RPL 1.3
REFERENCE
MANUAL

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The MICRODATA REALITY computer has been supplied with a powerful interpretive language called REAL-TIME PROCESSING LANGUAGE (RPL). Programs coded in this language are called PROC's (for stored procedures) and may be executed individually, chained, or called as subroutines by other PROCs. Functions have been provided to allow execution of ENGLISH statements within a PROC just as they would be executed from TCL.

This manual has been written for the experienced programmer and no attempt has been made to provide detailed definition of computer functions except as required to present the workings of RPL. In some cases the explanations used differ from the actual machine level operations. These deviations are used to simplify the explanations and all functions may be assumed to operate as described. Every attempt has been made to provide precise and unambiguous explanations and examples.

Each PROC being executed from a terminal is allotted a complete set of resources, unique to itself, as described herein. Data areas in use by one terminal are unique and will not be interfered with by operations performed at any other terminal. The only shared resources which will affect PROC operations are disk files which may be accessed by many processes at one time.

CHAPTER 2

BUFFERS AND WORK AREAS

2.1 PRIMARY AND SECONDARY INPUT BUFFERS

There are two input buffers used for obtaining input from external devices, the primary and secondary input buffers. Access may be made to only one input buffer at a time. This buffer is designated the current input buffer and is determined by the current condition of the input buffer stack pointer. When off, the input buffer stack pointer will cause all general input buffer references to use the primary input buffer; and, when on, the same commands will access the secondary input buffer. The input buffer stack pointer may not be manipulated directly, but will be set on and off automatically as part of the execution of certain commands.

Because the operation and access of the two buffers is substantially different, they are explained separately.

2.1.1 PRIMARY INPUT BUFFER

The primary input buffer is used for input of data from all external devices, although many of its characteristics make it desirable as working storage if space is available. Attributes within the primary input buffer are separated by attribute marks and are referenced by use of a percent sign (%) and a numeric value designating which attribute is to be used (%9 is the reference for attribute 9 of the primary input buffer). Attributes within the primary input buffer are numbered consecutively starting with 1.

Attributes in the primary input buffer may be any size with a minimum of no characters (a null attribute) and a theoretically unlimited maximum. The primary input buffer may contain virtually any number of attributes of any size and access to any field in the buffer is random.

Associated with the primary input buffer is a buffer pointer which may indicate any attribute or character within an attribute. This pointer is used to conditionally direct the operation of various commands (as will be discussed later).

Additionally, a further characteristic of the primary input buffer is that it retains all keyboard input entered in TCL at the time a PROC was initiated. This means that additional parameters may be keyed in which will supply data to the PROC at program initialization time. All characters keyed in will be loaded into the primary input buffer starting at attribute

1. All spaces will be converted to attribute marks, and the end of the buffer will be established when new-line is keyed or at a maximum input of 140 characters.

2.1.2 SECONDARY INPUT BUFFER

The secondary input buffer is limited in scope and function and is affected by few commands. As with the primary input buffer, attributes within the secondary input buffer are separated by attribute marks and are numbered consecutively starting with 1. There is no limit on the size of fields in the secondary input buffer or on the total size of the buffer. Direct access of attributes within the secondary input buffer is through the use of the A command. The secondary input buffer has a buffer pointer which functions the same way the primary input buffer pointer does.

When any command that turns the input buffer stack pointer off is executed, the secondary input buffer is effectively cleared. No further access may be made of those items remaining in it.

2.2 PRIMARY AND SECONDARY OUTPUT BUFFERS

The output buffers are used for executing ENGLISH statements within a PROC, they also may be used for storage of working parameters on a temporary or permanent basis. Attributes within the output buffers are separated by attribute marks and are referenced by use of the pound or number sign (#) and a numeric value for the attribute (#5 is the reference to the fifth attribute of the current output buffer). Attribute numbers within the output buffers start with item 1 and continue consecutively.

There are two output buffers, the primary and secondary output buffers, only one of which may be referenced at a time. The one currently available is designated by the output buffer stack pointer which may be set on or off. When off, the stack pointer will cause all commands using the output buffer to reference only items in the primary output buffer, and when on all references will access the secondary output buffer. A field within either output buffer may have any quantity of characters within the limits of size of the buffer with a minimum of none (a null attribute).

The number of characters which may be loaded into the primary output buffer is equal to the LOGON work space allotted for the account. The maximum number of characters which may be loaded into the secondary output buffer is constant at 2K.

Associated with each output buffer is a buffer pointer

which is used to indicate the end of the buffer. This pointer is used in the execution of certain commands which will be explained later.

2.3 FILE BUFFERS

File buffers are normally used for reading and writing records from files but may be used for working storage if available. Attributes within a file buffer are separated by attribute marks which are transparent to the programmer but are constructed automatically by the processor as needed. Reference to an attribute beyond those already established will result in construction of enough null attributes to reach the item referenced.

There are nine file buffers, numbered 1 through 9. Direct references to items within a file buffer are by use of an ampersand (&), the file buffer number, a period as a separator, and a numeric value representing the attribute within the buffer which is being referenced (&3.2 references attribute 2 of file buffer 3). Attribute numbers within a file buffer start with Ø, which is used as the key to read and write a record from a file, and continue with 1, 2, 3, etc to a virtually unlimited size.

An attribute in a file buffer may have any number of characters with a minimum of none (a null attribute). A file buffer may have any number of attributes and may extend to any length though they will be truncated to a maximum length of 32,267 bytes when written to a file.

2.4 SELECT REGISTERS

Select registers are used for holding the results of an ENGLISH 'SELECT' or 'SSELECT' statement within a program. They may also be used as a vehicle to split multivalued attributes into their component parts. A select register may contain any quantity of values at a time with no limit on the size.

There are five select registers available, numbered 1 through 5. Values within a multivalued attribute in a select register are available only sequentially, one value at a time. Direct reference to the next value in a select register is by use of an exclamation point and a numeric value representing the register being referenced (!3 references the next value in select register 3). Use of a direct reference bumps the select register pointer to the next attribute mark in the multivalued chain and makes the next item available. This is true reguardless of the use of the direct reference and if a value is to be used more than once, it should be moved to a hold area and referenced from there.

Additional values added to a select register by any means will delete all previously existing values and replace them entirely with the new list. No portion of the previous list which had been unused will remain.

CHAPTER 3

EXPLANATION STANDARDS

3.1 SYMBOLS USED IN ILLUSTRATIONS

The symbols explained below are used when illustrating the contents of a buffer since these are the same symbols which will appear on the terminal when the buffer is displayed. These symbols should not be confused with the normal characters used to represent them. To avoid confusion the normal characters will not be used in any buffer illustrations except as specifically noted within the accompanying text (see exception in sub-value mark explanation). Anyplace these symbols appear where the illustration is not of a buffer, these definitions do not hold unless specifically defined.

[Beginning of buffer, used to designate the start point of a buffer. This symbol is not part of the buffer itself and does not contribute to the length of the buffer.

^ Attribute mark (N[cs]), used to separate different attributes within a buffer. This symbol actually occupies a byte within the buffer and indicates the end of a field. It is not part of the field but is used only as a separator.

] Value mark (M[cs]), used to separate different values within a vultivalued field. This symbol occupies a byte within the attribute and indicates the end of a value within a multivalued field. It is part of an attribute and will be moved with the field when the entire field is moved.

\ Sub-value mark (L[cs]), used to separate different subvalues within one value of a multivalued field. Functionally, the subvalue mark operates the same way the value mark explained above does. (Note: because the backslash must be used for many explanations, this symbol will be specifically defined in the text when used. All other occurances may be assumed to be a normal backslash L[s])

Below is a series of examples which will serve to illustrate the use of these symbols.

```
File buffer 1
  [M3182^SMI, INC^203]257]385]511^DESPLAINES, ILL^
where  &1.0 = M3182
       &1.1 = SMI, INC
       &1.2 = multivalued with values of 203, 257, 385,
              and 511
       &1.3 = DESPLAINES, ILL
    (end of buffer)

Primary input buffer
  [^ABC^285.73^10]11^
where  %1 = (null value)
       %2 = ABC
       %3 = 285.73 (in external format)
       %4 = multivalued with values of 10 and 11
    (end of buffer)

file buffer 4
  [^
```

This sequence of symbols designates a null buffer cleared of all items. The only attribute that could be said to exist in this buffer is &4.0 and it is a null item.

In cases where a buffer pointer is required to illustrate the functioning of certain commands, the pointer location will be illustrated by an up arrow below the line showing the contents of the buffer. As an example, when showing the input buffer pointer at %3, the display would be:

```
[ABC^DEF]GHI^JKL^MNO^
           ^
```

Since clear recognition of single or multiple spaces which are not readily apparent may sometimes be essential to the understanding of an explanation, individual underlines or dashes below the line will serve to delineate the location and quantity of spaces on the line above. These definitions will be used only where an uncertainty may exist, and, where not used, any space perceived may be assumed to be a single blank character. As shown below:

```
[ABC^ DEF]  GHI]J K   L^
          --    - ---
```

The space immediately prior to the D may be assumed to be a single blank character, and the quantity of blanks in the remaining significant gaps is indicated by the dashes below the buffer display line (two blanks between the value mark and the G, one between the J and K, and three between the K and L).

3.2 BUFFER DISPLAYS

Buffers will be displayed in examples as described above. Where the text discussion pertains to only one buffer, all displays may be assumed to be of that buffer only. Where any ambiguity may arise concerning which buffer is being shown, a direct reference immediately preceeding the display will indicate the first attribute shown. If the input or output buffers are shown and it is significant that it is the primary or secondary buffer, a P or S will preceed the reference.

All such references and the buffers to which they pertain are outlined in the table below:

DIRECT REFERENCE	BUFFER DISPLAYED
%1	Either Input Buffer
P%1	Primary Input Buffer
S%1	Secondary Input Buffer
#1	Either Output Buffer
P#1	Primary Output Buffer
S#1	Secondary Output Buffer
&1.Ø	File Buffer 1
&2.Ø	File Buffer 2
&3.Ø	File Buffer 3
&4.Ø	File Buffer 4
&5.Ø	File Buffer 5
&6.Ø	File Buffer 6
&7.Ø	File Buffer 7
&8.Ø	File Buffer 8
&9.Ø	File Buffer 9
!1	Select Register 1
!2	Select Register 2
!3	Select Register 3
!4	Select Register 4
!5	Select Register 5

For example, the following is the display of the secondary output buffer:

```
S#1     [ABC^DEF^GHI^JKL^MNO^
```

3.3 INDIRECT REFERENCE

Reference may be made to any attribute of any buffer by specifying the buffer and using a value in the primary input or primary or secondary output buffers to designate which attribute is required. To do so, make the normal first half of the direct reference (%, #, &N.) and without any intervening spaces, follow it with the direct reference to the attribute in the input or output buffer which contains the attribute number desired. For example, using only the primary

input buffer containing:

[3^4^7^1^8^B^1Ø^ABC^2^

The use of %%2 would be an indirect reference to %4 since the value of %2 is 4. Any such reference would make the value 1 available from the fourth attribute of the primary input buffer. Similarly, using the same input buffer contents, the references in the two columns below are equivalent:

Indirect	Direct
%%5	%8
&3.%1	&3.3
&7.%7	&7.1Ø
#%3	#7
&1.%6	&1.Ø

Note: as in the last example above, the value of the item used in the second half of the indirect reference must be numeric. If not numeric, zero will be assumed.

All the above examples use the primary input buffer to supply the indirect reference parameter, however, the current output buffer may be used in the same manner. No other buffers may be used to supply the indirect parameter, although attributes in all buffers may be accessed by an indirect reference as shown above.

CHAPTER 4

INITIALIZATION AND COMMENTS

A PROC program will be executed by the interpreter starting with the first command and proceeding with the second, third, etc. Execution may be directed to other than the next command by use of the jumps and branches provided. If execution is proceeding normally and the next line of the program to be executed turns out to be the end of the program, control will drop out of the processor and be returned to TCL.

Many incorrectly coded instructions will cause the program to terminate or continue after outputting an error message. These errors are outlined in the text accompanying the instruction explanation, however, other errors not outlined may also occur. It is up to the individual programmer to decide the proper steps to be taken in case of any abnormal ending to a program.

4.1 PQ statement

-Format- PQ

The first statement of any PROC must be the literal value PQ which indicates to the interpreter that the item is a RPL interpretive program that must be run by the rules of the PROC processor. Any item stored in any file on the computer is capable of being executed as a PROC if it contains a PQ as the first attribute. Conversely, any item not having a PQ in the first statement will not be executed as a PROC.

4.2 C statement

-Format- Cliteral-string

The C statement is used for inserting comments into the body of a program for explanations of how the program works, etc. All such statements may contain a literal string behind the C which may contain any characters at all. It is generally recommended that enough comments be included in a program to allow another programmer who is unfamiliar with the PROC to understand how the program functions. This will also insure that the person who coded the program in the first place will be able to go back and understand the functioning later.

Comment statements may be inserted anywhere in the program but care should be taken not to place them between any multiple line statements such as L or T statement which continue on the next line. Some examples of valid C

statements are included below:

```
C   THIS PROGRAM WILL CALCULATE THE NEXT PRIME NUMBER

C *** INPUT BUFFER USAGE
C       %1 - NEXT AVAILABLE FILE BUFFER 1 LOCATION
C       %2 - NUMBER TO TEST
C       %3 - NEXT DIVISOR

CALCULATE TAX FOR SALE WITH HALF ROUNDING TO NEAREST CENT
```

In the last example, the C is contained within the text of the comment. This is valid as long as the first character on the line is a C.

4.3 Other Comments

Comments may be included in a program other than those in a comment statement. Any command which does not load a literal string may be terminated with a space after the last required operand. Any subsequent characters are skipped by the processor in finding the next command to execute. Therefore, a comment may be placed at the end of any such command and not disturb the execution of the program. This function is not available on such commands as IH or H since these load the entire following string as a literal.

Use of this capability for loading comments should be minimized. These comments will not be cleared when a program is optimized and will result in a longer program than necessary being stored. Additionally, execution of a program will be slowed since the processor must take time to scan the characters, even if they are not executed as a command.

CHAPTER 5

INPUT BUFFER OPERATIONS

5.1 RI command

-Format- RIparameter

The RI command is used to reset the primary input buffer. It will clear the buffer and reset the pointer to the new end of the buffer. The RI command will set the input buffer stack pointer to the primary input buffer and perform the clear operation designated. When used alone, RI will clear the entire primary input buffer. When used with a numeric value following the RI, it will clear all the input buffer from the attribute designated and place the end of buffer after the preceeding attribute. The following will illustrate the function of the RI command:

Before	Command	After
[ABC^DEF]GHI^JKL^MNO^	RI3	[ABC^DEF]GHI^
[ABC^DEF]GHI^JKL^MNO^	RI	[^

The RI command may also use a direct or indirect reference to provide the attribute from which the buffer is to be cleared. This may be coded as:

RIindirect-reference

For example:

```
RI%3
RI&3.2
RI&3.%5
RI#1
```

The RI command may also be used to clear the input buffer from a given column position. By coding RI(n), where n is a numeric literal, the command will clear the input buffer from the n'th character to the end. If the end of the first attribute occurs before the n'th character, all attributes other than %1 will be cleared and the end of buffer established at %2.

5.2 S command

-Format- Snumeric-literal

The S statement will set the input buffer stack pointer to the primary input buffer and will set the primary input buffer pointer to the attribute mark immediately before the item specified by the numeric value in the second half of the command. For example, S17 will set the pointer to the first character of the seventeenth item in the primary input buffer. All operations dealing with the current location of the input buffer will then reference the seventeenth item. If, however, the sixteenth item has not been established, the primary input buffer pointer will reference the current end of the buffer. The results of execution of an S5 statement with two different buffer contents is shown below with the arrow below the line designating the resultant position of the primary input buffer pointer:

```
[ABC^DEF^GHI^JKL^MNO^PQR^STU^
 ^

[ABC^DEF^
 ^
```

The contents of any attribute may be substituted for the numeric literal in the S command by coding the command with a direct or indirect reference. This form will cause the input buffer pointer to be turned off and the primary input buffer pointer to be moved to the attribute designated by the value of the attribute used. For example:

```
Before:    &4.Ø     [ABC^DEF^3^GHI^
           P%1      [ABC^DEF^GHI^JKL^MNO^PQR^
                        ^

Command:   S&4.2
After:     P%1    [ABC^DEF^GHI^JKL^MNO^PQR^
                      ^
```

The input buffer pointer may be set to a character within the first attribute by coding S(n). This will set the input buffer stack pointer to the primary input buffer and will move the input buffer pointer to the n'th column of the first attribute. If the first attribute of the primary input buffer contains less than n characters, the pointer will move to the attribute mark between %1 and %2. As shown below:

```
Before:    P%1      [ABCDEFGHIJKL^MNO^PQR^
                        ^

Command:   S(7)
After:     P%1      [ABCDEFGHIJKL^MNO^PQR^
                          ^
```

```
Before:    P%1      [ABCDEFGHIJKL^MNO^PQR^
                                  ^

Command:   S(23)
After:     P%1      [ABCDEFGHIJKL^MNO^PQR^
                                  ^
```

5.3 F (forward) command

-Format- F

The F command will move the current input buffer pointer forward to the next attribute mark. If the input buffer pointer is already at the end of the buffer, a new null field will be created. For example:

```
     Before                              After
[ABC^DEF]GHI^JKL^MNO^          [ABC^DEF]GHI^JKL^MNO^
    ^                                      ^

[ABC^DEF^                      [ABC^DEF^
        ^                              ^

[ABC^DEF]GHI^                  [ABC^DEF]GHI^
 ^                              ^
```

5.4 B command

-Format- B

The B command will move the current input buffer pointer back to the previous attribute mark. If the pointer is already at the beginning of the buffer, this command will not have any effect. For example:

```
     Before                              After
[ABC^DEF]GHI^JKL^MNO^          [ABC^DEF]GHI^JKL^MNO^
          ^                            ^

[ABC^DEF^                      [ABC^DEF^
 ^                              ^

[ABC^DEF]GHI^                  [ABC^DEF]GHI^
        ^                            ^
```

5.5 IH command

-Format- IHliteral-string

This command will input a string of characters or the contents of a direct or indirect reference following the IH into the current input buffer at the location specified by the input buffer pointer. The value entered will replace the attribute immediately after the pointer. For literals only, multiple blanks are considered as one blank, leading or trailing blanks are ignored, single backslashes become null

fields, and multiple or imbedded backslashes become blanks. A space appearing within the literal string being input will be converted to an attribute mark. Use of the IH command with only a backslash following (as IH\) will be used to null an attribute and all appearances of a space-backslash-space sequence within the literal string will establish a null attribute.

Certain special characters may not appear in an IH statement literal string. Among these are the value mark (M[cs]) and the sub-value mark (L[cs]). If any invalid characters do appear in this statement, it will cause the resultant values loaded in the buffer to be truncated.

The operation of the IH command is illustrated below:

```
Before:     [ABC^DEF]GHI^JKL^MNO^
                     ^

Command:    IH\
After:      [ABC^DEF]GHI^^MNO^
                     ^
```

Note that %3 is now a null attribute and the backslash was not loaded.

```
Before:     [ABC^DEF]GHI^JKL^MNO^
                     ^

Command:    IH12345
After:      [ABC^DEF]GHI^12345^MNO^
                     ^


Before:     [ABC^DEF]GHI^JKL^MNO^
                ^

Command:    IH12
After:      [ABC^12^JKL^MNO^
                ^
```

As illustrated above, the input value replaces the entire contents of the next attribute in the current input buffer.

```
Before:     [ABC^DEF]GHI^JKL^MNO^
                     ^

Command:    IH123 456 7\9 \ Ø
After:      [ABC^DEF]GHI^123^456^7 9^^Ø^MNO^
                     ^
```

This example shows that the entire new value loaded replaces only the previous contents of one attribute. Because five new attributes replaced the one previously existing as %3, the former value of %4 (MNO) now appears as %8. Had there been subsequent values in %5, %6, etc., they would also be bumped up to %9, %1Ø, etc., respectively. Notice also that the backslash within the 7\9 field was converted to a space when loaded.

```
Before:     [ABC^DEF]GHI^JKL^MNO^
                                ^

Command:    IH123 456 789 \ Ø
After:      [ABC^DEF]GHI^JKL^MNO^123^456^789^^Ø^
                            ^
```

In the case where the pointer is at the end of the buffer, no attribute is replaced but additional attributes are built as needed to load the values specified in the command.

Backslashes which appear with other characters will be converted to spaces before being loaded into the input buffer and will appear in the same attribute with the other characters:

```
Before:     [ABC^DEF]GHI^JKL^MNO^
                        ^

Command:    IH12 34\\56 7Q\\\ \3B
After:      [ABC^DEF]GHI^12^34  56^7Q   ^ 3B^MNO^
                        ^     --    ---  -
```

Multiple contiguous spaces appearing within an IH statement will have no more significance than one space appearing alone. All such contiguous strings of spaces will generate only one attribute mark each.

```
Before:     [ABC^DEF]GHI^JKL^MNO^
             ^

Command:    IH123   456      789
                ---   ------   ---
After:      [ABC^123^456^789^JKL^MNO^
             ^
```

The second function performed by an IH statement is loading the contents of a direct or indirect reference location into the attribute of the input buffer indicated by the current input buffer pointer. This function will only be intepreted by the processor if the reference immediately follows the IH without any intervening characters. Any other buffer or the select registers may provide the source item. If the select registers are used as the source item, the current value of the multivalued field will be moved to the input buffer, and the pointer will move to the next value. The operation of this function is illustrated below:

```
Before:     %1    [ABC^DEF^GHI^JKL^MNO
                        ^

            &2.Ø  [111^222^333^444^555^
Command:    IH&2.3
After:      %1    [ABC^DEF^444^JKL^MNO^
                        ^
```

```
Before:     %1    [Ø^1^2^3^4^5^6^7^8^
                      ^
            &3.Ø  [ABC^DEF^GHI^JK L^MNO^
Command:    IH&3.%4
After:      %1    [Ø^1^JK L^3^4^5^6^7^8^
                      ^

Before:     %1    [ABC^DEF^GHI^JKL^MNO^
                         ^
            !1    [QQQ^RRR^MMM^SSSS^T^
                         ^
Command:    IH!1
After:      %1    [ABC^DEF^MMM^JKL^MNO^
                         ^
            !1    [QQQ^RRR^MMM^SSSS^T^
                             ^
```

The last example above shows the movement of the select register pointer after transfering the current value. All other examples illustrate that the input buffer pointer does not move during one of the IH command operations.

```
Before:     %1    [ABC^DEF^GHI^JKL^MNO^
                         ^
            &1.Ø  [123^456^^^^555^666^
Command:    IH&1.3
After:      %1    [ABC^DEF^^JKL^MNO^
                         ^

Before:     %1    [ABC^DEF^GHI^JKL^MNO^
                         ^
Command:    IHCHANGE&1.3
After:      %1    [ABC^DEF^CHANGE&1.3^JKL^MNO^
                         ^
```

When a direct (or indirect) reference appears within other characters of a literal string of an IH command, it will load as a literal and will not call the referenced value.

Items being loaded into the input buffer via the IH command may be converted from internal to external format or vice-versa. This is accomplished by appending the conversion code immediately behind the direct or indirect reference of the item to be moved into the current attribute of the input buffer as shown below:

```
IH&3.2:D:     Will convert the date to external format
IH%3;MD2;     Will remove decimal and leave two
                 decimal digits
```

Further description of conversions may be obtained by referring to the section on conversions in Chapter 15.

CHAPTER 6

OUTPUT BUFFER OPERATIONS

6.1 STON, STOFF commands

-Format- STON
-format- STOFF

The stack on and stack off commands affect only the output buffer stack pointer and will not affect either buffer's data pointer or the contents of the buffers. The STOFF command will place the stack pointer to the primary output buffer and the STON command will point it at the secondary output buffer. In the text accompanying explanations of commands dealing with the output buffer the term 'current output buffer' will refer to the output buffer which the stack pointer is designating since the other buffer may not be manipulated until the stack pointer is changed to point to it.

6.2 RO command

-Format- RO

The RO command is used to clear the current output buffer and reset the pointer to the beginning of the buffer. The buffer cleared may be either the primary or secondary output buffer depending on the setting of the stack pointer. The stack pointer and the contents of the other output buffer will not be affected by the RO command. The following table and examples will illustrate the function of the RO command:

Stack	Clears
OFF	Primary Output Buffer
ON	Secondary Output Buffer

```
  Before:    P#1    [ABC^DEF^GHI^JKL^MNO^
                       ^
             S#1    [11^22^33^44^55^
                     ^
```

With the stack pointer on, RO will result in:

```
  After:     P#1    [ABC^DEF^GHI^JKL^MNO^
                       ^
             S#1    [^
                     ^
```

6.3 BO command

-Format- BO

The BO command will move the current output buffer pointer back over one parameter to the previous attribute mark. If the pointer is already at the beginning of the buffer this command will have no effect. Since the output buffer pointer also designates the end of the buffer, this command has the effect of deleting the last attribute from the end of the current output buffer. For example, illustrating only the current output buffer:

```
      Before                          After
[ABC^DEF]GHI^JKL^                [ABC^DEF]GHI^
                ^                               ^
[^                               [^
 ^                                ^
```

6.4 H command

-Format- Hliteral-string

The H command will input a literal string following the H into the current output buffer starting at the location specified by the output buffer stack pointer. The string loaded will extend the length of the buffer as required to accomodate the input. All spaces will be replaced by one attribute mark each. After completion, the buffer pointer will be directed to the attribute mark immediately following the last character loaded.

The operation of the H command is illustrated below with only the current buffer being shown; therefore, the example may represent operation within the primary or secondary output buffers, depending on the setting of the stack pointer.

```
Before:    [ABC^DEF^
                   ^

Command:   HNEXT
After:     [ABC^DEFNEXT^
                        ^


Before:    [ABC^DEF^
                   ^

Command:   H NEXT
After:     [ABC^DEF^NEXT^
                        ^
```

Note as illustrated in the above two examples, the H command does not automatically construct a new attribute mark upon starting as does the IH statement, but appends the string entered immediately at the position of the buffer pointer.

```
Before:    [ABC^DEF^
                   ^

Command:   H   NEW     INPUT
            ---   -----     ---
After:     [ABC^DEF^^^NEW^^^^^INPUT^^^^
                                    ^
```

As previously stated, all spaces will produce one attribute mark per space coded in the literal.

Attributes located elsewhere in any buffer may be moved into the output buffers by using the H statement followed by a direct or indirect reference which designates what attribute to move. Any internal buffer may be used to provide the source value including another attribute of the current output buffer. The only buffer which cannot be used to provide the source value is the other output buffer since there is no way of referencing both buffers in one statement. The following is a series of valid moves using the H statement:

```
Before:    #1      [ABC^DEF^GHI^^
                                ^
           &3.Ø    [111^222^333^444^
Command:   H&3.2
After:     #1      [ABC^DEF^GHI^333^
                                   ^


Before:    #1      [ABC^DEF^GHI^
                               ^
           &3.Ø    [111^222^333^444^
Command:   H&3.2
After:     #1      [ABC^DEF^GHI333^
                                  ^
```

CHAPTER 7

DATA MOVE OPERATIONS

7.1 A command

-Format- Ac(m,n)

The A command is used to move selected data from the input buffers to the current output buffer. The source location may be either the primary or secondary input buffer depending on the setting of the input buffer stack pointer or the form of the command used. The object location will be the current location of the current output buffer. There are ten possible forms of the A command using four optional parameters. The A command will move all or part of one attribute in the input buffers to the output buffers depending on the form of the statement used. The valid forms of the A command are listed below with symbolic parameters:

A	Ac
Ap	Acp
A(m)	Ac(m)
A(,n)	Ac(,n)
A(m,n)	Ac(m,n)

The symbolic parameters used above are defined below with a short description of the function which they perform.

P A numeric literal designating the attribute of the current input buffer from which to obtain the source string. When used, the entire attribute will be moved.

M A numeric literal designating a starting character position in the first attribute of the primary input buffer. If less than m characters exist in the first attribute when this parameter is specified, no move will occur.

N A numeric literal designating a number of characters to be extracted from the source item for the move. If not specified, the command will move all characters until the next attribute mark is encountered. If less than n characters follow in the source attribute, only they will be moved.

C Any non-numeric character which will be used to surround the string to be moved into the output buffer. If not specified, an attribute mark will be used. When the destination of the move is the secondary output buffer, this parameter will be ignored.

The general function of the A command will be to move a string of characters from the input buffers to the current output buffer. The first character moved will be the current position of the input buffer pointer, or the location to which that pointer is moved by a p or m parameter specified in the command. The characters which will be moved will be to the next attribute mark or the quantity specified in the n parameter if used. The resultant position of the current input buffer pointer will be the next character after the last one moved.

The destination of the character string moved may be either the primary or secondary output buffer, depending only on the output buffer stack pointer. The resultant string that will be moved to the destination will vary depending on whether the destination is the primary or secondary output buffer. If directed to the primary output buffer, the resultant string will be surrounded by the character c or by attribute marks if c is not specified. If the character defaults to attribute marks, one attribute mark will appear at the front of the string which is loaded, but none will be placed at the end of the string. If the destination is the secondary output buffer, the source string will be moved intact into the current position of the buffer and any surround characters will be ignored.

The following series of examples will illustrate the functioning of the A command. In most cases, the resultant contents of both output buffers will be shown to highlight the difference in operation of the command, though any operation will affect only the current output buffer. The input buffer may be either the secondary or primary input buffer unless specifically designated in the buffer description by P or S.

```
Before:     %1      [ABC^DEF^GHI^JKL^MNO^
                      ^
            #1      [PREVIOUS^INPUT^
                              ^
Command:    A
After:      %1      [ABC^DEF^GHI^JKL^MNO^
                         ^
            P#1     [PREVIOUS^INPUT^DEF^
                                   ^
            S#1     [PREVIOUS^INPUTDEF^
                                ^
```

```
Before:     %1      [ABC^DEFGHI^JKL^MNO^
                        ^
            #1      [SSELECT^FILENAME^^
                            ^
Command:    A
After:      %1      [ABC^DEFGHI^JKL^MNO^
                           ^
            P#1     [SSELECT^FILENAME^FGHI^
                                     ^
            S#1     [SSELECT^FILENAME^FGHI^
                                     ^
```

In the two examples above, the move from the source item ended when an attribute mark was encountered, the input buffer pointer was replaced to the end of the source attribute, and a new attribute mark was constructed in the output buffer before the data was moved into its new location. Either the primary or secondary input buffer could be used here to supply the source item and the functioning would be the same.

```
Before:     %1      [ABC^DEF^GHI^JKL^MNO^
                           ^
            #1      [SSELECT^FILENAME^^
                            ^
Command:    A2
After:      %1      [ABC^DEF^GHI^JKL^MNO^
                        ^
            P#1     [SSELECT^FILENAME^DEF^
                                      ^
            S#1     [SSELECT^FILENAME^DEF^
                                     ^
```

Despite the prior positioning of the input buffer pointer in the above example, the move came from attribute 2 of the input buffer since it was explicitly stated in the command. The resultant positioning of the input buffer pointer is immediately after the last character moved.

```
Before:     %1      [ABC^DEFGHI^JKL^MNO^
                        ^
            #1      [SSELECT^FILENAME^
                             ^
Command:    A"
After:      %1      [ABC^DEFGHI^JKL^MNO^
                           ^
            P#1     [SSELECT^FILENAME"FGHI"^
                                      ^
            S#1     [SSELECT^FILENAMEFGHI^
                                     ^
```

```
Before:     %1      [ABC^DEF^GHI^JKL^MNO^
                    ^
            #1      [SSELECT^FILENAME^^
                      ^
Command:    A"2
After:      %1      [ABC^DEF^GHI^JKL^MNO^
                      ^
            P#1     [SSELECT^FILENAME^"DEF"^
                                       ^
            S#1     [SSELECT^FILENAME^DEF^
                             ^
```

The two examples above illustrate the use of the character surround capability of the A command. As shown in both cases no additional attribute mark was constructed in the output buffer as in the other previous examples. Despite the exclusive use of the double quote for the illustrations, any non-numeric character may be used for this function.

```
Before:     %1      [ABC^DEFGHIJKL^MNO^PQR^
                        ^
            #1      [SSELECT^FILENAME^
                             ^
Command:    A(,2)
After:      %1      [ABC^DEFGHIJKL^MNO^PQR^
                        ^
            P#1     [SSELECT^FILENAME^GH^
                                     ^
            S#1     [SSELECT^FILENAMEGH^
                             ^
```

This operation moved from the current position of the input buffer pointer for a length of two characters. The input buffer pointer may have been left where it was by another similar command executed sometime before the operation shown above.

```
Before:     P%1     [ABCDEFGHIJKL^MNOPQR^STUVWX^
                                    ^
            #1      [SSELECT^FILENAME^
                             ^
Command:    A(3,5)
After:      P%1     [ABCDEFGHIJKL^MNOPQR^STUVWX^
                          ^
            P#1     [SSELECT^FILENAME^CDEFG^
                                     ^
            S#1     [SSELECT^FILENAMECDEFG^
                             ^
```

```
Before:     P%1     [ABCDEF^GHI^JKL^MNO^
                                    ^
            #1      [SSELECT^FILENAME^
                                    ^
Command:    A(3,7)
After:      P%1     [ABCDEF^GHI^JKL^MNO^
                           ^
            P#1     [SSELECT^FILENAME^CDEF^
                                    ^
            S#1     [SSELECT^FILENAMECDEF^
                                    ^
```

In all A(m,n) operations the source item will come from the first attribute of the primary input buffer. The second example above shows the operation when the end of the attribute is encountered before the number of characters specified in the n parameter have been moved. In this case the operation terminates at the attribute mark and the primary input buffer pointer is set at the %2 attribute mark.

7.2 MV command

-Format- MV object-item source-item1,source-item2,...

A move statement may be used to move data into any buffer area (a file buffer, a select register, or the input or output buffers). A move statement must have two operands, the second of which may be a complex string. The first of the two operands will be the object of the move, where the values are to be moved. The first operand may consist of any valid direct or indirect reference. The second operand may contain one value or a string of values separated by commas. Each value will be moved into successive locations in the first operand location, and may individually be a literal, a direct reference, an indirect reference, or a built up combination of any of the preceeding, constructed according to syntax explained later.

The simplest form of the move statement contains only one value as the second operand. This may be any valid direct or indirect reference:

```
MV &3.1 &5.2
MV &3.2 %17
MV !1 &7.%3
MV %1 !2
```

Note: the above is the only valid form which may be used with a select register in the first operand. All other forms are valid for any reference in the first operand.

The second operand may also contain a literal which must be surrounded by double quotes:

```
MV &9.Ø "BEN FRANKLIN"
```

If the literal is simply two double quotes without any intervening literal string, the attribute specified as the object of the move will be cleared to a null item:

```
Before:     &4.Ø    [ABC^DEF^GHI^JKL^MNO^
Command:    MV &4.3 ""
After:      &4.Ø    [ABC^DEF^GHI^^MNO^
```

The second operand may contain a string of values, separated by commas, which will be loaded into consecutive attributes of the file buffer, starting at the location specified in the first operand. The statements on the left below and the series of statements on the right are logically equivalent:

```
MV &2.1 &5.2,%3,!5          MV &2.1 &5.2
                            MV &2.2 %3
                            MV &2.3 !5

MV &1.5 "FIRST",&3.1        MV &1.5 "FIRST"
                            MV &1.6 &3.1
```

A string of references used as the second operand will all be filled with the values which existed before the move statement was executed, even if one or more of those references is an attribute which was changed elsewhere in the same move statement. The following:

```
MV &5.2 &3.1,&5.2,&5.3
```

Will move the old values of &5.2 and &5.3 into &5.3 and &5.4 respectively and will not fill all three attributes with the previous contents of &3.1. Therefore, the previous statement is logically equivalent to (note order of move):

```
MV &5.4 &5.3
MV &5.3 &5.2
MV &5.2 &3.1
```

If the object of the MV statement (the first operand) does not have items defined to the point where the first item will be loaded, the processor will construct enough null attributes to load the source items in the locations they were directed to by the statement. The current end of buffer will then be established at the last item loaded.

```
Before:     &1.Ø    [ABC^DEF^
Command:    MV &1.5 "GHI","JKL"
After:      &1.Ø    [ABC^DEF^^^^GHI^JKL^
```

If the object of the move statement is the input buffer, the input buffer stack pointer will be directed to the primary

input buffer (turned off) and the primary input buffer pointer will be redirected to the first item loaded by the statement.

```
Before:     %1      [ABC^DEF^GHI^JKL^MNO^PQR^STU^VWX^
                                                 ^

Command:    MV %3 "12","34","56"
After:      %1      [ABC^DEF^12^34^56^PQR^STU^VWX^
                        ^
```

Additionally, should any spaces be included in literals or direct reference item, they will be loaded in one item and will not cause extra attribute marks to be formed as in the IH and IP statements, nor are backslashes converted to blanks by this command.

```
Before:     %1      [ABC^DEF^GHI^JKL^
                     ^

Command:    MV %2 "12  34  56\\"
                      --  --
After:      %1      [ABC^12  34  56\\^GHI^JKL^
                     ^   --  --
```

If the object of the move statement is the output buffer, the current output buffer pointer will not move unless the quantity of attributes is changed by the move statement. It will always point to the end of the buffer.

```
Before:     #1      [ABC^DEF^GHI^JKL^MNO^
                                        ^

Command:    MV #3 "12","34","56"
After:      #1      [ABC^DEF^12^34^56^
                                    ^


Before:     #1      [ABC^DEF^GHI^JKL^
                                    ^

Command:    MV #7 "12","34","56"
After:      #1      [ABC^DEF^GHI^JKL^^^12^34^56^
                                              ^
```

When used alone as the last item in a string of the second operand, an asterisk (*) will move all remaining items of the buffer referenced in the previous item of the string (which must be a direct or indirect reference) as if they were individually referenced. The asterisk so used may not follow any item other than a buffer reference (may not follow a literal, etc) and must be the last item in the string.

```
MV &5.2 &7.1,"SAME",&4.7,*
```

Will operate the same as:

```
MV &5.2 &7.1,"SAME",&4.7,&4.8,&4.9,&4.10,.....
```

The move will stop when the previously established last

attribute of file buffer 4 is reached and the end of buffer 5 will be created at that point. All previous attributes existing beyond that point in file buffer 5 will be lost.

When used in conjunction with a numeric value the *n item in the second operand of the MV statement will move n additional items from the just referenced buffer into the n contiguous locations in the first operand location. If the end of the source buffer is reached, the move will continue filling the object buffer with null attributes until the quantity of items specified has been loaded. The source buffer will not be changed.

```
Before:     &1.Ø    [ABC^DEF^GHI^JKL^MNO^PQR^STU^VWX^
            &2.Ø    [1^2^3^4^5^6^7^8^9^Ø^
Command:    MV &1.2 &2.1,*3
After:      &1.Ø    [ABC^DEF^2^3^4^5^STU^VWX^

Before:     &1.Ø    [ABC^DEF^GHI^JKL^MNO^PQR^STU^VWX^
            &2.Ø    [1^2^3^4^5^
Command:    MV &1.2 &2.2,*4
After:      &1.Ø    [ABC^DEF^3^4^5^^^VWX^
```

Any item in the second operand may be a built up combination of other attributes or literals by appending them together with an imbedded asterisk. This asterisk will not appear in the built up string but is used only as a means of combining other items.

```
MV &5.2 &4.7*%3
```

Assuming the previous values of &4.7 and %3 were '413A 7' and '7A1\B' respectively, the resulting contents of &5.2, due to the above instruction, would be '413A 77A1\B'. These items may be as complex as desired. For example:

```
MV &5.2 &4.7*"*"*%3,"73"*&1.Ø*"SAME"
```

The source items in the MV statement may also contain 'A(m,n)' items constructed as defined in the corresponding section devoted to discussion of that command. In this case the source is always the primary input buffer and no change will occur in the contents of the output buffer unless otherwise used as the object of the MV command. A facility has been added to the MV command to allow efficient conversion and storage of numeric items being loaded from card image records. This is the N(m,n) type of extraction which will function similarly to the A(m,n) command except that it will drop all leading insignificant leading zeros when moving the field to the object buffer. These functions could be used as shown below:

```
Before:     %1       [ABC0001234^3152^MM1^
            &1.0     [AAA^BBB^CCC^DDD^EEE^FFF^
Command:    MV &1.1 A(1,5),N(4,6),A(8,6)
After:      &1.0     [AAA^ABC00^123^234 31^EEE^FFF^
```

An end of buffer may be forced on the object buffer by using a back arrow (←, O[s]) in the last item in the second operand of the move command. The object buffer will then be truncated to the last item loaded in the move statement and all remaining attributes previously existing will be lost.

```
Before:     &4.0     [AA^BB^CC^DD^EE^FF^GG^HH^II^JJ^KK^
Command:    MV &4.2 "START",&4.5,*2,←
After:      &4.0     [AA^BB^START^FF^GG^HH^
```

If divergent attributes must be changed without disturbing intervening items, a series of commas in the second operand will skip the number of attributes in the object buffer which were skipped with the commas. With this function, the previous contents of the skipped attributes of the object buffer will remain unchanged.

```
Before:     &8.0     [ABC^DEF^GHI^JKL^MNO^PQR^STU^VW^
Command:    MV &8.2 "START",,,,"END"
After:      &8.0     [ABC^DEF^START^JKL^MNO^PQR^END^VW^
```

If the skip function is used and the next attribute after the skip falls beyond the current end of the buffer, null attributes will be constructed to fill the gap up to the item at which the next load is to take place.

```
Before:     %1       [ABC^DEF^
Command:    MV %2 "START",,,,,,"END"
After:      %1       [ABC^START^^^^^^END^
```

All of the functions available in the move statement may be combined in one MV statement in any order desired, except the back arrow, which must be the last item in the second operand. Mixing of complex built up items, skipped items, multiple moves by *N, literals, and direct and indirect references may be accomplished by following the syntax rules for each item individually. For example an *N may be used as long as it follows a reference to an item in a file buffer and not a literal or a built up string. The following is a series of examples of complex mixed operands which may be constructed when using the MV statement.

```
MV %3 &5.2,*3,"THEN*"*&3.%5*%3,,,&7.1,*3,
MV &7.1 #1,#2,#3,#4,*3,"UNUSED",%1*" "*%2*" "*%3,"","
MV #1 ←
```

7.3 MVA command

-Format- MVA object-item source-item

The MVA command is used to move items into a multivalued string in an attribute of the buffer areas. The source item (second operand) and object item (first operand) must be single direct or indirect references (literals, etc are not valid). The source item will be placed into the object item string intact, separated by value marks, and placed in collating sequence. If the source item itself is a multivalued string, it will not be split into component parts but will be placed, as is, into the object item.

```
Before:   &1.Ø   [ABC^DEF[GHI^JKL^MNO^
Command:  MVA &1.1 &1.3
After:    &1.Ø   [ABC^DEF]GHI]MNO^JKL^MNO^

Before:   &1.Ø   [ABC^DEF]GHI^JKL^FOR^
Command:  MVA &1.1 &1.3
After:    &1.Ø   [ABC^DEF]FOR]GHI^JKL^FOR^
```

In both of the above cases the source item was moved into the object string in collating sequence and an extra value mark was constructed to separate it from the rest of the data.

```
Before:   &1.Ø   [ABC^DEF]GHI^JKL^MNO^
          &3.Ø   [AAA^BBB^EXTRA]MONEY]WAGE^END^
Command:  MVA &1.1 &3.2
After:    &1.Ø   [ABC^DEF]EXTRA]MONEY]WAGE]GHI^JKL^MNO^
```

The multivalued string in &3.2 was placed intact into the string in &1.1 without splitting the individual values and placing them in collating order.

```
Before:   &1.Ø   [ABC^^^^DEF]GHI^JKL^MNO^
          %1     [123^456^789^111^222^333^
Command:  MVA &1.2 %3
After:    &1.Ø   [ABC^^789^^DEF]GHI^JKL^MNO^
```

In the case where the object item is null, the MVA command will function like a MV command and will simply move the source item to the new location.

```
Before:   &1.Ø   [ABC^DEF]GHI]JKL]MNO^PQR^
          &7.Ø   [123^456^GHI^789^
Command:  MVA &1.1 &7.2
After:    &1.Ø   [ABC^DEF]GHI]JKL]MNO^PQR^
```

Where the source item already exists in the object string in the proper collating order, the item will not be reloaded. Duplicates will not exist in a multivalued field loaded value by value with an MVA statement.

```
Before:   &1.Ø    [ABC^DEF]GHI]JKL]MNO^PQR^
          &7.Ø    [123^456^GHI]JKL^789^
Command:  MVA &1.1 &7.2
After:    &1.Ø    [ABC^DEF]GHI]GHI]JKL]JKL]MNO^PQR^
```

When the source value contains a value mark, it cannot compare equal to any value in the the object field since comparison in the object field will restart every time a value mark is encountered. For this reason, as shown above, duplicates may appear in the the multivalued string despite the fact that an identical string already exists in the object field.

```
Before:   &1.Ø    [ABC^DEF]GHI]JKL]MNO^PQR^
          &7.Ø    [123^456^PQR]GHI^789^
Command:  MVA &1.1 &7.2
After:    &1.Ø    [ABC^DEF]GHI]JKL]MNO]PQR]GHI^PQR^
```

Duplicates may also be caused, out of collating order, anytime the source item contains a multivalued field, since no comparison is possible between values in the source and object field.

7.4 MVD command

-Format- MVD object-item source-item

The MVD command will cause a value to be deleted from a multivalued field. The value of the source field, designated by a direct or indirect reference in the second operand, will be deleted from the multivalued string of the object field, designated by a direct or indirect reference in the first operand. If the value exists more than once in the first operand field, only the first occurance will be deleted. If the value does not exist at all in the first operand field, there will be no change to it. If the source item is a multivalued field, nothing will be changed in the first operand field since no match will ever be found. When the object item is the input buffer, the buffer pointer will be moved to point to the beginning of the object attribute.

The operation of the MVD command is illustrated below. For simplicity of the displays, the source item will always be in the same buffer as the object item though this is not a restriction of the command.

```
Before:   &1.Ø    [ABC^DEF]GHI]JKL]MNO^GHI^
Command:  MVD &1.1 &1.2
After:    &1.Ø    [ABC^DEF]JKL]MNO^GHI^
```

```
Before:    &1.Ø     [ABC^DEF]GHI]JKL]GHI^GHI^
Command:   MVD &1.1 &1.2
After:     &1.Ø     [ABC^DEF]JKL]GHI^GHI^

Before:    %1       [ABC^DEF]GHI]JKL^GHI]JKL^
Command:   MVD %2 %3
After:     %1       [ABC^DEF]GHI]JKL^GHI]JKL^
                       ^
```

The last example illustrates the action when the source item is a multivalued filed. Nothing can be deleted despite the exact match since the comparison in the object field restarts when a value mark is encountered. This prohibits a match to a field which has a value mark in it.

CHAPTER 8

TERMINAL AND TAPE INPUT

8.1 IP command

-Format- IPoptions/prompt-character/object-item

The IP statement will input a value from the terminal into an internal buffer. The actual value loaded and the location to which it is loaded will vary depending on the format of the input statement. Associated with any input statement is a prompt character which will appear at the current location on the terminal to show that the program is ready to accept input. The default value for this prompt character is the colon (:), as used in prompting TCL input.

The simplest form of the input statement is using IP only on a line. This form uses the current prompt character and inputs the actual value entered on the terminal keyboard into the current location of the current input buffer. If no value is entered at the keyboard (a new-line is entered), the previous contents of the current location of the input buffer will remain unchanged. If a value is entered, it will replace entirely the previous contents of that location in the input buffer.

Given the sequence of instructions:

```
S3
IP
```

The contents of %3 will be affected as follows:

%3 before input	input value	%3 after input
ABC	(CR)(no input)	ABC
ABC	123	123
ABC	X	X
ABC	37ACD	37ACD

If spaces are entered at the keyboard, any space or string of spaces will create one attribute mark in the input buffer. The entire string entered at the keyboard will nevertheless replace only one attribute in the input buffer as shown below:

```
Before:    [ABC^DEF]GHI^JKL^MNO^
               ^
Input:     123 456    789
               -   ----
After:     [ABC^123^456^789^JKL^MNO^
               ^
```

Because three new items replaced the previous one, the values which were %3 and %4 (JKL and MNO) must now be referenced as %5 and %6 respectively. If a single backslash is entered at the keyboard, it will load a null field. However, a string of backslashes surrounded by spaces will generate the same quantity of spaces in an attribute as the quantity of backslashes entered. Backslashes otherwise entered as part of another item will be converted to spaces.

```
Before:    [ABC^DEF]GHI^JKL^MNO^
                ^

Input:     12 \ 34 \\\ 3\M
After:     [ABC^12^^34^    ^3 M^JKL^MNO^
                ^       ---  -
```

A space or series of spaces entered alone at the keyboard will load a null attribute.

```
Before:    [ABC^DEF]GHI^JKL^MNO^
                        ^

Input:         (space)
               -

After:     [ABC^DEF]GHI^^MNO^
                        ^
```

By changing the format of the statement to IPB, all spaces entered will not be converted to attribute marks but will be loaded as literal spaces into the one attribute designated by the input buffer pointer and all backslashed entered will load without conversion. For example, assuming an IPB statement doing the input:

```
Before:    [ABC^DEF^GHI^JKL^MNO^
                ^

Input:     12  34\M  \
             --    --

After:     [ABC^DEF^12  34\M  \^JKL^MNO^
                ^  --    --
```

Either of the two above formats, IP or IPB, will use the current prompt character which defaults to the colon (:), used also to prompt TCL input. Any other character may be specified as the prompt character by appending it immediately behind the IP or IPB statement (an IP? Will use the question mark as the prompt character). Any character so specified in an input statement will become the current prompt character and will be used to prompt input until another character is appended to an IP or IPB statement. Obviously a B cannot be used as a prompt character in an IP statement since this would be interpreted by the processor as an IPB statement. Use of an N for the prompt character, as IPN will indicate no prompt, and in this case the cursor will not move when the IPN command is executed. One of the common prompts which may be used is a blank character, in which case a space should be entered

behind the IP or IPB statements. The following are a series of examples of input statement with prompt characters:

Input statement	Prompt character
IP	(space)
-	
IPN	(no prompt)
IPBX	X
IP?	?

In cases where it is necessary to clear the object attribute if no data is entered at the terminal during the IP command, an F may be appended to force input. This will cause the attribute to which the input would be loaded to be cleared if no input is made. The forced input function may be combined with the blank conversion by coding either IPBF or IPFB. Any prompt character will follow the options.

The input entered at the keyboard may be directed to areas other than the input buffer. Alternate input areas may be the output buffers or any of the file buffers. Input will be directed to the attribute specified in a direct or indirect reference appended to the IP statement (IP&3.2 will place the input into attribute 2 of file buffer 3). The primary input buffer may also be specified as the object of the input statement by using a reference to the attribute in which the input is to be placed. Any input made by using this format will be loaded into one attribute, regardless of spaces entered, and all backslashes will remain as entered. Because of this functioning there is no difference between the operation of the IP and the IPB statements when used in this format. Prompt characters may be included by coding the IP, prompt character, and object attribute in order into one statement. For example, to obtain a prompt character of a question mark and place the input into attribute 4 of file buffer 7, use an IP?&7.4 statement. This function may also be executed with either or both of the options as IPB, IPF, or IPBF.

If, when using the directed form of the input statement, the object attribute falls beyond the current end of the buffer, null attributes will be automatically constructed to fill the area up to the attribute into which the item is to be loaded. Following is a series of examples of the use of the directed input statement. In all cases, the buffer shown is the buffer into which the input is placed and will be designated by a direct reference preceeding the display.

```
Before:    &6.Ø    [ABC^DEF^GHI^JKL^MNO^
Command:   IP&6.3
Input:     12  34M\  X
             --    -- -
After:     &6.Ø    [ABC^DEF^GHI^12  34M   X ^MNO^
                                 --   --- -

Before:    %1      [ABC^DEF^
Command:   IP$%7
Input:     385.72
After:     %1      [ABC^DEF^^^^^385.72^

Before:    #1      [ABC^DEF^GHI^JKL^MNO^
Command:   IPBX#1
Input:     DES PLAINES, ILL
After:     #1      [DES PLAINES, ILL^DEF^GHI^JKL^MNO^

Before:    #1      [ABC^DEF^
                      ^

Command:   IP?#5
Input:     NEXT
After:     #1      [ABC^DEF^^^NEXT^
                             ^
```

Since the output buffer pointer always points to the end of the buffer, this input has caused the pointer to move.

```
Before:    &1.Ø    [ABC^DEF^
Command:   IP&1.5
Input:     (new-line, no input)
After:     &1.Ø    [ABC^DEF^^^^^
```

In this case, even where no input was made, the buffer was still expanded to the point specified in the input command.

8.2 IN command

-Format- INprompt-character

The IN command is included only for the purpose of compability with previous versions of the processor. Capability to perform all functions performed by this command have been provided by other operations. See previous issues of documentation of the PROC processor for operation of this command.

8.3 IT command

-Format- ITоptions

The IT command is used to input data from a magnetic tape to the input buffer. A full block of data, up to a maximum length of 500 bytes, is read from the tape and placed into attribute 1 of the primary input buffer. The input buffer stack pointer is turned off and the primary input buffer pointer is redirected to point at attribute 1. No other attributes within the input buffer will be affected. One exception to the previous rule is if attribute marks (X'FE') or segment marks (X'FF') appear within the block of data being loaded. If they do appear, attribute marks or segment marks will respectively cause extra attributes or end of buffer to be created, either moving or deleting other items within the buffer.

Two options are provided to allow loading of tapes which have other than 7-bit ASCII codes on them. The first is the ITA command which will mask out the first bit of all 8-bit ASCII characters to present 7-bit ASCII characters. The second is the ITC command which will convert 8-bit EBCDIC to 7-bit ASCII characters.

8.4 EI Command

-Format- EIRPp(x,y),object-item,edit

The EI statement will input a value from the terminal, check it for compliance with the editing required, and, if acceptable, place it into an internal buffer. The EI command incorporates most functions required to input and edit a value that could be included in a single statement. Operation of the EI command is substantially faster than the equivalent statement string coded otherwise, and considerably more efficient in its use of system resources.

The EI statement consists of five separate parts designating how the command will operate and what it will do. The sections must appear in the order shown in the format and will perform as outlined below:

R (optional) designates a required field. If entered in the EI statement, an entry must be made in this field if it was blank originally.

Pp (optional) designates a prompt character. If not designated the default prompt character will be used. If designated, the character 'p' (which may be any character) will be used to prompt input and will become the prompt character. An N is a special prompt character that will cause no prompt character to be

output.

(x,y) (required) designates a column and row for input on the terminal screen. Both parameters must be supplied and either or both may be any valid direct or indirect reference. The location specified will be the first input character, and any prompt will appear in the position to the left of this location. Specifying column Ø will cause the prompt character to appear in the last location of the previous row.

Object-item (required) designates the field to be changed. This field must be a valid direct or indirect reference to any internal buffer.

Edit (required) designates the input and display editing to be applied to the field. Previous contents of a field are not checked for validity under the edit specifications. Valid edit specifications are listed in the table below.

Xnn an alpha-numeric field with maximum length specified by the numeric literal nn

Nn.m an unsigned numeric field of length n (including decimal) containing m decimal digits. Decimal digits are optional and this format may appear as Nn

Sn.m a signed numeric field of length n (including decimal and sign) with m decimal digits. Decimal digits are optional and this format may appear as Sn

D8 a date field

Specifications for numeric fields are different from those used elsewhere in RPL. The major difference is that the length n includes not only the numeric characters to the left of the decimal, but all other characters in the field as well. The signed numeric specification for -12.34 would then be S6.2. This would allow no more than two places to the left of the decimal, even if the negative sign was not entered.

The command will function so as to replace the normal sequence of statements used when loading or updating an item on a file. Initially the instruction will move the cursor to the position (x,y) on the terminal screen and output a mask depending on the edit specifications. If the edit requires alpha-numeric input (Xnn), a string of X's equal to the length will be displayed on the terminal screen. For example, the edit specifications below will result in the masks shown in the table:

Edit Specification	Mask Displayed
X23	XXXXXXXXXXXXXXXXXXXXXXX
X2	XX
X1Ø	XXXXXXXXXX

If the field is numeric, no mask will be displayed unless the original field was null, then a zero field, right justified, will be output. The mask displayed for all date format input will be MM/DD/YY.

The next step in the command will be to display the existing contents of the field, if any, starting at (x,y). The display will be edited to external format if numeric or date editing is specified. The old contents of the object field will not be checked for validity under the specified edit format. This may cause the display values to be other than expected. Dates will be displayed in mm/dd/yy format and numeric fields will be displayed in nnnn.nn with actual display depending on the edit specifications. The following table shows the total display, including masks, of typical items with various edit specifications:

EDIT SPECIFICATION	FIELD CONTENTS	DISPLAY
X15	ABCDE	ABCDEXXXXXXXXXX
X4	ABCDEF	ABCDEF
X5	(null)	XXXXX
N7.2	45	Ø.45
S5	271	271
S5.1	ABC	AB.C
S7.2	(null)	Ø.ØØ
D8	3311	Ø3/31/77
D8	(null)	MM/DD/YY

Next, a prompt character will be displayed unless a null prompt was specified. The prompt character will appear to the left of the position specified in the command for input. If no prompt specification appears in the EI command, the default prompt will be used. If a prompt is specified, it will become the default prompt character.

The processor will then wait for input at the first position of the input field. Certain special characters entered at this point will have special meaning, see below:

Null will not change the old field contents. Will not be accepted if the old contents was null and the command specifies this field is required. In this case, the error message "FIELD REQUIRED" will be displayed and the command will re-prompt the field for input.

← Null out the old field. The old contents of the field will be cleared. This input will not be accepted for a

required field.

* Transfer to previously defined abort location (see E* command specification). If no transfer specified, the * will be accepted as a normal character input which must pass the edit.

^ or END transfer to previously defined post location (see E^ command specification). If no transfer specified, the ^ will be accepted as a normal character input which must pass the edit.

Other entry accepted as characters. Certain special characters, such as value mark, attribute mark, etc, will not be accepted and will disappear from any input string.

After input any error message is cleared and the input value is checked for validity against the edit specifications. If the edit is alpha-numeric, any characters are valid and the length test is all that is required. If the length of the entered field exceeds the length specified in the edit, an error message "FIELD EXCEEDS MAX LENGTH" will be displayed and input will be requeued.

Numeric fields require a more complex test. First, two types of input are valid for most fields. Either a number with the specified quantity of decimal places or a number without a decimal is acceptable. The input processor will convert a number entered without a decimal as if it had been entered with the number of places specified in the edit rule. All other characters except for the optional minus sign must be numeric to pass the edit test. A minus sign will only be accepted if the Sn.m edit is specified.

If an input fails the numeric input test for any reason, the error message "FIELD MUST BE NUMERIC IN FORMAT ..." will be displayed. The actual format shown at the end of this message will be different from the edit specification in the command. The displayed format will be 'SaN.bN'. The S will be displayed only if signed editing is specified, and the .bN only if .m is in the edit specification. In this case b and m will be the same value unless m was specified as zero, then .bN will not be displayed. The aN parameter will be the quantity of digits allowed to the left of the decimal point. The following table will illustrate the relationship between edit specifications, input, and results in the object field or any error message displayed.

EDIT	ORIGINAL	DISPLAY	INPUT	RESULT/ERROR
S5.2	1Ø	' Ø.1Ø'	123	123
S5.2	1Ø	' Ø.1Ø'	-1.35	-135
S5.2	ABC	' A.BC'	null	ABC
S5.2	-38	' -.38'	ABC	... FORMAT S1N.2N
N8.2	14583	' 145.83'	-123	... FORMAT 5N.2N
S8.2	14583	' 145.83'	-123	-123

If the edit requires date formating, any of four inputs are acceptable. The input may be in the form MM/DD/YY or MMDDYY. However, if the first digit of the month is zero, the input may omit it and input as M/DD/YY or MDDYY. Input is checked for numerics in the proper locations and for roughly valid dates. That is, the month is checked for a value between 1 and 12, and the day is checked for a value between 1 and 31. No check is made for 31 Feb, etc. If the date input is invalid, the error message "FIELD MUST BE DATE FORMAT" will be displayed and the input will be requeued.

A total of four different messages may be displayed in case of invalid input. All will be displayed at screen location Ø,23 and will be cleared automatically immediately after the next input is made. Each output of an error message will be preceeded by an audible signal from the terminal, and, when cleared, the first 6Ø locations of line 23 will be spaced over. The four error messages output by the processor for this command are listed below:

```
FIELD REQUIRED
FIELD MUST BE DATE FORMAT
FIELD MUST BE NUMERIC IN FORMAT SaN.bN
FIELD EXCEEDS MAX LENGTH
```

After an acceptable input has been entered it will be converted to internal format (if numeric or date edit), stored in the object field location, and the processor will proceed with the next line of the program. At this time, it may be possible that further edits other than those provided in the EI command are required. For example, a payment on an invoice may not exceed the invoice amount. In any such case, a facility has been provided to allow the processor to put out any error message and clear it after the next input has been received. This command is coded as follows:

```
EEerror message
```

The command coded as above will output the literal error message following the EE command and flag the processor so any further input through the EI command will clear the message automatically. A sample program sequence using this function is shown below:

```
350 EIRP?(30,10),%7,N7.2
    IFN %7 [ %5 GO 360
    EEPAYMENT GREATER THAN AMOUNT DUE
    GO 350
360 ...
```

The EI command will therefore perform most functions required in normal screen input. For example, the following command:

```
EIRP?(5,14),&3.7,X15
```

performs all functions shown in the following sequence of instructions:

```
    MV %3 "","",""
100 T (5,14),"XXXXXXXXXXXXXXX",(5,14),&3.7,(4,14)
    IPB?%3
    IF %4 T (0,23),S60
    IF %3 = -]* GO 110]GO nn
    IF # %3 IF &3.7 GO 140
    IF %3 GO 120
110 T (0,23),"FIELD REQUIRED"
    MV %4 "E"
    GO 100
120 S5
    U11A4
    %3
    IFN %5 [ 15 GO 130
    T (0,23),"FIELD EXCEEDS MAX LENGTH"
    MV %4 "E"
    T (5,14),S%5
    GO 100
130 MV &3.7 %3
140 T (5,14),S15,(5,14),&3.7
```

While the above example shows use of three input buffer fields as scratch locations, the EI instruction uses no fields other than the one to be updated.

8.5 E* Command

-Format- E*]stub-statement

The E* command is a utility function of the EI command to designate what action to take if an asterisk is entered in response to an EI command input. This statement should be executed before any EI commands which must utilize the * exit function. Then when an asterisk is input into a field in an EI command, control is passed to the stub statement of the E* command. Only one stub statement will be active at any one time, the last one executed. If it is required that this function be deleted, an E*K command may be executed. From

that point on, all asterisks entered into a terminal for an EI command will be considered only a character without any special significance.

Function of an E* command will be cancelled automatically if an inter program tranfer is accomplished, and will not be restored even if the program is reentered via a RTN statement.

8.6 E^ Command

-Format- E^]stub-statement

The E^ command is a utility function of the EI command to designate what action to take if an up arrow is entered in response to an EI command input. This statement should be executed before any EI commands which must utilize the ^ exit function. Then when an up arrow is input into a field in an EI command, control is passed to the stub statement of the E^ command. Only one stub statement will be active at any one time, the last one executed. If the stub statement is not a GO, the next statement in line will be executed after the stub. If it is required that this function be deleted, an E^K command may be executed. From that point on, all up arrows entered into a terminal for an EI command will be considered only a character without any special significance.

Function of an E^ command will be cancelled automatically if an inter program tranfer is accomplished, and will not be restored even if the program is reentered via a RTN statement.

CHAPTER 9

JUMPS AND BRANCHES

Jumps within or between PROCS may be accomplished by several commands available to the programmer. The jumps may be made unconditionally, or may be coded into the stubs of an IF statement and executed conditional upon a comparison being true. Additionally it is possible to perform subroutine jumps which will store a location to be returned to later so that a section of code which must be executed several times within a PROC may be coded only once and accessed from several places during execution. The commands which make these functions possible are outlined in this chapter.

Two levels of subroutine return stacks are provided which will separate the local intra-program subroutines from the inter-program subroutines.

9.1 Intra-Program Jumps and Branches

Local jumps and branches are performed without regard to the relationship to other programs which may be called as subroutines or which may call the current program. Any local jump or branch which is attempted will only transfer to locations within the same program or will cause a program execution abort. Each individual program will create its own subroutine return stack which is not available to other programs. Intra-program jumps and branches will not transfer control to another program in any operation.

9.1.1 LABELS

Numeric labels may be provided in the program to allow jumping or branching operations to other locations in the program. All locations which must be accessed from within the program must have a numeric label as the first item of the line. These labels may be any integer numeric value up to 2147483647 and must be followed on the line by a blank before the normal statement starts. Leading zeros do not change a label so label Ø21 is interchangeable with 21. Any number of labels may be used in a program.

Any label may contain any statement following it on the same line as long as a space follows the label. The following is a series of examples of statements with labels on them:

```
135 F;&3.2;&1.5;-;?%33
Ø3115 C  SUBROUTINE TO CALCULATE NEXT SAVE VALUE
1 T (2,1),"ENTER COMPANY NAME",S25,(2Ø)
ØØØ23 IF %25 = END GO 3825
1Ø L HDR,(3),&3.15,(5Ø),"MONTHLY STATUS REPORT"
```

It is generally recommended that label numbers be assigned in ascending numerical order from the front of the program for ease of reading, though this is not a requirement of the processor. Duplicate labels may exist in an unoptimized program but only the first one encountered from the front of the program will be used (see GO statement explanation). Duplicate labels are considered a fatal error by the optimizer.

9.1.2 MARK statement

-Format- MARK

A MARK is an alternate label form which is used as an access point for a jump. A MARK statement must appear on a line by itself or may be executed conditionally if coded in the stub portion of an IF statement. When executed, the mark command stores the current instruction counter for later reference. Only the last mark executed is remembered. See the GO command explanation for actual operation of the MARK function.

9.1.3 GO command

-Format- GO label

The GO command is used for a jump to another location in a PROC. The location to which the execution will proceed is specified by the operand in the GO command which may be a label number, a FORWARD or a BACK. The functions performed by each of these forms is dissimilar and will be discussed separately below.

GO (label number) The execution will proceed to the first occurance of the label in the PROC. This is accomplished by scanning the program from the front to the back until a statement is found starting with a label number equal to the one appended to the GO statement. Execution then continues with that statement. If that label does not exist in the PROC, the program execution will terminate with an error statement. GO statements may be coded as shown below:

```
GO 381335
GO ØØ135
```

GO FORWARD The execution will proceed to the next MARK statement occuring as the first item of a statement.

This is accomplished by scanning the program from the next statement until a MARK statement is found. Execution then continues with the following statement. If there is no further MARK in the program, an error will occur during execution.

GO BACK The execution will proceed to the last MARK actually executed by the PROC. This may or may not be the last previous MARK physically occuring in the program. If no previous MARK has been executed in the program, an error will occur during execution of the GO statement.

9.1.4 GOSUB command

-Format- GOSUB label

The GOSUB statement is used as a branch to a subroutine from which a return is desired. Execution of the GOSUB command will result in transfer of control to the label number specified in the command and the creation of a return location on the top of the local subroutine return stack. There is no limit to the number of return locations which may be placed on the return stack.

9.1.5 RSUB command

-Format- RSUB n

The RSUB command transfers program execution to the location specified in the top entry on the subroutine return stack and that entry is deleted from the stack making the next entry avaliable for a subsequent RSUB. If there are no return locations on the subroutine return stack, the RSUB will not function but will fall through to the next statement in the program.

An alternate form of the command is to append a space and a numeric value to the statement as 'RSUB n'. This will transfer execution to the n'th statement after the GOSUB which created the return stack address.

For example, if the following series of statements appear in a program:

```
GOSUB 1Ø0
GO 3Ø0
MV &3.Ø &7.1,&7.2
T (Ø,5),"ADDRESS  "
```

the following return statements executed from the subroutine at label 1Ø0 will return to the statements shown:

Return	Next command executed
RSUB	GO 300
RSUB 1	GO 300
RSUB 2	MV &3.0 &7.1,&7.2
RSUB 3	T (0,5),"ADDRESS "

9.1.6 KSUB command

-Format- KSUB n

The KSUB command will delete a variable number of return locations from the local subroutine return stack. When used by itself the KSUB command will delete the entire return stack listing. If used with an optional numeric value preceeded by a space as 'KSUB n', it will delete the number of return locations specified by the numeric value. If fewer locations appear on the stack than the number required to delete, the entire stack will be deleted. For example:

Command	Returns Deleted From Stack
KSUB	All
KSUB 1	One
KSUB 2	Two
KSUB 15	Fifteen

9.2 Inter-Program Transfers

The RPL language allows two or more programs to be logically linked together. This function serves an important purpose in that it releases the programmer from the restriction of a maximum of 32,267 bytes in a program. Two different operations are provided to allow the secondary program to be called as a returnable subroutine or simply used as an extension of the primary program. Use of the subroutine capability to branch to another program and return should be minimized since it takes much longer to execute than an internal subroutine branch and return.

All data elements will remain intact during an inter-program transfer and will be passed to the called program with the exception of the local subroutine return stack and MARK save counters. Any data element may be modified and passed on to another program or back to the calling program.

9.2.1 ( ) Inter-program jump

-Format- (file-name item-name)

The inter-program jump will kill the entire local subroutine stack and transfer control to the first command of the designated program. If the item designated is not a program or is not on file, program execution will terminate with an error message. Valid examples of the transfer command are presented below:

```
(USER PRGM1)
(M/DICT IN3A)
(DICT VENDOR UPDATE)
```

If the file to which the transfer is to be made is a dictionary only file the form (dict file-name item-name) is much more efficient than the form (file-name item-name), though both are acceptable.

9.2.2 [ ] Inter-program branch

-Format- [file-name item-name]

The inter-program branch will stack a return location to the next instruction and transfer control to the first command of the designated program. Additionally, any local subroutine returns will be saved in a hold area and the local subroutine return stack will be cleared. If the item designated is not a program or is not on file, program execution will terminate with an error message. Some valid forms of the inter-program branch are:

```
[M/DICT PROGRAM-3]
[MASTER SCREEN]
[DICT PROGRAMS ENTRY]
```

If the file is a dictionary only file, the form [DICT file-name item-name] is much more efficient than the form [file-name item-name], though both are acceptable.

9.2.3 RTN command

-Format- RTN n

The RTN command will cause an inter-program return to the next instruction after the branch command which created the stack location. Optionally a numeric value n may be loaded after the RTN (as RTN n), and, if used, will cause return to the n'th line following the branch. The local subroutine stack will be cleared and the local stack for the program to which control will be passed is restored. If there is no

return location on the inter-program return stack, the program will be exited and control will be passed to TCL.

For example, if the command sequence in the calling program is:

```
[MASTER PRGM1]
GO 3855
MV &2.Ø &7.2
IF &3.2 # END GO 1Ø
```

then the following table will illustrate the returns for the RTN command given in PRGM1.

Command	Next Command Executed
RTN	GO 3855
RTN 1	GO 3855
RTN 2	MV &2.Ø &7.2
RTN 3	IF &3.2 # END GO 1Ø

CHAPTER 10

CONDITIONAL OPERATIONS

The conditional execution commands provided for the PROC processor are extremely powerful. The coding of the commands is simple and interpretation is easy. The number of combinations of operations which may be performed with one statement is very large. The operations include two types of commands with up to seven different formats and six different operators that may be used in each. Since full explanation of all possible cases would take a volume in itself, many variations are left to inference where one of such a type of operation has already been covered. Such extensions of logic are only used where the average programmer will have no trouble deducing the operation of a particular statement from one which is described in full.

The IF statement processor provided in the PROC language allows conditional operations to be performed depending on the outcome of an alphabetic ASCII or numeric integer comparison. The results of these statements may be conditional execution of any other PROC statements including GO or GOSUB operations. Compound IF statements may be constructed or conditional multivalued IFs which can become so complex that they will be hard to follow later. It is recommended that the multivalued IF capability be used sparingly so as not to unduly complicate the jobs of other programmers who will have to review or correct the programs later.

10.1 IF command

-Format- IF operand1 operator operand2 stub-statement

The IF command is used to perform an ASCII comparison of two operands according to an equality operator supplied and to execute an appended stub statement if the comparison is true. The first of the two operands may be any direct or indirect reference of the form A, !N, %N, #N, or &M.N. The second of the operands may be any direct or indirect reference of the form !N, %N, #N, &M.N or a literal string. The value of the first operand is compared to the value of the second operand, and, if they compare as specified by the operator, the stub will be executed. If they do not compare as specified, the next statement in line in the program will be executed. The six operators are described below:

Operator	Statement TRUE if:
=	Operand 1 equals Operand 2
#	Operand 1 not equal to Operand 2
>	Operand 1 greater than Operand 2
<	Operand 1 less than Operand 2
]	Operand 1 greater than or equal Operand 2
[	Operand 1 less than or equal to Operand 2

The shorter of the two operands is logically extended to the same length as the longer by appending Hex ØØ's on the right. The operands are then compared left to right until the first unequal character is encountered or until the end of the values is reached. The condition is then set 'GREATER THAN', 'LESS THAN' or 'EQUAL' depending on the conclusion reached and this condition is then compared to the operator. If the condition satisfies the operator, the result of the IF statement is TRUE and the stub is executed; if not, the result is FALSE and the stub is skipped and the next statement is executed.

The following table of examples will illustrate the results of the IF statement comparison:

OPR1	OPR2	OPERATOR					
		=	#	>	<	]	[
ABC	ABC	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
ABCD	ABC	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
123	ABC	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
123	Ø123	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
3	138	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE

The last two cases show that numeric values are not considered since this is only an ASCII comparison. Two numerically equal values are unequal due to the presence of a leading zero on one, and the smaller of two numeric values shows greater than the other since it is compared with the most significant digit of the other instead of being aligned numerically first. Such results should be expected in an ASCII compare.

Valid forms of the basic IF statement are shown below with their stubs.

```
IF &3.1 = ABC GO 78Ø
IF %#3 > AAA MV &1.2 %#3
IF #5 # #1 L (3),&1.5
```

The pound sign (#) will not be confused when used as a 'NOT EQUAL' or as an output buffer reference if all items in the statement are properly separated by spaces as shown in the third example above.

Attributes may be compared to a null value by using "" in the second operand in place of a literal string. For example, the following statements will each compare the first operand to a null value:

```
IF &3.1 = "" GO 78Ø
IF %#3 = "" MV &1.2 #3
IF #5 = "" T (Ø,23),"NOT ON FILE",B
IF %15 # "" XPROGRAM FINISHED
```

As previously stated, an A function can be used to provide the value for the first operand. This means that any valid form of an A statement may appear excepting those using the surround character. Valid forms are A, An, A(n), A(,m), or A(n,m). When used in this context in the IF statement, no value is moved to the output buffer (as when the A command is used alone). The format supplied is merely used to extract the value to be compared from the input buffers and the buffer pointers are not moved as a result of the reference. For example:

```
IF A = END GO 999
IF A(3,2) > 99 A(3,2)
IF A2 [ 99 MV %2 &1.5
IF A = A IHNEXT
```

In the last statement the A in the second operand is interpreted as a literal value since the A for input buffer extraction is not valid in the second operand.

The two special cases of the basic IF statement exist which will compare an attribute to a null value not specified in the command. The first of these is coded without an operator and only one operand, which may be any direct or indirect reference. In this case, the statement will be TRUE if the referenced attribute is anything other than a null. These statements may be coded as follows:

```
IF A GO 385
IF &3.%2 IHC
```

These commands are equivalent to a 'NOT EQUAL' comparison to a null literal as 'IF A # "" ...'.

The second special case is coded with a 'NOT EQUAL' operator and a second operand only, which in this case may be an A statement to extract items from the input buffer. The statement will be TRUE if the referenced value is null. For example:

```
IF # A GO 385
IF # &3.%2 IHC
```

The last commands are equivalent to a 'EQUAL' comparison to

a null literal as 'IF A = "" ...'.

If the second operand in any IF statement is to be a literal which must contain either a single or double quote (' or "), it may be surrounded by the other quote item as a delimiter. For example:

```
IF %3 = 'AB"C' ...
IF &3.%4 = "PROGRAMMER'S" ...
```

10.1.1 IF statement with mask

The second form of the IF statement will compare the value of an attribute in the first attribute to a mask in the second operand. Masks may specify character type and length or specific non-numeric characters which must appear in a given position. Numeric characters may not be specifically defined since they signal the start of a mask length. Masks must be surrounded by parentheses in the second operand of the IF statement and may contain any implicit characters in any position by having that character coded into the mask. For example:

```
IF %3 = (ABC) ...
```

In order for the statement to compare TRUE, attribute 3 of the primary input buffer must contain an A in the first position, a B in the second position, a C in the third position, and no other characters in the attribute. Of course in this simplified version the statement is equivalent to:

```
IF %3 = ABC ...
```

The unique capability of the mask statement occurs when a numeric in the mask is followed by an A, an N, or an X. In this case the combination of numeric and characters A, N, or X specifies that a given number of Alphabetic, Numeric, or Alphameric characters must occur in that position. A numeric literal may not otherwise appear in a mask. For example, the statement

```
IF %3 = (ABC3A) ...
```

will be TRUE only if the first three positions of %3 are ABC and are followed immediately by exactly three other alphabetic characters. The results of the test on several values is shown below:

Condition	Values			
TRUE	ABCDEF	ABCXRQ	ABCAAA	ABCUZB
FALSE	ABC123	ABCAA	ABCDEFG	UBCABC

The function of this statement, therefore, could not be duplicated by any less than 17,576 individual IF statements specifying ABCAAA through ABCZZZ as the second operand. Additional valid masks which may be coded into a PROC include:

```
IF &3.2 = (AB-4N) ...
IF #31 # (4N) ...
IF %5 = (4A-2A) ...
IF %7 = (3N-2N-4N) ...
```

The last statement could be used to test for valid input of a social security number.

A numeric zero preceeding a type character will allow any quantity of the character type specified to be accepted and break when a different character is encountered. This function has particular significance in cases such as numeric input with a decimal point as in:

```
IF %3 = (ØN.2N) ...
```

This comparison will test TRUE if %3 has any quantity of numerics, a decimal point, and two numerics. Values such as 123.45 or 175137.15 or .89 will satisfy this test.

Care must be exercised in the use of the ØX type comparison in a mask. Any further specifications within the mask after the ØX will cause the compare to always be FALSE. This will occur since the ØX comparison will use up the entire remaining source field regardless of the contents and leave nothing for the following portion of the mask to compare against. The following examples will, therefore, always compare FALSE, regardless of the contents of Operand 1.

```
IF A = (ØX.2N) ...
IF %3 = (ABC2N-ØX-M) ...
```

A special case occurs when numerics are specified in the mask since optional plus or minus signs may be accepted even when not coded in the statement. When numerics are specified in the mask without a sign immediately preceeding the numeric mask, the statement will compare TRUE if the specified number of numerics appear alone or if they are preceeded by a plus or minus sign. If the mask specifies a plus sign and numerics, the numeric value in the argument may appear alone or may be preceeded by a plus sign. If the mask is a minus sign and numerics, the argument must be preceeded by a minus sign. In all of these cases, the sign will not be counted in the length of numeric characters specified. The following table will show the results of numeric mask comparisons in the statement:

```
IF A = (mask) ...
```

Value	Mask		
	(4N)	(+4N)	(-4N)
1234	TRUE	TRUE	FALSE
+1234	TRUE	TRUE	FALSE
-1234	TRUE	FALSE	TRUE
-123	FALSE	FALSE	FALSE
A1234	FALSE	FALSE	FALSE
1234A	FALSE	FALSE	FALSE

Use of a mask with other than an EQUAL or NOT EQUAL operator is more complex than usual. The comparison is made initially only on the type of characters specified and the result is compared to the operator to determine if the statement is TRUE or FALSE. A null field will compare LESS THAN any mask value. For example, with the following statement:

```
IF %3 > (3N) ...
```

the contents of %3 must contain three numeric characters (preceeded by an optional sign) and at least one other character of any type for the statement to compare TRUE. The following table will illustrate the function of the general statement:

```
IF %3 operator (mask) ...
```

%3	Operator (mask)				
	> (2N)	< (2N)	[(2N)	> (2A)	< (2A)
+12	FALSE	FALSE	TRUE	FALSE	TRUE
123	TRUE	FALSE	FALSE	FALSE	TRUE
-1BC	FALSE	TRUE	TRUE	FALSE	TRUE
ABC	FALSE	TRUE	TRUE	TRUE	FALSE
AB12	FALSE	TRUE	TRUE	TRUE	FALSE
A12	FALSE	TRUE	TRUE	FALSE	TRUE
12B	TRUE	FALSE	FALSE	FALSE	TRUE
AA	FALSE	TRUE	TRUE	FALSE	FALSE
(null)	FALSE	TRUE	TRUE	FALSE	TRUE

10.1.2 Multivalued IF comparisons

The second operand of an IF statement may be a literal multivalued field or a direct or indirect reference to an attribute in a buffer which is a multivalued field. In this case, a comparison is made between the first operand and each value of the second operand. The statement is TRUE if any of the comparisons are TRUE, except in a NOT EQUAL comparison. A NOT EQUAL comparison will only execute the stub if all values in the second operand compare not equal to the first operand.

The multivalued second operand may contain anything in a value which is valid as a second operand by itself; i.e., literals, masks, direct or indirect references, etc. The examples below are valid:

```
IF %3 = &7.1]""](4N)]ABC GO 385
IF &7.%2 > AA]&5.1]&7.22](2A) X OFF
IF !1 [ &1.2]&7.15]AAA]9999 MV &5.2 &7.25,&5.8
```

The comparison of multivalued fields will stop when a space not within quotes is encountered. The remainder of the statement will be considered the stub by the processor. If the contents of &7.1 were multivalued in the first example above, the first operand would be compared to each value of the field before proceeding to the next value in the second operand. If contents of the second operand were contained in an attribute within a buffer and a reference was made within the statement to that attribute, it would execute the same as above. The next statement is equivalent to the first example above:

```
Command:    IF %3 = &1.2 GO 385
Where:      &1.Ø    [ABC^DEF^&7.1]""](4N)]ABC^123^
```

1Ø.1.3 Multivalue stubs

If the second operand of the IF statement is a multivalued field, the stub may contain a series of multivalued statements. When this case occurs, the first stub statement will be executed when the first value of the second operand compares TRUE; the second stub statement when the second value compares TRUE; etc. Each value in the stub must be a complete statement by itself without any value marks in it. If more than one value in the first operand compares TRUE, the stub associated with the first one will be executed. If the values in the second operand are exhausted before the last stub statement, the remaining stub statements will be skipped and the next statement in the program will be executed. If the stub statements are exhausted before the values in the second operand, the last stub statement will be executed if any of the remaining values compare TRUE.

The following table will illustrate the operation of this statement:

```
IF %3 = ABC]""](2N) GO 11Ø]IHC]MV &1.2 %3
(next statement)
```

%3	Next execution
ABC	GO 11Ø
(null)	IHC (then next statement)
43	MV &1.2 %3 (then next statement)
PB	(next statement)

If a direct or indirect reference within a multivalued second operand contains a multivalued field, the same stub will be executed if any of the values compare TRUE. Each value within the referenced field will not access a different stub statement as they would if they were coded into the second operand.

1Ø.1.4 Compound IF statements

An IF statement may appear in the stub of another IF statement. In this case, the final stub will be executed if both IF statements compare TRUE. Any quantity of IF statements may be appended together in this manner and the final stub of the last IF will only be executed if all comparisons are TRUE. For example:

```
IF %3 IF &3.2 = 1 IF %4 > (2N) GO 748
```

The stub will be executed if %3 is not null, and &3.2 is a literal 1, and %4 starts with two numerics and has at least one other character.

Should one of the IF statements in the compound structure be a multivalued IF, any TRUE comparison will satisfy that statement and allow the stub to be executed. Only the last of the IF statement in a compound IF statement may contain a multivalued stub in order for it to operate as if it were coded alone.

1Ø.2 IFN statement

-Format- IFN operand1 operator operand2 stub-statement

The IFN statement will algebraically compare numeric values within the two operands according to an equality operator and execute an appended stub statement if the comparison is TRUE. The first of the two operands may be any direct or indirect reference of the form A, !N, %N, #N, or &M.N. (The form A(M,N) will only work if the remainder of the field is specified.) the second operand may be any direct or indirect reference of the form !N, %N, #N, &M.N, or a literal string. The value of the first operand is compared to the value of the second operand, and, if they compare as specified by the operator, the statement is TRUE, and the stub statement is executed. If they do not compare as specified, the statement

is FALSE, the stub is skipped, and the next statement in the program is executed. The six operators that may be used are:

Operator	Statement TRUE if:
=	Operand 1 equals Operand 2
#	Operand 1 not equal to Operand 2
>	Operand 1 greater than Operand 2
<	Operand 1 less than Operand 2
]	Operand 1 greater than or equal Operand 2
[	Operand 1 less than or equal to Operand 2

The value of each operand up to the first non-numeric character is converted to internal binary integer format, and the first is compared algebraically to the second. The result of the comparison will show operand 1 greater than, less than, or equal to the second operand. If the resultant of the comparison satisfies the operator supplied, then the statement is TRUE and the stub statement is executed; else the statement is FALSE, the stub is skipped, and the next statement in the program is executed.

The following table of examples will illustrate the functioning of the IFN statement:

OPR1	OPR2	OPERATOR					
		=	#	>	<	]	[
123	34	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
123	Ø123	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
123	-123	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
123	+123	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
123	12.3	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
123	123A	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
123	ABC	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
123	(NULL)	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE

The last three cases above show that each operand is considered ended when a non-numeric character is encountered (one plus or minus sign preceeding any numerics is allowed). In case no numerics appear before a non-numeric, or for a null field, the value of zero is used for that operand. A special case of an acceptable non-numeric is the decimal point; it will be skipped, as shown in example five above, and the remaining value compared as if it were not present.

The IFN statement may use multivalued fields in the second operand and multivalued stub statements. These functions will operate the same as in the IF statement allowing for the numeric comparisons outlined above. Refer to the corresponding sections in the IF statement for coding and operation of these functions.

CHAPTER 11

DISK FILE I/O

The RPL language is provided with a full complement of disk file input/output instructions for standard REALITY files. Input/output from a disk file is accomplished by using one of nine file buffers provided specifically for this function. The programmer is therefore limited to access from a maximum of nine files at one time. File buffers not required for file input/output may be used for working storage.

Records which are read from or written to a file will use attribute zero of the file buffers as a key, and the first data attribute of the record will be in attribute one.

11.1 F-OPEN command

-Format- F-OPEN file-nbr file-name

The F-OPEN command will open a file buffer to a designated file and allow read, write, or delete operations to be performed with records in the file. Files specified in the F-OPEN command may be data files, dictionaries, or the account Master Dictionary (M/DICT). If an open is attempted on an item which is not a file, an error message will result and program execution will be terminated. File buffers will remain open until an exit is made to TCL or until a particular buffer is opened to another file. An open command does not change the content of any of the buffers.

Examples of valid commands of this type are:

```
F-OPEN 1 M/DICT
F-OPEN 7 VENDOR
F-OPEN 4 DICT CUSTOMER
```

F-OPEN commands may also be executed using the contents of any attribute of an internal buffer to provide the name of the file. For example:

```
F-OPEN 1 %3
F-OPEN 7 &4.%2
F-OPEN 4 &4.2
F-OPEN 4 DICT &4.3
```

```
Where:    %1      [ABC^1^M/DICT^DEF^
          &4.0    [ABC^VENDOR^DICT CUSTOMER^CUSTOMER^
```

The examples above are all equivalent to the previous illustrations which named the files in the statements.

11.2 F-INPUT command

-Format- F-INPUT file-nbr file-name

The F-INPUT command operates identically to the F-OPEN command in all respects except it will allow only read operations to take place on the file. Writes or deletes are illegal from a file buffer that has been opened with the F-INPUT command and will cause program termination with an error message if attempted.

11.3 F-READ command

-Format- F-READ file-nbr item-name

The F-READ command will read a record from a file and place it in the file buffer specified. The file from which the record will be read is the one to which the buffer has been opened by an F-OPEN or F-INPUT command. If the buffer has not been opened to a file an error message will occur. The record which will be read may be specified by a literal in the command or may be a direct or indirect reference to an attribute in a buffer or select register containing the record key. The file buffer will be cleared and an attempt will be made to read the record specified. If the record is found, it is placed in the file buffer and program execution will proceed with the second instruction following the read. If the record is not found, the next instruction after the read will be executed. An attempt to read a record with a null key will result in an unsuccessful read and the return will be to the following statement. No previous attributes in the buffer will remain in any case. For example:

```
F-READ 1 1Ø35-Ø1
GO 25
MV &3.1 &1.Ø
```

If the item 1Ø35-Ø1 is found on file, it will be placed in the file buffer and the MV instruction will be executed next. If the item is not found, the buffer will be cleared and the GO instruction will be executed.

If the read is successful and the buffer to which the item was read was opened with an F-OPEN command, a record lock will be placed on the item within the file. This will prohibit another buffer opened with an F-OPEN command in this, or any other program, from reading the same item. If an attempt is made to read an item which has a record lock from an F-OPEN'ed file, the program will stall and an audible signal will be heard at the terminal at about 1Ø second intervals. When the record is freed by the process which locked it, it will be read by the waiting program and execution will proceed normally.

A file which has been opened with an F-INPUT command will be able to read any record, even if it has been locked by another process. A record read from a file opened with an F-INPUT command will not have a record lock place on it, thereby leaving it available to all other processes.

11.4 F-WRITE command

-Format- F-WRITE file-nbr

The F-WRITE command will write a record from a file buffer to a disk file. The file to which the record will be written is the one to which the buffer was opened with an F-OPEN command. No record key need be named in the command since the key of the record written will be the value of attribute zero. Any record existing in the file with the same key will be deleted before writing the new record. If the buffer was not opened to a file or was opened with an F-INPUT command, an error message will be displayed and program execution will terminate. Writing a record to a file opened with an F-OPEN command will free any record locks which have been set for that file. The contents of the file buffer will not be changed by the F-WRITE command.

Valid examples of the command are:

```
F-WRITE 1
F-WRITE 8
F-WRITE 7
```

If an attempt is made to write a record with a null attribute for the key, no item will be written and program execution will continue normally.

11.5 F-DELETE command

-Format- F-DELETE file-nbr

The F-DELETE command will delete the record specified in attribute zero of the file buffer from the file to which the buffer has been opened by an F-OPEN command. If the buffer has not been opened to a file or has been opened with an F-INPUT command, an error message will occur and program execution will stop. If the item does not exist on the file, no change will occur. Execution of an F-DELETE command will release any record locks held by this file buffer. The contents of the file buffer will not be changed by execution of this command.

Examples of valid commands are:

```
F-DELETE 1
F-DELETE 3
F-DELETE 8
```

11.6 F-CLEAR command

-Format- F-CLEAR n

The F-CLEAR command will clear the entire contents of the file buffer designated by the numeric literal n. Execution of the command has no effect on file access of a file and any opened files need not be reopened after clearing the buffer.

11.7 F-FREE command

-Format- F-FREE

The F-FREE command will release all record locks held by all file buffers within the process. Since this is not a selective command, the programmer should only use it sparingly.

CHAPTER 12

ARITHMETIC CALCULATIONS

12.1 F; (function) command

-Format- F;operand1;operand2;operand3;...;?destination

The F; statement is used for performing arithmetic functions such as add, subtract, multiply, and divide. All functions assume fixed point integer arithmetic. Calculations are performed in a stack architecture with twenty-three entries in the stack. The calculations specified in the statement are performed using only the last two entries in the stack. When a calculation completes, it deletes the two entries used and places the results in the bottom of the stack. The stack architecture and the available arithmetic functions are explained below:

Stack Architecture
- Stack 1
- Stack 2
- Stack 3
- Stack 4
- Etc. Through Stack 23

The operations which may be performed by the function statement are described below. Each must be coded individually into the statement and separated by a semicolon from the surrounding operators. The last operator, and only the last operator, must be a '?' operation placing the results in a storage location. Results which must be placed in two locations or intermediate results which must be saved must be moved later or split into two calculations.

DIRECT OR INDIRECT REFERENCE if the item contains a numeric value, move item to stack 1 and bump stack 1 to stack 2, stack 2 to stack 3, etc. Stack 23 item will disappear. If the item contains a valid operator as described below, the operation designated by the contents will be performed as if that operator were coded into the statement. If the item does not contain either of the above, a numeric zero will be loaded into stack 1, stack 1 will be bumped down to stack 2, etc.

\+ ADD stack 1 to stack 2 and place results in stack 1. Bump stack 3 down to stack 2, stack 4 to stack 3, and so to stack 23. Loads an unknown value into stack 23.

- SUBTRACTS stack 1 from stack 2 and places results in stack 1. Bump stack 3 down to stack 2, etc.

* MULTIPLY stack 1 by stack 2 and place results in stack 1. Bump stack 3 down to stack 2, etc.

/ DIVIDE stack 2 by stack 1 and place quotient in stack 1. Any remainder will be dropped and stack 3 will be bumped down to stack 2, etc.

R REMAINDER, divide stack 2 by stack 1 and place the remainder in stack 1. Bump stack 3 down to stack 2, etc.

/H DIVIDE AND HALF-ROUND, divide stack 2 by stack 1 and half round the last digit if the remainder exceeds half the divisor. Place the results in stack 1 and bump stack 3 down to stack 2, etc

/R DIVIDE WITH QUOTIENT AND REMAINDER, divide stack 2 by stack 1 and place the remainder in stack 1 and the quotient in stack 2. The remaining items in the stack do not move.

↔ FLIP, exhanges the contents of stack 1 and stack 2. The remaining items of the stack remain unchanged.

? PLACE RESULTS, when coded as the last item in a function statement, places the contents of stack 1 into the location specified in the appended direct or indirect reference. If the destination is the input buffer, the input buffer stack pointer will be turned off and the primary input buffer pointer will be directed to the attribute loaded.

?P PLACE IN INPUT BUFFER, the item in stack one is placed into the primary or secondary input buffer at the location specified by the current input buffer pointer.

?Pn PLACE FIXED LENGTH RESULTS, the item in stack one is placed in the input buffer at the location specified by the current input buffer pointer. If the result is less than n digits it will be expanded with leading zeros to the length specified. If the result has more than the quantity of digits specified, it will be placed intact into the attribute.

CONSTANT constants may be loaded into stack 1 by coding the literal numeric value of the item to be loaded. Stack 1 will be bumped to stack 2, etc.

The F; (function) statement will allow any combination of add, subtract, multiply, and divide operations to be executed in sequence as long as the restrictions of the stack architecture are followed. For example:

```
F;&3.22;%3;7;#5;-;+;&3.21;/;9;R;?&2.%15
F;%1;%2;+;%3;%4;-;2;/;+;%3;%4;-;/;?%5
F;%1;%2;+;%3;%4;-;/H;?%5
```

The second and third examples above are equivalent with the function of the half round, assuming positive figures, being performed in example two. The automatic half rounding function executed in the third example is the same as the second example. If negative figures appear, the second example would have to test if a subtraction would be more appropriate, however, the third example would function correctly regardless of the data supplied.

The following series of examples will illustrate the operation of the F; statement in detail to show the actual functions performed as each item in the statement is executed.

For the following statement:

```
F;&3.%2;%5;%6;%7;%8;%9;-;/;*;R;?&7.2
```

the operations will proceed as follows:

operation	stack1	stack2	stack3	stack4	stack5
&3.%2	123	*	*	*	*
%5	73	123	*	*	*
%6	7	73	123	*	*
%7	58	7	73	123	*
%8	17	58	7	73	123
%9	5	17	58	7	73
-	12	58	7	73	123
/	4	7	73	123	*
*	28	73	123	*	*
R	17	123	*	*	*

The results of 27 now located in stack 1 will be loaded into &7.2. Note that in this example a value was left in the stack because it was not used or moved to stack 1. The asterisks appearing in the example designate an unknown value since any item which has not been established by the F; command may contain any unknown characters. Care should be exercised in using the F; function statement due to this fact.

12.2 +, - Commands

```
-Format-   +numeric-literal
-format-   -numeric-literal
```

The + (-) command will add (subtract) an appended numeric literal to the attribute of the current input buffer designated by the current input buffer pointer. The result of the add will be placed into the buffer starting at the first

character of the current attribute. If the result is a numeric value that requires fewer characters than the original, it will be expanded by leading zeros until it is the same length and then it will be placed in the buffer. If the results is larger than the original, it will overlay characters beyond the end of the current attribute until it is loaded. This action may cause the field to merge into the next field if the attribute mark is overlayed. If the original field is not numeric, it will be assumed as zero and the actual value added will replace the original contents following the replacement rules outlined above. Since this command does not expand or contract the input buffer when executed it will complete faster than the F; function command.

```
Before:    [ABC^123^456^789^
               ^

Command:   +28
After:     [ABC^151^456^789^
               ^


Before:    [ABC^93^27^
               ^

Command:   +15
After:     [ABC^10827^
               ^
```

As illustrated above, the last character of the results extended beyond the original field, so it had to overlay the attribute mark creating an erroneous results in %2, and also creating a problem when addressing %3, %4, etc.

```
Before:    [ABC^-2^15^
               ^

Command:   +5
After:     [ABC^03^15^
               ^
```

The use of a plus or minus sign in the original field will cause the results to become the algebraic results of the sum since the + or - will be processed correctly by the addition.

```
Before:    [ABC^122^15^
               ^

Command:   +-3
After:     [ABC^122^15^
               ^
```

Appearance of any non-numeric (such as the minus in this example) will cause the value to be interpreted as only the numerics appearing before the non-numerics, and the addition will proceed accordingly. In this case, since there were no numerics before the minus sign, the value of zero was added to the intitial value in the source attribute.

```
Before:    [ABC^123^\^456^
                 ^

Command:   +1
After:     [ABC^123^1^456^
                 ^


Before:    [ABC^123^
           ^

Command:   +1
After:     [Ø01^123^
           ^
```

Both of the examples above show operations with the starting value in the buffer attribute being non-numeric. As stated before, the original value is assumed zero and the command proceeds accordingly.

```
Before:    [ABC^Ø000^123^
                ^

Command:   +15
After:     [ABC^Ø015^123^
                ^
```

If the result of the addition is too small for the original field, the value will be expanded with leading zeros until it fits the field size.

```
Before:    [ABC^-158^123^
                ^

Command:   +150
After:     [ABC^-Ø08^123^
                ^
```

If a minus sign must appear in the results to make the correct algebraic results, it will appear before any digits loaded.

```
Before:    [ABC^123^Ø17^
                ^

Command:   +ØØØ1
After:     [ABC^124^Ø17^
                ^
```

Any unnecessary leading zeros will be dropped from the numeric value before it is returned to the input buffer.

CHAPTER 13

ENGLISH LANGUAGE PROCESSING WITHIN PROCS

ENGLISH language statements may be processed in PROCs by loading them in the primary and secondary output buffers and using the P commands to process them through the ENGLISH processor. When executed this way the primary output buffer is executed first as one ENGLISH statement, with the secondary output buffer supplying additional parameters which may be used for designating additional operations that would normally be performed from TCL after the primary function was complete.

13.1 P command

-Format- P

The P command will convert all attribute marks in the primary and secondary output buffers to spaces and cause them to be executed as a TCL statement. The contents of the primary output buffer will be executed as an ENGLISH, TCL1, or TCL2 verb, and the secondary output buffer will provide additional parameters for the operation. All system output would be displayed on the terminal just as if the function were being executed from TCL. Before returning to the next step in the program, both output buffers are cleared and the output buffer stack pointer is turned off. For example, to perform the program equivalent of the TCL operation:

```
COPY VENDOR 0001
TO: (SAVE)
```

it would be necessary to code:

```
STOFF
RO
HCOPY VENDOR 0001
STON
RO
H(SAVE)
P
```

This coding would cause the COPY statement to be executed from the primary output buffer and would provide the additional response required by the processor from the secondary output buffer.

An additonal function which could be performed would be to provide a series of select record keys to be used by a processing loop for updating records such as:

```
STOFF
RO
HSSELECT INVOICE WITH DATE <= "10/13/75"
H BY CUSTOMER
STON
RO
HPQ-SELECT 3
P
```

This series of commands would perform the sort select as specified and place the string of keys of the items selected in Select Register 3, as required by the secondary output buffer contents. When this series of statements is executed, the number of items selected would be displayed just as it would if the SSELECT were done from TCL.

13.2 PH command

-Format- PH

The operation of the PH command is identical to the operation of the P command with only one exception: the PH command will suppress any system output to the terminal running the program. For example, if a SELECT statement is processed by the PH command, the operation will proceed normally, but the system output of the number of items selected will not occur.

13.3 PP command

-Format- PP

The PP command is a programmer's debugging tool which will not normally be included in an operational program. Generally it performs the same functions as the P command, however, it displays the contents of the output buffers prior to execution and stops to allow selective override action of the operation.

The primary output buffer is displayed on the terminal starting at the current cursor location and continues on the next line if line overflow occurs. The secondary output buffer starts on the left margin of the line following the last word of the primary output buffer and is followed immediately by a back arrow designating the end of buffer. The system will then prompt with a question mark on the following line and wait for the operator to enter an option (followed by new-line) for processing the output buffers. The options available are:

N Kill the process and return immediately to TCL.

S Supress processing of the output buffers, clear both buffers, and return to the next step in the program.

(No input) Process the output buffers as an ENGLISH statement, then proceed to the next statement in the program.

13.4 PQ-SELECT verb

-Format- PQ-SELECT n

The PQ-SELECT auxiliary verb has been added to allow the results of an ENGLISH SELECT or SSELECT verb to be directed to the select register specified by the numeric value n. When processing from a program, the select statement should be loaded into the primary output buffer, and the PQ-SELECT should be loaded into the secondary output buffer. When executed together as a TCL statement, the select will be performed as directed, and resultant string of keys generated will be loaded into the select register designated in the PQ-SELECT statement. The previous contents of the select register will be cleared even if no items are selected. For example, the following series of statements within a program:

```
STOFF
RO
HSSELECT CUSTOMER WITH BALANCE > "Ø" BY ZIP
STON
RO
HPQ-SELECT 4
P
```

Will cause the select statement to generate a string of keys to items in the CUSTOMER file and will load them into select register 4. The select register pointer will be set to the first value in the select register after the operation.

A caution must be noted with this instruction. If the selection processor does not find any items on the file to return, the PQ-SELECT verb will not be executed. In this case, the contents of the select register designated will not be changed from the previous contents, and could result in items being returned in the select register when none are expected. Proper operation of this facility therefore requires that the select register be cleared before executing the P statement.

13.5 PQ-RESELECT verb

-Format- PQ-RESELECT n

The PQ-RESELECT verb will reset the pointer of any select register to the first value in the register. Use of values in a select register does not destroy the ones used, but merely moves the pointer down the string of values. Therefore, this verb was provided to allow any select register pointer to be reset to the front of the string without regenerating it. The program statements:

```
STOFF
RO
HPQ-RESELECT 4
P
```

will re-establish the string of values in select register 4 by moving the pointer back to the first value in the string. This function may be used anytime, even if the pointer has not yet been advanced to the end of the select register string.

CHAPTER 14

TERMINAL AND LINE PRINTER OUTPUT

Formatted terminal and line printer output is provided to allow displaying or printing items which are necessary for operator information. Line printer capabilities include the use of the spooler header capability to eliminate line counting to determine the end of page. If necessary, other spooler functions may be performed for the program by execution of the SP-ASSIGN and other spooler verbs as provided in the ENGLISH processing capability.

14.1 T command

-Format- T opr1,opr2,opr3,...

The T statement provides the programmer with the capability of controlling output to the CRT terminal. The statement itself is a string of terminal commands, separated by commas, which allow cursor control and data output items to be appended together to any length. Generally a T statement is one line long; however, if the last character of the statement is a comma which would normally be used as a separator, the statement will be continued on the next line of coding. The individual commands which may be used are outlined below.

(M,n) Position Cursor, where m and n are numeric literals. Will position the cursor to column m, line n on the screen.

(M) Position Cursor, where m is a numeric literal less than 79, will position the cursor to column m on the current line.

(,N) Position Cursor, where n is a numeric literal less than 24, will position the cursor to line n in the current column.

"Literal" Display Literal, will display the literal surrounded by double quotes on the screen starting at the current cursor location.

REFERENCE, will display the contents of the attribute specified by a direct or indirect reference starting at the current cursor position. May contain an appended ENGLISH conversion code as described in Chapter 15.

B Bell, used to sound the audible signal at the terminal. Cursor does not move

Sn Spaces, outputs n spaces to the terminal starting at the current cursor location. The maximum value which may be used for n is 99. The cursor is positioned after the last space output. A direct or indirect reference to any attribute may be used to supply the number of spaces to generate (as S&3.%2).

*Cn Character Repeat, outputs the literal character c, n times, starting at the current cursor location. The maximum value which may be used for n is 99. The cursor is positioned after the last character output. The number of characters to output may be supplied by coding any direct or indirect reference in place of the numeric literal N.

C Clear, clears the entire screen and moves the cursor to 'home' position at column Ø, line Ø.

U Up, moves the cursor up one line on the screen. Column position does not change.

Xnn Hex Output, outputs the single byte character represented by the hexadecimal equivalent nn to the terminal. If this is a displayable character or cursor control character, the cursor will move accordingly. The character which will be displayed may be provided by coding any direct or indirect reference in place of the literal nn. See Appendix A for hex character representation.

Inn Integer Output, outputs the single byte character represented by the integer equivalent nn to the terminal. If this is a displayable character or cursor control character, the cursor will move accordingly. The character which will be displayed may be provided by coding any direct or indirect reference in place of the literal nn. See Appendix A for integer character conversion.

T Tag, creates a location for internal command looping and sounds the audible signal at the terminal.

D Delay, causes a short delay in processing before the next command in the string is executed.

L Loop, moves statement parsing back to the last tag command executed and causes all commands except other L commands to be executed again. Upon completing execution, proceeds to the next command in the string. If there is no previous tag in the T statement command string, no loop will occur and the next command in the string will be executed.

The function of the last three commands is to provide the

facility of flashing a message on the CRT screen without creating an unnecessarily long string of coding. For example, the following statement will cause the message INVALID to be flashed on the screen three times before proceeding to the next statement.

```
T (Ø,7),T,(Ø),"INVALID",B,D,(Ø),S7,L,L,L
```

The equivalent statement without the internal looping would have to code the message, bell, delay, spacing, and positioning three times.

Since the double quote character (") is used as a field delimiter for a literal value which must be displayed, it cannot itself be displayed from a coded literal field. This restriction may be circumvented by coding the double quote as either the hex equivalent (X22) or the integer equivalent (I34) as shown below:

```
T (Ø,15),"USE ",X22," FOR DUPLICATES"
```

The above line will display the message:

```
USE " FOR DUPLICATES
```

The ENGLISH conversion utility may be called to allow display of any value as provided by the ENGLISH processor. This function will allow date, numeric, or hexadecimal conversion as items are displayed and will not change the internal value of the attribute. For example:

```
T (5,8),S12,(5),&3.2:MD2,12 :
T (7Ø,Ø),%1:D2/:
```

See Chapter 15 for further details on conversions.

14.2 O command

-Format- Oliteral-string

The O command will output a literal string following the O to the terminal starting from the current location of the cursor. If the string runs beyond the end of a line, the next character will continue at the beginning of the next line. At the end of the literal string the O command will move the cursor to the beginning of the next line unless the last character is a +. This will cause the cursor to remain positioned after the last character output (output of the + will be suppressed when it is the last character). Valid forms of this command are:

```
OTRANSFER TO MS18
O  INPUT YOUR AGE  +
 --     -    -   --
```

14.3 L statements

-Format- L opr1,opr2,opr3,....

The L statement is provided to give the programmer the capability of controlling data output to the system printer. The statement is a string of printer control and data output operations separated by commas. While the statement is coded on one line in the program, it may be continued in the next statement by placing an operation separating comma as the last character of the line. Lines of any length up to 140 bytes may be output to the printer, though the coded line which performs the list may be of any length. An attempt to output a single line longer than 140 bytes will cause the program to abort during execution.

The operations which may be performed in the L statement are outlined below:

(n) Column, will set the output position to the column designated by the numeric value n.

"Literal" Output Literal, will output the literal value contained within the double quotes starting at the current column position.

REFERENCE Attribute Reference, will output the contents of the attribute specified by a direct or indirect reference starting at the current column position. May contain an appended ENGLISH conversion code as described in Chapter 15.

T Top of Form, will cause the printer to advance to the next top of form position on the printer. Must appear as the first function on an L statement.

N Space Lines, a numeric value coded as the first function of an L statement will cause the printer to space the quantity of extra lines specified before printing the following data. Use of a 1 will cause double spacing.

Lnn New Line, will cause the subsequent printing to appear on a new line after spacing nn lines. The number of line to advance before continuing printing may be specified by using any direct or indirect reference in place of the literal nn.

+ Continue, when used as the last command in the string will cause output to be held in a work area and merged with the next L statement in the program. The first command in the next line must begin with a column positioning (n). The hold area for this function is volatile so use should be restricted to small areas of the program.

The L statement may be used as shown in the following listings:

```
L (5),"ACCOUNT",&4.2,(3Ø),&7.2:MD2,12 :
L (3),"SAVE FOR NEXT LINE",
(27),&7.1:D2/:,(52),
"FOR STORE",&7.3
```

The second example above, which consists of the last three lines, shows how one L statement may be continued on succeeding lines by making a comma the last character on the previous lines. Any number of continuation lines may be used this way.

Special L Command Operations

The following commands must appear on a line by themselves in an L statement. They must not appear imbedded within a L statement with other operations. The only valid forms of these commands is:

```
L C
L N
```

C Close, will close the printer (it is automatically opened on the first L statement). Causes completion of a spooler file and places the file in the print queue.

N No Printing, used to send L statement output to the terminal when used alone in a command (as 'L N'). May be used to provide controlled output to a printing terminal or as a debugging tool. Will be reset to system printer output when a printer close statement is executed.

14.3.1 L HDR command

-Format- L HDR,opr1,opr2,opr3,...

The header statement allows the processor to perform line and page counting based on the terminal characteristics set for the line printer. The header may consist of any quantity of lines which will print automatically at the top of a new page when the previous page is filled. The functions available for the header statement are separated by commas, and are listed below:

(n) Column, will set the output position to the column designated by the numeric value n.

"Literal" Output Literal, will output the literal value contained within the double quotes starting at the current column position.

REFERENCE Attribute Reference, will output the contents of the attribute specified by a direct or indirect reference starting at the current column position. May contain an appended ENGLISH conversion code as described in Chapter 15.

T Time & Date, outputs the system time and date at the time the header is executed. Format is HH:MM:SS DD MMM YY.

P Page Number, increments the page number by one and outputs it at the current column position. The page counter is set to zero upon program initialization and will count up to 9999. The output is left justified and zero supressed.

Z Zero Page Counter, when executed within a header statement, will cause the page counter to be set to zero. (Execution of a second header statement does not otherwise change the page counter)

L Line Feed, causes a line feed within the header statement but does not terminate the header function.

The header statement, like the L statement may be continued on the next line in the program by coding a comma as the last character of the line.

Headers are held in a hold area which will not be altered except by executing another L HDR statement. Since attributes named in the header statement are loaded into this hold area at the time the statement is executed, subsequent changes made to the attributes in the internal buffers will not change the values displayed in the header. Use of the top of form operation in an L statement should be eliminated when using the header statement since this will cause the line counter and page counter to lose track of where the printer is processing. As a substitute for this function, another header statement should be executed to get to the top of the next form.

CHAPTER 15

MISCELLANEOUS COMMANDS

15.1 X command

-Format- Xliteral-string

The X command is used for exiting from a PROC to TCL. Execution of an X command will output a literal string appended to it to the current position of the terminal and return control to TCL. This is a non-returnable function which should be used only when no other PROC instructions need to be executed. Following are some examples of X statements with appended messages to be output to the terminal before exiting the PROC:

Command	Terminal Output
XEND OF PROCESSING	END OF PROCESSING
X	(no output)
XINVALID RECORD ABORT	INVALID RECORD ABORT
XBYE SEE YA LATER	BYE SEE YA LATER

15.2 U commands (user exits)

-Format- Unnnn
(see additional information published locally)

The commands beginning with a U are user exits to other routines written in other languages, usually in assembler, to perform functions which would be too complex or would take too much time in RPL. User exits will vary from site to site with some being used at only one location while others are universally used. Consult your local system programmer for operation of these commands.

15.3 D command

-Format- Ditem-reference

The D command is generally a programmers' tool for displaying the contents of an internal buffer at the terminal during program execution. The display will be the entire contents of the buffer from the referenced item to the end of the buffer. If the item referenced in the command is a numeric value only, the display will be the primary input buffer from the attribute specified by the numeric value; ie, a D3 will display %3 to end of buffer. If the reference is a direct or indirect reference to any internal buffer (primary input, either output, or file buffers) that attribute

referenced and the rest of the buffer will be displayed. If the reference is to one of the select registers, the contents of the register from the current value on will be displayed and the pointer will be incremented to the next value.

Where attribute zero is specified for a buffer that does not have an attribute zero, the entire buffer will be displayed. All buffer pointers other than the select registers will not be affected by any D command. If an attribute is referenced in the D command which is beyond the current end of the buffer, null attributes will be constructed up to that point and a null attribute will be displayed.

As an example of the type of display which can be expected, if the following buffer contents represented all buffers, the next table will illustrate what is displayed by the D commands.

[3^DEF]GHI^JKL^^MNO^

Command	Buffer	Display
DØ	P%1	[3^DEF]GHI^JKL^^MNO^
D&1.1	&1.1	^DEF]GHI^JKL^^MNO^
D%%1	P%3	^JKL^^MNO^
D!1	!1	[3^DEF^GHI^JKL^^MNO^
next D!1	!1	^DEF^GHI^JKL^^MNO^

15.4 TR command

-Format- TR

The TR command is a programmers' tool used to initialize a trace utility built into the RPL language processor. Execution of the TR command will cause all succeeding instructions to be displayed on the terminal before they are executed. Statements which are scanned more than once by the processor, such as the IF command if TRUE, will be displayed one time for each parsing, with secondary displays omitting the portion of the statement which has already been processed. If the trace facility is already on, the command will not perform any function.

The trace function slows execution of a program significantly so it may be advantageous to execute the command as the stub of an IF statement which will only execute when the process in question occurs.

15.5 TROFF command

-Format- TROFF

The TROFF command will turn off a tracing utility which was previously initiated with the TR command. The TROFF command will not perform any function if trace is not operational at the time of execution.

15.6 Conversions

ENGLISH conversions may be performed in IH, L, and T statements. The IH command provides the programmer with the capability of converting attributes from internal to external formats in storage. The T and L statement conversions will allow display of data in either format without changing the stored value. For all statements, the conversion code should immediately follow the attribute reference providing the value to be converted. If the conversion code is surrounded by colons (as in :MD2:), the conversion will be from internal to external format. If the conversion code is surrounded by semicolons (as in ;MD2;) the conversion will be from external to internal format. The following series of examples will illustrate the function of the conversion operations:

Input Value	Conversion	Output Value
12345	:MD2:	123.45
12345	;MD2;	1234ØØ
3Ø79	:D2/:	Ø6/Ø5/76
3Ø39	;D2/;	3Ø39
53	:MX:	3533

15.7 Indirect Parenthetical Expressions

Anyplace in the program where a parenthetical expression is used to provide values to the processor, a direct or indirect reference may be substituted which will use the value in the attribute as if it were coded directly into the statement. This allows incrementing an expression without coding many different statements into the program. Examples of statements which may use this function are coded below:

```
A(%5)              A(,&3.2)              A(1,%%3)
A(%3,4)            A(&2.#3)              A(&3.2,#%1)
MV &3.2 A(%5,3),N(&3.2,&7.%9),A(,%3)
T (%3,%4),"LITERAL",(#8),&3.%2,(&7.2),%3:MD2,12 :
L (1),&4.2,(&1.5),&4.4:MD2,1Ø :
S(%3)
IF &3.2 = (%4) GO 385
(MASTER %5)
```

15.8 Compound Command Statements

Several commands may be included in one statement by separating them with subvalue marks. This facility is primarily provided to allow multiple commands in one line where an error return is involved, or to allow multiple commands to be executed in one value of a multivalued stub for an IF command. Where this ability is involved, any subroutine return functions will be to the next command in the subvalue string unless a statement skip function is used such as the F-READ statement. Then the return will count only attribute marks in figuring where the return will be.

The following series of statements will illustrate the use of the compound statement.

```
IF &3.2 = A]""](ØN) GO 3Ø]MV &7.3 "SAVE"\GO 15]GO 38

F-READ 1 &3.2
T (5,23),&3.2," NOT ON FILE"\ GO 3ØØ
MVA &3.3 &1.Ø

IF # &5.1 MV &7.1 &3.2\GOSUB 3ØØØ\IF %3 GO 5ØØ

T (3),"REPEAT INPUT"\C GO BACK TO BEGINNING\GO 1Ø
```

In the third example above, the return from the GOSUB statement will be to the following IF unless a RSUB n format is used. In that case, the return will be to the n'th statement following in the program, and never to the last subvalue statement of the stub.

The C command, when executed, will always comment out that line through the next attribute mark. This means that comments can not normally be used as imbedded commands in lines with value marks or sub-value marks. The purpose of this definition of a comment is to allow code to be temporarily dropped from programs through the use of a comment statement at the front of the line. Therefore, the GO statement in the last example will never be executed.

CHAPTER 16

OPTIMIZING PROCS

16.1 PQ-COMP verb

-Format- PQ-COMP filename item-name
(executed from TCL)

The PQ-COMP verb will perform an abbreviated compilation on an RPL program to increase its efficiency. In so doing it will save both the source and object programs to allow later changes to be added without any difficulty. The actual compilation of the program consists of deleting all comment statements, changing all intra-program transfers to direct jumps, and converting all IH and H statements to direct load format. These changes could speed up program execution time for a large program by a factor of ten or more. Additionally, the compiler provides a system time and date stamp in the first attribute of the program (multivalued with the PQ statement).

The compiler will place the compiled version of the program in the same file and under the same name as the uncompiled version. The source version will have a dollar sign placed in front of the name and it will be stored back in the same file. If the source version already has a dollar sign, the compilation will proceed as if it did not.

For example:

Source	Compiled	New Source
MS3	MS3	$MS3
$CHG	CHG	$CHG

This arrangement allows the source item or the compiled program to be run without the need to change inter-program transfer routines in other programs. Also, should further changes be needed to a previously compiled program, the source version may be edited and recompiled without having to copy it back over the old compiled program.

A source item which contains 500 or less bytes will not be processed by the PQ-COMP verb since it will occupy only one frame in storage. The execution of these programs should not be abnormally long, however, an override capability is provided as outlined in the options below.

Options are provided for the PQ-COMP verb which will allow processing of short programs, listing on the printer, compilation suppression, etc. These options must be enclosed in parentheses, separated by commas, and included after the

last item in the list of items to be processed. The options are:

L List the program

P Send compiler output to the system printer.

F Do not generate a compiled program. If used the compile will not change any item ID's.

(Numeric) A numeric value designating the smallest item to be processed by the compiler (defaults to 500 if not specified). Generally used to obtain listing of programs smaller than 500 bytes.

As an example:

```
PQ-COMP PRGM GL2C1 (L,P,F)
PQ-COMP USER * (L,P)
```

The asterisk in the last example signifies that the compiler is to perform the requested operations on all items in file USER. Use of the asterisk in a file in which no programs have been compiled is inefficient since some programs may be compiled twice. This is due to the new source program being stored in a group that may not have already been compiled. The best solution where all programs need to be compiled is to perform a SELECT prior to the PQ-COMP and use the generated item list to provide the source items to the compiler.

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
Ø	ØØ	6Ø	NUL	none	P[cs]
1	Ø1	Ø1	SOH	none	A[c]
2	Ø2	Ø2	STX	none	B[c]
3	Ø3	Ø3	ETX	none	C[c]
4	Ø4	37	EOT	none	D[c]
5	Ø5	2D	ENQ	none	E[c]
6	Ø6	2E	ACK	none	F[c]
7	Ø7	2F	BEL	none	G[c]
8	Ø8	16	BS	none	H[c]
9	Ø9	Ø5	HT	none	I[c]
1Ø	ØA	25	LF	none	J[c]
11	ØB	Ø8	VT	none	K[c]
12	ØC	ØC	FF	none	L[c]
13	ØD	ØD	CR	none	M[c]
14	ØE	ØE	SO	none	N[c]
15	ØF	ØF	SI	none	O[c]
16	1Ø	1Ø	DLE	none	P[c]
17	11	11	DC1	none	Q[c]
18	12	12	DC2	none	R[c]
19	13	3A	DC3	none	S[c]
2Ø	14	3C	DC4	none	T[c]
21	15	3D	NAK	none	U[c]
22	16	32	SYN	none	V[c]
23	17	26	ETB	none	W[c]
24	18	18	CAN	none	X[c]
25	19	19	EM	none	Y[c]
26	1A	3F	SUB	none	Z[c]
27	1B	27	ESC	none	
28	1C	1C	FS	none	
29	1D	1D	GS	none	
3Ø	1E	1E	RS	none	
31	1F	1F	US	none	
32	2Ø	4Ø	space	blank	space
33	21	5A	!	!	A[cs]
34	22	7F	"	"	B[cs]
35	23	7B	#	#	C[cs]
36	24	5B	$	$	D[cs]
37	25	6C	%	%	E[cs]
38	26	5Ø	&	&	F[cs]
39	27	7D	'	'	G[cs]

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
40	28	4D	(	(	H[cs]
41	29	5D	)	)	I[cs]
42	2A	5C	*	*	J[cs]
43	2B	4E	+	+	K[cs]
44	2C	6B	,	,	L[cs]
45	2D	60	-	-	M[cs]
46	2E	4B	.	.	N[cs]
47	2F	61	/	/	O[cs]
48	30	F0	0	0	P[cs]
49	31	F1	1	1	Q[cs]
50	32	F2	2	2	R[cs]
51	33	F3	3	3	S[cs]
52	34	F4	4	4	T[cs]
53	35	F5	5	5	U[cs]
54	36	F6	6	6	V[cs]
55	37	F7	7	7	W[cs]
56	38	F8	8	8	X[cs]
57	39	F9	9	9	Y[cs]
58	3A	7A	:	:	Z[cs]
59	3B	5E	;	;	
60	3C	4C	<	<	
61	3D	7E	=	=	
62	3E	6E	>	>	
63	3F	6F	?	?	
64	40	7C	@	@	
65	41	C1	A	A	
66	42	C2	B	B	
67	43	C3	C	C	
68	44	C4	D	D	
69	45	C5	E	E	
70	46	C6	F	F	
71	47	C7	G	G	
72	48	C8	H	H	
73	49	C9	I	I	
74	4A	D1	J	J	
75	4B	D2	K	K	
76	4C	D3	L	L	
77	4D	D4	M	M	
78	4E	D5	N	N	
79	4F	D6	O	O	

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
80	50	D7	P	P	
81	51	D8	Q	Q	
82	52	D9	R	R	
83	53	E2	S	S	
84	54	E3	T	T	
85	55	E4	U	U	
86	56	E5	V	V	
87	57	E6	W	W	
88	58	E7	X	X	
89	59	E8	Y	Y	
90	5A	E9	Z	Z	
91	5B	80	[	[	
92	5C	E0	/	/	
93	5D	90	]	]	
94	5E	5F	^	^	
95	5F	6D	-	-	
96	60	79		none	0[cs]
97	61	81		none	1[cs]
98	62	82		none	2[cs]
99	63	83		none	3[cs]
100	64	84		none	4[cs]
101	65	85		none	5[cs]
102	66	86		none	6[cs]
103	67	87		none	7[cs]
104	68	88		none	8[cs]
105	69	89		none	9[cs]
106	6A	91		none	
107	6B	92		none	
108	6C	93		none	
109	6D	94		none	
110	6E	95		none	
111	6F	96		none	
112	70	97		none	0[c]
113	71	98		none	1[c]
114	72	99		none	2[c]
115	73	A2		none	3[c]
116	74	A3		none	4[c]
117	75	A4		none	5[c]
118	76	A5		none	6[c]
119	77	A6		none	7[c]

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
120	78	A7		none	8[c]
121	79	A8		none	9[c]
122	7A	A9		none	
123	7B	C0		none	
124	7C	6A		none	
125	7D	D0		none	
126	7E	A1		none	
127	7F	07	DEL	none	
128	80	04		none	
129	81	06		none	
130	82	08		none	
131	83	09		none	
132	84	0A		none	
133	85	13		none	
134	86	14		none	
135	87	15		none	
136	88	17		none	
137	89	1A		none	
138	8A	1B		none	
139	8B	20		none	
140	8C	21		none	
141	8D	22		none	
142	8E	23		none	
143	8F	24		none	
144	90	28		none	
145	91	29		none	
146	92	2A		none	
147	93	2B		none	
148	94	2C		none	
149	95	30		none	
150	96	31		none	
151	97	33		none	
152	98	34		none	
153	99	35		none	
154	9A	36		none	
155	9B	38		none	
156	9C	39		none	
157	9D	3B		none	
158	9E	3E		none	
159	9F	41		none	

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
160	A0	42		none	
161	A1	43		none	
162	A2	44		none	
163	A3	45		none	
164	A4	46		none	
165	A5	47		none	
166	A6	48		none	
167	A7	49		none	
168	A8	4A		none	
169	A9	4F		none	
170	AA	51		none	
171	AB	52		none	
172	AC	53		none	
173	AD	54		none	
174	AE	55		none	
175	AF	56		none	
176	B0	57		none	
177	B1	58		none	
178	B2	59		none	
179	B3	62		none	
180	B4	63		none	
181	B5	64		none	
182	B6	65		none	
183	B7	66		none	
184	B8	67		none	
185	B9	68		none	
186	BA	69		none	
187	BB	70		none	
188	BC	71		none	
189	BD	72		none	
190	BE	73		none	
191	BF	74	space	none	
192	C0	75	@	@	
193	C1	76	A	A	
194	C2	77	B	B	
195	C3	78	C	C	
196	C4	8A	D	D	
197	C5	8B	E	E	
198	C6	8C	F	F	
199	C7	8D	G	G	

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
200	C8	8E	H	H	
201	C9	8F	I	I	
202	CA	9A	J	J	
203	CB	9B	K	K	
204	CC	9C	L	L	
205	CD	9D	M	M	
206	CE	9E	N	N	
207	CF	9F	O	O	
208	D0	A0	P	P	
209	D1	AA	Q	Q	
210	D2	AB	R	R	
211	D3	AC	S	S	
212	D4	AD	T	T	
213	D5	AE	U	U	
214	D6	AF	V	V	
215	D7	B0	W	W	
216	D8	B1	X	X	
217	D9	B2	Y	Y	
218	DA	B3	Z	Z	
219	DB	B4	[	[	
220	DC	B5	/	/	
221	DD	B6	]	]	
222	DE	B7	^	^	
223	DF	B8	-	-	
224	E0	B9	@	@	
225	E1	BA	a	A	
226	E2	BB	b	B	
227	E3	BC	c	C	
228	E4	BD	d	D	
229	E5	BE	e	E	
230	E6	BF	f	F	
231	E7	CA	g	G	
232	E8	CB	h	H	
233	E9	CC	i	I	
234	EA	CD	j	J	
235	EB	CE	k	K	
236	EC	CF	l	L	
237	ED	DA	m	M	
238	EE	DB	n	N	
239	EF	DC	o	O	

APPENDIX A
CHARACTER SET

Decimal	Hex	EBCDIC	ASCII	Prism	Entered
240	F0	DD	p	P	
241	F1	DE	q	Q	
242	F2	DF	r	R	
243	F3	E1	s	S	
244	F4	EA	t	T	
245	F5	EB	u	U	
246	F6	EC	v	V	
247	F7	ED	w	W	
248	F8	EE	x	X	
249	F9	EF	y	Y	
250	FA	FA	z	Z	
251	FB	FB	SB	[	K[cs]
252	FC	FC	SVM	/	L[cs]
253	FD	FD	VM	]	M[cs]
254	FE	FE	AM	^	N[cs]
255	FF	FF	SM	-	O[cs]

APPENDIX B

ALPHABETICAL LISTING OF INSTRUCTIONS WITH ALTERNATE FORMS

Command	Alternate	Format
+		+numeric-literal
-		-numeric-literal
A		A Ac Ap Acp A(m) Ac(m) A(,n) Ac(,n) A(m,n) Ac(m,n)
B	BACK	B
BO		BO
D		Dnumeric-literal Dreference
E*		E*]stub-statement
E^		E^]stub-statement
EE		EEerror message
EI		EIRPp(column,row),reference,edit
F	FORWARD	F
F-CLEAR	F-C	F-CLEAR filenbr
F-DELETE	F-D	F-DELETE filenbr
F-FREE	F-F	F-FREE
F-INPUT	F-I	F-INPUT filenbr filename
F-OPEN	F-O	F-OPEN filenbr filename
F-READ	F-R	F-READ filenbr itemname
F-WRITE	F-W	F-WRITE filenbr
F;		F;opr1;opr2;...;?destination
GO	G	GO label
GOSUB	GS	GOSUB label

		GSUB
H		Hliteral-string
IF		IF opr1 operator opr2 stub
IFN		IFN opr1 operator opr2 stub
IH		IHliteral-string
IN		INprompt
IP		IPprompt-destination IPBprompt-destination IPFprompt-destination
IT		IT ITA ITC
KSUB	KS	KSUB n
L		L opr1,opr2,opr3,...
L HDR		L HDR,opr1,opr2,opr3,...
MARK	M	MARK
MV		MV destination source1,source2,...
MVA		MVA destination source
MVD		MVD destination source
O		Oliteral-string
P		P
PH		PH
PP		PP
PQ		PQ
RI		RInumeric-literal
RO		RO
RSUB	RS	RSUB n
RTN	RT	RTN n
S		Snumeric-literal Sreference S(numeric-literal)

STOFF	STOF ST OFF ST OF	STOFF
STON	ST ON	STON
T		T opr1,opr2,opr3,...
TR	TRON	TR
TROFF	TROF	TROFF
U		Unnnn
X		Xliteral-string

TABLE OF CONTENTS

APPENDIX C

1 Resource Locks

U0191	locks a resource
U1191	unlocks a resource
U3191	unlocks all resources for this line

A resource represented by a numeric value in the following line is locked to inhibit a similar instruction from completing in another program until that resource is unlocked. This instruction may be used to inhibit operation of two programs which will interfere with each other, or one program that may not be executed from two terminals at once.

The 'resource' which will be locked is an intangible represented by a numeric literal on the following line. If an attempt is made to execute a resource lock, and that resource is already locked, program execution will be suspended. The locked terminal will display a pound sign (#) at intervals until the resource is unlocked, then execution will proceed normally.

Execution of:

```
U0191
N
```

Will lock resource 'N' until released by:

```
U1191
N
```

The execution of U3191 at any time in a program will unlock all resources which have been locked by this process. A numeric value does not need to follow this user exit.

'N' may be any numeric literal, however, to insure future compability, it should be limited to the range of Ø to 7.

2 Inhibit Break Key Operation

U4191	Inhibit break key
U5191	Allow break key

These two user exits will allow the programmer to selectively inhibit the operation of the break key on the terminal keyboard. After use of the inhibit user exit (U4191) the break key will not operate. Depression of the break key will bring no results until U5191 is executed to allow the break key to be functional again.

3 Obtain terminal line number

U0194 obtain terminal line number

The line number of the terminal and the account name which the executing terminal is logged on to will be placed into the current position of the output buffer.. The data loaded is the same as that in response to the WHO verb.

For example:

```
Before:    #1      ABC^187^^
                              ^

Command:   U0194
After:     #1      ABC^187^5^BCP^^
                                  ^
```

4 System Time

U019F Get System Time

This function will place the current system time in the current position of the input buffer in the form HH:MM:SS. The time is based on a twenty four hour clock and 1:00 PM would be loaded as 13:00:00.

For example:

```
Before:    %1      [ABC^DEF^GHI^JKL^MNO^
                     ^

Command:   U019F
After:     %1      [ABC^08:43:21^GHI^JKL^MNO^
                     ^
```

5 G and S Correlative Extraction

U01A4 Extract G or S Correlative Field

The user exit will extract the value specified in the second statement following from the field specified by the first statement following and place the results in the current position of the input buffer. The sequence of instructions:

```
U01A4
attribute reference
:Gncm:
```

Will place the separated value into the current position of the input buffer where:

n = the number of separator characters to advance before starting

c = the separator character (not numeric)

m = the number of separator characters to advance before stopping (the number of fields to extract).

For example:

```
Before:     &1.Ø     [ABC^DEF^GHI*JKL*MNO*PQR^STU^
            %1       [111^222^333^444^555^
                        ^

Command:    UØ1A4
            &1.2
            :G1*2:
After:      %1      [111^JKL*MNO^333^444^555^
                       ^
```

Any non-numeric character may be used for the separator, despite the exclusive use of the asterisk here.

If a Ø or a null is used for the start point (n in the definition), the extraction will start from the beginning of the attribute.

Use of an S in place of the G will cause the string of characters extracted to be split into their individual fields and placed into successive attributes of the input buffer. For example:

```
Before:     &1.Ø     [ABC^DEF^GHI*JKL*MNO*PQR^STU^
            %1       [111^222^333^444^555^
                        ^

Command:    UØ1A4
            &1.2
            :S1*2:
After:      %1      [111^JKL^MNO^444^555^
                       ^
```

Note that in this case the extracted fields were split further by the S correlative function, and the two fields extracted were placed into two different attributes.

6 System Date

U119F Get System Date

The current system date is placed in the current position of the input buffer in the form DD MMM YYYY.

For example:

```
Before:     %1      [ABC^DEF^GHI^JKL^MNO^
                        ^
Command:    U119F
After:      %1      [ABC^DEF^13 NOV 1976^JKL^MNO^
                        ^  -  -
```

7 Date Format Verification

```
U419F     MM/DD/YY Editing
U519F     YYMMDD Editing
U619F     MMDDYY Editing
```

The current location of the input buffer is checked for a valid date in the format specified by that user exit. The two digits of the month are checked for a value between Ø1 and 12, the two digits of the day are checked for a value between Ø1 and 31, and the two digits of the year are checked for numerics. No checking is done for such invalid dates as 3Ø FEB or 31 NOV, or 29 FEB on the wrong year. If the verification of the date shows a valid entry, the next command executed is the third statement following the user exit. Otherwise, the error return is the next instruction after the user exit.

For example:

```
U419F
T (Ø,5),"INVALID DATE INPUT",D,(Ø),S18,L,L
GO 5ØØ
IH%3;D;
```

If the input to the user exit did not match the date format specified, the T statement would be executed, then the GO statement. If the date was valid, the IH conversion would be executed next.

8 Insert Character (ENGLISH conversion)

```
U719Fc    Insert Character into Attribute
```

The character following the user exit (c above) is inserted between the second and third, and the fourth and fifth characters of the field. This user exit is valid anyplace an ENGLISH conversion may be used and will truncate any values to six characters if they are longer.

For example:

```
Before:     %1      [ABC^DEF^GHI^MMDDYYGGG^JKL^
                      ^
Command:    IH%4:U719F/:
After:      %1      [ABC^MM/DD/YY^GHI^MMDDYYGGG^JKL^
                      ^
```

NOTE: execution of this user exit on attributes which are shorter than six characters will give unpredictable results which will vary depending on the location of the source field.

9 Load Attribute Length

U11A4 Count an Attribute Length

The length of the attribute specified in the direct or indirect reference in the following statement is placed in the current position of the input buffer.

For example:

```
Before:     &1.Ø    [ABC^DEFGHI^JKL^MNO^
            %1      [AAA^BBB^CCC^DDD^
                        ^
Command:    U11A4
            &1.1
After:      %1      [AAA^BBB^6^DDD^
                        ^
```

1Ø Unique Key Generator

U21A4 Load Unique Key

A unique five character hexadecimal key will be placed in the current location of the input buffer. Each successive use of this user exit will result in the key being incremented by one resulting in over a million keys before a repetition.

For example:

```
Before:     %1      [ABC^DEF^GHI^JKL^MNO^
                        ^
Command:    U21A4
After:      %1      [ABC^DEF^Ø1B4F^JKL^MNO^
                        ^

Next Use:   %1      [ABC^DEF^Ø1B5Ø^JKL^MNO^
                        ^
```

11 Multivalued Field Search

U51E8 Scan Multivalued Field for Match

This function will scan a multivalued field until a match is found for the arguement. The arguement shall be the contents of the current location of the input buffer and the multivalued field shall be designated by a direct or indirect reference in the statement immediately following the user exit.

The contents of the arguement will be compared character by character (up to the length of the arguement) to the first characters of each value of the string until a match is found. The entire value at which the match is found will then overlay the arguement in the input buffer. If no match is found, or if a value is encountered which collates higher than the arguement, then a null field is loaded over the arguement. For example:

```
Before:     &1.Ø    [ABC^AB]ABCD]ABDE]M]AC^Ø12^
Command:    U51E8
            &1.1
```

Assuming the input buffer pointer is at %3,

```
%3 Before:   A      ABD    AC       M      RS
%3 After:    AB     ABDE   (NULL)   M      (NULL)
```

If the argument contains a value mark, only those characters occuring before the value mark will be used in scanning for a comparison, all other characters will be compared as characters only. This function was intended for use on sorted multivalued fields only, but will work as described above for fields in any order.

12 Duplicates in Multivalued Fields

U11E9 Place Duplicates in Multivalued Field

When used immediately before an MVA instruction in a program, this user exit will place the subject into the object field again, even if that value already exists in the multivalued string.

For example:

```
Before:     &1.Ø    [ABC^DEF^ABC]DEF]GHI^JKL^
Command:    U11E9
            MVA &1.2 &1.1
After:      &1.Ø    [ABC^DEF^ABC]DEF]DEF]GHI^JKL^
```

A similar MVA command in the above example executed without the user exit would result in no change to any fields.