required to be version checked the first time they are accessed in a session and thus be given a fifth storage candidacy value. If, on the other hand, the required stored object is considered likely to be stable and not require even version checking; e.g., logon screens, it will be coded with a sixth storage candidacy value for storage without version checking so as to create a substantially permanent object.

Continuing, where a RS 400 includes a large amount of combined RAM and disk capacity, it would permit more objects to be stored. However, if objects were simply coded in anticipation of the larger capacity, the objects would potentially experience difficulty, as for example, undesired forfeiture due to capacity limitations if such objects were supplied to RS 400 units having smaller RAM and disk sizes. Accordingly, to take advantage of the increased capacity of certain RS 400 units without creating difficulty in lower capacity units, objects suitable for storage in large capacity units can be so coded for retention between sessions with a seventh and eighth storage candidacy value depending upon whether the stored large capacity object requires version checking or not. Here, however, the coding will be interpreted by smaller capacity units to permit only cacheable storage to avoid undesirable forfeiture that might result from over filling the smaller capacity units

Where an object is coded for no version checking need may nonetheless arise for a version check at some point. To permit version checking of such objects, a control object is provided at RS 400 that may be version checked on receipt of a special communication from delivery system 20. If the control object

fails version check, then a one shot version checking attribute is associated with all existing objects in RS 400 that have no version checking attributes. Thereafter, the respective objects are version checked, the one shot check attribute is removed and the object is caused to either revert to its previous state if considered current or be replaced if stale.

Still further, objects required to be stored at RS 400 which are not version checked either because of lack of requirement or because of no version check without a control object, as described above, can accumulate in RS 400 as dead objects. To eliminate such accumulation, all object having required storage are version checked over time. Particularly, the least recently used required object is version checked during a session thus promoting the object to the top of the usage list if it is still to be retained at RS 400. Accordingly, one such object will be checked per session and over time, all required objects will be version checked thereby eliminating the accumulation of dead objects.

However, in order to work efficiently, the version check attribute of the object should be ignored, so that even required object can be version checked. Yet, in certain circumstances, e.g., during deployment of new versions of the reception system software containing new objects not yet supported on delivery system 20 which may be transferred to the fixed storage file of RS 400 when the new version is loaded, unconditional version checking may prematurely deletes the object from the RS 400 as not found on delivery system 20. To avoid this problem, a sweeper control segment in the control object noted above can be used to act as a switch to turn the sweep of dead objects on and off.

With respect to version checking for currency, where an object stored at RS 400 is initially fetched or accessed during a session, a request to delivery system 20 is made for the object by specifying the version id of the object stored at RS 400.

In response, delivery system 20 will advise the reception system 400 either that the version id of the stored object matches the currency value; i.e., the stored object is acceptable, or deliver a current object that will replace the stored object shown to be stale. Alternatively, the response may be that the object was not found. If the version of the stored object is current, the stored object will be used until verified again in accordance with its storage candidacy. If the stored object is stale, the new object delivered will replace the old one and support the desired screen. If the response is object not found, the stored object will be deleted.

Therefore, based on the above description, the method aspect of the invention is seen to include steps for execution at storage facility 439 which enables object reception, update and deletion by means of a combination of operation of the LRU algorithm and interpretation of the storage candidacy and version control values. In turn, these procedures cooperate to assure a competent supply of objects at RS 400 so as to reduce the need for intervention of delivery system 20, thus reducing cost of information supply and transactional support so as to speed the response to user requests.

TBOL interpreter 438 shown in FIG. 8 provides the means for executing program objects, which have been written using an interpretive language, TBOL described above. TBOL interpreter 438 interprets operators and operand contained in program object 508, manages TBOL variables and data, maintains buffer and stack facilities, and provides a run time library of TBOL verbs.

TBOL verbs provide support for data processing, program flow control, file management, object management, communications, text display, command bar control, open/close window, page navigation and sound. TBOL interpreter also interacts with other native modules through commands contained in TBOL verbs. For example: the verb "navigate" will cause TBOL interpreter 438 to request object interpreter 435 to build a PPT based on the PTO id contained in the operand of the NAVIGATE verb; "fetch" or "GET" will cause TBOL interpreter 438 to request an object from object storage facility 439;

"SET.sub.-- FUNCTION" will assign a filter to events occurring at the keyboard manger 434; and "FORMAT," "SEND," and "RECEIVE" will cause TBOL interpreter 438 to send application level requests to object/communications manager interface 433.

Data areas managed by TBOL interpreter 438 and available to TBOL programs are Global External Variables (GEVs), Partition External Variables (PEVs), and Runtime Data Arrays (RDAs).

GEVs contain global and system data, and are accessible to all program objects as they are executed. GEVs provide a means by which program objects may communicate with other program objects or with the RS native code, if declared in the program object. GEVs are character string variables that take the size of the variables they contain. GEVs may preferably contain a maximum of 32,000 variables and are typically used to store such information as program return code, system date and time, or user sex or age. TBOL interpreter 438 stores such information in GEVs when requested by the program which initiated a transaction to obtain these records from the RS or user's profile stored in the interactive system.

Partition external variables (PEVs) have a scope restricted to the page partition on which they are defined. PEVs are used to hold screen field data such that when PEOs and window objects are defined, the fields in the page partitions with which these objects are to be associated are each assigned to a PEV. When applications are executed, TBOL interpreter 438 transfers data between screen fields and their associated PEV. When the contents of a PEV are modified by user action or by program direction, TBOL interpreter 428 makes a request to display manager 461 to update the screen field to reflect the change. PEVs are also used to hold partition specific application data, such as tables of information needed by a program to process an expected screen input.

Because the scope of PEVs is restricted to program objects associated with the page partition in which they are defined, data that is to be shared between page partitions or is to be available to a page-level processor must be placed in GEVs or RDAs.

RDAs are internal stack and save buffers used as general program work areas. RDAs are dynamically defined at program object "runtime" and are used for communication and transfer of data between programs when the data to be passed is not amenable to the other techniques available. Both GEVs and RDAs include, in the preferred embodiment, 8 integer registers and 8 decimal registers. Preferably, there are also 9 parameter registers limited in scope to the current procedure of a program object.

All variables may be specified as operand of verbs used by the virtual machine. The integer and decimal registers may be specified as operand for traditional data processing. The parameter registers are used for passing parameters to "called" procedures. The contents of these registers are saved on an internal program stack when a procedure is called, and are restored when control returns to the "calling" procedure from the "called" procedure.

TBOL interpreter 438, keyboard manger 434, object interpreter 435, and object storage facility 439, together with device control provided by operating environment 450, have principal responsibility for the management and execution of partitioned applications at the RS 400. The remaining native code modules function in support and ancillary roles to provide RS 400 with the ability display partitioned applications to the user (display manager 461), display advertising (ad manager 442), to collect usage data for distribution to interactive network 10 for purposes of targeting such advertising (data collection manager 441), and prepare for sending, and send, object and messages to interactive network 10 (object/communications manager interface 443 and link communications manager 444) Finally, the fatal error manager exists for one purpose: to inform the user of RS 400 and transmit to interactive network 10 the inability of RS 400 to recover from a system error.

Display manager 461 interfaces with a decoder using the North American Presentation Level Protocol Syntax (NAPLPS), a standard for encoding graphics data, or text code, such as ASCII, which are displayed on monitor 412 of the user's personal computer 405 as pictorial codes. Codes for other presentation media, such as audio, can be specified by using the appropriate type code in the presentation data segments. Display manager 461 supports the following functions: send NAPLPS strings to the decoder; echo text from a PEV; move the cursor within and between fields; destructive or non-destructive input field character deletion; "ghost" and "unghost" fields (a ghosted field is considered unavailable, unghosted available); turn off or on the current field cursor; open, close, save and restore bit maps for a graphics window; update all current screen fields by displaying the contents of their PEVs, reset the NAPLPS decoder to a known state; and erase an area of the screen by generating and sending NAPLPS to draw a rectangle over that area. Display manager 461 also provides a function to generate a beep through an interface with a machine-dependent sound driver.

Ad manager 442 is invoked by object interpreter 435 to return the object id of the next available advertising to be displayed. Ad manager 442 maintains a queue of advertising object id's targeted to the specific user currently accessing interactive network 10. Advertising objects are pre-fetched from interactive system 10 from a personalized queue of advertising that is constructed using data previously collected from user generated events and/or reports of objects used in the building of pages or windows, compiled by data collection manager 466 and transmitted to interactive system 10.

Advertising objects 510 are PEOs that, through user invocation of a "LOOK" command, cause navigation to partitioned applications that may themselves support, for example, ordering and purchasing of merchandise.

An advertising list, or "ad queue," is requested in a transaction message to delivery system 20 by ad manager 442 immediately after the initial logon response. The logon application at RS 400 places the advertising list in a specific RS global storage area called a SYS.sub.-- GEV (system global external variable), which is accessible to all applications as well as to the native RS code). The Logon application also obtains the first two ad object id's form the queue and provides them to object storage facility 439 so the advertising objects can be requested. However, at logon, since no advertising objects are available at RS local storage facilities 440 ad objects, in accordance with the described storage candidacy, not being retained at the reception system between sessions, they must be requested from interactive network 10.

In a preferred embodiment, the following parametric values are established for ad manager 442: advertising queue capacity, replenishment threshold for advertising object id's and replenishment threshold for number of outstanding pre-fetched advertising objects. These parameters are set up in GEVs of the RS virtual machine by the logon application program object from the logon response from high function system 110. The parameters are then also accessible to the ad manager 442. Preferred values are an advertising queue capacity of 15, replenishment value of 10 empty queue positions and a pre-fetched advertising threshold of 3.

Ad manager 442 pre-fetches advertising object by passing advertising object id's from the advertising queue to object storage facility 439 which then retrieves the object from the interactive system if the object is not available locally. Advertising objects are pre-fetched, so they are available in RS local store 440 when requested by object interpreter 435 as it builds a page. The ad manager 442 pre-fetches additional advertising objects whenever the number of pre-fetched advertising objects not called by object interpreter 435; i.e. the number of remaining advertising objects, falls below the pre-fetch advertising threshold.

Whenever the advertising i.d. queue has more empty positions than replenishment threshold value, a call is made to the advertising queue application in high function system 110 shown in FIG. 2, via object/communications manager interface 443 for a number of advertising object id's equal to the threshold value. The response message from system 110 includes a list of advertising object id's, which ad manager 442 enqueues.

Object interpreter 435 requests the object id of the next advertising from ad manager 442 when object interpreter 435 is building a page and encounters an object call for a partition and the specified object-id equals the code word, "ADSLOT." If this is the first request for an advertising object id that ad manager 442 has received during this user's session, ad manager 442 moves the advertising list from the GEV into its own storage area, which it uses as an advertising queue and sets up its queue management pointers, knowing that the first two advertising objects have been pre-fetched.

Ad manager 442 then queries object storage facility 439, irrespective of whether it was the first request of the session. The query asks if the specified advertising object id pre-fetch has been completed, i.e., is the object available locally at the RS. If the object is available locally, the object-id is passed to object interpreter 435, which requests it from object storage facility 439. If the advertising object is not available in local store 440, ad manager 442 attempts to recover by asking about the next ad that was pre-fetched. This is accomplished by swapping the top and second entry in the advertising queue and making a query to object storage facility 439 about the new top advertising object id. If that object is not yet available, the top position is swapped with the third position and a query is made about the new top position.

Besides its ability to provide advertising that have been targeted to each individual user, two very important response time problems have been solved by ad manager 442 of the present invention. The first is to eliminate from the new page response time the time it takes to retrieve an advertising object from the host system. This is accomplished by using the aforementioned prefetching mechanism.

The second problem is caused by pre-fetching, which results in asynchronous concurrent activities involving the retrieval of objects from interactive system 10. If an advertising is prefetched at the same time as other objects required for a page requested, the transmission of the advertising object packets could delay the transmission of the other objects required to complete the current page by the amount of time required to transmit the advertising object(s). This problem is solved by the structuring the requests from object interpreter 435 to the ad manager 442 in the following way:

1. Return next object id of pre-fetched advertising object & pre-fetch another;

2. Return next advertising object id only; and

3. Pre-fetch next advertising object only.

By separating the function request (1) into its two components, (2) and (3), object interprets 435 is now able to determine when to request advertising object id's and from its knowledge of the page build process, is able to best determine when another advertising object can be pre-fetched, thus causing the least impact on the page response time. For example, by examining the PPT, object interpreter 435 may determine whether any object requests are outstanding. If there are outstanding requests, advertising request type 2 would be used. When all requested objects are retrieved, object interpreter 435 then issues an advertising request type 3. Alternatively, if there are no outstanding requests, object interpreter 435 issues an advertising request type 1. This typically corresponds to the user's "think time" while examining the information presented and when RS 400 is in the Wait for Event state (D).

Data collection manager 441 is invoked by object interpreter 435 and keyboard manger 434 to keep records about what objects a user has obtained (and, if a presentation data segment 530 is present, seen) and what actions users have taken (e.g. "NEXT," "BACK," "LOOK," etc.)

The data collection events that are to be reported during the user's session are sensitized during the logon process. The logon response message carries a data collection indicator with bit flags set to "on" for the events to be reported. These bit flags are enabled (on) or disabled (off) for each user based on information contained in the user's profile stored and sent from high function host 110. A user's data collection indicator is valid for the duration of his session. The type of events to be reported can be changed at will in the host data collection application. However, such changes will affect only users who logon after the change.

Data collection manager 441 gathers information concerning a user's individual system usage characteristics. The types of informational services accessed, transactions processed, time information between various events, and the like are collected by data collection manager 441, which compiles the information into message packets (not shown). The message packets are sent to network 10 via object/communication manager interface 443 and link communications manager 444. Message packets are then stored by high function host 110 and sent to an offline processing facility for processing. The characteristics of users are ultimately used as a means to select or target various display objects, such as advertising objects, to be sent to particular users based on consumer marketing strategies, or the like, and for system optimization.

Object/communications manager interface 443 is responsible for sending and receiving DIA (Data Interchange Architecture described above) formatted messages to or from interactive network 10. Object/communications manager 443 also handles the receipt of objects, builds a DIA header for messages being sent and removes the header from received DIA messages or objects, correlates requests and responses, and guarantees proper block sequencing. Object/communications manager interface 443 interacts with other native code modules as follows: object/communications manager 443 (1) receives all RS 400 object requests from object storage facility 439, and forwards objects received from network 10 via link communications manager 444 directly to the requesting modules; (2) receives ad list requests from ad manager 442, which thereafter periodically calls object/communications manager 443 to receive ad list responses; (3) receives data collection messages and send requests from data collection manager 441; (4) receives application-level requests from TBOL interpreter 438, which also periodically calls object/communications manager interface 443 to receive responses (if required); and(5) receives and sends DIA formatted objects and messages from and to link communications manager 444.

Object/ communications manager interface 443 sends and receives DIA formatted messages on behalf of TBOL interpreter 438 and sends object requests and receives objects on behalf of object storage facility 439. Communication packets received containing parts of requested objects are passed to object storage facility 439 which assembles the packets into the object before storing it. If the object was requested by object interpreter 435, all packets received by object storage facility 439 are also passed to object interpreter 435 avoiding the delay required to receive an entire object before processing the object. Objects which are pre-fetched are stored by object storage facility 439.

Messages sent to interactive network 10 are directed via DIA to applications in network 10. Messages may include transaction requests for records or additional processing of records or may include records from a partitioned application program object or data collection manager 441. Messages to be received from network 10 usually comprise records requested in a previous message sent to network 10. Requests received from object storage facility 439 include requests for objects from storage in interactive system 10. Responses to object requests contain either the requested object or an error code indicating an error condition.

Object/communications manager 443 is normally the exclusive native code module to interface with link communications manager 444 (except in the rare instance of a fatal error). Link communications manager 444 controls the connecting and disconnecting of the telephone line, telephone dialing, and communications link data protocol. Link communications manager 444 accesses network 10 by means of a communications medium (not shown) link communications manager 444, which is responsible for a dial-up link on the public switched telephone network (PSTN). Alternatively, other communications means, such as cable television or broadcast media, may be used. Link communications manager 444 interfaces with TBOL interpreter for connect and disconnect, and with interactive network 10 for send and receive

Link communication s manager 444 is subdivided into modem control and protocol handler units. Modem control (a software function well known to the art) hands the modem specific handshaking that occurs during connect and disconnect. Protocol handler is responsible for transmission and receipt of data packets using the TCS (TRINTEX Communications Subsystem) protocol (which is a variety of OSI link level protocol, also well known to the art).

Fatal error manager 469 is invoked by all reception system components upon the occurrence of any condition which precludes recovery. Fatal error manager 469 displays a screen to the user with a textual message and an error code through display manager 461. Fatal error manager 469 sends an error report message through the link communications manager 444 to a subsystem of interactive network 10.

The source code for the reception system software as noted above is described in parent application Ser. No. 388,156 filed Jul. 28, 1989, now issued as U.S. patent

SAMPLE APPLICATION

Page 255 illustrated in FIG. 3b corresponds to a partitioned application that permit's a user to purchase apples. It shows how the monitor screen 414 of the reception system 400 might appear to the user. Displayed page 255 includes a number of page partitions and corresponding page elements.

The page template object (PTO) 500 representing page 255 is illustrated in FIG. 9. PTO 500 defines the composition of the page, including header 250, body 260, display fields 270, 271, 272, advertising 280, and command bar 290. Page element objects (PEOs) 504 are associated with page partitions numbered; e.g., 250, 260, 280. They respectively, present information in the header 250, identifying the page topic as ABC APPLES; in the body 260, identifying the cost of apples; and prompt the user to input into fields within body 260 the desired number of apples to be ordered. In advertising 280, presentation data and a field representing a post-processor that will cause the user to navigate to a targetable advertising, is presented.

In FIG. 9, the structure of PTO 500 can be traced. PTO 500 contains a page format call segment 526, which calls page format object (PFO) 502. PFO 502 describes the location and size of partitions on the page and numbers assigned to each partition. The partition number is used in page element call segments 522 so that an association is established between a called page element object (PEO) 504 and the page partition where it is to be displayed. Programs attached to this PEO can be executed only when the cursor is in the page partition designated within the PEO.

PTO 500 contains two page element call segments 522, which reference the PEOs 504 for partitions 250 and 260. Each PEO 504 defines the contents of the partition. The header in partition 250 has only a presentation data segment 530 in its PEO 504. No input, action, or display fields are associated with that partition.

The PEO 504 for partition 260 contains a presentation data segment 530 and field definition segments 516 for the three fields that are defined in that partition. Two of the fields will be used for display only. One field will be used for input of user supplied data.

In the example application, the PEO 504 for body partition 260 specifies that two program objects 508 are part of the body partition. The first program, shown in Display field 270, 271, 272, is called an initializer and is invoked unconditionally by TBOL interpreter 438 concurrently with the display of presentation data for the partition. In this application, the function of the initializer is represented by the following pseudo-code:

1. Move default values to input and display fields;

2. "SEND" a transaction to the apple application that is resident on interactive system 10;

3. "RECEIVE" the result from interactive system 10; i.e. the current price of an apple;

4. Move the price of an apple to PEV 271 so that it will be displayed;

5. Position the cursor on the input field; and

6. Terminate execution of this logic.

The second program object 508 is a field post-processor. It will be invoked conditionally, depending upon the user keystroke input. In this example, it will be invoked if the user changes the input field contents by entering a number. The pseudo code for this post-processor is as follows

1. Use the value in PEV 270 (the value associated with the data entered by the user into the second input data field 270) to be the number of apples ordered.

2. Multiply the number of apples ordered times the cost per apple previously obtained by the initializer;

3. Construct a string that contains the message "THE COST OF THE APPLES YOU ORDERED IS $45.34;";

4. Move the string into PEV 272 so that the result will be displayed for the user; and

5. Terminate execution of this logic.

The process by which the "APPLES" application is displayed, initialized, and run is as follows.

The "APPLES" application is initiated when the user navigates from the previous partitioned application, with the navigation target being the object id of the "APPLES" PTO 500 (that is, object id ABC1). This event causes keyboard manager 434 to pass the PTO object id, ABC1 (which may, for example, have been called by the keyword navigation segment 520 within a PEO 504 of the previous partitioned application), to object interpreter 435. With reference to the RS application protocol depicted in FIG. 6, when the partitioned application is initiated, RS 400 enters the Process Object state (B) using transition (1). Object interpreter 435 then sends a synchronous request for the PTO 500 specified in the navigation event to object storage facility 439. Object storage facility 439 attempts to acquire the requested object from local store 440 or from delivery system 20 by means of object/communication manager 443, and returns an error code if the object cannot be acquired.

Once the PTO 500 is acquired by object/communications manager 443, objectinterpreter 435 begins to build PPT by parsing PTO 500 into its constituent segment calls to pages and page elements, as shown in FIG. 4d and interpreting such segments. PFO and PEO call segments 526 and 522 require the acquisition of the corresponding objects with object id's <ABCF>, <ABCX> and <ABCY>. Parsing and interpretation of object ABCY requires the further acquisition of program objects <ABCI> and <ABCJ>.

During the interpretation of the PEOs 504 for partitions 250 and 260, other RS 400 events are triggered. This corresponds to transition (2) to interpret pre-processors state (C) in FIG. 6. Presentation data 530 is sent to display manager 461 for display using a NAPLPS decoder within display manager 461, and, as the PEO <ABCY> for partition 260 is parsed and interpreted by object interpreter 435, parameters in program call segment 532 identify the program object <ABCI> as an initializer. Object interpreter 435 obtains the program object from object storage facility 439, and makes a request to TBOL interpreter 438 to execute the initializer program object 508 <ABCI>. The initializer performs the operations specified above using facilities of the RS vitual machine. TBOL interpreter 438, using operating environment 450, executes initializer program object 506 <ABCI>, and may, if a further program object 508 is required in the execution of the initializer, make a synchronous application level object request to object storage facility 439. When the initializer terminates, control is returned to object interpreter 435, shown as the return path in transition (2) in FIG. 6.

Having returned to the process object state (B), object processor 435 continues processing the objects associated with PTO <ABC1>. Object interpreter continues to construct the PPT, providing RS 400 with an environment for subsequent processing of the PTO <ABC1> by pre-processors and post-processors at the page, partition, and field levels. When the PPT has been constructed and the initializer executed, control is returned to keyboard manager 434, and the RS enters the wait for event (E) State, via transition (4), as shown in FIG. 6.

In the wait for event state, the partitioned application waits for the user to create an event. In any partitioned application, the user has many options. For example, the user may move the cursor to the "JUMP" field 296 on the command bar 290, which is outside the current application, and thus cause subsequent navigation to another application. For purposes of this example, it is assumed that the user enters the number of apples he wishes to order by entering a digit in display field 271.

Keyboard manager 434 translates the input from the user's keyboard to a logical representation independent of any type of personal computer. Keyboard manager 434 saves the data entered by the user in a buffer associated with the current field defined by the location of the cursor. The buffer is indexed by its PEV number, which is the same as the field number assigned to it during the formation of the page element. Keyboard manager 434 determines for each keystroke whether the keystroke corresponds to an input event or to an action or completion event Input events are logical keystrokes and are sent by keyboard manager to display manager 461, which displays the data at the input field location. Display manager 461 also has access to the field buffer as indexed by its PEV number.

The input data are available to TBOL interpreter 438 for subsequent processing. When the cursor is in a partition, only the PEVs for that partition are accessible to the RS virtual machine. After the input from the user is complete (as indicated by a user action such as pressing the RETURN key or entry of data into a field with an action attribute), RS 400 enters the Process Event state (E) via transition (4).

For purposes of this example, let us assume that the user enters the digit "5" in input field 270. A transition is made to the process event state (E). Keyboard manager 434 and display manager 437 perform a number of actions, such as the display of the keystroke on the screen, the collection of the

keystroke for input, and optionally, the validation of the keystroke, i.e. numeric input only in numeric fields. When the keystroke is processed, a return is made to the wait for event state (D). Edit attributes are specified in the field definition segment.

Suppose the user inputs a "6" next. A transition occurs to the PE state and after the "6" is processed, the Wait for Event (D) state is reentered. If the user hits the "completion" key (e.g.,ENTER) the Process Event (E) state will be entered. The action attributes associated with field 272 identify this as a system event to trigger post-processor program object <ABCJ>. When the interpretive execution of program object <ABCJ> is complete, the wait for event state (D) will again be entered. The user is then free to enter another value in the input field, or select a command bar function and exit the apples application.

While this invention has been described in its preferred form, it will be appreciated that changes may be made in the form, construction, procedure and arrangement of its various element and steps without departing from its spirit or scope.

* * * * *