APPLICATION BUILDER

REFERENCE MANUAL

RELATE^{tm}/3000

Relational Data Base Management System

CRI, INCORPORATED
5333 Betsy Ross Drive
P. O. Box 58004
Santa Clara, CA 95052
(408) 980-9898

HPB-RF02
8803

TABLE OF CONTENTS

INTRODUCTION

This manual describes APPLICATION BUILDER, an application construction facility available for RELATE(tm)/3000 — CRI's Relational Data Base Management System. RELATE/3000 is designed to provide database manipulation capabilities to both technical and non-technical users. BUILDER extends this capability by enabling screen-based applications to be quickly created for end-users that have little knowledge of RELATE databases, or even of computers in general.

BUILDER is both a tool and an environment for the creation and use of screen-based applications. BUILDER commands are extensions to RELATE commands. Most RELATE commands may also be executed from within BUILDER. BUILDER provides additional capabilities specifically designed to control screen-based applications.

In order to most effectively use BUILDER, you should be familiar with RELATE.

CAPABILITIES AND FEATURES

The major features of BUILDER include:

* Applications may be stored in standard HP EDITOR files or QEDIT files. Application developers are not required to learn a new editor. Programs may be used to assist in the creation of applications.

* BUILDER can be used on virtually any cursor addressable CRT. BUILDER does not require HP or HP-compatible block mode terminals, although block mode is also supported.

* New terminal types may be defined as required thus providing support for a wide range of CRTs.

* The straightforward BUILDER syntax allows a wide range of users to create their own applications. Very little programming knowledge is required to create simple applications, yet the full power of RELATE is available for complex applications.

* Screen layouts and actions are stored in the same file for easy system maintenance.

* Debugging facilities are built-in to simplify system development.

* Function keys and function key labeling are supported.

* BUILDER can perform validity checking on a field by field or character by character basis, thus increasing the responsiveness and user friendliness of applications.

* The BUILDER language is a superset of the commands available in RELATE.

* BUILDER can call SPL, COBOL, FORTRAN, and PASCAL subroutines that reside in an SL for CPU intensive operations. For larger systems a method is available to compile the subroutines into BUILDER itself.

* Both screen-based and scrolled I/O can be used in the same application.

* Powerful substitution routines are built-in for menu oriented applications.

* Application files need not be (but can be) compiled.

* BUILDER applications can be run in batch mode.

* BUILDER encourages installation-wide standards for field and error enhancements.

* Jobs can be created and streamed directly from BUILDER.

SECTION 1

C O N C E P T S

NOTATION

The following notation is used to define the syntax of the commands described in this manual:

CAPITALIZED WORDS	are the names of commands or keywords in commands.
lower-case words	identify words that are names or expressions to be specified by the user.
[ ] (Square Brackets)	are used to indicate that the enclosed item is optional and may be omitted.
| (Vertical Bars)	between items indicate that one of the items must be chosen.
... (Ellipses)	indicate that the previous item (or, if in square brackets, the previous bracketed item) may be repeated zero or more times.

Commands are recognized in any combination of upper and lower case letters. They are processed as if only upper case characters had been used (except for information in double quotes). Commands may be up to 1500 characters long before parameter substitution takes place (as described later). Any commands which are RELATE commands must not contain more than 1500 characters after the substitution.

All BUILDER commands must be completely written out. No abbreviations are understood unless specifically noted. Any command not recognized as a BUILDER command is passed to RELATE. RELATE commands may be abbreviated as specified in the RELATE reference manual.

SPECIAL CHARACTERS

& (ampersand)	An ampersand entered as the last non-blank character on a line indicates that the command continues on the next line.
//	When executing the RELATE Command Interpreter through BUILDER, the "//" terminates the Command Interpreter and returns the user to BUILDER.
Control-H	(or Backspace) One previously typed character is deleted each time this key is used. If the cursor is at the beginning of a variable, this key will cause a backtab to the beginning of the previous field.
Control-Q	The Control-Q resumes output suspended by the Control-S.
Control-S	The Control-S suspends output to the terminal. Output may be resumed by entering a Control-Q.
Control-Y	This sequence is used to terminate lengthy printout or cancel the execution of an operation. See the CREATE PARTITION command for additional information.
ESCAPE	(sometimes labeled ESC, ALT, or ALTMODE) When pressed and followed by another character, a function key operation will occur.
RETURN	Pressing the RETURN key informs BUILDER that editing and entry on the screen is completed and processing can begin.
TAB	(or Control-I) Moves the cursor to the start of the next variable in the current screen. If another variable cannot be found, the cursor will be placed at the first variable.
other	See Section 2 for descriptions of keys that will BACKTAB, HOME, move the cursor LEFT, RIGHT, UP or DOWN, and REDRAW the screen.
ENTER	The ENTER key is used by applications running in Block mode to indicate that editing and entry on the screen is completed and processing can begin. If the application is NOT running in Block Mode, this key should never be used as it can put invalid data on your screen.
RETURN/ENTER	Indicates that ENTER should be pressed if operating in Block Mode, and RETURN should be pressed otherwise.

GLOSSARY

The definitions given below refer specifically to BUILDER:

actions – Commands attached to a variable, function key, or screen.

application – The result of creating and executing an application file. Sometimes used synonymously with "application file".

application file – An editor file containing one or more SCREEN sections and BUILDER commands.

array screen – A screen that contains two or more variables with the same name. An array screen allows several records from a file to be displayed at the same time.

block mode – A method of terminal data transfer where the data for a screen is collected and edited by an intelligent terminal without intervention by the computer. When an enter is issued the whole screen (or perhaps the unprotected variables) is transferred in a large block to the computer for processing.

character mode – A method of terminal data transfer where individual characters are read and acted upon.

creator – The person who uses BUILDER to create an application.

CRT – A Cathode Ray Tube. An abbreviation for terminals that use a television type screen for output.

cursor – An underline or inversed video mark on a screen that shows the operator where the next character typed will be placed.

cursor addressable – The ability to move a cursor directly to any given point on the screen by sending instructions to the terminal. In BUILDER, these instructions are defined when a terminal is configured and are issued without explicit instructions by the creator or user of an application.

device independent – A program is device independent if it can be used on many different devices (terminals). If a program depends on functions keys, memory, or special capabilities of any kind, it is not device independent.

end-user – The user of an application.

enhancement – Displaying a variable or text on a screen in inverse video, half-bright, color, etc. Enhancements are normally used to indicate required variables or variables in error.

entering screen – Signaling the computer that the current screen is complete and the computer should take action on the data entered. The completion of a screen is signaled by pressing the RETURN key.

ENTER section – this section is invoked through BUILDER with the use of the RETURN key, not with the ENTER key. The ENTER key should NEVER be used with BUILDER.

field – A column of information in a file. The same type of data is stored in the field in each record. Each field has a length, a type, and restrictions as to content.

form – Same as a screen. A fill-in-the-blanks display presented on a CRT.

function keys – Special keys that, when pressed, cause the application to perform an action. Often referred to as an f-key. In some modes, an f-key is simulated by pressing the escape key followed by a number key.

global – A variable which has been declared either in the GLOBAL section or in some other screen previous to the current screen which has been reached by at least one CALL SCREEN.

layout – The placement of variables and information on a screen.

line mode – A method of terminal data transfer in which a line or a complete variable is read and acted upon.

menu – A screen that displays several choices and allows the user to select one or more of the choices.

message line – Normally, the message line is the last line on the screen. Text displayed by BUILDER is generally placed in the message line.

partition – A table which contains status information on a set of open files. Partitions correspond directly to the cursors described in the RELATE Host Language Interface manual.

prompt – A message or character displayed to the user that indicates that the computer requires input from the user.

RETURN/ENTER – Indicates that ENTER should be pressed if operating in Block Mode, and RETURN should be pressed otherwise.

screen – A fill-in-the-blanks display presented on a CRT.

scrolling – The act of moving lines on a screen up to add new lines at the bottom of the screen. Old lines typically disappear off the top of the screen.

terminal type – A string that identifies the maker and model number of a specific terminal. (e.g., HP2623A or TVI920C.)

variable – A variable is a named area in an application which can store a single value. Variables can be used in computations and can appear on a screen. Each variable has a length, a type, and restrictions as to content.

EXPRESSION EVALUATION

Some BUILDER commands use expressions in their syntax. The expression analyzer used by BUILDER is separate from the one used by RELATE. The BUILDER expression analyzer is different because of the way variables are handled. In a RELATE command, a variable in an expression such as "1+PARTNO" means the field PARTNO in the file that is the current path. In BUILDER, PARTNO refers to the value of a variable in the BUILDER application. All arithmetic in BUILDER is performed with floating point numbers. This allows for approximately 16 digits of accuracy.

All commands described in this manual use the BUILDER expression evaluator. All RELATE commands described in the RELATE Reference Manual use the RELATE expression evaluator. Most of the time, this distinction can be ignored because the syntax of both expressions is very close; advanced developers, however, may wish to know what the differences are and when one or the other evaluator is being used.

Hierarchy

The hierarchy of expression evaluation is as follows:

1) unary +, -

2) NOT

3) **

4) *, /

5) +, -

6) <, >, <=, >=, <>

7) AND

8) OR

Operators at the same level are not guaranteed to be performed in any particular order. Parentheses may be used to override the above hierarchy or to emphasize the order of evaluation. If nested parentheses are encountered, the inner expressions are evaluated first.

Operators

An expression obtains meaning by connecting constants, variables, functions, and built in variables with various operators. The standard operators available in BUILDER expressions are as follows:

NAME	SYMBOL	EXAMPLE	MEANING
unary plus	+	+5	Positive five.
unary minus	-	-5	Negative five.
plus	+	3+A	Add three and the value of A.
minus	-	3-A	Subtract the value of A from three.
multiply	*	3*A	Multiply three times the value of A.
divide	/	3/A	Divide three by the value of A.
exponent	**	A**2	The value of A squared.
NOT	NOT	NOT A	1, if A is not zero or blank. Otherwise 0.
AND	AND	A AND B	1, If both A and B are non-zero or non-blank. Otherwise 0.
OR	OR	A OR B	1, If A or B or both are non-zero or non-blank. Otherwise 0.
less than	<	A<3	1, if A is less than three.
less than or equal	<=	A<=3	1, if A is less than or equal to three.
equal	=	A=3	1, if A equals three.
greater than	>	A>3	1, if A is greater than three.
greater than or equal	>=	A>=3	1, if A is greater than or equal to three.
not equal	<>	A<>3	1, if A does not equal three.

Comparing Alphabetic Data

When comparisons are made with alphabetic data, the shorter string is assumed to be padded with blanks up to the length of the longer string.

Quotes Around Constants

Date and text constants must be enclosed in quotes, in both BUILDER and RELATE commands. Examples:

```
IF TODAY="7/4/84"

SELECT * WHERE NEW_DATE="%today"
```

Comparing Dates

Dates are compared as dates, not as alphabetic data. If a date is compared to a string, the string is evaluated as a date in the current BUILDER date format (see SET OPTION). Comparisons are not allowed between dates and numbers.

Comparing Numeric and Alpha Data

A numeric variable whose value is zero or blank will compare as equal to either the constant zero (0) or a blank string. In order to differentiate between a zero and a blank, substitute the variable into a string (IF "%x"=" ") or check its length.

SYSTEM DEFINED VARIABLES

The following special variables are defined within the BUILDER interpreter. These variables may be used in BUILDER expressions. The value of each is normally maintained by BUILDER but may be changed by the application with an assignment statement. The variables may also be substituted into commands; however, any calculation required to determine the value of the variable will not be made prior to the substitution. (E.g.:

```
DISPLAY "THE TIME IS %$TIME"
```

will give no value for $TIME unless $TIME has been used earlier without substitution.

```
DISPLAY $TIME
```

will work since no substitution is involved. However,

```
SHOW VARIABLE $TIME
```

does not cause $TIME to be recalculated; it will only display the most recently calculated time.) This will affect the substituted value of $CLOCKTIME, $CPUTIME, $EOF, $FOUND, $STATE, and $TIME.

NAME	MEANING
$BREAK	After a RECORD POINT or RECORD REWIND has been used with the BREAK keyword, this will contain a number indicating which field in the current key has changed value, with the most significant field being one. If $BREAK is zero then BREAK has not been set. If $BREAK is one greater than the value set with BREAK, then no field in the current key has changed value.
$CHANGED	A numeric variable whose value is set to 1 if the user has changed any variable on the current screen or in response to a PROMPT. This is reset only be assigning zero to $CHANGED. See the $CHANGED function to check individual variables.
$CLOCKTIME	A numeric variable which contains the number of tenths of seconds since midnight last occurred.
$CPUTIME	A numeric variable that gives the number of CPU milliseconds that the current process has accumulated.
$DATE	An alphabetic variable which contains the current date in "M/D/Y" format. The format of the date can be changed with the SET OPTION command.
$DEVICE	A string containing the terminal's device type.
$EOF	A numeric variable that indicates if an end of file occurred during the last RECORD READ or RECORD ADD command. A value of one indicates that an end of file was encountered, a value of zero indicates otherwise. The value is reset to zero by RECORD REWIND, RECORD RESET, and a RECORD READ to a valid record.

$ERRMSG	A string containing the error message corresponding to the last error number.
$ERROR	A numeric variable that contains the number of the last error encountered on a RELATE or operating system command being executed after an IGNORE ERROR command within BUILDER. A value of zero indicates that no error has occurred since the beginning of the application or since the last IGNORE ERROR command.
$FOUND	A numeric variable that indicates if the current path (indicated by a SELECT, OPEN FILE, SET PATH, etc.) contains a record or the last RECORD POINT command found a record satisfying the search criteria. A value of one indicates that a record was found, a value of zero indicates that a record was not found. The internal method of evaluating $FOUND may take a large amount of time for a complex SELECT.
$INFORMATION	Returns the string specified with the INFO= parameter on the RUN BUILDER command, if any. If INFO= was not specified, nothing is returned. For example, `:RUN BUILDER.PUB.RELATE45;INFO="STARTUP"` will place "STARTUP" into $INFORMATION.
$KEY	A string variable containing one character. This character is the value of the last function key pressed (such as "1" if F1 was pressed) or the first character of the last response from a PROMPT command which contained no destination variable.
$RECORDS	A numeric variable containing the number of records processed by the last RELATE command. This value corresponds to to partition locations 29 and 30.
$SCREEN	An alphabetic variable whose value is the name of the current screen.
$STATE	An alphabetic variable that contains the name of the current state in the current screen.
$SUBSCRIPT	A numeric variable containing the current array variable subscript value. This variable is normally controlled by BUILDER but can be manipulated by the application. See the Array Variables section for more information.
$TIME	Returns the time of day as a number in HHMMSS format.
$VARIABLE	A string containing the variable name where the cursor last resided.

$BITS (expression, startbit [,count])
Returns the value of the indicated bits. The expression is converted to a 16 bit integer before the bits are extracted. The bits are numbered 0 through 15 starting with the left-most position. If startbit is below 0 or above 15 a zero is returned. If count is not specified, one bit is assumed. If count is specified, it is treated modulo 16.

$CHANGED (varexpression)
Returns the value 1 (TRUE) if the value of the variable whose name is a result of the varexpression has been changed by the user. If the varexpression results in the name of an array variable, only the current subscript of the array is checked. Examples:

```
    IF $CHANGED("ADDRESS")
or:
    VARNAME:="ADDRESS"
    IF $CHANGED(VARNAME)
```

This variable is considered to have been $CHANGED by the user if the user types a new value into the screen or in response to a PROMPT. Once the user has changed the value of a variable, this function will continue to return TRUE unless the application turns off the changed flag with:

CLEAR of that variable
RESET of that variable
INITIAL value is specified in a DECLARATION.

Also allowed is a numeric varexp. This will return a count of the number of array variables with that $SUBSCRIPT which have been changed. If -1 is specified, the count of non-array variables whose values have been changed is returned.

$CONCAT (stringexpression [,...])
Returns a string composed of the concatenated strings. Leading and trailing blanks are removed from the strings before the concatenation takes place unless the strings are constants.

$DAY (dateexpression)
Returns an integer representing the day of the month for the date.

$DAY_DIFF (startdateexpr, enddateexpr)
Returns the number of days between the two dates. If the enddate falls before the startdate, a negative value will be returned.

$DAY_WEEK (dateexpression)
Returns an integer representing the day of the week of the date, where day 1 is Sunday and day 7 is Saturday.

$FORMAT_DATE (dateexpression, format)
Converts the indicated date to a new format. The format must be a valid RELATE date format.

$IN (expression, matchitem1, ...)

Returns a value indicating which of the matchitems (1 through n) the first expression is exactly equal to. If no match is found, zero is returned. The expression may be a string or a numeric value. The matchitems must be of the same type as the first expression.

$ITEM (list, itemnumber [,alphabetic])

Returns the item from the list as indicated by the itemnumber. List is a character string containing items separated by commas. If the itemnumber is less than one or the item does not exist, a null string is returned. If the itemnumber references alphabetic information or the alphabetic parameter is one, a string will be returned. Items within the list are either numeric or alphabetic; the latter may be, but need not be, enclosed in opening and closing quotes or parentheses. Parentheses or quotes enclosing an alphabetic element of the list are removed prior to being returned. Quotes take precedence over parentheses. A numeric value is otherwise returned.

EXAMPLES:

a)
```
NAME:=$ITEM("FRED,SUE,JOE",WHICHNAME,1)
```

b)
```
NAME:=$ITEM("(FRED),(SUE),(JOE)",WHICHNAME,1)
```

c)
```
NAMELIST:="""FRED"",""SUE"",""JOE"""
NAME:=$ITEM(NAMELIST,WHICHNAME,1)
```

d)
```
NAME:="FRED"
AGE:=35
INFOLIST:=$CONCAT(NAME,",",AGE)
NEWNAME:=$ITEM(INFOLIST,1,1)
NEWAGE:=$ITEM(INFOLIST,2,0)
```

e)
```
LIST:="(SMITH,FRED),(JONES,SUE),(BROWN,JOE)"
FULLNAME:=$ITEM(LIST,WHICHNAME,1)
FIRSTNAME=$ITEM($ITEM(LIST,WHICHNAME,1),2,1)
```

$ITEMS (list [,start, finish])

Returns a subset of the list. List is a character string as described in $ITEM containing items separated by commas. If the start and finish parameters are not included, the number of items in the list is returned. If start and finish are included, the items are returned as a string.

$JCW (nameexpression)

Returns the value, set by MPE's SETJCW command, of the JCW whose name is a result of the nameexpression. Examples:

```
X:=$JCW("TERMTYPE")
JCW_NAME:="TERMTYPE"
X:=$JCW(JCW_NAME)
```

$LAST_DAY (dateexpression)

Returns a date (month, day, and year) equal to the last day of the month in the month and year of the original date.

$LENGTH (stringexpression)

Returns the position of the last non-blank character in the string.

$MATCH (stringexpression, matchstringexpression)

Returns the position in the string where the matchstring first occurs. If the matchstring does not occur in the string, a zero is returned. This definition is a subset of the RELATE $MATCH function of the same syntax and does not recognize any of the special characters used in RELATE's $MATCH.

$MONTH (dateexpression)

Returns an integer representing the month of the date.

$NEW_DATE (dateexpression, days)

Returns a date the indicated number of days before or after the original date. If days is negative, a date before the original date will be returned.

$NUMERIC (stringexpression)

Returns a one if the string corresponds to a valid number. If the string does not correspond to a valid number, a zero is returned.

$PAD (stringexpression, padstring, newlength [,mode])

Returns a string of newlength containing the original string padded with padstring on the front or back as determined by the mode. If the mode is not supplied or is zero, the padstring is applied before the first non-blank characters of string. If the mode is non-zero, the padstring is placed after the last non-blank character of string. The padstring, which also has leading and trailing blanks removed, may contain more than one character and will be repeated until string is newlength characters long.

$PARTITION (element)

Returns the value of the element in the current RELATE partition. If the element references the first half of an item defined as a double word value, both will be returned. Reference to an element not defined or to the second half of a double word value will produce an error. For a complete description of the contents of the partition, see the Cursor Format description in the Host Language Interface section of the RELATE reference manual.

$RANDOM (number) If a non-zero number is fed to this function, it will initialize the $RANDOM function seed for additional calls. If the same non-zero number is fed to $RANDOM, it will initialize an identical series of "random" numbers. It is suggested that the first call be something like $RANDOM ($TIME). Additional calls should use 0 (zero) as the number passed. $RANDOM(0) returns a random number between zero and .999...

$RDBINFO(mode [,qualifier])

Returns information from the RELATE RDBINFO intrinsic. The information returned and the functions' format depend on the mode and the qualifier. For a complete explanation of the capabilities, please refer to the detailed $RDBINFO discussion later in this section.

$READ (parexpr, fldexpr, keyexpr [,keyexpr [,...]])

Returns the value(s) of the indicated field from the record designated by the key expression in the current path of the indicated partition. Parexpr is an expression that should evaluate to the name of an existing partition containing a current path. Fldexpr is an expression that should evaluate to the name of a field in that path. Keyexpr is an expression containing the value to be searched for in the current index. Each additional keyexpr corresponds to additional fields in the current index.

If more than one record is found to match the given key values, the values for the requested field will be taken from all qualified records and appended together, with a space between values, up to a length of 1600 characters. Trailing blanks will be removed from each value before the append is performed.

The $READ will read all qualified records and leave the record pointer positioned at the record after the last qualified record.

Example:

```
*** INITIAL
   NOTE   each of these files contains the fields ERRNUM,
   NOTE   SEQUENCE, and TEXT.  There may be several records,
   NOTE   with different sequence numbers, for each
   NOTE   error number.
   CREATE PARTITION FRENCH
      OPEN FILE FRMSG
      SET INDEX ERRNUM
   CREATE PARTITION GERMAN
      OPEN FILE GERMSG
      SET INDEX ERRNUM
   PROMPT "WHAT LANGUAGE?", LANG

*** DECLARATION
   MSG; LENGTH=1500
   LANG; LENGTH=10; UPPER; RANGE="GERMAN","FRENCH"

*** ENTER
  IF some sort of error
     NOTE This error is application error number 2750
     MSG:=$READ(LANG,"TEXT",2750)
     SCROLL MSG
  ENDIF
```

$SUBSTR(stringexpression, start [,length])
: Returns the portion of the string beginning with start for a count of length characters. The first character in the string is one. If the length is not specified, the remainder of the string is returned.

$TEMPLATE (stringexpression)
: Returns the string after it has been processed by the substitution algorithm. For example, $TEMPLATE("The color is %colorname.") would return "The color is red." if the variable colorname contained the text "red".

$UPS (stringexpression)
: Returns the string expression with all lowercase alphabetic characters upshifted.

$YEAR (dateexpression)
: Returns an integer representing the year of the indicated date, including the century.

$RDBINFO

The $RDBINFO function simplifies the interface between the RELATE Host Language Interface RDBINFO subroutine and a BUILDER application. Through this function the application can verify that fieldnames exist in a file, the data type of a field, the size of a file, indexes, etc.. The format of the call is:

$RDBINFO(mode [,qualifier])

If the mode is alphabetic, it is assumed to be the name of a field in the current path. The function will return the number of the field. This value can be used as a qualifier in a call to obtain information about the field. If the field does not exist in the current path, a minus one will be returned. If the mode is "$LINE", the number zero will be returned. If the mode is alphabetic, the qualifier should not be specified.

If the mode is numeric, it corresponds directly to the RDBINFO modes as defined in the Host Language Interface section of the RELATE manual. If the particular call returns more than one value, these values will be placed into a string separated by commas with alphabetic information placed in quotes. The order and type of each item will correspond directly to the order and type of the HLI description. For example, if a file contains field number 3 called TEXT, which is an alphabetic field containing 40 characters, a call of the form $RDBINFO(201,3) would return this string:

"TEXT",4,1,20,40,0,1,1,1,0,0

This string can be manipulated with the $ITEM and $ITEMS functions. For example, the number representing the data type can be obtained by using $ITEM($RDBINFO(201,3),3). Calls that return a single item will return a numeric value.

All modes except 202 and 203 have been implemented.

SECTION 2

TERMINAL INTERFACE

TERMINAL INTERFACE

An application implemented with BUILDER normally presents the end-user with a screen environment. Each screen typically contains one or more blanks (variables) into which the user places information. The format of the screens is under the control of the creator of the application and is not dictated by BUILDER.

BUILDER is device independent and can be made to function on virtually any CRT that allows direct cursor addressing. When BUILDER is installed, a configuration file (BLDRTERM) is provided containing the information required to correctly interface with many different devices. If a device at your installation is not included, anyone with write access to the configuration file can create a new terminal type by entering the operational characteristics of the device through BUILDER into BLDRTERM. See Appendix B for further information.

BUILDER can operate in character, line, or block mode. The mode can be controlled on an installation-wide basis by adjusting the driver default. The choice of which mode to use will affect the user interface in a number of areas. Specifically, it will influence the use of function keys, editing capabilities, cursor control, and, to a certain extent, the design of the application. The trade-offs are discussed below.

Terminal Types

BUILDER is terminal independent. That is, any BUILDER application can work without modification on any CRT that has cursor addressing. However, BUILDER must know what type of terminal you are using in order to correctly use the capabilities of the terminal. Because of this, when an application is started, BUILDER will normally prompt with:

```
Terminal Type?
```

This is a request for you to enter the manufacturer and model of the terminal you are using. If you are unsure, type "HELP" and a list of possible terminal types will be displayed. If your particular terminal is not in the list, another by the same manufacturer often will work. This list can be extended by someone on your site (see Appendix B).

Once you have given BUILDER this information it will be remembered as long as you remain logged on. When you log off, the computer knows that you may have moved to another terminal and will ask you again when you log back in.

If your installation uses only one type of terminal, this prompt can be eliminated by including the command:

```
SETJCW TERMTYPE=number
```

in the BUILDER UDC definition. The number must be the position (starting at 1) of the desired terminal type in the list presented when "HELP" is typed in response to the "Terminal Type?" prompt.

Whenever you specify the terminal type, BUILDER initializes the MPE job control word (JCW) TERMTYPE to correspond to the proper terminal type. The terminal type will therefore be known until you log off. If a valid terminal type is entered which does not produce the proper results on the terminal, resetting the JCW to zero will cause BUILDER to prompt for the terminal type again when BUILDER is restarted.

If BUILDER is run in batch mode, the terminal type should be included in the batch job as data for BUILDER. Most likely the terminal type "JOB" should be used. See the Script Files Appendix for additional information.

Character Mode vs. Line Mode vs. Block Mode

When the terminal type is specified, BUILDER looks in the configuration table for that terminal type (see the Creating New Terminal Drivers Appendix) to determine which mode the application should be run in. It is possible to have drivers for each terminal in each mode by (perhaps) assigning a prefix or suffix to the device name or maintaining separate configuration files for drivers for each mode.

Character Mode Description:

Data is transmitted to the computer and processed by the application one character at a time. Each character is echoed to the terminal by BUILDER, rather than by the operating system. As each key is pressed, the character is transmitted to the computer. As the character is received by the computer, it is passed immediately to the application for processing. Most characters are simply echoed to the terminal and perform no special functions. However, certain characters can be assigned to perform useful actions (such as cursor positioning).

Character Mode Advantages:

- Immediate notification of errors upon completion of a variable.
- Function keys can be used for help easily, especially using the technique of help on the variable where the cursor resides ($VARIABLE).
- Enhancement attributes such as required/optional, blinking/inverse, etc., can be changed as other variables are completed.
- Additional processing or calculation can be initialized based upon the completion of a variable.
- Expensive block mode terminals are not required.
- Applications can be terminal independent.
- Special keys that generate one character are acted upon immediately without having to terminate the character sequence with a RETURN.
- Prohibitively expensive — and not fully functional — over packet-switching networks or protocols (for example, X.25).

Character Mode Disadvantages:

- Characters can be dropped on a busy system without type-ahead. This problem will decrease slightly as more users perform operations with BUILDER, and increase as the user is physically moved further away from the computer (as in the case of access through a modem, DS line, or packet switching network). Although dropped characters may not be a problem for a decision support system (because good control of the screen will provide proportionally better benefits), such character drop may be unacceptable to a data entry clerk.
- Special keys and function keys that emit more than one character (such as arrow keys on HP terminals and function keys on most terminals) cannot always be processed correctly. However, if function keys cannot be used, function sections can still be processed by using the ESCAPE key (described in the next section).

Two-character Mode Description:

Data is transmitted to the computer one character at a time. Each character is echoed to the terminal by the operating system rather than by BUILDER. The computer saves each character until two are received, at which point both characters are passed to the application for processing.

Two-Character Advantages:

- All of the advantages of Character Mode, plus:
- Better handles special keys which generate two characters (such as arrow keys and function keys on HP terminals).

Two-Character Disadvantages:

- Characters are processed in pairs, so a special key which generates t w o characters pressed after typing a single character will transmit only the first character of the special key. (See the section on Hewlett-Packard terminals.)

Line Mode Description:

Data is transmitted to the computer one character at a time but BUILDER processes one variable at a time. The operating system echoes the characters to the terminal and saves the characters until the "end of a line" is signalled by the receipt of a RETURN, TAB (control-I), Function key, or the filling of a variable on a screen. The saved characters are then passed to the application for processing.

Line Mode Advantages:

- Most of the advantages of character mode, plus:
- No dropping of characters on a busy system.
- Function keys which generate more than one character but end in a RETURN (such as on HP terminals) work well.

Line Mode Disadvantages:

- Special keys can be used but require a RETURN or other line-completion key to be pressed after the special key is pressed to indicate "end of line" so that the key can be processed. Since the operating system echoes these characters to the terminal but BUILDER doesn't process them until a RETURN is received, the action may appear to take place immediately (such as for an arrow key), but in fact BUILDER will know nothing of its movement. In addition, in some cases this movement can result in an invalid situation, such as the placement of the cursor over a label, and may generate an error or other response when TAB or RETURN is pressed, although BUILDER will generally place the cursor in the proper position.

- Special keys which generate more than one character (such as arrow keys on HP terminals) can be used in line mode. However, since BUILDER reads a variable at a time, a portion of the sequence of an arrow key could terminate a variable and leave the system in a mode in which it has not read the remaining portion of the key sequence, resulting in the prompt "Which Function Key?".

Block Mode Description:

Data is transmitted between the computer and the terminal in blocks. The individual characters are saved at the terminal until the user presses the ENTER key or a Function key. As the application processes the data from the terminal, it "saves" the answer in a block. When the application is ready to transmit back to the terminal, the entire block is transmitted.

Block Mode Advantages:

- User can move over the screen using any special cursor positioning and editing keys before sending the screen to BUILDER with the ENTER key or a Function key.
- Transmission is in blocks; there are fewer I/O interrupts for the computer and transmission is therefore also considerably less expensive over packet-switching networks or protocols (for example, X.25).

Block Mode Disadvantages:

- There is no notification of errors until the entire screen is sent.
- There is no data validation until the entire screen is sent and processed.
- Only one error will be returned with each sending, so multiple errors on a screen will require multiple sendings (and processing) of the full screen.
- Enhancement attributes such as required/optional, inverse/blinking, etc., cannot be changed as variables are completed.
- Additional processing or calculation cannot be initiated when a variable is filled.
- Variable sections will be processed, but not until the full screen is sent.
- Block mode terminals, which are generally more expensive, are required.
- Variables declared as POSITION will allow the user to type new values over the existing information. However, BUILDER will remember the true value and replace it whenever the screen is redrawn.
- POS_ENHANCE and TAB=CLEAR are ignored.
- JUSTIFY, BLANK, and UPPER are not processed until the screen is sent.
- Debug mode cannot be entered by pressing "Escape / Return." However, for debugging purposes, a function key with a deliberate syntax error in it can be used to force BUILDER into debug mode.
- All PROMPTs are done in CHARACTER or LINE mode, so the user must press RETURN in response to PROMPTs instead of ENTER.
- PROMPTs are also the only places in which the user can type control or escape sequences that BUILDER will recognize.
- Applications which are designed for use in character or line mode will not always function well in block mode, for the above reasons.

Hewlett-Packard Terminals:

Hewlett-Packard terminals use two characters for special keys. The first character is usually Escape. This is an unprintable character but still is transmitted to the computer. The second character is an upper or lowercase character. Therefore, each special key transmits at least two characters to the computer.

BUILDER assumes that, when it receives an Escape character by itself, the user wants to initiate a function key sequence. Therefore, in line mode, the user is prompted for "Which Function key?".

If the cursor is at the last position of a variable on the screen, there is not enough room for all of the characters of the function key or special key sequence to be transmitted. Only the first character, usually an Escape, will go. Again, BUILDER thinks the user is beginning a function key sequence and will prompt "Which Function key?" Pressing the number of the function key desired (eg: type "1" for function key 1) completes the request properly.

In two-character mode, every two characters are processed, so the second character of the special key is processed along with the Escape.

In line mode, a RETURN key must be pressed after using the special key to complete the group of characters sent to the computer. Function keys can be loaded to send the RETURN needed.

BUILDER provides several generic HP terminal types in the terminal configurator:

HPFKEY	Line mode
HPCHARFKEY	2-Character mode
HPBLOCK	Block mode

Note that HP2640 terminals are not supported in block mode.

Special Keys

Some special keys found on virtually all terminals are used by BUILDER:

TAB	Skips to the next variable on the screen.
BACKSPACE, DELETE	Deletes the previous character. DELETE will only work on Block Mode terminals.
ESCAPE	Normally begins a function key sequence. If the terminal has function keys, they can usually be used. For more information, see the Function Key section below.
RETURN/ENTER	Indicates that either RETURN or ENTER should be pressed, depending on whether or not Block Mode is being used. See below.
RETURN	Completes a screen in character or line mode. Also used to terminate a response to a PROMPT.
ENTER	Completes a screen in block mode. Do not use this key if operating in character or line mode as the result is unpredictable.

Other keys on the terminal may perform special functions depending on the actual configuration of the device. The following functions may be available on your terminal; if they are available, they may use different keys depending on the terminal type. To determine what keys perform these functions on a particular terminal, use the Control-H function (H for help). Press the escape key, then hold the control key down and press the H key.

The following special keys can be defined:

BACKTAB	The cursor is positioned at the beginning of the previous variable. Normally, this is a control-B.
HOME	The cursor is positioned at the first variable on the screen.
UP, DOWN	The cursor is positioned at the beginning of the closest variable up or down from the current variable. Normally, a control-U and control-D are used.
LEFT, RIGHT	The cursor is moved right or left one character within a variable. If the cursor is moved out of the current variable, the cursor is positioned to the previous or next variable on the screen. Normally, a control-L and control-R are used.
REDRAW	The screen is cleared and then redrawn. Normally, a control-Z is used.

These special functions may correspond directly to the cursor control (arrow) keys found on most CRTs. If the application is in character mode and the arrow keys emit a single character (as in the case of TeleVideo terminals), the arrow keys can be used. The keys may also be used in line mode but, as mentioned above, BUILDER will not know the true location (and the validity of the location) until TAB or RETURN is pressed.

Single character cursor positioning can be simulated effectively by the user actually keying, for example, a control-B to backtab. This works quite well in character mode. In line mode, the control-B would need to be followed by a TAB or RETURN in order to have the request processed.

Function Keys

Screen-based applications often use special "function" keys that appear on many terminals. Function keys are often labeled F1, F2, F3,... or PF1, PF2, PF3,... and a particular terminal may have from zero to eight or more function keys. A number of inexpensive CRTs do not have function keys so a special method has been developed to provide function key capability for these terminals. The method selected to emulate function keys is called escape functions and will work on any terminal.

All terminals have a special key called the escape key. It is labeled ESCAPE or ESC or sometimes ALT or ALTMODE. If the user presses the escape key, the prompt "Which Function key?" will appear at the bottom of the screen. The ESCAPE thus serves to get the application's "attention" and denotes the beginning of a special function. Most built-in function keys also begin with an escape (and as mentioned above, may or may not be useful depending on the operating mode of the application). The response to the prompt is the function key value. The application can be designed to perform special processing when the key is pressed.

If the application is being run in line mode, the actual function keys on the terminal can often be used. In this case, the configuration of the terminal will determine what function the application actually sees. This allows a number of different terminals, which emit different function key sequences, to operate the same application. For example, HP terminals normally emit an ESCAPE "p" RETURN when F1 is pressed. This can be translated in the terminal driver to yield the function "1". The function "1" may be obtained on a terminal without function keys by pressing ESCAPE, "1".

On some terminals, the ESCAPE 1 sequence must be followed by RETURN. On some terminals, the ESCAPE itself must be followed by RETURN before the 1 is pressed.

On some terminals, pressing a function key or cursor positioning key causes the prompt "Which Function Key?" to be given if one of the following cases exist:

a) the cursor is positioned in a variable of LENGTH=1,
b) the cursor is positioned on the right-most character of a variable on the screen.
c) A special key (function key, arrow key, etc) which generates a sequence of more than one character and begins with an ESCAPE has failed to send the entire sequence to the computer (see detailed description under Two-Character and Line Mode).

If a function key was pressed, type the value of the function key desired (e.g.: "1" for Function 1). If a cursor positioning key was pressed, press RETURN to clear the prompt from the screen.

SECTION 3

APPLICATION BUILDER ENVIRONMENT

BUILDER ENVIRONMENT

BUILDER is a program that interprets and acts on the contents of an application file. An application file is an EDITOR or QEDIT file containing screen section delimiters and commands. BUILDER uses the RELATE/3000 Host Language Interface (HLI) routines to execute RELATE commands.

Creator Mode vs. User Mode

BUILDER runs in one of two modes: creator or user. When an application is being developed, BUILDER should be run in creator mode. After the application has been completed, BUILDER should be run in user mode.

The difference between creator mode and user mode rests primarily in the interpreter's response to errors. If an error occurs in creator mode, the application file's name and the line number that caused the error are displayed. BUILDER then enters DEBUG mode. If an error occurs in user mode, the error is displayed in much the same way; however, the user is not placed into DEBUG mode. Instead, if the error is in the INITIAL section of the application, the application is terminated. If the error is in the INITIAL section of a subsequent screen, the original screen is redisplayed. If the error is in another section, the section is terminated. An error in user mode is generally caused by insufficient data validation in the application or a failure to anticipate a processing problem. An error in creator mode is usually a logical or typographic error in the application.

When BUILDER enters DEBUG mode (which is not MPE's DEBUG mode), the creator is given a number of options. These options include going into a simulated RELATE Command Interpreter or invoking an editor.

If the creator goes into the Command Interpreter, the environment is not changed. Thus, BUILDER commands can be issued to determine the state of the application and perhaps correct the problem without actually terminating the test. Upon exiting the Command Interpreter the application can be resumed. If the error is more severe, the creator may invoke an editor to adjust the application file and restart the test.

In creator mode, the user may invoke the editor or the RELATE Command Interpreter at any time by entering a special escape function. This option is not available if the application is running in user mode. A more detailed description of the capabilities of DEBUG mode is given later in this section.

Running Builder

The UDC "BUILDER" is supplied by CRI and should be added to the commands available at the user's site during the installation of the product. The UDC runs BUILDER in creator mode. Hence, to run BUILDER in creator mode, when the operating system prompt appears, the user can type:

```
        BUILDER
```

In addition, the name of the application file to use can be included on the same command line; for example, for an application file called APFILE:

```
        BUILDER APFILE
```

If you wish to set up your own UDC, creator mode is enabled by the alternate entry "CREATOR". If this entry point is not specified on the RUN command, BUILDER is run in user mode. The entry point "ENDUSER" can be used to emphasize that BUILDER should run in end-user mode. For example, include this line in your UDC to run BUILDER in creator mode:

```
       RUN BUILDER.PUB.RELATE45,CREATOR
```

If the application file name is not specified on the BUILDER command line, BUILDER will execute the application found in the file BLDRAPP. BUILDER will attempt to open this file when it begins execution. If the file cannot be opened, BUILDER will prompt with "Application File?". A valid application file name must then be given.

The location, and hence content, of an application file can be hidden by creating a NOHELP, NOBREAK UDC to invoke the application. The command sequence would be similar to the following:

```
    AP
    OPTION NOHELP,NOBREAK
    FILE BLDRAPP=APSYS/BIGBOSS.GROUP.ACCOUNT
    CONTINUE
    RUN BUILDER.PUB.RELATE45,ENDUSER
    RESET BLDRAPP
```

This sequence assumes the application exists in the file APSYS, which is protected by the lockword BIGBOSS, and that BUILDER has been installed in PUB.RELATE45. The CONTINUE command will cause the UDC to process the RESET command (and hence eliminate the reference to the file containing the application) even if BUILDER terminates abnormally.

APPLICATION FILES

BUILDER application files are normally either editor files created with HP's EDITOR or QEDIT files created with QEDIT, an editor for the HP/3000 by ROBELLE Inc. Any program that creates compatible files may also be used to create application files.

An application file is divided into screens and each screen is divided into sections. Each section contains information defining an aspect of the screen. Screens and screen sections are separated from each other within the application file by delimiter lines. Delimiter lines contain special characters at the beginning of the line. Any characters may be used but they must be left justified and must be followed, after one or more spaces, by a section name. The first line in an application file defines the delimiter characters for that application file.

A screen may contain the section names shown below. In this case the delimiter lines contain three asterisks.

```
*** ACTION string
*** DECLARATIONS [STATE=statename]
*** ENTER [STATE=statename]
*** FUNCTION [character] [;LABEL="text"] [;STATE=statename]
*** GLOBAL [LINEDRAW=linechar]
*** INITIAL EVERYTIME
*** INITIAL [STATE=statename]
*** LAYOUT [LINEDRAW=linechar]
*** SCREEN screenname
*** TIMEOUT
*** VARIABLE [variablename]
```

All sections need not appear on every screen and the order in which the sections appear is not important except for the SCREEN section and the ACTION sections. The screen delimiter line marks the beginning of a new screen and must appear first.

EXAMPLE:

```
*** SCREEN FIRSTSCREEN
*** LAYOUT
    (screen layout definition)
*** DECLARATIONS
    (local variables used in screen definition)
*** FUNCTION 1
    (commands executed if function 1 is pressed)
*** ENTER
    (commands executed when the screen is completed)
*** SCREEN SECONDSCREEN
*** LAYOUT
    (layout of second screen definition)
*** ENTER
    (commands executed when the screen is completed)
```

ACTION Section

delimiters ACTION string

Contains commands that are executed when a particular action is requested.

string	Required. The string which is to be matched to the contents of the ACTION variable when RETURN/ENTER is pressed. The string which names an action section can be terminated with an atsign ("@"). The atsign indicates that characters to the right of the atsign position in the action variable should be ignored when matching the variable. The absence of an atsign indicates that an exact match is required. Thus, naming an action section "H" requires that "H" and nothing else be placed into the variable. If, however, the section is named "H@", any text in the action variable which begins with an "H" will match.

When RETURN/ENTER is pressed, BUILDER checks the contents of the variable that is declared in the DECLARATION section to be the ACTION variable. If an action variable is not declared in the current STATE, ACTION sections will have no effect. The content of the action variable is matched to the ACTION section names to determine which section should be executed. If a match cannot be found, the ENTER section is executed. BUILDER reads sequentially through the SCREEN for the matching ACTION section. The first one it finds that matches will be the section executed.

EXAMPLES:

```
*** LAYOUT
    ENTER CHOICE AND PRESS RETURN
                [CHOICE]
             1. ADD
             2. DELETE
             3. PRINT

*** DECLARATION
    CHOICE:ACTION:RANGE=1/3
*** ACTION 1
    CALL SCREEN ADD
*** ACTION 2
    CALL SCREEN DELETE
*** ACTION 3
    PRINT
```

delimiters DECLARATIONS [STATE=statename]

Specifies options for the variables defined on the screen layout and creates temporary variables that can be used in computation but which are not part of the layout.

STATE	Optional. When the SET STATE command is executed, the declarations within the appropriate DECLARATION section are used to modify existing variables. This can be used to change fields from optional to required, change enhancements, set initial values, etc. The NAME, LENGTH, ARRAY or GLOBAL options of a variable may not be changed in a DECLARATION section containing a statename. State names must start with a letter, contain only letters, numbers, and underscores ("_"), and be no more than 15 characters long.

All variables used in a screen must appear in the LAYOUT or be declared in a DECLARATION section that has no STATE. Lines in the DECLARATION section consist of a list of variable names separated by commas (",") optionally followed by a semicolon (";") and a list of variable options separated by semicolons. Declaration lines may begin in any column and may be continued on additional lines by terminating each line with an ampersand ("&"). Nothing other than declaration lines or blank lines may appear in the DECLARATION section. The double dash ("--") may be used on any line in the DECLARATION section to indicate that the rest of the line is comment.

If no options are specified, the variable defaults to an OPTIONAL ALPHABETIC variable. If the variable appears in the LAYOUT, its default LENGTH will be its length in the LAYOUT; otherwise, it will default to LENGTH=10.

All the variables in the list are assigned the options specified in the option list. Each line of the DECLARATION section consists of:

```
                 variablename[,variablename[...]]
      [;ALPHA | NUMERIC[;DECIMAL=number] | DATE[="format"]]
          [;OPTIONAL | REQUIRED | DISPLAY | POSITION]
                          [;ACTION]
                     [;ARRAY=elements]
               [;ARRAYORDER=ACROSS | DOWN]
                     [;BLANK[=YES | NO]]
                    [;DEBLANK[=YES | NO]]
               [;ECHO[=YES | NO | ON | OFF]]
                    [;ENHANCE=enhlist]
                         [;GLOBAL]
                     [;INITIAL=value]
         [;JUSTIFY=LEFT | RIGHT | CENTER | NONE]
                     [;LENGTH=length]
                     [;NAME=newname]
                  [;POS_ENHANCE=enhlist]
                    [;RANGE=rangelist]
                  [;SUB_ENHANCE=enhlist]
                   [;TAB=CLEAR | SKIP]
                    [;UPPER[=YES | NO]]
```

data type	Optional. May be one of ALPHA, NUMERIC, or DATE. If not specified, the default is ALPHA. ALPHA: The variable may contain any printable character. NUMERIC: The variable may only contain numeric values. DATE[="dateformat"]: The variable represents a date in the given format or, if a format is not given, in the current SET OPTION DATE format. The format must be a valid RELATE date format enclosed in quotes. Dates are accepted and displayed in the indicated format.
DECIMAL=number	This keyword can only be used on NUMERIC variables. Number must be a positive integer referring to the number of decimal positions to be displayed in the variable. If, instead of a number, FLOAT is specified, the maximum decimal positions allowed by the size of the variable are displayed. The value actually displayed is the value used in calculations and for updating files. The default is FLOAT.
variable type	Optional. May be one of OPTIONAL, REQUIRED, DISPLAY, or POSITION. If not specified, OPTIONAL is the default. This attribute applies to the variable only in the current screen; it is not carried from one screen to another. OPTIONAL: The variable is optional. A value need not be placed into the variable. REQUIRED: The variable is required. If an automatic VERIFY takes place (which can be turned off with the SET OPTION VERIFY command), if the VERIFY command is executed, or if the value of the variable is changed, an error is generated unless data is entered. DISPLAY: The variable is display only. The application can place data into the variable but the user cannot position the cursor onto the variable. POSITION: The variable is treated as a DISPLAY variable but the cursor may be positioned to the variable.
ACTION	The variable is an ACTION variable. Only one variable on each screen can be declared as an action variable. When RETURN/ENTER is pressed, the value of the action variable determines which ACTION section should be executed. If STATE names appear on any DECLARATION sections, any SET STATE command to a state matching one of these sections will cause the variable to no longer be an ACTION variable. A screen need not have an action variable.
ARRAY=elements	Indicates that the variable is to be treated as an array variable containing the specified number of elements. If this is not specified, the number of elements is determined by the LAYOUT

of the current screen. This keyword is ignored if it specifies fewer elements than appear on the screen. If more are specified than are in the LAYOUT, the larger number will be used. (For further information, see the ARRAY VARIABLES commentary later in this section.) This keyword cannot be specified if the DECLARATION delimiter line contains a state name.

ARRAYORDER=ACROSS|DOWN	Indicates if the array variable should be assigned subscripts left to right (across) or top to bottom (down). By default, array elements are assigned across.
BLANK[=YES|NO]	If the value of the variable is zero, it will be displayed and processed as a blank. If BLANK is specified without YES or NO, YES is assumed. The default if BLANK is not specified is NO.
DEBLANK[=YES|NO]	If DEBLANK=NO, leading and trailing blanks are not removed when the variable is used in the substitution algorithm. Enough blank space must be left in the line at the location of the substituted variable to accomodate the full length of the variable, or the error "NOT ENOUGH WHITE SPACE" may be encountered. If DEBLANK is specified without YES or NO, YES is assumed. The default if DEBLANK is not specified is YES.
ECHO[=YES|NO|ON|OFF]	If ECHO=NO, any values assigned or typed into the variable, on the screen or in response to a PROMPT, will not be displayed. If YES or NO is not specified, YES is assumed. The default is YES.
ENHANCE=enhlist	This specifies the variable's display enhancement. By default, a variable's enhancement is determined by its display option (either OPTIONAL, REQUIRED, DISPLAY, or POSITION). If the default is not appropriate, the enhancement may be changed. Enhlist should contain one or more of the keywords (or their abbreviations) BLINKING, HALFBRIGHT, INVERSE, UNDERLINE, or NONE separated by commas. If NONE is specified, none of the other enhancements can be specified. If SUB_ENHANCE is also used in the same command or DECLARATION, whichever is specified last takes precedence.
GLOBAL	Indicates that the variable has been defined on a screen above the current screen and that a new variable should not be created. By default, a variable included in the DECLARATION section or in the LAYOUT is assumed to represent a new variable in the absence of this keyword. The keyword causes all attributes of the variable to be duplicated in the current screen. The attributes can selectively be overridden by inclusion of the declarative keywords. This keyword cannot be specified if the DECLARATION delimiter line contains a state name. Read the section on GLOBAL VARIABLES for more detailed information.

INITIAL=value	This specifies the value to be assigned to the variable whenever the INITIAL section or appropriate SET STATE command is executed. The value may be a quoted string or a numeric constant. The value will be displayed when the screen is initially drawn (or redrawn).
JUSTIFY=just	Indicates the justification of data within the variable. The justification must be one of: LEFT, RIGHT, CENTER, or NONE. The default is LEFT. The variable is justified when data is changed on a screen or assigned by an assignment statement.
LENGTH=length	The number of characters available for the variable is specified. A variable may not contain more than 1500 characters. If this keyword is specified for a variable that appears in the LAYOUT, earlier in the same DECLARATION, or in a DECLARATION section with a state name, the length must match the originally calculated length, or an error will result.
NAME=newname	The variable name is changed to the name specified. The new name should be used when the variable is referred to in the other sections. This keyword cannot be specified if more than one variablename is specified on that line or if the DECLARATION delimiter line contains a state name.
POS_ENHANCE=enhlist	This will cause the variable to change enhancements to the indicated list whenever the cursor is positioned on the variable. The enhlist may contain the same items described in ENHANCE.
RANGE=rangelist	Requires that any value entered into the variable be one of the legal values specified in the rangelist. Rangelist is a list of ranges separated by commas. A range is either: a single number, a quoted string, or a range specified: [low] / [high]. Either low or high may be absent. Low or high may be numbers or quoted strings. If the rangelist consists entirely of individual quoted strings then BUILDER will match the user's input with the rangelist until a partial match is found. If an unambiguous partial match is found the variable will be set to the complete value of the response.
SUB_ENHANCE=enhlist	Specifies the display enhancements of the variable at the current subscript. The enhlist may contain the same items described in ENHANCE.

If used in the DECLARATION section, this will be the first element of an array, or the variable itself for a non-array variable. If ENHANCE is also specified, whichever enhancement was specified last in the DECLARATION will take precedence.

Used in a MODIFY VARIABLE command, non-array variables are considered to have a subscript of zero.

TAB=CLEAR|SKIP	Indicates the action BUILDER should take when a change has been made to a variable and the TAB or RETURN/ENTER key is struck. If CLEAR is specified, data in the variable beyond the cursor is blanked. If SKIP is specified, the data is not blanked. The default is SKIP for LEFT justified variables and CLEAR for CENTER or RIGHT justified variables and dates.
UPPER[=YES|NO]	All letters placed into the variable are converted to uppercase before being used. The keyword cannot be specified with NUMERIC. If UPPER is specified without YES or NO, YES is assumed. The default if UPPER is not specified is NO.

EXAMPLES:

```
*** LAYOUT

    EMPLOYEE NAME:[ENAME                                    ]

    SEX:[S]    AGE:[age]      DATE HIRED:[hdate     ]

*** DECLARATIONS
    ENAME,S,AGE;REQUIRED
    S              ;NAME=SEX_CODE;RANGE="M","F"
    AGE            ;NUMERIC;RANGE=1/120
    HDATE          ;DATE="M/D/Y"
    ANSWER         ;LENGTH=1;UPPER;RANGE="Y","N";REQUIRED

*** ENTER
    IF SEX_CODE="M"
       PROMPT "DO THEY OWN A CAR?", ANSWER
    ENDIF
```

delimiters ENTER [STATE=statename]

Contains commands that are executed, if an executable ACTION section isn't found, when the user indicates that data entry is complete by pressing the RETURN/ENTER key.

STATE	Optional. Depending upon the value of the current state the appropriate ENTER section will be executed when the user presses RETURN/ENTER. If no ENTER section is found with a matching state name then the ENTER section without a state name will be executed. State names must start with a letter, contain only letters, numbers, and underscores ("_"), and be no more than 15 characters long.

Normally this section contains the majority of processing to be performed by the screen. The ENTER key (which exists on some terminals) should only be used if operating in Block Mode.

EXAMPLES:

```
*** LAYOUT
   [cno   ] [name    ] [address       ]

   TYPE IN NEW DATA AND PRESS RETURN TO ADD
*** ENTER
   IGNORE ALL ERRORS
   RECORD ADD
   IF $ERROR
      FAIL $ERRMSG
   ELSE
      DISPLAY "RECORD ADDED."
   ENDIF
```

delimiters FUNCTION [character] [;LABEL="string"] [;STATE=statename]

FUNCTION sections contain commands that are executed in response to function keys.

character	Required. Valid characters are uppercase A through Z (a lowercase character entered as a function will cause the uppercase function to execute), the numbers 0 through 9, and any special character except semicolon (";"), double quote ("""), comma (","), period ("."), and slash ("/"), which are special functions defined in BUILDER. When a function key is struck, the section corresponding to the key is executed. All but one FUNCTION section must be named with a single character. See the Creating and Modifying Terminal Drivers Appendix for information on how to define F10 and above.
LABEL="text"	Optional. If specified, the LABEL keyword must be followed by text within quotes. This text will be used if the terminal has function key labels. If the text contains a vertical bar ("|"), the text before the bar will appear on the first line of the function key label and the text after the bar will appear on the second line or be disregarded. If no LABEL is specified and the terminal has function key labels, all functions defined in the screen will be labeled as "F1", "F2", etc.. If the terminal doesn't have function key labels, the LABEL may be displayed on the message line when it is not otherwise being used. See the Creating Terminal Drivers Appendix for further information.
STATE	Optional. Depending upon the value of the current state the appropriate FUNCTION section will be executed when the user presses a function key. If no FUNCTION section is found for that key with a matching state name, then the FUNCTION section without a statename is executed, if one exists. If a matching local function without a statename does not exist then the matching function in the GLOBAL section will be executed. Functions in the GLOBAL section may not have STATES. State names must start with a letter, contain only letters, numbers, and underscores ("_"), and be no more than 15 characters long.

The number of the function key struck is placed into the built-in variable $KEY by BUILDER. If the terminal does not have function keys, the ESCAPE key followed by a character (and RETURN in line mode) will also cause the FUNCTION sections to execute.

If an unnamed FUNCTION section exists, it will be executed if a function key is pressed which does not have a matching FUNCTION section.

EXAMPLES:

```
*** FUNCTION 1; LABEL= "READ|NEXT"
    RECORD READ
    IF $EOF
       RECORD REWIND
       RECORD READ
    ENDIF
*** LAYOUT
    CUST:[cnum]
*** FUNCTION 8; LABEL="EXIT"
    NOTE If I press F8,BUILDER will give message
    NOTE "Required variable.  Must have value.".
    NOTE Use the SET OPTION VERIFY command to fix this.
    RETURN SCREEN
```

delimiters GLOBAL [LINEDRAW=linechar]

Defines INITIAL, DECLARATION, TIMEOUT, and FUNCTION sections available to the entire application.

LINEDRAW	Optional. If specified, a character must be specified to indicate that the terminal's linedrawing set will be used. Valid characters are |, !, @, #, $, ^, *, –, =, +, ~, /, and ". This character, when placed vertically or horizontally adjacent to another LINEDRAW character anywhere in a LAYOUT section, will result in a line on the screen. A minimum of two adjacent characters are required to draw a line. Single characters will not be altered. If run on a terminal without linedrawing capability, each LINEDRAW character will be replaced with a space. The character used to begin delimiter lines should not be used as the LINEDRAW character. This LINEDRAW character will be in effect for every LAYOUT section in the application unless LINEDRAW is used explicitly for that LAYOUT section.

This section is similar to a SCREEN section. Any sections inside a GLOBAL section are treated as global to the entire application. The GLOBAL section may contain a DECLARATION section to declare global variables. The GLOBAL section may contain an INITIAL section that is executed once at the start of the application. The GLOBAL section may also contain FUNCTION sections to declare global function keys. These function keys can be executed from any screen in the application unless superceded by a FUNCTION section of the same name explicitly declared in a screen.

If a TIMEOUT section is included, it will be used for any screen that times out if the screen doesn't have a TIMEOUT section of its own.

The GLOBAL section must appear in the first application file of your application. There may only be one GLOBAL section in the application.

EXAMPLES:

```
*** GLOBAL
*** DECLARATION
    ANSWER; LENGTH=1; UPPER; RANGE="Y","N"

*** FUNCTION 7; LABEL="PREVIOUS|SCREEN"
    PROMPT "OK TO EXIT?",ANSWER
    IF ANSWER="Y"
       EXIT
    ENDIF

*** SCREEN MAIN
***     ....body of first screen....
```

INITIAL Sections

delimiters INITIAL [STATE=statename]
delimiters INITIAL EVERYTIME

These sections contain commands that are executed before the screen is displayed.

STATE	Optional. Indicates that this section contains commands that will be executed when a SET STATE command with this name is executed. State names must start with a letter, contain only letters, numbers, and underscores ("_"), and be no more than 15 characters long.

The INITIAL section without a STATE name will be executed any time a CALL SCREEN or SET SCREEN to the screen containing the INITIAL section is performed, or, if the screen is the first screen in the application, when the application begins.

Assignment of defaults and other initialization tasks can be performed in these sections. When the screen is displayed, any changes made in this section will appear. A REFRESH command can be included in the INITIAL section to force the screen to be displayed before the section is completed.

The INITIAL EVERYTIME section is executed after the INITIAL section and again before the screen is redisplayed after any FUNCTION, ENTER, or ACTION section has been executed. See the Execution Processing Loop discussion later in this section for a more detailed description.

EXAMPLES:

```
*** INITIAL
    OPEN FILE CUST
    SET INDEX CNO
    CREATE PARTITION MAIN_INVOICE
    OPEN FILE INVMAS
    SET INDEX CNO
    SET PARTITION DEFAULT
*** INITIAL EVERYTIME
    CHOICE:=""
    TIME:=$TIME
```

delimiters LAYOUT [LINEDRAW=linechar]

Contains the format of the screen.

LINEDRAW	Optional. If specified, a character must be specified to indicate that the terminal's linedrawing set will be used. Valid characters are |, !, @, #, $, ^, *, –, =, +, ~, /, and ". This character, when placed vertically or horizontally adjacent to another LINEDRAW character anywhere in the LAYOUT section, will result in a line on the screen. A minimum of two adjacent characters are required to draw a line. Single characters will not be altered. If run on a terminal without linedrawing capability, each LINEDRAW character will be replaced with a space. The character used to begin delimiter lines should not be used as the LINEDRAW character.

Each line between the LAYOUT section delimiter line and the next delimiter line encountered in the application file becomes a line on the screen. The layout should be typed into the editor file as it should appear on the screen.

Variables are defined by surrounding their location with brackets and placing the name of the variable between the brackets ([variablename]). Brackets will be displayed as blanks when BUILDER displays the layout on the screen. There is no way to have a square bracket appear in the final layout. The length of the variable does not include the brackets. A variable name cannot exceed 15 characters, must start with a letter, and may only contain letters, digits, and underscores ("_"). BUILDER follows RELATE conventions when dealing with IMAGE data item names.

The last line of the screen is reserved as the message line and should not be used as part of the layout. If the layout consists of more lines than can fit on a screen, the extra lines are ignored. If the layout is wider than the screen, the extra characters are ignored.

EXAMPLES:

```
*** LAYOUT LINEDRAW=+
+++++++++++++++++++++++++++++++++++++++++++++++++
+      INVENTORY  ADDITION                      +
+++++++++++++++++++++++++++++++++++++++++++++++++
+  PART NUMBER:          [part_num      ]       +
+  QUANTITY RECEIVED:    [qty]                  +
+  DATE RECEIVED:        [rdate    ]            +
+  MANUFACTURER NUMBER:[mno      ]              +
+++++++++++++++++++++++++++++++++++++++++++++++++
+ PRESS RETURN TO ADD; PRESS F1 FOR PREVIOUS SCREEN  +
+++++++++++++++++++++++++++++++++++++++++++++++++
```

delimiters SCREEN screenname

Defines the beginning of a new screen.

screenname	Required. A screen name must be specified. A screen name must not exceed 15 characters in length, must start with a letter and contain only letters, digits, and underscores ("_"). A screen name cannot be duplicated in the same application. This name can be referenced by CALL SCREEN and SET SCREEN commands.

Anything between the screen delimiter line and the next delimiter line is ignored by BUILDER. This area can be used for comments.

delimiters TIMEOUT

Contains commands that are executed when the number of seconds indicated with the SET OPTION TIMEOUT command have passed with no user input.

This will occur at any time a BUILDER application is waiting for a user response, including PROMPTS, except while in DEBUG mode. A screen with no variables in the LAYOUT section will not time out unless a PROMPT occurs.

The TIMEOUT section applies only to the current screen, not to any screens called by this screen.

EXAMPLES:

```
*** INITIAL
   SET OPTION TIMEOUT=10
*** TIMEOUT
   DISPLAY "PLEASE DO SOMETHING."
```

delimiters VARIABLE [variablename]

Contains commands that are executed when the value of the indicated variable is changed by the user.

variablename	Optional. The name of a variable in the application with which this sequence of actions should be associated. All but one VARIABLE section must be named.

When a variable is changed by the user, the corresponding VARIABLE section is executed. If a variable is used in a FAIL command the appropriate variable section will also subsequently execute when the screen is completed. If a VARIABLE section does not exist for the variable, the unnamed VARIABLE section is executed. If an unnamed VARIABLE section does not exist, no actions are taken.

EXAMPLES:

```
*** LAYOUT

   PART:[partnum] MANUFACTURER:[mnum] PRICE:[price]

*** DECLARATION
   OLD_PRICE;NUMERIC

*** VARIABLE PARTNUM
   RECORD POINT USING PART_MAST
   IF NOT $FOUND
      FAIL "No such part number."
   ELSE
      RECORD READ
   ENDIF

*** VARIABLE PRICE
   IF PRICE>OLD_PRICE
      DISPLAY "Cost has increased."
   ENDIF
```

ARRAY VARIABLES

The creator of an application may wish to design a screen that displays data from a database file. This can easily be done by creating variables on the screen with the same name as fields from the file and issuing a RECORD READ command to move data from the file to the screen. However, sometimes it may be desirable to place several records from a file onto the same screen. In this case, several variables on the screen must all have the same name as a single field in the database file. This situation is handled by array variables.

Definition

An array variable is a collection of variables that all have the same variable name. If a screen contains one or more array variables then the screen is called an array screen. If a variable appears three times on a screen, we say that there are three elements of the array variable on the screen. The number of elements in an array variable is normally determined by the number of times the variable appears in the LAYOUT, but can be adjusted upwards with the ARRAY keyword in the DECLARATION section.

Subscript Numbers

Each element of an array variable is assigned a subscript number starting at zero. The subscript numbers are assigned left to right and top to bottom on the screen automatically by the screen system unless specified otherwise in the DECLARATIONS section using the ARRAYORDER keyword. Most operations performed on variables will occur only on one element of the array at a time. Which element is determined by the $SUBSCRIPT built-in variable.

$SUBSCRIPT

At the beginning of the execution of a section, the current value of $SUBSCRIPT will be the same as the variable where the cursor was positioned before execution of the section. The current subscript may then be changed by an assignment to $SUBSCRIPT or by using the ARRAY loop command.

If $SUBSCRIPT is assigned a value that is higher than the number of elements in the array, and an array variable is used, BUILDER will wrap around the array. For example, in a 3-element array, the first element will be used whenever the value of $SUBSCRIPT is 0, 3, 6, 9, and so on.

ARRAY/ENDARRAY Loops

The ARRAY command is used to loop through a sequence of commands once for each defined array element. The array command consists of an ARRAY statement that begins the loop and an ENDARRAY statement that ends the loop. All commands within an array loop are executed once with a different array element each time through the loop. An ARRAY loop is terminated at the ENDARRAY command when a reference is made during the loop to a variable in the last element of any array or when no array element is referenced within the loop. In addition, the EXITARRAY command can be used to terminate a loop early, but it is recommended that this be avoided, if possible, since BUILDER's array processing automatically handles many common array processing situations. The last element of the array is not determined by the number of elements in the layout, but by the definition of the variable.

If an assignment is made within an ARRAY loop to $SUBSCRIPT, it will affect all subsequent commands that use array variables until an ENDARRAY command is executed. In other words, an assignment can be made to $SUBSCRIPT within an ARRAY/ENDARRAY loop if desired, but the change will only affect commands until the ENDARRAY is reached, at which point the loop will continue as if no assignment had been made.

EXAMPLES:

For example, consider the screen:

```
*** LAYOUT
                         DISPLAY RECORDS IN PARTS FILE

PART NUMBER    DESCRIPTION                     COST    QUANTITY

[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]
[PARTNO    ]   [DESCR                     ]    [COST ] [QTY     ]

*** ENTER
   ARRAY
      RECORD READ
   ENDARRAY
```

The preceding layout will display one record in each array element on the screen. There are four array variables on the screen: PARTNO, DESCR, COST, QTY. Each array variable contains eight elements. Eight database records, therefore, can be displayed on the screen at one time.

Array DECLARATIONS

If an array variable is REQUIRED and all variables on the current screen with the same subscript are blank then a required variable error is not generated for that blank array variable.

If an array variable is declared as GLOBAL and does not appear in the LAYOUT, then the number of elements will correspond to the earlier declaration of the variable. If it does appear in the LAYOUT section, the number of elements available will be the number appearing in the LAYOUT, ignoring the array keyword if it is specified in the local DECLARATION section.

Array screens are not limited to containing only array variables. For example:

```
*** LAYOUT
                      DISPLAY DETAIL LINES FOR INVOICES

     CUSTOMER:[CNO  ] [CNAME                                    ]

     INVOICE :[INVNO] DATE INVOICED:[DINV    ]

  LINE   PART               QUANTITY          PRICE      TOTAL
         NUMBER             ORDERED           EACH       AMOUNT
[LNUM ][PNO             ] [QTYORD      ]    [PRICE    ][TOTAMT      ]
[LNUM ][PNO             ] [QTYORD      ]    [PRICE    ][TOTAMT      ]
[LNUM ][PNO             ] [QTYORD      ]    [PRICE    ][TOTAMT      ]
[LNUM ][PNO             ] [QTYORD      ]    [PRICE    ][TOTAMT      ]
[LNUM ][PNO             ] [QTYORD      ]    [PRICE    ][TOTAMT      ]

                          TOTAL OF INVOICE  :      [INVTOT          ]
*** DECLARATION
   CNAME, DINV, INVTOT      ;DISPLAY
   PRICE, TOTAMT, INVTOT    ;NUMERIC; DECIMALS=2
   QTYORD, LNUM, CNO, INVNO;NUMERIC
   CNO, INVNO               ;REQUIRED
   LNUM,PNO,QTYORD,PRICE,TOTAMT; DISPLAY
   DINV                     ;DATE="M/D/Y"
   TEMPCNO                  ;NUMERIC

*** VARIABLE CNO
  NOTE    Save CNO before clearing screen
  TEMPCNO:=CNO
  CLEAR SCREEN
  RECORD POINT USING CUST; KEY=TEMPCNO
  IF $FOUND
      RECORD READ USING CUST
  ENDIF

*** VARIABLE INVNO
   RECORD POINT USING INV; KEY=INVNO
   IF $FOUND
      RECORD READ USING INV
   ENDIF

   RECORD POINT USING DETAILS; KEY=CNO, INVNO
   IF $FOUND
      INVTOT:=0
      ARRAY
         RECORD READ USING DETAILS
         TOTAMT:=QTYORD*PRICE
         INVTOT:=INVTOT+TOTAMT
      ENDARRAY
   ENDIF

*** INITIAL
   DISPLAY "PLEASE WAIT FOR FILES TO BE OPENED..."
   CREATE PARTITION CUST
     OPEN FILE CUST
```

```
        SET INDEX CNO
        DISPLAY "2..."
      CREATE PARTITION INV
        OPEN FILE INV
        SET INDEX INVNO
        DISPLAY "1..."
     CREATE PARTITION DETAILS
        OPEN FILE INVDET
        SET INDEX CNO,INVNO,LNUM
```

Array Loop Nesting

ARRAY/ENDARRAY loops may be nested, and BUILDER will correctly track $SUBSCRIPT in each loop. Be careful not to reference an element of the outer array loop within the inner loop, as $SUBSCRIPT will reflect the inner loop count and the wrong element of the outer array would be referenced. An example:

```
*** SCREEN PAYMENTS
*** LAYOUT
                 DISPLAY OF MONTHLY PRINCIPLE & INTEREST PAYMENTS
   LOAN OF[YR]YEARS            >> PRESS RETURN TO CALCULATE AMOUNTS

LOAN AMT[I      ]%[I     ]%[I     ]%[I     ]%[I     ]%[I     ]%[I     ]%
[A       ][P1    ][P1    ][P1    ][P1    ][P1    ][P1    ][P1    ]
[A       ][P2    ][P2    ][P2    ][P2    ][P2    ][P2    ][P2    ]
[A       ][P3    ][P3    ][P3    ][P3    ][P3    ][P3    ][P3    ]
[A       ][P4    ][P4    ][P4    ][P4    ][P4    ][P4    ][P4    ]
[A       ][P5    ][P5    ][P5    ][P5    ][P5    ][P5    ][P5    ]
[A       ][P6    ][P6    ][P6    ][P6    ][P6    ][P6    ][P6    ]
[A       ][P7    ][P7    ][P7    ][P7    ][P7    ][P7    ][P7    ]
[A       ][P8    ][P8    ][P8    ][P8    ][P8    ][P8    ][P8    ]
[A       ][P9    ][P9    ][P9    ][P9    ][P9    ][P9    ][P9    ]

*** DECLARATIONS
   I; NUMERIC; DECIMALS=2; JUSTIFY=RIGHT
   STINT,INCR,P1,P2,P3,P4,P5,P6,P7,P8,P9; NUMERIC
   A,STAMT,YR,AMOUNT,X                   ;NUMERIC
   A,I,P1,P2,P3,P4,P5,P6,P7,P8,P9        ;DISPLAY
   STAMT                                 ;INITIAL=50000
   INCR                                  ;INITIAL=0.5
   STINT                                 ;INITIAL=6.5
   YR                                    ;INITIAL=30
   MONTHLY_APR                           ;NUMERIC; DECIMALS=4
   INCRA                                 ;NUMERIC ;INITIAL=25000

*** ENTER
   NOTE   Display loan amounts starting at $50,000 and going up by
   NOTE        $25,000.  Display interest rates starting at 6.5% and
   NOTE        going up by .5%.
   ARRAY
      I:=STINT + (INCR*$SUBSCRIPT)
   ENDARRAY
```

```
ARRAY
  A:=STAMT + (INCRA*$SUBSCRIPT)
ENDARRAY
ARRAY
  AMOUNT:=A
   X:=$SUBSCRIPT+1
  ARRAY
    MONTHLY_APR:=(I/12.00)/100.00
    P[%X]:= &
     AMOUNT*MONTHLY_APR/(1.00-(1.00/((1.00+MONTHLY_APR)**(YR*12))))
  ENDARRAY
   REFRESH
ENDARRAY
```

GLOBAL VARIABLES

A global variable is, in general, one whose value needs to be passed from one screen to another.

Variable Declaration

Each variable has a place of origin: the screen in which it first came into existence (was DECLARED). A variable is DECLARED by either:

a) appearing in the LAYOUT section of a screen; and/or

b) being listed in a DECLARATION section that doesn't have a STATE label in either the GLOBAL section or any SCREEN.

GLOBAL keyword

At a variable's point of origin, the GLOBAL keyword should NOT be specified. The GLOBAL keyword is used for that variable in the DECLARATION section of screens below its point of origin. The GLOBAL keyword means "this variable has been declared previously."

If a variable is declared (per above definition) in a screen and it is not explicitly specified as a GLOBAL variable, it is assumed to be a NEW variable and will have no connection with any variables in screens above it.

If the SET OPTION VARIABLE=GLOBAL command has been issued, or if no SET OPTION VARIABLE command has been issued, and a variable is NOT declared in a given screen but is used in that screen somewhere other than in the LAYOUT or DECLARATION, it is assumed to be a GLOBAL variable; that is, it is treated the same as if it was listed in the DECLARATION section with a GLOBAL keyword.

If the SET OPTION VARIABLE=LOCAL command has been issued, and a variable is not declared in a given screen but is used there, an error will occur.

Availability of Use

If a variable's point of origin was the DECLARATION in the GLOBAL section, it can be referenced from anywhere else in the application as a GLOBAL variable. If its point of origin was a screen in the application, it can only be referenced in that screen or if that screen does a CALL SCREEN to at least one other screen.

Phrased another way: picture your application as a hierarchy of screens with the GLOBAL section at the top of the hierarchy and the first screen in the application directly below that. Whenever a CALL SCREEN is done, the called screen becomes a new, lower level in the hierarchy. When a SET SCREEN is done, the new screen replaces the previous screen at the same level in the hierarchy. A variable can be referenced if its point of origin was anywhere in the hierarchy higher than the current screen. (See the following chart.)

Matching variable names

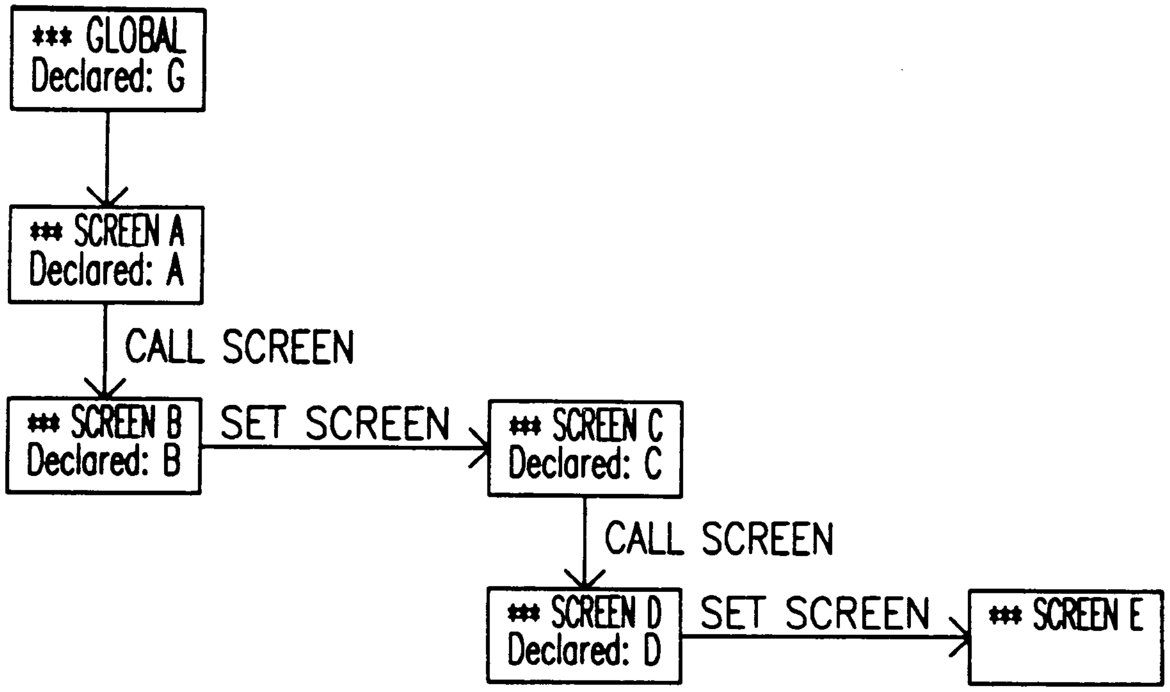
A variable determined to be GLOBAL will match with the variable with the same name that has the most recent point of origin, not with the first declared variable of the same name. For example:

```
*** SCREEN A
*** DECLARATION
  X
*** INITIAL
  X:=2
  CALL SCREEN B

*** SCREEN B
*** DECLARATION
  X
*** INITIAL
  X:=3
  CALL SCREEN C

*** SCREEN C
*** DECLARATION
  X; GLOBAL
*** INITIAL
  X:=4
```

In the above example, displaying the value of X in screen A will always show as 2. The value of X in screen B will show as 3 before going to screen C, and 4 after returning from screen C.

Another global variable example:

```
*** GLOBAL
*** DECLARATIONS
   NAME; LENGTH=20

*** INITIAL
   OPEN FILE CUSTOMER
   NOTE    This file contains fields NAME, ADDRESS, and CITY

*** SCREEN MAIN
*** LAYOUT
   PRESS F1 FOR NEXT SCREEN

*** DECLARATION
   ADDRESS; LENGTH=30

*** FUNCTION 1;LABEL="NEXT SCR"
   CALL SCREEN NEXT

*** SCREEN NEXT
*** LAYOUT
   [CITY                         ]

*** INITIAL
   RECORD READ
   NOTE  The above command will read values into the NAME,
   NOTE       ADDRESS, and CITY variables.
```



Variables available for global use in screen E: G, A, C

RECORD POINTER

All of BUILDER's RECORD commands:

```
RECORD ADD
RECORD DELETE
RECORD NEXT
RECORD POINT
RECORD READ
RECORD REPOINT
RECORD RESET
RECORD REWIND
RECORD UPDATE
$READ function
and all RELATE commands
```

affect or are affected by the current record pointer.

BUILDER actually maintains two pointers: one that points to the last record processed, and one that points to the position in the current index that was most recently processed.

RECORD POINT, RECORD REPOINT, RECORD REWIND, and any RELATE command set both of these pointers.

RECORD READ and RECORD NEXT look at the index pointer to determine which record to read; if the current index pointer has already been read, then both the index and record pointers will be moved to the next record, and that one will be read.

$READ performs its own POINT and possibly several reads, leaving both the index and record pointers positioned on the record AFTER the last record it read.

RECORD ADD alters the record pointer but does not affect the index pointer.

RECORD UPDATE and RECORD DELETE affect the record using the record pointer, not the index pointer.

RECORD RESET has no effect on either pointer.

See each command for additional information on how the pointer is affected.

The net effect is that one partition can be used to both ADD and READ without losing the READ pointer.

STATES

STATEs in BUILDER allow one screen to be used for several different purposes. The screen's current STATE is defined by using some combination of the following:

```
*** DECLARATIONS [STATE=statename]
*** INITIAL [STATE=statename]
*** ENTER [STATE=statename]
*** FUNCTION [character] [;LABEL="text"] [;STATE=statename]
SET STATE statename
```

DECLARATION

A DECLARATION section with a STATE name attached acts only as a series of MODIFY VARIABLES for the variables listed in that section which will take place whenever a SET STATE is done to the indicated STATE. Hence, an unlabeled DECLARATION section may still be necessary to define layout variables and create other variables. Variables modified by a labeled DECLARATION section will not be unmodified when a different STATE is set. If there is no labeled DECLARATION section for a STATE, the SET STATE command will not execute any DECLARATION section.

INITIAL

An INITIAL section with a STATE name attached is executed every time you enter that state (ie: with a SET STATE command). The unlabeled INITIAL section is only executed when you come into a screen (ie: with a SET or CALL SCREEN command). If there is no labeled INITIAL section for a STATE, the SET STATE command will not execute any INITIAL section.

ENTER

When RETURN/ENTER is pressed, an ACTION section will occur if appropriate. If no ACTION section is appropriate, then the ENTER section with the appropriate STATE label is executed. If there is no ENTER section so labeled, the unlabeled ENTER section is executed.

FUNCTIONs

Function key labels are redrawn to match the function keys for that state, if needed, whenever a SET STATE is done. When a function key is pressed, the FUNCTION section matching the current STATE, if there is one, is executed. If not, and there is an unlabeled FUNCTION section matching the function key, it is executed.

SET STATE

The SET STATE command will:

1. Modify variables and set INITIAL values based on the DECLARATION section matching that state if one exists.
2. Execute the INITIAL section for that state, if one exists.
3. Redraw the function key labels to match the function keys for that state.

EFFICIENCY

A SET STATE operation can take almost as much time as going to a new screen, depending on how many variables are declared and how many function keys are redefined. STATES are best used for operations that the user will be performing periodically (specifically, as a replacement for moving from one screen to a very similar one), rather than frequently (such as checking for errors after a file update or other subroutine-type operations).

EXECUTION PROCESSING LOOP

The interpreter executes an application by following a well defined sequence of steps. These steps determine when to evaluate each of the individual sections. The creator controls BUILDER's interface with the user and the processing by placing commands and specifications into these sections.

At the highest level, the system displays a screen, waits for input, processes the input, and then displays a screen again. The action taken by the system is determined by the data input and the sections that exist in the application. A more detailed description of this processing loop follows. This execution processing loop is interrupted when an error occurs or SET SCREEN, CALL SCREEN, or RETURN SCREEN commands are executed.

1) Evaluate the LAYOUT section, if one exists.

2) Evaluate the unlabeled DECLARATION section, if one exists.

3) Execute the unlabeled INITIAL section, if one exists.

4) Execute the INITIAL EVERYTIME section.

5) Display the new screen layout or update any changes made to the existing layout.

6) (Non-Block Mode) Wait for TAB, a function key, RETURN, or until a variable has been filled.

6) (Block Mode) Wait for a function key or ENTER to be pressed.

7) If the value of a variable was changed by the user (in block mode, all changed variables are checked at this point), perform the actions in the VARIABLE section associated with the variable.

8) If neither a function key nor RETURN/ENTER was pressed, continue at step 5.

9) If no FAIL or FINISH was encountered in a VARIABLE section, perform an automatic VERIFY (unless disabled with the SET OPTION VERIFY command).

10) If no FAIL or FINISH or automatic BUILDER error has been encountered, execute the appropriate FUNCTION, ACTION or ENTER section.

11) Whether or not a FAIL, FINISH, or automatic message was encountered, return to step 4.

CHANGING SCREENS

There are two methods in BUILDER to move from one screen to another:

```
        SET SCREEN name
or:     CALL SCREEN name        followed by        RETURN SCREEN
```

SET SCREEN

1) The new screen is searched for in the current application file.

2) If the new screen cannot be found in the current application file, the screen name is treated as a file name and the system attempts to open the file. If the file still cannot be opened, the first screen in the file is executed ~~an error results.~~ If the file can be opened, the screen name is searched for in the new file. If the screen cannot be found, an error results.

3) The previous screen definition and all variables defined on the screen are released.

4) The new screen is obtained from the LAYOUT section and the normal processing loop, described in the previous section, is begun.

CALL SCREEN / RETURN SCREEN

1) When the CALL SCREEN is executed, the new screen is searched for in the current application file.

2) If the new screen cannot be found in the current application file, the screen name is treated as a file name and the system attempts to open the file. If the file still cannot be opened, an error results. If the file can be opened, the screen name is searched for in the new file. If the screen cannot be found, an error results.

3) The previous screen definition is released and all variables defined on the screen are saved.

4) The new screen is obtained from the LAYOUT section and the normal processing loop, described in the previous section, is begun.

5) **If a CALL SCREEN is used, a RETURN SCREEN, not a SET SCREEN, should always be used to return to the previous screen.**

6) When the RETURN SCREEN is executed, the layout of the screen being returned to is located. If the previous screen existed in another application file, the file is accessed.

7) The existing screen and all variables defined by the screen are released.

8) The return screen is obtained from the LAYOUT section, and command execution continues at the command after the CALL SCREEN command.

SUBROUTINES

A screen without a LAYOUT section is treated as a subroutine screen. CALL SCREEN and SET SCREEN commands may be used to reference the subroutine screen. Commands can be executed in such a screen without prompting the user for input or displaying output. Subroutine screens are treated differently than other screens with respect to the execution processing loop.

When a subroutine screen is called, the existing screen is not erased, and step 5 of the Builder Interpreter's processing loop is not performed. The screen is not redrawn, and it does not wait for input from the user. An ACTION or ENTER section is executed immediately. The subroutine screen may prompt the user for data using the PROMPT command.

If a CALL SCREEN is used, all variables from the calling screen may be referenced in the subroutine screen. The subroutine screen may change any of the values available to it.

A SET SCREEN will destroy the previous screen definition so variables from the previous screen cannot be used in the subroutine screen, but actions can be performed in the subroutine screen without requesting input from the user.

The final command in a subroutine screen should be a RETURN SCREEN or SET SCREEN.

If there is no LAYOUT section, BUILDER will loop continually through the ENTER section until a CALL SCREEN, RETURN SCREEN or SET SCREEN is encountered.

See the Performance section for additional comments.

SUBSTITUTION ALGORITHM

Before a command is executed it is passed through the following substitution procedure. This procedure substitutes data from variables defined in the application into the command. The substitution takes place in all parts of any line in the section being executed: within quoted strings, comments, and keywords. A command is not checked for validity until after substitution has been done.

The substitution procedure first separates the command into words. A word is a sequence of characters separated by blanks or by a special character. A word contains letters, digits, or any of the following characters:

Percent sign ("%") as the first character.
Dollar sign ("$").
Underscore ("_").

If a word starts with a percent sign, the word is assumed to be a variable name known at the current state of the application. The value of the variable, with leading and trailing blanks removed (unless the variable was declared with option DEBLANK=NO), is substituted into the command replacing the variable name and the percent sign. If the variable was declared with DEBLANK=NO, the full size of the variable is used. If the variable does not exist, the name is simply removed and no error is generated.

The substitution procedure also has several other capabilities that allow for optional substitution of constant information and lists. These capabilities treat brackets ("[ ]") and braces ("{ }") in special ways. All brackets and braces (not preceded by a percent sign ("%")) must be paired or an error will be generated.

Any string enclosed in brackets is optional. If all of the variables within the brackets contain blank values then everything between the brackets is ignored and removed from the resulting command. If any variable within the brackets contains a non-blank value, then all constants and all substituted variables within the brackets remain in the command. The brackets are removed after the substitution.

Any string enclosed in double brackets ("[[" and "]]") is also optional. However, if any of the variables within the brackets contain blank values, then everything between the brackets is ignored. Otherwise, the enclosed string remains in the command. The brackets are removed after the substitution.

Any string enclosed in braces is a list. A list is one or more optional variables separated by commas or other delimiters. The following algorithm is used to compute the value of the string within the braces. Nested braces or brackets are treated as variables in the following context.

1) All characters until the first non-blank variable within the braces are ignored. The variable is substituted into the string.

2) The next variable within the list is located. If another variable does not exist in the list, the remaining characters are included and the substitution is completed.

3) If the variable found is blank, the characters between the last variable up to and including the current variable are removed. If the variable is non-blank, then include all the characters from the last variable until, and including, the current variable. Continue at step 2.

If a character special to the substitution algorithm needs to be included in a command, the character should be preceded with a percent sign. For example, to include a left bracket in a command a "%[" should be entered (instead of just "["). Likewise, to include a single percent sign enter "%%".

All variables available to the application may be used in the substitution algorithm. This includes system-defined variables such as $DATE and $FOUND. All substitution is performed before any execution, including the evaluation of functions and built-in variables. If a system-defined variable which normally performs calculations is used (such as $FOUND), these calculations are not performed and the existing value of the variable is used. $VARIABLE can be substituted accurately on its first use.

All BUILDER commands and most RELATE commands require that alphabetic information be enclosed within quotes. There may be a problem when the message contains a substituted variable and the substituted variable contains a quote. For example, assume the variable TEXT contains the string: THIS "IS" QUOTED. The variable TEXT contains two quotes. After substitution the SELECT command

```
SELECT * WHERE TEXT="%text"
```

would normally be:

```
SELECT * WHERE TEXT="THIS "IS" QUOTED."
```

But this would cause a RELATE syntax error. Because of this error, the substitution routines, by default, change a single set of quotes to a double set. So after substitution you would really get:

```
SELECT * WHERE TEXT="THIS ""IS"" QUOTED."
```

Now your command is syntactically correct. There are cases, however, when you don't want the quotes changed. This capability to change quotes can be controlled with the SET OPTION command.

Examples:

Assume the following variables and the following values:

```
COLOR  ~=3
LINE   ~=
MARKER ~=2
EMPNO  ~=1408
SALARY ~=
DEPT   ~=SALES
F1     ~=EMPNO
F2     ~=SALARY
F3     ~=
F4     ~=DEPT
F5     ~=
P      ~=
X      ~=7
```

Then the following templates will be substituted to produce the corresponding RELATE commands.

```
Original Command:
   DRAW GRAPH X [;COLOR=%color]
                [;LINE=%line]
                [;MARKER=%marker]
              BY Y

Executed:
   DRAW GRAPH X ;COLOR=3
                ;MARKER=2
              BY Y
```

The "X" is not substituted into the above command because it is not preceded by a percent sign.

```
Original Command:
   PRINT[:%p] {%f1,%f2,%f3,%f4,%f5}

Executed:
   PRINT EMPNO,SALARY,DEPT


Original Command:
   SELECT * WHERE {[EMPNO=%empno]
            AND [SALARY=%salary]
            AND [DEPT="%dept"]}

Executed:
   SELECT * WHERE EMPNO=1408
            AND DEPT="SALES"
```

EXAMPLES:

```
  *** SCREEN OPEN
  *** LAYOUT

          NAMES OF FILES TO OPEN:     [FNAME1          ]
                                      [FNAME2          ]
                                      [FNAME3          ]

             PRESS RETURN TO OPEN FILES.

 *** ENTER
   NOTE   If an fname is left blank, don't do OPEN FILE at all
   [OPEN FILE %FNAME1]
   [OPEN FILE %FNAME2]
   [OPEN FILE %FNAME3]


 *** SCREEN REPORT
 *** LAYOUT
                                 EMPLOYEE SEARCH

    LAST NAME OF EMPLOYEE:[LNAME              ]
    FIRST NAME:           [FNAME              ]
    DATE OF BIRTH:        [BDATE    ]
    AGE:                  [AGE]
    SALARY RANGE  -  LOW: [LSAL       ]  HIGH: [HSAL        ]

      FILL IN KNOWN VALUES AND PRESS RETURN

*** DECLARATIONS
   BDATE;        DATE="M/D/Y"
   AGE;          NUMERIC
   LSAL,HSAL;    NUMERIC; DECIMALS=2

*** ENTER
   SELECT ● {WHERE [LNAME="%LNAME"] AND &
                   [FNAME="%FNAME"] AND &
                   [BDATE="%BDATE"] AND &
                   [AGE=%AGE]       AND &
            [[SALARY>=%LSAL AND SALARY<=%HSAL]] }
   IF $FOUND
       PRINT
   ENDIF
```

DEBUGGING FACILITIES

A number of debugging facilities have been incorporated into BUILDER. These facilities are only available if the application is running in CREATOR mode. The facilities are invoked automatically when an error is encountered during the execution of the application in creator mode. The error, as well as the filename and line number of the error, is displayed and then debug mode is entered. The facilities may also be invoked with the "/" function (which consists of an ESC followed by a "/"). At this point, the creator may edit or modify the application status to fix the error and restart or resume the application.

When debug mode is entered, the following prompt will appear:

```
Entering Debug Mode...
1-EDITOR, 2-COMMANDS, 3-RESTART, 4-RESUME, 5-EXIT, 6-EDIT TERMINAL
DRIVER
(1-6)?
```

1-EDITOR

Selecting EDITOR will provide a second set of choices allowing entry into the HP editor, QEDIT, or an editor of the creator's choice. The application file in use by BUILDER is closed before the editor is invoked so that the application under test can be changed. When the editor is exited, control is returned to the debug mode prompt. Once an editor has been chosen, the file RDBEDIT will be set to match the chosen editor and the editor choices will not be repeated until after BUILDER has been terminated. If a file equation exists for RDBEDIT before Debug Mode is entered, then this choice will go directly into the editor program specified by RDBEDIT.
If RDBEDIT does not reference a valid editor program, HP's editor will be used.

2-COMMANDS

Selecting COMMANDS enters a form of the RELATE Command Interpreter. From this interpreter, most RELATE or any BUILDER commands may be executed. While in the Command Interpreter, the status is the same as it was while executing the application. That is, the same files are open, the same current partition, path, and index are in effect, the same variables exist, etc. Any changes to this status will be maintained if the execution of the application is continued. The substitution algorithm is applied to commands executed in this mode. If "//", END, or EXIT is entered, the Command Interpreter is exited and control is returned to the debug mode prompt.

The BREAK command can be used to directly enter the Command Interpreter in the middle of a section. This command is available in both creator mode and end-user mode, although it will not normally exist in a completed application.

3-RESTART

Restart will close all files and partitions, remove all variables, and restart BUILDER from the beginning. BUILDER will prompt for the name of the application file to be used (BLDRAPP). The previous value of BLDRAPP is the default.

4-RESUME

The RESUME option will redraw the screen and the application will continue from the point at which debug mode was entered.

5-EXIT

Exit will cause termination of BUILDER and a return to the operating system.

COMPILING AN APPLICATION FILE

To enter compile mode,

```
        :RUN BUILDER.PUB.RELATE45,COMPILE; LIB=P
```

(BUILDER may not be installed in PUB.RELATE45. Verify where it is installed at your site.)

BUILDER will prompt for "Output File?". BUILDER will create a file with this name in which to place the compiled version of your application.

BUILDER will then prompt for "Application File?" continually. This allows you to place several related application files into one compiled output file. The first screen in the first application file must be the initial screen of the application. When you have no more application files, press RETURN. BUILDER will then terminate.

To use the compiled version, specify its filename as the application file.

The compilation process does not generate object code. Instead, the application is compressed and certain tables are built which improve BUILDER's operation.

WARNING: Compiled BUILDER files are not guaranteed to be downward or upward compatible. Installation of a new version of BUILDER may require the recompilation of applications. This information can be found in the release notes for the product.

SAMPLE APPLICATION

```
*** GLOBAL
*** FUNCTION 7: LABEL="PREVIOUS|SCREEN"
   RETURN SCREEN

*** FUNCTION 8; LABEL="EXIT"
   PROMPT "ARE YOU READY TO EXIT?",ANSWER
   IF ANSWER="Y"
      EXIT
   ENDIF

*** DECLARATIONS
   ANSWER; UPPER; LENGTH=1; RANGE="Y","N"

*** SCREEN MAIN
   This is a sample application for a PARTORD file containing:
      INVOICE:        invoice number
      PART:           part number
      QTY:            quantity ordered
      PRICE_EACH:     price per part

*** LAYOUT
                     WELCOME TO THE PARTS ORDERS DATABASE
                              TODAY IS [today    ]

                    PLEASE ENTER THE LETTER OF YOUR CHOICE
                             AND PRESS RETURN: [c]

                              M - Multiple line screen
                              S - Single line screen

*** DECLARATIONS
   TODAY; DISPLAY; DATE="M/D/Y"
   C; NAME=CHOICE; ACTION; UPPER

*** INITIAL
   TODAY:=$DATE
   IGNORE ALL ERRORS
   OPEN FILE PARTORD
   IF $ERROR <> 0
      DISPLAY "UNABLE TO OPEN PARTORD FILE."
      EXIT
   ENDIF
   SET INDEX INVOICE, PART

*** INITIAL EVERYTIME
   CHOICE:=" "

*** ACTION M
   CALL SCREEN MULTI
```

```
*** ACTION S
   CALL SCREEN SINGLE

*** FUNCTION 7; LABEL=" "
  NOTE   Can't return from here: override GLOBAL function

*** ENTER
   FAIL "INVALID CHOICE",CHOICE

*** SCREEN SCAN
*** LAYOUT
                         CUSTOMER DATABASE
        [invoice   ] [part          ] [qty    ] [total    ]
        [invoice   ] [part          ] [qty    ] [total    ]
        [invoice   ] [part          ] [qty    ] [total    ]
        [invoice   ] [part          ] [qty    ] [total    ]
        [invoice   ] [part          ] [qty    ] [total    ]
        [invoice   ] [part          ] [qty    ] [total    ]
        [invoice   ] [part          ] [qty    ] [total    ]

     PRESS RETURN TO SAVE CHANGES ON SCREEN
*** DECLARATIONS
   INVOICE, PART, QTY, INVNO: NUMERIC
   TOTAL, INVOICE, PART      : DISPLAY
   TOTAL                     : NUMERIC; DECIMALS=2
   PRICE_EACH                : NUMERIC; DECIMALS=2; ARRAY=7

*** INITIAL
   NOTE   Automatically read the first 6 records
   RECORD REWIND
   ARRAY
      RECORD READ
      TOTAL:=QTY*PRICE_EACH
   ENDARRAY

*** VARIABLE QTY
   TOTAL:=QTY*PRICE_EACH

*** ENTER
   ARRAY
      RECORD POINT; KEY=INVOICE, PART
      IF $FOUND
         RECORD UPDATE
      ENDIF
   ENDARRAY

*** FUNCTION 1; LABEL= "MORE|RECORDS"
   IF $EOF
      FINISH "There are no more qualified records.  Use F3."
   ENDIF
   ARRAY
      RECORD READ
      TOTAL:=QTY*PRICE_EACH
   ENDARRAY
```

```
*** FUNCTION 2; LABEL="DELETE"
   RECORD POINT;KEY=INVOICE, PART
   IF $FOUND
      RECORD DELETE
   ELSE
      FAIL "CANNOT DELETE - DOESN'T EXIST"
   ENDIF
   NOTE  -- your pointer will now be displaced for your next
   NOTE     set of READs.  Having a separate PARTITION for
   NOTE     DELETEs would help.

*** FUNCTION 3; LABEL="INVOICE|SEARCH"
   PROMPT  "WHICH INVOICE (blank for start of file)?",INVNO
   IF INVNO=" "
      RECORD REWIND
   ELSE
      RECORD POINT; KEY=INVNO
      IF NOT $FOUND
         FAIL  "INVOICE NOT FOUND"
      ENDIF
   ENDIF
   ARRAY
      RECORD READ
      TOTAL:=QTY*PRICE_EACH
   ENDARRAY

*** FUNCTION
   DISPLAY  "ONLY FUNCTIONS 1 THROUGH 3 AND 7 THROUGH 8 ARE DEFINED"

*** SCREEN SINGLE
*** LAYOUT

        INVOICE NUMBER    : [invoice]
        PART NUMBER       : [part              ]
        QUANTITY ORDERED  : [qty      ]
        TOTAL ORDER       : [total      ]

         TYPE DATA AND PRESS RETURN TO ADD.

*** DECLARATIONS
  INVOICE, PART, QTY        ; NUMERIC
  TOTAL                     ; NUMERIC; DECIMALS=2; DISPLAY
  PRICE_EACH                ; NUMERIC; DECIMALS=2

*** VARIABLE INVOICE
   NOTE   FIND THE FIRST RECORD WITH THIS INVOICE #
   RECORD POINT; KEY=INVOICE
   IF NOT $FOUND
      CLEAR VARIABLE PART, QTY,TOTAL, PRICE_EACH
   ELSE
      RECORD READ
      TOTAL:=QTY*PRICE_EACH
   ENDIF
```

```
*** FUNCTION 1; LABEL="READNEXT"
    IF $EOF
       FINISH "NO MORE LINES FOR THIS INVOICE"
    ELSE
       RECORD READ
    ENDIF

*** FUNCTION 2; LABEL="DELETE"
    RECORD POINT; KEY=INVOICE,PART
    IF NOT $FOUND
       FAIL "CAN'T DELETE - DOESN'T EXIST", PART
    ELSE
       RECORD DELETE
    ENDIF

*** ENTER
    RECORD POINT KEY=INVOICE, PART
    IF NOT $FOUND
       IGNORE ERROR
       RECORD ADD
       IF $ERROR
          FAIL $ERRMSG
       ELSE
          CLEAR SCREEN
          FINISH  "RECORD IS ADDED"
       ENDIF
    ELSE
       IGNORE ERROR
       RECORD UPDATE
       IF $ERROR
          FAIL $ERRMSG
       ELSE
          CLEAR SCREEN
          DISPLAY  "RECORD IS ADDED"
       ENDIF
    ENDIF
```

SECTION 4

A P P L I C A T I O N   C O M M A N D S

BUILDER Commands

BUILDER commands that are longer than one line width may be continued on the next line by placing an ampersand ("&") as the last non-blank character on a line. If an ampersand is found to be the last non-blank character on a line, then the next line is taken to be a continuation of the line.

Leading and trailing blanks on a line are ignored so indentation may be performed without fear of running into the maximum command length prematurely. The maximum length of a command is 1500 characters before substitution and 1500 characters after substitution. If there are one or more blanks before the ampersand ("&") continuation character only a single blank will be included in the resulting command string.

If leading blanks should be significant to the command (e.g., in a DISPLAY statement), the leading blanks can be made significant by preceding the blanks with "[ ]". The brackets designate an optional substitution and will substitute nothing into the string. Blanks after the brackets will then be significant with respect to the command.

Keywords in BUILDER commands may not be abbreviated unless explicitly described otherwise. Plural keywords can be abbreviated to the singular (eg: VARIABLES to VARIABLE).

Using RELATE (and MPE) Commands

Any RELATE command that may be executed programmatically may be included in an application. The substitution algorithm is applied to the command before it is executed by RELATE. BUILDER passes any command that is not a BUILDER command to the RELATE command interpreter. This means that operating system commands can be used by preceding them with a colon (":"). RELATE commands may be abbreviated as specified in the RELATE reference manual. If RELATE cannot understand the command, an error occurs.

Execution of any RELATE command will cause the current path (if any) to be rewound (the current record pointer is moved to the first record in the current path). The only exception to this is the NOTE command, which can be used to document executable sections.

RELATE commands that go into prompt mode (eg: ADD, CREATE FILE, REPORT, etc.) will overwrite part of the message line with their first line of output. If nothing is currently displayed on the message line, this will have no noticeable effect. To preserve text on the message line, precede the RELATE command with:

```
    SCROLL ""
```

RELATE understands any command line beginning with a colon (":") to be an MPE command. Any programmatically executable MPE command can be requested through RELATE. The substitution routine is applied to the command before it is passed to MPE.

See the CREATE PARTITION command for comments on some RELATE commands.

RELATE commands do not change the values of any variables or built-in functions of BUILDER.

EXAMPLES:

```
*** DECLARATIONS
   NEWDAY; DATE
*** FUNCTION 4
   NOTE  Date constants must be in quotes
   SELECT NEWDAY=$NEW_DATE($DATE,"%olddate")
   NOTE  RELATE commands only affect the current file/path.
   NOTE  They do not affect the values of variables.  The
   NOTE  current path now has one field called NEWDAY.  If
   NOTE  I "DISPLAY NEWDAY", I will get blanks.
   RECORD READ
   NOTE  Now the variable NEWDAY has a value in it, because the
   NOTE  first (in this case, the only) record of the SELECT
   NOTE  has been read into matching variable names.
   SELECT
```

```
*** ENTER
   OPEN FILE %fname;PATH=F
   PRINT[:%p] {%field1,%field2,%field3}
   SELECT @ WHERE %field=%value
   IF NOT $FOUND
      DISPLAY "No records can be found for the search value."
   ELSE
      ARRAY
         RECORD READ
      ENDARRAY
   ENDIF
```

ARRAY
ENDARRAY

Loops through commands once for each array element.

Any commands that appear between the ARRAY command and the matching ENDARRAY command will be executed once for each element of any array referenced inside the loop. The commands that use the current variables will use different array elements each time through the loop. Any variable that is not an array variable can be used every time through the loop. BUILDER maintains the current value of $SUBSCRIPT internally. **The duration of the loop is unaffected by assignments made to $SUBSCRIPT.**

An array variable is a variable that appears more than once on the same screen. An array screen is any screen containing one or more array variables. The $SUBSCRIPT built-in variable is incremented each time through the loop. **The loop terminates when commands within the loop access the last element of any array variable, when no array element is accessed within the loop, or when an EXITARRAY is encountered.**

For further information on array processing, see the ARRAY VARIABLES discussion in section 3.

EXAMPLES:

```
*** LAYOUT
     [partno   ] [desc                        ] [price     ]
     [partno   ] [desc                        ] [price     ]
     [partno   ] [desc                        ] [price     ]
     [partno   ] [desc                        ] [price     ]

     F1=Next four records        F2=Update these four records
*** FUNCTION 1
    NOTE    This reads records from the current path
    NOTE    until all array elements are filled.
    ARRAY
       RECORD READ
    ENDARRAY
*** FUNCTION 2
    NOTE    This goes through each array element,
    NOTE    searches for the part number, and updates the record.
    SET INDEX BY PARTNO
    ARRAY
       RECORD POINT
       IF NOT $FOUND
          FAIL "PART NUMBER %partno DOES NOT EXIST.", PARTNO
       ELSE
          RECORD UPDATE
       ENDIF
    ENDARRAY
```

variablename := expression

Performs arithmetic or character manipulations and assigns the value of the expression to the indicated variable.

variablename	Required. The variablename must be the name of a variable defined within the application.
expression	Required. The expression must correspond to the rules given in the EXPRESSION EVALUATION section. The expression is evaluated and the result is assigned to the given variable.

The ":=" may not be separated by a space or appear on two lines.

EXAMPLES:

```
*** VARIABLE TIME
    NOTE   Date constants must be enclosed in quotes
    NEW_DATE:="%old_date"
    MESSAGE:=$CONCAT("AGE FOR ",NAME," IS %age")
*** ENTER
    FILETYPE:="MPE"
    FILEMODE:="SHARED"
    COMMAND:="OPEN FILE %fname [;TYPE=%filetype]"
    COUNT:=COUNT+1
    START_DATE:="1/31/84"
    MAXIMUM:=500/NUMBER
```

BREAK

Causes control to be transferred to the BUILDER interpreter.

Any valid BUILDER command can be executed in the BUILDER interpreter. To return to the application, enter a double slash ("//").

This command is most useful in debugging applications.

EXAMPLES:

```
  *** ACTION DEBUG
        BREAK
*** ENTER
      IF HOSPITAL=" "
         FINISH "MUST FILL IN HOSPITAL NUMBER",HOSPITAL
      ELSE
         NOTE  my add doesn't seem to work well... need to
         NOTE     check things out here
         BREAK
         RECORD ADD
      ENDIF
```

CALL PROCEDURE slprocname

Calls procedures written in SPL, COBOL, FORTRAN, or PASCAL. The procedure must be resident in an SL and must conform to BUILDER calling conventions.

slprocname	Required. The slprocname must be the name of a procedure that can be loaded from the group, account or system SL (segmented library). If no such procedure exists, this command will return an error.

Any procedure called in this way must contain exactly three parameters. All three parameters must be integer arrays. If the user's procedure does not use any of these arrays, they can be single word items as they are passed by reference.

The first array is the current partition being used in the application. If it is used in your procedure in any HLI calls (see the RELATE manual), RELATE assumes that it has at least 50 words. It has already been initialized and filled by BUILDER.

The second array is the SCA (screen maintenance) table being used by the screen system. There are currently no user routines available that use this array, and it should not be manipulated by the user.

The third array is the PAR (parameter) array and contains information about the CALL PROCEDURE command that called the procedure. If the application file has anything on the same line as the CALL PROCEDURE command after the procedure name, BUILDER will ignore the additional text. The procedure can obtain the information through the PAR array (described below). This is useful for passing parameters to the procedure from the application.

OFFSET	CONTENT
0	The word address of the CALL PROCEDURE command.
1	The length (in bytes) of the entire command.
2	The byte offset into the command of the first non-blank character after the procedure name. If no parameters are passed this value is zero.
3	PAR(1) minus PAR(2)
4	Not used.
5	The number of fields in the current path.
6	The word address of the data record maintained by BUILDER for the current path.
7	The number of words in the data record.
8	The word address of the FLDs which comprise the current index.
9	The number of fields in the current index.

Any operation that is CPU intensive or requires complex computation can be done in a traditional programming language without requiring that the entire application be written using traditional methods.

EXAMPLES:

```
*** ENTER
    CALL PROCEDURE COMPUTE'TOTALS
    PROMPT  "CUSTOMER NAME?",CNAME
    CNUM:=50
    CALL PROCEDURE DB'UPDATE  CUSTOMER  %cname  %cnum
    NOTE  The command after substitution is:
    NOTE  CALL PROCECURE DB'UPDATE CUSTOMER   EASTERN LUMBER  50
------------------------------------------------------------------------

PROCEDURE  COMPUTE'TOTALS(PRT,SCA,PAR);
INTEGER ARRAY  PRT,SCA,PAR;
BEGIN
       <<note:  do some processing>>
       END
------------------------------------------------------------------------

PROCEDURE  DB'UPDATE(PRT,SCA,PAR);
INTEGER ARRAY PRT,SCA,PAR
BEGIN
  INTEGER COMMAND'LEN,START'OF'PARMS,LEN'OF'PARMS,
     NUM'OF'FIELDS;
  INTEGER POINTER COMMAND,FIELD'INFO;

  @COMMAND:=PAR(0);          <<pointing to "CALL PROCEDURE DB'UP..."> >
  COMMAND'LEN:=PAR(1);       <<PAR(1) is=51                          >>
  START'OF'PARMS=PAR(2);     <<PAR(2) is=26,points to "CUSTOMER...."> >
  LEN'OF'PARMS:=PAR(3);      <<PAR(3) is=25                          >>
  NUM'OF'FIELDS:=PAR(5);
```

CALL SCREEN screenname

Displays a new screen.

screenname	Required. The screenname must be the name of a screen in the current application file or the name of a file in the same group and account as the original application file. MPE FILE equations are not referenced.

When the CALL SCREEN command is issued, the current file and the location of the CALL SCREEN instruction in the file are saved by BUILDER. When a RETURN SCREEN command is issued, the screen from where the call was issued becomes the current screen and the commands following the CALL SCREEN command are executed.

Before the CALL SCREEN command is processed the current values of the QUOTES, SUBSTITUTE, and VERIFY options (see SET OPTIONS) are saved. A subsequent RETURN SCREEN will return these items to their original values.

Calls may be nested. The maximum nesting level varies depending on the screens and variables involved, but it is usually about 20. SET SCREEN commands may be issued between the CALL and the RETURN without affecting the return location. **A RETURN SCREEN, not a SET SCREEN, should always be used to return to a screen where a CALL SCREEN was used. Failure to do so results in a recursive grabbing of memory.**

The CALL SCREEN command executes the INITIAL section of the new screen. It gets the new screen layout, displays the new screen, waits for user input, and then begins processing the new screen's actions. If the new screen initialization section contains an error or there is an error in the screen layout, then the CALL SCREEN command returns an error. If the specified file does not exist or operating system restrictions prevent the file from being read, an error will also be returned.

EXAMPLES:

```
*** ENTER
    CALL SCREEN SCR6
    DISPLAY  "This will display after returning from SCR6"
```

CLEAR [ALL|RECORD [USING partitionname]|SCREEN|VARIABLE variablenamelist [;SUBSCRIPT]]

Clears various groups of variables.

ALL	Optional. If specified, all the variables in the application are cleared.
RECORD	Optional. If specified, all the variables in the current screen which also exist in the current path in the current partition are cleared.
USING	Optional. If specified, the keyword must be followed by the name of an existing partition. All variables in the screen which also exist in the current path on the specified partition are cleared.
SCREEN	Optional. If specified, all the variables in the current LAYOUT are cleared.
VARIABLE	Optional. Clears the indicated variables. This has the same effect as an assignment of the form variablename:="" for all subscripts for each variable in the list. Hence, if this is used on an array variable within an ARRAY loop without the SUBSCRIPT parameter, the loop will not occur again as the highest element of an array has been referenced.
SUBSCRIPT	Optional. If specified, only variables that exist with the current subscript value are cleared. The keyword must be used in combination with the other keywords. For example, CLEAR RECORD;SUBSCRIPT will clear all variables in the current path for the current subscript.

If no keywords are specified, 'SCREEN' is assumed.

EXAMPLES:

```
*** LAYOUT
        ENTER CHOICE AND PRESS RETURN:[X]
            1 - ADD RECORD          2 - CLEAR SCREEN
  [CNO         ]   [NAME                 ] [ADDRESS                  ]
*** DECLARATION
    X: ACTION
*** ACTION 1
   NOTE
   NOTE  Clear all variables which also exist in the current path.
   NOTE
   RECORD ADD
   CLEAR RECORD
*** ACTION 2
   NOTE
   NOTE  Clear all variables in the layout.
   NOTE
   CLEAR
```

CREATE PARTITION partitionname; SEGMENT=number

Creates the indicated partition and causes the partition to become the current partition.

partitionname	Required. Partitionname must contain only letters, digits, and the underscore ("_") character and must not currently exist. The length of the partitionname cannot be more than 15 characters. The indicated partition is created and made the current partition.
SEGMENT	Optional. If specified, this keyword must be followed by an integer between one and ten thousand. If this number has not been used previously, BUILDER will create a new RELATE process. Normally, one process is sufficient for performing all desired operations and this keyword need not be used. Occasionally, however, more files need to be open than can be accessed from multiple partitions within the default process. All processes remain active until BUILDER terminates.

A partition can be thought of as representing a RELATE process. All RELATE commands will affect only the current partition. Files opened in partition A cannot be referenced if partition B is the current partition.

One partition is created automatically by BUILDER and is, by default, the current partition. Its name is DEFAULT.

Segment Resources

If a file is to be opened in more than one SEGMENT, it must be accessed in SHARED or SEMI-EXCLUSIVE mode in each segment.

Image databases are SEGMENT resources. This means that, once the database is open, it need not be opened again in other partitions in the same SEGMENT, although different sets may be opened in different partitions. In addition, purging the partition in which the database was originally opened will not cause the database to be closed.

If a MODIFY FIELD is done to any file, it affects the base file. This means that all open paths to this file in the same SEGMENT will automatically have the same modification made. If one path needs to have different field names, a SELECT is a better way to do this.

Control-Y is only enabled in one segment at a time. By default, the original process segment created automatically by BUILDER has control-Y enabled and it will be disabled in all other segments. Control-Y will allow an enduser to interrupt any command, including RECORD UPDATE and RECORD POINT. RELATE's SYSTEM command can be used to disable Control-Y or to enable it in a different segment. MPE will only allow the Control-Y to be enabled in one segment at a time. The SYSTEM command will only affect commands executing in the SEGMENT in which it was executed.

RELATE's TERMINAL command also only affects commands executing in the segment in which it was executed.

EXAMPLES:

```
*** GLOBAL
   COMMENTS:  This application allows scanning and updating of
        information about what animals are kept in which zoo
        buildings.

*** INITIAL
   CREATE PARTITION BLDG
      OPEN FILE BLDG
      SET INDEX BLDNUM
   CREATE PARTITION A_READ
      OPEN FILE ANIMALS
      SET INDEX BLDNUM
   CREATE PARTITION A_UPD
      OPEN FILE ANIMALS
      SET INDEX BLDNUM, NAME
   SET PARTITION DEFAULT

*** SCREEN REVIEW
*** LAYOUT
    BUILDING NUMBER:[BLD]   BUILDING NAME:[BNAME                        ]
                            CARETAKER    :[CNAME                        ]

   ANIMAL NAME        ANIMAL TYPE
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]
[ANAME              ][TYPE                  ] [AGE]  [DIET              ]

*** DECLARATIONS
   BLD: NAME=BLDNUM
   ANAME:   DISPLAY

*** VARIABLE BLDNUM
   RECORD POINT USING BLDG
   IF NOT $FOUND
      CLEAR SCREEN
      FINISH
   ENDIF
   RECORD READ USING BLDG
   RECORD POINT USING A_READ
   IF NOT $FOUND
      CLEAR VARIABLE ANAME, TYPE, AGE, DIET
      FINISH
   ENDIF
   ARRAY
      RECORD READ USING A_READ
   ENDARRAY
```

```
*** ENTER
   IF $CHANGED("BNAME") OR $CHANGED("CNAME")
       RECORD UPDATE USING BLDG
       RESET VARIABLE BNAME, CNAME
   ENDIF
   ARRAY
      RECORD POINT USING A_UPD
      IF $FOUND
         RECORD UPDATE USING A_UPD
      ELSE
         FAIL "BAD PROBLEM HERE",TYPE
      ENDIF
   ENDARRAY

*** FUNCTION 8; LABEL="EXIT"
   EXIT
```

DISPLAY [expression]

Places text into the message line of the screen (unless preceded by a SCROLL).

expression	Optional. The expression can be any valid BUILDER expression. If the resulting message is longer than the width of the terminal, it is truncated on the right. If the expression is omitted, the message line is cleared.

The cursor remains positioned after the last character in the message until another operation causes the cursor to move. If a RELATE or operating system command is executed, the cursor will be returned to the left margin in case the command causes output.

EXAMPLES:

```
*** ENTER
    DISPLAY "THE ""%fname"" FILE HAS BEEN LISTED TO THE PRINTER."
*** FUNCTION 1
    L:=$LENGTH(NAME)
    DISPLAY L+10
```

ERASE SCREEN

Immediately clears the entire screen display.

The screen will be redrawn at the next automatic or explicit REFRESH.

EXAMPLES:

```
*** FUNCTION 3; LABEL="PRINT"
   ERASE SCREEN
   PRINT
```

EXIT

Terminates BUILDER.

EXAMPLES:

```
*** ENTER
      .
      .
      .
      IF $ERROR=BADERR
         DISPLAY "CATASTROPHIC ERROR.  CANNOT CONTINUE"
         EXIT
      ENDIF
```

EXITARRAY

Allows early exit from an ARRAY loop.

Normally, an ARRAY loop will only terminate if the highest numbered element of any array variable has been accessed during the latest loop or if no array variable has been accessed during the latest loop. EXITARRAY causes control to pass immediately to the command following the first ENDARRAY after the EXITARRAY.

EXAMPLES:

```
*** LAYOUT
   [TEXT                                    ]
   [TEXT                                    ]
   [TEXT                                    ]
   [TEXT                                    ]
*** ENTER STATE=OLDWAY
   FLAG:=0
   ARRAY
      IF FLAG
         NOTE   Notice I avoided using an array variable
         NOTE   so now array loop will stop.
      ELSE
         IF TEXT=""
            NOTE I assume there's no more text after this.
            FLAG:=1
         ELSE
            NOTE   Do my normal updating & stuff
         ENDIF
      ENDIF

*** ENTER STATE=NEWWAY
   ARRAY
      IF TEXT=""
         EXITARRAY
      ENDIF
      NOTE   Do my normal updating and stuff
   ENDARRAY
```

FAIL [expression] [,variablename]

Terminates in an error state the processing of the section being executed.

expression	Optional. A valid BUILDER expression whose results will be displayed in the message line with FAIL enhancement (see SET OPTION).
variablename	Optional. If provided, the name must be of a variable on the current screen. The specified variable is highlighted as an error variable and the cursor is positioned to that variable. Subsequent use of VERIFY, a function key or a RETURN/ENTER will cause the variable's VARIABLE section to be executed.

EXAMPLES:

```
*** ENTER
   IF PARTNO<=0
      FAIL "PART NUMBER %partno IS INVALID.", PARTNO
   ENDIF
   DISPLAY "PART NUMBER %partno IS VALID."
```

FINISH [expression] [,variablename]

Terminates the processing of the section being executed.

expression	Optional. A valid BUILDER expression whose results are placed into the message line with FINISH enhancement (see SET OPTION).
variablename	Optional. If specified, the cursor will be positioned to this variable after the message is displayed.

EXAMPLES:

```
*** ENTER
    IF REQUEST="A"
       .
       .
       .
       IF NOT $ERROR
          FINISH "Process 'A' has been completed."
       ELSE
          FAIL $ERRMSG
       ENDIF
    ELSEIF PROCESS="B"
       .
       .
       .
       FINISH "Process 'B' has been completed."
    ELSE
       FAIL "Invalid process requested.", REQUEST
    ENDIF
```

IF condition
[ELSEIF condition] [...]
[ELSE]
ENDIF

Allows conditional execution of subsequent commands.

condition	Required. May be any expression containing constants, screen variables or built-in variables as described in the EXPRESSIONS section of this manual. If the condition evaluates to zero or blank, then the condition is false, otherwise the condition is true.

When the condition specified in the IF command is evaluated to be true, the commands following the IF and before the next ELSEIF, ELSE, or ENDIF will be executed. When the condition evaluates to false, the commands following the IF command will be ignored until an ELSEIF, ELSE, or ENDIF command is reached. If an ELSE command follows an IF command, then the statements between the ELSE and the next ENDIF will be executed only if the condition on the IF command was false. Neither an ELSEIF nor an ELSE need be specified. As many ELSEIFs as desired may be used, but each ELSEIF must have a condition associated with it.

The IF command must be totally contained within a section. There may be SET SCREEN or CALL SCREEN commands within the conditional sections of the IF, but the IF, ELSEIF, ELSE, and ENDIF must all be contained in the same section. IF structures can be nested.

EXAMPLES:

```
*** ENTER
    RECORD READ
    IF $EOF
       DISPLAY "NO MORE DATA IN FILE."
    ELSE
       DISPLAY "FOUND DATA IN FILE."
    ENDIF
    IF COST<=0
       FAIL "PRICES MUST BE GREATER THAN ZERO.", COST
    ENDIF
    IF TODAY=1
       DISPLAY "TODAY IS SUNDAY."
    ELSEIF TODAY>=2 AND TODAY<=6
       DISPLAY "TODAY IS A WEEKDAY."
       IF TODAY=4
          SCROLL "THIS IS THE MIDDLE OF THE WEEK."
       ENDIF
    ELSE
       DISPLAY "TODAY IS SATURDAY."
    ENDIF
```

IGNORE [ALL] ERRORS
IGNORE ERROR *error*

Instructs BUILDER to continue processing if any error or a specific error is encountered on the next command.

error	Optional. If specified, the value indicates the number of an error which should be ignored.

Normally, if an error is encountered when BUILDER is run in creator mode, processing will suspend and the creator is given the option of entering debug mode. If an error is encountered when BUILDER is in end-user mode, the error is reported and the section executing is terminated. In either mode, the IGNORE ERROR command will allow the application to take an action based on the error that occurred.

This command is only in effect for the single command immediately following the IGNORE ERROR, and that one command is the only one that can cause $ERROR to be set.

The syntax and operation of BUILDER's IGNORE ERROR command is identical to the syntax and operation of RELATE's IGNORE ERROR command.

The error number can be determined by referencing the $ERROR built-in variable. When the IGNORE command is executed, it will set $ERROR to zero. $ERROR cannot be reset in any other manner.

EXAMPLES:

```
*** ENTER
    IGNORE ALL ERRORS
    OPEN FILE TEST
    IF $ERROR<>0
       FAIL "The 'TEST' file cannot be accessed."
    ENDIF
    IGNORE ALL ERRORS
    RECORD READ
    IF $ERROR<>0
       DISPLAY $ERRMSG
    ENDIF
```

MODIFY VARIABLE variablelist; options

The characteristics of the indicated variables are changed to reflect the options specified.

variablelist	Required. A list of variablenames defined within the application, separated by commas.
options	Required. A list of one or more variable options as described in the DECLARATION section, separated with semicolons (";"). The attributes of the variablename specified are changed to correspond with the options provided. The NAME, ARRAY, LENGTH, and GLOBAL options cannot be specified.

Any display or format modifications made to GLOBAL variables only affect the variables locally and will not be passed back with the variable after a RETURN SCREEN.

EXAMPLE:

```
*** LAYOUT
      CUSTOMER NUMBER:[cust          ]
      ADDRESS:[addr                           ]
      PHONE:[phone          ]
      DID WE SELL TO THEM THIS YEAR?[s]
             TOTAL SALES:[tot_sales]
             BALANCE DUE:[bal       ]
             CREDIT STATUS:[cr       ]
      SALESPERSON:[salesman       ]
*** VARIABLE CUST
   NOTE  Obtain the desired data.
   SELECT * WHERE CUST=%cust
   RECORD READ
   NOTE  Adjust the customer number variable so it cannot be
   NOTE  changed.
   MODIFY VARIABLE CUST;DISPLAY
*** VARIABLE S
   IF S="N"
      NOTE  Then don't allow tabbing to others.
      MODIFY VARIABLE TOT_SALES,BAL,CR;DISPLAY
   ELSE
      MODIFY VARIABLE TOT_SALES,BAL,CR;OPTIONAL
      SET CURSOR TOT-SALES
   ENDIF
```

PAUSE [seconds]

Causes BUILDER to wait for the indicated number of seconds or until a key is struck. The application will continue processing at the command following the PAUSE command.

seconds	Optional. If specified, this must be a valid integer which indicates, in seconds, how long BUILDER should pause. If not specified, BUILDER will pause for one second. Striking a key will terminate the pause.

The PAUSE command is ignored in batch mode.

EXAMPLES:

```
*** ENTER
    DISPLAY "PLACE SPECIAL PAPER IN PRINTER AND PRESS ANY KEY."
    PAUSE 300
    PRINT:P
```

PROMPT "text" [,variablename]

Displays the text in the message line (unless preceded by a SCROLL) and waits for a response. If a variablename is specified, the response is placed into it.

text	Required. A string containing the message that should be displayed.
variablename	Optional. If specified, this must be the name of an existing variable. The user must press RETURN/ENTER before the response is validated and then placed into the variable. If not specified, the system will accept a one character response without waiting for RETURN/ENTER, which is placed into the built-in variable $KEY.

Regardless of the interface mode of the application (LINE, CHARACTER, or BLOCK), the prompt is read in LINE mode. If an escape is included in the reply, BUILDER cannot interpret it until a RETURN is entered. This may cause the terminal in use to enter undesired modes.

If a function key is pressed in response to the prompt and no variable name is specified, the function key is treated as the response to the prompt and $KEY is cleared to blank. If a variablename is specified, then the function section for the new function key is executed and the PROMPT is reissued.

EXAMPLES:

```
*** ENTER
   PROMPT "PRINT FILE?", FILENAME
   OPEN FILE %filename
   PRINT:P
*** FUNCTION 2
   PROMPT "DO YOU REALLY WANT TO EXIT?"
   IF $KEY="Y"
      EXIT
   ENDIF
```

PURGE PARTITION partitionname

Releases the indicated partition.

partitionname	Required. Partitionname must be the name of a partition created with the CREATE PARTITION command. The specified partition is released and all files attached to the partition, if not open in other partitions, are closed. The partition may not subsequently be referenced unless it is recreated.

If the current partition is purged, DEFAULT becomes the current partition.

EXAMPLES:

```
      .
      .
      .
 *** ENTER
     CALL SCREEN UPDATE

 *** SCREEN UPDATE
 *** INITIAL
     CREATE PARTITION EMPS
       OPEN FILE EMPS
       SET INDEX ENUM

 *** FUNCTION 7; LABEL="PREVIOUS|SCREEN"
     PURGE PARTITION EMPS
     RETURN SCREEN
      .
      .
      .
```

RECORD ADD
[USING partitionname]
[;VARIABLES=varname | (varname,fldname)[,varname | (varname,fldname)][...]]

Adds one record to the file(s) referenced by the current path.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has ADD access. If not specified, the current partition is used. A record is added to the file(s) referenced by the current path on the indicated partition. Note that the USING clause, if present, must come before other optional clauses.
varname	Optional. If specified by itself, the varname must be both a valid BUILDER variable and a valid fieldname in the current file. If specified with a fldname, the varname must be a valid BUILDER variable which will be associated with the field specified by fldname. If the VARIABLES= option is used, only those fields listed will be have data added; all other fields will contain zeroes or blanks.
fldname	Optional. The fldname must be a valid fieldname in the current file. A fieldname may be used with more than one varname.

The data for the record is taken from any variables in the application (or only those on the current screen if SET OPTION VARIABLES=LOCAL is set) with names that match fieldnames in the current path (exception: VARIABLES= clause).

If the add is unsuccessful, a BUILDER error will result and the $ERROR built-in variable will be set to reflect the cause of the first error encountered. Additionally, the $EOF built-in variable is set to one if an end-of-file occurred and zero otherwise.

If the current screen is an array screen, the data for the record is taken from the element referenced by the current value of $SUBSCRIPT. If the add occurred within an ARRAY loop, then the data is taken from the variables corresponding to the iteration through the array loop. If all of the array variables used are blank, no record is added and no error results.

EXAMPLES:

```
*** ACTION MAIN
    IGNORE ERRORS
    RECORD ADD
    IF $EOF
       FAIL "Output file is full, the record could not be added."
    ELSEIF $ERROR
       FAIL $ERRMSG
    ENDIF
```

```
*** ACTION PARTS
   ARRAY
      IGNORE ERRORS
      RECORD ADD USING PARTS
      IF $EOF
         FAIL "Output file is full, the record could not be added."
      ELSEIF $ERROR
         FAIL $ERRMSG
      ENDIF
   ENDARRAY

*** ACTION CONDITIONAL
   NOTE Only add data into the PARTNUM and DESCRIPTION fields;
   NOTE  use screen data from the PARTN and DESCRIPTION variables
   RECORD ADD USING PARTS; VARIABLES=(PARTN,PARTNUM),DESCRIPTION
```

RECORD DELETE [USING partitionname]

Deletes one record from the file(s) referenced by the current path.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has DELETE access. If not specified, the current partition is used. The current record on the file(s) referenced by the indicated partition are deleted.

The current record is set with RECORD POINT, RECORD ADD, RECORD NEXT, RECORD REPOINT, and RECORD READ.

EXAMPLES:

```
*** INITIAL
   CREATE PARTITION PARTS
   OPEN FILE PARTS
   SET INDEX PARTNUM
   SET PARTITION DEFAULT
*** ENTER
   RECORD POINT USING PARTS
   IF $FOUND
      RECORD DELETE USING PARTS
      DISPLAY "The part has been deleted."
   ENDIF
```

RECORD NEXT [USING partitionname]

Reads a record from the file referenced by the current path without moving data from the database fields into BUILDER variables.

partitionname	Optional. If specified, the partition name must exist and should reference a path to which the application has READ access. If not specified, the current partition is used. The pointer moves on the file referenced by the current path on the indicated partition without actually reading data.

The $EOF built-in variable is set to 1 if the RECORD NEXT causes an end of file; otherwise it is set to 0. An end-of-file is maintained by BUILDER for each partition. Subsequent reads from the same partition will continue to return an end-of-file until the end-of-file is reset. The end-of-file is reset if the current path on the partition is changed or if a RECORD POINT, RECORD REPOINT, RECORD RESET, or RECORD REWIND is performed.

EXAMPLES:

```
*** INITIAL
    OPEN FILE CLASSES
    SET INDEX ROOM
*** ENTER
    RECORD REWIND; BREAK=1
    RECORD READ
    WHILE NOT $EOF
        SCROLL "%room,%location"
        WHILE NOT $EOF
            NOTE  Skip all other classes in this same room.
            RECORD NEXT
        ENDWHILE
        RECORD RESET
        RECORD READ
    ENDWHILE
```

RECORD POINT [USING partitionname]
[;KEY=variablelist]
[;BREAK=num]

Uses the current index on the current path to locate a specific record.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has READ access. If not specified, the current partition is used. A record containing the indicated key values is searched for in the file(s) referenced by the current path on the indicated partition. Note that the USING clause, if present, must come before other optional clauses.
KEY	Optional. If specified, the keyword must be followed by one or more variable names separated by commas. The content of the variables is used to construct the key. If not specified, all variables which match fields in the current index will make up the key.
BREAK	Optional. If specified, the keyword must be followed by a numeric value which indicates the number of fields in the key which should be considered to cause a control break. The value cannot exceed the number of fields in the current index. An end-of-file will be returned on a record read, instead of reading a record, as soon as one of the values of these fields changes. To read the file beyond the key pointed to, BREAK should be specified as zero. If not specified and KEY is not specified, BREAK is assumed to be zero. If BREAK is not specified and KEY is specified, BREAK is assumed to be the number of fields in the variablelist.

The search key used for the point is constructed from data in variables on the current screen or from the supplied variablelist. If a field in the current index does not appear on the screen, and the variablelist is not supplied, the point operation will fail to find a record and an error will result.

If a record is found in the file that matches the key constructed from the current screen then the built-in variable $FOUND is set to one, $EOF is set to zero, and the current record pointer is moved to that record. If a record cannot be found, $FOUND is set to zero; in addition, $EOF is set to one if a non-zero BREAK was specified and zero otherwise.

If an exact match is not found, RECORD RESET can be used to turn off $EOF (except in IMAGE) and a RECORD READ will return the record containing the next higher value of the key.

Any BREAK specified will remain in effect until another POINT, a RECORD RESET, a RECORD ADD, or any RELATE command is done or an attempted RECORD READ beyond the end of the file causes $EOF to be set. In any of these cases, the break will be reset to zero.

EXAMPLES:

```
*** SCREEN S1
*** LAYOUT
    PART #:[PARTNO    ]   DESCRIPTION:[DESC              ]
                          QUANTITY ON HAND:[QOH    ]

*** INITIAL
   OPEN FILE PARTS
   SET INDEX PNO
*** VARIABLE PARTNO
   NOTE  Observe that the field in the file is PNO and the
   NOTE   variable is PARTNO.
   RECORD POINT KEY=PARTNO
   IF NOT $FOUND
      STATUS:=3
      FINISH "NO SUCH PART NUMBER"
   ELSE
      RECORD READ
   ENDIF

*** SCREEN S2
*** LAYOUT
 CUSTOMER #          INVOICE              DATE           AMOUNT
   [CNO  ]          [INVNO        ]   [DATE      ]   [AMT         ]
   [CNO  ]          [INVNO        ]   [DATE      ]   [AMT         ]
   [CNO  ]          [INVNO        ]   [DATE      ]   [AMT         ]
   [CNO  ]          [INVNO        ]   [DATE      ]   [AMT         ]

   TYPE ONE CNO AND INVOICE NUMBER, THEN PRESS RETURN TO READ
*** INITIAL
   OPEN FILE INVOICES
   SET INDEX CNO, INVNO
*** ENTER
   RECORD POINT; BREAK=2
   IF NOT $FOUND
      FINISH "CUST #%CNO, INVOICE #%INVNO NOT FOUND."
   ELSE
      ARRAY
         RECORD READ
      ENDARRAY
      IF $EOF
         IF $BREAK<3
            SCROLL "END OF INVOICE #%invno"
         ENDIF
         IF $BREAK=1
            SCROLL "END OF CUST #%cno"
         ENDIF
         RECORD RESET
         NOTE  The RESET allows reading next time thru loop
      ENDIF
   ENDIF
```

```
                         RECORD READ
                      [USING partitionname]
[;VARIABLE=varname | (varname,fldname)[,varname | (varname,fldname)][...]]
```

Reads a record from the file(s) referenced by the current path.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has READ access. If not specified, the current partition is used. One record, at the current index pointer, is read from the file(s) referenced by the current path on the indicated partition. Note that the USING clause, if present, must come before other optional clauses.
varname	Optional. If specified by itself, the varname must be both a valid BUILDER variable and a valid fieldname in the current file. If specified with a fldname, the varname must be a valid BUILDER variable which will be associated with the field specified by fldname. If the VARIABLE$\overset{S}{\wedge}$= option is used, only those BUILDER variables listed will have values read into them.
fldname	Optional. The fldname must be a valid fieldname in the current file.

Each RECORD READ reads one record from the current path in the current (or requested) partition. If the record at the current index pointer has not yet been read, it will be; otherwise, the current record pointer will be moved to the next record and that one will be read. The current record pointer is affected by RECORD POINT, RECORD REWIND, SHOW VARIABLES, $READ, and any RELATE command.

Any fields from the current path with the same name as variables in the application (or only those on the current screen if SET OPTION VARIABLES=LOCAL is set) are copied into the screen variables (exception: VARIABLE$\overset{S}{\wedge}$= clause).

The $EOF built-in variable is set to one if the read caused an end of file, otherwise it is set to zero. Except within an ARRAY loop, an attempted read resulting in the setting of $EOF does not alter the content of any BUILDER variables. An end-of-file is maintained by BUILDER for each partition. Subsequent reads from the same partition will continue to return an end-of-file until the end-of-file is reset. The end-of-file is reset if the current path on the partition is changed or if a RECORD POINT, RECORD REPOINT, RECORD RESET, or RECORD REWIND is performed.

$EOF is also set to one if the READ would cause a change in values of fields in the keylist previously set by a RECORD POINT, RECORD REPOINT, or RECORD REWIND. The $BREAK system-defined variable can be checked to ascertain which field caused the break.

If the current screen is an array screen, the data from the record is copied into the element referenced by the current value of $SUBSCRIPT. If the read occurred within an ARRAY loop, then the data is copied into the variable element corresponding to the iteration through the array loop. Each time an $EOF is encountered during an ARRAY loop, all referenced BUILDER array variables whose names are fields in the file are set to blanks.

EXAMPLES:

```
*** DECLARATIONS
    TODAY;DATE="M/D/Y"

*** INITIAL
    SELECT TODAY:=$DATE
    RECORD READ
    SELECT

*** ENTER
    RECORD READ
    IF $EOF
       DISPLAY "No additional records exist in the file."
    ENDIF
    ARRAY
       RECORD READ USING PARTS
    ENDARRAY
```

RECORD REPOINT [USING partitionname]
[;KEY=variablelist]
[;BREAK=num]

Uses the current index on the current path to relocate a record that has previously been read to preserve the multi-user checksum.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has READ access. If not specified, the current partition is used. A record containing the indicated key values is searched for in the file(s) referenced by the current path on the indicated partition. Note that the USING clause, if present, must come before other optional clauses.
KEY	Optional. If specified, the keyword must be followed by one or more variable names separated by commas. The content of the variables is used to construct the key. If not specified, all variables which match fields in the current index will make up the key.
BREAK	Optional. If specified, the keyword must be followed by a numeric value which indicates the number of fields in the key which should be considered to cause a control break. The value cannot exceed the number of fields in the current index. An end-of-file will be returned on a record read, instead of reading a record, as soon as one of the values of these fields changes. To read the file beyond the key pointed to, BREAK should be specified as zero. If not specified and KEY is not specified, BREAK is assumed to be zero. If BREAK is not specified and KEY is specified, BREAK is assumed to be the number of fields in the variablelist.

The search key used for the point is constructed from data in variables on the current screen or from the supplied variablelist. If a field in the current index does not appear on the screen, and the variablelist is not supplied, the point operation will fail to find a record and an error will result.

If a record is found in the file that matches the key constructed from the current screen then the built-in variable $FOUND is set to one and $EOF is set to zero. If a record cannot be found $FOUND is set to zero; in addition, $EOF is set to one if a non-zero BREAK was specified.

If an exact match is not found, RECORD RESET can be used to turn off $EOF (except in IMAGE) and a RECORD READ will return the record containing the next higher value of the key.

This command is only useful for access to an updatable path open in shared mode. It allows BUILDER to find a record that has previously been read and verifies that no other user has altered the record before an UPDATE or a DELETE is done. The checksums that indicate the status of a record are maintained on a segment basis, so if the file is opened in different partitions within the same segment, the checksums will be properly maintained.

EXAMPLES:

```
*** SCREEN XYZ
*** LAYOUT
  PART NUMBER                   DESCRIPTION                      QUANTITY ON HAND
     [PNO     ]            [DESC                 ]                  [QOH    ]
     [PNO     ]            [DESC                 ]                  [QOH    ]
     [PNO     ]            [DESC                 ]                  [QOH    ]
     [PNO     ]            [DESC                 ]                  [QOH    ]
     [PNO     ]            [DESC                 ]                  [QOH    ]
*** INITIAL
  OPEN FILE PARTS; MODE=SHARE
  SET INDEX PNO
  ARRAY
     RECORD READ
  ENDARRAY
  NOTE   The first few records have been read into the
  NOTE       screen.  The user can now go out to lunch.

*** FUNCTION 1; LABEL="NO ERROR"
  NOTE   In order to update the records on the screen,
  NOTE      the current record pointer must be returned to
  NOTE      each record in turn.
  ARRAY
     RECORD POINT
     IGNORE ERRORS
     RECORD UPDATE
     IF $ERROR
        DISPLAY $ERRMSG
     ENDIF
  ENDARRAY
  NOTE   This method will find the record and make the update,
  NOTE       even if some records were updated by other people
  NOTE       while the user was out to lunch.

*** FUNCTION 2; LABEL="ERROR"
  ARRAY
     RECORD REPOINT
     IGNORE ERRORS
     RECORD UPDATE
     IF $ERROR
        DISPLAY $ERRMSG
     ENDIF
  ENDARRAY
  NOTE   With this method, an attempted UPDATE on a record
  NOTE      that another person altered while the user was
  NOTE       out to lunch will generate an error.
```

RECORD RESET [USING partitionname]

Resets $EOF and the end-of-file indicator on the current (or specified) partition. A subsequent RECORD READ on the partition will return the next record in the file.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has READ access. If not specified, the current partition is used.

This command is designed to be used in conjunction with RECORD POINT with a non-zero BREAK. Since reaching a break causes an end-of-file condition, and BUILDER will not read from a path with an end-of-file condition, RECORD RESET needs to be called in order to continue reading after a break has occurred. This command does not alter the current BREAK and KEY settings, nor does it alter the current position of the record pointer.

EXAMPLES:

```
*** FUNCTION 1 ; LABEL="INITIAL"
   NOTE   The KEY= gives me an implied BREAK=1
   RECORD POINT KEY=ZIPCODE
   IF NOT $FOUND
      FINISH "NO SUCH ZIP CODE"
   ENDIF
   ARRAY
      RECORD READ
   ENDARRAY

*** FUNCTION 2 ; LABEL="MORE"
   IF $EOF
      NOTE   This means sometime through the last loop, $EOF was set
      RECORD RESET
   ENDIF
   ARRAY
      RECORD READ
      IF $SUBSCRIPT=0 AND $EOF
         FINISH "NO MORE IN FILE"
      ENDIF
   ENDARRAY
```

RECORD REWIND [USING partitionname] [;BREAK=fields]

The current path on the indicated partition is rewound and the end-of-file indicator for the partition is reset. A subsequent read from the path will return the first record in the path.

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has READ access. If not specified, the current partition is used.
BREAK	Optional. If specified, the keyword must be followed by a numeric value which indicates the number of fields in the key which should be considered to cause a control break. The value cannot exceed the number of fields in the current index. An end-of-file will be returned as soon as one of the values changes. To read the portion of the file beyond the key pointed to, BREAK should be specified as zero. If not specified, BREAK is assumed to be zero.

EXAMPLES:

```
*** FUNCTION 1;LABEL="NEXT"
    NOTE  Read another record
    RECORD READ
    NOTE  If at end of file, return to beginning
    IF $EOF
       RECORD REWIND
       RECORD READ
    ENDIF
```

RECORD UPDATE
[USING partitionname]
[;VARIABLES=varname|(varname,fldname)[,varname|(varname,fldname)][...]]

Updates the current record (as established by a previous RECORD READ, RECORD NEXT, RECORD POINT, RECORD REPOINT, SHOW VARIABLES, $READ function, or any RELATE command).

partitionname	Optional. If specified, the partitionname must exist and should reference a path to which the application has UPDATE access. If not specified, the current partition is used. A record is updated on the file(s) referenced by the current path on the indicated partition. Note that the USING clause, if present, must come before any other optional clauses.
varname	Optional. If specified by itself, the varname must be both a valid BUILDER variable and a valid fieldname in the current file. If specified with a fldname, the varname must be a valid BUILDER variable which will be associated with the field specified by fldname. If the VARIABLES= option is used, only those fields listed will be modified; the values in all other fields will remain the same.
fldname	Optional. The fldname must be a valid fieldname in the current file.

Any variables in the application (or only those in the current screen if SET OPTION VARIABLES=LOCAL is used) that have the same name as fields from the current path are copied into the records being updated. All other fields in the record remain unchanged (exception: VARIABLES= clause).

EXAMPLES:

```
*** ENTER
    RECORD POINT
    IF NOT $FOUND
       DISPLAY "The requested record cannot be located."
    ELSE
       RECORD UPDATE
    ENDIF
```

REFRESH [SCREEN]

Instructs BUILDER to update the content of the screen to reflect the current values of variables in the application.

SCREEN	Optional. If specified, the keyword instructs BUILDER to redraw the entire screen as opposed to simply updating its content. This action is normally only required after output to the terminal has been done by anything other than BUILDER and RELATE and the original screen has been destroyed or moved.

If REFRESH is executed in the INITIAL section it will also cause the screen to be displayed. Normally, BUILDER does not update the screen until the end of a section is reached. In some situations (particularly those involving a long processing delay) periodically updating the screen provides a friendlier system.

The PROMPT command will not automatically cause a REFRESH. If intermediate results need to be displayed, a REFRESH should be performed before the PROMPT.

EXAMPLES:

```
*** INITIAL EVERYTIME
    NOTE  Read next customer
    RECORD READ
    NOTE  The new customer data is not yet displayed
    REFRESH
    PROMPT  "Is this customer OK?"
    IF $KEY <> "Y"
       CALL SCREEN SPECIAL_REPORT
    ENDIF
    NOTE  This is where an automatic refresh is done
*** ENTER
    :RUN BIGPRNT
    REFRESH SCREEN
```

RESET [ALL|RECORD [USING partitionname]|SCREEN|VARIABLE variablenamelist [;SUBSCRIPT]]

Resets the value returned from the $CHANGED function for various groups of variables.

ALL	Optional. If specified, all the variables in the application will no longer be flagged as having been changed by the user.
RECORD	Optional. If specified, all the variables in the current screen which also exist in the current path in the current partition are no longer flagged as having been changed by the user.
USING	Optional. If specified, the keyword must be followed by the name of an existing partition. All variables in the screen which also exist in the current path on the specified partition are no longer flagged as having been changed by the user.
SCREEN	Optional. If specified, all the variables in the current LAYOUT are no longer flagged as having been changed by the user.
VARIABLE	Optional. Resets the indicated variables. If an array variable name is specified, all subscripts of the variable are reset.
SUBSCRIPT	Optional. If specified, only variables that exist with the current subscript value are reset. The keyword must be used in combination with the other keywords. For example, RESET RECORD;SUBSCRIPT will reset all variables in the current path for the current subscript.

If no keywords are specified, SCREEN is assumed.

This command has no effect on the $CHANGED built-in variable, which indicates if ANY variables have been changed by the user. To reset the $CHANGED built-in variable, its value must be set to zero.

EXAMPLES:

```
*** ENTER
   IF $CHANGED("ADDRESS") OR $CHANGED("CITY")
      RECORD ADD USING NEWADDR
      RESET VARIABLE ADDRESS,CITY
   ELSE
      RECORD UPDATE USING CUSTOMER
      $CHANGED:=0
      RESET SCREEN
   ENDIF
```

RETURN SCREEN

Causes a return from a previously executed CALL SCREEN command.

If more than one CALL SCREEN command is pending, then the last one executed is the one returned to. If no CALL SCREEN commands are pending, then this command returns an error.

EXAMPLES:

```
*** ENTER
   DISPLAY "Calling SCREENB."
   CALL SCREEN SCREENB
   DISPLAY "Returned from SCREENB."
*** SCREEN SCREENB
*** LAYOUT
   .
   .
   .
*** ENTER
   DISPLAY "Processing SCREENB."
   .
   .
   .
   RETURN SCREEN
```

SCROLL [expression]

Scrolls a message at the bottom of the screen.

expression	Optional. The expression can be any valid BUILDER expression. If the result of the expression is longer than the width of the terminal, the message is split into two or more lines, and each line is scrolled separately onto the bottom of the screen. Except when a word fills the entire line, a string is always split between words. If the expression is omitted, a blank line is placed at the bottom of the screen.

Unlike the DISPLAY command, the SCROLL command can be used to write multi-line messages. The SCROLL command will scroll the screen up one line and write the message on a new line at the bottom of the screen.

The cursor remains positioned after the last character in the message until another operation causes data output.

Any PROMPT or DISPLAY after a SCROLL in the same section will also cause the screen to scroll. The screen will be redrawn automatically at the end of the section.

EXAMPLES:

```
*** ENTER
    RECORD READ
    WHILE NOT $EOF
        SCROLL "NAME=%name, ADDRESS=%address"
        RECORD READ
    ENDWHILE
```

SET CURSOR variablename

Positions the cursor to the beginning of the indicated variable.

variablename	Required. The variablename must be an OPTIONAL, REQUIRED, or POSITION variable on the current screen.

The SET CURSOR command also causes a REFRESH to be executed.

EXAMPLES:

```
 *** LAYOUT
     COMMAND: [CLINE]

     INVOICE NUMBER:        AMOUNT:          DATE:
     [INVNO       ]         [AMT      ]      [DATE      ]
     [INVNO       ]         [AMT      ]      [DATE      ]
     [INVNO       ]         [AMT      ]      [DATE      ]
     [INVNO       ]         [AMT      ]      [DATE      ]
 *** INITIAL EVERYTIME
    SET CURSOR CLINE
 *** ENTER
   PROMPT "WHICH INVOICE (1-4)?",X
   $SUBSCRIPT:=X-1
   SET CURSOR INVNO
 *** DECLARATIONS
    X: NUMERIC

 *** SCREEN REPORT
 *** LAYOUT
        PRESS RETURN TO GENERATE REPORT

        DEPARTMENT DESIRED:[DNO           ]

        JOB NUMBER:         [JNO         ]

 *** DECLARATION
    JNO: POSITION
    DNO: REQUIRED

 *** ENTER
    SET CURSOR JNO
    REFRESH
    STREAM USER.ACCT
       OPEN FILE X
       SELECT * WHERE DEPT=%DNO
       EXECUTE XRPT
    ENDSTREAM
```

```
SET OPTION
[DATE = "dateformat"]
[ENHANCE = (type = enhlist)]
[MESSAGE = CLEAR | KEEP]
[QUOTES = ON | OFF | YES | NO]
[SUBSTITUTE = ON | OFF | YES | NO]
[TAB = CLEAR | SKIP | DEFAULT]
[TIMEOUT = seconds]
[VARIABLE = LOCAL | GLOBAL]
[VERIFY = [ENTER[,FUNCTION]] | NONE]
```

Selects operational characteristics of the BUILDER application.

DATE	Optional. If specified, the keyword must be followed by a valid date format specification. Subsequent references to the $DATE built-in variable will return the current date in the format specified. Also, any additional variables declared as DATE with no format specified will use this format.
ENHANCE	Optional. If specified, the keyword must be followed by the item to enhance and the enhancement mode. The type must be one of the following:

ACTION	The default enhancement for ACTION variables.
BUILDER	The default enhancement for messages automatically emitted by BUILDER (e.g., syntax error messages).
DISPLAY	The default enhancement for DISPLAY variables.
ERROR	The enhancement for variables in error.
FAIL	The display enhancement for messages generated by the FAIL command.
FINISH	The display enhancement for messages generated by the FINISH command.
MESSAGE	The display enhancement for messages generated by the DISPLAY and SCROLL commands.
OPTIONAL	The default enhancement for OPTIONAL variables.
POSITION	The default enhancement for POSITION only variables.
PROMPT	The default enhancement for the text contained within a prompt.
REQUIRED	The default enhancement for REQUIRED variables.

The type keyword must be followed by one or more of the keywords (or their abbreviations): BLINKING, HALFBRIGHT, INVERSE, UNDERLINE or NONE, separated by commas.

The default enhancement types are determined by the configuration of the terminal driver.

MESSAGE

Optional. If specified, the keyword must be followed by either CLEAR or KEEP. CLEAR causes anything displayed in the message line to be cleared as soon as a TAB, RETURN/ENTER, or function key is used. KEEP causes anything displayed in the message line to remain until something else is displayed. The new setting remains in effect for all screens until is is explicitly altered. KEEP can prevent function key labels from being displayed on terminals where the labels are displayed on the message line.

The default is CLEAR.

QUOTES

Optional. If specified, the keyword must be followed by YES, NO, ON, or OFF. If NO or OFF is specified, the substitution algorithms will not replace a quote (") with two quotes (""). If YES or ON is specified, the substitution is enabled. This applies only to the current screen.

The default is ON.

SUBSTITUTE

Optional. If specified, the keyword should be followed by YES, NO, ON or OFF. If NO or OFF is specified, subsequent commands are evaluated without the substitution algorithm that is normally performed. If YES or ON is specified, the substitution algorithm is enabled. This applies only to the current screen.

The default is ON.

TAB

Optional. If specified, the keyword should be followed by CLEAR, SKIP, or DEFAULT. The new setting for this option becomes the default TAB option for all variables created after this command has executed (on CALLED or SET screens). It also overrides any previous setting for a GLOBAL variable redeclared after this command has executed. If CLEAR is specified, after this command, when a TAB is pressed to jump to the next variable, all characters in the variable after the cursor will be set to blanks before jumping to the next variable. If SKIP is specified, when a TAB is pressed, the cursor skips to the next variable without changing any information in the original variable. Specifying DEFAULT uses the BUILDER default for TAB.

The default is SKIP for LEFT justified variables and CLEAR for CENTER or RIGHT justified variables and dates.

TIMEOUT

Optional. If specified, the keyword must be followed by either NONE or an integer indicating the number of seconds to wait for a user to press a key before executing a TIMEOUT section. Setting TIMEOUT equal to zero turns off the timeout checking.

The TIMEOUT value applies only to the current screen. A SET OPTION TIMEOUT in the GLOBAL INITIAL section affects only the first screen in the application and PROMPTS in the GLOBAL INITIAL section.

If there is a TIMEOUT section in the screen, that TIMEOUT section will be executed; otherwise, the GLOBAL TIMEOUT section will be used. If neither TIMEOUT section can be found when a time-out occurs, an error will be generated.

VARIABLE	Optional. If specified, the keyword must be followed by the keyword LOCAL or GLOBAL. Use of GLOBAL indicates that any variable in a called screen that is not declared as a GLOBAL may still reference a global variable if its name is the same and it doesn't appear in the LAYOUT or DECLARATION. If LOCAL is specified, no variable will be treated as global unless it is declared as GLOBAL in the declaration section. The default is GLOBAL.
VERIFY	Optional. If specified, the keyword must be followed by the keyword ENTER or FUNCTION (or both) or the keyword NONE. Use of the keywords ENTER or FUNCTION will cause BUILDER to verify the contents of all variables on the screen when RETURN/ENTER or a function key is pressed. If NONE is specified, this verification is not performed and the VERIFY command must be used if verification is desired. This command applies only to the current screen. The default is ENTER, FUNCTION.

EXAMPLES:

```
*** ENTER
    SET OPTION ENHANCE=(ACTION=I,B)
    SET OPTION SUBSTITUTE=OFF
    SELECT CONSTANT="%fname=",FNAME
    SET OPTION SUBSTITUTE=ON
    SET OPTION TIMEOUT=60
    PRINT

*** TIMEOUT
    SCROLL "LOGGING YOU OFF THE SYSTEM"
    EXIT
```

SET PARTITION partitionname

The partition with the specified name becomes the current partition.

partitionname	Required. The given partitionname is made the current partition.

The partition must have been previously created with the CREATE PARTITION command. The special partitionname DEFAULT can be used to reference the partition automatically created by BUILDER.

EXAMPLES:

```
*** INITIAL
    CREATE PARTITION PARTS
    OPEN FILE PARTS
    CREATE PARTITION CUSTOMER
    OPEN FILE CUST
    SET PARTITION DEFAULT
*** ACTION 1
    SET PARTITION PARTS
    SET INDEX PARTNUM
    SET PARTITION DEFAULT
*** ACTION 2
    RECORD READ USING CUST
    SET PARTITION PARTS
    SET INDEX CNUM
    RECORD POINT
    WHILE NOT $EOF
       RECORD READ
    ENDWHILE
    SET PARTITION DEFAULT
```

SET SCREEN screenname

Displays a new screen.

screenname	Required. This screenname must be the screen name from a SCREEN delimiter line from within the current application file, or the name of an application file in the same group and account as the original application file. MPE FILE equations are not referenced.

The SET SCREEN command is executed immediately. No commands following the command in the section will be executed. All variables defined on the screen are removed and can no longer be accessed. The SET SCREEN command executes the initialization section of the new screen, gets the new screen layout, displays the new screen, and waits for user input. If the new screen initialization section contains an error or there is an error in the screen layout, then the SET SCREEN command returns an error.

In general, do not use SET SCREEN to exit a screen that has been entered using CALL SCREEN. Instead, use RETURN SCREEN.

EXAMPLES:

```
*** ENTER
    SET SCREEN SCR6
    DISPLAY  "This DISPLAY command will never be executed"
```

SET STATE statename

Changes the state of the current screen.

statename	Required. Name of new state. A state name must begin with a letter, contain only letters, digits, and underscores ("_"), and be no more than 15 characters long.

Changing the state will cause the DECLARATION section associated with the state to be evaluated, the INITIAL section for the STATE to be executed, and the function key labels to be redrawn to reflect the new state. Any variables declared as ACTION variables must be redeclared as ACTION variables if the state has a DECLARATION section of its own, or they will no longer be ACTION variables.

Whenever the RETURN/ENTER key or a FUNCTION key is pressed, the ENTER or FUNCTION section associated with the current state is executed.

EXAMPLES:

```
*** LAYOUT
  EMPLOYEE NUMBER: [num          ]
  LAST NAME:       [last          ] FIRST NAME: [first]
  DATE HIRED:      [h_date         ]
  DATE TERMINATED: [t_date         ]
*** DECLARATION
   H_DATE,T_DATE;DATE
   NUM;NUMERIC;RANGE=1/

*** DECLARATION STATE=ADD
   NUM,H_DATE,LAST,FIRST; REQUIRED
   T_DATE; DISPLAY

*** DECLARATION STATE=CHANGE
   NUM,H_DATE; DISPLAY
   FIRST,LAST,T_DATE; OPTIONAL

*** DECLARATION STATE=SEARCH
   NUM;OPTIONAL
   LAST, FIRST, H_DATE, T_DATE; DISPLAY

*** INITIAL STATE=SEARCH
   CLEAR SCREEN

*** FUNCTION 1
   SET STATE ADD

*** FUNCTION 2
   SET STATE CHANGE
```

```
*** FUNCTION 3
   SET STATE SEARCH

*** FUNCTION 4; STATE=ADD
   RECORD ADD

*** FUNCTION 4; STATE=CHANGE
   RECORD UPDATE

*** FUNCTION 5; STATE=SEARCH
   RECORD POINT
   IF $FOUND
      RECORD READ
   ENDIF
```

SHOW PARTITIONS

Displays the names of all partitions that currently exist. This command is most useful in debugging applications.

EXAMPLES:

```
    *** ENTER
        CREATE PARTITION PARTS_PART
        OPEN FILE PARTS
        CREATE PARTITION CUSTOMER_PART
        OPEN FILE CUSTOM
        SHOW PARTITIONS

  Displays:

        PARTS_PART
        DEFAULT
        CUSTOMER_PART (Current)
```

STREAM [logon]
ENDSTREAM

Allows a job to be streamed from BUILDER.

logon	Optional. If specified, the logon must contain at least a valid user and account name and any required passwords. The logon may contain all parameters available on the MPE JOB command (excluding the keyword JOB). If not specified, the entire JOB command should immediately follow the STREAM command.

If logon is included, the commands between the STREAM and ENDSTREAM commands must be RELATE commands. The BUILDER stream facility will create a job containing the logon, the command RELATE, the commands between the STREAM and ENDSTREAM commands, an EXIT and an !EOJ.

If logon is not included, the commands between the STREAM and ENDSTREAM commands are placed directly into a file and streamed. The STREAM command should be followed by a JOB command; the ENDSTREAM command should be preceded by an EOJ.

Before the commands are streamed, the substitution algorithm is applied. This can be used to place passwords, program names, etc., into the job. If a more flexible mechanism is required, the SUBMIT command can be used.

When the STREAM command is executed, BUILDER creates a file called BLDRJOB in the session temporary domain. Any existing file of the same name is purged. When the ENDSTREAM command is executed, the file is closed and streamed. If desired, the file can then be saved and renamed by the application.

The STREAM and ENDSTREAM commands must be entirely contained within a single action section of an application.

EXAMPLES:

```
*** ENTER
    STREAM DATA.SALES/%acctpass;OUTCLASS=FASTLP
       OPEN FILE SLSDATA
       REPORT USING SLSREP1
    ENDSTREAM
```

The following example will produce the same results:

```
*** ENTER
    STREAM
       !JOB DATA.SALES/%acctpass;OUTCLASS=FASTLP
       !RELATE
       OPEN FILE SLSDATA
       REPORT USING SLSREP1
       EXIT
       !EOJ
    ENDSTREAM
```

```
   >EXIT
   >!EOJ
   SUBMIT JOB
*** FUNCTION 2
   OPEN FILE SALES
   >PAGE HEADING=("SALES BY REGION"),("AS OF %today")
   >FIELDS=(REGION),(TOTAL)
   >GO
   SUBMIT REPORT:P
*** FUNCTION 3
   PROMPT "FILENAME TO SAVE LETTER AS?",fname
   >Ace Services
   >110 South Central
   >CHICAGO, IL
   >%today
   >
   >DEAR %contact:
   IF STATUS="PAST DUE"
      >%latemsg
   ELSE
      >%stdmsg
   ENDIF
   >
   >SINCERELY,
   >
   >%user
   SUBMIT FILE
   :SAVE BLDRSBMT
   :PURGE %fname
   :RENAME BLDRSBMT, %fname
   DISPLAY "LETTER ""%fname"" HAS BEEN CREATED."
```

VERIFY

Validates the contents of all variables.

Compares all variables against the requirements given in the DECLARATIONS section and verifies that the VARIABLE sections for all variables whose values have changed have been executed. The execution of this command will verify that numeric variables only contain numeric data, that variables with ranges are within the specified bounds, and so forth.

VERIFY is normally executed automatically whenever a FUNCTION key or RETURN/ENTER is pressed. This can be changed by the SET OPTION VERIFY command, which can disable automatic verification in either or both instances.

When a screen variable is changed by the user, BUILDER will always attempt to verify the content of the variable. The content is checked against any RANGE (as specified in the DECLARATIONS) and any associated VARIABLE section is executed. If an error is encountered, the variable is highlighted and the cursor is returned to the variable. If the value is changed again, the variable is reverified. The user may TAB out of a field that has failed verification. When RETURN/ENTER or a function key is pressed, or VERIFY is executed, all variables on the screen that failed verification are again verified.

EXAMPLES:

```
*** LAYOUT

    NAME OF FILE TO OPEN:[fname        ]
       LOCKWORD:[lockword  ]
        PRESS F1 TO EXIT

*** DECLARATIONS
    FNAME,LOCKWORD;REQUIRED

*** INITIAL
    SET OPTION VERIFY=NONE

*** FUNCTION 1
    NOTE  This function can be executed because no automatic
    NOTE     verification is done.
    EXIT

*** ENTER
    NOTE  Make sure verification is done sometime.
    VERIFY
    OPEN FILE %fname/%lockword
```

SECTION 5

D E S I G N I N G
S C R E E N - B A S E D   A P P L I C A T I O N S

CREATING AN APPLICATION

The process of creating and debugging an application is a straightforward three-step process.

1. Using HP's EDITOR or QEDIT (or any other editor that creates a compatible file), create a file containing screen delimiters, BUILDER commands, and RELATE commands. Keep this file under any name. This is your "application file".

2. Run BUILDER in CREATOR mode (this can usually be done by simply typing BUILDER). When BUILDER requests "Application File:", respond with the name of the file you created above.

3. If BUILDER detects an error and you are running in CREATOR mode, it places you in BUILDER Debug mode. From there you can edit your application file or use BUILDER or RELATE commands to adjust your data or the application, then resume or restart the execution of your application.

The ability to debug interactively from debug mode using an editor and the BUILDER interpreter makes debugging simple and fast. Using the editor, you can make changes directly to your application and resume execution at the point the error was detected.

If you choose to enter RELATE from debug mode, you are actually entering the BUILDER interpreter interactively. You can now enter any RELATE command and any BUILDER command exactly as you would when placing the command into your application file. This includes specifying variables to be substituted into commands.

For example, if you have a variable on your screen called TEXT, and you are in the BUILDER interpreter, you can ascertain its value in a number of ways:

```
)DISPLAY TEXT
)DISPLAY "The value of text is %text"
)SHOW VARIABLE TEXT
```

You can see what files are currently open:

```
)SHOW PATHS
```

You can change the values of variables or perform actions just as you could in your application file.

```
)OPEN FILE %filename
)RECORD READ
)DISPLAY NAME
)NAME:="MARY"
)RECORD UPDATE
```

The consistent use of standard practices and techniques simplifies the development of applications. The list of standards need not be long to be effective. The standards should address basic techniques for transfer of control and processing, the use of function keys, internal application documentation, readability factors such as indentation and utilization of upper and lower case characters, the order of sections in a screen, etc. The layout of the screens is also of importance. This includes the use and location of screen names, screen headings, version numbers, and variable enhancements. This section describes a number of techniques which can be used in developed applications.

Application Security

BUILDER allows you to protect an application or parts of an application from unauthorized use. The four principle methods of securing an application are described below. An assumption is made that when a system is delivered to the end-user most of the screens used in the system will have been placed into a single application file.

The simplest way of securing an application involves using MPE for logon passwords, possibly UDCs to start the application and possibly a lockword on the application file. This method takes very little time to set up or alter and requires little additional input by the user. The primary drawback of this method is that users can only be restricted on an application basis and cannot be restricted to individual functions within an application.

A slightly more complex and flexible method is to restrict certain functions and operations by the logon name of the user. In this case, the user need not learn any new passwords. Because checks must be made for each user at each restricted function, this method can create problems if the number of users becomes large. Additionally, the application file must be edited when the security matrix needs to be changed, which could produce errors.

An alternative to the above method is to request and verify passwords when an attempt is made to execute a restricted function. This method is easy to implement but requires that the user remember one or more application specific passwords. As with the above method, a new application file is required if the security matrix must be altered.

The final method results in the most flexible access control. The method consists of a file into which the security matrix is placed. The matrix consists of at least the screen name from which the function is requested, the function name, and the user names of those allowed to perform the function. Before a function is performed, this file is checked to verify that the user has the appropriate authorization. This method adds slightly more code to the application than the others but results in greater flexibility. Specifically, access capabilities can be changed by adding, deleting, or updating the authorization file, and do not require a re-release of the application. If these changes are made frequently, it is possible to add an area to the application, appropriately secured, to assist with this task. Another major benefit of this approach is the ability to get a complete summary of the security situation simply by looking at the file.

The first method can continue to be implemented with CALL SCREENs. The second method would likely require the use of SET SCREEN commands, rather than attempting to CALL through several levels of intermediate screens based on the entered text, especially since the latter requires that you also RETURN SCREEN through the same levels. In general, SET SCREEN allows implementation of non-hierarchical systems, or the use of non-hierarchical access through a hierarchical system. The use of SET SCREEN will increase the number of "paths" through an application and thus make it more difficult to test. Also, a SET SCREEN, SET SCREEN sequence will execute more slowly than a CALL SCREEN, RETURN SCREEN sequence.

The nesting level of CALLs is finite. It is possible to construct an application so that a CALL is made, but a RETURN is never executed. This will eventually cause a table overflow in BUILDER. If a CALL SCREEN is used, make sure that the only way to return is through the use of RETURN SCREEN and not a SET SCREEN.

Help Screens

On-line help is one of the best ways to make an application user-friendly. Even if a manual for the application exists, it can't be assumed that the users will read it, understand it, or even have access to it. Even users who have learned the application may forget portions if it is not used periodically.

There are several techniques for providing help. Perhaps the primary technique is the use of understandable error messages. The application should detect errors and print some useful messages on the display line of the screen. The messages should be specific and should assist in correcting the problem. Such messages are much better than the generalized messages generated by RELATE and better than the use of an error number (even if the error is described in the manual for the application).

Help can also take the form of one or more screens of information that can be displayed in response to a function key or instructions given during the use of an application. If the help information is contained on several screens, users should be allowed to view the first screen for as long as desired. When a user wishes to continue, he or she can press a key. If the sequence of help screens is long, a way should be provided to terminate the help sequence prematurely.

If help is contained on a single screen, the screen should normally be CALLed. This will cause BUILDER to save all of the information on the current screen before displaying the help screen. If the help information consists of a sequence of screens, the first should be CALLed and the subsequent screens should be SET. Help can also be placed in a SCROLL command. The advantage of using the SCROLL command is that the screen being described is not erased before the help is given.

As noted above, help can be invoked in a number of ways. The most flexible method is through the use of a function key. When a function key is used, BUILDER will remember the current location of the cursor. This information can be obtained through the $VARIABLE built-in variable. This method allows help to be associated with individual fields on the screen as well as the overall screen. Because BUILDER operates in character or line mode, calling a help screen will not cause any data the user has placed on the current form to be lost.

IMPROVING PERFORMANCE

Here are some guidelines on improving the execution speed of a BUILDER application.

1. Line mode is slightly faster than character mode. Block mode is faster for initial data entry because no verification is done, but slower if any errors are found as each pressing of ENTER will cause all variables to be reverified.

2. Whenever possible, open each file in a separate partition. Do RECORD processing commands with the USING option or SET PARTITIONs instead of SET PATHs. BUILDER typically has enough memory for 10 to 15 partitions.

3. Do not use SELECT to validate key values. Use RECORD POINT followed by $FOUND. Use SELECT only to do joins and complex file operations.

4. When using a SELECT command, try it through RELATE to verify that it is not creating a temporary index.

5. Within IF-ENDIF blocks, use FAIL or FINISH with a message instead of DISPLAY if no further processing needs to be done.

6. COMPILE the application.

7. ARRAY loops are more efficient than WHILE loops.

8. Use RELATE commands whenever possible to process multiple records, rather than looping (especially the DELETE, LET, COPY, and UPDATE commands).

9. Use the RANGE and INITIAL options in the DECLARATION section instead of BUILDER code to initialize and validate values for variables.

10. Opening files takes time - open as many as possible at the beginning of the application and refrain from closing.

11. Do all SET INDEXes when the file is opened and avoid doing elsewhere unless absolutely necessary. It is faster to have two partitions for the same file with different current indexes.

12. A CALL SCREEN followed by a RETURN SCREEN is faster than two SET SCREENS.

13. Repeating identical code in several places gives faster execution than treating it as a subroutine screen, although maintenance becomes more involved.

14. "Stack" variable names in the DECLARATION section (eg: X,Y,Z; NUMERIC) and commands (eg: CLEAR VARIABLE A,B,C).

15. Use of LINEDRAW slows down the drawing of screens on some terminals.

16. Use STATEs instead of several separate screens with similar LAYOUT sections. Do not use STATEs for actions performed frequently by users or as subroutines.

SECTION 6

PROGRAMMING LANGUAGE INTERFACE

BUILDER PROGRAMMING LANGUAGE INTERFACE

BUILDER was designed to allow the development of complete applications, but some applications will require more speed or complexity than that provided by BUILDER commands. For this reason the CALL PROCEDURE command was included to allow the application developer to leave the BUILDER environment and write some of the application in a traditional programming language (SPL, FORTRAN, COBOL II, or PASCAL). A set of procedures is provided to perform communication between these subroutines and BUILDER.

To use the package, the application developer writes a subroutine conforming to the BUILDER calling conventions (described in this section) and places it in a group, account, or system SL. When a BUILDER application is run that executes a CALL PROCEDURE command to this subroutine, BUILDER looks first in the group, then account, then system SL for the subroutine. Note that BUILDER looks in the group and account in which the user is logged on, not the group and account in which the BUILDER program resides.

WARNING: It is possible for a user written routine invoked from BUILDER to invalidate the BUILDER environment resulting in program aborts or erroneous processing. CRI will attempt to resolve problems resulting from the use of these routines. However, this service is not provided under the standard software maintenance contract and must be purchased on a time and materials billing basis.

SUBROUTINE CALLING CONVENTIONS

There are two conventions that may be used for writing subroutines. The first convention is more straightforward, but requires substantial overhead each time a new procedure is called. The second method requires that an extra interface routine be written by the application developer, but requires less overhead and may be used to permanently install the subroutines into the BUILDER interpreter.

Method 1

Each subroutine must contain exactly three parameters, each of which is an integer array. The first parameter is the current partition in BUILDER at the time of the CALL PROCEDURE command. The second parameter is a table defining the current state of BUILDER. This may be used in future releases for some purposes. The last parameter may be used to pass information from the CALL PROCEDURE statement. See the CALL PROCEDURE documentation for further information.

Example:

```
      SUBROUTINE TEST(CUR,SCA,PAR)
      INTEGER CUR(1),SCA(1),PAR(1)
C . . .      perform computation
      RETURN
      END
```

The subroutine should be compiled and placed into a group, account, or system SL relative to where the end user is using the application. Once the subroutine is resident in the SL it may be called directly by BUILDER's CALL PROCEDURE command. The subroutine may call the communication routines to communicate with the BUILDER program.

Method 2

The application developer writes a special routine called CMDUSER01, which in turn calls all the other subroutines written by the application developer. When BUILDER encounters a CALL PROCEDURE command it searches for the CMDUSER01 routine, which will then pass control along to the appropriate subroutine. In this method there are no calling conventions required for the individual subroutines. In order to call a subroutine, the subroutine must not only exist in the SL but must also be called by the CMDUSER01 routine.

The calling convention for CMDUSER01 is (SPL example):

```
PROCEDURE CMDUSER01(NAME,NUMBER,CUR,SCA,PAR);
BYTE ARRAY NAME;
INTEGER NUMBER;
INTEGER ARRAY CUR,SCA,PAR;
```

The CUR, SCA, and PAR arrays are exactly as described in method 1, and these may be passed down to a subroutine as required.

INSTALLING ROUTINES IN AN SL

The communication package is supplied with BUILDER as a USL file called PLEUSL. This USL file can be installed in the group, account, or system SL so that it can be called by subroutines written by the application developer.

The following MPE command can be used to install the communication package in a group or account SL.

```
:SEGMENTER
 -SL SL
 -USL PLEUSL
 -PURGESL SEGMENT,PLEUSL
 -ADDSL PLEUSL
 -EXIT
```

The following MPE command can be used to install a segment of user subroutines into an SL. Assume that the USL file containing the subroutines is called USERUSL and the segment name is USERSEG. If method 2 is used as the calling convention, CMDUSER01 should also be contained in this segment.

```
:SEGMENTER
 -SL SL
 -USL USERUSL
 -PURGESL SEGMENT,USERSEG
 -ADDSL USERSEG
 -EXIT
```

COMMUNICATIONS PACKAGE CALL SUMMARY

The communications package provides an interface between the data structures maintained by BUILDER and a subroutine written in a traditional language. The routines allow data to be retrieved and updated and partitions to be accessed and adjusted.

CALL NAME	DESCRIPTION
BLDRGETVAR	Move data from a BUILDER variable to a subroutine.
BLDRPUTVAR	Move data to a BUILDER variable from a subroutine.
BLDRGETCUR	Get the RELATE cursor given a BUILDER partition name.
BLDRPUTCUR	Inform BUILDER that the current path has changed in the partition.

BLDRPUTVAR (name, data, datalen, error)

Moves data from a variable in the host language procedure into a variable in the BUILDER application.

name	Required. A character array terminated by a blank, null, or backslash of a maximum of 15 characters. This contains the name of a BUILDER variable.
data	A character array of length DATALEN containing the alphabetic value to be placed into the specified BUILDER variable.
datalen	Required. An integer containing the length of the DATA variable. If DATALEN is longer than the BUILDER variable the data will be truncated and ERROR=-1 returned. If DATALEN is shorter than the BUILDER variable then the BUILDER variable will be padded with blanks to the right.
error	An integer containing the error number returned by this call. Zero means no error. A one means the variable is not known by BUILDER. A minus one means data has been truncated.

All data, including numbers, are treated as alphanumeric data and should be placed into the variable as ASCII characters. If the BUILDER variable is an array variable then the value of the current subscript will be changed.

BLDRPUTCUR (partitionname, partition, error)

Returns an updated partition, fetched by BLDRGETCUR, to the BUILDER program.

partitionname	Required. A character array terminated by a blank, null, or backslash of a maximum of 15 characters. This should contain the name of an existing BUILDER partition.
partition	An integer array of 50 words containing an HLI partition.
error	An integer containing the error number returned by this call. Zero means no error. A one means the partition name is not known by BUILDER.

This routine is required if the procedure uses the RELATE HLI calls to change the current path. This routine is not required if only RDBREAD, RDBPOINT, RDBDELETE, RDBUPDATE, or other calls that do not change the current path are executed.

FORTRAN PLE INTERFACE EXAMPLE

```
$CONTROL USLINIT, SEGMENT=SEG2
      SUBROUTINE PLETEST(MAINCUR,SCA,PAR)
      INTEGER MAINCUR(1),SCA(1),PAR(1)
      CHARACTER*15 NAME
      CHARACTER*80 DATA
      INTEGER DATALEN,ERR
      INTEGER CUR(50)

      CALL BLDRGETVAR("FIRST ",DATA,80,ERR)
      CALL BLDRPUTVAR("SECOND ",DATA,80,ERR)
      CALL BLDRGETCUR("MAIN ",CUR,ERR)
      CALL RDBCLOSE(CUR)
      CALL BLDRPUTCUR("MAIN ",CUR,ERR)
      RETURN
      END
```

PASCAL PLE INTERFACE EXAMPLE

```
TYPE
    small_int                  = -32768..32767;
    intarray                   = ARRAY [1..50] of small_int;
    string15                   = PACKED ARRAY [1..15] of CHAR;
    string80                   = PACKED ARRAY [1..80] of CHAR;

PROCEDURE pletest (VAR maincur, sca, par : intarray);

VAR
    name                   : string15;
    data                   : string80;
    datalen,err            : small_int;
    cur                    : intarray;
    datalen                : small_int;

{BUILDER externals}
PROCEDURE bldrgetvar (VAR bldrvar        : string15;
                      VAR data           : string80;
                      VAR datalen        : small_int;
                      VAR err            : small_int); external SPL;

PROCEDURE bldrputvar (VAR bldrvar        : string15;
                      VAR data           : string80;
                      VAR datalen        : small_int;
                      VAR err            : small_int); external SPL;

PROCEDURE bldrputcur (VAR par            : string15;
                      VAR cur            : intarray;
                      VAR err            : small_int); external SPL;

PROCEDURE bldrgetcur (VAR par            : string15;
                      VAR cur            : intarray;
                      VAR err            : small_int); external SPL;

BEGIN
         datalen:=80;
         bldrgetvar ('first ', data, datalen, err);
         bldrputvar ('second ', data, datalen, err);
         bldrgetcur ('main ', cur, err);
         rdbclose (cur);
         bldrputcur ('main ', cur, err);
END;
```

A P P E N D I C E S

APPENDIX A

COMMAND and SECTION FORMATS

PART I - COMMANDS

ARRAY ... ENDARRAY

Performs automatic looping through array variables.

variablename := expression

Performs arithmetic or character manipulations and assigns the value of the expression to the indicated variable.

BREAK

Causes the application to suspend and invokes the BUILDER command interpreter.

CALL SCREEN screenname

Displays a new screen and saves the current screen for a subsequent return.

CALL PROCEDURE slprocname

Executes a procedure in an SL written in SPL, COBOL, FORTRAN, or PASCAL.

CLEAR [ALL|RECORD [USING partitionname]|SCREEN|VARIABLE variablenamelist [;SUBSCRIPT]]

Clears sets of variables.

CREATE PARTITION partitionname [;SEGMENT=number]

Creates the indicated partition.

DISPLAY [expression]

Places text into the message line.

ERASE SCREEN

Clears the entire screen display.

PURGE PARTITION partitionname

Purges the indicated partition.

RECORD ADD [USING partitionname]
[;VARIABLE=varname|(varname,fldname)[,varname|(varname,fldname)][...]]

Adds a record to the file(s) referenced by the current path in the current or specified partition.

RECORD DELETE [USING partitionname]

Deletes a record from the file(s) referenced by the current path in the current or specified partition.

RECORD NEXT [USING partitionname]

Reads a record from the file(s) referenced by the current path without moving data from the database fields into BUILDER variables.

RECORD POINT [USING partitionname] [KEY=variablelist] [;BREAK=num]

Locates a record in the file(s) referenced by the current path in the current or specified partition.

RECORD READ [USING partitionname]
[;VARIABLE=varname|(varname,fldname)[,varname|(varname,fldname)][...]]

Reads a record from the file(s) referenced by the current path in the current or specified partition.

RECORD REPOINT [USING partitionname] [KEY=variablelist] [;BREAK=num]

Re-locates a record in the file(s) referenced by the current path in the current or specified partition in preparation for a delete or update.

RECORD RESET [USING partitionname]

Resets the end-of-file on the current or specified partition. A subsequent read using the partition will retrieve the next record.

RECORD REWIND [USING partitionname] [;BREAK=fields]

Rewinds the file(s) referenced by the current path in the current or specified partition.

SET SCREEN screenname

Causes a lateral transfer of control to a new screen.

SET STATE statename

Changes current state of screen. This will cause the DECLARATION section associated with this state to be evaluated, the INITIAL section to be executed, and the function key labels to be redrawn to reflect the new state.

SHOW PARTITIONS

Displays the partitions which currently exist.

SHOW VARIABLES [OPTIONS] [variablenamelist]

Displays variables known by BUILDER and their options.

STREAM [logon] ... ENDSTREAM

Causes the creation of a file (BLDRJOB) which is streamed.

SUBMIT INTO filename
SUBMIT REPORT[:P]
SUBMIT JOB

Causes the creation of a file (BLDRSBMT) which is either closed, streamed as a job, or used by CREATE to produce a report.

TRACE ON|OFF

Enables or disables the display of commands as they are executed by BUILDER.

VERIFY

Verifies that all changed variables contain valid data as described in the DECLARATION of the variable and as determined by the variable's VARIABLE section (if any).

WHILE condition ... ENDWHILE

Causes looping within a command section.

delimiters INITIAL [STATE=statename]

Contains commands that are executed before the screen is displayed.

delimiters INITIAL EVERYTIME

Contains commands that are executed after the INITIAL section and before the screen is redisplayed after a FUNCTION, ENTER, or ACTION section has been executed.

delimiters LAYOUT [LINDRAW=linechar]

Contains the format of the screen.

delimiters SCREEN screenname

Defines the beginning of a new screen.

delimiters TIMEOUT

Contains commands that are executed when the number of seconds indicated by SET OPTION TIMEOUT have elapsed with no user input.

delimiters VARIABLE [variablename]

Contains commands that are executed when the value of the indicated variable is changed by the user.

APPENDIX B

CREATING AND MODIFYING TERMINAL DRIVERS

The screen system in BUILDER is table-driven. The tables are stored in the BLDRTERM file in the directory where BUILDER is installed. If a user has write access to this file, the file can be updated to adjust existing terminal driver characteristics and to add new terminal types. When BUILDER is invoked, it will normally prompt:

```
    Terminal Type?
```

When "NEW" is entered to this request and the user has write access to the BLDRTERM file, the configuration section of BUILDER is entered. The configuration is accomplished through a question and answer sequence. In most menus, BUILDER will prompt for:

```
            CHOICE?
```

Enter the number of the action to perform. Typing a "?" will regenerate the list of valid responses.

In the configuration, items appearing in brackets represent the existing and default values. To retain the indicated value, the RETURN key may be pressed. To remove the indicated value, enter a space and then press RETURN. If a desired sequence consists of several characters, the characters should be separated by spaces.

To include a printing character simply type the character. The keyword ESC may be entered to indicate an escape; a caret ("^") may be used to indicate a control character. For example, a control-H, which normally corresponds to a backspace, would be entered as ^H. Non-printing characters can be entered by placing a pound sign ("#") in front of the decimal value of the character. For example, a DEL could be entered as #177. When the sequence is subsequently displayed, printing characters will be displayed as printing characters and control characters will be displayed as control characters, regardless of how they were entered.

8) Delete type.

The configurator will respond with:

Delete terminal type:

Enter the name of the terminal type to be deleted and press RETURN. The terminal type will be deleted.

Main Menu

```
 1) HELP
 2) Save configuration.
 3) Discard configuration changes.
 4) CHARACTER, LINE, or BLOCK mode driver. []
 5) Cursor positioning sequences.
 6) Device sequences.
 7) Input key definition.
 8) Enhancement sequences.
 9) Initialize and Reset sequences.
10) Function Key specification.
11) Character mode typeahead limit. []
12) Function key label specification.
13) Linedrawing sequences.
14) Block mode driver sequences.
```

The MAIN MENU starts the configuration process and determines if the additions or changes are made a permanent part of the configuration file. The default operating mode, either character or line, can be adjusted at this point.

2) Save configuration.

Any changes made to the terminal configuration are saved in the BLDRTERM file. The system will then return to the TERMINAL CONFIGURATION MENU.

3) Discard configuration changes.

Changes made to the configuration are ignored. The system will return to the TERMINAL CONFIGURATION MENU.

4) CHARACTER , LINE, or BLOCK mode terminal driver. []

Enter either "C" (for character), "L" (for line), or "B" (for block).

5) Cursor positioning sequences.

The CURSOR POSITIONING SEQUENCES menu is displayed.

6) Device sequences.

The DEVICE FUNCTIONS menu is displayed.

7) Input key definition.

The INPUT KEY DEFINITIONS menu is displayed.

8) Enhancement sequences.

The ENHANCEMENT SEQUENCES menu is displayed.

9) Initialize and Reset sequences.

The INITIALIZATION menu is displayed.

6) Sequence to separate row/column numbers. []
Enter the sequence, if any, that is used to separate row numbers from column numbers in a cursor positioning sequence.

7) Sequence to finish cursor sequence. []
Enter the sequence, if any, that is used to indicate the end of a cursor positioning sequence.

8) ROW or COLUMN specified first. []
Enter ROW if the row number should be given first when BUILDER issues a cursor positioning sequence. Enter COLUMN if the column number should be given first.

9) Column number representation. []
Enter 1, 2, or 3 to indicate how the column numbers should be represented in the cursor positioning sequence. If 1 is entered, the sequence is displayed as a single character, modified by the response to questions 11 and 13. If 2 or 3 is entered, the sequence is displayed in ASCII in 2 or 3 digits.

10) Row number representation. []
Enter 1, 2, or 3 to indicate how the row numbers should be represented in the cursor positioning sequence. If 1 is entered, the sequence is displayed as a single character, modified by the response to questions 12 and 14. If 2 or 3 is entered, the sequence is displayed in ASCII in 2 or 3 digits.

11) Left column number. []
Enter the value of the leftmost column number on the terminal. Typical values for this position are 0, 1, and 32.

12) Top row number. []
Enter the value of the uppermost row number on the terminal. Typical values for this parameter are 0, 1, and 32.

13) Increment right. []
Enter the value to be added to the value supplied in question 11 to move the cursor to the right by one character position. This value is typically 1, but may occasionally be -1.

14) Increment down. []
Enter the value to be added to the value supplied in question 12 to move the cursor down one line. This value is typically 1, but may occasionally be -1.

Device Functions

```
1) HELP
2) Return to the main menu.
3) Clear screen.                []
4) Clear line.                  []
5) Clear partial screen.        []
6) Home cursor (top LEFT).      []
7) Home DOWN cursor.            []
8) Move cursor LEFT.            []
9) Move cursor RIGHT.           []
```

14) Insert character mode ON. []

Enter the sequence that will turn on the mode that causes all characters subsequently typed to be inserted in the line rather than replacing characters. Characters are not shifted from one line to the next, but are lost as they are shifted off the right side of the current line.

15) Insert character mode OFF. []

Enter the sequence that will turn off insert mode.

Input Key Definitions

```
 1) HELP
 2) Return to the main menu.
 3) Meta-key.          []
 4) Home cursor.       []
 5) Backspace.         []
 6) Cursor UP.         []
 7) Cursor DOWN.       []
 8) Cursor LEFT.       []
 9) Cursor RIGHT.      []
10) TAB.               []
11) Backtab.           []
12) Re-draw screen.    []
```

Options 4 through 11 define keys which cause BUILDER to reposition the cursor location. Each of these items requests the following information:

CHAR(s) output by ‹keyname› key. []

Enter the actual character emitted by the terminal when the appropriate key is pressed. For example, the backspace key will normally emit a control-H.

Does this key echo its function. []

Enter YES if the key causes the terminal to respond in the same manner as the description of the key. For example, if the backspace key causes the cursor to be backed up one character by the terminal enter YES. Enter NO if the key does not echo its function.

Does this key destructively write characters onto the screen. []

Enter YES if the key erases a character on the screen when it is struck. Enter NO if the key does not change the content of the screen.

Is this a special key on the keyboard. []

Enter YES if the label on the key is not the same as the function that the key performs. For example, if the terminal has a key containing a left arrow which can be used to move the cursor one character to the left enter YES.

3) Meta-key. []

Enter the key that will be used to start special function key sequences. This should normally be the ESC key.

4) Home cursor. []

Enter the key that, if entered, should cause the menu to home the cursor to the topmost variable on the screen.

Enhancement method. []

Enter either CHAR, HP, MODE, or NONE. If CHAR is entered, the display enhancement sequence is assumed to occupy a character on the display. If HP is entered, the display enhancement sequence does not occupy a character on the display. If MODE is entered, the terminal only recognizes a single enhancement type that does not occupy a character on the display. NONE indicates that there is no enhancement sequence for this enhancement.

Begin enhancement. []

Enter the sequence that should be used to begin the particular enhancement desired.

End enhancement. []

Enter the sequence that should be used to end the enhancement.

6) Define default variable enhancements.

The DEFINE VARIABLE ENHANCEMENTS menu is displayed.

7) Define default message line enhancements.

The DEFAULT MESSAGE LINE ENHANCEMENTS menu is displayed.

Single Enhancements

```
 1) HELP.
 2) Return to the main terminal enhancement menu.
 3) INVERSE method (1).     []
 4) Begin INVERSE.          []
 5) End INVERSE.            []
 6) UNDERLINE method (2).   []
 7) Begin UNDERLINE.        []
 8) End UNDERLINE.          []
 9) BLINKING method (3).    []
10) Begin BLINKING.         []
11) End BLINKING.           []
12) HALFBRIGHT method (4).  []
13) Begin HALFBRIGHT.       []
14) End HALFBRIGHT.         []
```

See the TERMINAL ENHANCEMENT SEQUENCES menu for explanations of these items.

Default Variable Enhancements

```
1) HELP.
2) Return to the main terminal enhancement menu.
3) OPTIONAL variables.         []
4) REQUIRED variables.         []
5) DISPLAY only variables.     []
6) POSITION only variables.    []
7) Variables in error.         []
8) ACTION variables.           []
```

Options 3 through 8 allow assignment of default display enhancements for variables. The default values should correspond to the number in parentheses of the terminal enhancement mode desired, as shown in the three ENHANCEMENTS menus. For example, if the default REQUIRED variable enhancement should be U, B (underline and blink) a 6 should be entered.

Default Message Line Enhancements

```
1) HELP.
2) Return to the main terminal enhancement menu.
3) Automatic BUILDER messages. []
4) DISPLAY command messages.   []
5) FINISH command messages.    []
6) FAIL command messages.      []
7) PROMPT text.                []
```

Options 3 through 7 allow assignment of default display enhancements for various types of messages emitted by BUILDER. The default values should correspond to the parenthetical number of the terminal enhancement mode desired, as shown in the three ENHANCEMENTS menus. For example, if FAILure messages should be enhanced with U, B (underline and blink) a 6 should be entered.

Initialization

```
 1) HELP.
 2) Return to the main menu.
 3) Initial sequence 1.            []
 4) Initial sequence 2.            []
 5) Initial sequence 3.            []
 6) Initial sequence 4.            []
 7) Initial sequence 5.            []
 8) Initial sequence 6.            []
 9) Initial sequence 7.            []
10) Initial sequence 8.            []
11) Initial sequence 9.            []
12) Initial sequence 10.           []
13) Escape echo sequence.          []
14) Reset sequence.                []
```

3) Initial Sequence 1. []

Enter the sequence that should be used to initialize the terminal. This might include the escape or control sequences to do such things as clear the screen

Option 11 will display a menu similar to this menu and allow the definition of function keys 9 through 16.

When a function key definition is performed, the system will request:

```
CHAR(s) output by F# key []:
Does key destructively write characters onto screen []:
Is this a special key on keyboard []:
Equivalent to what escape function []:
```

CHAR(s) output by F# key []:

The characters actually output by the function key should be entered, not including any carriage return or sequence terminators.

If the function key is programmable, for best results it should be programmed to issue a single character (see INITIALIZATION menu), and this character should appear here.

If the full function key sequence (see INITIALIZATION) terminates with a carriage return, then the driver can be set to run in LINE mode; however, other special keys may not work since they don't terminate with RETURN. Generally, CHARACTER mode is used, and the character mode typeahead limit should be set to the number of characters, not including the carriage return, that the fkey outputs (see MAIN MENU to set these values).

Does key destructively write characters onto screen []:

If the use of the function key disturbs the content of the screen, answer yes.

Is this a special key on keyboard []:

If there is a key on the keyboard labeled as the appropriate function key, answer YES.

Equivalent to what escape function []:

When the function key is struck, BUILDER will actually respond with this character. This is the character that can be used along with the ESC key in lieu of an existing function key (for example: an "ESC 1" can be typed by the user instead of pressing function key 1; or an "ESC :" could be set up here to execute function 10). This is also the character that the FUNCTION section in the BUILDER application file should be labeled with (eg: "FUNCTION 1" for function 1; "FUNCTION :" for function 10).

Function Key Label Specification

```
1) HELP.
2) Return to the main menu.
3) Width of each function key display window. []
4) Number of lines in display window. []
5) Function key labels are displayed on message line? []
6) Specify lead-in sequence for setting F1-F8 labels.
7) Specify lead-in sequence for setting F9-F16 labels.
8) Specify trailing sequence for F1-F8.
9) Specify trailing sequence for F9-F16.
```

5) Frequency of Init/End Sequence.

This determines how long the terminal remains in linedraw mode:

CHAR:resend the start sequence before each linedraw character.

LINE: will remain in linedraw mode for the rest of the line after the start sequence.

SCREEN: will remain in linedraw mode until the end sequence is sent.

Block Mode Sequences

```
1) HELP.
2) Return to the main menu.
3) Block Mode type.                []
4) Block Mode ON.                  []
5) Block Mode OFF.                 []
6) Page Mode.                      []
7) Field Mode.                     []
8) Keyboard Lock.                  []
9) Keyboard Unlock.                []
10) Format Mode ON.                []
11) Format Mode OFF.               []
12) Start Protected.               []
13) Start Unprotected.             []
14) Field Separator.               []
15) Block Separator.               []
16) Transmit Block.                []
17) Transmit Position.             []
18) Transmit Status                []
19) Useful Status Bits.            []
20) Valid Block Status.            []
21) Block Mode Timeout.            []
22) Block Mode Retry Count.        []
```

APPENDIX C

VPLUS/3000 COMPARISON

VPLUS/3000 is a screen management utility provided by Hewlett-Packard. BUILDER and VPLUS can generally implement the same systems, but several major conceptual differences exist. For those familiar with VPLUS, this section will describe these differences.

Orientation

VPLUS is a collection of subroutine calls available to programmers. These calls simplify a screen level interface between an application and an end-user. VPLUS calls are embedded in an application program written in a traditional language. This program essentially controls the interface by determining which screens to display, as well as processing the contents of the screens. The application program is responsible for interfacing with any data files required, as input or output, by the application.

BUILDER is an interpreter designed specifically for applications, just as BASIC interpreters are used for highly interactive program development. BUILDER has a built-in screen system, and, unlike VPLUS, this screen system is not called explicitly by an application. Instead, those commands executed to perform certain operations cause implicit screen actions. BUILDER has the capacity to call external procedures written in traditional languages to perform complex or CPU intensive operations. BUILDER can also execute complete programs. Normally, applications can be developed entirely with BUILDER facilities—without using these more traditional mechanisms.

It should be apparent, then, that VPLUS and BUILDER occupy very different positions in regard to applications. VPLUS can be thought of as being "under" an application program providing support functions. BUILDER is "above" the application, which is written in a higher level language, and when necessary can call user written routines to provide support functions. VPLUS programs must explicitly control the user interface; BUILDER applications, for the most part, implicitly control the user interface.

User Interface

VPLUS is designed to work on HP and HP-compatible terminals in block mode. A major benefit of block mode is that users can fill and edit an entire screen worth of information without computer interaction. The value of this capability increases as the distance between the user and the computer is increased, particularly if the interface is through DS lines or a packet switching network (such as X.25). A major problem with block mode is that it provides a rather poor user interface because of this lack of communication. In block mode, the computer generally responds only to a function key or the ENTER key. Errors in data and feedback from the computer cannot be obtained until these keys are pressed.

BUILDER operates in either character or line mode. Either mode allows a high degree of user interaction during the data entry process. BUILDER sections are available to allow data validation as each variable is entered. This validation can consist of simple format checking or can be as complex as validating the existence of keys. It is possible to ignore these capabilities and present an environment similar to VPLUS by simply deferring any validation until the screen is completed.

VPLUS returns information in character format, which must then be translated into the appropriate format for use in the application. These translations are usually quite simple and in the hybrid situation can either be done before the subroutine is called, which will then allow passing of character strings or long values, or in the subroutine, which would then perform the same conversions that would have been required with VPLUS.

APPENDIX D

SCRIPT FILES

BUILDER provides mechanisms that will allow the construction of application test "scripts," off-line testing using these scripts, and demonstrations. To create a script, either an editor can be used or BUILDER can be placed in a mode in which data and commands entered by the user are echoed to a file. This script can subsequently be used as a substitute to on-line entry of the data.

Script File Format

The script file should be an unnumbered ASCII file. The file should contain the data and commands that would normally be required to execute the application interactively, including, if run on-line, responses to the Terminal type? and Application File? prompts. If run in batch, responses to these prompts should be included as data for the BUILDER program rather than as part of the script file. Text should be entered as is, and commands must begin with a backslash character ("\").

The following commands are available:

\E	ESCAPE (for function keys)
\Q	QUIT
\R	RETURN
\T	TAB
\^X	control-X (or any other character)
\\	A backslash
\#num#	The ASCII character corresponding to num.
\%	The beginning of a comment. The comment extends to the end of the line.
\P	Pause for one second.
\W	Wait for key to be pressed to continue.

A typical sequence would look like this:

```
1\r3\t\t11-3-83\r\e1
```

This sequence corresponds to entering the digit 1, a RETURN, the digit 3, TAB, TAB, the date 11-3-83, RETURN, and ESCAPE 1 (function key 1). The sequence can be created through an editor or by BUILDER if the echo capability is enabled.

Creating a Script File with ECHOing

The echo capability can be enabled in two ways. To enable the echo capability while BUILDER is running, an ESCAPE period (ESC.) function should be entered. BUILDER will respond with a request for a file name into which the commands should be placed. BUILDER will create a new ASCII file with 72 byte records and a default length of 1023 records. All keystrokes, including data and special keys, entered by the user will be placed into the script file in the format described above. The only exception is that function keys, rather than being output in a generic fashion (\E1 for function key 1), are output with exactly the character(s) that the function key on that terminal generate (for example, \^A for function 1 on a WY50).

If a new screen is entered, the screen name will be placed in a comment line.

Examples:

Activity Occurring	JOB	OFFLINE	WY50
terminal initialization	no output	-INITIAL-	\e':\e+ etc*
enter a screen	*** SCREEN name	*** SCREEN name	*** SCREEN name
user presses F1	\e1\r	\e @(00,23)@(00,23) -CL- which function key? 1	\e \e\e-7 \e-7 \eT which function key? 1
layout is displayed	no output	-CS- @(03,00) MY SCREEN TITLE @(00,23) etc not including fkey labels	\e+\e- #MY SCREEN TITLE etc including fkey labels
Builder terminates	no output	-RESET-	\ef\r\eA10\e-7**

* Initialization sequence from terminal configurator.
** Terminal reset sequence from terminal configurator.

Testing Applications

The use of scripts and logs can be used to validate a new release of an existing application. To do this, a script is created for the current release of the application. BUILDER is then run with this script as input and a disc or tape file as output (which is equated to BLDRLOG). To validate the new application, the original script should be used on it. The output should be saved and then compared with the original output. Discrepancies should be checked to determine if the changes were by design or are a problem in the application.

APPENDIX E

SPECIAL FUNCTION SEQUENCES FOR HP TERMINALS

```
*** SCREEEN TEST
*** LAYOUT
Special function sequences for HP Terminals.


                    1.   Home cursor.
                    2.   Turn off function key labels.
                    3.   Turn on function key labels.
                    4.   Copy graphics plane.
                    5.   Copy alphabetic plane (current page).
                    6.   Turn on numeric pad designation.
                    7.   Turn on graphics pad designation.
                    8.   Skip lines on built-in printer.
                    9.   Turn on the graphics plane.
                    0.   Turn on the alphabetic plane.
                    E.   Exit


   ACTION [A]
*** VARIABLES
   A: ACTION
   ESC; LENGTH=1; INITIAL=""
   CODE; LENGTH=1
*** INITIAL EVERYTIME
   A:=" "
*** ACTION 1
   NOTE  This action homes the cursor.
   DISPLAY "%esc&a0c0Y"
   PAUSE 5
*** ACTION 2
   NOTE  This action turns off the function key labels.
   DISPLAY "%esc&j@"
*** ACTION 3
   NOTE  This action turns on the function key labels.
   DISPLAY "%esc&jB"
*** ACTION 4
   NOTE  Copy the graphics plane to the printer.
   DISPLAY "%esc&p7s6dF"
   NOTE  Must prompt for completion code.
   PROMPT "Completion code?", CODE
*** ACTION 5
   NOTE  Home the cursor and copy the current page to the printer.
   DISPLAY "%esc&a0c0Y[%esc&p3s6dF]"
   NOTE  Must prompt for completion code.
   PROMPT "Completion code?", CODE
   NOTE  Roll the screen down one line.
   DISPLAY "[%esc]T"
*** ACTION 6
   NOTE  Change to numeric pad.
   DISPLAY "%esc&k0O"
```

INDEX

[illegible]

[illegible]

READER COMMENT SHEET
BUILDER Reference Manual
HPB-RF02
8803

We welcome your evaluation of this manual and its related software product. Your comments and suggestions assist us in improving our publications and software. Please use additional pages if necessary.

1. Does this manual clearly and accurately describe all the features of its associated software?
2. Are the concepts and words in this manual easy to understand?
3. Is the format of this manual convenient in arrangement and readability?
4. Are the index and table of contents complete and useful?
5. Are the examples clear, correct, and informative?

COMMENTS:

From (Optional):	NAME:	______________________
	PHONE:	______________________
	COMPANY:	______________________
	ADDRESS:	______________________
		______________________
		______________________