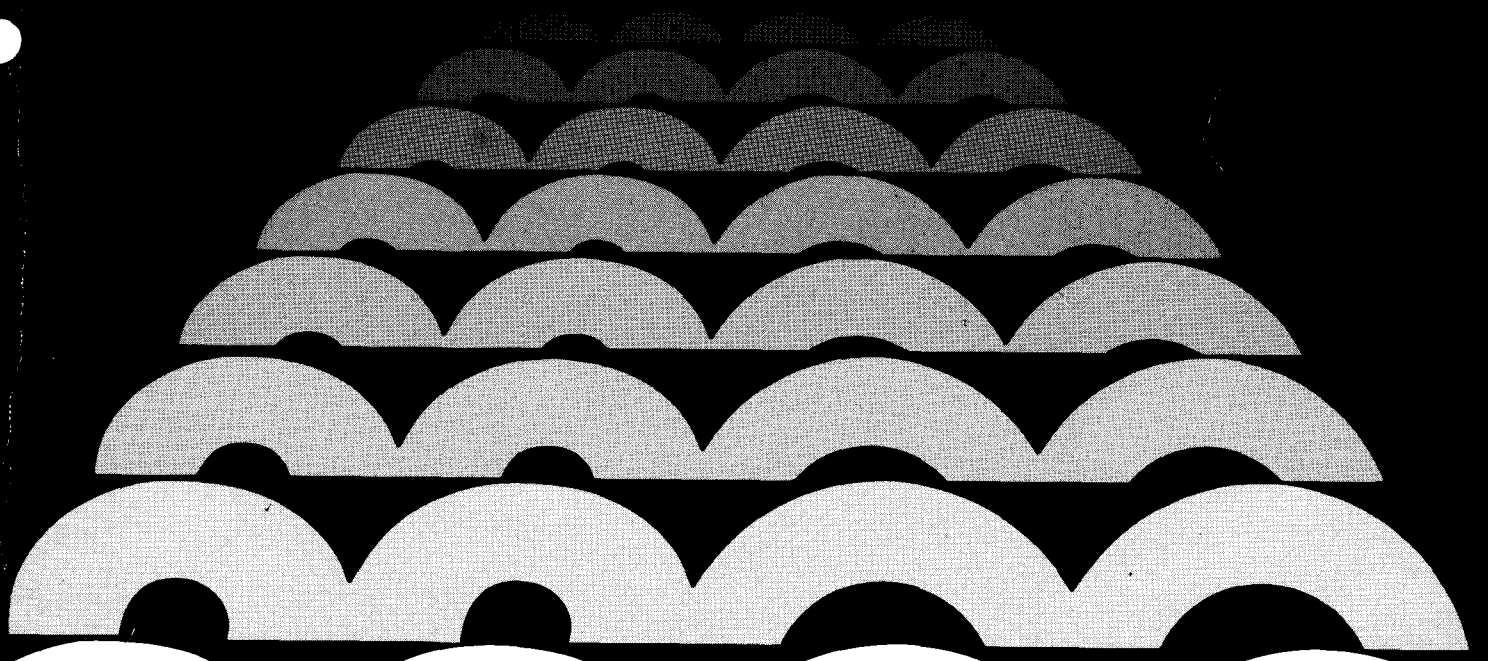
WANG

General I/O Instruction Set Reference Manual



2200

GENERAL I/O INSTRUCTION SET REFERENCE MANUAL

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Disclaimer of Warranties and Limitation of Liabilities

The staff of Wang Laboratories, Inc., has taken due care in preparing this manual; however, nothing contained herein modifies or alters in any way the standard terms and conditions of the Wang purchase agreement, lease agreement, or rental agreement by which this equipment was acquired, nor increases in any way Wang's liability to the customer. In no event shall Wang Laboratories, Inc., or its subsidiaries be liable for incidental or consequential damages in connection with or arising from the use of this manual or any programs contained herein.

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

PREFACE

This manual is designed for readers already familiar with a Wang system and its BASIC language.

The manual describes five BASIC language statements belonging to the General I/O Instruction Set, namely, $GIO, $IF ON, $TRAN, $PACK, and $UNPACK. The statements are standard in the System 2200T, available as part of Option 23 or 24 for the System 2200S, and available as Option 2 for the System 2200 B or C.

A copy of this manual is supplied with each system whose central processor includes the statements described herein.

CONTENTS

TABLES

CHARTS

FIGURES

EXAMPLES

CHAPTER 1

GENERAL I/O INSTRUCTION SET OVERVIEW

1.0 GENERAL DESCRIPTION

Five Wang BASIC language statements are described in this document. Collectively, the statements are called the General I/O Instruction Set. Individually, the statements are referred to by their mnemonic codes: $GIO, $IF ON, $TRAN, $PACK, and $UNPACK. See Table 1-1.

Table 1-1. General I/O Instruction Set

Statement	Description
$GIO	A general I/O statement designed to perform data input, data output, and I/O control operations with a programmable signal sequence. The statement supports I/O operations for Wang's Model 2209 Nine-Track Tape Drive, the Model 2228 Communications Controller, the Model 2227B Buffered Asynchronous Communications Controller, and is ideally suited to control I/O operations for specially interfaced non-Wang peripheral devices or instruments.
$IF ON	A statement designed to test the device-ready condition of a specified output device or test the data-ready condition of a specified input device and initiate a branch to a specified line number if a ready condition is sensed.
$TRAN	A statement designed to facilitate high-speed character code translations using a table lookup procedure or a character replacement procedure.
$PACK $UNPACK	Statements designed to facilitate data packing and unpacking (by fields, delimiters, or Wang's standard record format) between a specified alphanumeric array buffer and specified arguments in an argument list. The statements can be used to pack and unpack records, card images, et cetera.

The $GIO statement is unlike any other BASIC language I/O statement. By a technique similar to machine language programming, the $GIO statement provides a "General Input/Output" capability by which I/O operations can be custom-tailored to meet the particular signal-sequence requirements of a wide variety of I/O peripherals. Primarily, the statement is designed to support non-Wang peripheral devices and instruments specially interfaced to a Wang system via interface controllers such as the following:

1. The Model 2207A I/O Interface Controller (RS-232-C -- Selectable Baud)
2. The Model 2227 Asynchronous Telecommunications Controller
3. The Model 2227B Buffered Asynchronous Communications Controller
4. The Model 2250 I/O Interface Controller (8-Bit-Parallel)
5. The Model 2252A Scanning Input Interface Controller (BCD 1-to-10-Digit-Parallel)

Wang peripherals such as keyboards and CRT's can be controlled by $GIO statements, if desired; other peripherals (particularly cassette and disk drives) cannot be controlled with $GIO statements. If a Wang device (e.g., the Model 2209 Nine Track Tape Drive) requires $GIO statements for control, Wang Laboratories provides specific recommendations for programmed signal sequences or utility programs designed to control I/O performance.

The $IF ON statement, for input and output device scanning applications, is more versatile than the KEYIN statement which can scan only input devices. Also, the $IF ON statement offers advantages for applications involving multi-character input since the first character is not received separately as is the case in a KEYIN operation (which is excellent for one-byte handshake applications). The general form of the $IF ON statement is given in Section 3.1.

The $TRAN statement provides a high-speed data conversion capability. Conversion of a specified block of data is implemented via a table look-up or character replacement procedure. By storing conversion tables in alphanumeric arrays and specifying a particular array in a $TRAN statement, any desired code conversion algorithm can be programmed easily. Data can be translated before transmission to (or after reception from) standard or nonstandard peripheral devices in a Wang configuration. Characters not in ASCII code can be received and then converted, if necessary, to the ASCII character set used in Wang systems. Similarly, output data can be converted to any code used by a non-Wang system or device. If the optional parameter R is specified in a particular $TRAN statement, the character replacement procedure is implemented. Furthermore, if an optional mask is specified in a particular $TRAN statement, selected bits in each byte stored in a data block are deleted before the translation is accomplished. The masking capability is useful for data editing and parity bit removal. The general form of the $TRAN statement is given in Section 2.1.

The $PACK and $UNPACK statements provide the capability to scatter (unpack) data from a record or to gather (pack) data into a record and convert the data simultaneously. Data can be taken sequentially from specified arguments in an argument list, converted, and then packed into one alphanumeric array (record). Conversely, data can be taken sequentially from an alphanumeric array, converted, and then stored in specified arguments in an argument list (unpacked). In a packing or an unpacking operation, the alphanumeric array (the record) is treated as a contiguous group of characters; that is, element boundaries within the array are ignored. Each field in a record can be identified sequentially, according to type and length, or by defining special delimiters. Several different field types can be specified for one record. The general forms of the $PACK and $UNPACK statements are given in Section 2.2.

The $PACK and $UNPACK statements do not implement direct data transfer (sending or receiving) with respect to I/O devices; however, a $PACK or a $UNPACK statement can be combined with a $GIO statement (or some other Wang BASIC language I/O statement) in a two-step operation designed to transfer data between the CPU (Central Processing Unit) and an I/O device. The $PACK and $UNPACK statements are especially useful when processing or preparing input/output records in formats required by non-Wang peripherals. Also, the statements are of value in many applications when data of different lengths and precisions are being packed or unpacked for optimal utilization of memory and peripheral storage areas.

1.1 SYNTAX FOR THE GENERAL I/O INSTRUCTION SET

The following syntax is used throughout this manual to denote the components in a general form of a statement:

1. Upper case letters (A through Z) must be written in an actual statement exactly as shown in a general form.

2. Lower case letters or words represent items for which specific information is to be substituted in an actual statement.

3. Hyphens joining lower case words (or words and numbers) signify single items.

4. Vertically stacked items represent alternatives, only one of which is to be selected.

5. When stacked items are enclosed in braces,{}, one item must be specified. The braces are not included in an actual statement.

6. When single or stacked items are enclosed in brackets, [], the items are optional and may be omitted. The brackets are not included in an actual statement.

7. The following characters must be written as shown in the general form, unless otherwise indicated by a note:

comma	,
equal sign	=
parentheses	()
pound-sign	#
slash	/

8. When an ellipsis, ..., follows an item, the item may be repeated many times successively in an actual statement.

9. Blanks, inserted for readability in the general form, are not required. Wang systems ignore blanks in an actual statement unless the blanks are embedded in double quotation marks.

10. The sequential order of the components in a general form must be preserved when writing an actual statement.

1.2 SOME SPECIAL TERMS

The term "alpharg" is used to simplify the general forms of statements in the General I/O Instruction Set. An alpharg is defined as follows:

$$\text{alpharg} = \left\{\begin{array}{l}\text{alphanumeric variable} \\ \text{STR function} \\ \text{alpha array designator} \\ \text{alpha array designator <s,n>}\end{array}\right\}$$

where, in conformity with the syntax in Section 1.1, the braces signify alternatives (only one of which is to be selected).

The notation <s,n> is called an "alpha array modifier" (see Section 1.3). Also, the notation <s,m,e> is called an "alpha array modifier" for $GIO statements used in telecommunications applications (see Appendix A, Table A-13, Note 1).

When an array is designated as an alpharg in any one of the statements in the General I/O Instruction Set, the system treats the entire array as a single string of contiguous characters (bytes) and ignores element boundaries within the array. However, as usual, the array dimensions must be specified in a separate DIM statement to be executed prior to execution of any statement containing the particular array.

1.3 ALPHA ARRAY MODIFIERS

Both "s" and "n" in the alpha array modifier notation <s,n> are defined by the same alternatives for statements included in the General I/O Instruction Set. That is,

$$\left\{\begin{array}{l}s \\ n\end{array}\right\} = \left\{\begin{array}{l}\text{integer} \geq 1 \\ \text{mathematical expression} \\ \text{alphanumeric variable}\end{array}\right\}$$

Only one item is used to specify "s" and another item of the same or a different type to specify "n".

> Note:
>
> 1. When a mathematical expression (or a single numeric variable) is specified for "s" or "n", the system evaluates the expression and truncates the result to an integer which must be greater than or equal to one.
>
> 2. When an alphanumeric variable is specified for "s" or "n", the system treats the first two bytes of the variable as a 16-bit binary number and ignores any remaining bytes stored in the variable; the dimension of the variable must be at least two bytes long.

The s-parameter specifies the starting byte of the modified array. The n-parameter specifies the number of consecutive bytes to be used in an operation, beginning with the starting byte. For example, the notation:

```
A$()  <5,7>
```

specifies "seven consecutive bytes of the A$-array, beginning with the fifth byte".

Either "s" or "n" can be omitted when specifying an array modifier. The default value for "s" is 1 -- the value used by the system if a modifier is of the form <, n>.

The default value for "n" is used by the system if a modifier is of the form <s> . The default value of "n" is not a fixed value but depends upon two factors:

1. the array length (the maximum number of bytes in the array), and

2. the specified starting byte in the array modifier.

To evaluate the default value for n, use:

```
default n = m-s+1
```

where

m = the maximum number of bytes in the array, and
s = the specified starting byte.

When "n" is not specified in an alpha array modifier, the system begins with the s-byte and uses all remaining bytes in the array.

An alpha array modifier specifies a particular portion of an alphanumeric array in much the same way a string (STR) function specifies a particular portion of an alphanumeric variable. However, since the maximum length of an array can be as high as 30,000 bytes in some cases (while the maximum length of an alphanumeric variable is 64 bytes), the alpha array modifier provides a powerful programming technique when working with large array buffer storage areas.

> Note:
>
> 1. When an array modifier is used in one statement within a program, neither the size nor the shape of the array is altered in memory.
>
> 2. A special format <s,m,e> rather than <s,n> is required for an array modifier if used in a $GIO statement performing a telecommunications input operation. See the notes following Table A-13.

1.4 SAMPLE ALPHARGS

Sample alphargs, some including alpha array modifiers, are presented in this section -- assuming a program contains the following dimension statement:

```
10 DIM A$40, B$(3)60, C$5
```

Upon execution, Line 10 reserves the following storage locations in memory:

a) 40 bytes identified by the alphanumeric variable A$.

b) 180 bytes identified as follows:

   1) in three groups of 60 bytes each, with respect to element boundaries, when the specific elements B$(1), B$(2), and B$(3) are used in statements, or

   2) as a contiguous string of 180 bytes, without regard to element boundaries, if the alpha array designator B$() is used in a General I/O Instruction Set statement.

c) 5 bytes identified by the alphanumeric variable C$.

Now, if a program containing line 10 also contains any $GIO, $TRAN, $PACK, or $UNPACK statements using the sample alphargs shown in the first column of Table 1-2, the second column identifies the bytes defined by each alpharg.

Table 1-2. Sample Alphargs

Alpharg	Specified Bytes (Assuming Line 10 Dimensions)
A$	All 40 bytes of A$.
STR(A$,5,10)	Ten bytes of A$ beginning with the fifth byte.
B$(2)	All 60 bytes of element B$(2).
STR(B$(2),8)	Fifty-three bytes of B$(2) beginning with the eighth byte.
B$()	All 180 bytes of the B$-array.
B$() <68,53>	Fifty-three bytes of the B$-array beginning with the sixty-eighth byte. Same as STR(B$(2),8).
For J=1 B$() <J+1,178>	All but the first and last bytes of B$().
C$=HEX(0080) B$() <1,C$>	The first 128 bytes of B$(). Here HEX(0080)= 0000000010000000 is interpreted as a binary number to the base 2. Thus, the decimal value is $1x2^7=128$.
C$=HEX(0002) B$() <C$>	Here the decimal value of C$ is 2. The default value of n becomes 180-2+1=179. Therefore, the notation specifies all bytes of B$() except the first.
B$() <,50>	Fifty bytes of B$() beginning with the first (since default s=1).

1.5 STROBES, SIGNAL LEVELS, AND SIGNAL SEQUENCES

To custom-tailor I/O operations for a non-Wang peripheral device, the engineer responsible for interfacing the device to a Wang system (or a programmer) must determine appropriate microcommand sequences for $GIO statements. Specification of microcommand sequences is similar to machine language programming and can be made directly or indirectly in $GIO statements by techniques described in Chapter 3.

A single microcommand represents a fundamental operation usually consisting of several steps. With a sequence of microcommands, a complete I/O operation can be constructed similar to a standard BASIC language I/O operation such as INPUT, PRINT, DATALOAD BT, et cetera.

Each microcommand is represented by a four-hexdigit-code (two bytes). The first pair of hexdigits usually identifies the type of operation and the "signal sequence" to be executed. The second pair of hexdigits usually specifies particular information; for example, the character to be output or the register containing the character to be output. Similarly, for an input operation, the second byte of the microcommand may specify the register for storage of an incoming character.

Seventeen categories of microcommands are available (see Table 3-1). Each of the seventeen microcommand categories contains subcategories presented in Appendix A, where the signal sequence corresponding to each microcommand is described using mnemonics and special notation. The notation and mnemonics are defined using the terms "strobe" and "level" (or "signal level") -- terms not ordinarily encountered by applications programmers using documentation for a Wang system; however, the terms are familiar to electronics engineers for whom the $GIO statement is geared.

Users of configurations containing only standard Wang peripheral devices do not need to know the strobes and signal levels exchanged between the CPU, a device controller board, and a particular peripheral device during execution of an I/O operation. A major feature of any high-level programming language, such as BASIC, is its simplified, conversational mode programming capability. Therefore, each I/O statement, except $GIO, executes a built-in signal sequence.

Actually, the $GIO statement operates within the framework of the conversational mode BASIC language yet permits the programming of customized signal sequences required by a wide variety of non-Wang I/O devices. A microcommand sequence, whether specified directly or indirectly in a $GIO statement, replaces the built-in signal sequence implicit in other I/O statements.

The description of the $GIO statement in Chapter 3 includes many references to signal sequences, signal levels, and strobes. Readers of Chapter 3 who do not have an electronics background should understand the syntax and fundamental operations of the $GIO statement without becoming concerned about any lack of understanding of signal levels and strobes. In general, the engineer responsible for interfacing a non-Wang I/O device to a Wang system is the person best qualified to determine the microcommand sequences needed to control a particular device and to know whether the installation of a device is compatible with an exchange of the particular strobes and levels included in each microcommand. Furthermore, when $GIO statements are needed to control a Wang I/O device, Wang Laboratories prescribes the microcommand sequence needed to implement each operation for the device.

The descriptions of the $TRAN, $PACK, and $UNPACK statements in Chapter 2 do not refer to strobes and signal levels since these data conversion statements do not implement direct data transfer between the CPU and any I/O devices.

CHAPTER 2

DATA CONVERSION USING $TRAN, $PACK, OR $UNPACK

2.0 INTRODUCTION

Three statements in the General I/O Instruction Set provide data conversion capabilities for Wang systems. The statements are $TRAN, $PACK, and $UNPACK.

The general form of the $TRAN statement represents two forms, depending upon whether the parameter R is or is not included in an actual statement. Upon execution, a $TRAN statement implements a code conversion operation on a specified block of data (the first argument) using a table or list defining the translation (the second argument). If an optional mask is specified, each data byte is masked by a logical AND operation before the translation. If the optional "replace" parameter R is specified, only selected data characters are translated.

The general form of the $PACK statement represents three forms, depending upon whether the parameter D, the parameter F, or neither D nor F is included in an actual statement. Upon execution, a $PACK statement implements a packing and/or data conversion operation which sequentially transfers data from each argument in an argument list and stores the data in a specified buffer, according to a prescribed format (a delimiter format, a field format, or the standard Wang record format).

The general form of the $UNPACK statement also represents three forms, depending upon whether the parameter D, the parameter F, or neither D nor F is included in an actual statement. Upon execution, a $UNPACK statement implements an unpacking and/or data conversion operation which separates data stored in a specified buffer, according to a prescribed format, and sequentially transfers the data to each argument in an argument list.

Using the field form of a $PACK or $UNPACK statement, numeric data can be converted into or from one or more of the following field types during execution of the packing or unpacking operation:

a) an ASCII free format,
b) an ASCII implied decimal format,
c) an IBM display format,
d) an IBM USASCII-8 format, and/or
e) an IBM packed decimal format.

Furthermore, the field width can be specified also.

General Form:

$TRAN (arg-1, arg-2) [hh] [R]

where:

arg-1 = An alpharg representing a block of data to be translated via a table-lookup or character replacement procedure.

arg-2 = An alpharg representing a table (or a list, if R is specified) of characters defining the translation.

hh = A pair of hexdigits representing a mask (optional). If specified, a mask defines the bits to be "deleted" (that is, replaced by zeros) in each arg-1 byte by executing a logical AND operation using the next successive arg-1 byte and the mask before translating the resulting character.

R = The "replace" parameter (optional). If specified, R indicates that only those masked or original arg-1 characters which match an even-numbered character in the arg-2 list are to be replaced by the preceding odd-numbered character in the list; if no match is found, the masked or original byte is returned to arg-1.

alpharg= { alphanumeric-variable | STR-function | alpha-array-designator | alpha-array-designator <s,n> }

2.1 THE $TRAN STATEMENT

The $TRAN statement is designed for code conversion operations, such as the conversion of EBCDIC codes to ASCII codes. Applications include byte substitution operations needed for hexadecimal code conversions, character verification, or initialization operations -- to name a few. Many other applications are possible with user ingenuity.

The general form of the $TRAN statement represents two different data conversion operations. The desired procedure is identified by the inclusion or omission of the parameter R.

The Character Replacement Procedure for $TRAN

If the R parameter is specified in an actual $TRAN statement, the arg-2 component of the statement must represent a list of "to-from" translation characters. Since the list of "to-from" characters cannot be specified directly in the $TRAN statement, the list must be stored in memory by an input operation or by execution of another statement prior to execution of the $TRAN statement, as shown in Example 2-1. The list must consist of pairs of bytes to be interpreted as follows:

a) The second byte in each pair of bytes is a "translate from" character; that is, the even-numbered bytes in an arg-2 list define the set of characters to be translated.

b) The first byte in each pair of bytes is a "translate to" character corresponding to a particular "from" character; that is, the odd-numbered bytes in an arg-2 list define the replacement characters for a translation operation.

c) The last pair of "to-from" characters must be followed by at least one pair of space characters, where each space character is denoted by a HEX(20) code. (Unpredictable results may occur if an arg-2 list is not terminated by a pair of space characters.)

During execution of a $TRAN statement with R specified, the following events occur:

1. The next successive arg-1 byte is masked, if a mask is specified.

2. The masked (or original) byte is compared with each successive "translate from" character in the arg-2 list.

3. As soon as a match is found, the corresponding replacement character (the preceding "translate to" character) is returned to the arg-1 byte position currently being processed.

4. If no match is found, the masked (or original) byte is returned to the arg-1 byte position currently being processed.

The Table Lookup Procedure for $TRAN

If the R parameter is omitted in an actual $TRAN statement, the arg-2 component of the statement must specify the translation table as a set of consecutive characters. Since the translation table cannot be specified directly in a $TRAN statement, the table must be stored in memory by an input operation or by other statements prior to execution of the $TRAN statement, as shown in Examples 2-2 through 2-4.

The sequential position of each character in a translation table is extremely important when R is not specified, since the table lookup procedure is a "displacement" procedure equivalent to the following steps:

1. The next successive arg-1 byte is masked, if a mask is specified.

2. The equivalent decimal system value of the masked (or original) byte is calculated, e.g., in the ASCII character set used by Wang systems, an unmasked uppercase G is represented by $(47)_{16} = (01000111)_2 = (71)_{10}$.

3. The decimal system value from Step 2 becomes the "displacement" used to locate the proper translation character in the arg-2 table. The displacement can be defined as the movement of an imaginary pointer which points to the first position in the table for a zero displacement and moves to the (m+1)th position for a displacement of "m". For example, since the decimal system value of an ASCII G is 71, its translation character must appear in position 72 in an arg-2 table if no mask is used in the translation operation. (See Example 2-2.)

4. The character found in the arg-2 table is stored in the arg-1 byte position currently being processed.

5. If the translation table is too short for the required displacement in Step 3, the masked (or original) byte is returned to the arg-1 position currently being processed.

Constructing a Table for a $TRAN Table-lookup Procedure

The System 2200 uses an 8-bit code to represent its character set consisting of three types of characters:

1. Graphic (printable) characters, e.g., A, a, 1, 7, ?, #.

2. Control (nonprintable) characters, e.g., carriage return, line feed, backspace.

3. Text atoms (BASIC language words), e.g., DIM, GOSUB, INPUT, stored in memory as single 8-bit codes whether entered from the keyboard by a single "keyword" keystroke or by multi-keystrokes (one for each character in the word).

The code for graphic and control characters is equivalent to 7-bit ASCII (American Standard Code for Information Interchange) plus a high-order "0" eighth bit. See Appendix C. The high-order eighth bit for text atoms is "1".

In hexadecimal notation, where two hexdigits equal 8 bits, the System 2200 graphic and control characters are represented by codes whose first hexdigit is less than eight, and text atoms are represented by codes whose first hexdigit is greater than or equal to eight. For example, A = $(41)_{16}$, 7 = $(37)_{16}$, z = $(7A)_{16}$, RUN = $(82)_{16}$, REWIND = $(A9)_{16}$.

Some computer systems represent graphic and control characters by an 8-bit code called EBCDIC (Extended Binary Coded Decimal Interchange Code). The ASCII and EBCDIC codes differ greatly with respect to the assignment of characters to particular binary values. See Charts 2-1 and 2-2.

In Charts 2-1 and 2-2, the character assignment squares represent 8-bit (one-byte) codes arranged sequentially in ascending order, according to the equivalent decimal system value shown in the lower right corner of each square. There are 128 character assignment positions in the ASCII chart with decimal values from 0 to 127 inclusive, and 256 positions in the EBCDIC chart with decimal values from 0 to 255 inclusive. When an assigned character appears in a particular position, the binary value is found by reading the four

high-order bits in the left margin of the chart and reading the four low-order bits at the top of the chart. Similarly, the hexadecimal value is found by reading the high- and low-order hexdigits in the left and top margins, respectively. For example, in Chart 2-1, E = $(01000101)_2$ = $(45)_{16}$= $(69)_{10}$.

Check the following characters in Charts 2-1 and 2-2:

Character	ASCII Code	EBCDIC Code
#	$(23)_{16} = (35)_{10}$	$(7B)_{16} = (123)_{10}$
A	$(41)_{16} = (65)_{10}$	$(C1)_{16} = (193)_{10}$
d	$(64)_{16} = (100)_{10}$	$(84)_{16} = (132)_{10}$
7	$(37)_{16} = (55)_{10}$	$(F7)_{16} = (247)_{10}$
space	$(20)_{16} = (32)_{10}$	$(40)_{16} = (64)_{10}$

After several character-by-character comparisons in the charts, one concludes that no simple functional relationship can be used to convert EBCDIC to ASCII code or vice versa. Hence, the technique of constructing a translation table should be understood.

Remember (or review) the description of the $TRAN table lookup procedure. Then, keep in mind one very important fact -- when a character is being translated, its decimal value becomes the displacement by which the appropriate translation character is found (the displacement is measured from the first position in the table). Therefore, a translation table must contain the "after translation characters" arranged in a sequence corresponding to the decimal values of the "before translation characters."

For simplicity, consider the following application. A System 2200 receives numeric-only data in EBCDIC code (with no signs or decimal points). The data is stored in a large array, B$(). A $TRAN statement converts the data from EBCDIC to ASCII code prior to execution of other data processing procedures. How is the translation table programmed?

Actually, one of several different translation tables can be used for such an application, as shown in the following summary. The summary demonstrates how storage requirements for a translation table can be reduced dramatically by using a mask in a $TRAN statement, when appropriate. Consider the following facts:

1. In Chart 2-2, the digits 0 through 9 are represented in EBCDIC by $(F0)_{16}$ through $(F9)_{16}$, i.e., by codes having decimal values ranging from 240 through 249.

2. In Chart 2-1, the digits 0 through 9 are represented in ASCII by $(30)_{16}$ through $(39)_{16}$.

3. Combining the facts in (1) and (2), one concludes that the ASCII code $(30)_{16}$ must occupy position 241 in an EBCDIC to ASCII translation table because the $TRAN table lookup procedure uses a displacement equal to 240 every time it encounters an EBCDIC code

$(F0)_{16}$ in the array B$(). Furthermore, the translation table must have a minimum of 250 bytes to accommodate displacements as large as 249 when $(F9)_{16}$ codes are encountered in B$(). Since an array is treated as a set of contiguous bytes by a $TRAN statement, the translation table can be defined by a 25-element array with 10 bytes per element. The first 240 bytes of the table can be filled with null characters $(00)_{16}$ and the last 10 bytes with the appropriate ASCII characters, as illustrated by the following sequence:

```
10 DIM T$(25)10
20 INIT (00) T$()
30 T$(25) = HEX(30313233343536373839)
```

Then, assuming the EBCDIC data is stored in the array B$() prior to Line 90 in the program, the following statement can be used to translate the data into ASCII code:

```
90 $TRAN (B$(), T$())
```

However, a more efficient translation table can be developed, as shown in (4).

4. If the mask $(0F)_{16} = (00001111)_2$ is used in the $TRAN statement, each of the codes $(F0)_{16}$ through $(F9)_{16}$ becomes one of the codes $(00)_{16}$ through $(09)_{16}$ before the displacement lookup procedure is implemented. Since the masked codes $(00)_{16}$ through $(09)_{16}$ have decimal values ranging from 0 through 9, a ten-byte translation table is sufficient for this numeric-only application as illustrated in the following sequence:

```
10 DIM T$10
20 T$ = HEX(30313233343536373839)
.
.
.
90 $TRAN (B$(),T$) 0F
```

5. In some cases, a literal string can be used instead of a HEX function to define a translation table, e.g., the statement

```
20 T$ = "0123456789"
```

when executed by the System 2200, automatically stores the ASCII codes HEX(30) through HEX(39) in the variable T$.

Now consider an application not restricted to conversion of numeric-only EDCDIC data. Assume the System 2200 receives data in EBCDIC code, including both control and graphic characters shown in Chart 2-2. If the data is stored in the array X$(), the following statement can be used to translate the data:

```
$TRAN (X$(),T2$())
```

where T2$() is defined by Lines 100 through 260 in Chart 2-3.

Next, consider an application requiring ASCII data stored in the array Y$() to be translated into EBCDIC data prior to its transmission from a System 2200 to another system. For this application, the following statement can be used:

```
$TRAN (Y$(),T1$())
```

where T1$() is defined by Lines 15 through 90 in Chart 2-3.

Observe in Chart 2-3 that the ASCII to EBCDIC translation table T1$() is constructed by arranging EBCDIC codes in a sequence corresponding to ASCII positions. Similarly, the EBCDIC to ASCII translation table T2$() is constructed by arranging ASCII codes in a sequence corresponding to EBCDIC positions.

Examples 2-2 and 2-3 demonstrate the use of translation tables where ASCII codes are converted into other ASCII codes.

CHART 2-1. ASCII Code*

High-order: 4-bits / Low-order: 4-bits hex-digit		0_{0_0}	0_{0_1}	0_{1_0}	0_{1_1}	1_{0_0}	1_{0_1}	1_{1_0}	1_{1_1}
		0	1	2	3	4	5	6	7
0_{0_0}	0	NUL 0	SOH 1	STX 2	ETX 3	EOT 4	ENQ 5	ACK 6	BEL 7
0_{0_1}	1	DLE 16	DC1 17	DC2 18	DC3 19	DC4 20	NAK 21	SYN 22	ETB 23
0_{1_0}	2	Space 32	! 33	" 34	# 35	$ 36	% 37	& 38	(apos.) ' 39
0_{1_1}	3	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55
1_{0_0}	4	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71
1_{0_1}	5	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87
1_{1_0}	6	grave accent ` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103
1_{1_1}	7	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119

		1_{0_0}	1_{0_1}	1_{1_0}	1_{1_1}
		8	9	A	B	C	D	E	F
0_{0_0}	0	BS 8	HT 9	LF 10	VT 11	FF 12	CR 13	SO 14	SI 15
0_{0_1}	1	CAN 24	EM 25	SUB 26	ESC 27	FS 28	GS 29	RS 30	US 31
0_{1_0}	2	(40	) 41	* 42	+ 43	(comma) , 44	(dash) — 45	(period) . 46	/ 47
0_{1_1}	3	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
1_{0_0}	4	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
1_{0_1}	5	X 88	Y 89	Z 90	[91	\ 92	] 93	↑ 94	(underline) _ 95
1_{1_0}	6	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
1_{1_1}	7	x 120	y 121	z 122	{ 123	¦ 124	} 125	~ 126	DEL 127

*Numbers in the lower right corner of each box represent the decimal equivalent of the binary and the hexadecimal code for the character shown in the box, e.g., A = $(41)_{16} = (01000001)_2 = (65)_{10}$.

CHART 2-2. EBCDIC Code*

High-order: 4-bits / Low-order: 4-bits		0_{0_0}	0_{0_1}	0_{1_0}	0_{1_1}	0_{0_0}	0_{1_0}	0_{1_1}	0_{1_1}	1_{0_0}	1_{0_1}	1_{1_0}	1_{0_1}	1_{1_0}	1_{1_1}	1_{1_0}	1_{1_1}
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL 0	SOH 1	STX 2	ETX 3	PF 4	HT 5	LC 6	DEL 7	8	RLF 9	SMM 10	VT 11	FF 12	CR 13	SO 14	SI 15
0001	1	DLE 16	DC1 17	DC2 18	DC3 19	RES 20	NL 21	BS 22	IL 23	CAN 24	EM 25	CC 26	27	IFS 28	IGS 29	IRS 30	IUS 31
0010	2	DS 32	SOS 33	FS 34	35	BYP 36	LF 37	ETB 38	ESC 39	40	41	SM 42	43	44	ENQ 45	ACK 46	BEL 47
0011	3	48	49	SYN 50	51	PN 52	RS 53	UC 54	EOT 55	56	57	58	59	DC4 60	NAK 61	62	SUB 63
0100	4	Space 64	65	66	67	68	69	70	71	72	73	¢ 74	(period) . 75	< 76	(77	+ 78	| 79
0101	5	& 80	81	82	83	84	85	86	87	88	89	! 90	$ 91	* 92	) 93	; 94	¬ 95
0110	6	(dash) — 96	/ 97	98	99	100	101	102	103	104	105	¦ 106	(comma) , 107	% 108	(underline) _ 109	> 110	? 111
0111	7	112	113	114	115	116	117	118	119	120	grave accent ` 121	: 122	# 123	@ 124	(apos.) ' 125	= 126	" 127
1000	8	128	a 129	b 130	c 131	d 132	e 133	f 134	g 135	h 136	i 137	138	139	140	141	142	143
1001	9	144	j 145	k 146	l 147	m 148	n 149	o 150	p 151	q 152	r 153	154	155	156	157	158	159
1010	A	160	~ 161	s 162	t 163	u 164	v 165	w 166	x 167	y 168	z 169	170	171	172	173	174	175
1011	B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
1100	C	{ 192	A 193	B 194	C 195	D 196	E 197	F 198	G 199	H 200	I 201	202	203	204	205	206	207
1101	D	} 208	J 209	K 210	L 211	M 212	N 213	O 214	P 215	Q 216	R 217	218	219	220	221	222	223
1110	E	\ 224	225	S 226	T 227	U 228	V 229	W 230	X 231	Y 232	Z 233	234	235	236	237	238	239
1111	F	0 240	1 241	2 242	3 243	4 244	5 245	6 246	7 247	8 248	9 249	250	251	252	253	254	255

*Numbers in the lower right corner of each box represent the decimal equivalent of the binary and the hexadecimal code for the character shown in the box, e.g., A = $(C1)_{16} = (11000001)_2 = (193)_{10}$.

Chart 2-3. Translation Tables for ASCII to EBCDIC Conversion and Vice Versa

```
10 DIM T1$(8)16, T2$(16)16
15 REM USE T1$() IN $TRAN FOR ASCII TO EBCDIC CONVERSION
20 T1$(1)=HEX(00010203372D2E2F1605250B0C0D0E0F)
30 T1$(2)=HEX(101112133C3D322618193F2722003500)
40 T1$(3)=HEX(405A7F7B5B6C507D4D5D5C4E6B604B61)
50 T1$(4)=HEX(F0F1F2F3F4F5F6F7F8F97A5E4C7E6E6F)
60 T1$(5)=HEX(7CC1C2C3C4C5C6C7C8C9D1D2D3D4D5D6)
70 T1$(6)=HEX(D7D8D9E2E3E4E5E6E7E8E900E000006D)
80 T1$(7)=HEX(79818283848586878889919293949596)
90 T1$(8)=HEX(979899A2A3A4A5A6A7A8A9C06AD0A107)
100 REM USE T2$() IN $TRAN FOR EBCDIC TO ASCII CONVERSION
110 T2$(1)=HEX(000102030009007F0000000B0C0D0E0F)
120 T2$(2)=HEX(10111213000008001819000000000000)
130 T2$(3)=HEX(00001C00000A171B0000000000050607)
140 T2$(4)=HEX(0000160000 1E00040000000001415001A)
150 T2$(5)=HEX(2000000000000000000002E3C282B00)
160 T2$(6)=HEX(260000000000000000021242A293B00)
170 T2$(7)=HEX(2D2F0000000000000007C2C255F3E3F)
180 T2$(8)=HEX(00000000000000000603A2340273D22)
190 T2$(9)=HEX(006162636465666768690000000000000)
200 T2$(10)=HEX(006A6B6C6D6E6F707172000000000000)
210 T2$(11)=HEX(007E737475767778797A000000000000)
220 T2$(12)=HEX(00000000000000000000000000000000)
230 T2$(13)=HEX(7B41424344454647484900000000000)
240 T2$(14)=HEX(7D4A4B4C4D4E4F505152000000000000)
250 T2$(15)=HEX(5C00535455565758595A000000000000)
260 T2$(16)=HEX(30313233343536373839000000000000)
```

Example 2-1. Character Replacement Using $TRAN

Assuming data has been stored in A$ by program logic executed prior to Line 100, the following sequence replaces each HEX(11) code in A$ by a HEX(0D) code and replaces each HEX(07) code by a HEX(00) code. The list of codes defining the conversion is assigned to L$ in Line 100. As required, the list is terminated by a pair of space characters.

```
100 L$=HEX(0D1100072020)
110 $TRAN (A$, L$) R
```

Example 2-2. Extracting the Low-order Hexdigit from ASCII Codes

The following program sequence "extracts" the low-order hexdigit from the hexadecimal notation for each ASCII character stored in X$ by employing a technique based on these considerations:

a) In hexadecimal notation, each character is of the form $(h_1 h_2)_{16}$, where h_1 represents the high-order hexdigit and h_2 represents the low-order hexdigit.

b) If a character of the form $(h_1 h_2)_{16}$ is ANDed with the character $(0F)_{16} = (00001111)_2$, the result is a character of the form $(0h_2)_{16}$ whose equivalent decimal system value is one of the sixteen integers in the range from 0 through 15 inclusive, depending upon which hexdigit h_2 actually corresponds to the original character. Thus, only sixteen unique characters are required in a translation table. Furthermore, these characters can be the sixteen symbols used in the hexadecimal number system, i.e., 0 through 9 and A through F.

```
10 T$="0123456789ABCDEF"
20 X$="*GO#"
30 HEXPRINT X$
40 $TRAN (X$, T$) 0F
50 PRINT X$

:RUN
2A474F23
A7F3
```

In Line 10, a table containing the characters corresponding to the sixteen hexadecimal symbols (called hexdigits) is assigned to T$. In Line 20, a sample set of ASCII characters is stored in X$ to demonstrate the effect of the actual $TRAN operation when the program is run. Upon execution, Line 30 prints the hexadecimal notation for each byte stored in X$ prior to the translation operations. In Line 40, the mask $(0F)_{16} = (00001111)_2$ causes the four high-order bits in a byte to be replaced by zeros before the displacement for the table lookup procedure is calculated.

Originally, the first byte of X$ is an asterisk character whose code is $(2A)_{16} = (00101010)_2$; after the logical AND operation with the mask $(0F)_{16}$, the result is $(00001010)_2$. The decimal equivalent of the resulting code is 10 which becomes the displacement for the table lookup procedure. The displacement moves the invisible pointer from the first to the eleventh position in the T$ table, where the character A is located; hence, the character A replaces the asterisk as the first byte in X$. The process continues with the second byte of X$ which is translated from the character "G" = $(47)_{16}$ to the character "7" by the same procedure.

Example 2-3. Using $TRAN When Testing Single Character Input

Assume an operator inputs a single character to indicate a desired procedure, e.g., S = save, L = load, E = edit, I = insert, D = delete. The following sequence tests the input character and branches to the corresponding procedure.

```
100 DIM A$1, T$12
110 T$="1S2L3E4I5D"
120 INPUT "OPTION DESIRED--S,L,E,I,D",A$
130 $TRAN (A$,T$)R
140 ON VAL(A$) - 48 GOTO 500,600,700,800,900
150 PRINT "INVALID. REENTER"
160 GOTO 120
```

2.2 THE $PACK AND $UNPACK STATEMENTS

As their names imply, the $PACK and $UNPACK statements implement inverse operations. The operations are completely independent of each other (one operation can appear in a program without the other operation). However, a general programming practice is recommended for those cases where data previously packed by a $PACK statement is subsequently unpacked by a $UNPACK statement; in such cases, the $UNPACK argument list should be identical in format to the $PACK argument list -- the formats are identical if the number, type, and sequential order of the variables match (names may differ).

A $PACK statement sequentially transfers data from one or more arguments in an argument list and stores the data in a specified buffer according to a prescribed format, i.e., a delimiter format (if the parameter D is specified), a field format (if the parameter F is specified), or the standard record format (if neither D nor F is specified). A $PACK operation can be particularly useful for gathering and formatting data in a large buffer area prior to outputting the data to a peripheral device capable of receiving large quantities of data rapidly. (The actual output operation must be controlled by another BASIC language statement such as DATASAVE BT or a customized $GIO output operation.)

A $UNPACK statement separates data in a formatted buffer (according to a prescribed delimiter, field, or the standard record format) and sequentially transfers the data to one or more arguments in an argument list. A $UNPACK operation can be particularly useful for separating and distributing data previously received from a peripheral device and stored in a large buffer area by a customized $GIO input operation or a DATALOAD BT operation.

The syntax for all three forms of the $PACK and $UNPACK statements is given in this section. The delimiter form for both statements is described in detail, including examples, in Section 2.3; the field form is described in Section 2.4; the standard Wang record form is described in Section 2.5.

General Forms:

$PACK [(D=alpha) (F=alpha)] alpharg FROM argument-list

$UNPACK [(D=alpha) (F=alpha)] alpharg TO argument-list

where:

D = the delimiter format parameter.

F = the field format parameter.

alpha = {alphanumeric-variable | alpha-array-designator} = the format specification variable.

alpharg = {alphanumeric-variable | STR-function | alpha-array-designator | alpha-array-designator <s,n>} = the data buffer (the "record").

argument-list = one or more arguments, separated by commas, with

each argument = {numeric-variable | numeric-array-designator | alphanumeric-variable | STR-function | alpha-array-designator}

1. If neither D nor F is specified in an actual statement, the standard Wang record format is implied.

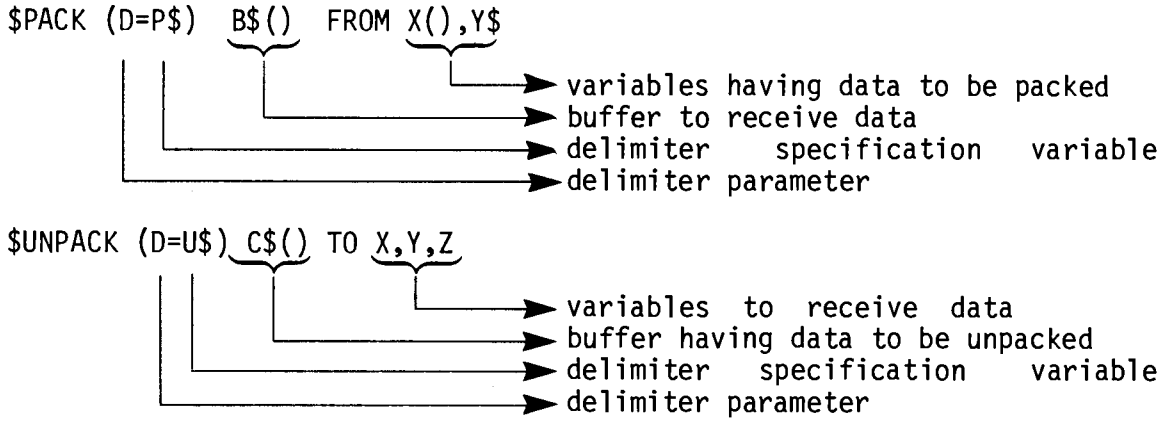
2. If the specified buffer for a $PACK or $UNPACK operation is an alphanumeric array with or without an alpha-array-modifier, the dimensions of the array must be specified in a DIM or COM statement earlier in the program logic. (Also, the dimensions of any arrays included in the argument-list must be specified.)

3. The data buffer size can be a critical consideration. For $PACK operations, the buffer size must not be too small to hold all the data from the argument list plus delimiter characters, if any, or control bytes (used in the standard record format). On the other hand, for some delimiter form $UNPACK operations, buffer size is not a critical consideration (see Section 2.3); for other forms of the $UNPACK operation, buffer size can be critical.

4. See Section 1.3 for a discussion of the alpha array modifier <s,n> .

2.3 THE DELIMITER FORM FOR $PACK AND $UNPACK OPERATIONS

When denoting the delimiter form for a $PACK or $UNPACK operation, a delimiter specification variable must be assigned to the parameter D, as shown in the following examples of valid syntax.



The Delimiter Specification Variable

Prior to execution of a delimiter form $PACK or $UNPACK statement, a value must be stored in the delimiter specification variable by execution of another BASIC language statement (see Examples 2-4 through 2-6). Since the delimiter specification value must be a two-byte code of the form shown in Table 2-1, a minimum of two bytes must be stored in the designated delimiter specification variable; any additional bytes are ignored by the system.

Observe in Table 2-1 that the second byte in a delimiter specification code defines the delimiter character. Only one character can be defined as a delimiter for a particular $PACK or $UNPACK operation, e.g., the code $(002C)_{16}$ specifies a comma as the delimiter since the second byte is $(2C)_{16}$ which represents a comma. See Appendix C. The specified delimiter is inserted automatically between sequential data values during a $PACK operation, or is recognized as the boundary for a data value during a $UNPACK operation.

As shown in Table 2-1, the first byte in a delimiter specification code must be $(00)_{16}$, $(01)_{16}$, $(02)_{16}$ or $(03)_{16}$. For a $PACK operation, any one of the four codes is equally suitable since the first byte is not utilized by the system when executing a $PACK statement but is needed to satisfy the syntax. On the other hand, the first byte determines the particular processing procedure used by the system when executing a $UNPACK statement.

Table 2-1. Valid Delimiter Specification Codes in Hexadecimal Notation

Two-byte Code*	Effect of First Byte for $PACK	Effect of First Byte for $UNPACK Operations
00hh	none	1. Display error message when data is insufficient for the next variable in the list. 2. Skip a variable in the list for each successive delimiter in the record.
01hh	none	1. Ignore remaining variables when data is insufficient for the next argument in the list. 2. Skip a variable in the list for each successive delimiter in the record.
02hh	none	1. Display error message when data is insufficient for the next variable in the list. 2. Ignore successive delimiters in the record.
03hh	none	1. Ignore remaining variables when data is insufficient for the next variable in the list. 2. Ignore successive delimiters in the record.

*hh = a two-hexdigit-code defining the actual delimiter character for a particular $PACK or $UNPACK operation, e.g., hh=2C denotes a comma as the delimiter.

Features of a Delimiter Form $PACK Operation

1. A specified portion of a buffer is packed if an alpha array modifier is included in a $PACK statement.

2. The buffer for a $PACK operation is not cleared before the operation begins. In applications where undesirable data and delimiters may arise from reuse of a buffer, initialization of the buffer by an INIT statement may be advisable prior to execution of the $PACK statement.

3. Data from each argument in the argument list is packed in the buffer sequentially with insertion of the specified delimiter between data corresponding to successive variables. See Figure 2-1.

4. If the argument list contains an array, a delimiter is inserted between data corresponding to successive elements. Data from a two-dimensional array are packed element by element, row by row sequentially.

5. No delimiter follows the last data value packed in the buffer.

6. Data from an alphanumeric variable is packed as shown in Figure 2-2. The number of bytes packed, including any trailing spaces, equals the dimensioned length of the variable (default value=16).

7. Data from a numeric variable is packed in a fixed point or a floating point format, depending on the magnitude of the value, as shown in Figure 2-3. All sign codes, the character E which denotes a floating point value, and the digits 0 through 9 are packed in the buffer using 8-bit ASCII codes shown in Appendix C. A fixed point value may occupy as few as two bytes or up to 15 bytes (leading and trailing zeros are omitted). A floating point value is packed in scientific notation occupying 15 bytes.

8. If the buffer is too small to hold the data corresponding to the next variable in the argument list, an error message (Code 97) is displayed.

9. If the buffer is large enough to hold all data values and delimiters, a delimiter form $PACK operation terminates after data from the last variable is stored in the buffer.

Alternative Processing Procedures for Delimiter Form $UNPACK Statements

The unpacking procedures listed in Table 2-1 represent all possible combinations of two alternative processing procedures for each of two buffer conditions which might arise during a $UNPACK operation.

For applications where the buffer being unpacked may contain insufficient data to supply the next receiving variable in the argument list, the following alternatives are available:

1. Display an error message, thereby stopping program execution until suitable program modifications are made.

2. Ignore the remaining receiving variables (letting them retain their current values) and continue program execution with the next statement.

For applications where the buffer being unpacked may contain successive delimiters between a pair of data values, the following alternatives are available:

1. Ignore the successive delimiters and transfer the next data value to the receiving variable currently being processed.

2. Skip one receiving variable in the argument list for each successive delimiter (letting skipped variables retain their current values).

Since both buffer conditions (successive delimiters and insufficient data) may occur during execution of a particular $UNPACK statement, a programmer should refer to Table 2-1 to determine which delimiter specification code corresponds to the desired processing procedure for the application being programmed.

Features of a Delimiter Form $UNPACK Operation

1. A specified portion of a buffer is unpacked if an alpha array modifier is included in a $UNPACK statement.

2. In general, when a buffer is being unpacked, data up to (but not including) the designated delimiter is stored in the next receiving variable in the argument list. However, if the receiving variable is numeric and the data is not a legal representation of a BASIC number, an error message (Code 20) is displayed.

3. When a delimiter is followed by one or more additional delimiters as shown schematically in Figure 2-4, the system ignores the successive delimiters if the delimiter specification code for the operation is of the form $(02hh)_{16}$ or $(03hh)_{16}$, or the system skips one receiving variable for each successive delimiter if the code is of the form $(00hh)_{16}$ or $(01hh)_{16}$. See Table 2-1.

4. When the buffer contains insufficient data to supply the next receiving variable, the system ignores all remaining receiving variables if the delimiter specification code is of the form $(01hh)_{16}$ or $(03hh)_{16}$, or the system displays an error message (Code 97) if the code is of the form $(00hh)_{16}$ or $(02hh)_{16}$.

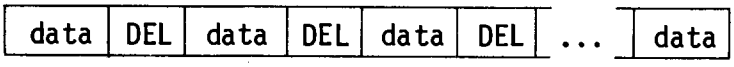
5. Data bounded by a delimiter is acceptable for unpacking to an alphanumeric receiving variable if formatted as shown in Figure 2-5. Any 8-bit code except the specified delimiter code can be unpacked; however, the number of bytes actually transferred equals either the number of bytes in the buffer up to the delimiter or the dimensioned length of the receiving variable, whichever is smaller.

6. Data bounded by a delimiter is acceptable for unpacking to a numeric receiving variable if formatted as shown in Figure 2-6. Any numeric value in the range from 10^{-99} to 10^{+100} (exclusive of the upper end point of the range) can be unpacked for storage in a numeric variable if its format duplicates any form acceptable by the system for entry via the keyboard.

7. A delimiter form $UNPACK operation, if not prematurely terminated by an error message, terminates when either the buffer is empty or the entire argument list is satisfied.

Additional Examples of Valid Syntax for $PACK and $UNPACK

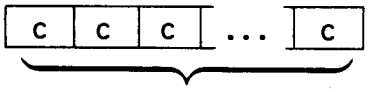
```
$PACK (D=STR(D$,4,2)) Q$() FROM X,Y,Z(1)
$PACK (D=D$) B$()<5, 100> FROM X()
$UNPACK (D=STR(Q$,3,2)) X$() TO X,Y,Z(1,2)
$UNPACK (D=M$) A$() <M,N> TO P()
```



where:

DEL = delimiter byte (any specified 8-bit code).
data = alphanumeric or numeric value (see Figure 2-2 and 2-3).

Figure 2-1. $PACK Delimiter Formatted Data Buffer



As many bytes as the dimensioned length of the alphanumeric variable (trailing spaces added, if necessary).

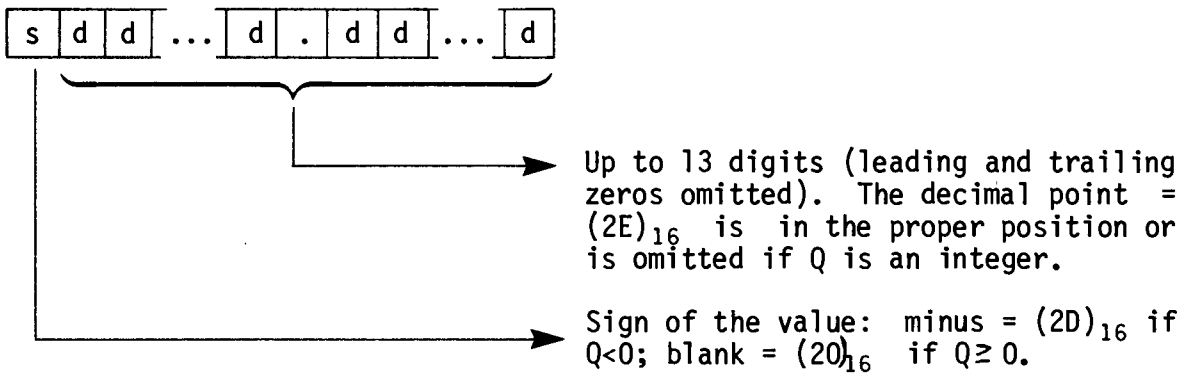
where:

c = any 8-bit code, except the specified delimiter code.

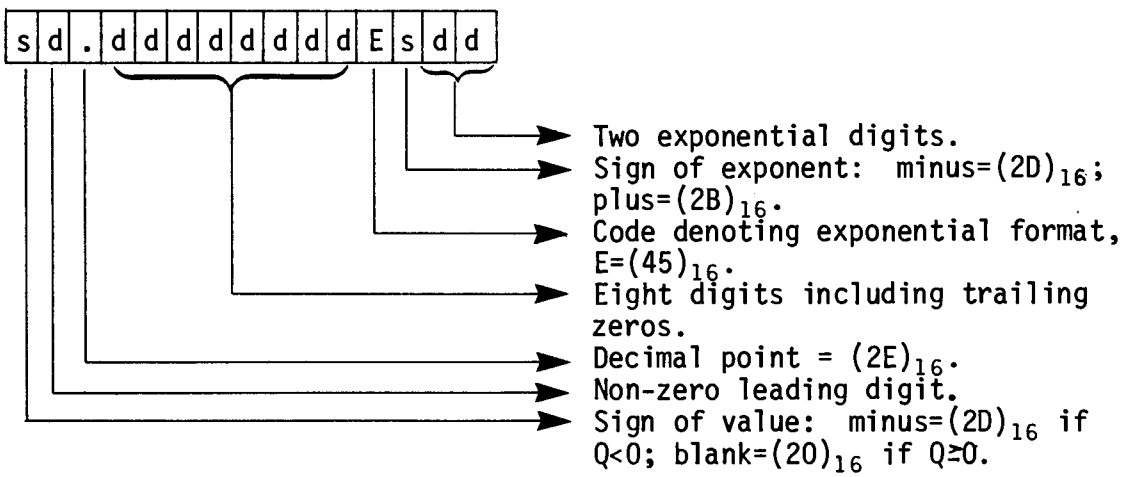
Figure 2-2. $PACK Delimiter Formatted Data from an Alphanumeric Variable

Data from a numeric variable is packed in the buffer in a fixed point format or a floating point format, depending upon the magnitude, i.e., the absolute value, |Q| .

Fixed Point Format if $.1 \leq |Q| < 10^{+13}$



Floating Point Format if $10^{-99} < |Q| < .1$, or if $10^{+13} \leq |Q| < 10^{+100}$



where, in either format,

d = any decimal digit (0 through 9) stored as an 8-bit ASCII character of the form $(3h)_{16}$, where "h" is any hexdigit 0 through 9. See Appendix C.

Figure 2-3. $PACK Delimiter Formatted Data from a Numeric Variable

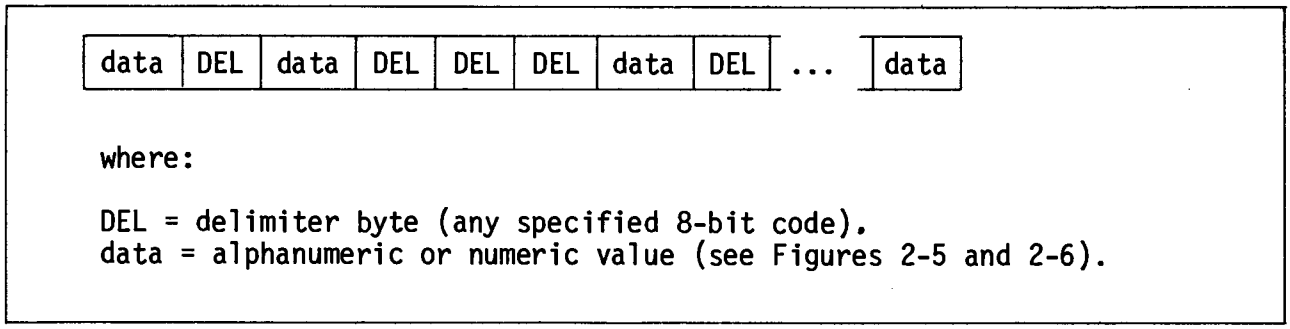


Figure 2-4. Delimiter Formatted Data Buffer Suitable for $UNPACK Operations

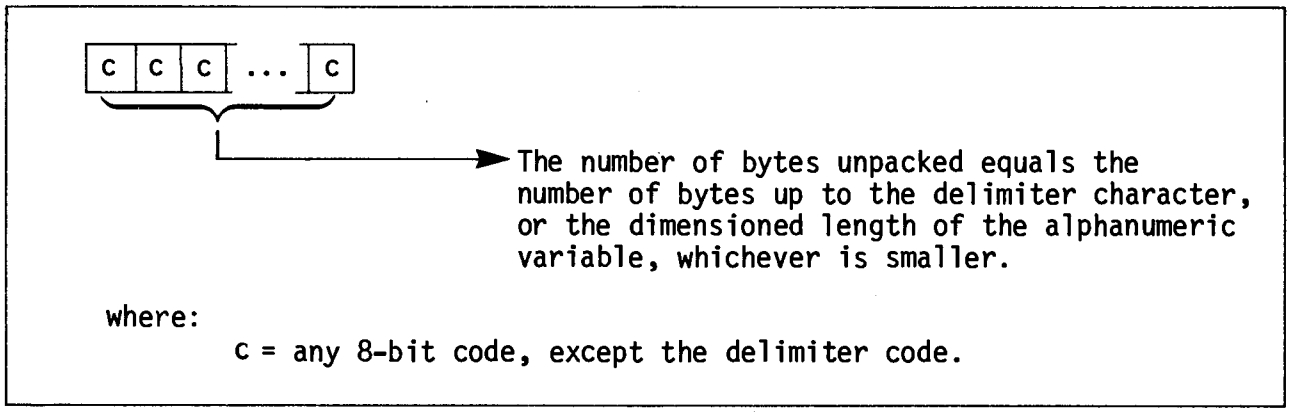
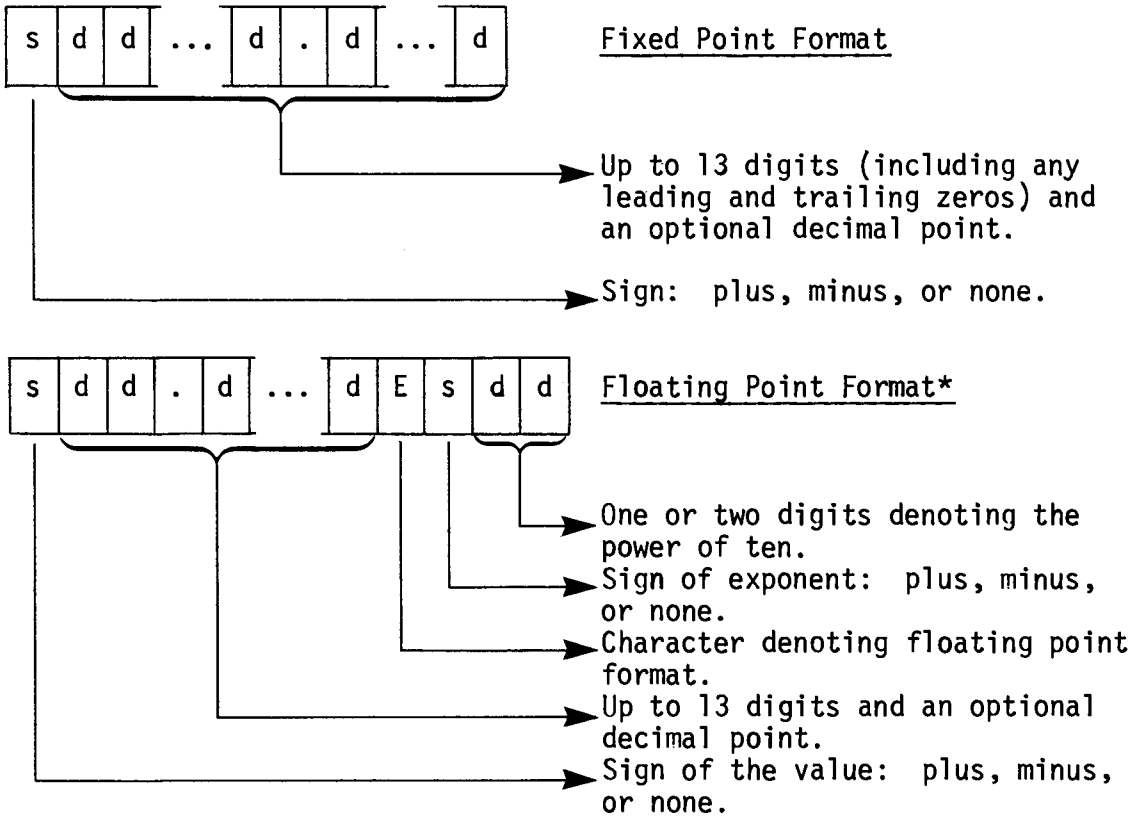


Figure 2-5. Delimiter Formatted Data Suitable for Unpacking to an Alphanumeric Variable

ASCII free format numbers with magnitude from 10^{-99} to 10^{+100} (excluding the value 10^{+100}) can be unpacked from the buffer if the number format is acceptable to the system for keyboard entry. Any leading, trailing, and/or embedded spaces are ignored. The following fixed and floating point formats are acceptable (digits, decimal points, signs and E must be ASCII codes shown in Appendix C).



*If unpacking into numeric arrays, this format is restricted to numbers whose exponents are < 10 in magnitude; no such restriction applies if unpacking numbers to specific numeric array elements.

Figure 2-6. Delimiter Formatted Data Suitable for Unpacking to a Numeric Variable

Example 2-4. A Delimiter Form $PACK Operation

The following program packs the values currently stored in the variables X, A$, and Y into the delimiter formatted record B$ with the specified delimiter character (a comma) separating each value from the successive value.

```
100 DIM B$30, A$5
110 A$ = "ABC": X = -12: Y = 4.56E-18
120 D$ = HEX(002C)
130 $PACK (D=D$) B$ FROM X, A$, Y
140 PRINT "B$="; B$
:RUN
B$=-12,ABC   , 4.56000000E-18
```

Example 2-5. A Delimiter Form $UNPACK Operation

The following program unpacks the five numeric data values, currently stored in the B$ record with commas (the specified delimiter character) as separators, and stores the values in the variables A,B,C,D and E.

```
10 DIM B$24
20 B$ = "123, -.4567,0,+5,.009"
30 D$ = HEX(002C)
40 $UNPACK (D=D$) B$ TO A,B,C,D,E
50 PRINT A;B;C;D;E
:RUN
 123 -.4567  0  5  9.00000000E-03
```

Example 2-6. A Delimiter Form $UNPACK Operation with Three Variations

The delimiter formatted record B$ contains three alphanumeric data values separated, as shown schematically, by one or more delimiters (defined to be a space character).


```
200 DIM B$12
210 B$ = "ABC DEF  GHI"
220 D$ = HEX(0320)
230 $UNPACK (D=D$) B$ TO W$, X$, Y$, Z$
240 PRINT"W$=";W$, "X$=";X$, "Y$=";Y$, "Z$=";Z$
:RUN
W$=ABC          X$=DEF          Y$=GHI          Z$=
```

Now, if Line 220 is changed as follows:

```
220 D$ = HEX(0020)
```

and the program is run a second time, the result becomes

```
:RUN
W$=ABC          X$=DEF          Y$=             Z$=GHI
```

Furthermore, if Line 220 is changed as follows:

```
220 D$ = HEX(0220)
```

and the program is run a third time, the result becomes

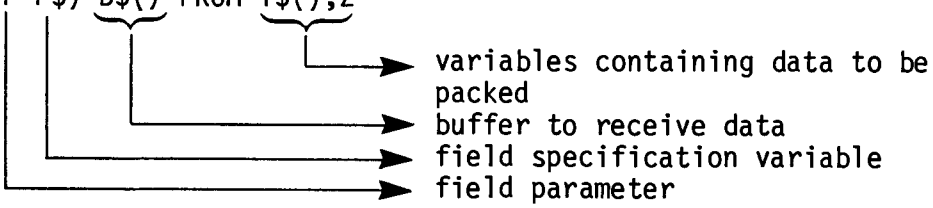
```
:RUN
40 $UNPACK (D=D$) B$ TO W$, X$, Y$, Z$
                                  ↑ERR 97
```

The first time the program in Example 2-6 is run, the first byte stored in the delimiter specification variable D$ is $(03)_{16}$. The second time the program is run, the first byte stored in D$ is $(00)_{16}$. The third time the program is run, the first byte stored in D$ is $(02)_{16}$. Therefore, as indicated in Table 2-1, the unpacking procedures during execution of Line 230 are different in the three variations of the program. Even though the data in B$ is insufficient for the four arguments specified in the argument list, the fourth argument and the extra delimiter character are ignored in the first unpacking procedure. In the second procedure, the argument Y$ is skipped when the extra delimiter character is encountered. In the third procedure, an error message is displayed; however, data is unpacked until the error condition arises.

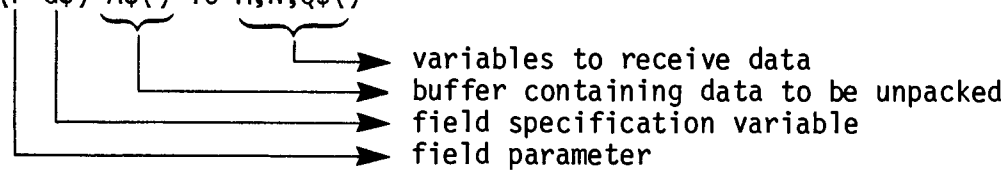
2.4 THE FIELD FORM FOR $PACK AND $UNPACK OPERATIONS

When denoting the field form for a $PACK or $UNPACK operation, a field specification variable must be assigned to the parameter F as shown in the following examples of valid syntax.

```
$PACK (F=F$) B$() FROM Y$(),Z
```




```
$UNPACK (F=G$) A$() TO M,N,Q$()
```



The Field Specification Variable

The value stored in a field specification variable prior to execution of a $PACK or $UNPACK operation must provide as many two-byte codes of the form shown in Table 2-2 (not counting any "skip field" codes) as the number of arguments in the argument list. See Examples 2-7 through 2-11.

Observe in Table 2-2 that the first byte of each field specification code defines the field type, e.g., the code (A0) indicates an alphanumeric field. Codes are available to denote five different types of numeric fields. The code $(00)_{16}$ indicates that a field is to be skipped.

As shown in Table 2-2, the second byte in each field specification code denotes the field width in hexadecimal notation. For an array, the field width is defined as the width of each element (not the width of the array as a set of contiguous bytes) since arrays are packed or unpacked element-by-element.

Table 2-2. Valid Field Specifications in Hexadecimal Notation

Two-byte Code	Field Type, Denoted by First Byte	Remarks
00hh	Skip the number of bytes denoted by the field width	
10hh	Numeric field in ASCII free format	See Fig. 2-9
$2h_\rho hh$	Numeric field in ASCII implied decimal format	See Fig. 2-10
$3h_\rho hh$	Numeric field in IBM display format	See Fig. 2-11
$4h_\rho hh$	Numeric field in IBM USASCII-8 format	See Fig. 2-12
$5h_\rho hh$	Numeric field in IBM packed decimal format	See Fig. 2-13
A0hh	Alphanumeric field	See Fig. 2-8

*hh = a two-hexdigit-code whose equivalent decimal system value is the field width, e.g., $(32)_{16} = (50)_{10}$ = a fifty byte field width. For an array, the second byte of its field specification code must define the field width for each element of the array (not the entire array).

h_ρ = the low-order hexdigit whose equivalent decimal system value defines the implied decimal point position, measured from the right side of the field, e.g., $h_\rho = 3$ means an implied decimal position three places from the right, h_ρ = B means an implied decimal position eleven places from the right.

The $PACK statement at the beginning of Section 2.4 requires two field specification codes for F$ since the argument list includes the alphanumeric array Y$() and the numeric variable Z. Now, suppose

F$ = HEX(A0187100D)

then

$(A018)_{16}$ = the field specification for each element of Y$(), where $(A0)_{16}$ denotes an alphanumeric field type, and $(18)_{16} = (24)_{10}$ denotes a 24-byte field width.

$(100D)_{16}$ = the field specification for Z, where $(10)_{16}$ denotes a numeric field in ASCII free format, and $(0D)_{16} = (13)_{10}$ denotes a 13-byte field width.

Or, suppose F$ = HEX(A01800071100D), then a 7-byte field width would be skipped between the value packed from the last element of Y$() and the value packed from Z.

Similarly, the $UNPACK statement requires three field specification codes for G$ since the argument list includes the numeric variables M and N followed by the alphanumeric array Q$(). Now, suppose

G$ = HEX(510B2509A025)

then

$(510B)_{16}$ = the field specification for M, where $(51)_{16}$ denotes an IBM packed decimal numeric field with an implied decimal position one place from the right, and $(0B)_{16} = (11)_{10}$ denotes an 11-byte field width.

$(2509)_{16}$ = the field specification for N, i.e., a 9-byte numeric field in ASCII implied decimal format with the decimal position five places from the right.

$(A025)_{16}$ = the field specification for Q$(), i.e., each element is a 37-byte alphanumeric field.

Notes:

1. The number of two-byte codes stored in the field specification variable (not counting any skip field codes) must equal the number of arguments in the argument list of a $PACK or a $UNPACK statement. If not, an error message (Code 97) is displayed during execution. Remember, however, that each array requires only one field specification to define the type and width of every element in the array since an array is always packed or unpacked element-by-element.

2. Each field type code in the field specification value must match the corresponding alphanumeric/numeric variable type in the argument list. If not, an error message (Code 43) is displayed during $PACK or $UNPACK execution.

3. Conversion of numeric data to (or from) up to five different field types can be performed automatically when a buffer is packed (or unpacked) by a field form $PACK (or $UNPACK) statement.

Features of a Field Form $PACK Operation

1. A specified portion of a buffer is packed if an alpha array modifier is included in a $PACK statement.

2. The buffer is not cleared before the packing operation begins. In applications where undesirable data may arise from reuse of a buffer, initialization of the buffer by an INIT statement may be advisable prior to execution of a $PACK statement.

3. Data from each variable in the argument list is packed in the buffer sequentially, according to the specified field type and width for the argument being packed. See Figure 2-7. If the field type code is numeric when the argument is alphanumeric or vice versa, an error message (Code 43) is displayed.

4. Data from an alphanumeric variable is packed as shown schematically in Figure 2-8.

5. Data from a numeric variable is converted automatically from the 8-byte (packed decimal) internal format used in the system memory to the format specified by the field type code. The five valid field types are shown schematically in Figures 2-9 through 2-13.

6. If the buffer is too small to hold the data corresponding to the next variable in the argument list, an error message (Code 97) is displayed.

7. If the buffer is large enough to hold all data values, a field form $PACK operation terminates after data from the last variable is stored in the buffer.

Features of a Field Form $UNPACK Operation

1. A specified portion of a buffer is unpacked if an alpha array modifier is included in a $UNPACK statement.

2. When a buffer is being unpacked, each successive data value is recognized as the number of contiguous characters corresponding to the specified field width of the next receiving variable. See Figure 2-7.

3. If the specified field type is numeric when the next receiving variable is alphanumeric or vice versa, an error message (Code 43) is displayed.

4. When an alphanumeric field is unpacked to an alphanumeric variable, the number of bytes stored in the variable equals the field width or the dimensioned length, whichever is smaller. See Figure 2-8.

5. When a numeric field is unpacked to a numeric receiving variable, the data is converted automatically from the specified field type to the 8-byte packed-decimal format used in the system memory. If the data contains an illegal character, an error message (Code 20) is displayed. The five valid field types are shown schematically in Figures 2-9 through 2-13.

6. A field form $UNPACK operation, if not prematurely terminated by an error message, terminates when either the buffer is empty or the entire argument list is satisfied.

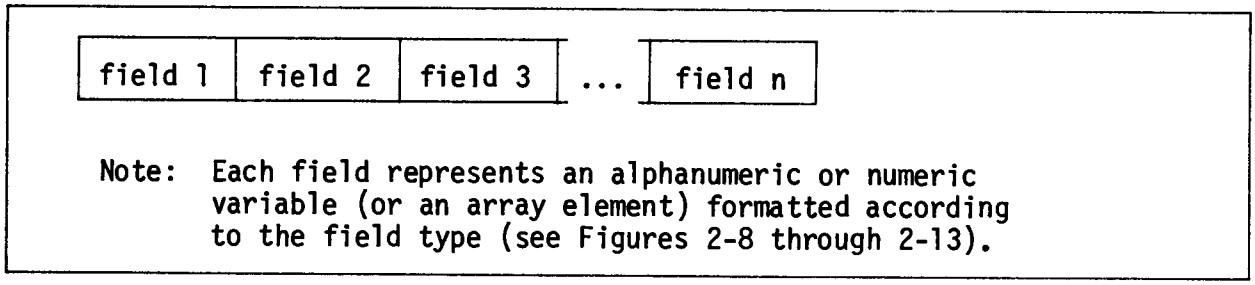


Figure 2-7. Field Formatted Data Buffer

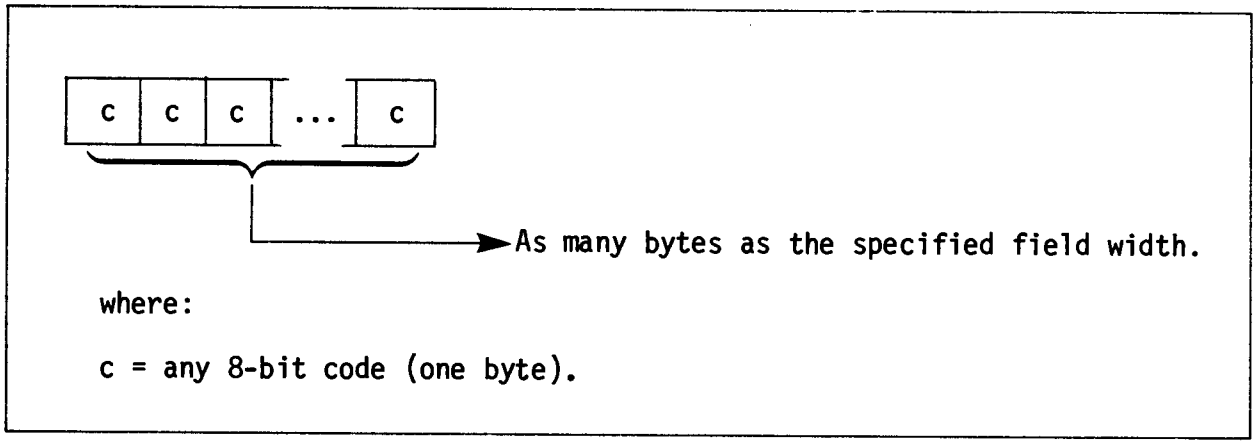


Figure 2-8. Alphanumeric Field Format, Denoted by Codes of the Form A0hh

ASCII free format numbers with magnitude from 10^{-99} to 10^{+100} (exclusive of the upper end point) can be unpacked from the buffer if the number format is acceptable to the system for keyboard entry. Any leading, trailing, and/or embedded spaces are ignored. The following fixed and floating point formats are acceptable (digits, decimal points, signs and E must be ASCII codes shown in Appendix C).

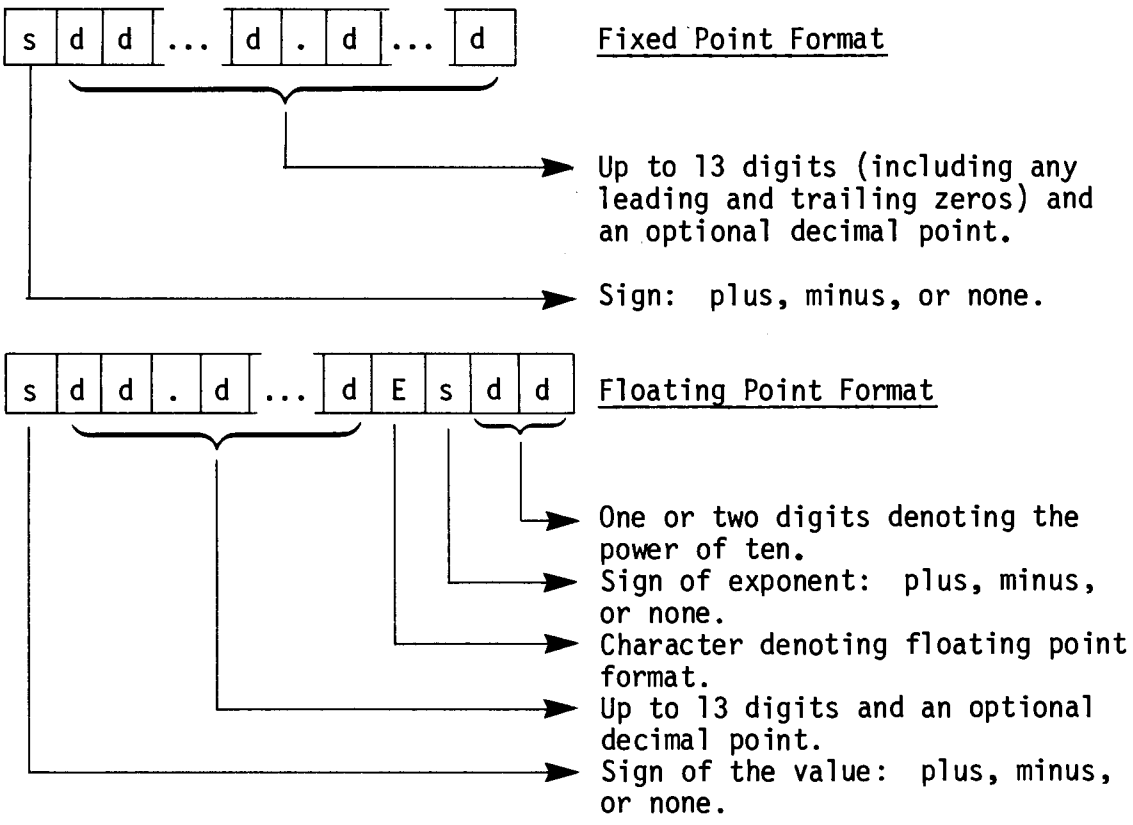
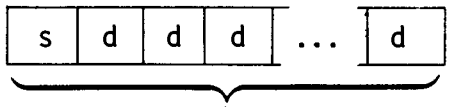


Figure 2-9. Numeric Field in ASCII Free Format, Denoted by Codes of the Form 10hh



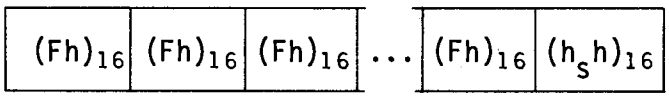
Note: Each byte, except the first one, represents a significant decimal digit in the value. The first byte represents the sign. No decimal point is stored.

As many bytes as the field width.

where:

s = a required sign, stored as an ASCII plus or minus; + = $(2B)_{16}$ and - = $(2D)_{16}$.
d = any decimal digit (0 through 9), stored as an ASCII code of the form $(3h)_{16}$.

Figure 2-10. Numeric Field in ASCII Implied Decimal Format, Denoted by Codes of the Form $2h_{\rho}hh$



Note: Each byte, except the last one, is a code of the form $(Fh)_{16}$ representing one decimal digit in the number. The last byte contains the sign code as the high-order hexdigit.

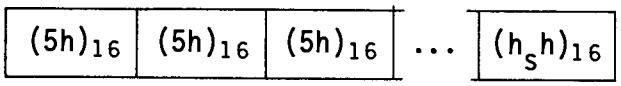
where:

F = the high-order hexdigit.

h = a low-order hexdigit (0 through 9, only), representing any decimal digit.

h_s = a hexdigit representing the sign of the number; C = +, D = - for a $PACK operation. For $UNPACK, B or D = -, and A,C, or E = +.

Figure 2-11. Numeric Field in IBM Display Format, Denoted by Codes of the Form $3h_{\rho}hh$



Note: Each byte, except the last one, is a code of the form $(5h)_{16}$ representing one decimal digit in the number. The last byte contains the sign code as the high-order hexdigit.

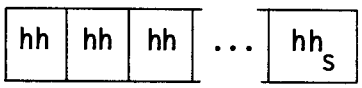
where:

5 = the high-order hexdigit.

h = a low-order hexdigit (0 through 9, only), representing any decimal digit.

h_s = a hexdigit representing the sign of the number; A = +, B = - for a $PACK operation. For $UNPACK, B or D = -, and A,C, or E = +.

Figure 2-12. Numeric Field in IBM USASCII-8 Format, Denoted by Codes of the Form $4h_\rho hh$



Note: Each byte, except the last one, contains two hexdigits representing two decimal digits in the number (thereby conserving memory by requiring only a half-byte per decimal digit).

where:

h = a hexdigit (0 through 9, only), representing any decimal digit stored as half a byte.

h_s = a hexdigit representing the sign of the number; C = +, D = - for a $PACK operation. For $UNPACK, B or D = -, and A,C, or E = +.

Figure 2-13. Numeric Field in IBM Packed Decimal Format, Denoted by Codes of the Form $5h_\rho hh$

Example 2-7. A Field Form $PACK Operation

The following program demonstrates how values from an alphanumeric variable and two numeric variables are packed into the B$ record, according to a prescribed field format.

```
10 DIM B$15
20 A$ = "ABC": X = -12: Y = +1.2345
30 F$ = HEX(A00520051005)
40 $PACK (F=F$) B$ FROM A$, X, Y
50 PRINT "B$="; B$
:RUN
B$=ABC  -0012 1.23
```

Example 2-8. A Field Form $PACK Operation, Illustrating Data Conversion to Several Different Numeric Field Formats

The following program demonstrates how numeric data can be converted to a specified numeric field format during a $PACK operation. For simplicity, the argument list contains only one numeric variable X whose current value is to be packed in the record B$, according to a prescribed field format. The dimension of B$ is the default length 16 bytes.

```
10 X = 12.345
20 F$ = HEX(100A)                    (See Table 2-2 and Figure 2-9.)
30 $PACK (F=F$) B$ FROM X
40 PRINT "B$ IN HEXADECIMAL NOTATION:"
50 HEXPRINT B$
60 PRINT "B$="; B$
:RUN
B$ IN HEXADECIMAL NOTATION:
2031322E33343520202020202020202020
B$=12.345
```

In the above result, a space character is stored in the first byte of B$. The decimal digit 1 is stored in the second byte of B$ as the ASCII 8-bit code $(31)_{16}$. The decimal point is stored in the fourth byte of B$ as the ASCII code $(2E)_{16}$. All sixteen bytes of B$ are shown in hexadecimal notation when Line 50 is executed.

Now, if Line 20 is changed as follows:

```
20 F$ = HEX(240A)                    (See Table 2-2 and Figure 2-10.)
```

and the program is run a second time, the result becomes

```
:RUN
B$ IN HEXADECIMAL NOTATION:
2B3030303132333435302020202020
B$=+000123450                    (Implied decimal point four digits
                                    from the right.)
```

If Line 20 is changed as follows:

```
20 F$ = HEX(3305)                    (See Table 2-2 and Figure 2-11.)
```

and the program is run a third time, the result becomes

```
:RUN
B$ IN HEXADECIMAL NOTATION:
F1F2F3F4C5202020202020202020202020
B$=QRSTE                          (Ignore this result; non-Wang codes
                                    should not be printed.)
```

If Line 20 is changed as follows:

```
20 F$ = HEX(4206)                    (See Table 2-2 and Figure 2-12.)
```

and the program is run a fourth time, the result becomes

```
:RUN
B$ IN HEXADECIMAL NOTATION:
5050515253A4202020202020202020202020
B$=PPQRS$                         (Ignore this result; non-Wang codes
                                    should not be printed.)
```

If Line 20 is changed as follows:

```
20 F$ = HEX(5506)                    (See Table 2-2 and Figure 2-13.)
```

and the program is run a fifth time, the result becomes

```
:RUN
B$ IN HEXADECIMAL NOTATION:
00001234500C2020202020202020202020
```

If Line 20 is changed as follows:

```
20 F$ = HEX(A010)                    (See Table 2-2 and Figure 2-8.)
```

and the program is run a sixth time, an error code is printed since a numeric value cannot be packed in an alphanumeric field format.

```
:RUN
30 $PACK (F=F$) B$ FROM X
                          ↑ ERR 43
```

If Line 20 is changed as follows:

```
20 F$ = HEX(2304)
```

an error code (↑ERR 56)is printed, when the program is run, since the field width is too small to hold the value.

Example 2-9. A Field Form $UNPACK Operation

The following program unpacks the buffer B$, according to a specified field format, and successively stores each field in the variables specified in the argument list.

```
90 DIM B$37
95 B$="J. SMITH  M5.80130  1234.5689"
100 FS=HEX(A00AA00110071003101O)
110 $UNPACK(F=F$) B$ TO A$, C$, X,Y,Z
120 PRINT A$, C$, X, Y,Z
:RUN
J. SMITH        M               5.8013          1
 234.5689
```

Example 2-10. A Field Form $UNPACK Operation, Specifying Several Numeric Formats

```
10 DIM B$21
20 B$ = "+123456789012345678901"
30 F$ = HEX(2015)
40 $UNPACK (F=F$) B$ TO X
50 PRINT X
:RUN
 1.23456789E+19
```

Now, change Line 30 and run again.

```
30 F$ = HEX(2206)
:RUN
 123.45
```

Now change Line 30 and run again.

```
30 F$ = HEX(1010)
:RUN
40 $UNPACK (F=F$) B$ TO X
                        ↑ ERR 20
```

Example 2-11. A Field Form $UNPACK Operation with Data Conversion

```
10 B$ = HEX(51525354A5)
20 F$ = HEX(4005)
30 $UNPACK (F=F$) B$ TO X
40 PRINT X
:RUN
 12345
```

2.5 THE STANDARD RECORD FORM FOR $PACK AND $UNPACK OPERATIONS

When neither the parameter D nor the parameter F is included in a $PACK or $UNPACK statement, the system packs or unpacks data according to the standard System 2200 record format shown schematically in Figure 2-14. The same format is used by the DATASAVE statement for cassette operations, and by the DATASAVE DC or DATASAVE DA statements for disk operations.

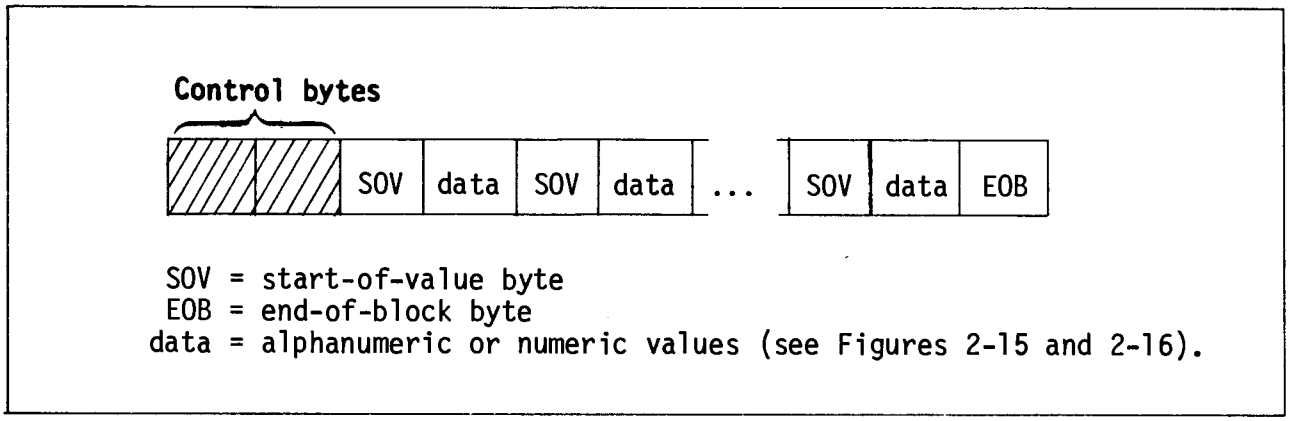
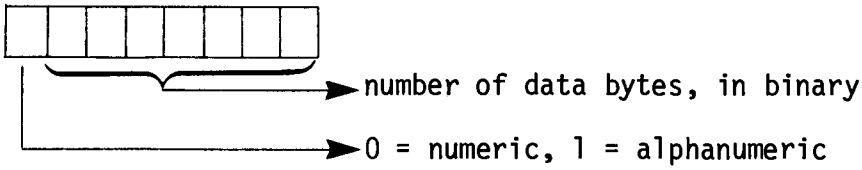


Figure 2-14. Standard Record Formatted Data Buffer

The control bytes, the SOV bytes, and the EOB bytes in the standard record format are described as follows:

1. Control bytes — For $PACK and $UNPACK operations, the only relevant information about control bytes is the fact that two such bytes mark the beginning of the format. The bytes are supplied or interpreted by the system automatically during execution of a standard record form $PACK or $UNPACK operation.

2. SOV bytes — An SOV (start of value) byte precedes data corresponding to each variable or array element represented in the format. The high-order bit in the byte indicates whether the data which follows represents a numeric or alphanumeric value. The seven low-order bits in the byte indicate the number of data bytes in the value (in binary).



3. EOB byte — The EOB (end of block) byte is an $(FD)_{16}$ code indicating the end of valid data in the record.

Two different data formats are possible in the standard record form, depending upon whether a particular value represents a numeric or alphanumeric variable. The data formats are shown in Figures 2-15 and 2-16.

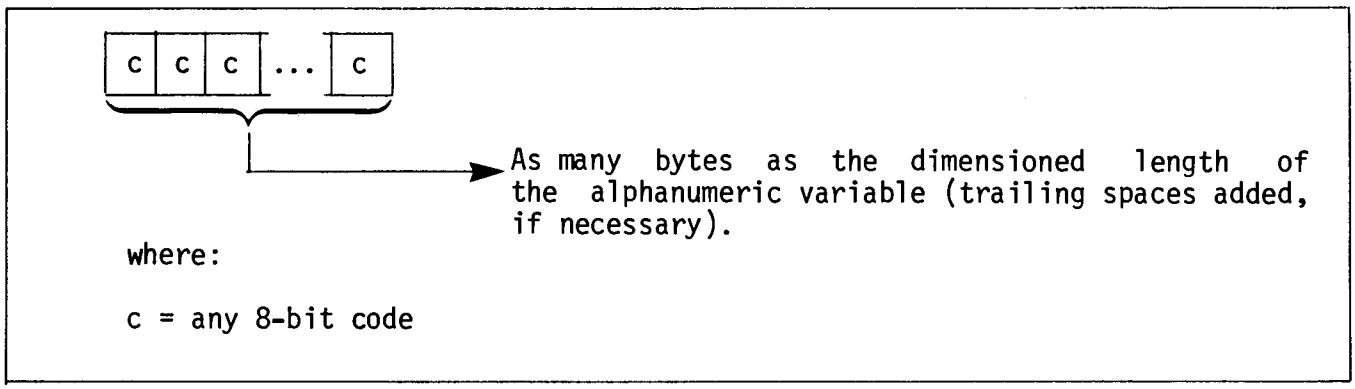


Figure 2-15. Standard Record Formatted Data for an Alphanumeric Variable

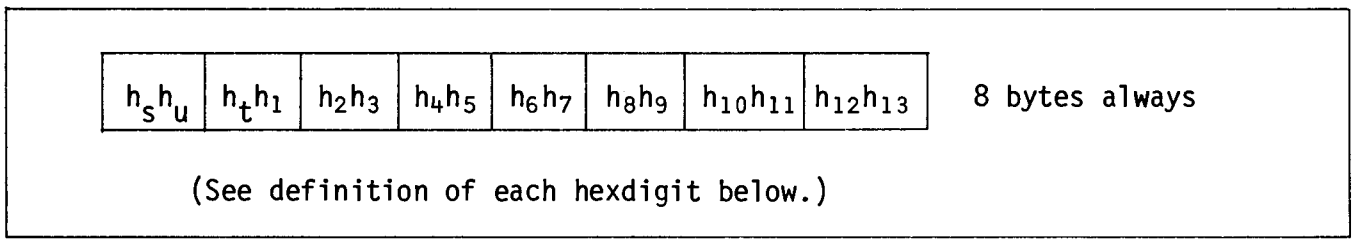


Figure 2-16. Standard Record Formatted Data for a Numeric Variable

The hexdigits in Figure 2-16 are defined as follows:

1. h_s = a hexdigit representing two signs for a numeric value (the sign of the number itself and the sign of the exponent), that is,

   h_s = 0 if the number is + and the exponent is +,
   h_s = 1 if the number is - and the exponent is +,
   h_s = 8 if the number is + and the exponent is -, and
   h_s = 9 if the number is - and the exponent is -.

2. h_u = a hexdigit (0 through 9, only) representing the units-position digit in a two-decimal-digit exponent.

3. h_t = a hexdigit (0 through 9, only) representing the tens-position digit in a two-decimal-digit exponent.

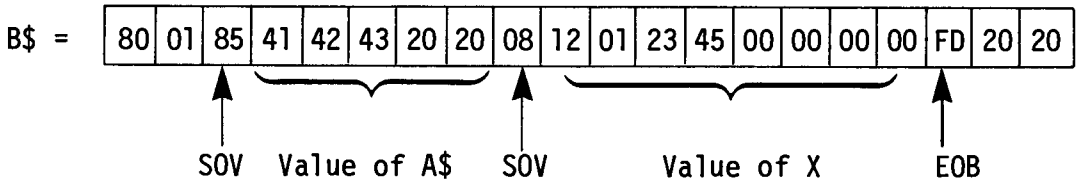
4. h_i = a hexdigit (0 through 9, only) representing (for i=1 through 13) one of the "significant" (sometimes called "mantissa") digits in the normalized form of a number; $h_1 > 0$; the implied decimal position is after h_1; thirteen digits are stored (trailing zeros are added, if necessary).

Example 2-12. A Standard Record Form $PACK Operation

The following program packs the values of A$ and X in the record B$, using the standard record format shown in Figure 2-14.

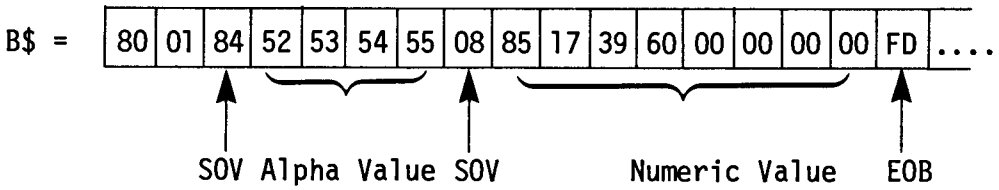
```
10 DIM B$20, A$5
20 A$ = "ABC": X = -123.45
30 $PACK B$ FROM A$, X
```

The result, in hexadecimal notation, is:



Example 2-13. A Standard Record Form $UNPACK Operation

If the record B$ contains one alphanumeric value and one numeric value as follows:



Then the following program unpacks B$ and stores the values in the variables Q$ and Z.

```
100 $UNPACK B$ TO Q$, Z
110 PRINT Q$, Z
:RUN
RSTU            7.39600000E-15
```

CHAPTER 3

I/O OPERATIONS USING $IF ON AND $GIO

3.0 INTRODUCTION

Two of the General I/O Instruction Set statements extend the I/O capabilities of Wang configurations. The statements are $IF ON and $GIO.

The $IF ON statement can test the device-ready condition of a specified output device or test the data-ready condition of a specified input device and initiate a branch to a specified line number if a ready condition is sensed.

The $GIO statement permits the design of customized input and output operations to support non-Wang peripheral devices interfaced to a Wang system via such interface controllers as the Models 2207A, 2227, 2227B, 2250, and 2252A. The $GIO statement is required for some Wang peripheral devices (e.g., the Model 2209 Nine-Track Tape Drive and the Model 2228 Communications Controller operating with a terminal emulator program); however, in such cases, Wang Laboratories documents the required $GIO statements by specifying the microcommand sequence needed for each input, output, or control operation or supplies a software package for the device. If desired, Wang keyboards and CRT's can be controlled by $GIO statements, but cassette and disk drives cannot.

3.1 THE $IF ON STATEMENT

For input device scanning applications, the $IF ON and KEYIN statements should be compared. For example, KEYIN is well-suited for handshake applications where a single control byte is received when a data ready condition is sensed, and for applications where special function codes should initiate a program branch. By contrast, $IF ON is preferred for applications where a multicharacter input device is to be tested since the statement does not receive the first character upon sensing a data ready condition.

For output device scanning applications, $IF ON tests the ready/busy signal on the controller which serves as the interface between the Wang central processor and the external device. The conditions which produce the controller's ready signal depend upon the characteristics of the device and the signals exchanged between the controller and the device. Therefore, a programmer must check the suitability and programming logic of a $IF ON statement when debugging an application program.

For example, a printer (even when not powered on) may appear ready if a $IF ON statement scans the device when the controller's one-byte buffer is empty. On the other hand, the printer may appear busy if scanned while the controller holds a byte of data awaiting a ready signal from the printer buffer.

$IF ON

General Form:

$IF ON [#f,] line-number
       [/xyy,]

where:

f = An indirect address - a file number (1,2,3,4,5 or 6) to which the address of the I/O device to be tested is assigned.

xyy = An absolute address - the three hexdigit address code of the I/O device to be tested.

line-number = A specified program line to which a branch is to be made if a ready condition is sensed.

1. If neither an indirect nor absolute address is specified, the address currently selected for the I/O-class parameter TAPE is the default address. (Care must be exercised to avoid defaulting to a cassette drive address since cassette drives cannot be controlled by $IF ON statements.)

2. Either a device-ready or a data-ready condition is tested, depending upon whether the specified address belongs to an output or an input device. The device-ready condition is tested if an output device address is in effect. The data-ready condition is tested if an input device address is in effect. If a not-ready condition is sensed, program execution proceeds to the next statement. If a ready-condition is sensed, program execution branches to the specified line number.

$GIO

General Form:

$GIO [comment] [#f | /xyy] (arg-1, arg-2) [arg-3]

where:

comment = An optional character string identifying the particular operation (e.g., WRITE, READ, CHECK READY); the comment is ignored by the system.

f = An indirect address - a file number (1,2,3,4,5 or 6) to which the address of the I/O device is assigned.

xyy = An absolute address - a three-hexdigit address code with the recommended value for x equal to "0" since the device type code is not utilized during a $GIO operation; the value of yy must be the two hexdigit preset address on the controller board into which the I/O device is plugged.

arg-1 = A customized microcommand sequence, defining the input or output operation as follows:

a) Directly, by a string of hexdigits with each four-hexdigit-code denoting one two-byte microcommand.

b) Indirectly, by an alpharg representing the microcommand sequence.

arg-2 = An alpharg representing a multi-purpose memory area whose byte-positions (called "registers") are used for storage of special-characters and error/status information (the dimensioned length must be at least 10 bytes).

arg-3 = An alpharg representing the data buffer (required for some microcommand sequences, not required for others).

alpharg = { alphanumeric-variable | STR-function | alpha-array-designator | alpha-array-designator <s,n> }

1. If neither an indirect nor absolute address is specified, the address currently assigned to the SELECT statement parameter TAPE is the default address. (Care must be exercised to avoid defaulting to a cassette drive address since cassette drives cannot be controlled by $GIO statements.)

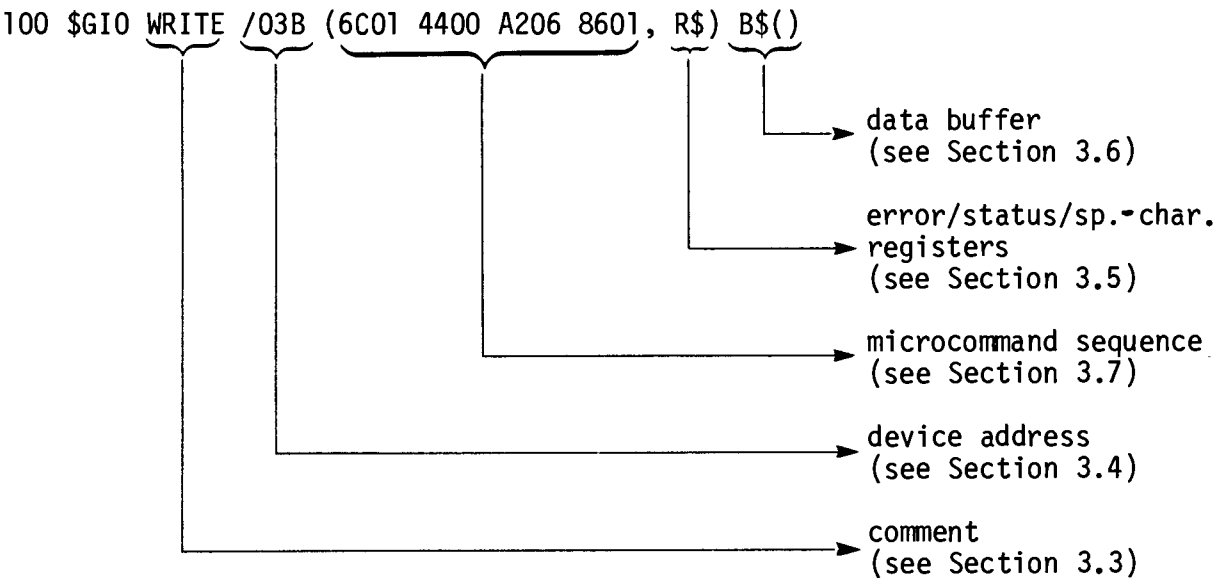
2. For telecommunications applications the alpha array modifier is of the form <s,m,e>. See Appendix A, Table A-13, Note 1.

3.2 THE $GIO STATEMENT

The $GIO statement, as indicated mnemonically, is a General Input/Output statement. The statement implements a particular input, output, or I/O control operation defined by the microcommand sequence specified directly or indirectly in an actual statement. Each microcommand in a particular sequence is denoted by a four-hexdigit-code which represents a set of fundamental operations. The operations associated with valid microcommands are defined in the tables presented in Appendix A of this manual.

By properly choosing a sequence of microcommands, a complete I/O operation can be custom-tailored to suit a non-Wang device interfaced to a Wang system. Thus, within the framework of Wang's high-level BASIC language, the $GIO statement provides a capability similar to machine language programming.

For example, consider the following statement:



The comment and device address in the illustration are optional components. The arg-3 data buffer B$() is required in this example by the presence of the microcommand A206 which represents a multicharacter output operation described in Appendix A, Table A-6 (a data buffer is not required when a sequence contains no multicharacter I/O microcommand). An arg-1 microcommand sequence is always required, whether specified directly (as illustrated) or indirectly by an "alpharg" representing a previously specified microcommand sequence. Also required, as illustrated by R$, is an arg-2 component representing the error/status/general-purpose registers for the $GIO operation.

3.3 OPTIONAL COMMENTS FOR $GIO OPERATIONS

The operation represented by a $GIO statement is not readily identifiable from its microcommand sequence. For this reason, a descriptive comment inserted after the mnemonic $GIO may prove helpful when reviewing or revising a program. Effectively, a comment preserves the conversational feature of Wang's BASIC language.

3.4 DEVICE ADDRESSES FOR $GIO STATEMENTS

Three different methods can be used to specify a device address for a particular $GIO operation:

1) Direct address specification using a slash character followed by a three hexdigit address code, for example,

```
200 $GIO READ /03A (M$, R$) B$()
```

2) Indirect address specification using a pound-sign followed by a file number to which the desired address has been previously assigned, for example,

```
300 SELECT #2 03A
310 $GIO READ #2 (M$, R$) B$()
```

3) Omitting an address, thereby implying the device address currently selected for TAPE-class operations should be used, for example,

```
400 SELECT TAPE 03A
410 $GIO READ (M$,R$) B$()
```

3.5 MICROCOMMAND SEQUENCES FOR $GIO OPERATIONS

Except for the $GIO statement, Wang's BASIC language is a "conversational" language. Each statement begins with a mnemonic or a verb which identifies the built-in function or operation to be performed. Sometimes the general form of a particular statement includes one or more parameters representing alternative procedures which can be selected by a programmer. In general, a programmer can use the convenient BASIC language for a wide range of applications even though the built-in procedures represented by individual statements cannot be altered.

Most BASIC language input and output operations are designed to optimize the performance capabilities of one or more Wang I/O devices. Some of the standard I/O operations provide signal sequences which may prove useful for program control of devices specially interfaced to a Wang system via one of Wang's interface controller boards. However, the $GIO statement is designed to fit the framework of the BASIC language while providing a technique by which input, output, and/or control operations can be custom-designed to suit special devices. A custom-designed I/O operation depends upon the selection of an appropriate microcommand sequence to define a particular $GIO operation.

Each microcommand in a sequence must be represented by a four-hexdigit-code of the form $h_1h_2h_3h_4$, where h_i (i=1, 2, 3, 4) is any hexdigit (0 through 9 or A through F) which is acceptable as indicated by the tables in Appendix A. The first two hexdigits in a microcommand code usually identify the type of operation, e.g., single character output with echo or multicharacter verify. The last two hexdigits supply information, e.g., a character to be stored, the number of a particular register for storage or retrieval of a character, or the number corresponding to a termination condition.

Table 3-1. Microcommand Categories

Category	Code Format*	Type of Operation	Buffer Required
1	$0h_2h_3h_4$ or $1h_2h_3h_4$	Control	no
2	$4h_2h_3h_4$	Single character output	no
3	$5h_2h_3h_4$	Single character output with acknowledge	no
4	$6h_2h_3h_4$	Single character output with echo	no
5	$7h_2h_3h_4$	Address strobe	no
6	$86h_3h_4$	Single character input	no
7	$8h_2h_3h_4$ ($h_2 \neq 6$)	Single character input with verify	no
8	$9h_2h_3h_4$	Single character input with echo	no
9	$Ah_2h_3h_4$	Multicharacter output	yes
10	$Bh_2h_3h_4$ (h_2=0,1,4, or 5)	Multicharacter output with acknowledge	yes
11	$Bh_2h_3h_4$ (h_2=2,3,6, or 7)	Multicharacter output with echo	yes
12	$Bh_2h_3h_4$ (h_2=8,9,C, or D)	Multicharacter output, each character requested	yes
13	BAh_3h_4	Multicharacter verify	yes
14	$C6h_3h_4$	Multicharacter input	yes
15	$Ch_2h_3h_4$ (h_2=0,1,4,5)	Multicharacter input with echo	yes
16	$Ch_2h_3h_4$ (h_2= 8 through F)	Multicharacter input, each character requested	yes
17	$Fh_2h_3h_4$	Telecommunications input	yes

*h_2, h_3, and h_4 are the second, third, and fourth hexdigits, respectively, in a microcommand code.

In Appendix A, general descriptions of each microcommand category are given. Table A-1 describes the fundamental operations represented by "control" microcommands whose codes have h_1 , i.e., the first hexdigit of a four-hexdigit-code, equal to 0 or 1. Table A-2 describes the fundamental operations represented by "single character data transfer" microcommands having h_1 =4, 5, 6, 7, 8 or 9. Table A-3 describes "multicharacter data transfer" microcommands having h_1=A, B, C or F.

The seventeen categories of microcommands listed in Table 3-1 encompass many subcategories of microcommands. Each microcommand subcategory represents a prescribed signal sequence defining a fundamental operation which may be appropriate for many different composite operations. In Appendix A, detailed information is presented as follows:

1) Table A-4 gives the signal sequences for 34 microcommand subcategories representing single character output.

2) Table A-5 gives the signal sequences for 14 microcommand subcategories representing single character input.

3) Table A-6 gives the signal sequences for 19 microcommand subcategories representing multicharacter output.

4) Table A-9 gives the signal sequences for 14 microcommand subcategories representing multicharacter input.

5) Table A-13 gives the signal sequences for 8 microcommand subcategories representing telecommunications input.

6) Table A-1 gives 9 microcommand subcategories representing control of I/O operations.

A microcommand sequence for a $GIO operation can consist of any number of codes belonging to the subcategories in Tables A-1, A-4, and A-5 but at most one code belonging to a subcategory in Table A-6, A-9, or A-13. Choosing a microcommand sequence is equivalent to programming the signal sequence needed to implement a desired operation.

Anyone programming $GIO operations for non-Wang devices interfaced to a Wang system must be thoroughly familiar with the hardware of the device to be operated and must know how the device is interfaced (i.e., what controller board and cables are used as the interface through which signals are exchanged between the CPU of the system and the external device).

When the $GIO statement is required to control a Wang device, Wang Laboratories documents the microcommand sequences needed for each type of operation or supplies a software package.

3.6 DIRECT OR INDIRECT SPECIFICATION OF A MICROCOMMAND SEQUENCE

After a microcommand sequence representing a desired operation has been chosen, either one of two methods can be used to specify the sequence in an actual $GIO statement -- the direct method or the indirect method.

In the direct method, the microcommand sequence is written in the $GIO statement as a string of hexadecimal symbols (i.e., any valid combination of the characters 0 through 9 and A through F). For readability, a space character can be inserted after each four hexdigit microcommand code. The space characters (the blanks) are ignored by the system. For example,

```
100 $GIO REWIND (6C07 4400 8601, R$)
```

In the indirect method, the microcommand sequence is stored in an alphanumeric variable or array prior to the $GIO statement. Then, the variable or array is used as the arg-1 component of the $GIO statement. No space characters can be inserted between microcommand codes for readability in this method. However, the dimension of the alphanumeric variable or array must be large enough to assure the presence of the "microcommand" code 2020 which is necessary to denote the end of the microcommand sequence. Unpredictable results may occur if at least one trailing blank does not follow the sequence. For example,

```
100 DIM M$8                          Reserves 8 bytes in M$.
110 M$ = HEX(6C0744008601)           Specifies 6 bytes; leaves two blanks.
120 $GIO (M$,R$)
```

The indirect method of specifying the microcommand sequence for a $GIO statement is useful when it is desirable to write only one $GIO statement in a program and redefine the microcommand sequence before a subsequent execution of the statement. Also, the indirect method conserves space in memory if a $GIO statement is written repetitively.

3.7 DATA BUFFERS FOR $GIO OPERATIONS

A data buffer is not required for every $GIO operation. As shown in Table 3-1, eight of the seventeen categories of microcommand codes require no buffer. A microcommand sequence can include any number of microcommands which do not require a buffer.

Each of the nine categories of microcommands which represent a multicharacter data transfer operation requires a specified data buffer to identify the data storage area to be used for a particular input or output operation. A $GIO microcommand sequence cannot include more than one multicharacter data transfer microcommand (i.e., a code with its first hexdigit h_1 = A,B,C, or F).

When specifying a data buffer for a $GIO operation, an alphanumeric scalar variable or a string function can be used if 64 or less bytes of data are to be transferred. For transfer of more than 64 bytes of data, an alphanumeric array (one or two dimensional) must be specified as the buffer for the operation. If desired, an alpha array modifier can be used to specify a particular portion of an array as a $GIO buffer.

Unless indicated otherwise by an alpha array modifier or a string function, the data buffer size for a $GIO operation is equal to the dimensioned length of the arg-3 alphanumeric array or variable.

The following rules apply to $GIO multicharacter data transfer:

1. When a one or two dimensional alphanumeric array is specified as the buffer, the array is treated as a set of contiguous characters (bytes) and element boundaries are ignored.

2. If the only specified termination condition for an input operation is the character "count", execution of the input microcommand terminates when the number of received characters equals the defined length of the arg-3 buffer. (See Tables A-9 and A-10.) For example, in Line 100 below, the B$-array (defined as a one-dimensional array with 10 elements, each 10 bytes long) provides a storage area for 100 contiguous bytes. In Line 150, the $GIO operation (defined by the microcommand code C640) implements a multicharacter input operation to be terminated by the character count , i.e., when the buffer is filled with 100 characters.

```
100 DIM B$(10)10
 .
 .
 .
150 $GIO (C640, R$) B$()
```

3. If either a "special character" or an "input strobe with ENDI level set to logic 1" is one of the termination conditions for an input operation, the buffer length should be large enough to receive and store all input data. Registers 9 and 10 in the arg-2 variable will contain the count of the number of characters actually received. If the number of received characters exceeds the available buffer space, an error bit is set in arg-2 Register 8; in such a case, the count in Registers 9 and 10 reflects the total number of received characters whether stored or not. (See Figure 3-1 in Section 3.8.)

4. For an output operation, the total number of characters sent equals the total number of characters in the defined length of the arg-3 data buffer unless a termination condition is specified. (See Table A-6 and A-7.) For example, in Line 40 below, the X$-array is defined as a two-dimensional array with 8 rows having 10 elements per row, with each element 20 bytes long. Therefore, the X$-array provides a storage area for 1600 contiguous bytes of information. In Line 70, the $GIO operation (defined by the microcommand code A000) implements a multicharacter output signal sequence which does not terminate until the 1600 characters in the buffer are sent out.

```
40 DIM X$(8,10)20
 .
 .
 .
70 $GIO (A000, R$) X$()
```

> Note:
>
> By using alpha array modifiers, different portions of one alphanumeric array can be used in a program for several $GIO operations as well as $PACK and $UNPACK operations.

3.8 ERROR/STATUS/GENERAL-PURPOSE REGISTERS

The $GIO statement uses its arg-2 component as a multipurpose memory area where special characters and error/status information are stored. The dimensioned length of this memory area must be at least 10 bytes in most cases, or at least 12 bytes if the arg-1 component (the microcommand sequence) defines a telecommunications input operation.

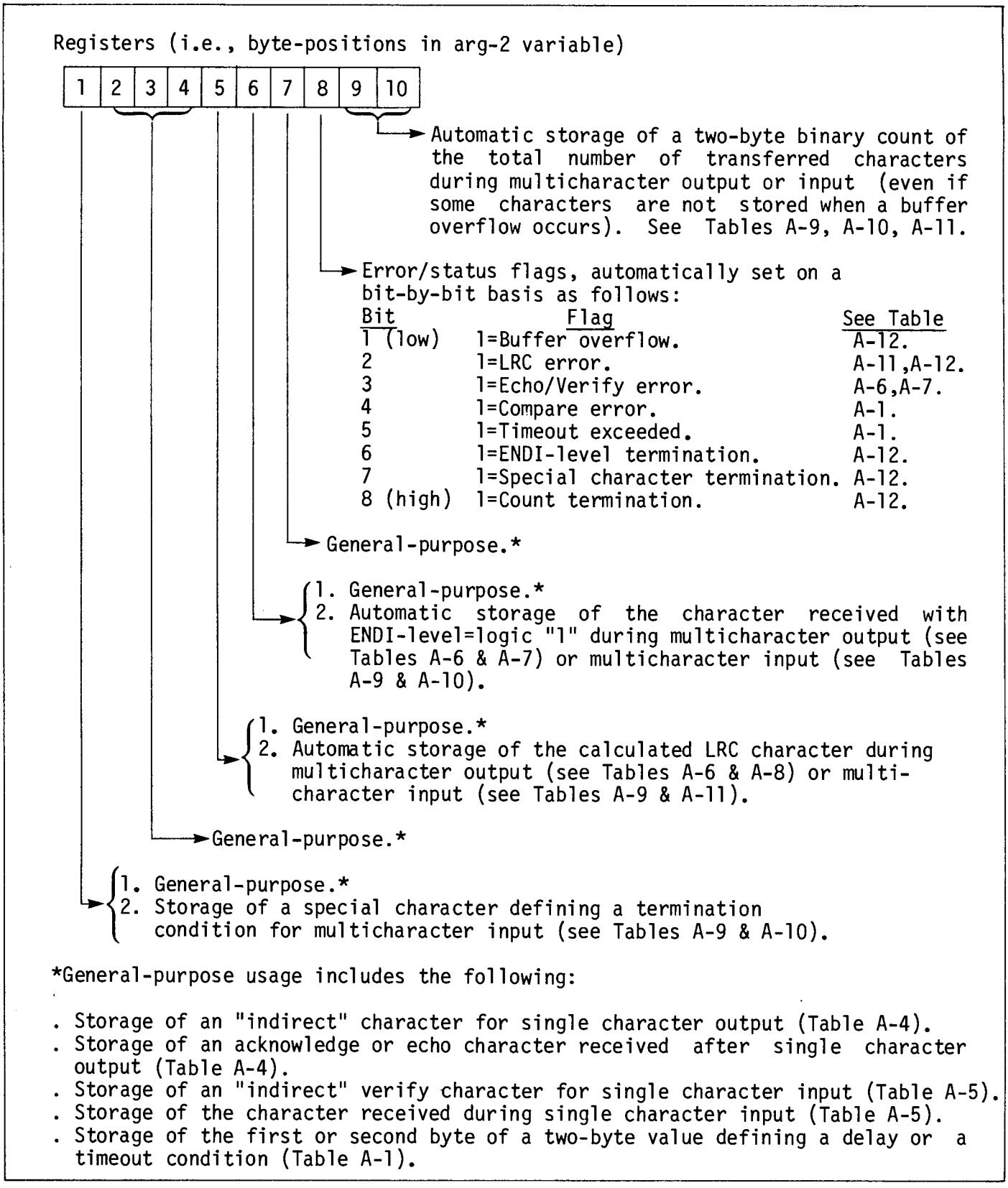
Each byte-position in an arg-2 memory area is called a register, and the arg-2 variable is referred to as the "error/status/general-purpose registers" to indicate the variable's multipurpose function. For example, Register 1 (the first byte-position) can be used to store a special character defining a termination condition for a multicharacter input operation; the termination character might be $(0D)_{16}$, i.e., a carriage return code, or perhaps $(13)_{16}$, i.e., an X-OFF code (if a punched tape is to be read), or some other character. On the other hand, Register 1 can be used for other purposes if a different operation is being programmed.

Registers 2 and 3 (or another pair) can be used to store a two-byte binary value defining a delay or a timeout condition associated with control microcommands of the form $12h_3 1$ and $12h_3 2$ in Table A-1. However, these registers can be used for other purposes also since valid usage of the first seven registers depends upon the type of operation being programmed and the alternatives available to the programmer.

Register 8 is used by the system, on a bit-by-bit basis, to set error/status flags and is not available to the programmer for other uses. Similarly, Registers 9 and 10 are used by the system and are not available to the programmer.

To clarify the relationships between the arg-2 registers and the arg-1 microcommands for a $GIO statement, Figure 3-1 presents a summary of register usage applicable to all but a small set of $GIO statements, i.e., to any statement except one having a microcommand of the form $Fh_2 h_3 h_4$. To be totally meaningful, the summary in Figure 3-1 should be studied in conjunction with the microcommand tables in Appendix A. The figure shows the registers accessed by the system for particular operations and the registers available to a programmer.

The error/status/general-purpose registers, together with the microcommands, provide great flexibility when a programmer is custom-tailoring an input or an output operation. Furthermore, information stored in the registers by the system during execution of a $GIO statement (e.g., error flags or a character count) can be tested, if desired, after execution is completed.



*General-purpose usage includes the following:

- Storage of an "indirect" character for single character output (Table A-4).
- Storage of an acknowledge or echo character received after single character output (Table A-4).
- Storage of an "indirect" verify character for single character input (Table A-5).
- Storage of the character received during single character input (Table A-5).
- Storage of the first or second byte of a two-byte value defining a delay or a timeout condition (Table A-1).

Figure 3-1. Register Usage for Any $GIO Statement Except One Having a Microcommand of the Form $Fh_2h_3h_4$

Initializing $GIO Registers

Registers 8, 9 and 10 in the arg-2 variable are initialized to binary zero $(00)_{16}$ automatically when execution of a $GIO statement begins. The other registers are not initialized automatically.

To initialize specific registers, either one (or both) of the following methods can be employed:

1. Use string (STR) functions to store information in particular registers prior to execution of a $GIO statement. See Example 3-1.

2. Use control microcommands of the form $Oh_2h_3h_4$ to store information in particular registers during execution of a $GIO statement. See Example 3-2 and Table A-1.

Keep in mind, however, that the rationale for initializing any register depends upon the kind of operation being customized.

As an illustration, let's assume data are to be read from a punched tape until an X-OFF code, $(13)_{16}$, is encountered. Fundamentally, such an application involves multicharacter input with a special termination character. If a microcommand of the form $Ch_2h_3h_4$ (where $h_2 \neq 2$ and $h_3 = 0, 1, 3, 5$ or 7) is chosen from Table A-9, the signal sequence generated by the microcommand includes "check T1" and "check T2" steps, at least one of which tests each input character with respect to a special termination character. According to Table A-10, the special termination character must be stored in arg-2 Register 1. Example 3-1 demonstrates how the special character can be stored prior to execution of the $GIO statement, and Example 3-2 demonstrates storage during execution.

Example 3-1. Initializing a $GIO Register Prior to Statement Execution

```
10 DIM R$10, I$(200)60
20 STR(R$,1,1) = HEX(13)
30 $GIO (C630, R$) I$()
```

In Line 20, the string function stores the code $(13)_{16}$ i.e., an X-OFF character in the first byte position of R$. Observe in Line 30 that R$ is the arg-2 component of the $GIO statement. The dimensioned length of R$ is 10 bytes, according to Line 10.

Only one microcommand C630 is used in the arg-1 component of the $GIO statement (see Line 30). Observe from Table A-9, that a microcommand of the form $C6h_3h_4$ implements multicharacter input with a particular signal sequence which may or may not be suitable to a particular device.

Example 3-2. Initializing a $GIO Register During Statement Execution

```
10 DIM R$10, I$(200)60
20 $GIO (0113 C630, R$) I$()
```

The microcommand 0113 in Line 20 of Example 3-2 is of the form $0h_2h_3h_4$, where h_2 =1 specifies the particular register (i.e., Register 1) in which the character denoted by h_3h_4 = 13 is to be stored.

Example 3-3. Testing the Status Code in Register 8

If desired, the status code in Register 8 (i.e., the individual bits) can be tested after a $GIO operation is executed by using a string function. For example,

```
200 $GIO READ (6C01 4400 C221 8601, R$) B$
210 IF STR(R$,8,1) = HEX(00) THEN 250
220 IF STR(R$,8,1) = HEX(20) THEN 400
```

Also, if preferred, individual bits in Register 8 can be tested during execution of a $GIO statement by including one or more microcommands of the form $16h_3h_4$ or $17h_3h_4$ in the arg-1 microcommand sequence. See Table A-1.

Example 3-4. Testing the Count in Registers 9 and 10

After a $GIO multicharacter input operation is completed, the binary count stored in Registers 9 and 10 can be tested and/or converted to a numeric value in floating point format by using the STR and VAL functions. For example,

```
300 $GIO READ (6C01 4400 C221 8601, R$) B$
310 C = 256*VAL(R$,9,1) + VAL(R$,10,1)
320 IF STR(R$,9,2) = HEX(0200) THEN 480
```

In Line 310, each byte of the two-byte binary count is converted to a floating point number using the VAL function. The floating point number obtained from Register 9 is multiplied by 256, added to the number obtained from Register 10, and the result stored in the numeric variable C.

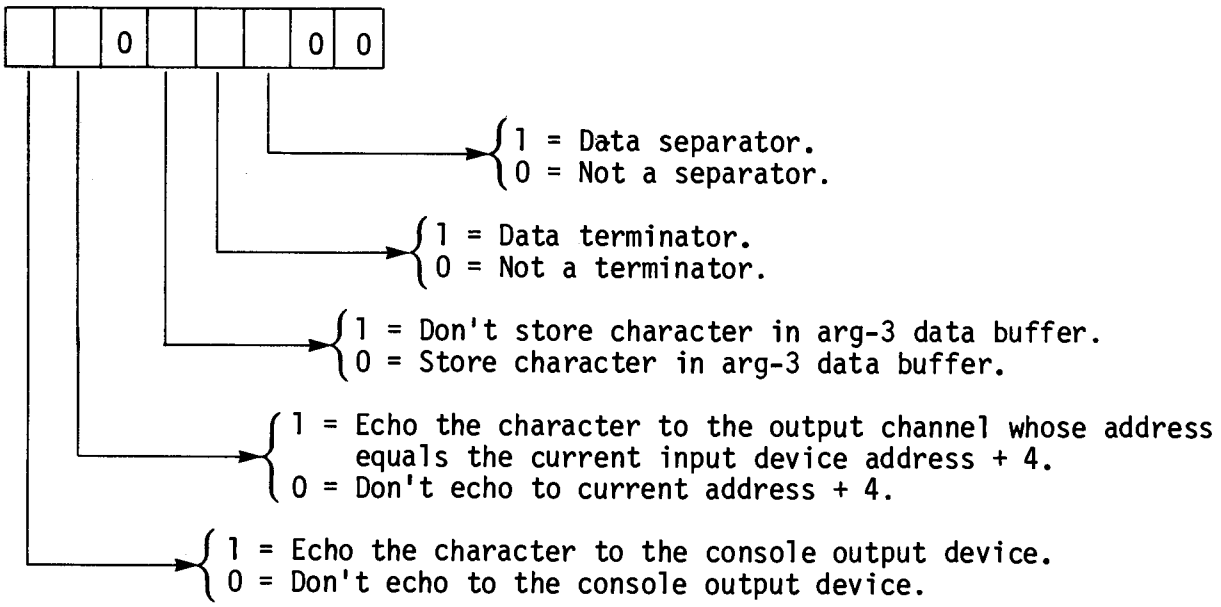
In Line 320, the two-byte binary count in Registers 9 and 10 is compared to $(0200)_{16}$, which is equivalent to $(512)_{10}$; then, if the count is equal to the decimal number 512, program execution branches to Line 480.

3.9 TELECOMMUNICATIONS (LINE ORIENTED) DATA INPUT

In Table A-13, the signal sequences for a set of microcommands of the form $Fh_2h_3h_4$ are described. If one of these multicharacter input microcommands is included in the arg-1 component of a $GIO statement, the dimensioned length of the arg-2 variable must be at least 12 bytes. See Figure 3-2.

Beginning with Register 11 in the arg-2 variable, a special character list can be stored. The list can include one or more data terminators and one or more data separators. Two registers are needed to define each special character since an "atom" specifying the action to be associated with a particular character must precede the character. The list must end with two space characters.

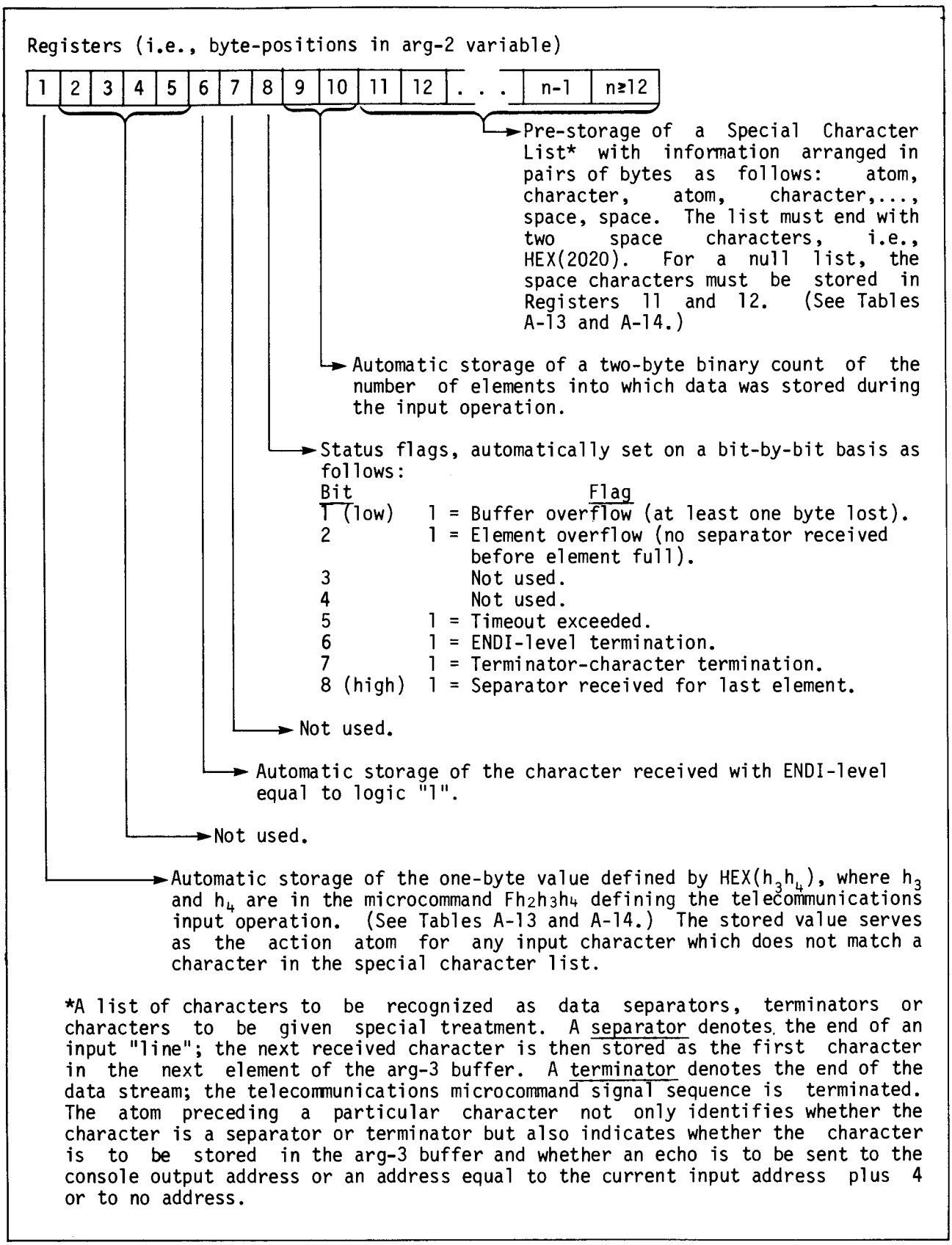
Table A-14 shows which bit-positions in an atom must be set to "1" to define particular actions. The information in the table can be presented schematically as follows:



If an input character does not match a character in the special character list, the hexdigits h_3h_4 in the microcommand $Fh_2h_3h_4$ serve as the atom defining the action to be taken. This "general" atom is stored in Register 1. (The system overrides any value other than zero for the low order hexdigit h_4 when storing the atom in Register 1.)

Registers 8, 9 and 10 are initialized to binary zero $(00)_{16}$ when execution of a $GIO statement begins. Other registers such as those used to define the special character list must be initialized by the programmer.

Figure 3-2 presents a summary of register usage for any $GIO statement which contains a microcommand of the form $Fh_2h_3h_4$. Table A-15 indicates where register usage for a "telecommunications" microcommand differs from the register usage shown in Figure 3-1 for all other $GIO statements.



*A list of characters to be recognized as data separators, terminators or characters to be given special treatment. A separator denotes the end of an input "line"; the next received character is then stored as the first character in the next element of the arg-3 buffer. A terminator denotes the end of the data stream; the telecommunications microcommand signal sequence is terminated. The atom preceding a particular character not only identifies whether the character is a separator or terminator but also indicates whether the character is to be stored in the arg-3 buffer and whether an echo is to be sent to the console output address or an address equal to the current input address plus 4 or to no address.

Figure 3-2. Register Usage for a $GIO Statement Having a Microcommand of the Form $Fh_2h_3h_4$.

3.10 PROGRAMMING $GIO OPERATIONS

Programming $GIO operations is similar to machine language programming since the $GIO microcommands are similar to machine language codes. Each microcommand represents a fundamental operation, usually multi-step. For some applications only one microcommand may be needed in a $GIO statement. For most applications several microcommands arranged in a particular sequence can produce the desired operation.

Anyone planning to program $GIO operations should be thoroughly familiar with the organization of the microcommand tables in Appendix A.

Before an appropriate microcommand sequence can be chosen, the application and hardware requirements must be defined and related to the inherent features of the available microcommands. Consideration must be given to questions such as the following:

- Is a multicharacter output operation required?
  ... If so, determine which microcommand in Table A-6 most fits the hardware.

- Is a multicharacter input operation required?
  ... If so, examine Tables A-9 and A-13.

- Are there timing considerations?
  ... If so, examine Table A-1 where the codes for setting delays or timeouts are discussed.

- Should more than one device be accessed by a single $GIO statement?
  ... If so, examine the address strobe codes in Table A-4.

- Should a single character be output at some stage in the operation?
  ... If so, examine Table A-4.

- Should a single character be received at some stage in the operation?
  ... If so, examine Table A-5.

Notes:

1. Tables A-4, A-5, A-6, A-9 and A-13 contain pairs of codes (e.g., $40h_3h_4$ and $44h_3h_4$, $B2h_3h_4$ and $B6h_3h_4$) whose signal sequences are identical in every respect but one -- the CPU outputs a character with an OBS strobe in one case and with a CBS strobe in the other case. For most hardware, the signal sequence with an OBS strobe rather than a CBS strobe is the appropriate choice.

2. The microcommands whose signal sequences output characters with an OBS strobe rather than a CBS strobe are appropriate for CRT's, printers, and many devices interfaced to a System 2200 CPU via controllers other than the Model 2250 I/O Interface Controller. The connector on the Model 2250 controller supports both OBS and CBS strobes; therefore, a programmer must know how an interfaced device is connected to such controllers before making decisions regarding the two types of output strobes.

Any reader who has used only high-level programming languages like Wang's BASIC language (excluding $GIO) may not be prepared for the extra effort needed to coordinate microcommands with hardware characteristics. Actually, the $GIO statement represents two levels of programming. The statement itself fits into the framework of the high-level BASIC language, except for its arg-1 component; selecting the microcommands for the arg-1 component (i.e., customizing the I/O operation) is equivalent to using a lower-level language. See the examples in Section 3.11.

3.11 SOME $GIO EXAMPLES

Some $GIO microcommand sequences are presented in the examples which follow. Two examples show how the CRT and keyboard can be used to test the function of some microcommands.

Example 3-5. A Multicharacter Output Operation

The microcommand sequence in Line 100 illustrates use of a delay, deselection of the current address and selection of another address, and output of particular characters as well as multicharacter output from the array B$(). The variable A$ serves as the error/status/general-purpose registers. Each microcommand is described briefly below.

```
100 $GIO /01D (0202 0300 4011 1221 7105 4000 711D 1200 A000 4013, A$) B$()
```

Microcommand functions:

0202 Store the character $(02)_{16}$ in Register 2.

0300 Store the character $(00)_{16}$ in Register 3.

4011 Send the character $(11)_{16}$ to the currently selected address, i.e., 1D.

1221 Set a delay condition equal to 50 microseconds multiplied by the two-byte binary value stored in Registers 2 and 3.

7105 Deselect the current address (i.e., 1D) and select the address 05 (the CRT).

4000 Send the character $(00)_{16}$ to the currently selected address, i.e., 05.

711D Deselect the current address 05 and select the address 1D.

1200 Disable the delay specified by the microcommand 1221.

A000 Output each character in the buffer using the following sequence:

WR = wait for a ready signal from the enabled device.
DATAOUT/OBS = send the next character with an OBS strobe.
LEND = the LRC End sequence specified by h_4=0 (i.e., none).

4013 Send the character $(13)_{16}$ to the currently selected address, i.e., 1D.

The microcommand sequence above might be suitable for output of data to a punch tape unit. The microcommand 4011 sends an X-ON character to the unit. The microcommands 1221, 7105 and 4000 send a null character to the CRT (after a delay of 25600 microseconds). A delay is introduced to allow the punch unit motor time to reach a specified condition. The delay is disabled before the multicharacter output operation begins.

Example 3-6. CRT Output Using PRINT and $GIO Statements

Duplicate and run the following program sequence:

```
10 DIM A$5
20 A$ = "ABCDE"
30 SELECT PRINT 005
40 PRINT A$;
50 PRINT "***"
60 $GIO /005 (A000, R$) A$
70 PRINT "@@@"
```

The microcommand A000 in Line 60 implements multicharacter output to the device with preset address 05. Each character in the dimensioned length of the arg-3 buffer A$ is sent to the device with an OBS strobe. Line 60 duplicates the output produced by lines 30 and 40.

Several features of the PRINT statement are not demonstrated by Line 40. Remove Line 10 from the program and run the modified program. Observe that the PRINT statement does not output the trailing space characters stored in A$ which has a default length of 16 bytes if Line 10 is removed. Note that the microcommand A000 outputs all 16 characters in A$, including trailing spaces.

Now, restore line 10 to the program, remove the semicolon in line 40, and run the second modification. Observe that the output of Lines 40 and 60 is not identical since the PRINT statement automatically outputs a carriage-return character and a line-feed character if there is no trailing punctuation.

Replace Line 60 as follows and omit the semicolon in Line 40:

```
60 $GIO /005 (A000 400D 400A, R$) A$
```

Now, run the modified program and observe that the microcommand 400D sends a carriage-return character $(0D)_{16}$ and the microcommand 400A sends a line-feed character $(0A)_{16}$.

Example 3-7. Keyboard Input Using INPUT and $GIO Statements

The following program sequence demonstrates a $GIO multicharacter input operation:

```
10  SELECT INPUT 001
20  INPUT A$
30  HEXPRINT A$
40  PRINT "?";
50  $GIO /001 (010D C610, R$) B$
60  HEXPRINT B$
```

The INPUT statement in Line 20 automatically outputs a question mark and a space character to the CRT to indicate the system is awaiting input via the currently selected device. The statement accepts characters (for temporary storage in the CPU buffer) and automatically sends an echo of each received character to the CRT. An error message appears on the CRT if 191 characters are received without a carriage-return. Processing of the temporarily buffered data begins as soon as a carriage-return character is received. Since the default dimension of A$ is 16 bytes, only 16 characters are stored in A$.

In Line 50, the $GIO microcommand 010D stores the carriage-return character $(0D)_{16}$ in Register 1. The microcommand C610 implements a multicharacter input operation via the currently addressed device (see Table A-9). Since h_3=1, each input character is compared to the special termination character stored in Register 1. If more characters are received than can be stored in the arg-3 buffer, the buffer-overflow flag is set in Register 8. Observe that no echo is sent to the CRT.

There are many features built into an INPUT statement which are especially suited for keyboard input, such as permitting a subroutine to be called by a special function key. Example 3-7 is not presented to offer an alternative procedure for keyboard input. Hopefully, Examples 3-6 and 3-7 show readers how to test the features of some microcommands using the CRT as an output device and the keyboard as an input device.

APPENDIX A -- $GIO MICROCOMMAND TABLES

In this appendix, valid microcommand codes for $GIO operations are presented in a set of tables. Each table contains a group of codes which represent related operations.

Every microcommand code in Tables A-1 through A-7 and A-13 is represented by a four hexdigit code of the form $h_1h_2h_3h_4$, where each subscript denotes the position of a hexdigit in the code (in ascending order from the leftmost to the rightmost position). In the tables, the first hexdigit is always specified; the second hexdigit is usually specified. The third and fourth hexdigits must be specified after checking the valid options given in one or more of the auxiliary tables included in the appendix.

Tables A-2 and A-3 give general descriptions for codes related to single character transfer (Table A-2) and multicharacter transfer (Table A-3). Each table has a Remarks column which directs the user to other tables where the signal sequences are given. After a user is familiar with the fundamental types of I/O operations for which microcommand codes are available, Tables A-2 and A-3 can be bypassed.

Every table which gives the signal sequences corresponding to some microcommand codes is followed by a legend which defines the mnemonics appearing in the table. Definitions for some of the mnemonics refer to IBS input strobes, OBS output strobes, or CBS output strobes. For those readers to whom the term "strobe" is unfamiliar, a brief explanation follows.

A strobe is a short-duration change in the voltage level of a particular direct-current circuit. For example, in the circuitry of Wang's Model 2250 I/O Interface Controller, a logic "0" is represented by a "high" level signal (between +2.4 and 3.6 volts DC) and a logic "1" is represented by a "low" level signal (between 0 and +0.4 volts DC). As shown in Figure A-1, output strobes from a Wang CPU to an external device via the Model 2250 interface controller have a pulse width of 5 microseconds, plus or minus 10%. Input strobes from an external device to the Model 2250 interface must have a pulse width which lies in the range from 5 to 20 microseconds.

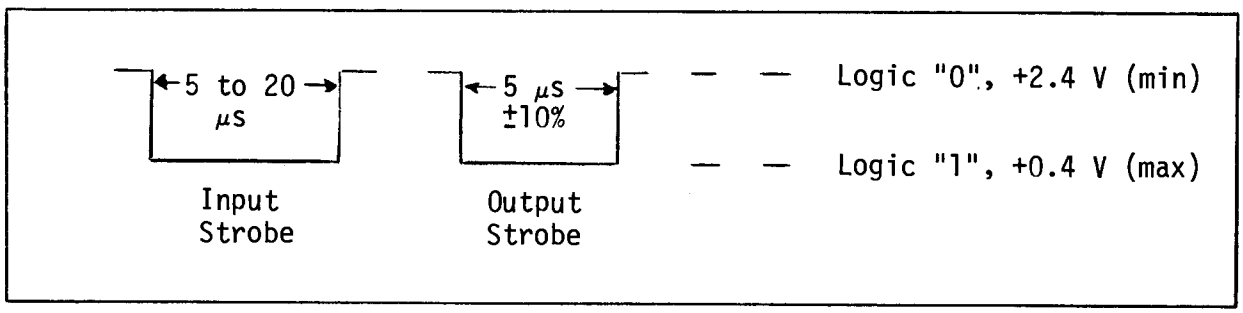


Figure A-1. Schematic of Input and Output Strobes for the Model 2250 Interface Controller

An input strobe received by the Model 2250 interface controller on a particular circuit from an external device indicates that data signals (8-bits in parallel, i.e., one byte) are available on other circuits, awaiting transfer into the CPU. Similarly, an output strobe sent from the CPU via the Model 2250 interface controller on a particular circuit indicates to an external device that data signals (8-bits in parallel) are available on other circuits, awaiting reception by the external device.

For some applications, bytes of information to be sent to a device fall in two classifications: (1) control data (e.g., instructions), and (2) output data to be stored. For this reason, the Model 2250 interface controller provides two different circuits for output strobes. On one circuit, an output strobe is designated as an OBS strobe (Output Data Strobe); on the other circuit, a strobe is designated as a CBS strobe (Control Output Strobe). Thus, a byte of information on the eight parallel data circuits can be identified by using either an OBS or a CBS strobe to indicate whether it is output data or control data.

If a microcommand table contains a code for a signal sequence which sends information via an OBS strobe, the table also contains another code for a signal sequence identical except for its use of a CBS strobe to send the information. When selecting microcommand codes to control a device interfaced via a Model 2250 controller, one must know whether both the CBS and OBS output strobe circuits are connected or only the OBS circuit.

If a device is interfaced to a Wang system via an interface which provides only one type of strobe for data output operations from the CPU, a $GIO microcommand sequence should consist of codes which send an OBS strobe.

Table A-1. Control Microcommands

Code	General Description	Function**
$0h_2h_3h_4$	Move Immediate Character to Register	Store the immediate character HEX(h_3h_4), specified by the pair of hexdigits h_3 and h_4, in Register h_2, specified by the hexdigit h_2.
$11h_3h_4$	Move Register to Register	Move the contents of specified Register h_3 to the specified Register h_4.
$14h_3h_4$	Compare Registers, Set Error	Compare the contents of specified Registers h_3 and h_4. If unequal, set the compare-error-bit in the status-code-register; i.e., set Position-bit-08 in Register 8 to "1".
$15h_3h_4$	Compare Registers, Terminate	Compare the contents of specified Registers h_3 and h_4. If unequal, set the compare-error-bit in the status-code-register and also terminate execution of the $GIO statement.
$16h_3h_4$	Compare Status-Code-Bits=0, Terminate	Test specified bits in the status-code-register for a "0" by executing a logical AND operation using the complement of the status-code-register and the mask specified by the binary equivalent of HEX(h_3h_4). If any specified bit is "0"; i.e., the result of the logical AND is not equal to HEX(00), terminate execution of the $GIO statement.
$17h_3h_4$	Compare Status-Code-Bits=1, Terminate	Test specified bits in the status-code-register for a "1" by executing a logical AND operation using the 8-bit code in the status-code-register and the mask specified by the binary equivalent of HEX(h_3h_4). If any specified bit is "1"; i.e., the result of the logical AND is not equal to HEX(00), terminate execution of the $GIO statement.
$12h_31$	Set Delay Condition for Output Strobes*	Prior to executing each subsequent output strobe (except an ABS strobe, if any) in the $GIO microcommand sequence, introduce a delay defined by the two-byte binary value stored in Registers h_3 and (h_3+1), where h_3 cannot exceed 6. The delay is equal to 50 microseconds multiplied by the decimal equivalent of the two-byte binary value; the minimum delay (50 microseconds) is specified by storing HEX(0001) in the designated registers; the maximum delay (approximately 3.28 seconds) is specified by storing HEX(FFFF). A HEX(0000) disables the delay condition.
$12h_32$	Set Timeout Condition for Sensing a Device Ready Signal or an Input Strobe*	Prior to executing each subsequent sensing operation for either a device-ready-signal or an input-strobe, implement a timeout defined by the two-byte binary value stored in Registers h_3 and (h_3+1), where h_3 cannot exceed 6. If an awaited device-ready-signal or input-strobe is not sensed within the allotted time span, set the timeout-array-bit in the status-code-register. The timeout is equal to 1 millisecond multiplied by the decimal value of the two-byte binary number; e.g., the minimum timeout (1 millisecond) is specified by storing HEX(0001) in the designated registers; the maximum timeout (approximately 65.5 seconds) is specified by storing HEX(FFFF). A HEX(0000) disables the timeout condition.
1200	Reset Delay/Timeout Condition	Disable the delay or the timeout condition specified by a prior microcommand of the form $12h_31$ or $12h_32$ in the $GIO sequence.

*Either a delay or a timeout condition, not both, can be active at one time. Specifying one condition disables the other condition.

**** See Figures 3-1 & 3-2 for a definition of register usage in the specified $GIO arg-2 variable.**

Table A-2. I/O Microcommands for Single Character Transfer

Code	General Description	Function (Optional and Automatic Features)	Remarks
$4h_2h_3h_4$	Single Character Output	1. Wait or don't wait for a Ready signal from the device. 2. Send an OBS or CBS output strobe with the specified immediate or indirect character.	See Table A-4
$5h_2h_3h_4$	Single Character Output with Acknowledge	(Steps 1 and 2 are the same as those for codes with h_1=4.) 3. Wait five microseconds. 4. Set the CPU Ready/Busy signal to Ready. 5. Await an input strobe with a character from the device. 6. Save or don't save the received character.	See Table A-4
$6h_2h_3h_4$	Single Character Output with Echo	(Steps 1 through 6 are the same as those for codes with h_1=5.) 7. Verify the received character by comparison with the output character; and, if unequal, set the echo-error-bit in the status-code-register. 8. Terminate or don't terminate execution of the $GIO statement if the received character and output character are unequal.	See Table A-4
$7h_2h_3h_4$	Address Strobe	1. Send an address strobe to the I/O bus with the specified immediate or indirect 8-bit device address code.	See Table A-4
$86h_3h_4$	Single Character Input	1. Set the CPU Ready/Busy signal to Ready. 2. Await an input strobe with a character from the device. 3. Save or don't save the received character.	See Table A-5
$8h_2h_3h_4$	Single Character Input with Verify	(Steps 1 through 3 are the same as those for codes with h_1=8, h_2=6.) 4. Verify the received character by comparison with the specified immediate or indirect character; and, if unequal, set the echo-error-bit in the status-code-register. 5. Terminate or don't terminate execution of the $GIO statement if the received character and the specified character are unequal.	See Table A-5
$9h_2h_3h_4$	Single Character Input with Echo	(Steps 1 through 3 are the same as those for codes with h_1=8, h_2=6.) 4. Wait or don't wait for a Ready signal from the device. 5. Send an echo of the received character using an OBS or CBS output strobe.	See Table A-5

Table A-3. I/O Microcommands for Multicharacter Transfer to or from the Specified $GIO Buffer

Code	General Description	Function (Optional and Automatic Features)	Remarks
Ah_2Oh_4	Multicharacter Output	Sequentially output each character as follows: 1. Wait or don't wait for a Ready signal from the device. 2. Send out the next character from the $GIO buffer using an OBS or CBS output strobe. (Optional) 3. Repeat Steps 1 and 2 if the current character is not the last character in the $GIO buffer. 4. Execute the specified LRC End Sequence corresponding to hexdigit h_4.	See Table A-6
$Bh_2h_3h_4$ (h_2=0,1,4,or 5)	Multicharacter Output with Acknowledge	Sequentially output each character as follows: 1. Wait or don't wait for a Ready signal from the device. 2. Send out the next character from the $GIO buffer using an OBS or CBS output strobe. 3. Wait five microseconds. 4. Set the CPU Ready/Busy signal to Ready. 5. Await an input strobe from the device. 6. Execute the specified termination-condition-check corresponding to hexdigit h_3. 7. Repeat Steps 1 through 6 if the current character is not the last character in the $GIO buffer. 8. Execute the specified LRC End Sequence corresponding to hexdigit h_4.	See Table A-6
$Bh_2h_3h_4$ (h_2=2,3,6,or 7)	Multicharacter Output with Echo	Sequentially output each character as follows: (Steps 1 through 4 are the same as those for codes with h_1=B, h_2=0,1,4, or 5.) 5. Await an input strobe and echo character from the device. 6. Verify the received character by comparison with the output character; and, if unequal, set the echo-error-bit in the status-code-register. 7. Execute the specified termination-condition-check corresponding to hexdigit h_3. 8. Repeat Steps 1 through 7 if the current character is not the last character in the $GIO buffer. 9. Execute the specified LRC End Sequence corresponding to hexdigit h_4.	See Table A-6

(Table A-3 continued on next page)

Table A-3. (Continued from preceding page)

Code	General Description	Function (Optional and Automatic Features)	Remarks
$Bh_2h_3h_4$ (h_2=8,9,C, or D)	Requested Multicharacter Output	Sequentially output each character as follows: 1. Set the CPU Ready/Busy signal to Ready. 2. Await an input strobe from the device. 3. Execute the specified termination-condition-check corresponding to hexdigit h_3. 4. Wait or don't wait for a Ready signal from the device. 5. Send out the next character from the $GIO buffer using an OBS or CBS output strobe. 6. Repeat Steps 1 through 5 if the current character is not the last character in the $GIO buffer. 7. Execute the specified LRC End Sequence corresponding to hexdigit h_4.	See Table A-6
BAh_3O	Multicharacter Verify	Sequentially verify each character in the $GIO buffer as follows: 1. Set the CPU Ready/Busy signal to Ready. 2. Await an input strobe and character from the device. 3. Verify the next character in the buffer by comparison with the received character; and, if unequal, set the echo-error-bit in the status-code-register. 4. Execute the specified termination-condition-check corresponding to hexdigit h_3. 5. Repeat Steps 1 through 4 if the current character is not the last character in the $GIO buffer.	See Table A-6
$C6h_3h_4$	Multicharacter Input	Sequentially receive and store characters until one or more specified termination conditions are satisfied. 1. Set the CPU Ready/Busy level to Ready. 2. Await an input strobe and character from the external device. 3. Execute the specified termination-condition-check, corresponding to hexdigit h_3 (if h_3=2,3,6, or 7 for an ENDI-level-check or if h_3=1,3,5, or 7 for a special-character-not-to-be-stored-check). 4. Save the input character in the next byte of the $GIO buffer (unless the character is ruled out by Step 3). 5. Execute the specified termination-condition-check corresponding to hexdigit h_3 (if h_3=0 for a special-character-to-be-stored-check or if h_3 =4,5,6, or 7 for a special-count-condition-check). 6. Repeat preceding steps until a valid termination condition is sensed. 7. Execute the specified LRC End Sequence corresponding to hexdigit h_4. (A special high-speed version is available. See code $C22h_4$ in Table A-9.)	See Table A-9
$Ch_2h_3h_4$ (h_2=0,1,4,5)	Multicharacter Input with Echo	Sequentially receive and store characters (echoing each one to the input device) until a valid termination condition is detected. (Steps 1 and 2 are the same as the events for micro-commands with h_1=C, h_2=6.) 3. Wait or don't wait for a Ready signal from the device. 4. Send an echo of the received character using an OBS or CBS output strobe. (Steps 5,6,7,8, and 9 are the same as Steps 3 through 7 for $C6h_3h_4$ microcommands.)	See Table A-9
$Ch_2h_3h_4$ (h_2=8 thru F)	Multicharacter Input Upon Request	After first requesting each character by sending an OBS or CBS strobe to the device, sequentially receive and store characters until one or more specified termination conditions are satisfied. 1. Set or don't set the CPU Ready/Busy level to Ready. 2. Wait or don't wait for a Ready signal from the device. 3. Send an OBS or CBS "request" strobe to the device.. 4. Wait for five microseconds. 5. Set or don't set the CPU Ready/Busy level to Ready. 6. Await an input strobe and character from the device. (Steps 7,8,9,10, and 11 are the same as Steps 3 through 7 for codes with h_1=C, h_2=6.)	See Table A-9
$Fh_2h_3h_4$	Telecommunications (Line-oriented) Input	Sequentially receive and store characters, after first masking the high-order eighth bit, if specified, and checking the resulting character for a match in the "special character list" stored in the $GIO arg-2 variable beginning with the eleventh byte. Atoms in the list (see Table A-14) define the action to be taken if a received character matches a special character. The hexdigits h_3h_4 specify the action-atom for all characters not in the list. Received data is stored in the $GIO arg-3 buffer which can be divided into elements by an alpha array modifier of the form <s,m,e>. Each "line" of data up to a separator character is stored in the next available element. Several methods of input termination are possible.	See Table A-13

Table A-4. Single Character Output Signal Sequences

Code	Signal Sequence	Character To Be Sent	Character To Be Saved	Type of Operation
$40h_3h_4$	WR, OBS/IMM	HEX(h_3h_4)		
$41h_3h_4$	OBS/IMM	HEX(h_3h_4)		
$42h_30$	WR, OBS/IND	Is in Register h_3		
$43h_30$	OBS/IND	Is in Register h_3		Single Character Output
$44h_3h_4$	WR, CBS/IMM	HEX(h_3h_4)		
$45h_3h_4$	CBS/IMM	HEX(h_3h_4)		
$46h_30$	WR, CBS/IND	Is in Register h_3		
$47h_30$	CBS/IND	Is in Register h_3		
			(Acknowledge)	
$50h_3h_4$	WR, OBS/IMM, W5, CPB, IBS	HEX(h_3h_4)	no	
$51h_3h_4$	OBS/IMM, W5, CPB, IBS	HEX(h_3h_4)	no	
$52h_3h_4$	WR, OBS/IND, W5, CPB, IBS, SAVE	Is in Register h_3	In Register h_4	
$53h_3h_4$	OBS/IND, W5, CPB, IBS, SAVE	Is in Register h_3	In Register h_4	Single Character Output with Acknowledge
$54h_3h_4$	WR, CBS/IMM, W5, CPB, IBS	HEX(h_3h_4)	no	
$55h_3h_4$	CBS/IMM, W5, CPB, IBS	HEX(h_3h_4)	no	
$56h_3h_4$	WR, CBS/IND, W5, CPB, IBS, SAVE	Is in Register h_3	In Register h_4	
$57h_3h_4$	CBS/IND, W5, CPB, IBS, SAVE	Is in Register h_3	In Register h_4	
			(Echo)	
$60h_3h_4$	WR, OBS/IMM, W5, CPB, IBS, VERIFY	HEX(h_3h_4)	no	
$61h_3h_4$	OBS/IMM, W5, CPB, IBS, VERIFY	HEX(h_3h_4)	no	
$62h_3h_4$	WR, OBS/IND, W5, CPB, IBS, SAVE, VERIFY	Is in Register h_3	In Register h_4	
$63h_3h_4$	OBS/IND, W5, CPB, IBS, SAVE, VERIFY	Is in Register h_3	In Register h_4	
$64h_3h_4$	WR, CBS/IMM, W5, CPB, IBS, VERIFY	HEX(h_3h_4)	no	
$65h_3h_4$	CBS/IMM, W5, CPB, IBS, VERIFY	HEX(h_3h_4)	no	
$66h_3h_4$	WR, CBS/IND, W5, CPB, IBS, SAVE, VERIFY	Is in Register h_3	In Register h_4	
$67h_3h_4$	CBS/IND, W5, CPB, IBS, SAVE, VERIFY	Is in Register h_3	In Register h_4	Single Character Output with Echo
$68h_3h_4$	WR, OBS/IMM, W5, CPB, IBS, VERIFY, TERM	HEX(h_3h_4)	no	
$69h_3h_4$	OBS/IMM, W5, CPB, IBS, VERIFY, TERM	HEX(h_3h_4)	no	
$6Ah_3h_4$	WR, OBS/IND, W5, CPB, IBS, SAVE, VERIFY, TERM	Is in Register h_3	In Register h_4	
$6Bh_3h_4$	OBS/IND, W5, CPB, IBS, SAVE, VERIFY, TERM	Is in Register h_3	In Register h_4	
$6Ch_3h_4$	WR, CBS/IMM, W5, CPB, IBS, VERIFY, TERM	HEX(h_3h_4)	no	
$6Dh_3h_4$	CBS/IMM, W5, CPB, IBS, VERIFY, TERM	HEX(h_3h_4)	no	
$6Eh_3h_4$	WR, CBS/IND, W5, CPB, IBS, SAVE, VERIFY, TERM	Is in Register h_3	In Register h_4	
$6Fh_3h_4$	CBS/IND, W5, CPB, IBS, SAVE, VERIFY, TERM	Is in Register h_3	In Register h_4	
$71h_3h_4$	ABS/IMM	HEX(h_3h_4)		Address Strobe*
$73h_30$	ABS/IND	Is in Register h_3		

*Codes of the form $7h_2h_3h_4$, where h_2=1 or 3, can be used as often as desired in a microcommand sequence to deselect the currently selected device address and select another address. The "immediate" or "indirect" address sent with an ABS strobe must be the preset address of the controller board for the device being selected. The preset address is the 8-bit code (one byte) corresponding to the last two hexdigits of the three-hexdigit-address-code usually used to select a particular device.

Legend for Table A-4.

WR = The CPU awaits a Ready signal from the enabled device.

OBS = The CPU sends an OBS output strobe.

CBS = The CPU sends a CBS output strobe.

/IMM = An "immediate" character (the two hexdigit code specified by h_3h_4) is sent with the output strobe. For example, if h_3h_4 = 3F in the microcommand code, the character HEX(3F) is sent out.

/IND = An "indirect" character (the character stored in the arg-2 register specified by h_3) is sent with the output strobe. For example, if h_3 = 1 in the microcommand code, the character stored in Register 1 is sent out.

W5 = The CPU waits five microseconds.

CPB = The CPU sets its Ready/Busy signal level to Ready.

IBS = The CPU awaits an input strobe from the enabled device.

SAVE = The character received with the input strobe is stored in the arg-2 register specified by h_4. For example, if h_4 = 2 in the microcommand code, the received character is stored in Register 2. (If h_4 = 0, the character is not stored.)

VERIFY = The character received with the input strobe is compared to the character sent with the output strobe. If the compared characters are unequal, set the echo-error-bit (bit-position-04 in arg-2 Register 8).

TERM = If the compared characters are unequal, immediately terminate execution of the $GIO statement.

ABS/IMM = The CPU sends an ABS address strobe to the I/O-bus with the "immediate" address specified by the two hexdigit code h_3h_4 in the microcommand code.

ABS/IND = The CPU sends an ABS address strobe to the I/O-bus with the "indirect" address stored in the arg-2 register specified by h_3 in the microcommand code.

Table A-5. Single Character Input Signal Sequences

Code	Signal Sequence	Compare Character	Received Character to be saved	Type of Operation
8600	CPB, IBS		no	Single Character Input
$860h_4$	CPB, IBS, SAVE		In Register h_4	
$80h_3h_4$	CPB, IBS, VERIFY/IMM	HEX(h_3h_4)	no	
$82h_3h_4$	CPB, IBS, SAVE, VERIFY/IND	Is in Register h_3	In Register h_4	Single Character Input
$88h_3h_4$	CPB, IBS, VERIFY/IMM, TERM	HEX(h_3h_4)	no	with Verify
$8Ah_3h_4$	CPB, IBS, SAVE, VERIFY/IND, TERM	Is in Register h_3	In Register h_4	
9200	CPB, IBS, WR, ECHO/OBS		no	
9300	CPB, IBS, ECHO/OBS		no	
$920h_4$	CPB, IBS, SAVE, WR, ECHO/OBS		In Register h_4	
$930h_4$	CPB, IBS, SAVE, ECHO/OBS		In Register h_4	Single Character Input
9600	CPB, IBS, WR, ECHO/CBS		no	with Echo
9700	CPB, IBS, ECHO/CBS		no	
$960h_4$	CPB, IBS, SAVE, WR, ECHO/CBS		In Register h_4	
$970h_4$	CPB, IBS, SAVE, ECHO/CBS		In Register h_4	

Legend for Table A-5

CPB = The CPU sets its Ready/Busy signal level to Ready.

IBS = The CPU awaits an input strobe from the enabled device.

SAVE = The character received with the input strobe is stored in the arg-2 register specified by h_4. For example, if h_4 = 3 in the microcommand code, the received character is stored in Register 3. (If h_4 = 0, the character is not stored.)

VERIFY/IMM=The character received with the input strobe is compared to the "immediate" character specified by the two hexdigit code h_3h_4 in the microcommand code. If the compared characters are unequal, set the echo-error-bit (bit-position-04 in arg-2 Register 8).

VERIFY/IND=The character received with the input strobe is compared to the "indirect" character stored in the arg-2 register specified by h_3 in the microcommand code. If the compared characters are unequal, set the echo-error-bit (bit-position-04 in arg-2 Register 8).

TERM = If the compared characters are unequal, immediately terminate execution of the $GIO statement.

WR = The CPU awaits a Ready signal from the enabled device.

ECHO/OBS = The CPU sends an echo of the received character with an OBS output strobe to the enabled device.

ECHO/CBS = The CPU sends an echo of the received character with a CBS output strobe to the enabled device.

Table A-6. Multicharacter Output Signal Sequences

Code	Signal Sequence	Check T Code*	Lend Code**	Type of Operation
$A0Oh_4$	(WR,DATAOUT/OBS), REPEAT,LEND		h_4	
$A1Oh_4$	(DATAOUT/OBS), REPEAT,LEND		h_4	Multicharacter
$A4Oh_4$	(WR,DATAOUT/CBS), REPEAT,LEND		h_4	Output
$A5Oh_4$	(DATAOUT/CBS), REPEAT,LEND		h_4	
$B0h_3h_4$	(WR,DATAOUT/OBS,W5,CPB,IBS, CHECK T), REPEAT,LEND	h_3	h_4	
$B1h_3h_4$	(DATAOUT/OBS,W5,CPB,IBS, CHECK T), REPEAT,LEND	h_3	h_4	Multicharacter
$B4h_3h_4$	(WR,DATAOUT/CBS,W5,CPB,IBS, CHECK T), REPEAT,LEND	h_3	h_4	Output with
$B5h_3h_4$	(DATAOUT/CBS,W5,CPB,IBS, CHECK T), REPEAT,LEND	h_3	h_4	Acknowledge
$B2h_3h_4$	(WR,DATAOUT/OBS,W5,CPB,IBS,VERIFY,CHECK T), REPEAT,LEND	h_3	h_4	
$B3h_3h_4$	(DATAOUT/OBS,W5,CPB,IBS,VERIFY,CHECK T), REPEAT,LEND	h_3	h_4	Multicharacter
$B6h_3h_4$	(WR,DATAOUT/CBS,W5,CPB,IBS,VERIFY,CHECK T), REPEAT,LEND	h_3	h_4	Output with Echo
$B7h_3h_4$	(DATAOUT/CBS,W5,CPB,IBS,VERIFY,CHECK T), REPEAT,LEND	h_3	h_4	
$B8h_3h_4$	(CPB,IBS,CHECK T,WR, DATAOUT/OBS), REPEAT,LEND	h_3	h_4	
$B9h_3h_4$	(CPB,IBS,CHECK T, DATAOUT/OBS), REPEAT,LEND	h_3	h_4	Requested Multi-
BCh_3h_4	(CPB,IBS,CHECK T,WR, DATAOUT/CBS), REPEAT,LEND	h_3	h_4	character Output
BDh_3h_4	(CPB,IBS,CHECK T, DATAOUT/CBS), REPEAT,LEND	h_3	h_4	(Each Character)
BAh_3O	(CPB,IBS,VERIFY,CHECK T), REPEAT	h_3		Multicharacter Verify
$A2Oh_4$	A high-speed version of $A0Oh_4$, allowing not more than 30μs per data character and not more than 45μs between final data character and LRC character. No timeout or delay is implemented in this version.		h_4	High-speed Output
$A6Oh_4$	SCAN DATA BUFFER, CALCULATE LRC, LEND		h_4	LRC Only Output

→ See Table A-8.
→ See Table A-7.

*See Table A-7.
**See Table A-8.

Legend for Table A-6.

(In this table, any sequence enclosed in parentheses is repeated for each character in the defined length of the arg-3 data buffer.)

WR = The CPU awaits a Ready signal from the enabled device.

DATAOUT/OBS = The CPU sends out the next character in the arg-3 buffer with an OBS output strobe.

DATAOUT/CBS = The CPU sends out the next character in the arg-3 buffer with a CBS output strobe.

W5 = The CPU waits five microseconds.

CPB = The CPU sets its Ready/Busy signal level to Ready.

IBS = The CPU awaits an input strobe from the enabled device.

VERIFY = The character received with the input strobe is compared to the character sent with the output strobe. If the compared characters are unequal, the echo-error-bit (i.e., bit-position-04 in arg-2 Register 8) is set to "1".

CHECK T = The CPU checks for the type of termination condition specified by hexdigit h_3 (see Table A-7) and executes a prescribed sequence if a valid condition is detected.

REPEAT = The output sequence within the parentheses is repeated for all characters in the defined length of the arg-3 buffer.

LEND = The LRC End Sequence specified by hexdigit h_4 is executed (see Table A-8).

Note:

When programming a multicharacter-output-with-echo operation, neither a timeout of the form $12h_32$ nor a delay of the form $12h_31$ can be in effect during execution of a microcommand of the form $B2h_3h_4$, $B3h_3h_4$, $B6h_3h_4$, or $B7h_3h_4$. A false indication of an echo error may occur if a timeout or delay is in effect.

Table A-7. Valid "Check T" Output Termination Codes for h_3 in Several Microcommand Categories

Output Termination Condition \ Microcommand Category	$Bh_2h_3h_4$ h_2=0,1,4,5	$Bh_2h_3h_4$ h_2=2,3,6,7	$Bh_2h_3h_4$ h_2=8,9,C,D	BAh_3O
None (go to next microcommand when buffer finished).	0	0	0	8
Terminate statement if echo or acknowledge character verifies unequal.		1		9
Terminate statement if ENDI level = logic "1" when an input strobe is received. Also save the character received in Register 6 and set the ENDI-bit in the status-code-register.	2	2	2	A
Terminate statement if either termination condition is detected.		3		B

Table A-8. Valid "Lend" Codes for h_4 in Several Output Microcommands

LRC End Sequence \ Microcommand Category	$Ah_2h_3h_4$	$Bh_2h_3h_4$ h_2=0 through 7	$Bh_2h_3h_4$ h_2=8,9,C,D
None (go to next microcommand).	0	0	0
WR, SEND LRC/OBS, SAVE LRC	2	2	-
SEND LRC/OBS, SAVE LRC	3	3	-
WR, SEND LRC/CBS, SAVE LRC	6	6	-
SEND LRC/CBS, SAVE LRC	7	7	-
SAVE LRC	4	4	4

Legend for Table A-8.

WR = The CPU awaits a Ready signal from the enabled device.

SEND LRC/OBS = The CPU sends the calculated LRC* character with an OBS strobe to the enabled device.

SEND LRC/CBS = The CPU sends the calculated LRC* character with a CBS strobe to the enabled device.

SAVE LRC = The calculated LRC character is saved in arg-2 Register 5.

*The LRC character is calculated by executing an Exclusive OR for all characters in the data buffer.

Table A-9. Multicharacter Input Signal Sequences

Code	Signal Sequence	Check T Code*	Lend Code**	Type of Operation
$C6h_3h_4$	(CPB,IBS, CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	Multicharacter Input
$C0h_3h_4$	(CPB,IBS,WR,ECHO/OBS, CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
$C1h_3h_4$	(CPB,IBS, ECHO/OBS, CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	Multicharacter
$C4h_3h_4$	(CPB,IBS,WR,ECHO/CBS, CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	Input with Echo
$C5h_3h_4$	(CPB,IBS, ECHO/CBS, CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
$C8h_3h_4$	(WR, OBS,W5,CPB,IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
$C9h_3h_4$	(OBS,W5,CPB,IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
CAh_3h_4	(CPB, WR, OBS, IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
CBh_3h_4	(CPB, OBS, IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	Requested Multi-
CCh_3h_4	(WR, CBS,W5,CPB,IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	character Input
CDh_3h_4	(CBS,W5,CPB,IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	(Each Character)
CEh_3h_4	(CPB, WR, CBS, IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
CFh_3h_4	(CPB, CBS, IBS,CHECK T1,SAVE DATA,CHECK T2),REPEAT,LEND	h_3	h_4	
$C22h_4$	(CPB,IBS, no timeout or delay, CHECK ENDI,SAVE DATA),REPEAT, LEND		h_4	High-speed Input
	Note: $C22h_4$ is a special case, optimized for maximum speed -- a character may be received every 30 μs approximately.			

→ See Table A-11.
→ See Table A-10.

*See Table A-10.
**See Table A-11.

Legend for Table A-9.

(In this table, any sequence enclosed in parentheses is repeated until a valid termination condition is detected.)

CPB = The CPU sets its Ready/Busy signal level to Ready.

IBS = The CPU awaits an input strobe from the enabled device.

WR = The CPU awaits a Ready signal from the enabled device.

ECHO/OBS = The CPU sends an echo of the received character with an OBS strobe to the enabled device.

ECHO/CBS = The CPU sends an echo of the received character with a CBS strobe to the enabled device.

OBS = The CPU sends an OBS strobe to the enabled device to request an input character.

CBS = The CPU sends a CBS strobe to the enabled device to request an input character.

W5 = The CPU waits five microseconds.

CHECK T1 = The CPU checks for a termination condition by ENDI-level (if h_3=2,3,6,or 7) and/or by a Special Character not-to-be-saved (if h_3=1,3,5,or 7). See Table A-10.

SAVE DATA= The received character is saved in the arg-3 data buffer.

CHECK T2 = The CPU checks for a termination condition by a to-be-saved Special Character (if h_3 =0) or by the Count (if h_3=4,5,6,or 7). See Table A-10.

REPEAT = The input sequence within the parentheses is repeated until a valid termination condition is detected.

LEND = The LRC End Sequence specified by hexdigit h_4 is executed (see Table A-11).

Table A-10. Valid "Check T1" and "Check T2" Input Termination Codes for h_3 in Microcommands of the Form $Ch_2h_3h_4$

h_3	ENDI-level Termination (Termination when an input strobe is received with the ENDI signal level set to logic "1".)	Special Character Termination (Termination when an input character is equal to the character stored in arg-2 Register 1.)	Count Termination (Termination when the number of characters received is equal to the defined length of the arg-3 data buffer.)
0		yes*	
1		yes**	
2	yes***		
3	yes***	yes**	
4			yes
5		yes**	yes
6	yes***		yes
7	yes***	yes**	yes

*If h_3 =0, the special character is saved in the data buffer and included in both the LRC calculation and the character count.

**If h_3 =1,3,5,or 7, the special character is not saved in the data buffer; therefore, neither the LRC calculation nor the character count includes the special character.

***The character received (when the ENDI level is logic "1") is saved in arg-2 Register 6.

(Note: For h_3 =3,5,6,or 7, termination occurs when one or more of the specified conditions occurs. The ENDI level is checked first, then the Special Character condition, and then the Count.)

Table A-11. Valid "Lend" Codes for h_4 in Microcommands of the Form $Ch_2h_3h_4$

h_4	LRC End Sequence*
0	None (Go to next microcommand.)
1	Calculate the LRC of the input data (not including the special or ENDI character) and save in arg-2 Register 5.
2	Calculate the LRC of the input data (not including the ENDI character) and save in arg-2 Register 5. Also, compare the LRC with the ENDI character (i.e.,character received when ENDI level = logic "1"). If compared characters unequal, set the LRC-error-bit in arg-2 Register 8. (See Table A-12.)

*Use h_4=2 only with h_3=2. If $h_3 \neq 2$ and an ENDI character is not received, the LRC character is compared to the contents of arg-2 Register 6.

Table A-12. Definition of Error/Status Bits in Arg-2 Register 8

Bit Position	Error/Status Condition (If Bit Position Set To "1")
01	Buffer overflow (i.e., more characters received than could be stored in the arg-3 data buffer). The total count of characters received (whether stored or not) is saved as a two-byte binary number in arg-2 Registers 9 and 10. (This condition cannot occur if h_3=4,5,6,or 7 in microcommands of the form $Ch_2h_3h_4$, see Table A-10.)
02	LRC compare error. (This condition can occur only if h_4=2 in microcommands of the form $Ch_2h_3h_4$, see Table A-11.)
20	Termination by ENDI level = logic "1". *
40	Termination by Special Character. *
80	Termination by Count. *

*This bit position can be set only if the condition occurs when more than one type of termination condition is specified (see h_3 =3,5,6 and 7 in Table A-10).

Note:

The four high-order bit positions in a byte are labeled 80,40,20,10, respectively; the four low-order bit positions are labeled 8,4,2,1, respectively.

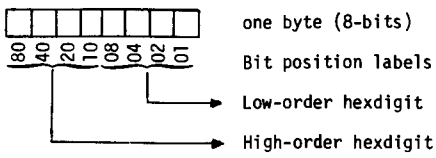


Table A-13. Telecommunications Input Signal Sequences

Code	Signal Sequence*
$F0h_3h_4$	(CPB,IBS, [WR,ECHO1/OBS],[WR,ECHO2/OBS],[SAVE DATA]),REPEAT
$F1h_3h_4$	(CPB,IBS, [ECHO1/OBS],[ECHO2/OBS],[SAVE DATA]),REPEAT
$F4h_3h_4$	(CPB,IBS, [WR,ECHO1/CBS],[WR,ECHO2/CBS],[SAVE DATA]),REPEAT
$F5h_3h_4$	(CPB,IBS, [ECHO1/CBS],[ECHO2/CBS],[SAVE DATA]),REPEAT
$F8h_3h_4$	(CPB,IBS,MASK,[WR,ECHO1/OBS],[WR,ECHO2/OBS],[SAVE DATA]),REPEAT
$F9h_3h_4$	(CPB,IBS,MASK,[ECHO1/OBS],[ECHO2/OBS],[SAVE DATA]),REPEAT
FCh_3h_4	(CPB,IBS,MASK,[WR,ECHO1/CBS],[WR,ECHO2/CBS],[SAVE DATA]),REPEAT
FDh_3h_4	(CPB,IBS,MASK,[ECHO1/CBS],[ECHO2/CBS],[SAVE DATA]),REPEAT

*The sequence inside each set of square brackets is optional, that is, execution may or may not occur after a received character (masked, if specified) is checked against each character in the "special character list" stored in arg-2 (beginning with the eleventh byte). Information in the special character list must be arranged in pairs of bytes as follows: atom**, character, atom, character,..., space, space. The list must end with two space characters, HEX(2020), or unpredictable results will occur. If the list is to be null, the eleventh and twelfth bytes must be blank. If a received character matches one of the characters in the list, the preceding atom defines the action to be taken (see Table A-14). If a received character does not match a special character, the hexdigits h_3h_4 in the microcommand serve as the atom defining the action to be taken for the character.

**An atom is an eight-bit code whose individual bits (if equal to "1") define a particular action. (See Table A-14.)

Legend for Table A-13.

CPB = The CPU sets its Ready/Busy signal level to Ready.

IBS = The CPU awaits an input strobe from the enabled device.

WR = The CPU awaits a Ready signal from the enabled device.

MASK = Set the high-order eighth bit of the received character to "0".

ECHO1/OBS = The CPU sends an echo of the received character with an OBS strobe to the console output device.

ECHO2/OBS = The CPU sends an echo of the received character with an OBS strobe to the output channel of the currently selected input device (i.e., to the device whose address is the input-device-address +4).

ECHO1/CBS = The CPU sends an echo of the received character with a CBS strobe to the console output device.

ECHO2/CBS = The CPU sends an echo of the received character with a CBS strobe to the output channel of the currently selected input device (i.e., to the device whose address is the input-device- address +4).

SAVE DATA = The received character is saved in the arg-3 data buffer.

REPEAT = The input sequence within the parentheses is repeated until a valid termination condition is detected.

Table A-14. Definition of Action Bits for Atoms in Special Character List

Bit Position	Action (If Bit Position Set to "1")
04	Character is a data "SEPARATOR" *
08	Character is a data "TERMINATOR" **
10	Do not store the character in the arg-3 data buffer.
40	Echo the character to the output channel whose address equals the current input device address +4.
80	Echo the character to the console output device.

*SEPARATOR = A character marking the end of an input "line"; the next received character is to be stored as the first byte in the next element of the arg-3 data buffer.

**TERMINATOR = A character marking the end of the input data; terminate execution of the data input.

(See the note following Table A-12 for a definition of the bit positions.)

Notes for Table A-13:

1. If an alpha array modifier is used when specifying the buffer for a microcommand chosen from Table A-13, the modifier must be of the form:

   <[s] , m, e>

   where s = starting byte, optional (default value=1),
   m = number of bytes per element (required),
   e = number of elements (required),
   and $\begin{Bmatrix} s \\ m \\ e \end{Bmatrix} = \begin{Bmatrix} \text{integer} \geq 1 \\ \text{expression} \\ \text{alpha variable} \end{Bmatrix}$

   If an alpha variable is used to define one of these parameters, the first two bytes in the variable are considered to be a 16-bit binary value; any remaining bytes of the variable are ignored.

2. The product of m and e (i.e., m*e) defines the total number of bytes available for use during execution of the microcommand.

3. The values of s, m, and e must not specify more bytes than exist in the array; therefore, the following condition must be satisfied by the values:

   $m*e - s + 1 \leq d$
   where d = the dimension of the array.

   For example, after the statement

   DIM B$(12)40

   reserves 480 contiguous bytes for the B$-array, the array modifiers below are interpreted as follows:

   B$() <81,40,1> is equivalent to B$(3),
   B$() <,80,6> splits the array into 6 80-byte elements,
   B$() <5,60,3> defines 3 60-byte elements, starting with the fifth byte of the array.

4. If the buffer is divided into elements by using an alpha array modifier, each "line" of data (denoted by a separator character) is stored in the next available element of the buffer. If a line overflows the element, the remaining characters are stored in the next element, and the element-overflow status bit is set (bit-position-02 in arg-2 Register 8). However, if there are no more elements, the remaining characters are lost, and the buffer-overflow status bit is set (bit-position-01 in Register 8).

5. A $GIO telecommunications input operation can be terminated by one or more of the following conditions:

   a) A "terminator" character is received (several different characters can be specified as data terminators).

   b) A "separator" character is received when data is being transferred to the last element in the buffer (several different characters can be specified as data separators).

   c) An input strobe with the ENDI level set to logic "1" is received; if so, the character is saved in arg-2 Register 6.

   d) A buffer overflow occurs.

   e) A timeout condition is exceeded (i.e., no character is received within a specified amount of time).

6. **Any address-strobe microcommand of the form $7h_2h_3h_4$ (see Table A-4) in the arg-1 component of a $GIO statement is ignored by a subsequent microcommand of the form $Fh_2h_3h_4$.**

Table A-15. Definition of Error/Status Registers for $GIO Operations with a Telecommunications Microcommand of the Form $Fh_2h_3h_4$

Byte Position	Bit Position	Usage
1*	all	Contains the second byte of the $Fh_2h_3h_4$ microcommand, i.e., the h_3h_4 code. **(With h_4 set to 0.)**
2,3,4,5	all	Not used.
6	all	Contains the ENDI-character, i.e., the character received with ENDI signal level = "1".
7	all	Not used.
8		Status code, set during $GIO execution as follows:
	01	1 = Buffer overflow (at least one byte lost).
	02*	1 = Element overflow (no separator code received before too many characters received for size of one element of the buffer).
	10	1 = Timeout (when active for microcommands of the Form $12h_3h_4$).
	20	1 = Data input termination by an ENDI-character
	40*	1 = Data input termination by a Terminator-character.
	80*	1 = Data input termination by a Separator-charactor received for last element of buffer.
9,10*	all	Contains the count (a two-byte binary value) of the number of elements into which input data was stored during execution of the $GIO statement.
11,...*		Contains the Special Character List, ending with two space characters, HEX(2020).

*Usage differs for non-telecommunications microcommands, i.e., microcommands where h_1 is not equal to F.

(See the note following Table A-12 for a definition of the bit positions in a byte.)

APPENDIX B -- ERROR CODES FOR THE GENERAL I/O INSTRUCTION SET

Presented in this section are five error codes which might occur when entering or running programs which include statements from the General I/O Instruction Set.

CODE 95

Error: Illegal Microcommand or Field/Delimiter Specification

Cause: The microcommand or field/delimiter specification is invalid.

Action: Use only the microcommands and field/delimiter specifications listed in the marual.

Example:
```
:RUN
:10 $GIO (1023, R$)
             ↑ERR 95
```

```
:10 $GIO (0123, R$)                    Possible Correction
```

CODE 96

Error: Missing Buffer

Cause: The $GIO arg-3 buffer was either omitted or already used by another multicharacter input, output, or verify microcommand.

Action: Define the buffer if it was omitted, or use two $GIO statements to separate multicharacter microcommands.

Example:
```
10 $GIO /03B (A000 C640, R$) B$
                      ↑ERR 96
```

```
10 $GIO /03B (A000, R1$) B1$
20 $GIO /03B (C640, R2$) B2$           Possible Correction
```

CODE 97

Error: Variable or Array Too Small

Cause: Insuffient space reserved in DIM statement or insuffient data for at least one argument in a $UNPACK statement.

Action: Change DIM statement or change argument list (or input more data) and run the program again.

Example:
```
10 DIM R$6
20 $GIO (0123, R$)

:RUN
:20 $GIO (0123, R$)
              ↑ERR 97
```

```
:10 DIM R$10                           Possible Correction
```

CODE 98

Error: Illegal Array Modifier Arguments

Cause: The arguments exceed the total size of the alpha array.

Action: Redefine the array modifier or redimension the array.

Example:
```
10 DIM A$(3)10, B$(4)64
20 $TRAN (A$() <10, 23 > , B$())
:RUN
:20 $TRAN (A$() <10, 23>, B$())
                          ↑ERR 98

:20 $TRAN (A$() <10, 20>, B$())
```
Possible Correction

CODE 59

Error: Illegal Alpha Variable For Sector Address (Standard message)

Cause: Alpha variable is not at least two bytes long.

Action: Redimension the alpha variable.

APPENDIX C - ASCII CONTROL AND GRAPHIC CHARACTERS IN HEXADECIMAL AND BINARY NOTATION

FORMATS:

HEXADECIMAL CODES: HEX (a_1 a_2)

7-BIT BINARY CODES: (b_7 b_6 b_5 b_4 b_3 b_2 b_1)

Note:
Character set for Wang systems: 8-bit codes ($b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1$) b_8=0, b_7 through b_1=ASCII.

THREE BITS (HIGH ORDER) →	000	001	010	011	100	101	110	111
FIRST HEX DIGIT (HIGH ORDER) →	0	1	2	3	4	5	6	7
CHARACTERS ↓								
	NUL	DLE	SP	0	@	P	`	p
	SOH	DC1	!	1	A	Q	a	q
	STX	DC2	"	2	B	R	b	r
	ETX	DC3	#	3	C	S	c	s
	EOT	DC4	$	4	D	T	d	t
	ENQ	NAK	%	5	E	U	e	u
	ACK	SYN	&	6	F	V	f	v
	BEL	ETB	' (apos.)	7	G	W	g	w
	BS	CAN	(	8	H	X	h	x
	HT	EM	)	9	I	Y	i	y
	LF	SUB	*	:	J	Z	j	z
	VT	ESC	+	;	K	[	k	{
	FF	FS	, (comma)	<	L	\	l	|
	CR	GS	- (dash)	=	M	]	m	}
	SO	RS	. (period)	>	N	↑	n	~
	SI	US	/	?	O	← (underline)	o	DEL

SECOND HEX DIGIT (LOW ORDER)	FOUR BITS (LOW ORDER)
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

b_7 b_6 b_5 7-BIT BINARY NOTATION

HEX a_1 a_2 b_4 b_3 b_2 b_1

LEGEND FOR ASCII CONTROL CHARACTERS

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible or attention signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
		SUB	Substitute
LF	Line Feed	ESC	Escape
VT	Vertical Tabulation	FS	File Separator
FF	Form Feed	GS	Group Separator
CR	Carriage Return	RS	Record Separator
SO	Shift Out	US	Unit Separator
SI	Shift In	DEL	Delete

To help us to provide you with the best manuals possible, please make your comments and suggestions concerning this publication on the form below. Then detach, fold, tape closed and mail to us. All comments and suggestions become the property of Wang Laboratories, Inc. For a reply, be sure to include your name and address. Your cooperation is appreciated.

700-3514F

TITLE OF MANUAL: **GENERAL I/O INSTRUCTION SET REFERENCE MANUAL**

COMMENTS:

Fold

Fold

(Please tape. Postal regulations prohibit the use of staples.)

WANG

Fold

FIRST CLASS
PERMIT NO. 16
Tewksbury, Mass.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

– POSTAGE WILL BE PAID BY –

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE
LOWELL, MASSACHUSETTS 01851

Attention: Technical Writing Department

Fold

Cut along dotted line.

Printed in U.S.A.
13-1019

United States

Alabama
Birmingham
Mobile

Alaska
Anchorage

Arizona
Phoenix
Tucson

California
Culver City
Fountain Valley
Fresno
Inglewood
Sacramento
San Diego
San Francisco
Santa Clara
Ventura

Colorado
Englewood

Connecticut
New Haven
Stamford
Wethersfield

District of Columbia
Washington

Florida
Miami
Hialeah
Jacksonville
Orlando
Tampa

Georgia
Atlanta
Savannah

Hawaii
Honolulu

Idaho
Idaho Falls

Illinois
Chicago
Morton
Park Ridge
Rock Island
Rosemont

Indiana
Indianapolis
South Bend

Kansas
Overland Park
Wichita

Kentucky
Louisville

Louisiana
Baton Rouge
Metairie

Maryland
Rockville
Towson

Massachusetts
Billerica
Boston
Burlington
Chelmsford
Lawrence
Littleton
Lowell
Tewksbury
Worcester

Michigan
Kentwood
Okemos
Southfield

Minnesota
Eden Prairie

Missouri
Creve Coeur

Nebraska
Omaha

Nevada
Las Vegas
Reno

New Hampshire
Manchester

New Jersey
Toms River
Mountainside
Clifton

New Mexico
Albuquerque

New York
Albany
Buffalo
Fairport
Lake Success
New York City
Syracuse

North Carolina
Charlotte
Greensboro
Raleigh

Ohio
Cincinnati
Cleveland
Middleburg Heights
Toledo
Worthington

Oklahoma
Oklahoma City
Tulsa

Oregon
Eugene
Portland

Pennsylvania
Allentown
Camp Hill
Erie
Philadelphia
Pittsburgh
Wayne

Rhode Island
Cranston

South Carolina
Charleston
Columbia

Tennessee
Chattanooga
Knoxville
Memphis
Nashville

Texas
Austin
Dallas
Houston
San Antonio

Utah
Salt Lake City

Vermont
Montpelier

Virginia
Newport News
Norfolk
Richmond

Washington
Richland
Seattle
Spokane
Tacoma

Wisconsin
Brookfield
Madison
Wauwatosa

International Offices

Australia
Wang Computer Pty., Ltd.
Adelaide, S.A.
Brisbane, Qld.
Canberra, A.C.T.
Darwin N.T.
Perth, W.A.
South Melbourne, Vic.3
Sydney, NSW

Austria
Wang Gesellschaft, m.b.H.
Vienna

Belgium
Wang Europe, S.A.
Brussels
Erpe-Mere

Canada
Wang Laboratories (Canada) Ltd.
Burnaby, B.C.
Calgary, Alberta
Don Mills, Ontario
Edmonton, Alberta
Hamilton, Ontario
Montreal, Quebec
Ottawa, Ontario
Winnipeg, Manitoba

China
Wang Industrial Co., Ltd.
Taipei
Wang Laboratories Ltd.
Taipei

France
Wang France S.A.R.L.
Paris
Bordeaux
Lyon
Marseilles
Nantes
Strasbourg
Toulouse

Great Britain
Wang (U.K.) Ltd.
Richmond
Birmingham
London
Manchester
Northwood Hills

Hong Kong
Wang Pacific Ltd.
Hong Kong

Japan
Wang Computer Ltd.
Tokyo

Netherlands
Wang Nederland B.V.
IJsselstein
Gronigen

New Zealand
Wang Computer Ltd.
Auckland
Wellington

Panama
Wang de Panama (CPEC) S.A.
Panama City

Singapore
Wang Computer (Pte) Ltd.
Singapore

Sweden
Wang Skandinaviska AB
Stockholm
Gothenburg
Malmo

Switzerland
Wang A.G.
Zurich
Basel
Geneva

Wang Trading A.G.
Zug

United States
Wang International Trade, Inc.
Lowell, Mass.

West Germany
Wang Laboratories, GmbH
Frankfurt
Berlin
Cologne
Dusseldorf
Essen
Freiburg
Hamburg
Hannover
Kassel
Munich
Nurnberg
Saarbrucken
Stuttgart

International Representatives

Abu-Dhabi
Argentina
Bahrain
Bolivia
Brazil
Canary Islands
Chile
Colombia
Costa Rica
Cyprus
Denmark
Dominican Republic
Ecuador
Egypt
El Salvador
Finland
Ghana
Greece
Guatemala
Haiti
Honduras
Iceland
India
Indonesia
Ireland
Israel
Italy
Jamaica
Japan
Jordan
Kenya
Korea
Kuwait
Lebanon
Liberia
Malaysia
Malta
Mexico
Morocco
Nicaragua
Nigeria
Norway
Paraguay
Peru
Phillippines
Portugal
Saudi Arabia
Scotland
Spain
Sri Lanka
Sudan
Syria
Thailand
Turkey
United Arab Emirates
Venezuela

WANG LABORATORIES, INC.
ONE INDUSTRIAL AVENUE, LOWELL, MASSACHUSETTS 01851, TEL. (617) 459-5000, TWX 710 343-6769, TELEX 94-7421

Printed in U.S.A.
700-3514F
6-80-3M
Price: see current list