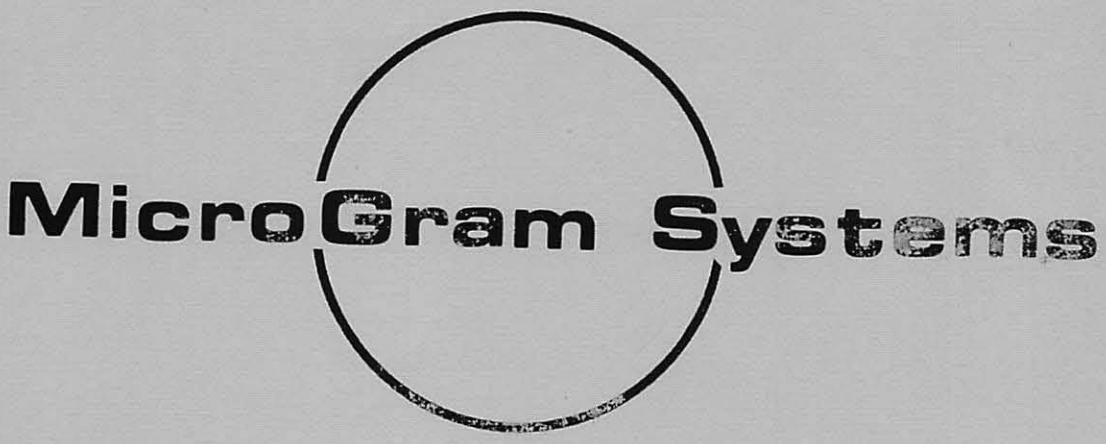
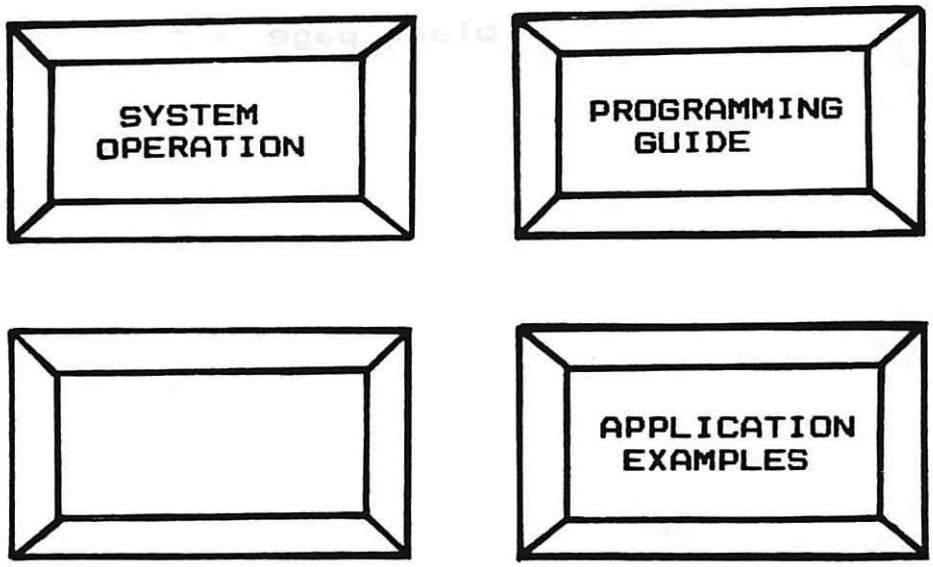
BETA/65



BETA/65

PROGRAMMING MANUAL



Published by MicroGram Systems, La Honda, California 94020.

blank page

PREFACE

This manual shows how to use the BETA/65 Programming System to create applications programs for your OSI Challenger computer. BETA/65 features integer precision selectable from one to fourteen bytes (33 decimal places). All routines are coded to allow mathematical operations on mixed-precision data. String functions are fully implemented, supporting development of text-oriented applications.

BETA/65 is based on a notation similar to APL, but uses BASIC-equivalent names, thereby allowing presentation on ASCII terminals. This brings the expressive power and notational simplicity of APL to the wide variety of computing machines employing the 6502 microprocessor.

Conditional branching and FOR . . NEXT loop structure are supported. Subroutines may be defined in either high-level or low-level code. Two types of parameters may be passed to high-level subroutines. Low-level code is accessed using the LINK function which can address assembly-coded routines by name or address.

BETA/65 supports graphics applications with access to a high-speed window generator. Video screen regions may be dedicated by the user to specialized displays. Also several printer output formats are available including double-spaced listing as well as compressed output.

Interactive features include an immediate mode, allowing computation from the editor, and a concurrent mode. During program run time the concurrent mode can execute any immediate entry from the user without significant interruption of the running program.

blank page

Table of Contents

Part One - SYSTEM OPERATION

Part Two - PROGRAMMING GUIDE

Part Three - APPLICATION EXAMPLES

blank page

Part One

System Operation

Part One - SYSTEM OPERATION

INTRODUCTION

In Part One, you are shown how to use the editor to enter, modify, and store high-level programs written in BETA notation. Part Two shows how to compose simple programs. Several examples are given in Part Three, showing system applications.

Programming is much like riding a bicycle in that reading about it does not necessarily qualify one to do it. We nevertheless recommend that you read this manual thoroughly. Try out the examples on you own machine and, above all, be creative. See Appendix A for system installation procedures.

SYSTEM MODES

Table 1-1 shows the available modes of system operation. Mode changes are made according to the user inputs diagrammed in Figure 1-1. For example, to enter the disk operating system (DOS) from the edit mode use

• EXIT (RETURN)

To re-enter the edit mode from the DOS use

* GO 0000 (RETURN)

It is assumed that the system has been properly loaded. Please consult Appendix A for machine operation procedures to load the BETA system into your computer memory.

In the above two examples, the notation (RETURN) means that the user should press the return key after typing in the preceding text. The first character in the text is supplied by the system to identify the operating mode. This character is called the prompt. The prompt key is given in Table 1-1.

In this manual, all examples will show the prompt as the first character to identify the mode used. Program listings will show the line number as the first set of characters in a line.

MODE	PROMPT	FUNCTION
EDIT (COMMAND)	•	Execute edit command (first character must be alphabetic).
EDIT (PROGRAM LINE)	•	Insert line statement into program (first character must be numeric).
IMMEDIATE	\	Execute one-line user input from edit mode.
STOP	\	Execute one-line user input at program breakpoint.
CONCURRENT	@	Execute one-line user input during program execution.
INPUT	?	Accept user data at program input point.
DOS	*	Execute disk operating system (DOS) command.

Table 1-1 System Mode Definition

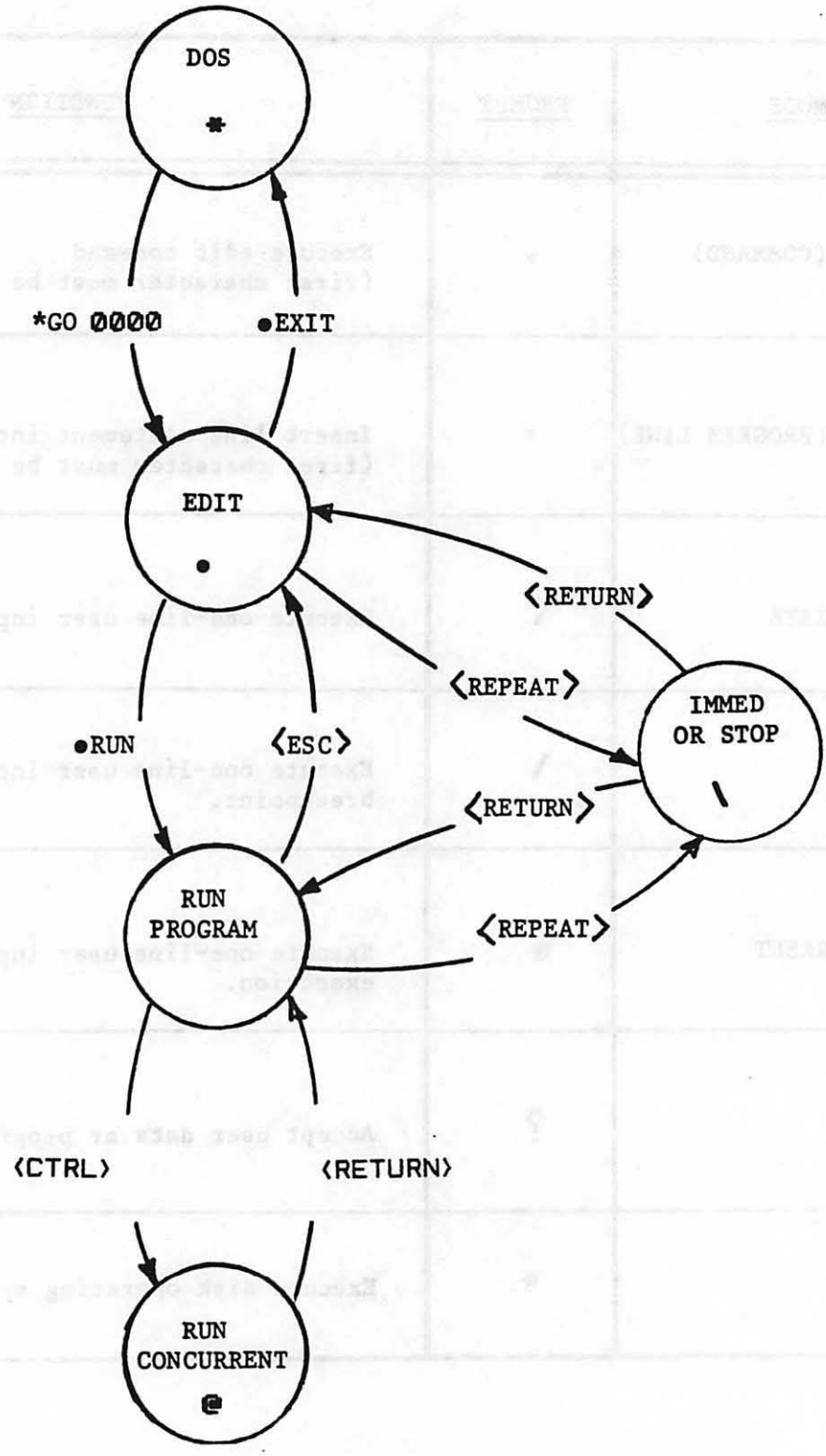


Figure 1-1 Mode Change Diagram

USE of EDITOR

Table 1-2 summarizes the commands available from the edit mode. Your program is also entered, line-by-line, from the edit mode.

Editor Operation

Editor commands allow you to modify, run, or list the transient program. Access to the resident disk operating system (DOS) is also possible using the editor so that your programs can be saved and retrieved for future use.

To initialize program working space the command NEW is entered from the keyboard and the RETURN key is pressed. This command effectively destroys the current program, freeing memory for the next transient program. You are cautioned to save the current program (using the DOS) prior to using the NEW command if future use is anticipated.

The WIDTH command is used to adjust the terminal display width to either 32 or 64 columns. The width may be alternated by repeating the command. As with most commands, WIDTH may be shortened to the first letter.

Listings may be double-spaced with the DOUBLE command. Repetition of this command resumes single-spaced output. The PACK command suppresses printer line feeds and carriage returns until the right margin is entered. This compacts the output stream to reduce paper consumption.

The MERGE command directs the keyboard entry stream to the printer. This gives a record of user activity mingled with the normal output.

COMMAND	FUNCTION
PACK	Printer output compression.
DOUBLE	Use double space for listings.
EXIT	Transfer control to disk operating system (DOS).
MERGE	Print input to output screen.
PONOFF	Turn printer on/off.
WIDTH	Use 32/64 column format for video display.
CHANGE old new	Replace old name by new name.
DELETE first last	Delete lines from first number to last.
MOVE address	Relocate program to new address.
NEW	Clear memory for new program.
CONT	Continue interrupted program execution.
FLIST	List functions available for program use.
FREE	Show nr. of bytes program space remaining.
LIST	List program from first line.
LIST line	List program from specified line.
RUN	Execute program from first line.
RUN line	Execute program from specified line.
VLIST	List variables defined for program use.
SLIST	List subroutines defined for program use.

- DELETE, MOVE, DOUBLE, CHANGE not available on some versions.

Table 1-2 Editor Commands

Keyboard input during editor operation is displayed to the user from the lower left portion of the display screen. This region is called the input or entry screen. Program listing and output are displayed from the upper region of the screen, called the output screen.

The user should verify any command before entering by checking the input screen prior to command entry. Use the RUBOUT key to erase any mistakes observed on the input screen. The command is entered by pressing the RETURN key. This action is denoted by (RETURN).

DOS Command Access

DOS commands may be accessed from the edit mode by the EXIT command. Available DOS commands are given in the OS65D Users Manual. For example, to save the resident program, use

```
* PUT 20 (RETURN)
```

To retrieve this program use

```
* LOAD 20 (RETURN)
```

These commands access disk track 20. Any available track may also be used. After loading a program from the disk use

```
* GO 0000 (RETURN)
```

to enter the edit mode with the saved program fully loaded. See Appendix A for further details.

Reserved Names

Reserved names and primitive functions may be displayed on the output screen by the command FLIST. These names should not be used as variable or label names within the program. Also, new subroutines, defined by the user, must use different names.

New names may use upper or lower case alphabetic characters and may be up to 15 characters long. Names may include numeric characters but the first character in a name must be alphabetic.

To avoid confusion with reserved names, which are all upper

case, you may wish to use only lower case characters for variables.

User-defined Names

The names which are defined as variables, labels, or arrays within the program may be listed by the command VLIST. The names which are defined as subroutines may be listed using the command SLIST.

Names are introduced to the system by usage. Any previously unused name appearing in a program line will be entered automatically into the symbol table as a scalar name. (A scalar is a variable which can be represented by a single number). In Part Two, the appropriate method for declaration of labels, arrays, and subroutines will be presented.

The CHANGE commands can be used to change names of either variables or subroutines. For example, SCHANGE JOB TASK will cause the subroutine name JOB to be replaced by TASK throughout the program.

During program development, all names used are stored by the system in the symbol table. Names which are mistakenly introduced or which fall into disuse take up name slots which may be needed for large programs. The CHANGE commands can be used for garbage collection to replace unused names. Names already in the symbol table should not be used as replacement names.

PROGRAM EDITING

Line Entry

Your program is entered from the edit mode. A BETA program is constructed from one or more lines of user input. For example, a two-line program might be listed as shown below.

```
10 123 + 456 @ length
20 PRINT length,"cm"
```

This small program, when executed, will store the value 579 at the memory location assigned to the variable named length. It will then display this value with centimeter units appended, and return to the edit mode.

Figure 1-2 shows how to enter this simple program from the editor. You may duplicate this program by typing in the text following the edit mode prompt. The RETURN key must be pressed following text completion of each entry.

Given the edit prompt on the input screen, an original line may be placed into the program by typing the line number followed by the line text, followed by (RETURN). You should allow for future line insertion by choosing line numbers which are separated by at least ten digits.

Main program line numbers should range from 0 to 32,767. The range from 32,768 to 65,535 should be used for 'background' program segments, e.g., data fields, print formats, subroutines, etc.

Line Change

Any program line may be changed using the edit mode by replacing the entire line with a new line. To replace line 20 of the example program with a new line, you should enter

```
.20 PRINT  length*10 ,"mm" (RETURN)
```

resulting in a new two line program. (Also see Figure 1-2.)

```
10  123 + 456 @ length
20 PRINT  length*10 ,"mm"
```

The modified program will compute the value of length as before. The

```
.
.
.
.NEW
.
.
.
.10 123 + 456 @ length
.20 PRINT length , "cm"
.
.LIST

10 123+456 @ length
20 PRINT length,"cm"

.RUN

579 cm

.20 PRINT length*10 , "mm"
.
.LIST

10 123+456 @ length
20 PRINT length*10,"mm"

.RUN

5790 mm

.
.
.
.
```

Figure 1-2 Program Entry

print statement multiplies by ten to display the value in millimeters.

Mistakes made during line entry may be erased by using the rubout key to backspace over the incorrect text. The correct text is then entered over the erased line segment.

To delete a program line, enter the line number, followed by (RETURN) with no line text.

Program Listing

The list command, LIST, will cause the first line of the program to be displayed. Subsequent lines may be displayed by pressing the LINE FEED key. Use the RETURN key to halt listing and return to the edit mode.

Selected lines may be listed by typing the desired line number following the list command. For example, LIST 123, will cause line number 123 to be the first line listed. The list command can be abbreviated using the letter 'L'. In addition, the space between the LIST command and the line number is optional.

Program Execution

The run command, RUN, will cause the program to be executed starting at the first line. The program will continue running until the last line is executed. At this time, the edit mode will be re-entered automatically. The user may interrupt execution at any time by pressing the escape key (ESC).

By using a line argument following the run command (e.g. RUN 20) the user can start execution at any selected line of the program. Compressed forms of the run command are allowed, i.e. R20 will cause program execution starting at line 20 (or first line following if line 20 is missing).

Variable Initialization

All program variables are set equal to zero whenever the program is executed using the RUN command. The CONT command continues an interrupted program with no change in variable values.

INTERACTIVE PROGRAMMING

Immediate Mode

All program functions are available from the immediate mode. This mode is accessed from the editor by pressing the REPEAT key. Following the breakpoint prompt, enter any valid statement. The statement syntax must correspond to that used for normal program text (line numbers must not be used).

For example, to convert hexadecimal numbers to decimal from the immediate mode* use

```
\ PRINT $89AB,$1234
```

To convert decimal numbers to hexadecimal values use

```
\ HEXPR 35243,4660
```

Notice that the print statement uses the comma to separate data items. Complex expressions may be used in the print list by separating the expression with commas. For example,

```
\ PRINT 1234,"plus",5678,"equals",1234 + 5678
```

is a valid print statement.

Strings included in the print list must be enclosed by quotation marks, as shown above. In general, any combination of simple expressions (containing only a name or number), complex expressions, or strings (surrounded by quotes) may be used in a print list.

To return to the editor--press RETURN with an empty line.

---

*Recall that \ denotes the immediate mode. Press the REPEAT key to enter this mode.

STOP Mode

The program may be interrupted during execution by pressing the REPEAT key. An interrupted program will display the breakpoint line number and enter the STOP mode.

Figure 1-3 shows entry, listing and execution of a small program illustrating the STOP mode. During execution of this program, press the REPEAT key and type in 'one-liners' for immediate execution.

User input during the STOP mode may be used to interrogate and/or modify program variables. Any valid statement entered from the keyboard will be executed automatically following RETURN. The system will return to the STOP mode after executing the command.

To return to program execution from the STOP mode, press RETURN (with no other input).

The run-time STOP is useful for debugging programs since program variables can be directly accessed. Any defined variable or subroutine can be used. For example, the breakpoint statement

```
\ PRINT "A,B,C=",A,B,C
```

will cause the variables A,B, and C to be displayed on the output screen, preceded by the message enclosed in quotes. Variables may also be modified at a run-time break point.

The user may not define any new variables or subroutines from the STOP mode. Only program entry lines can contain new names.

The user may cause program interruption by inserting the instruction STOP at a selected point in the program. This has the same consequences as user interrupt using the REPEAT key but allows absolute specification of the breakpoint within the program.

```
.
.
.NEW
.
.
.
.10 PRINT  "Interactive test program."
.20 PRINT (I+1@I)  DELAY 1000  GOTO 20

.LIST

10 PRINT "Interactive test program."
20 PRINT(I+1@I) DELAY 1000 GOTO 20

.RUN

Interactive test program.
1
2
3                                    { Press
                                     { REPEAT
AT LINE NR 20

\ PRINT I
3

\ 12 @ I


\ PRINT I
12
                                     { Press
                                     { RETURN
13
14
15
16                                   { Press
                                     { ESCAPE
AT LINE NR 20
.
.
```

Figure 1-3 Program Illustrating STOP Mode

Concurrent Mode

For some applications, the processor must be dedicated to controlling a real-time process over an extended interval. It is often desirable to use the immediate mode during such operation to check values, modify gains, etc. This is done by a time-sharing technique which tests for user keyboard activity at each program line start.

Real-time input is enabled by pressing the CONTROL key to the down position. This allows keyboard input and calculation during program run. The constraints on input are the same as the STOP mode.

INPUT Mode

During program execution, user input may also be provided using INPUT. This instruction causes the program to halt and interrogate for data. Figure 1-4 illustrates use of the INPUT instruction in a simple program.

Following the RUN command and display of the print message the program will display the INPUT prompt (?). Numeric data separated by spaces is keyed in followed by RETURN. The variables, I and J, are assigned the values, 10 and 11, respectively.

If too few values are entered by the user, the system will re-prompt for more data. When enough values are entered, the system continues execution. If too many values are keyed in, the system will ignore values not needed.

```
.
.
.NEW
.
.
.
.10 PROMPT "I,J="
.20 INPUT I,J
.30 PRINT I,J
.
.LIST

10 PROMPT "I,J="
20 INPUT I,J
30 PRINT I,J

.RUN

 I,J=
? 10 11
10 11

.                          { Press
\ PRINT I,J                { REPEAT
10 11

\                          { Press
.                          { RETURN
.
.
```

Figure 1-4 Program Illustrating INPUT Mode

Part Two

Programming Guide

Part Two - PROGRAMMING GUIDE

Instruction	Function
CALL	Call subroutine named in list.
SUBR	Define subroutine named in list.
%	Subroutine reference/value list separator.
:	Define label or data field.
\ (CONTROL 5)	Load argument value into result register.
TEST	Same as \ (use with conditional branch).
DELAY	Stop execution for N msec.
HEXPR	Print value as hexidecimal to output screen.
PRINT	Print value as decimal (or ASCII string) to output screen.
!	Remark follows to end of line.
PROMPT	Same as PRINT but to input screen.
REF	Reference data field pointer.
READ	Save data from data field pointer at variable named in list.
INPUT	Accept data from user and save at variable named in list.
FOR	Start FOR ... NEXT loop.
NEXT	Repeat FOR ... NEXT loop.
FROM	FOR...NEXT loop variable initialization.
TO	FOR...NEXT loop end test.
STEP	FOR...NEXT increment.
GOTO	Transfer to label or line in argument list.
LINK	Transfer to machine-coded routine.

Table 2-1 BETA/65 Keywords.

INTRODUCTION

Part One discussed how program lines are combined to create programs. Program lines, in turn, consist of components, called symbols, numbers, and strings. Part Two presents the rules of combination so that valid lines may be composed from these components.

Tables 2-1, 2-2, 2-3, and 2-4 show the instructions available for program use. These instructions may be classified into three categories: keyword, dyadic, and monadic.

Keywords are directives to the system to perform an indicated function. Table 2-1 shows the keywords which require list components to function. BETA/65 keywords recognize two types of list components--expressions and assignments. An expression is a series of mathematical operations on data which computes to a numerical value. A value assignment indicates where data is to be stored.

Table 2-2 shows the keywords which function without data. These are called niladic functions. For example, the function CLR will cause the video screen to clear whenever encountered in program text.

An expression is composed of mathematical functions which operate on data objects, called primaries. All hand calculators perform expression evaluation using numbers as data objects. Computers extend this capability by using symbolic names, representing numbers, as data objects. These are called variables, since they may be modified in the course of program execution.

Two types of mathematical functions, dyadic and monadic, are used to combine primaries into expressions:

Dyadic functions (see Table 2-3) operate on data contained in a result register and values computed from the primary list. The result of this operation is automatically saved in the result register and is available for use by the next instruction.

Monadic functions (see Table 2-4) operate on the value computed from its primary to produce a result. This result is treated as a primary value for the preceding function in the program text.

Instruction	Function
END	Terminate program execution and return to editor.
ESC	Interrupt program execution and return to editor.
STOP	Break program execution and prompt for user one-line input.
RET	Return from subroutine execution to calling program.
WAIT	Wait for LINEFEED/RETURN.
CLR	Clear video screen
SCR	Scroll video screen.

Table 2-2 Niladic Instructions

Instruction	Function
*	Multiply result by value found in arg. list.
/	Divide result by value found in argument list.
+	Add result to value found in argument list.
-	Subtract from result the value found in argument list.
∧ (CTRL 0)	Take result to power found in argument list.
MOD	Divide result by value and save remainder as new result.
QMULT	Multiply result by power of two found in arg. list.
QDIV	Divide result by power of two found in arg. list.
MIN	Load argument value if less than result.
MAX	Load argument value if greater than result.
AND	Logical AND result with argument value.
OR	Logical OR result with argument value.
XOR	Exclusive OR result with argument value.
POKE	Store low byte from result at address(es) in arg. list.
@ (SHIFT 0)	Store result at variable named in assignment.
IFPL	Branch if result zero or greater.
IFMI	Branch if result less than zero.
IFEQ	Branch if result equal to zero.
IFNE	Branch if result not equal to zero.

Table 2-3 Dyadic Functions

Instruction	Function
(	Open parenthetic expression.
,	Not used.
)	Close parenthetic expression.
PEEK	Load one-byte data from address found in argument list.
NEG	Negate argument value.
-	Negate argument value.
CHR$	Change argument value to one-byte string.
ABS	Absolute value of argument.
SGN	±1 with same sign as argument.
SQR	Square root of argument.
NOT	Logical complement argument value.
DIM	Reserve space for array named in list.
AT	Align display output cell to argument value.
TAB	Align print head to argument value.

Table 2-4 Monadic Functions

ARITHMETIC EXPRESSIONS

An arithmetic expression is composed of dyadic and/or monadic functions which operate on data items, called primaries. A simple primary is either a number or symbolic name representing a number. Any expression may be used to create a primary by parenthetic enclosure.

Arithmetic expressions are normally preceded by a keyword. Mutiple expressions are separated by commas and the last expression in a list is ended by either a keyword or end-of-line.

Dyadic Functions

Dyadic functions operate on two numbers to produce a result. Multiply, divide, add, and subtract are familiar dyadic functions. The segment below shows an example using three dyadic functions, + for addition, * for multiplication, and @ for variable storage.

```
\ 123 + 456 * beta @ alpha
```

The BETA interpreter uses a repetitive cycle to evaluate this expression, viz,

(1) read next operator
(2) read next value
(3) execute operator function using old result and next value to produce new result.
(4) save new result and repeat from (1)

This cycle may be represented symbolically as

$$result_k \ DYAD_k \ value_k \longrightarrow result_{k+1}$$

This cycle is executed by the system on code text. The part of the system which does this job is called the interpreter.

It is not necessary for the user to be concerned with details of this cycle since they are already handled by the system. However, it

may be useful to keep the above cycle in mind along with its relation to the actual expression text when composing a program to evaluate arithmetic expressions. This cycle does not include details of the actual interpreter which must handle repeat arguments and monadic functions. These are discussed later.

Evaluation Sequence

Arithmetic expressions are evaluated from left-to-right. With the exception of parentheses and monadic operators, to be discussed later, BETA does not assign evaluation priority to arithmetic operators. It is not necessary to consult an operator priority list to understand the order of operator execution.

The notation consistently follows the left-to-right rule including post-assignment for result values. This means that assignment follows rather than precedes the arithmetic expression. For example, to equate the sum of two variables to a third variable, write A + B @ C rather than C @ A + B.

Use of Parentheses

With parentheses, arithmetic expressions written in BETA are coded the same as those written in BASIC or FORTRAN. Hence parentheses provide a bridge of commonality between these languages. Consider the polynomial expression

$$3X^2 - 4X + 5.$$

By factoring X from the first two terms this can be rewritten as

$$(3* X - 4) * X + 5.$$

This expression may be evaluated and the result stored at variable P using the following program.

```
10 2 @ X
20 (3 * X - 4) * X + 5 @ P
```

In line ten, the variable X is initialized by loading the value 2 (or any other value desired by the user) into the result register and transferring the result into the memory location for X. Note

that the load operation occurs by default and is not listed when it is the first operation in a line.

In line twenty the complex expression, (3 * X - 4), is first evaluated using the first-level result register and is then loaded into the zero-level result register. Following multiplication by X and addition of 5 the result value of 9 is stored at location P.

Use of parentheses is recommended, especially if the programmer is not familiar with the notation used for this system. With experience, the user will observe that omission of parentheses is possible in some cases. For example, consider the following program.

```
10 2 @ X
20 3 * X - 4 * X + 5 @ P
```

Line ten is executed as before. Line twenty can be broken into six interpreter cycles:

(1) load immediate value, 3, into result register
(2) multiply result by X
(3) subtract 4 from result
(4) multiply result by X
(5) add 5 to result
(6) assign result value of 9 to variable P.

This approach saves two bytes of memory space compared with that using parentheses. Also a slight savings in runtime is gained by avoiding result transfer from parentheses level one to level zero.

Simple Examples

The following examples are designed to illustrate expression evaluation using the BETA system. This notation is most concise for 'conversion' type problems (see first and second examples) or polynomial evaluation (see third example). Other types, e.g., vector dot product, require use of parentheses (see fourth example). The fifth example, from feedback control theory, shows how to reformulate an expression so that parentheses are not needed.

Length Conversion

Measurements of a length interval are given in meters, centimeters, and millimeters. It is required to compute the total length in millimeters. This is most efficiently done by first converting the meters component into centimeters by multiplying by 100 centimeters/meter. The centimeters component is added to this result which is then multiplied by 10 millimeters/centimeter. Finally, the millimeters component is added to this result to yield the final result.

This conversion is expressed in BETA notation as follows:

```
Nm * 100 + Ncm * 10 + Nmm @ Lmm
```

where Nm, Ncm, and Nmm are the length components in meters, centimeters, and millimeters, respectively. Lmm is the total length, expressed in millimeters.

Time Conversion

Elapsed time is given in hours, minutes, and seconds. The total number of seconds in this interval is readily computed using

```
Nhr * 60 + Nmin * 60 + Nsec @ Tsec
```

where Nhr, Nmin, and Nsec represent a breakdown of elapsed time in hours, minutes and seconds, respectively. The hours component is first multiplied by 60 (the number of minutes per hour) to convert into minutes. The minutes component is then added to the result and the new result is multiplied by 60 (the number of seconds per minute) to convert into seconds. The seconds component is then added to the result which is stored at Tsec.

Base Conversion

Base conversion is a frequent problem for computer users. The decimal value of the octal number, 7356 (base 8), may be displayed using PRINT (7 * 8 + 3 * 8 + 5 * 8 + 6). When data is given in binary notation use PRINT (1 * 2 + 0 * 2 + 1 * 2 + 1) to display the value, 1011 (base 2) = 11 in decimal notation. This procedure can be extended to any length number written in any base.

In general, a number may be expressed in an arbitrary base as a polynomial in that base, e.g.

$$N = (A_3 * b^3) + (A_2 * b^2) + (A_1 * b^1) + (A_0 * b^0)$$

Given the coefficients, A_k, and base, b, compute N as

```
A3 * b + A2 * b + A1 * b + A0 @ N
```

The inverse problem, given N and b compute A_k, is solved iteratively using the MOD function. See Part Three for examples.

Sum of Squares

Sum of squares is a computation required for statistics, vector arithmetic, noise measurement, etc. For example,

$$X^2 + Y^2 + Z^2$$

is the diagonal length (squared) of a box with sides X, Y, and Z.

If an arithmetic expression consists of the sum of several products, each product pair should be enclosed by parentheses. The following expression computes the sum of three squares.

```
(X * X) + (Y * Y) + (Z * Z) @ D2
```

Use of parentheses to enclose the <u>first</u> product pair is optional.

Feedback Control

A frequent expression occurring in feedback control problems is the summation of products, each consisting of state variable measurements multiplied by gain values. Such expressions are called linear combinations. For position control using rate feedback the control effort may be computed as

```
(Xdot * Krate) + (X * Kpos) @ feedback
```

where X and Xdot are measured position and rate values while Kpos and Krate are feedback gain values. See Figure 2-1.

In many practical cases the gain values are constant and this expression may be computed as

```
Xdot * tau + X * Kpos @ feedback
```

where tau is the ratio, Krate/Kpos, which is pre-computed. See Figure 2-2.

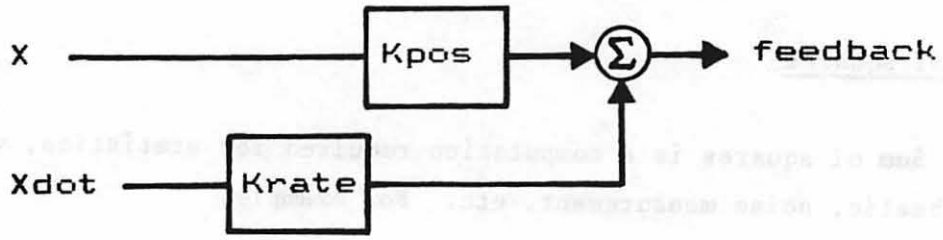


Figure 2-1 Position Control with Rate Feedback

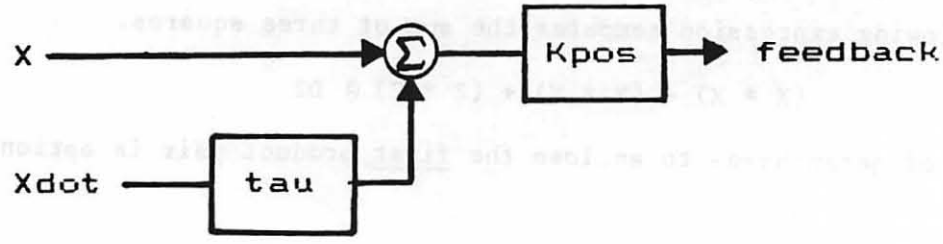


Figure 2-2 Modification to Avoid Parentheses

Writing Valid Primaries

BETA permits five primary types to appear following a function:

(1) positive or negative numbers

(2) names of program variables

(3) array elements

(4) parenthetic expressions

(5) monadic functions.

Any number of primaries may follow the function. These may appear in any combination of the above types. When several primaries appear in a list following a dyadic function the system will repeat the function consecutively for each primary.

Negative numbers which appear in a primary list must be enclosed by parentheses. A negative number may be used in the first primary without parentheses, however. The monadic function, NEG, should be used for negation instead of the negative sign, especially wherever the latter usage might be confused with subtraction.

Primaries in a list are separated using blanks. Commas are used for separation in expression lists.

The parenthetic expression, (3 * X - 4), is an example of a valid primary. It is composed of three simple primaries: 3, X, and 4. It is legal to substitute this primary wherever a simple primary might be used in a primary list. Also, any of the three simple expressions may be replaced by any valid primary. In this way, primaries of arbitrary complexity can be developed.

List properties of keywords and dyadic/monadic functions are summarized in Table 2-5.

List Properties of BETA/65 Operators

Operator Group	List Properties	Partition
PRINT HEXPR PROMPT DELAY DISK %	Multiple Expressions	Comma
TEST \ = (..) TO STEP	Single Expression	----
CLR SCR WAIT ESC STOP NEXT RET END	No List	----
READ INPUT INLINE KBD	Multiple Assignments	Comma or Space
@ (Shift 0)	Multiple Assignments	Space
LET REF FOR	Single Assignment	----
* / + - ^ MOD QMULT QDIV MIN MAX AND OR XOR VS CON$ POKE DPOKE	Multiple Primaries	Space
NEG ABS SGN SGZ SQ SQR NOT AT TAB CHR$ STR$ ASC VAL LEN PEEK DPEEK	Single Primary	----
IFPL IFMI IFEQ IFNE GOTO	Single Branch Target	----
LINK	Multiple Link Targets	Comma
:	Data Field	Comma or Space

Monadic Functions

Monadic functions, such as square root, are functions of a single variable. An arithmetic statement involving the monadic function ABS might be as shown below:

```
\ ABS gamma @ alpha
```

This statement is designed to load the absolute value of the variable, gamma, and store the result at the location of the variable, alpha.

Referring to the interpretive cycle discussed previously, observe that the load operator (\) cannot function until a value for ABS gamma is computed. This computation is done using the first level result register while the load operation is pendent. The first level result is then used as primary value for the pendent load operation which causes the transfer from first to zero level. This result is then transferred to the memory location for alpha.

A primary for any monadic function must follow the same rules as primaries for dyadic functions. This means that primary types are the same for monads and dyads. However, monads will use only the first primary. Primaries following the first are interpreted as components of the pendent dyadic function primary list.

Note that simple expressions (i.e., numbers or names of variables) may be used as primaries without parenthetic enclosure.

Negate is a monadic function, denoted by the minus character, (-). For example,

```
\ beta * - alpha = gamma
```

computes the negative value of alpha before multiplying by beta. Since negate must immediately follow either a dyadic or monadic function, there is no possibility of confusion with subtraction, also denoted by the minus character, which must always be preceded by an primary.

Monadic functions may be primaries of monadic functions. For example,

```
\ SQR ABS alpha = gamma
```

is a valid expression.

ASSIGNMENT

Variables may be assigned values by use of the assignment operator (@). This instruction copies the current result into the variable location(s) named in the assignment list which follows. Multiple assignment of the same result can be made using several components. For example,

```
\0 @ U  \100 @ V @ W
\0 @ X Y Z
```

In the first line, the value zero is assigned to variable U. The variables V and W are then assigned the value one hundred. In the second line, the short-form notation is used to zero X, Y and Z.

The instructions READ and INPUT are also assignment operators. These operators, along with @, will accept only a variable name or array element as list components. That is, only the name of a program variable can be included in the assignment list. Array names with index values may be included in assignment lists. For example, 10 @ A(2), will store the value 10 in array A, element 2.

A run-time error will result if a label name is found in an assignment list. This is necessary to prevent writing data into program code.

Assignment operators must be followed by at least one assignment. The assignment list is terminated by the next arithmetic operator or end of expression.

The BASIC instruction, LET, may also be used for value assignment. For example, the statement

```
LET C = 1 + 2 * 3
```

assigns the value 9 to the storage location of variable C. An array element may be used as assignment. With LET, only one assignment component is allowed.

DATA FIELDS

Data fields are program lines containing one or more constants and/or strings. Data fields are useful for passing initial values to run-time variables. They are indispensable if large arrays must be initialized. Data fields may also contain print formats and be used as dispatch tables for user-defined routines. The colon operator (:) is used to denote a data field. A lable name may also be used. For example,

```
100 :data  135
```

The combination, :data , is called a label. The label name (in this case data) may be referenced using the REF function.

Initializing Data

The REF instruction may be used to reference a labeled data field. The READ instruction will sequentially copy data contained in the data field which follows the referenced label.

```
10 :data  123  456  789
20 REF data READ  A B C
```

In line 20, the data pointer is aligned to line 10 text followed by transfer of data values to variable locations in the READ assignment list.

It is acceptable to intermingle labels in the data field and continue the data field over several lines. For example,

```
10 :row1  123  -123
20 :row2  -456  456
30 REF row1  READ A B C D
40 REF row2  READ E F
```

This allows the variables E and F to share the last two values of the data field with variables C and D. Note that negative numbers may be included in data fields.

Initializing Arrays

In the following example the eight element array, A(8), is initialized with each element equal to the constants in the order shown on lines ten and twenty.

```
10 :values  5 3 8 12
20 :2 4 6 8
30 REF values FOR I FROM 1 TO 8 READ A(I) NEXT I
```

In line 20 note that the label name is omitted. This is acceptable as long as a preceding line contains a label name for alignment reference. The array must be dimensioned to eight elements or more using DIM A(8).

Print Formats

A data field may be used to format print displays by referencing with the TAB instruction. This is useful for printing data in columned tables.

```
PRINT TAB col, A, B, C, D, E

:col  16 24 32 40 48
```

Several examples of formatted print statements are shown in Table 2-6.

Application - Print Graphics

The TAB instruction may use a non-label argument to specify which column must be used for printing. This feature can be used for plotting data profiles. For example, the statement

```
PRINT TAB(A), "A"
```

will reference the print head to the value of A and print the character A at this column.

When several variables are to be plotted, the following form should be used:

```
PRINT TAB(A),"A",TAB(B),"B",TAB(127)
```

The TAB instruction automatically folds the argument (modulo 128) to avoid off-scale print commands.

Formatted PRINT

```
:tabset  12 18 24
 ....
PRINT TAB tabset, alpha, beta, gamma
```

Graphics PRINT (single variable)

```
PRINT CHR$ 29  ! Set 128 char/line.
 ....
PRINT TAB(alpha), "A"
```

Graphics PRINT (several variables)

```
PRINT TAB(A),"A",TAB(B),"B",TAB(127)
```

Hexadecimal print

```
HEXPR 0, 15, 16, 255, 256
```

PRINT to Input Screen

```
PROMPT "Enter positive number"
```

Table 2-6 Formatted PRINT Examples

ARRAYS

One Dimensional Arrays

One dimensional arrays can be used in arithmetic expressions if properly dimensioned. For example, the segment

```
10 A(1) + A(2) @ A(3)
....
30 DIM A(3)
```

will store the sum of the first two elements of the array, named A, into the third element.

It may be convenient to locate dimension statements at lines above 32,768 since this region is bypassed during main program execution.

Multidimensional Arrays (advanced programming)

Although arrays can have only one dimension, it is possible to reference elements in an N dimensional fashion. Consider, for example, the case of two dimensions where it is desired to represent a matrix having 2 rows and 3 columns. Reference to the element at row I = 0, 1 and column J = 0, 1,2 is done using the notation A(I*3+J). The array should be dimensioned, in this case, for six elements (from zero to five) using DIM A(5). The enclosed number is one less than the product of rows with columns.

The array elements corresponding to row I and column J are as shown below.

		Col J		
		0	1	2
Row I	0	A(0)	A(1)	A(2)
	1	A(3)	A(4)	A(5)

Additional dimensions can be referenced by extending this convention.

PROGRAM CONTROL

Control functions are used to modify the execution sequence of a program. They are used to either repeat or skip segments of the program. Control action may be based on value of the result register or value of a designated loop variable.

Loops

Segments of a program may be repeated several times using either FOR...NEXT or implicit looping. The former is generally more convenient. For example,

```
10  FOR I = 1 TO 7 STEP 2
20  PRINT I
30  NEXT I
```

This program will print values 1,3,5,7. The STEP instruction is optional, defaulting to either +1 or -1 depending on the sign needed to step the loop variable (in this case I) from initial to final value.

FOR must be followed by a scalar assignment. The keywords =, TO, and STEP are each followed by an expression indicating initial, final and step values, respectively. Use of an assignment variable following NEXT is optional. TO and STEP may be replaced by commas.

Explicit Loop (FOR...NEXT)

```
FOR K = first_value TO last_value
 ....
NEXT K
```

Nested Explicit Loops

```
FOR I = 0 TO 10 STEP 2
FOR J = 0 TO 12 STEP 3
 ....
NEXT J
NEXT I
```

Shortform Explicit Loops

```
FOR I = 0, 10, 2
FOR J = 0, 12, 3
....
NEXT NEXT
```

TABLE 2-7 FOR...NEXT Loops

Nested Loops

Loops may be nested as shown below.

```
10  FOR I = 1 TO 2
20  FOR J = 1 TO 3
30  PRINT I,J
40  NEXT J
50  NEXT I
```

Multiple nesting is limited only by machine stack limits. Table 2-7 illustrates FOR . . NEXT usage.

Branch Instructions

Branches are used to modify the sequence of program execution. The GOTO instruction causes an unconditional branch while the conditional branches. IFEQ, IFNE, IFPL, and IFMI, cause branching based on result value.

Branching may occur to either labels or to line numbers. Labels are places in the program text consisting of a colon followed by a name used for reference. Label branching may be either forward or backward from the branch point. Line branching must be forward. A backward branch to the beginning of the same line may be used.

A branch instruction must be followed by either a name or a number to indicate label or line branching, respectively. Use of non-label names will result in a run-time error. This is necessary to prevent branching to a data location.

The conditional branches act by testing the value of the result register. When the branch condition is met the instruction sequence is restarted at the program line number (or label) given in the branch assignment. When the branch condition is not met, the execution sequence continues with the first instruction after the branch assignment.

Restart after branching will cause the result stack to be reset to the zero level. This means that all branches must be to locations which are outside of parentheses. A branch to a point inside parentheses will very likely cause a parentheses range error at run time. Branching from inside parentheses is acceptable.

Branching into or out of FOR...NEXT loops should not be done. Complex loop exit conditions should be programmed using implicit loops instead of FOR...NEXT loops.

Implicit Loops

The branch instructions may be used to define a variety of implicit loops. Implicit loops are inherently more adaptable to user requirements but are also more prone to programming errors. During program checkout a print statement should be included within each

loop to verify loop sequencing. This can be removed after verification to allow full-speed operation.

A program subject to endless looping can be aborted by pressing the escape (ESC) key. The output screen will display the line number. This information may be useful in isolating the loop error.

Table 2-8 shows some examples using implicit loops.

Implicit Loop (using branch-to-line)

```
10  40 @ index
20  ..... \ index - 1 @ index IFPL 20
```

Implicit Loop (using branch-to-label)

```
10  40 @ index :loop
20 ....
30 index - 1 @ index IFPL loop
```

Initialize Array Using Implicit Loops

```
10   44 @ index :loop
20   0  @ array(index)
30   index - 1 @ index IFPL loop
98   END
100   !
110  DIM array(44)
```

Table 2-8 Implicit Loop Examples

SUBROUTINE CALLS

Subroutines may be written in either machine code or high-level code and called from any point in the program. Access to subroutines written in high-level code is done using the CALL construct. Access to machine code is possible using the LINK instruction.

High-level Subroutines

The following segment illustrates the format used for high-level subroutines.

```
100  SUBR TRANSFER   output % input
110  LET output = input
120  RET
```

In line one hundred the keyword, SUBR, denotes the user-defined subroutine, which is named TRANSFER. The list following the subroutine name is called the reference list. Only dummy names can be used in this list.

The % character is used to separate the reference list from the value list which follows. As with reference lists, only dummy names may appear in subroutine value lists.

The next line contains the subroutine text. In general, the text may be continued over several lines. The subroutine is terminated by the RET statement.

Subroutine Calls

The preceding subroutine may be called as follows:

```
10  CALL TRANSFER alpha % 12345
```

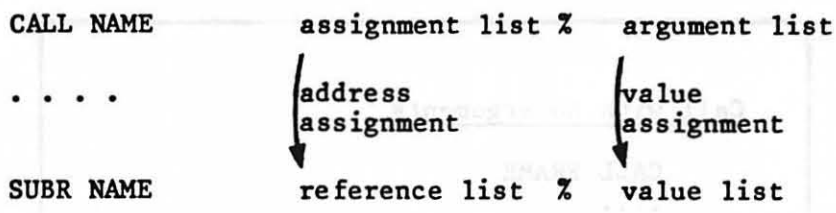
When this call is executed the value 12345 is transferred to the variable alpha. The list to the right of the % separator may include any valid expression. For example,

```
CALL TRANSFER beta % 123+456
```

is a valid call. The list to the left of the separator must contain only assignments.

List Rules

A subroutine call performs the address and value assignments illustrated below:



As stated previously, only dummy names may be used in subroutine lists. Dummy names are names which appear as variable or label names within the subroutine text. Calls to subroutines must contain assignments or expressions in one-to-one correspondence with names in the subroutine list.

The CALL assignment list follows the rules stated earlier for assignment, i.e., variable names and/or array names with index values are valid assignment list components. Label names may also be used, provided usage of the corresponding dummy name within the subroutine text follows established rules for labels.

The CALL expression list follows the previously stated rules for composing expressions. Multiple expressions must be separated by commas.

Several examples of subroutine calls with assignments and/or expressions are shown in Table 2-9.

A subroutine call will reference the variables named in the subroutine reference list to addresses of variables given by the call assignment list. This causes any reference to dummy names in the subroutine text to refer instead to the corresponding name in the call assignment list. The variables named in the subroutine value list are initialized to the values contained in the call expression list. This type of subroutine parameter listing is termed call by reference/value.

Call with no arguments

```
CALL FRAME
....
SUBR FRAME....RET
```

Call with argument

```
CALL SAVE % alpha
....
SUBR SAVE % input....RET
```

Call with assignment

```
CALL LOAD beta
....
SUBR LOAD output....RET
```

Call with assignment and argument

```
CALL TRANSFER beta % alpha
....
SUBR TRANSFER output % input....RET
```

Table 2-9

Subroutine Call Examples

Recursive Calls

High-level subroutines may call other subroutines. Subroutines may also call themselves. It should be remembered that any call to a high-level subroutine will modify any or all dummy variables.

All variables are shared and may be used throughout the program. The system does not support internal variables for subroutines.

Machine-Coded Subroutines

Subroutines written in machine code may be accessed using the LINK instruction. This instruction mimics the assembly code JSR. The LINK argument list contains the name or address of the subroutine which is accessed. This subroutine must be terminated with RTS. Multiple arguments may be included in the link list. For example, the segment

```
10 LINK clear,frame
```

consecutively transfers control to machine coded routines named clear and frame and then continues program execution from the next line.

Machine-coded subroutines may be assigned addresses using the data-field construct.

```
10 :clear HHHH

20 LINK clear
```

The LINK instruction transfers control to address HHHH and continues program execution upon return from this machine-coded routine. This construct allows absolute addressing (i.e., use of routine name) for LINK instructions. The data-field reference can appear anywhere in the program.

MACHINE ACCESS

The PEEK and POKE functions are used to access memory locations external to the system and the user program. External memory locations are typically dedicated to peripheral input and/or output registers. Such I/O ports allow the software to communicate with the external or 'real' world.

I/O ports are dedicated to memory locations within the 64K address range of the 6502 processor. Assignment of I/O ports must not overlap memory used for system or program software. See system memory map for system and program memory limits.

Data Output

Byte data is written to output ports using the POKE instruction. For example, the segment

```
\ 63 POKE 54048
```

loads the value 63 and transfers this number as one byte data to memory location 54048 (hex $D320).

Data Input

The PEEK instruction is used to access memory locations for byte data previously stored. For example,

```
\ PEEK 54048 @ alpha
```

loads the byte data at location 54048 and stores this data into program variable alpha for general use by the program.

Function Re-vector

The VECTOR operator may be used to dynamically re-vector any function. For example, the segment

```
VECTOR PRINT HHHH
```

may be used to redefine the print function. It is assumed that HHHH is a memory address where an alternate print routine has been assembled.

New primitives may be introduced by the VECTOR instruction. Syntactic usage of new primitives must correspond to the rules established for each category. The machine-coded routine must be terminated as shown below:

Function Category	Routine Introduction	Routine Terminator
Niladic	VECTOR fN1 NNNN	RTS
Monadic	VECTOR fM1 MMMM	RTS
Dyadic	VECTOR fD1 DDDD	RTS
Header	VECTOR fH1 HHHH	RTS

The routines may access the value pointer and the result pointer at 24 (hex) and 28 (hex), respectively.

ADDITIONAL FUNCTIONS

In the following sections, the functions QDIV, QMULT, MIN, MAX, AND, OR, XOR, ⋀, and MOD are discussed. As these are dyadic functions, usage mimics multiply, divide, add, subtract, etc.

The monadic functions, CHR$, ABS, and SGN are also discussed.

QMULT and QDIV Functions

The QDIV command provides a method of fast division when the divisor is a power of two. For example, the program segment

```
100 QDIV 4
```

divides the initial result value of 100 by 4 and causes the new result to equal 25.

The QMULT command allows fast multiplication when the multiplier is a power of two. The segment

```
100 QMULT 8
```

multiplies the initial result by 8 and gives the new result 800.

Both QDIV and QMULT retain the sign of the initial result.

Minimization and Maximization Functions

The MIN function compares the result with values found in the argument list. If the value is less than the result, then the result is replaced by the value. If the value is greater than or equal to the old result, then the old result is retained.

The minimum function may be used to provide a ceiling, or upper limit, to values which can be assumed by variables. For example, the segment

```
X MIN Xmax @ X
```

replaces X by the ceiling value, Xmax, if the initial value is higher than Xmax.

The MAX function compares the result with values found in the argument list. If the value is greater than the result, then the result is replaced by the value. If the value is less than or equal to the old result, then the result is not changed.

The maximum function provides a floor, or lower limit, to values which can be assumed by variables. To limit the variable X to a range above Xmin, for example, the following segment is used.

```
X MAX Xmin @ X
```

If X is initially larger than Xmin, then no change is made and the old value is saved. When X is smaller than Xmin, then Xmin replaces X.

Both MIN and MAX are dyadic functions and may have several values in their argument list. This is useful when a large number of argument values must be searched for the maximum or minimum.

```
10 X0 MIN X1 X2 X3 X4 @ minvalue
20 X0 MAX X1 X2 X3 X4 @ maxvalue
```

Line 10 searches several data items to yield the minimum value while line 20 searches the same list to find the maximum value. The variables, minvalue and maxvalue, indicate lower and upper bounds for the series, X0, X1,..,X4.

Application - Variable Limiting

The following program segment limits the variable X to a range between 100 and 200 units.

```
X MAX 100 MIN 200 @ X
```

For comparison, the following shows the same program written in BASIC.

```
10 IF X ≤ 100 THEN X = 100
20 IF X ≥ 200 THEN X = 200
30 ....
```

Although more readable, the BASIC version requires twice as many operations as the condensed BETA notation.

The same limiting could be obtained using the following BETA program segment.

```
10 TEST X-100 IFPL 20 \100 @ X
20 TEST X-200 IFMI 30 \200 @ X
30 ....
```

This version illustrates appropriate application of the TEST instruction. This instruction duplicates the load function (\). In this example, the use of TEST is preferred in order to high-light subsequent branch activity.

Logical Functions

The operators, AND, OR, and XOR perform the indicated logical function on corresponding bits of the result and argument value.

The AND function may be used to mask high or low bytes of a number. When high bytes are masked out the AND function performs a folding function. This is useful for graphical presentation of high accuracy data. See Part Three for examples. When low bytes are masked out the effects of data round-off are simulated.

The XOR function may be used to logically complement the result when the argument value is completely filled with ones (i.e. equal to minus one). With zero argument value the XOR function leaves the result unchanged. Selected bits in the result may be 'flipped' when corresponding bits of the argument value are set to one.

When a logical function is applied to different length data the smaller word is extended by duplicating the sign bit.

Modularity Function

The MOD function may be used for base conversion, random number generation, variable folding, etc. As implemented, the modularity function uses the divide function but replaces the result with the integer remainder instead of the quotient. The expression, X MOD base, is equivalent to X-(X/base*base). Several examples using the modularity function are presented in Part Three.

Print Directives

The monadic function CHR$ may be used to send non-alphanumeric data to the printer. These cause specific printer action or reconfiguration. For example, a line feed results when the following segment is executed.

```
PRINT CHR$ 13
```

Consult your printer manual for numbers to be used in the CHR$ argument.

Absolute and Sign Functions

The ABS function computes the absolute value of the argument. This provides a measure of how far the argument deviates from zero without regard for sign.

The SGN function transforms a positive number to +1 while a negative number is transformed to -1. This function is useful for control problems where only one bit of information is needed to switch a light, heater, or relay actuated mechanism.

NOT Function

The NOT function computes the logical complement of its argument value. This is a monadic function. Hence usage mimics ABS, SGN, etc.

Part Three

Application Examples

Part Three - APPLICATION EXAMPLES

INTRODUCTION

The following programs have been selected to represent possible system applications. Comments precede each program. These programs may be modified and combined with others created by the user.

An informal name convention is used herein. Pre-defined functions and user-defined subroutine names are fully capitalized. Variable and label names are lower cased. However, the first letter of a variable or label may be capitalized (e.g., a proper name). Also, single upper-case letters (A through H) are used for array names while single upper-case letters (I through M) are used for loop index variables.

Benchmark Programs

Benchmark programs are useful for comparing different systems. Figure 3-1 shows two popular timing benchmarks. The first program loops 10,000 times through line 26 which performs several arithmetic operations. (See Microcomputing, April 1982, p. 140, for comparison of this benchmark on several systems.)

The second benchmark, starting at line 50, modifies the preceding by reversing the divide/multiply sequence. This requires four byte precision compared with two for the first example. Additional run time is required due to added operations on higher bytes.

In line 56, note that the intermediate result requires four bytes precision while the end result, stored at variable A, requires only two bytes. Precision levels are adjusted automatically subject to default limit values of seven and four bytes for intermediate and stored values, respectively. These limit values may be adjusted to as high as 14 bytes each by changing page zero locations 70 and 71, respectively.

Increasing the data size limit means that more memory must be dedicated to data storage. This may be important if large arrays are used in a program.

```
.LIST

1 !  BENCHMARK TIMING LOOPS-TRACK 02
2 !
10 !  TWO-BYTE LOOP- K/K*K+K-K @ answer
12 !
20 PRINT "Start!"
22 LET K=0
24 :loop1 \K+1 @ K-10000 IFEQ 28
26 K/K*K+K-K @ answer GOTO loop1
28 PRINT "Finish!"
30 END
36 !
38 !
40 ! FOUR-BYTE LOOP- K*K/K+K-K @ answer
42 !
50 PRINT "Start!"
52 LET K=0
54 :loop2 \K+1 @ K-10000 IFEQ 58
56 K*K/K+K-K @ answer GOTO loop2
58 PRINT "Finish!"
60 END
```

```
.RUN

start
finish



] !  Run time = 28 seconds.
```

```
.RUN 40

start
finish



] !  Run time = 54 seconds.
```

Figure 3-1

Benchmark Programs

Concurrent User Input

During program execution, you may enter any valid program line by pressing the CONTROL key until the concurrent prompt appears on the input display. (This prompt is the small @ appearing at the lower-left display.)

To demonstrate concurrent entry, run the program shown in figure 3-2. Press the CONTROL key down to enter the concurrent mode. Modify the period using the input line, @ 10 @ Imax. Try several values.

To halt this program - press the ESC key.

```
1 ! ************************************
2 ! ** CONCURRENT USER INPUT PROGRAM **
3 ! ********* SAVED ON TRACK 04 *******
8 ! ************************************
9 !
10 3 POKE 56832 !  Turn on tone generator.
20 LET Imax=50 LET J=0 ! Iniz. variables.
30 :loop
40 J POKE 57089 !  Set tone frequency.
50 DELAY Imax
60 J+1 @ J-256 IFMI loop
70 LET J=0 GOTO loop


.RUN


@  10 @ Imax ! Concurrent user input.


@ 5 @ Imax   ! Etc.


AT LINE NR 40@

\  1 POKE 56832   ! Turn off tone.
```

Figure 3-2

Concurrent User Input Program

Print Character Selection

This program shows how to use character codes to modify printer output. Four different character formats will be printed when this program is run. See Figure 3-3. Print directives are given in lines 10, 20, 30, and 40 to modify character size. The following lines in each case print a sample line. When long lines are printed it may be necessary to include a delay so that computer output does not overload the printer.

Line 50 restores the printer to standard format before returning to the editor.

```
2 ! ****************************
4 ! * PRINT CHARACTER SELECTION *
6 ! **** SAVED ON TRACK 06 ******
8 ! ****************************
9 !
10 PRINT CHR$(29)
12 PRINT 1234567890," CONDENSED PRINT  condensed print"
14 !
16 !
20 PRINT CHR$ 30
22 PRINT 1234567890," STANDARD PRINT  standard print"
24 !
26 !
30 PRINT CHR$(29),CHR$(31)
32 PRINT 1234567890," BOLD PRINT  bold print"
34 !
36 !
40 PRINT CHR$(30),CHR$(31)
42 PRINT 1234567890," WIDE PRINT  wide print"
44 !
46 !
50 PRINT CHR$(30)
52 PRINT "End of test."
```

```
. RUN

1234567890 CONDENSED PRINT condensed print

1234567890  STANDARD PRINT  standard print

1234567890  BOLD PRINT  bold print

1234567890   WIDE PRINT   wide print

End of test.
```

Figure 3-3

Print Character Selection

Print Graphics with Two Variables

This program shows how to plot two variables vs. the row variable. (Actually, one variable is set equal to the row variable while the other is set equal to twice the row variable.) See Figure 3-4.

Line 10 selects condensed character size to allow 128 characters/line. Lines 15 to 40 generate an endless loop which increments the loop index variable by 4 for each iteration.

Line 20 prints the index value starting at print column one. The ASCII character 'N', representing the index variable is printed in column N as set by the reference function. The line continuation function () is used to inhibit printout until the print line is completed in the succeeding program line.

Line 25 prints '0' in column 2N. The index value is again printed starting at print column 125.

To terminate this program - press the ESC key.

```
1 ! ****************************************
2 ! * PRINT GRAPHICS WITH TWO VARIABLES **
3 ! *********  SAVED ON TRACK 08  ********
4 ! ****************************************
5 !
10 PRINT CHR$(29) !  Set condensed print.
12 !
15 :loop LET 0=2*N
20 PRINT N,
22 ) PRINT TAB N,"N",
24 ) PRINT TAB 0,"0",
26 ) PRINT TAB 120,N
30 LET N=N+4 !  Increment index.
40 GOTO loop !  Restart at next row.
48 !
50 ! ****************************************
51 ! **** REMINDER ON SYSTEM OPERATION ****
52 ! ** TO INTERRUPT PROGRAM-PRESS ESCAPE *
54 ! *** TO CONTINUE PROGRAM-ENTER CONT ***
55 ! ****************************************
```

Figure 3-4

Print Graphics with Two Variables

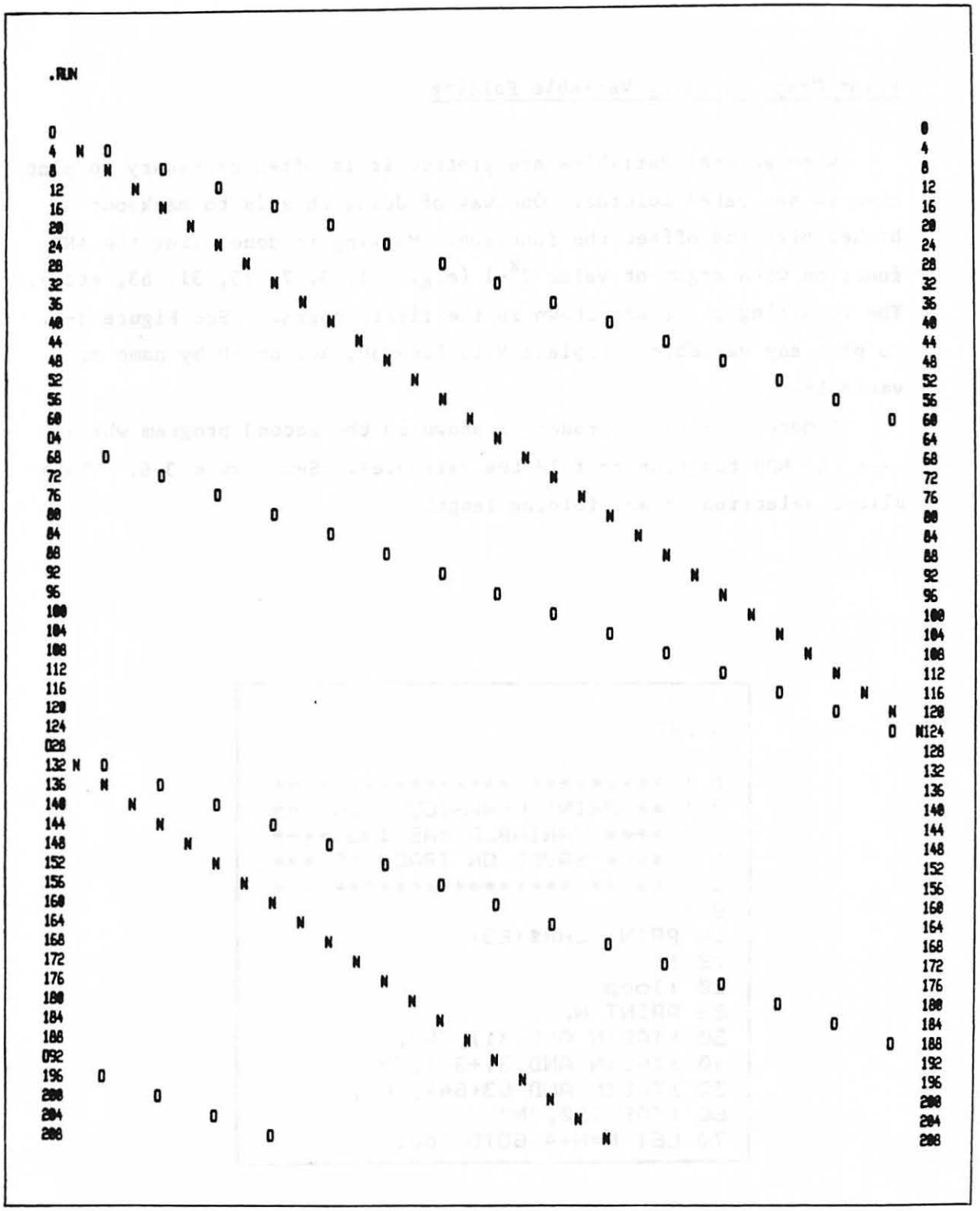


Figure 3-4 (cont.)

Print Graphics with Two Variables

Print Graphics using Variable Folding

When several variables are plotted it is often necessary to plot them in separated columns. One way of doing this is to mask out higher bits and offset the function. Masking is done using the AND function with argument value 2^k-1 (e.g., 1, 3, 7, 15, 31, 63, etc.). The resulting plots are shown in the first program. See Figure 3-5. To plot any variable - replace N in line 30, 40, or 50 by name of variable.

A more flexible approach is shown in the second program which uses the MOD function to fold the variables. See Figure 3-6. This allows selection of any folding length.

```
.LIST

1 ! ***************************
2 ! ** PRINT GRAPHICS USING **
3 ! **** VARIABLE MASKING ****
4 ! **** SAVED ON TRACK 10 ***
5 ! ***************************
9 !
10 PRINT CHR$(29)
12 !
20 :loop
22 PRINT N,
30 )TAB(N AND 31),"A",
40 )TAB(N AND 31+32),"B",
50 )TAB(N AND 63+64),"C",
60 )TAB 120,"N"
70 LET N=N+4 GOTO loop
```

Figure 3-5
Print Graphics using Variable Masking

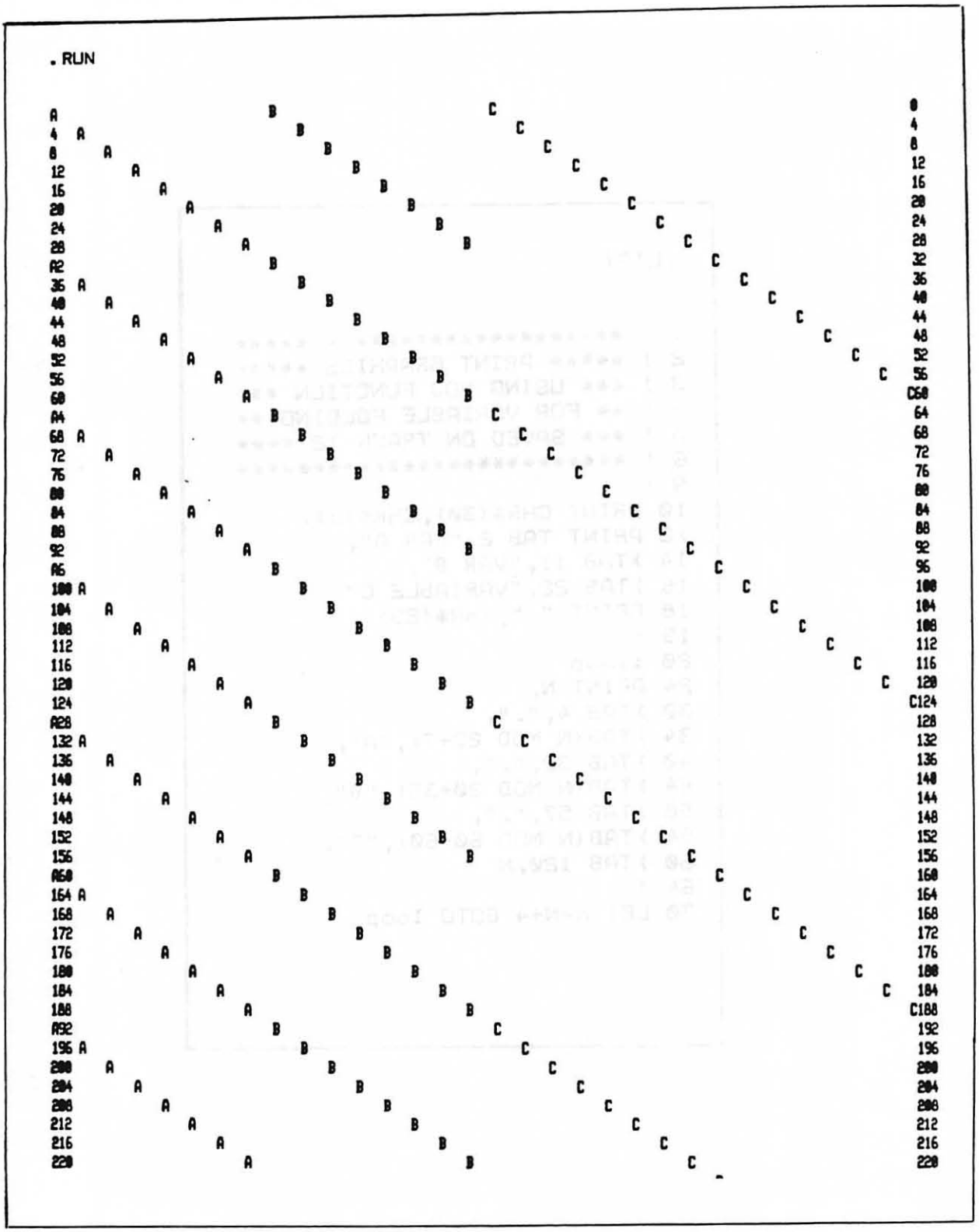


Figure 3-5 (cont.)
Print Graphics using Variable Masking

```
.LIST


1 ! ***************************
2 ! ***** PRINT GRAPHICS *****
3 ! *** USING MOD FUNCTION ***
4 ! ** FOR VARIABLE FOLDING **
5 ! *** SAVED ON TRACK 12 ****
6 ! ***************************
9 !
10 PRINT CHR$(30),CHR$(31)
12 PRINT TAB 2,"VAR A",
14 )TAB 11,"VAR B",
16 )TAB 22,"VARIABLE C"
18 PRINT " ",CHR$(29)
19 !
20 :loop
24 PRINT N,
30 )TAB 4,".",
34 )TAB(N MOD 20+7),"A",
40 )TAB 32,".",
44 )TAB(N MOD 20+35),"B",
50 )TAB 57,".",
54 )TAB(N MOD 60+60),"C",
60 )TAB 120,N
64 !
70 LET N=N+4 GOTO loop
```

Figure 3-6
Print Variables using Variable Folding

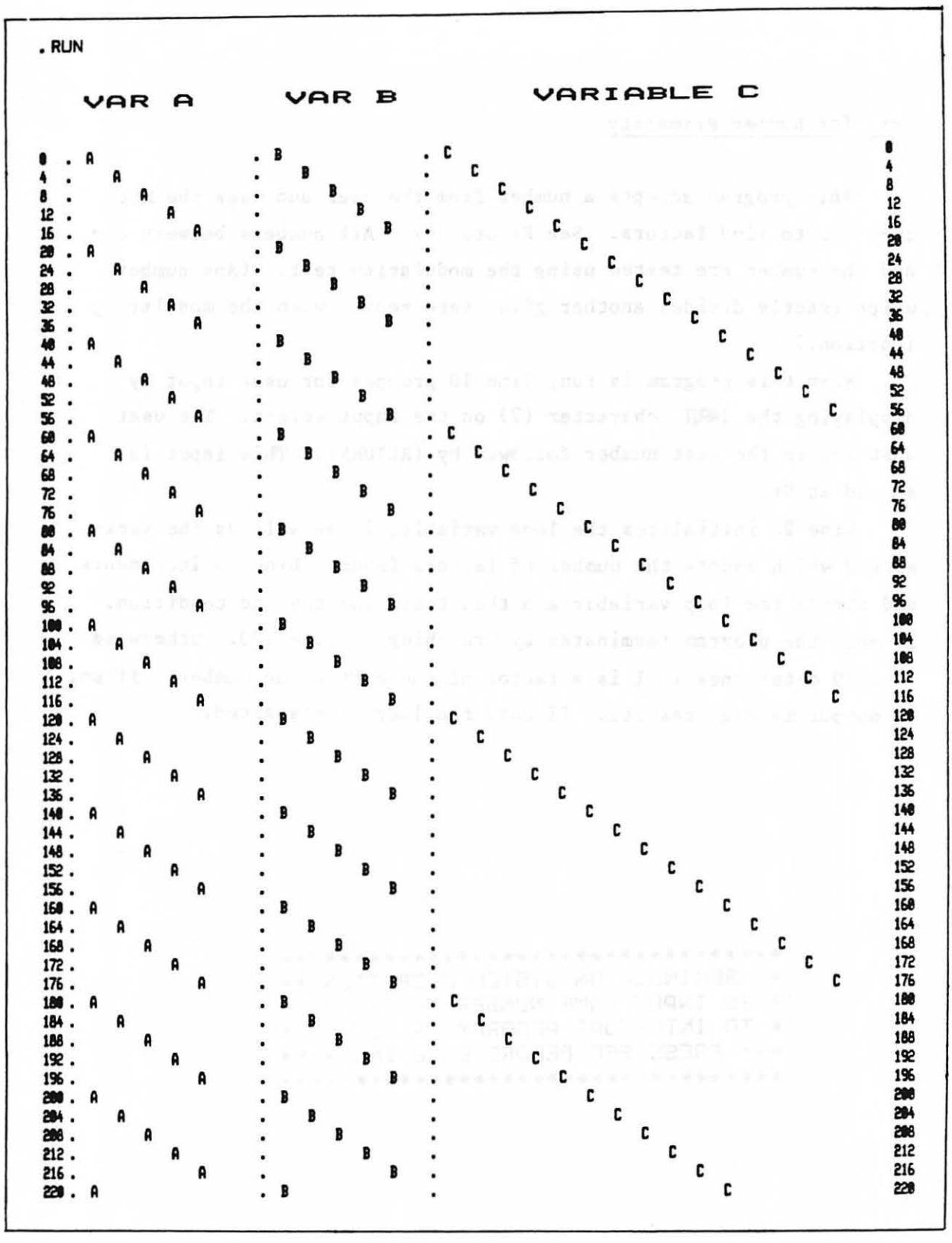


Figure 3-6 (cont.)
Print Variables using Variable Folding

Test for Number Primality

This program accepts a number from the user and uses the MOD function to find factors. See Figure 3-7. All numbers between one and the number are tested using the modularity test. (Any number which exactly divides another gives zero result with the modularity function.)

When this program is run, line 10 prompts for user input by displaying the INPUT character (?) on the input screen. The user must key in the test number followed by (RETURN). This input is stored at Nr.

Line 20 initializes the loop variable, I, as well as the variable J which counts the number of factors found. Line 30 increments and stores the loop variable and then tests for the end condition. If met, the program terminates by branching to line 100. Otherwise line 40 determines if I is a factor of the test prime number. If so, an output message results. If not, the loop is restarted.

```
!************************************
!** REMINDER ON SYSTEM OPERATION **
!* TO INPUT SAME NUMBER-PRESS (;) *
!* TO INTERRUPT PROGRAM-PRESS (;) *
!*** PRESS ESC BEFORE ENTERING ****
!************************************
```

```
1 ! ***************************
2 ! ** TEST FOR PRIME NUMBER **
3 ! **** SAVED ON TRACK 14 ****
4 ! ***************************
5 !
10 :start PROMPT "Enter number." INPUT Nr
20 LET I=1 LET J=1 ! Initialize indices.
30 :loop
32 TEST I+1 @ I-Nr IFEQ 100 ! Test for end.
40 TEST Nr MOD I IFNE loop ! Test I as factor.
45 !
47 !  Emit output message if I is a factor.
50 PRINT J," ",I,"times",Nr/I,"equals",Nr
60 LET J=J+1 ! Increment factor index.
70 GOTO loop
90 !
100 PRINT " " ! Skip a space.
110 PRINT " " GOTO start
```

```
.RUN

 ? 10

1     2 times 5 equals 10
2     5 times 2 equals 10

 ? 100

1     2 times 50 equals 100
2     4 times 25 equals 100
3     5 times 20 equals 100
4     10 times 10 equals 100
5     20 times 5 equals 100
6     25 times 4 equals 100
7     50 times 2 equals 100

 ? 25543

1     7 times 3649 equals 25543
2     41 times 623 equals 25543
3     89 times 287 equals 25543
4     287 times 89 equals 25543
5     623 times 41 equals 25543
6     3649 times 7 equals 25543

 ? 0  !  To exit - enter random nr. with ESC down.


AT LINE NR 20
```

Figure 3-7

Test for Number Primality

Showing that a Mersenne Number is Not Prime

This program shows how to work with numbers larger than four bytes. (This is the limit for input and output conversion.) See Figure 3-8. Large input numbers may be composed as shown in line 30. A technique for printing out large numbers is shown in lines 100 to 150. The result and data size limits must be raised before running this program.

A Mersenne number is any number in the series 0, 1, 3, 7, 15, ..., 2^k-1. Until 1903, it was speculated that all such numbers are prime. In 1903, F. N. Cole demonstrated that M67 $=2^{67}-1$ is a composite (i.e., non-prime) number with factors 193,707,721 and 761,838,257,287. This was done by hand calculation. (See Scientific American, February 1983, Letters Column.)

The actual value of M67 is equal to 147,573,952,589,676,412,927. The user may wish to modify the program to give single line output of this value. See Figure 3-4 which shows how this is done using the REF operator.

```
.LIST
1 !*****************************************************
2 !* SHOWING THAT A MERSENNE NR IS NOT PRIME-TRACK 16 *
3 !* POKE 14 INTO $46 AND $47 BEFORE RUNNING PROGRAM **
4 !* TO GIVE 14-BYTE PRECISION FOR RESULT AND DATA ****
5 !*****************************************************
9 !
10 LET M67=2^67-1
20 LET factor1=193707721
30 LET factor2=761838*1000000+257287
40 HEXPR Mnr,factor1*factor2,factor1*factor2-M67
45 PRINT " "
50 !
100 LET remainder=M67 !  Initialize remainder to M67.
105 LET divisor=1000^6 !  Set divisor larger than M67.
110 :loop
120 PRINT remainder/divisor
140 LET remainder=remainder MOD divisor
150 TEST divisor/1000 @ divisor IFNE loop
```

```
] 14 POKE 70 71
```

```
.RUN

$07FFFFFFFFFFFFFFF $07FFFFFFFFFFFFFFF $00

147
573
952
589
676
412
927
```

Figure 3-8

Showing that a Mersenne Number is Not Prime

Number Base Conversion

This program converts four numbers, representing place values of a number written with an arbitrary base, into a decimal number. See Figure 3-9. Each number must be positive and less than the base value.

Base Conversion Program

This program generalizes the number base conversion program to allow an arbitrary number of places. See Figure 3-10.

These programs convert numbers in an arbitrary base to decimal base. See Mersenne example (Figure 3-8 lines 100 - 150) for conversion to base 1000 output. This method may be modified for conversion from decimal to any base.

```
1 ! ************************************
2 ! ** NUMBER BASE CONVERSION-TRACK 18 **
3 ! ************************************
4 !
10 :loop
15 PROMPT "Enter base." INPUT base
20 PROMPT "Enter place values." INPUT a,b,c,d
25 !
30 LET result=a*base+b*base+c*base+d
40 PRINT a,b,c,d,"(base",base,") equals",result
50 GOTO loop
```

```
0 0 0 0 (base 2 ) equals 0
0 1 1 1 (base 2 ) equals 7
1 0 0 1 (base 2 ) equals 9
1 1 1 1 (base 2 ) equals 15


1 2 3 4 (base 8 ) equals 668
7 7 7 7 (base 8 ) equals 4095

1 2 3 4 (base 10 ) equals 1234
9 8 7 6 (base 10 ) equals 9876
9 9 9 9 (base 10 ) equals 9999

1 2 3 4 (base 16 ) equals 4660
1 0 0 0 (base 16 ) equals 4096
15 15 15 15 (base 16 ) equals 65535


AT LINE NR 20
```

Figure 3-9
Number Base Conversion

```
1 ! ************************************
2 ! ** BASE CONVERSION PROGRAM-TRACK 20 **
3 ! ************************************
9 !
10 PROMPT "Enter base." INPUT base
12 !
20 PROMPT "Enter nr of places." INPUT nr
22 !
30 PROMPT "Enter place values (separated by spaces)."
32 FOR I=nr-1 TO 0 INPUT A(I) NEXT
38 !
40 CALL CONVERT A result % nr-1,base
42 !
50 FOR I=nr-1 TO 0 PRINT A(I) NEXT ,
52 ! PRINT "base(",base,")equals",result
98 END
1000 !
1010 SUBR CONVERT array,answer % N,base
1020 LET answer=array(N)
1030 FOR I=N-1 TO 0
1040 LET answer=answer*base+array(I)
1050 NEXT RET
1060 DIM A(3)
```

```
1 0 0 1 base ( 2 ) equals 9


7 7 base ( 8 ) equals 63


1 2 3 4 base ( 10 ) equals 1234


7 15 15 15 base ( 16 ) equals 32767
```

Figure 3-10
Base Conversion Program

Clock Readout

The program listed in Figure 3-11 shows how to convert time (given in seconds) into hours, minutes, and seconds. This procedure should be useful for readout of time when an internal clock is available. Some machines have a hardware feature which stores time (usually in fractions of a second) into a designated memory location.

The total time is first divided by 3600 (seconds per hour) to derive the number of integer hours in the interval. The remainder is then divided by 60 (minutes per second) to derive the number of integer minutes in the interval. The remainder of this division is the seconds component. Use the MOD function to compute the remainder after integer division.

Components of a Number in Any Base

The program listed in Figure 3-12 shows how to convert any positive number into components representing the number in any base. For example, the number 246 is represented in base 10 by the coefficients of a polynomial in 10^n as

$$(2*10^2) + (4*10^1) + (6*10^0)$$

Briefly, the algorithm finds the highest place value by dividing the number by $10^2 = 100$ and printing this value. The remainder is then divided by $10^1 = 10$ giving the tens place. Finally, the remainder is assigned to the ones place.

The similarities between this program and the clock readout program shown in the previous example should be noted.

```
 1 ! ****************************************
 2 ! ******* CLOCK READOUT-TRACK 22 *********
 3 ! ****** CONVERT TIME (IN SECONDS) *******
 4 ! ** TO TIME (IN HOURS,MINUTES,SECONDS) **
 5 ! ****************************************
 9 !
10 PROMPT "Enter elapsed time (in seconds)."
12 INPUT Tsec
14 !
20 LET Nhr=Tsec/(60*60)
22 LET Nsec=Tsec MOD (60*60)
28 !
30 LET Nmin=Nsec/60
32 LET Nsec=Nsec MOD 60
38 !
40 PRINT TAB col,Nhr,":",Nmin,":",Nsec
44 !col 10,12,15,17,20

       0  :  1 : 40


       0  : 16 : 40

       2  : 46 : 40


      27  : 46 : 40

.M

]   !  Entry times were 100,1000,10000,100000 seconds.
```

Figure 3-11
Clock Readout Program

```
1 ! ****************************************
2 ! ** COMPONENTS OF A NUMBER IN ANY BASE **
3 ! ******* SAVED ON TRACK 24 **************
4 ! ****************************************
9 !
10 PROMPT "Enter number and base (in decimal)."
20 :start
22 INPUT number,base
28 !
30 LET remainder=number LET divisor=1
32 TEST divisor*base @ divisor*base-number IFMI 32
38 !
40 :loop PRINT remainder/divisor,
42 ) LET remainder=remainder MOD divisor,
44 ) TEST divisor/base @ divisor IFNE loop
```

```
.RUN

 : 9 2

1 0 0 1

 : 63 8

7 7

 : 1234567890 10

1 2 3 4 5 6 7 8 9 0

 : 32767 16

7 15 15 15
```

Figure 3-12
Number Represented in any Base

Square Root Algorithm

Frequently used functions should be written in native code to minimize run time. To reduce development time, it is often convenient to prototype the function by writing a high-level version. This also allows alternate versions to be easily written for comparison. Figure 3-13 shows a square rooter prototyped in high-level language.

When the series of odd integers are summed, starting with one, the result is a perfect square. For example, 1+3 = 4, 1+3+5 = 9, ... etc. This relationship is used in the square root subroutine to sum N odd integers until the input argument is matched. The number N is returned as the output argument which is the square root value. For input arguments which are not perfect squares, fractional values are dropped.

This algorithm is quite slow when the square root of large numbers must be found.

```
40000 ! ***************************
40001 ! ** SQUARE ROOT ALGORITHM **
40002 ! **** SAVED ON TRACK 26 ****
40003 ! ***************************
40009 !
40010 SUBR SQRT root % number
40012 TEST number IFMI 40030 IFNE 40040
40020 LET root=0 RET
40030 PRINT "negative argument" END
40040 1 @ odd @ sum-number IFEQ 40070
40050 odd+2 @ odd+sum @ sum-number IFMI 40050 IFEQ 40070
40060 LET root=odd QDIV 2 RET
40070 LET root=odd QDIV 2+1 RET
40080 ! END OF SQUARE ROOT ALGORITHM
40090 !
40098 !
1 ! ************************************************
3 ! ** SIMPLE PROGRAM USING SQUARE ROOT ALGORITHM **
5 ! *************  SAVED ON TRACK 26  *************
7 ! ************************************************
9 !
10 :loop PROMPT "Enter positive number." INPUT nr
20 CALL SQRT sqroot % nr
30 PRINT "The square root of",nr,"is",sqroot
40 GOTO loop
   .RUN

   The square root of 0 is 0
   The square root of 1 is 1
   The square root of 2 is 1
   The square root of 3 is 1
   The square root of 4 is 2
   The square root of 5 is 2
   The square root of 6 is 2
   The square root of 7 is 2
   The square root of 8 is 2
   The square root of 9 is 3
   The square root of 10 is 3
   The square root of 100 is 10
   The square root of 1000 is 31
   The square root of 10000 is 100
   The square root of 100000 is 316
   The square root of 1000000 is 1000
   The square root of 10000000 is 3162
```

Figure 3-13
Square Root Algorithm

Euclid's Greatest Common Denominator Algorithm

This program demonstrates the recursive property of subroutines by using Euclid's greatest common denominator algorithm. This algorithm finds the largest integer which divides two given numbers. See Figure 3-14. The text for the subroutine, called GCD, is shown in lines 100 to 150. Lines 10 to 40 shows use by a main program which accepts two integer values and uses these as input arguments in a call to GCD.

Line 20 prompts for user input. The first value is stored at integer1 and the second value is stored at integer2. These are used as arguments for the subroutine call to GCD in line 30. Following return from this call the program is restarted.

Euclid's algorithm is based on the assumption that a common divisor of two integers is also a divisor of the absolute difference of the two integers. Recursive calls to GCD replace the larger of two initial integers by the absolute difference. The call sequence is terminated when the two integers are equal. This test is done in line 120.

To terminate this program - following the INPUT prompt, enter two random numbers and press ESC followed by return.

The 256 byte stack size limits recursion to 85 levels. Use of large numbers may cause return stack damage.

```
!**********************************
!** REMINDER ON SYSTEM OPERATION **
!** TO EXIT INPUT LOOP PRESS (;) **
!** TWICE (ONCE FOR EACH VALUE) ***
!**  PRESS ESC BEFORE ENTERING ****
!**********************************
```

```
1 !  EUCLID'S GCD ALGORITHM
2 !  SAVED ON TRACK 28 ****
3 !
10 PROMPT "Enter two integers."
12 :start
20 INPUT integer1,integer2
30 CALL GCD % integer1,integer2
40 PRINT " " GOTO start
98 !
100 SUBR GCD % I,J
110 PRINT I,J
120 TEST I-J @ K IFEQ 150 IFMI 140
130 CALL GCD % K,J RET
140 CALL GCD % I,NEG K
150 RET
```

```
 9 15
 9 6
 3 6
 3 3

 15 9
 6 9
 6 3
 3 3

 49 18
 31 18
 13 18
 13 5
 8 5
 3 5
 3 2
 1 2
 1 1

 48 18
 30 18
 12 18
 12 6
 6 6
```

Figure 3-14

Euclid's Greatest Common Denominator Algorithm

Complex Number Facility

Some applications may require use of complex numbers. While calculation with complex numbers is possible using only real numbers it is much more convenient to define functions which operate on complex numbers.

In the program shown in Figure 3-15, eight subroutines are given which operate on two element arrays. Array element zero is the real component while array element one is the imaginary component.

These programs use an intermediate result register which is itself a two element array. This array must be dimensioned in order to reserve space. Note that the name, dummy, although not dimensioned is used as an array. This is true in general for dummy names used in subroutine reference lists. These are vectored to locations found in the call assignment list. Dimension statements for dummy reference parameters waste storage space.

Some example segments using the complex number facility are shown in Figure 3-16. Notice that when the % separator is used in the subroutine its use in the call list is optional.

```
40000 !*********************************
40002 !** COMPLEX ARITHMETIC FACILITY **
40004 !****** SAVED ON TRACK 30 ********
40008 !*********************************
40009 !
40010 DIM res(1)
40012 SUBR COMPLEX % real,imag
40014 LET res(0)=real LET res(1)=imag
40016 RET
40018 !
40020 SUBR LOAD dummy
40022 LET res(0)=dummy(0)
40024 LET res(1)=dummy(1)
40026 RET
40028 !
40030 SUBR SAVE dummy
40032 LET dummy(0)=res(0)
40034 LET dummy(1)=res(1)
40036 RET
40038 !
40040 SUBR SHOW dummy PRINT dummy(0),dummy(1) RET
40048 !
40050 SUBR PLUS dummy
40052 LET res(0)=res(0)+dummy(0)
40054 LET res(1)=res(1)+dummy(1)
40056 RET
40058 !
40060 SUBR MINUS dummy
40062 LET res(0)=res(0)-dummy(0)
40064 LET res(1)=res(1)-dummy(1)
40066 RET
40068 !
40070 SUBR TIMES dummy
40072 LET real=res(0)*dummy(0)-(res(1)*dummy(1))
40074 LET imag=res(1)*dummy(0)+(res(0)*dummy(1))
40076 LET res(0)=real LET res(1)=imag RET
40078 !
40080 SUBR OVER dummy
40082 LET temp=dummy(0)*dummy(0)+(dummy(1)*dummy(1))
40084 LET real=res(0)*dummy(0)+(res(1)*dummy(1))
40086 LET imag=res(1)*dummy(0)-(res(0)*dummy(1))
40088 LET res(0)=real/temp LET res(1)=imag/temp
40090 RET
40092 !
40093 ! ****************************************
40094 ! ** END OF COMPLEX ARITHMETIC FACILITY **
40095 ! ****************************************
```

Figure 3-15
Complex Number Facility

```
1 ! ****************************
2 ! ** EXAMPLES USING COMPLEX **
3 ! **** ARITHMETIC FACILITY ***
4 ! **** SAVED ON TRACK 30 *****
5 ! ****************************
9 !
10 CALL COMPLEX 15,10 CALL SAVE A
12 CALL SHOW A
18 !
20 CALL LOAD A CALL SAVE B
22 CALL SHOW A CALL SHOW B
28 !
50 CALL LOAD A CALL PLUS B CALL SAVE C
52 CALL SHOW A CALL SHOW B CALL SHOW C
58 !
60 CALL COMPLEX 20,30 CALL TIMES B CALL SAVE C
62 CALL SHOW C
68 !
70 CALL COMPLEX 1000,2000 CALL OVER B CALL SAVE C
72 CALL SHOW C
78 !
80 END
98 !
100 DIM A(1) DIM B(1) DIM C(1)

 .RUN

 15 10
 15 10 15 10
 10 30 20
 0 650
 107 61
```

Figure 3-16
Use of Complex Number Facility

```
\ PEEK $4E XOR 8 POKE $4E


.LIST

1 !  EXAMPLES USING COMPLEX FACILITY
2 !
10 COMPLEX 15,10 SAVE A
12 SHOW A
14 !
20 LOAD A SAVE B
22 SHOW A SHOW B
24 !
50 LOAD A PLUS B SAVE C
52 SHOW A SHOW B SHOW C
54 DELAY 500
56 !
60 COMPLEX 20,30 TIMES B SAVE C
62 SHOW C
66 DELAY 500
68 !
70 COMPLEX 1000,2000 OVER B SAVE C
72 SHOW C
74 !
98 END
99 !
100  DIM A(1) DIM B(1) DIM C(1)


.RUN

15 10
15 10 15 10
15 10 15 10 30 20
0 650
107 61
```

Figure 3-16a
Listing with CALL Suppressed

Window Facility

Windows are rectangular areas on the video screen which may be used for selected data display. The window facility listed in Figure 3-17 defines several subroutines which access the output screen. Page zero locations of screen constants are given in Figure 3-18. The routines CLEAR and SCROLL are assembly-coded for high-speed operation. These are available for program use by the LINK command.

Figure 3-19 shows usage of the window facility to create a small window within the output screen. The output screen format is normally restored before ending the program.

```
.L40000

40004 ! ******************************
40005 ! *** VIDEO WINDOW FACILITY ***
40006 ! ***** SAVED ON TRACK 32 *****
40007 ! ******************************
40009 !
40010 SUBR OPEN label % tab1,blank
40012 REF label READ begrow,nrow,begcol,ncol
40014 begrow*64+begcol+53248 DPOKE 104
40016 begrow+nrow*64+begcol+tab1+53248 DPOKE 106 108
40018 ncol POKE 110 \nrow POKE 111
40022 blank POKE 113 LINK CLEAR RET
40028 !
40030 !
40038 !
50000 :CLEAR 37091 :SCROLL 37094
50008 :full 0 31 0 63
50010 :normal 0 19 0 63
50012 !
50014 ! ******************************
50016 ! ** END OF WINDOW FACILITY **
50018 ! ******************************
```

Figure 3-17
Video Window Facility

HEX	DEC	FUNCTION
$68	104	Screen upper-left address.
$6A	106	Screen output reset.
$6C	108	Screen output pointer.
$6E	110	Number of columns less one.
$6F	111	Number of rows less one.
$70	112	Increment from row-to-row.
$71	113	Screen blank character.

Figure 3-18
Video Screen Page Zero Addresses

```
1 ! ************************************
2 ! ** EXAMPLES USING WINDOW FACILITY **
3 ! ** SAVED ON TRACK 32 ***************
4 ! ************************************
9 !
10 ! First clear output screen.
12 CALL OPEN normal % 1,32
18 !
20 ! Specify pane size.
24 !pane 10 10 10 27
28 !
30 !Open pane.
32 CALL OPEN pane % 1,161
34 PRINT "  Windows allow dedicated"
35 DELAY 100
36 PRINT "screen areas to be used"
37 DELAY 100
38 PRINT "for specialized displays."
39 DELAY 100
40 SCR DELAY 100
42 PRINT "  Scroll entry is used"
43 DELAY 100
44 PRINT "for this screen."
45 DELAY 100 LINK SCROLL
46 DELAY 100 LINK SCROLL
47 DELAY 100 LINK SCROLL
48 DELAY 6000 LINK CLEAR
49 !
50 PRINT AT(2*64+2),"  This screen uses the"    "
52 PRINT AT(4*64+2),"PIXEL function to ad-"
54 PRINT AT(6*64+2),"dress picture elements"    "
56 PRINT AT(8*64+2),"individually."
58 DELAY 6000
59 !
60 LINK CLEAR
62 PRINT AT(6*64+2),"(( PRESS 'C' TO FINISH ))"
64 ESC
98 !
100 CALL OPEN full % 1,32
102 CALL OPEN normal % 8,32
104 !
106 !
110 ! ************************************
112 ! * END OF WINDOW FACILITY EXAMPLES *
114 ! ************************************
--
```

Figure 3-19
Use of Video Window Facility

BETA/65 LOAD PROCEDURE

(1) BOOT TO OS65D DISK OPERATING SYSTEM

(2) INSERT YOUR BETA/65 DISK INTO THE A-DRIVE

(3) LOAD AND GO ACCORDING TO FOLLOWING TABLE

MODEL/VERSION	LOAD TRACK	ENTRY ADDRESS
C4P v.3.1	A*LO 02	A*GO 5000
C1P v.3.1	A*LO 02	A*GO 5004
C4P v.3.3	A*LO 02	A*GO 5808
C1P v.3.3	A*LO 02	A*GO 580C

(4) TYPE "HELP" AND PRESS THE RETURN KEY

Notice: Please refer to your OS65/D Users Manual for instructions to initialize your system to A* command prompt. Refer to your BETA/65 Programming Manual (Appendix A) for further details.

BETA/65 BACKUP PROCEDURE

(1) INSERT A BLANK DISK INTO A-DRIVE

(2) INITIALIZE BLANK DISK USING A*INIZ

(3) REPLACE DISK WITH BETA/65 DISK

(4) LOAD BETA/65 OBJECT CODE WITH A*LO 02

(5) RE-INSERT THE BLANK DISK INTO A-DRIVE

(6) SAVE OBJECT CODE WITH A*PU 02

Warning: Do not remove write-protect tab from your BETA/65 diskette. Maintain protected backup copies.

BETA/65 LOAD USING BEXEC*

(1) LOAD BACKUP DISK FROM EM USING :!LO 02

(2) MODIFY HEADER ACCORDING TO MODEL AS SHOWN

C4P v.3.1	C1P v.3.1	C4P v.3.3	C1P v.3.3
327E/4C	327E/4C	3A7E/4C	3A7E/4C
327F/00	327F/04	3A7F/08	327F/0C
3280/50	3280/50	3A80/58	3A80/58

(3) RESAVE ON BACKUP DISK USING :!PU 02

(4) MODIFY AND SAVE BEXEC* ON OS65D (SEE LISTING 1)

On boot prompt remove OS65D disk from drive and replace with BETA/65 disk, modified according to steps (1), (2), and (3) above. Enter "BETA" to load BETA/65 system.

```
;LIST     LIST

 10 REM BASIC EXECUTIVE
 20 REM
 24 REM SETUP INFLAG & OUFLAG FROM DEFAUL
 25 X=PEEK(10950): POKE 8993,X: POKE 8994,X
 30 PRINT:PRINT "BASIC EXECUTIVE FOR OS-65D VERSION 3.1":PRINT
 40 PRINT "18 SEP 1978 RELEASE"
 50 GOTO 100
 60 PRINT : INPUT "FUNCTION";A$
 70 IF A$="CHANGE" THEN RUN "CHANGE"
 80 IF A$="DIR"    THEN RUN "DIR   "
 90 IF A$="UNLOCK" THEN 10000
 92 IF A$="BETA"   THEN 11000
 100 PRINT
 110 PRINT "FUNCTIONS AVAILABLE:"
 120 PRINT "   CHANGE - ALTER WORKSPACE LIMITS"
 130 PRINT "   DIR    - PRINT DIRECTORY"
 140 PRINT "   UNLOCK - UNLOCK SYSTEM FROM END USER MODIFICATIONS"
 142 PRINT "   BETA   - LOAD BETA/65 SYSTEM DISK
 150 GOTO 60
 10000 REM
 10010 REM UNLOCK SYSTEM
 10020 REM
 10030 REM REPLACE "NEW" AND "LIST"
 10040 POKE 741,76 : POKE 750,78
 10050 REM
 10060 REM ENABLE CONTROL-C
 10070 POKE 2073,173
 10080 REM
 10090 REM DISABLE "REDO FROM START"
 10100 POKE 2893,55 : POKE 2894,8
 10110 PRINT : PRINT "SYSTEM OPEN" : END
 11000 REM
 11010 REM LOAD BETA/65
 11020 DISK!"XQT 02"

OK
```

```
 10                ; ****************************
 20                ; * DIRECT BOOT ROUTINE ******
 30                ; * MODIFIES OS65D TO PERMIT *
 40                ; * LOAD AND GO OF MACHINE ***
 50                ; * CODE SAVED ON TRACK 02 ***
 60                ; * BY USING ( BREAK-D ) *****
 70                ; ****************************
 90                ;
100 00E1=          OSIBAD=$E1
110                ;
120 2321=          INDST =$2321
130 2322=          OUTDST=$2322
140                ;
150 2A51=          OS65D3=$2A51
160 2AC6=          DEFDEV=$2AC6
170                ;
180 2CE4=          BUFBYT=$2CE4
190 2E1E=          OSBUF =$2E1E
200                ;
210 2C22=          XQT   =$2C22
220                ;
230 2294                 *=$2294
240                ;
250 2294 A21E            LDX #OSBUF
260 2296 86E1            STX OSIBAD
270 2298 A22E            LDX #OSBUF/256
280 229A 86E2            STX OSIBAD+1
290                ;
300 229C AEC62A          LDX DEFDEV     ; LOAD DEFAULT DEVICE
310 229F 8E2123          STX INDST      ; SET DEFAULT INPUT
320 22A2 8E2223          STX OUTDST     ; SET DEFAULT OUTPUT
330 22A5 D00C            BNE MSSG
340 22A7 F00A            BEQ MSSG
350 22B3                 *=$22B3
360 22C4           MSSG  *=$22C4
370 22C4 A000      SETBUF LDY #0
380 22C6 8CE52C          STY BUFBYT+1   ; RESET BUFFER INDEX
390                ;
400 22C9 A930            LDA #'0        ; PUT '02' IN BUFFER
410 22CB 8D1E2E          STA OSBUF
420 22CE A932            LDA #'2
430 22D0 8D1F2E          STA OSBUF+1
440 22D3 A90D            LDA #13
450 22D5 8D202E          STA OSBUF+2
460                ;
470 22D8 4C222C          JMP XQT
490                ;
500                ; ****************************************
510                ; * NOTICE-USE THE FOLLOWING PROCEDURE  ******
520                ; * TO PREPARE BACKUP DISK FOR DIRECT BOOT  **
530                ; ****************************************
540                ; ** (1) COPY TRACK 02 TO BACKUP DISK *********
550                ; ** (2) COPY TRACKS 00 AND 01 TO SAME DISK ***
560                ; ** (3) ASM BOOT ROUTINE TO MODIFY PAGE $22 **
570                ; ** (4) EXIT TO DOS AND INSERT OS65D V 3.1 ***
580                ; ** (5) *CA 0200=13,1  LOADS COPY ROUTINE ****
590                ; ** (6) *GO 0200  ENTERS COPY ROUTINE  *******
600                ; ** (7) INSERT BACKUP DISK INTO DRIVE *********
610                ; ** (8) W2200/2200,8 SAVES MODS TO TRACK 00 ***
620                ; ** (9) TO TEST-PRESS BREAK-THEN D ************
630                ; ****************************************
```

APPENDIX B - ERROR CODES

Tables B-1 and B-2 show error codes for line entry and run execution, respectively.

Forward Reference Errors - A forward reference error following a RUN command means that the program contains a call to a subroutine which has not been defined. Use SLIST to list the names of defined subroutines. All referenced names (which also appear after the keyword CALL) must be defined in the program text using a subroutine definition statement i.e., SUBR NAME, etc.

Data Integrity - When possible, avoid use of dummy names outside of the defining subroutine. As all names are shared, unexpected data modifications may occur if names are not localized.

Use of dummy names outside of the defining subroutine is permitted, notwithstanding. Please review subroutine and call list rules. Remember that dummy parameters named in the SUBR reference list are assigned to locations given in the CALL assignment list. Dummy parameters are not reset by RET.

Code Damage - Program code can be damaged by improper use of the POKE and/or LINK instructions. Maintain backup copies! If damage is not extensive (i.e., within one line) it is often possible to repair by line re-entry. If system damage is suspected then both system and program must be reloaded.

Word Size Limit - Result and data size limits are initialized to 7 and 4 bytes, respectively. These values may be changed to as high as 14 bytes each. Before running the program poke new size limits into page zero locations 70 and 71 for result and data size, respectively.

ENTRY ERROR CODES

AN - Assignment to number.

AS - Assignment to string.

BD - Bad data-improper ASCII number.

BR - Branch to line 65535 or greater.

CC - Cannot continue terminated program.

CD - Program code damage.

CM - Edit command error.

IC - Illegal ASCII character.

MP - Missing primary.

MK - Missing keyword.

ML - Program or symbol table memory limit.

MN - Missing number or string.

NL - Symbol table name limit.

UV - Undefined variable.

Table B-1

Entry Error Codes

RUN-TIME ERROR CODES

AD - Reference to undimensioned array.

AR - Array index exceeds dimension range.

BR - Branch to non-label or backward line branch.

DM - Data mismatched with function.

DR - Data reference error.

DS - Data size larger than limit.

DZ - Attempt to divide by zero.

FR - Forward reference-CALL to undefined SUBR.

HS - Handshake error-printer not active.

NA - Negative argument where positive value req'd.

NT - No TO in FOR...NEXT loop.

OD - Out of data in READ instruction.

PL - Parentheses level-reswlt stack limit exceeded.

RM - Result mismatched with function.

RS - Result size larger than limit.

SD - Stack damage-improper FOR..NEXT or SUBR entry.

TF - Too few parameters in CALL list.

TM - Too many parameters in CALL list.

UF - Undefined function used in program.

Table B-2

Run-time Error Codes

Selection of high values for result size may cause PL (parentheses level) error due to result stack limit of 128 bytes. At 8 bytes per level this allows 16 levels.

Selection of high values for data size may cause ML (memory limit) errors at run time, especially when large arrays are used.

Symbol Table Full - Up to 96 variable and/or label names may be used within a single program. Up to 128 subroutine names may be defined as well. If these limits are exceeded, use VLIST (or SLIST) along with a program listing to find inactive names. Use the VCHANGE (or SCHANGE) command to modify names.

With variables, it is often possible to reduce the number of names by using the same name for different variables. This is often possible with subroutine dummy variables (but beware of side effects). It may also be possible to organize several variables into an array so that only one name is needed.

The number of bytes available for symbol table use may be displayed by the edit command FREE. This command also displays the number of bytes remaining in the program space.

Undefined Variable. An undefined variable (UV) error following entry of an immediate expression means that the expression contains a name not found in the symbol table. Variable and subroutine names are admitted into the symbol table only by entry (from the edit mode) of a program line containing the new name.

Parentheses Level. Selection of high values for result size may cause PL (parentheses level) error due to result stack limit of 128 bytes. At 8 bytes per level this allows 16 levels. Parenthetic expressions using long strings may overflow at lower levels.

A PL error will also occur if negative levels are entered or if an expression does not return to the zero (base) level.

Use of Minus Character. The minus character (-) has three different meanings in BETA notation:

1. Minus Sign - denotes a signed number with negative value.
2. Subtraction - appearing between two primaries, the minus character denotes subtraction of the primary following from the current result.
3. Negation - directly following any function, the minus character denotes negation of the primary following.

In expressions, any minus character immediately following a primary denotes subtraction. For example,

A - B, 2 - 3, (A + B) - (C + D), SQR(A) - ABS(B)

means that the second primary in each of the four expressions is to be subtracted from the first.

In expressions, any minus character immediately following a keyword, comma, left parenthesis, or function (monadic or dyadic) denotes negation. For A = 2, PRINT -A, -A + A, -(A + A), A * -1, ABS(-2) will output -2 0 -4 -2 2. Removal of the parentheses from ABS(-2) will give the same result but may appear to the unwary as sub traction. For readability, monadic primaries should be enclosed by parentheses.

In a data field containing only numbers, the minus character invariably denotes a negative number. The BETA preprocessor actually codes all negated numbers as negative (twos-complement) to enhance run speed.

To summarize: Following a function, the minus character denotes negation. Following a primary, the minus character denotes subtraction.

In a primary list where one or more of the primaries is negated, the negated primary(s) must be enclosed by parentheses. For example, the statement MAX A (-B) is the correct form when -B is compared to A. However, parentheses are optional with the first primary. To minimize confusion, use NEG, which also performs the negation function.

Use of Commas. Commas are used to partition expressions in keyword lists. For example,

```
PRINT  3 - 1, 2 + 2, 1 + 2 * 3
```

will print 2 4 9 into the output stream. Expressions may be of arbitrary complexity, subject only to line size limits. To extend a keyword list into the next line, use a comma to end the last expression on the first line and a "greater than" character (>) to begin the first expression on the next line. For example,

```
123 PRINT  3 - 1, 2 + 2
124 > 1 + 2 * 3
```

will cause printout (normally occurring at each end-of-line) to occur at the end of line 124.

Use of commas in assignment lists is optional. For example,

```
INPUT A A(1) A(2)
```

and

```
INPUT A,A(1),A(2)
```

will produce equivalent results.

Use of commas in data fields is also optional. Inconsistent use of the comma may cause some confusion. For example, the data field usage

```
: data 123,456 234,567
```

may appear to the unwary as two six-digit numbers when actually there are four numbers in this data field.

PAGE ZERO MEMORY MAP

HEX	DEC	FUNCTION
$00	0	BETA/65 re-entry direct.
$03	3	Mask constant=$E0
$04	4	ERROR indirect.
$06	6	LINESTART return indirect.
$08	8	Dyadic return indirect.
$0A	10	Monadic return indirect.
$0C	12	Pointer to primitive vectors.
$0E	14	Pointer to vector table.
$10	16	Pointer to primitive names.
$12	18	Pointer to name table.
$14	20	Reset to output buffer.
$16	22	Pointer to output buffer.
$18	24	Reset to serial buffer.
$1A	26	Pointer to serial buffer.
$1C	28	Source pointer.
$1E	30	Destination buffer.
$20	32	Pointer to translator code buffer.
$22	34	Pointer to run-time code.
$24	36	Pointer to run-time argument.
$26	38	Pointer to JMP indirect stack.
$28	40	Pointer to current result.

PAGE ZERO MEMORY MAP (CONT)

HEX	DEC	FUNCTION
$2A	42	Pointer to page-zero disk save.
$2C	44	Pointer to variables vector table.
$2E	46	Pointer to subroutines vector table.
$30	48	Pointer to program names table.
$32	50	Pointer to start of program code.
$34	52	Pointer to current program line.
$36	54	Pointer to start of program data.
$38	56	Pointer to available memory.
$3A	58	Spare word.
$3C	60	REF pointer to imbedded data field.
$3E	62	External pointer to machine data/code.
$40	64	FOR..NEXT LINESTART save buffer.
$42	66	FOR..NEXT CODEPOINT save buffer.
$44	68	TAB pointer to imbedded data field.
$46	70	Result size limit.
$47	71	Data size limit.
$48	72	Count of unassigned variables.
$49	73	Count of unassigned subroutines.
$4A	74	CONCUR interrupt timing count.
$4B	75	Program change flag.
$4C	76	Stack level for error recovery.
$4D	77	TAB flag.
$4E	78	Output format byte.
$4F	79	Spare format byte.
$50-$55		Program spares.

PAGE ZERO MEMORY MAP (CONT)

HEX	DEC	FUNCTION
$56	86	End of available program space.
$58-$5B		System spares.
$5C	92	Printer tab reset.
$5D	93	Printer column index.
$5E	94	Printer margin reset.
$5F	95	Printer margin index.
$60	96	DOS entry address.
$62	98	DOS command execute address.
$64	100	DOS command buffer pointer.
$66	102	DOS program load address.
$68	104	Output screen upper/left address.
$6A	106	Output screen reset address.
$6C	108	Output screen pointer.
$6E	110	Number of columns less one.
$6F	111	Number of rows less one.
$70	112	Row-to-row increment.
$71	113	Screen blank character.
$72	114	Entry screen upper/left address.
$74	116	Entry line address.
$76	118	Entry cursor address.
$78	120	Entry start column.
$79	121	Entry start gate.
$7A	122	End of first entry line.
$7B	123	Start of second entry line.
$7C	124	End of second entry line.
$7D	125	Number of entry screen lines.
$7E	126	Entry screen blank character.
$7F	127	Entry screen prompt character.

PAGE ZERO MEMORY MAP (CONT)

HEX	DEC	FUNCTION
$80	128	Terminal width port.
$82	130	Keyboard entry port.
$84	132	RS-232 control port.
$86	134	RS-232 data port.
$88	136	Terminal setup byte.
$89	137	Spare byte.
$8A	138	Keyboard complement byte.
$8B	139	~~Spare byte.~~ Clock adjustment.
$8C	140	RS-232 control mask.
$8D	141	RS-232 control reference.
$8E	142	~~Spare word.~~ Clock adj. for CONCUR.
$90-$9F		System spares.
$A0-$A3		Spare flags.
$A4	164	Current code token.
$A5	165	INPUT prompt save.
$A6	166	MIN,MAX argument index.
$A8	168	MIN,MAX argument locator.
$AA-$AF		Temporary storage.
$B0-$CF		Result copy for mult/divide.
$D0-$DF		Argument copy for mult/divide.
$E0-$FF		Result accumulator for mult/divide.

NOTE: NEW command resets $2A-$55.

ORIG command resets $00-$29,$56-$8F.

USER PROGRAM MEMORY MAP

HEX ADDRESS	FUNCTION
$3279	Program file start address.
$327B	Program file end address.
$327D	Program size (tracks).
$327E	Program direct link.
$3281-$32FF	Page-zero disk save area.
$3300-$33FF	Vector table for variables.
$3400-$34FF	Vector table for subroutines.
$3500-$38FF	Symbol table for program names.
$3900	Start of program code.
$61FF	End of available memory.

Addresses for v.3.1.
Version 3.2 subtract one page.
Version 3.3 add eight pages.

BETA/65 SYSTEM MEMORY MAP

HEX ADDRESS	FUNCTION
$6800-$6AFF	System buffer space.
$6B00-$6FFF	Text functions.
$7000-$71FF	System boot/loader.
$7200-$72FF	Primitive vector table.
$7300-$74FF	Primitive name table.
$7500-$7AFF	KERNEL- System kernel routines.
$7B00-$7EFF	ARITH - System arithmetic routines.
$7F00-$86BF	EXTEN - System extensions.
$86C0-$8EFF	HSED - High-speed editor routines.
$8F00-$93FF	HSIO - High-speed input/output.
$9400-$9D7F	HMED - High-memory editor.
$9D80-$9FFF	BASIC - Extensions in BASIC.

STRING FUNCTIONS

LEN	Length of string.
ASC	Convert first character to number.
VAL	Convert string to number.
CHR$	Convert number to one-char. string.
STR$	Convert number to string.
CON$	Concatenate (connect) two strings.
VS	Verify string equivalence.
KBD	Load and assign character from keyboard.
INLINE	Input user entry line as string.
DISK	"COMMAND" - Send COMMAND for DOS execution.
PROMPT	"MESSAGE" - Send MESSAGE to entry screen.

PACKED STRINGS

Space for packed strings may be reserved using

```
STRING A$(500)
```

This statement assigns 500 bytes for the string named A$. Use of ($) in string names is optional. Elements of A$ may be accessed using, e.g., A$(I), where I is between 1 and 500. Any maximum length (up to 65535) may be used, subject to memory limit.

STRING BRANCHING

VS will combine two strings to give a result equal to zero or one. If the strings match, zero is loaded. If not, one is loaded. If unequal length strings are compared, the comparison is done only to the length of the shorter string.

The branch function IFEQ will branch if the current result is an empty string.

The branch function IFNE will branch if the current result is a non-empty string.

The branch functions IFPL and IFMI will continue with string result.

STRING TRUNCATION

Strings copied into variable space are truncated to first three characters. Use STRING statememt to define storage for long strings.

STRING CONSTANTS

String constants must be enclosed by quotation marks, e.g., ".....". Internal quotations must use CTRL 2 to enter each mark, e.g., "..".."..".

Any string function will accept a string constant as an argument. Also, string constants may be used in data fields. Use the READ function to access data field strings.

For example,

```
:string_data "ABCDEFGH","1234567890"
.......
REF string_data READ A$, N$
```

will read alpha and numeric characters into A$ and N$, respectively. You must reserve space using a STRING statement to avoid truncation.

OS65/D COMMANDS

ASM	Load assembler.
BASIC	Load BASIC.
CALL MMMM=TT,S	Load track TT, sector S, into MMMM.
DIR TT	Print directory for track TT.
EM	Load Extended Monitor.
EXAM MMMM=TT	Load entire track TT into MMMM.
GO MMMM	Execute machine code from MMMM.
HOME	Reset drive head to track 00.
INIT	Initialize entire disk.
INIT TT	Initialize track TT.
IO II,OO	Change input flag to II, output to OO.
IO II	Change input flag to II.
IO ,OO	Change output flag to OO.
LOAD NAME	Load named file.
LOAD TT	Load track TT.
MEM IIII,OOOO	Point input to IIII, output from OOOO.
PUT NAME	Save named file.
PUT TT	Save file to track TT.
RE ASM	Restart assembler.
RE BAS	Restart BASIC.
RE EM	Restart Extended Monitor.
RE MON	Restart PROM Monitor.
SA TT,S=MMMM/P	Save memory from MMMM on track TT, sector S, for P pages.
SELECT X	Select drive (X=A,B,C,or D).
XQT NAME	Execute named object file.
XQT TT	Execute object file on track TT.

From BETA/65 Editor- Use !COMMAND to send command.
From BETA/65 Program- Use DISK "COMMAND".

EDITOR COMMANDS

NEW	Clears space for new program.
LIST	List program from first line.
RUN	Execute program from first line.
CONT	Cont. program interrupted by ESC.
EXIT	Exit to DOS (*GO 0000 restarts BETA).
HELP	Run program from line 32768.
FREE	Show bytes left for names and program.
FLIST	Display pre-defined functions.
VLIST	DISPLAY names used for variables.
SLIST	Display names used for subroutones.
VCH	Change variable named first to last.
SCH	Change subroutine named first to last.
PONOFF	Turn printer line on/off.
DOUBLE	Output with space between lines.
PACK	Compress output to printer.
MERGE	Merge entry into output stream.
WIDTH	Change video display width.
ORIG	Restore default output format.

DOS ACCESS Use ((!COMMAND))

ADDENDA

KEYBOARD MODS

SHFT 0	Gives assignment character (@).
CTRL 0	Gives exponent character (^).
CTRL 2	Puts quote marks into strings.
CTRL 5	Gives backslash character (\).

CONTROL CTRL numeric gives """@\{|[]^~_

SHIFT LOCK UP gives l.c. with SHIFT UP
u.c. with SHIFT DOWN

SHIFT LOCK DOWN gives u.c. with SHIFT UP
l.c. with SHIFT DOWN

NEW FUNCTIONS

DPEEK	Loads two bytes instead of one.
DPOKE	Stores lower two bytes of current result.
MIN	Use DPEEK $A8 to locate min in list.
MAX	Use DPEEK $A8 to locate max in list.
SGZ	Same as SGN but zero argument gives zero.

NOTICE

Do not use array name as FOR..NEXT index variable.
Do not use e.g., :label...GOTO label in same line.

Line references are updated at each linestart. Branches to labels or CALL to SUBR will lose reference until the next line is started. It is bad practice to put more than one statement on a line which contains a label branch target or SUBR statement.

List Properties of BETA/65 Operators

Operator Group	List Properties	Partition
PRINT HEXPR PROMPT DELAY DISK %	Multiple Expressions	Comma
TEST \ = (..) TO STEP	Single Expression	----
CLR SCR WAIT ESC STOP NEXT RET END	No List	----
READ INPUT INLINE KBD	Multiple Assignments	Comma or Space
@ (Shift Ø)	Multiple Assignments	Space
LET REF FOR	Single Assignment	----
* / + - ^ MOD QMULT QDIV MIN MAX AND OR XOR VS CON$ POKE DPOKE	Multiple Primaries	Space
NEG ABS SGN SGZ SQ SQR NOT AT TAB CHR$ STR$ ASC VAL LEN PEEK DPEEK	Single Primary	----
IFPL IFMI IFEQ IFNE GOTO	Single Branch Target	----
LINK	Multiple Link Targets	Comma
:	Data Field	Comma or Space

```
\  TEST  \\  !  PRINT  HEXPR  PROMPT  DISK
=  TO  STEP  DELAY  fH1  fH2  fH3  fH4
:  //  REF  //  LET  //  //  //
FOR  //  READ  INPUT  INLINE  KBD  GOTO  LINK

*  /  +  -  ^  MOD  VS  CON$
fD1  fD2  fD3  fD4  QMULT  QDIV  MIN  MAX
AND  OR  XOR  //  ///  ///  ///  %
POKE  DPOKE  ///  @  IFPL  IFMI  IFEQ  IFNE

(  [  <  -  NEG  ABS  SGN  SGZ
PEEK  DPEEK  SQ  LEN  ASC  CHR$  VAL  STR$
AT  TAB  NOT  SQR  fM1  fM2  fM3  fM4
DIM  `  (  )  STRING  `  ///  ///

)  ]  >  ,  `  ///  ///  ///
//  //  ;  ?  ///  ///  ///  ///
SUBR  CALL  //  //  CLR  SCR  //  VECTOR
WAIT  ESC  STOP  `  NEXT  RET  END
```