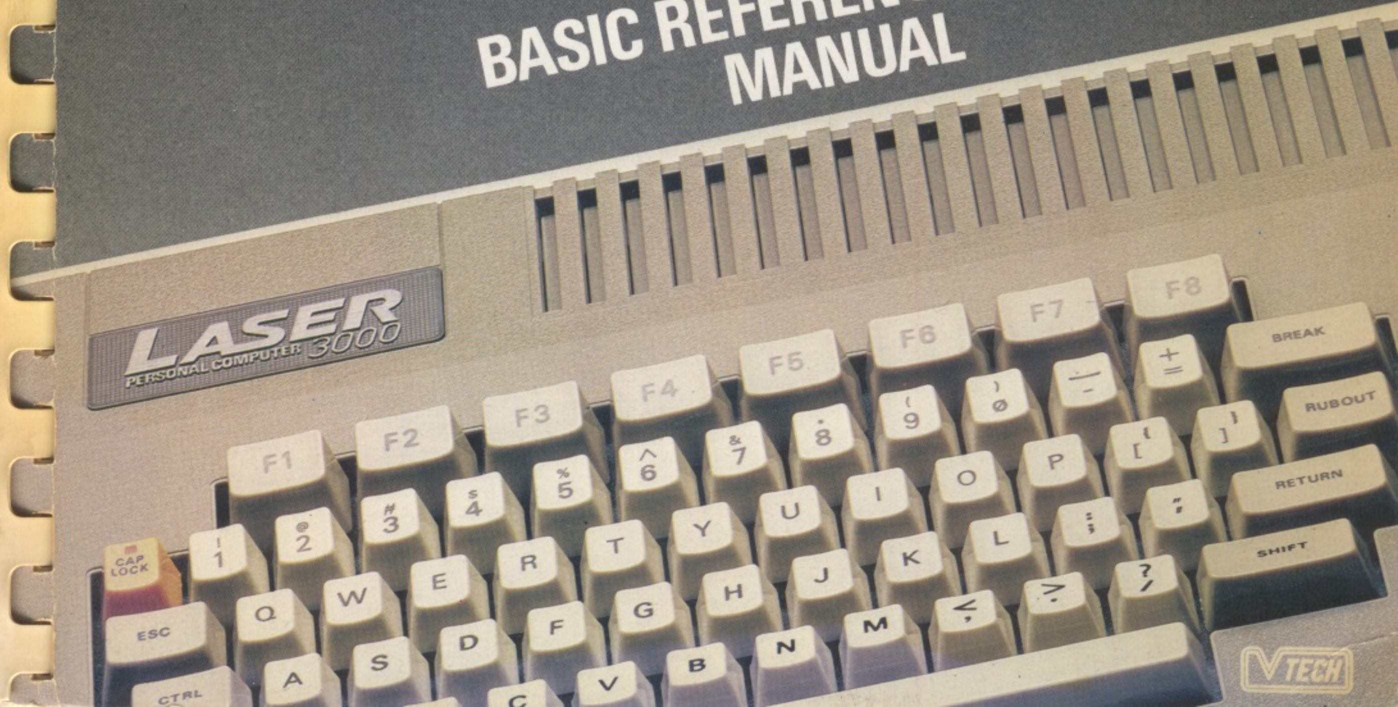
LASER™ 3000 PERSONAL COMPUTER

BASIC REFERENCE MANUAL



SECOND EDITION – 1984

CONTENTS

115	● ROT
116	● RUN
117	● SAVE
118	● SCALE
119	● SHLOAD
121	● SOUND
127	● SPC
128	● SPEED
129	● STOP
130	● STORE
131	● STR$
132	● SWAP
133	● TAB
134	● TEXT
135	● TROFF
136	● TRON
137	● USR
138	● VTAB
139	● WAIT
140	● WIDTH
141	● XDRAW

CHAPTER 1

INTRODUCTIONS

INTRODUCTION

The Laser 3000 Basic is a full implementation of the most popular microcomputer programming language. Basic is run through a program called an *Interpreter.*

This allows you to enter Basic commands and have them executed immediately.

TO START BASIC

Turn on your Laser 3000. The logo and the prompt sign (]) will appear on your display monitor. Immediately beside the prompt sign there will be a flashing cursor.

The machine is now ready for you to enter Basic commands, or a Basic program.

THIS MANUAL

The manual is divided into four sections:

- Some background information on Basic programming – this chapter describes how the Laser 3000 handles its implementation of Basic, as well as giving you general information on the strengths and limitations of the language.
- Laser 3000 Basic Statements – this chapter presents all the statements and commands used in Basic. It is arranged in alphabetical order.
- Laser 3000 Basic Functions – this presents all of the Laser 3000's built-in Basic functions, and it is also arranged alphabetically.
- Appendices and Index – these contain the ASCII and keyboard character codes, error messages, and list of reserved words.

This Basic manual completely describes the language as it is implemented on the Laser 3000 computer.

However, it is not intended as a guide to learning the language, although if you followed through the descriptions of the commands and functions a number of times you would get a fair grasp of it.

If you are completely new to Basic, there are a large number of books available to help you learn it. Among them are:

BASIC from the ground up, by David E. Simon, Hayden, 1978.

BASIC, by Robert L. Albrecht, Leroy Finkel, and Jerry Brown, John Wiley & Sons, 1973.

VARIABLES

Variables are the names you assign to values that change in your Basic program.

The values can be given directly — initialised — as in this example:

```
A = 63.999
```

Or they can take up new values as a result of the program's execution. For instance, in the next example, the value of I varies from 1 to 10.

```
10   For I = TO 10

20   PRINT I

30   NEXT
```

When the Laser 3000 Basic starts running, all variables that have no explicitly assigned values (as in the first example) are assumed to be zero.

VARIABLE NAMES

Laser 3000 Basic variable names must start with an alphabetic letter. They can be up to 40 characters long, and can represent either numbers or strings.

The variable names cannot be reserved words – for a list of reserved words see the Appendices – nor can they have reserved words embedded in them. For instance: **ADIMMY** contains the reserved word **DIM**, and is not an allowable variable; and **DIMMY** starts with the reserved word, and is not permissible.

Reserved words include all the Basic commands, statements, functions, and operator names.

Integer varibales are denoted by a percentage sign (%) immediately following the variable names. This type of variable can only contain integer values, for example:

```
I%=1234
```

String variables must always end with the dollar sign ($). This declares to the Basic interpreter that it is dealing with a string variables, and it allocates extra memory to handle it. For example:

SENTENCE$ is a valid string variables, but SENTENCE – without the $ sign at the end – is not.

ARRAY VARIABLES

An array is a matrix or table of values that is referenced by the same variable name. The specific values are accessed by subscripts which are used in conjunction with the array's name.

The number of subscripts for an array is the same as the number of dimensions for it, and are defined in the DIM statement. For instance:

DIM AARRAY (10, 10, 10) sets up a three-dimensional array where the subscripts for each dimension range from 0 to 10. Thus it is equivalent to a table containing 11x11x11 or 1331 values.

DIM BARRAY (9) sets up a one-dimensional array with single subscript which can range from 0 to 9. Thus it equivalent to a table of 10 values.

EXPRESSIONS AND OPERATORS

An expression can be a constant, whether numeric or string, or a variable, or a combination of variables, constants, and operators which work together to produce a single value. There are four types of operator:

- arithmetic
- logical
- relational
- functional

ARITHMETIC OPERATORS

These are the common mathematical operators, and they are always performed in a set order of preference. They are listed below in this order:

Operator	*Operation*	*Example*
^	Exponentiation	A ^ B
–	Negation	-A
*,/	Multiplication, division	A*B A/B
+,–	Addition, subtraction	A+B A-B

This order of operations can be changed by using parentheses, as the expressions within parentheses are evaluated first.

Within parentheses, the above order is kept to. Some examples of how this is done follows:

Algebraic expression	*Basic expression*	*Result*
2+10 ÷ 2	2+10/2	7
(2+10) ÷ 2	(2+10)/2	6

LOGICAL OPERATORS

These operators work on values according to their logical states to proguce a result which is either one (1) or zero (0) — "true" or "false". A non-zero value corresponds to a "true" state, while a zero value corresponds to a "false" state. The outcome of a logical operation is as shown in the table following. The operators are listed in order of precedence.

NOT

A	NOT A
1	0
0	1

AND

A	B	A AND B
1	1	1
1	0	0
0	1	0
0	0	0

OR

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

RELATIONAL OPERATORS

These operators share some similarity with the logical operators, in that their result is only ever one of two things - zero (0) or positive one (+1) - "false" or "true."

Like the logical operators, these relational operators can be used to make decisions regarding program branching.

Operator	*Relation*	*Example*
=	Equality	A=B
<>	Inequality	A<>B
<	Less than	A< B
>	Greater than	A > B
<=	Less than or equal to	A<= B
> =	Greater than or equal to	A>= B

The equal sign (=) is also used to assign a value to variable (see LET statement in Chapter 3 of this manual.)

If arithmetic, relational, and logical operators are combined in one expression, the order of precedence of evaluation is: arithmetic, then relational, then logical.

FUNCTIONAL OPERATORS

The Laser 3000 has a number of built-in or "intrinsic" functions which may be used in either direct or indirect mode. These are described in Chapter 3.

Examples are TAN (A) , which calculates the tangent of angle A, and LEFT$(X$,3), which returns the three leftmost characters of string X$.

You can define your own functions using the DEF FN command. (see DEF FN command in Chapter 3)

STRING OPERATIONS

Two strings can be compared using the relational operators. These work by comparing the numeric ASCII values of each corresponding character of each string.

The conditions for equality or inequality depend on whether the ASCII codes are higher or lower. Also, a short string is relatively less than a long string.

Example:

```
10 IF "Laser" >"Resal" GOTO 30
20 PRINT "Laser"
30 END
RUN
Laser
```

Two strings can be combined - concatenated - using the plus (+) operator.

Example:

```
50 X$ = "Laser  "
60 Y$ = "3000"
70 PRINT X$ + Y$
RUN
Laser 3000
```

CHAPTER 2

SOME BACKGROUND TO LASER 3000 BASIC PROGRAMMING

SOME BACKGROUND TO LASER 3000 BASIC PROGRAMMING

When Basic is started, it displays the prompt "] ". This means that it is ready to accept commands from the keyboard.

At this command level, it can be used in either of two modes:

direct —	which is when you enter a command and have it executed immediately.
indirect —	which is when command lines are started with line numbers, and a program is built up for later execution. Programs are started by entering the RUN command.

LINE FORMAT

In direct mode, the commands and functions are entered as laid out in this manual. See the two chapters after this one for these formats.

In entering a program, the line is laid out like this:

nnnnn *Basic statement (:Basic statement . . .)* Where nnnnn is the line number.

The parentheses indicate options. The length of your line is limited to 239 characters, and a line is always finished when you hit RETURN.

Hitting RETURN adds a non-printing carriage return character at the end of a line. The basic interpreter takes this carriage return as indicating the end of a program line.

The line numbers must be in the range of 0 to 63999. They relate to the order in which a Basic program is stored in memory, and the interpreter always executes a program in the sequence of the line numbers (unless the program branches otherwise.)

CONSTANTS

As their name implies, these are values that do not change. In Basic they can be either numeric or string values. Some string constants are:

"$64,000"

"May the Force be with you"

There are two types of numeric constant.

1. Integer constants
   Whole numbers in the range from -32767 to +32767.

2. Floating point constants
   Positive or negative numbers that are represented in exponential form. These are made up of three parts: the fixed point part, in decimal form; the E which signifies exponentiation and the exponent, which must be an integer. The range of values for floating point constants is from 1E-38 to 8.5E + 37.

Example:

```
256.1024E-7 = .00002561024
4096E7 = 40960000000
```

USING A PRINTER WITH THE LASER 3000

The Laser 3000 has a built-in printer interface and a ROM-based printer driver.

You can use the Laser 3000 with a printer which has a standard Centronics interface port to print out textual material.

If you have a Epson type dot - matrix graphics printer, you can also get hard copy of the graphical displays.

TEXT

Steps for using the printer:

1 Connect the interface cable between the Laser - 3000 and the printer

2 Switch on the Laser 3000 before switching on the printer.

3 Initialise the printer by typing

- PR#1 if you are in Basic
- 1 CTRL-P if you are in the Kernel

4 From now on, any character displayed on the screen will be printed out by the printer.

5 To stop this, type

- PR#0 if you are in Basic
- 0 CTRL-P if you are in the Kernel

GRAPHICS

Any of the 6 graphic pages of the Laser 3000 – from HGR1 to HGR6, – can be printed out to a Epson - type dot - matrix printer by using the Basic command - PRINT SCREEN.

There are two ways of telling the computer which graphic page is to be printed:

- One is to use a Basic command to refer to this page before using PRINT SCREEN.

Example:

```
10    REM REFER TO WHICH PAGE
20    HGR
30    HCOLOR = 7
40    REM DRAW A CIRCLE IN THIS PAGE
50    DRAW SCIRCLE (140, 96), 50
60    REM PRINT OUT THIS PAGE
70    PRINT SCREEN
80    END
```

Note that PRINT SCREEN command will automatically select and unselect your printer. You do not have to type PR # 1 and PR # 0.

- The second method which can be used to tell PRINT SCREEN which graphic page to print out is by Poking a value into a zero page location before using this command.

The address of this zero page location is $E6 (# 230)

LOCATION	DATA					
(# 230) ($E6)	# 32 $20	#64 $40	#34 $22	#66 $42	#33 $21	#65 $41
page to be printed	HGR1	HGR2	HGR3	HGR4	HGR5	HGR6

(Note: # implies a decimal value, and $ implies a hexdecimal value.)

Example:

```
5 REM    LOAD A PICTURE INTO HGR1'S MEMORY
10       PRINT CHR$(4); ''BLOAD PICTURE, A$2000''
15 REM   TELL 'PRINT SCREEN' TO PRINT HGR1
20       POKE 230, 32
25 REM   PRINT HGR1
30       PRINT SCREEN
40       END
```

OTHER FEATURES

If you wish to obtain more sophisticated graphics print-outs, the command PRINT SCREEN may not be adequate, and you must perform some tricks. However, they are quite simple.

The procedures are:

1 Initialise the printer by typing.

- PR#1 — if you are in Basic
- 1 CTRL-P — if you are in the Kernel

2 Get your graphics ready either by drawing it now or by loading a binary image from disk to the corresponding graphic memory.

3 Poke two locations in order to get the desired page and effects – more will be mentioned later.

4 Type CTRL-Q to start printing out the graphics.

5 After the picture has been output, disable your printer by typing.

- PR#0 — if you are in Basic
- 0 CTRL-P — if you are in the Kernel

Note: The sequcence of steps 1 and 2 is not important.

The two locations are:

	Hexadecimal	Decimal
	$6F9	(# 1785)
	$779	(# 1913)

1785 ($6F9) – Poke values into this location to select different graphics-modes

HGR page	HGR1, 2	HGR3, 4	HGR5, 6
Poke 1785 ($6F9) with	00($00)	2($02)	1($01)

1913 ($779) – each bit of this one-byte location, when set (=1), is used to select the many features available for printing graphics.

bit	6	5	4	3	2	1	0
Select option	enlarge printing	Inverse printing	EOR page 1&2	OR page 1&2	AND page 1&2	print page 2	print page 1

NOTE: 1. Enlarge printing is available only in low resolution and bit image graphics.

2. When both bit 0 and bit 1 are set to 1, the primary page will be printed on the left half and the secondary page will be printed on the right half of the paper.

3. Bit 7 must be set to zero.

Four examples on the use of these locations are:

Example

---

1. PRINT HGR3 in inverse mode
   (a) Poke 1785, 2 *to select HGR3,4*
   (b) Poke 1913, 33

```
Because: ($779)      0 0 1 0 0 0 0 1
       = ( # 1913)       ↑         ↑
                      inverse    Page 1 of 560 X
       =  $21                    192 GR mode
       =  #33                    HGR3.
```

---

2. PRINT HGR2 in inverse and enlarged mode.
   (a) Poke 1785, 0 *to select HGR1,2*
   (b) Poke 1913, 98

```
Because: ($779)      0 1 1 0 0 0 1 0
       = (#1913)       ↑ ↑         ↑
                  enlarge |     page 2 of 280 X 192
       =  $62             |     compatible graphics
       =  # 98         inverse  mode = HGR2
```

---

3. PRINT HGR1 ORed with HGR2 in enlarged mode.
   (a) Poke 1785, 0 *to select HGR1, or 2*
   (b) Poke 1913, 73

```
Because: ($779)      0 1 0 0 1 0 0 1
       = (#1913)       ↑     ↑     ↑
                  enlarge    |   For AND. OR. EOR
       =  $49                |   Functions the re-
       =  #73               OR   sulting picture will
                                 be in page 1.
```

---

4. PRINT HGR3 AND HGR4 in inverse mode.
   (a) Poke 1785,   2   *to select HGR3, or 4*
   (b) Poke 1913, 37

$$\begin{array}{llll} \text{Because:} & (\$779) & 0\;0\;\underset{\substack{\uparrow \\ \text{inverse}}}{1}\;0\;0\;\underset{\substack{\uparrow \\ \text{AND}}}{1}\;0\;\underset{\substack{\uparrow \\ \text{same reason as in} \\ \text{example 3.}}}{1} \\ & = (\#1913) & \\ & = \$25 & \\ & = \#37 & \end{array}$$

CHAPTER 3

LASER 3000 BASIC COMMANDS AND STATEMENTS

LASER 3000 BASIC COMMANDS AND STATEMENTS

All of the Laser 3000's Basic commands and statements are given in this chapter, with each laid out as follows:

Purpose: Tells what the command or statement is used for.

Format: Shows the correct layout for the command or statement. You will be able to follow the layout if you keep the following rules in mind:

1. Words given in capital letters must be input exactly as shown.
2. You must enter any items given in lower case *italic letters*.
3. Items indicated in square brackets are optional [optional] .
4. Items followed by three periods...mean that the particular item may be repeated as often as you like.
5. Quotation marks, commas, full-stops, hyphens, semicolons, and equal signs must be used as indicated.

Comments: Describes the circumstances in which the command is used.

Examples: Gives sample programs or program sections in which the command or statement is used.

AMPERSAND COMMAND (&)

Purpose: To jump into a machine language command starting at hex location $3F5.

Format: &

Comments: A machine language subroutine must be placed at $3F5 before using this command, otherwise an unexpected result may occur, which might even destroy your program.

Example:

```
CALL -151
3F5 :  4C 00 C3
CTRL - C
&
]
```

You enter the system kernel and place a JUMP machine instruction at location $3F5. Executing the AMPERSAND COMMAND will direct control to address $C300 where the 80 column display firmware is located.

CALL

Purpose: To use an assembly language subroutine.

Format: CALL *expression*

Comments: This statement is one means of transferring program flow to an assembly language subroutine.

expression is the entry address of the machine language routine, and it must be in decimal.

Example:

```
300 ASSEM = 64600
310 CALL ASSEM

RUN
```

This **CALL** returns control to the **ROM**-based Laser 3000 kernel, which then clears the screen, and displays the prompt in the **HOME** position.

CHR$

Purpose: Converts an ASCII code to its equivalent character.

Format: CHR$ (*n*)

Comments: This operation returns the single character corresponding to the number *n,* which must be between 0 and 255.

The ASCII characters codes are listed in Appendix IV.

Examples: PRINT CHR$ (81).

The following example would print all the upper case letters of the alphabet (ASCII codes 65 through 90).

```
10 FOR I = 65 TO 90
20 PRINT  CHR$(I);
30 NEXT I
```

CLR

Purpose:	To clear all variables, arrays and strings
Format:	CLR
Comments:	All variables will be cleared to zero.
Example:	CLR

CONT

Purpose: To restart a program running again after it has been halted.

Format: CONT

Comments: The program resumes at the next instruction after the break occurred.

CONT is often used in debugging a program, in conjunction with STOP. Once the program has been halted, you can examine and change variable in the program, and then use CONT to resume it. CONT may not work if you change the program while it is halted.

Examples: See CONT used in the example for the STOP statment.

DATA

Purpose: To store constant numbers and string values in your program so they can be used in conjunction with the READ statement.

Format: DATA *constant* (*,constant*)...

Comments: DATA statements are non-executable and may be placed anywhere in the program.

No numeric or string expressions can be used in the DATA statement. The constant may be a number or a string. There is no need to enclose a string with quotation marks (''), but any spaces in between are ignored.

The numeric constants may be in any format, i.e., fixed point, floating point or integer.

The variable type given in the READ statement must agree with the corresponding constant in the DATA statement.

Examples: See examples for the READ statement.

DEF FN

Purpose: To define and name a function that is written by the user.

Format: DEF FN *name (real variable) = expression.*

Comments: The *name* is exactly the same as a variable name. A user defined string function is not allowed. The *real variable* is the variable that will be used when the function is evaluated.

The *expression* can be as long as a line (239 characters long).

If you need to program functions that require more room than that, you should implement your function as a subroutine.

Examples:

```
10 GEE = 9.8
20 DEF FNDIS (T) = GEE *T^2/2
30 INPUT "Time?"; T
40 PRINT "Distance is" ; FNDIS (T)
```

This would calculate the distance that a body has fallen after T seconds, using the function DIS which is derived from the formula $s=\frac{1}{2} gt^2$, where s is distance, g the acceleration due to gravity, and t the time that has elapsed since the object was dropped from a stationary position.

DEL

Purpose:	Removes program lines.
Format:	DEL *line number 1, line number 2*
Comments:	This deletes the lines from *line number 1* to *line 2,* inclusive.
Example:	DEL 10, 110
	This removes all the lines between 10 and 110, including lines 10 and 110.

DIM

Purpose: This sets the maximum subcripts for a variable and allocates enough storage to accomodate them.

Format: DIM *variable (subscripts) [, variable (subscripts), . . .]*

Comments: When the Laser 3000 Basic interpreter encounters a DIM statement, it initialises all the elements of the array to zero, if it is a numeric array.

For a string array, all elements are initially null strings (i.e. empty strings). However the length of each element can be different as a result of program execution.

If an array is used in a Basic program without a corresponding **DIM** statement, the interpreter assumes the value of the subscript to be 10.

The maximum number of dimensions and maximum number of elements in each dimension depend on the amount of free memory in the system.

Examples:

```
10  DIM A (10, 10)
20  FOR I = 1 TO 10
30  FOR J = 1 TO 10
40  IF I = J THEN A (I, J) = 1
50  PRINT A (I, J); "   ";
60  NEXT J
70  PRINT
80  NEXT I
90  END
```

This will build up an array whose diagonal elements are all ones, with the rest of the elements remaining zero.

DRAW

Purpose: To draw geometric shapes

Format: DRAW *shape* *(shapeparameters, ...)*

Comments: This command can be used to draw the following *shapes,* whether solid (S) – coloured in; or hollow (H) – in outline:

. CIRCLES – SCIRCLE and HCIRCLE

. SQUARE – SSQUARE and HSQUARE

The *shapeparameters* for SCIRCLE and HCIRCLE are:

DRAW SCIRCLE *(x,y), r [, c, sr, er]* – where *x* is the x co-ordinate of the graphics screen, *y* is the y co-ordinate of the screen, *r* is the length of the major axis of the ellipse to be drawn, *c* is the circularity of the ellipse and *c* must be in the range 0 to 1. When *c* =1 a circle is drawn, and *sr* is the start radian, and *er* is the end radian and must be greater than *sr.* If *c, sr* and *er* are not specified, the default values are *c* =1, *sr* =0, *er* =2π. Also note that the drawn ellipse can be orientated in any direction by using the ROT command before the DRAW SCIRCLE or DRAW HCIRCLE command. The same *shapeparameters* apply to HCIRCLE.

The final three are optional. If they are included, an ellipse is drawn, its flatness being determined by the circularity *shapeparameter c.*

In low resolution graphics and bit image graphics modes, the *x* co-ordinate can range between 0 and 279, whereas in double resolution graphics, *x* ranges between 0 and 559 while the *y* co-ordinate can range between 0 and 191 in all graphics modes.

The *shapeparameters* for **SSQUARE** and **HSQUARE** are:

DRAW HSQUARE *(X1, Y1 TO X2, Y2 [TO X3, Y3 . . .])* where each pair of co-ordinates define the diagonal corners of the square (the Laser 3000 works out where the third and fourth corners should go).

The *x* and *y* co-ordinates can have the same values as in DRAW SCIRCLE and DRAW HCIRCLE command. Exactly the same *shape-parameters* apply to DRAW SSQUARE.

Example:

```
10    HGR5
20    HCOLOR = 3
30    DRAW SCIRCLE (140, 96), 50
40    END
```

The final three are optional. If they are included, an ellipse is drawn, its flatness being determined by the circularity *shapeparameter c.*

```
10    HGR5
20    HCOLOR = 4
30    DRAW SSQUARE (0, 0  TO 10, 10)
40    END
```

This draws a solid yellow coloured square with the point (0, 0) diagonally opposite to the point (10, 10).

Drawing shapes

In any of the Laser 3000's graphics modes, you can draw and move around free-form shapes.

The general graphics commands on the Laser 3000 i.e. HPLOT, DRAW only give static shapes. With shapes you define yourself, you can animate your creations, either moving, rotating, or changing their sizes.

Setting up the shapes

The first step is to sketch on paper the shape you want, and then break this down into a series of directed lines (i.e. vectors). For instance, a rectangle could be broken down like this:

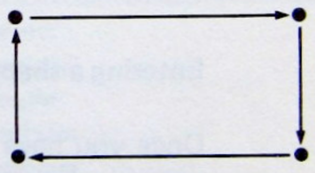


Figure 1. Vectors for a rectangle

As the vectors can only point to either the left or right or up and down, diagonal lines must be approximated by a number of them which, taken together, give the impression of a diagonal line. This is shown below:

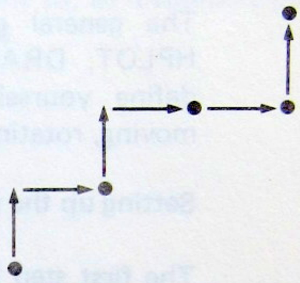


Figure 2. Vectors for a diagonal line

Entering a shape table

Once you have defined your shape, and broken it down into vectors, the next step is to convert the vectors into binary codes so that your Laser can accept them and reproduce the shape on the display later.

Two types of vector are possible: 1. move and plot; and 2. move but do not plot.

For each of these two basic types there are four directions: up, down, left, and right. In all there are eight shape vectors, and they have the following three-bit binary codes:

000	Move Up
001	Move right
010	move down
011	move left
100	move up and plot
101	move right and plot
110	move down and plot
111	move left and plot

For instance, the diagonal line can be represented as follows, starting from the left:

100	move up and plot
101	move right and plot
100	move up and plot
101	move right and plot
101	move right and plot
101	move up and plot

The shape table in the Laser's memory is made up of separate bytes, which means that only two complete vectors - of three bits each - and an incomplete vector - of only two binary bits - can be stored in each byte, as there are only 8 bits within a single byte.

These incomplete vectors are movement without plotting, and they are the only ones possible in this part of the shape table byte. As this is the case, unless you can arrange your shape such that the non-plotting vectors are the very third vector in it, you should set these two bits to zero (i.e. unused).

To create a triangle, your shape definition will look something like:

3rd VECTOR (UNUSED)	2nd VECTOR	1st VECTOR	HEX. DATA	COMMENT
00	101	100	2C	move up and right with plot
00	101	100	2C	move up and right with plot
00	101	100	2C	move up and right with plot
00	101	100	2C	move up and right with plot
00	101	110	2E	move down and right with plot
00	101	110	2E	move down and right with plot
00	101	110	2E	move down and right with plot
00	111	110	3E	move down and left with plot
00	111	111	3F	move left and left with plot
00	111	111	3F	move left and left with plot
00	111	111	3F	move left and left with plot

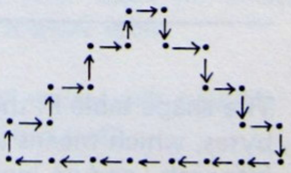


Figure 3. Vectors for a triangle

THE SHAPE TABLE

In the previous pages, you have learned how to create a single shape definition as a whole shape table.

But, in fact, a shape table can consist of more than one shape definitions so that more than one shape can be manipulated by using the DRAW, XDRAW, ROT and SCALE commands.

Figure 3 shows the general format of a shape table. You can see that the first few bytes of the shape table are used to tell the Laser how many shape definitions are within the shape table, and where these shape definitions are, relative to the starting address of the shape table. The last byte of your shape definition must be zero to signify the end of the shape table.

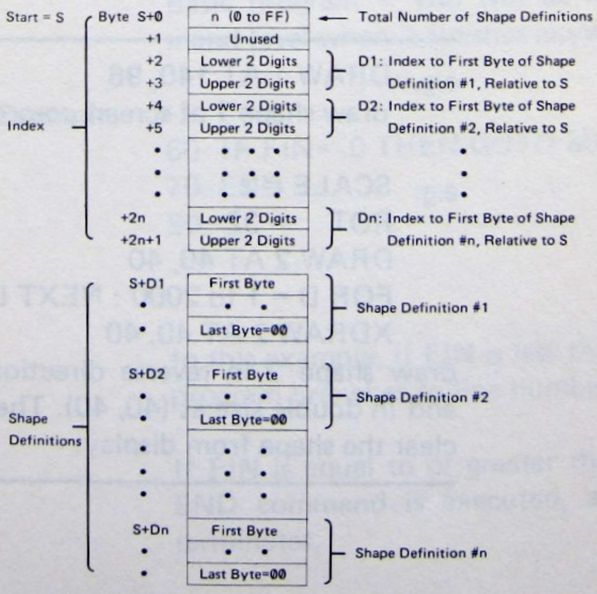


Figure 3 General Format of a Shape Table.

BEFORE USING DRAW, XDRAW, ROT AND SCALE

Before you can use any one of the following commands: DRAW, XDRAW, ROT and SCALE, make sure you have done the followings:

1 Entered the shape table
2 Told the Laser where the shape table is

Item 1 has been discussed in the previous pages. Item 2 is a very simple task; just enter the starting address of the shape table into hex location $E8 (lower two digits) and $E9 (upper two digits).

For example, if your shape table resides from address $1000 on, you can enter the Laser's kernel (CALL-151) and type the following E8:00 10.
Type **CTRL–C** and **RETURN** to go back to Basic, Laser 3000 is now ready to interpret your shape commands.

---

e.g.
```
DRAW 1 AT 140, 96
```
draw shape 1 at screen co-ordinates (140, 96)

e.g.
```
SCALE = 2
ROT   = 32
DRAW 2 AT 40, 40
FOR D = 1 to 2000 : NEXT D
XDRAW 2 AT 40, 40
```
draw shape 2 in reverse direction (i.e. rotated 180°) and in double size at (40, 40). Then wait for a while and clear the shape from display.

END

Purpose: Finishes program excution and returns you to command level.

Format: **END**

Comments: This command may be placed anywhere in your program — though it may not be all that useful as the first statement of your program.

END is the most orderly way to stop your Basic program when it has done what you require.

This is because, unlike the similar command **STOP**, it does not cause a **BREAK** message to be displayed.

However, **END** is not essential at the end of a Basic program — you will be returned to command level when it finishes anyway.

Example:

```
⋮
60  IF FIN<0 THEN GOTO 80
70  END
80
⋮
```

In this example, if **FIN** is less than zero, then the program branches to line number 80.

If **FIN** is equal to or greater than zero, then the **END** command is executed, and the program terminates.

FLASH

Pupose:	To cause all computer messages to alternate between character and background colour.
Format:	**FLASH**
Comments:	**FLASH** causes the display to alternate between **NORMAL** mode and **INVERSE** display mode.

FOR...NEXT

Purpose: Loops around a group of instructions a specified number of times.

Format: **FOR** *variable=n* **TO** *m* [**STEP** *i*]
NEXT [*variable*] [,*variable*]. . .

Comments: The *variable* — which is optional with the **NEXT** — acts as a counter for the number of times the instructions within the loop surrounded by the **FOR** and **NEXT** are executed.

n is the initial value of the counter, *m* is the final value of the counter, and *i* is the step or increment.

All the instructions in the loop are executed down to the **NEXT**.

Then the counter is incremented by *i*. (If you do not give a value for *i*, the Laser 3000 Basic interpreter assumes *i* is one.)

Then a check to see whether the value of the counter is greater than *m*. If it is not, the loop is gone through again.

If it is greater than *m*, then the program continues with the instructions that follow the **NEXT**.

The value of *i* can also be negative, in which case it is as though *m* and *n* are exchanged from their positive roles.

In other words, *n* is greater than *m*, and the counter is reduced each time through the loop until it is less than *m*.

FOR...NEXT loops can be written inside each other, or *nested.* In these cases, the *variable* names must be different, and each **FOR** must be matched with its corresponding **NEXT**.

If the *variable* for the **NEXT** is not given, the interpreter assumes that the **NEXT** refers to the **FOR**...directly above it.

Alternatively, one **NEXT** can serve a number of **FOR**s, when it is given as **NEXT** *variable1, variable2, variable3* etc.

Examples:

```
10 FOR N=2 TO 100 STEP 2
20 PRINT N/2
30 NEXT
```

This would print out the numbers from one to 50

```
100 FOR N=100 TO 2 STEP -2
110 PRINT N/2
120 NEXT
```

This would also print out the numbers between one and 50, but in reverse order to the first example.

```
200 FOR K=1 TO 2
210 FOR L=1 TO 5
220 PRINT K*L;" ";
230 NEXT L, K
```

This would print out the numbers:

```
1 2 3 4 5 2 4 6 8 10
```

GET

Purpose: Reads a character from the keyboard without echoing it on the screen. No carriage return is necessary.

Format: GET *variable*

Comments: The *variable* may be a string or arithmetic variable.

When the program expects an arithmetic variable and an non-numeric key is pressed, the "Syntax error" message will result.

Example:

```
10 GET A$
20 C$ = C$ + A$
30 PRINT C$
40 GOTO 10
```

GOSUB...RETURN

Purpose: To direct the program flow into, and to return from, a subroutine.

Format:

```
GOSUB linenumber
.
.
.
RETURN
```

Comments: A subroutine may be called any number of times from within a program, and it is possible to call another subroutine from within a subroutine, which, in turn, may call another subroutine. Nesting of subroutines can be 25 levels deep. The *linenumber* needed in the GOSUB statement is the first line of the subroutine.

The RETURN statement terminates the execution of the subroutine, and returns the interpreter to the line **immediately** following the most recent GOSUB statement.

However, there is no way that the interpreter can distinguish between a subroutine and ordinary program lines. So, to avoid executing the subroutine when it is not required, you should put a GOTO, STOP, or END in the line before it starts.

Example:

```
] 10 INPUT A
] 20 GOSUB 50
] 30 PRINT A
] 40 END
] 50 IF A < 100 THEN 80
] 60 A=A+50
] 70 RETURN
] 80 A=A+200
] 90 RETURN
] RUN
? 40
240
] RUN
? 170
220
```

GOTO

Purpose: To direct the program flow to another part of the Basic program.

Format: GOTO *linenumber*

Comments: GOTO takes your Basic program out of its normal sequence – one line following the other – and continue execution at a point either many lines ahead, or many lines behind the line containing the GOTO.

If the line the GOTO refers to is a REMark or DATA line – which is not executable, then the instruction executed is the next executable line after *linenumber*.

GOTO can be very handy in debugging programs. You can use it in direct mode to enter a program at a certain point, rather than having the program run through from its beginning.

Example:

```
] 10 INPUT A$
] 20 B$=B$+A$
] 30 PRINT B$
] 40 GOTO 10
] RUN
? T
T
?H
TH
? I
THI
? S
THIS
?
```

HCOLOR

Purpose: To set the colour of subsequently plotted graphics.

Format: HCOLOR = *colour code*

Comments: For low resolution and double resolution graphics, the colours given by *colour code* are as follows:

Code	Colour
0	black
1	green
2	magenta
3	white
4	black
5	red
6	blue
7	white

For bit image graphics, the *colour codes* become:

Code	Colour
0	black
1	green
2	magenta
3	cyan
4	yellow
5	red
6	blue
7	white

Example:

```
]10    HGR5
]20    FOR I = 0 TO 279
]30    HCOLOR = I/40 + 1
]40    HPLOT I,0 TO I, 191
]50    NEXT I
```

Running this program and you will see 7 coloured vertical bars.

HGR	HGR1	HGR2
HGR3	HGR4	HGR5
HGR6		

Purpose: To set up the graphics modes.

Comments: HGR sets up the Laser 3000's mixed text and low resolution grahics mode, which has a resolution of 280 pixels by 160 pixels and four lines of text at the bottom of the screen.

The command will clear the screen, displays the primary graphic page. The cursor will be placed just under the graphics screen, i.e. the third line from the last on the text screen.

HGR1 and HGR2 have much the same effect, except that the resolution becomes 280 by 192. Text display is not available in this mode. HGR1 displays the primary page and HGR2 displays the secondary page.

HGR3 and HGR4 are the double resolution graphics set–up commands. The resolution in this mode is 560 by 192. HGR3 displays the primary page of this mode, and HGR4 displays the secondary page.

The final graphics set-up commands are HGR5 and HGR6, which allows you to use the Laser 3000's 280 by 192 bit-image graphics mode. HGR5 displays the primary page and HGR6 displays the secondary page.

HIMEM

Purpose: To set the highest memory location available to a Basic program.

Format: HIMEM: *address*

Comments: This command is used to protect the area of memory above a program for data or machine language routines.

The *address* must be in the range −65535 to 65535.

HOME

Purpose: To clear screen and position the cursor at the upper left corner of the display screen.

Format: HOME

Comments: Characters outside the display window will not be cleared.

Example:

```
] HOME
```

All characters in the display window will be cleared. The cursor returns to the home position. Characters beyond the display window remain unchanged.

HPLOT

Purpose: To draw either lines or dots.

Format:

HPLOT *x1, y1*
HPLOT TO *x1, y1*
HPLOT *x1, y1* TO *x2, y2* [, TO *x3, y3* . . . . .]

Comments: The first form of this command causes a dot to be plotted at the position given by *x*1, *y*1 co-ordinates.

The second form causes a line to be drawn from a previously specified plotted dot to the position given by the *x*1, *y*1 co-ordinates.

The third form of HPLOT draws lines from point to point as given by the pairs of *(x,y)*

x ranges from 0 to 279 in both low resolution and bit image graphic modes and 0 to 559 in double resolution graghics mode. *y* ranges from 0 to 191 for all graphic modes.

Example:

```
10  HGR3
20  HCOLOR = 1
30  HPLOT 0,0 TO 559,191
```

This program will plot a green line from the top left-hand corner to the bottom right-hand corner of the screen.

HTAB

Purpose: To move the cursor a given number of places to the right of the left margin.

Format: HTAB *(displacement)*

Comments: *displacement* ranges from 1 to 255. If *displacement* is greater than the display window width, then the cursor simply wraps round to the left-most of the same line.

Example:

```
10   HOME
20   HTAB (20)
30   PRINT "20 HORIZONTAL DISPLACE-
     MENTS"
```

IF...GOTO and IF...THEN...

Purpose:	To direct program flow depending on the result of an evaluation.

Format:	IF *expression* GOTO *linenumber*
	IF *expression* THEN *statement*

Comments:	If the expression is true, the statement following GOTO or THEN is executed, otherwise it is ignored and the program continues with the next line.

Examples:

```
] NEW
] 10 INPUT A,B
] 20 IF A<B GOTO 50
] 30 PRINT A;" IS LARGER THAN "; B
] 40 GOTO 10
] 50 PRINT A;" IS SMALLER THAN"; B
] 60 GOTO 10
] RUN
? 37,22
37 IS LARGER THAN 22
? 40, 90
40 IS SMALLER THAN 90
```

In statement 20, A is compared with B. If A is smaller than B, statement 50 will be executed; otherwise program continues to statement 30.

```
] NEW
] 10 INPUT A
] 20 IF A>B THEN B=A
] 30 PRINT B; "IS THE LARGEST"
] 40 GOTO 10
] RUN
? 37
37 IS THE LARGEST
? 40
40 IS THE LARGEST
```

The above program will print out the largest number so far entered.

IN#

Purpose: To accept input from a selected input device.

Format: IN# *device no*

Comments: The number given in *device no* must be between 0 and 8. This number determines which device your Laser 3000 will expect input from.

Example: IN# 0

This command changes input from a peripheral device to the keyboard.

IN# 8

This command enables you to redefine the function keys as follows:

1. IN# 8 RETURN
2. press the function key
3. new key sequence
4. ESC ESC

INPUT

Purpose: Allows you to enter values from the keyboard while a program is executing.

Format: INPUT *[''prompt'';]* *variable 1 [, variable 2 . . .]*

Comments: When the Basic interpreter comes across an INPUT statement it displays either the *''prompt string''*, or it just displays a question mark if the *''prompt string''* has not been included in the statement. Only one prompt string is allowed and it must appear immediately after INPUT.

Example:

```
] 10  INPUT   ''A = '';A
] 20  INPUT   B
] 30  PRINT ''A = '';A ; '' B = '';B

] RUN
A = 10
? = 20

A = 10 B = 20
]
```

INVERSE

Purpose: To reverse the character and background colour of all characters displayed.

Format: INVERSE

Comments: If a TEXT *character, background, border* command was executed, INVERSE sets all computer message to have *background* colour with *character* background. otherwise, all computer message will have dark characters on a white background.

Example:

```
]    TEXT RED, GREEN, BLUE
]    INVERSE
```

All characters will be in green formed by a red rectangle.

LEFT$

Purpose: Returns a specified number of characters from the left-hand side of a character string.

Format: LEFT$(*string$,n*)

Comments: The number *n* must be between 1 and 255, and if it is greater than the length of *string$,* then the LEFT$ function will return the entire string *string$* to the program.
LEFT$ works similarly to the RIGHT$ and MID$ string functions.

Examples:

```
] 10 A$= "LASER 3000"
] 20 B$= "IS"
] 30 PRINT LEFT$(B$, 1)
] 40 PRINT LEFT$(A$, 10)
] RUN
I
LASER 3000
]
```

LET

Purpose: To assign a value to a variable.

Format: [LET] *variable= expression*

Comments: LET is an optional statement, and is becoming less frequently used.

The equal sign (=) has exactly the same effect as LET.

The *expression* can be either a constant or an arithmetic expression.

If you attempt to assign a numeric value to a string variable, then the message "Type mismatch" will be displayed.

Examples:

```
] 10 LET A= 10
] 20 PRINT A
] 30 LET B= 40
] 40 LET B= A
] 50 PRINT B
] RUN
10
10
```

LIST

Purpose: To display on the screen the Basic program that is currently in memory.

Format: LIST [*linenumber* [,*linenumber*]]

Comments: If the *linenumber* (s) is (are) omitted, then LIST causes the entire program to be displayed.
If the first *linenumber* – is used, then LIST will display the program from that line to the end of the program.
If both are used, then LIST displays only those program lines in the range given by them. It also includes these lines.
If just *linenumber* is used, then LIST shows the lines from the first up to and including *line-number.*

In all cases when you use *linenumber*, it must be less than 63,999.

If you use just *linenumber* by itself, then just that line – if it exists – will be displayed.

Examples:

```
] 10 REM LASER 3000 PRESENTS
] 20 REM LASER BASIC
] 30 REM EASY TO USE
] 40 REM AND MORE....

] LIST
10 REM LASER 3000 PRESENTS
20 REM LASER BASIC
30 REM EASY TO USE
40 REM AND MORE....

] LIST 30
30 REM EASY TO USE

] LIST 20, 40
20 REM LASER BASIC
30 REM EASY TO USE
40 REM AND MORE

] LIST -30
10 REM LASER 3000 PRESENTS
20 REM LASER BASIC
30 REM EASY TO USE
```

LOAD

Purpose: To load a program from a data cassette tape into the computer.

Format: LOAD

Comments: This command is the opposite to SAVE, which stores a program on to a cassette tape. LOAD does not check wether your cassette tape unit is playing or recording, but, it does cause the Laser 3000 to issue a ''Beep'' at the start of LOADing, and to issue another at the end. Your cassette player should be in play mode when you use LOAD, and record mode when using SAVE.

LOMEM

Purpose: To set the lowest memory location available to a Basic program.

Format: LOMEM: *address*

Comments: This command is used to protect the area of memory below a program for data or machine language routines.

The *address* must be in the range −65535 to 65535.

MID$

Purpose: To return a specified number of characters from within a given string.

Format: MID$(*X$*,*i* [,*j*])

Comments: Both *i* and *j* must be between 1 and 255. MID$ return *j* characters of string *X$* starting from the *i* th character.

If *j* is not specified, then MID$ has the same effect as the RIGHT$(*X$*,*i*) function.

Also, if *i* is greater than LEN*(X$)*, then a null string is returned.

Example:

```
] 10 X$="Program     in"
] 20 Y$="     Fortran        Basic        Cobol"
] 30 PRINT X; MID$ (Y$, 11, 8)
] RUN
] Program in Basic
]
```

NEW

Purpose: Clears the current program from memory and clears all variables associated with it.

Format: NEW

Comments: This command is the most commonly used to free memory before entering a new program into the Laser 3000.

Basic returns to command level after executing NEW

Example: NEW

NOISE

Purpose: To produce noise from within programs

Format: NOISE *periodicity*, *duration* [, *volume*, *frequency control*]

Comments: The first parameter – *periodicity* can be equal to either 1 or 2.

When it equals to 1, white noise will be produced. When it equals to 2, periodic noise will be produced.

Duration can range from 1 to 255, with each unit being 1 video frame time, i.e. 1/50 or 1/60s.

Volumn can range from 0 to 15, with 15 being the loudest.

Frequency control equals to 1, 2, 3 or 4. This parameter affects the tone of the produced noise.

Example: NOISE 1, 120, 15

Produce a WHITE NOISE for about 2 seconds.

NORMAL

Purpose: To return the video display from either inverse or flashing modes to the default mode.

Format: NORMAL

Comments: If a **TEXT** *character, background, border* command was executed, **NORMAL** sets the display with *character* colour and *background* background. Otherwise, **NORMAL** sets the display with white characters on a dark background.

ONERR GOTO

Purpose: To avoid halting the program when an error is encountered.

Format: ONERR GOTO *linenumber*

Comments: Using this statement facilitates error trapping, as it can direct the program to a routine (an error handling foutine) dealing with error conditions that may arise in your program.

The RESUME statement can be used to come out from the error trapping routine.

The ONERR GOTO statement may be located anywhere within the program, but it is good practice to have it as early as possible, as this statement must be executed before the occurance of an error to avoid program interruption.

```
] 10 ON ERR GOTO 100
] 20 GET A
] 30 PRINT A
] 40 GOTO 20
] 100 PRINT ''INTEGERS ONLY''
] 110 RESUME
] RUN
1
1
2
2
A
INTEGERS ONLY
3
3
```

ON...GOSUB and ON...GOTO

Purpose: To direct the program flow depending on the value of a expression.

Format: ON *expression* GOSUB *linenumber 1 [, linenumber 2 . . .]*
ON *expression* GOTO *linenumber 1 [, linenumber 2 . . .]*

Comments: The value of the *expression* must always be an integer less than or equal to 255. When it is evaluated, it directs program flow to the corresponding line number in the list following either the GOSUB or the GOTO statements.

For instance, if the expression comes to five, then the program will branch to the fifth line number, and if it comes to nine, it will go to the ninth line number.

Example:

```
10  INPUT X
20  ON X  GOSUB 100, 200, 300
30  END

100   PRINT "Start of subroutine for X=1"
150   RETURN
200   PRINT "Start of subroutine for X=2"
250   RETURN
300   PRINT "Start of subroutine for X=3"
350   RETURN
]     RUN
? 2

Start of subroutine for X = 2
]
```

PAINT

Format:	PAINT (*x*,*y*), *colour, boundary.*
Purpose:	To fill a closed region on the screen with a selected colour.
Comments:	Starting from point(*x*,*y*), the region surrounded by the *boundary* colour is filled with the defined *colour.*

PAINT can be invoked only in bit image graphics mode with *x* in the range 0 to 279, *y* in the range 0 to 191, and both *colour* and *boundary* in the range 0 to 7.

The colour codes are as follows:

0	black
1	green
2	magenta
3	cyan
4	yellow
5	red
6	blue
7	white

PAINT can paint any type of figure. However, if the shape is extremely complicated with many corners, an error message could result.

PEEK

Purpose:	To read the byte at a specified memory location.
Format:	PEEK(*i*)
Comments:	*i* must be an integer in the range 0 to 65535. The byte returned by PEEK will be an integer between 0 and 255.
Example:	] I=PEEK (48345)

Example:

```
10 HGR5 : REM BIT IMAGE GRAPHICS
MODE
20 HCOLOR = 7
30 DRAW HCIRCLE (140, 90), 80 : REM
DRAWS A CIRCLE
40 PAINT (140, 90), 4, 7
```

When the program is executed, a white circle centered at (140, 90) will be painted yellow inside.

POKE

Purpose: To write a byte of data into a specified memory location.

Format: POKE *n*,*m*

Comments: The data to be placed in memory is *m*, which must be between 0 and 255. The memory location is *n*, and this must be in the range 0 to 65,535.

Important: The Laser 3000 does not check on the address you use in the POKE command, so if you POKE a value into one of its dedicated memory areas, or into your Basic program area, you may find that the machine ceases to operate.

Example:

```
] POKE 1000, 10
```

POP

Purpose: To change the action of a **RETURN** from a subroutine

Format: **POP**

Comments: **POP** effectively removes the top address from the stack of subroutine **RETURN** addresses.

Example:

```
10   GOSUB   100
20   END
100  GOSUB   200
110  PRINT "THIS STATEMENT
     NEGLECTED"
120  RETURN
200  POP
210  RETURN
RUN
```

Program flows from statement 10, 100, 200, 20. statement 110 is skipped due to **pop** action.

PR#

Purpose: To switch the output to the selected device

Format: PR#*device no*

Comments: The number given as *device no* must be between 0 and 8. PR#0 will turn off all selected device whereas PR#8 will list all function keys. If there is nothing connected at the given device then your Laser 3000 will suspend operation, and you will have to RESET the machine.

PRINT

Purpose: To display characters on the display screen.

Format: PRINT [*list*] [;]
? [*list*] [;]

Comments: The *list* is a number of values – either variables or constants – which may be strings or numbers.

If literal strings are to be printed out, they must be enclosed by quote marks ("literal").

If the *list* is not given, then PRINT will output a blank line, which can be handy for spacing out results as you display them on the screen.

If you separate the values in the list by commas, then each value will start in the next tab field, each of which comprises 16 column.

If you separate the value by one or more blanks or by a semicolon, then all the values will be run together.

Depending on the width of the display you have – see the WIDTH command – PRINT may run values over to the next line.

PRINT will use either integer or fixed point format for outputting numbers depending on whether they are expressible in nine or fewer digits.

Examples:

```
] PRINT 10, 20
10          20
] PRINT 10;20
1020
]
```

PRINT USING

Purpose: To format output in a desired way.

Format: PRINT USING *formatfield*; *expression1*; *expression2*; . .

Comments: The parameter *formatfield* determines just how your program results will look when they are printed on the display screen or to a printer. It consists of special formatting characters which may be classified into two types:

- string type, which control the layout of strings; and
- numeric type, which affect the layout of numbers.

STRING TYPE CHARACTERS

Character	*Action*
" ! "	Only the first character is printed.
" & "	The entire string is printed.
"/ n spaces /"	The next field will print n+2 characters. if the string is longer than the field, the extra characters are ignored. If field is longer than the string, the string will be moved left, left-justified in the field, and padded with spaces to the right.

Example:

```
10    A$ = " VIDEO "
20    B$ = " TECHNOLOGY "
30    PRINT  USING " &   "; A$; B$
40    PRINT  USING " ! " ; A$; B$
50    PRINT  USING " /    / " ; A$; B$
RUN
VIDEOTECHNOLOGY
VT
VIDETECH
```

NUMERIC TYPE CHARACTERS

Character	*Action*
#	The hash (#) sign is used to represent each digit position of a number. A decimal point may be inserted at any position in the field. If the number to be printed has fewer digits than the number of positions given, then the number will be moved to the right (right-justified), and the area before it filled with spaces. If the number to be printed has more digits than the number of positions allocated to it, the format field will be filled with @ to indicate overflow. Numbers are also truncated as necessary.

Character	*Action*
+/–	A plus sign (+) at the beginning of the format string will cause the sign of the number to be printed before the number. A minus sign (–) at the end of the format field will cause negative numbers to be printed with a trailing sign. You cannot use a minus sign at the front of a format field.

Example:

```
10    A = .776 : B = -2.3 : C = 1234
20    PRINT  USING "##.##"; A ; B ; C
30    END

RUN
    0.77  -2.30@@@@@@
```

This layout can be improved by putting some spaces at the end of the format field. This will separate the printed values on the line.

```
20    PRINT  USING " ##.##  "; A; B; C
30    END

RUN
    0.77    -2.30   @@@@@@
```

Example:

```
10   A = .776 : B = -2.3 : C = 1234
20   PRINT  USING "+##.##      "; A; B; C
30   PRINT  USING "##.##-       "; A; B; C

RUN
  +0.77      -2.30    @@@@@@
   0.77      2.30-    @@@@@@
```

Character	*Action*
**	A double asterisk at the start of a format field causes leading spaces in the number to be filled with asterisks. This also allocates two more digit positions to the format. For example: PRINT USING " **#.# " ; 2.23; -.9; 123 Resulting in – ***2.2 **-0.9 *123.0
$$	Using this at the beginning of a format field causes a dollar ($) sign to be printed to the left of the output number. The $$ also sets up two more digit positions, one of which is taken up with the dollar sign itself. For example: PRINT USING "$$#.# " ; 2.23; -.9; 123 $2.2 -$0.9
**$	This combines the effects of the last two formatting characters sets. For example: PRINT USING " **$#.# "; 2.23; -.9; 123 Resulting in this output: ****$2.2 ***-$0.9 **$123.0

Character	*Action*
^^^^	Four carats (^) at the end of a format field specify that the number should be output in exponential form. The four carats make spaces so that the sign of the exponent may be printed out as well. The significant digits are left-justified and the exponent is adjusted. For example: PRINT USING"##.#^^^^ "; 999; -.892; .0005 will result in this output: 99.9E+01 -89.2E-02 50.0E -05
,	A comma in front of the decimal point in the format field causes a comma to be printed to the left of every three significant digits from the decimal point. For example:] PRINT USING "####,.#"; 1234.56 1,234.5]

READ

Purpose: To read values from a DATA statement and to assign them to variables.

Format: READ *variable* [,*variable...*]

Comments: The READ statement must be accompanied by the DATA statement. Enough data must be specified by the DATA statement in order to be READ otherwise on 'OUT OF DATA ERROR' may result.

variable can be either numberic or string variables.

DATA statements can be re-used after they have been READ once, but to do this you must use the RESTORE command.

Example:

```
] 10 READ A
] 20 READ B$
] 30 PRINT A; " " ; B$
] 40 DATA 10, IS TEN

] RUN
10 IS TEN
```

RECALL

Purpose: To read a numeric array values that have been written to a data cassette tape.

Format: RECALL *arrayname*

Comments: This command is used in conjunction with the STORE command, which writes array values on to cassette tape.

The *arrayname* does not have to be the same as that used in the STORE command, but the DIMensions must match up, otherwise the values will be scrambled.

Example:

```
DIM      EX (7, 6, 2)
RECALL     EX
```

REM

Purpose: To let you REMind yourself by REMarks of what your program should do.

Format: REM *remark*

Comments: REM statements are not executed, and they only appear when your Basic program is listed. You will find them useful to document your programs with, despite the fact that they take up memory space.

They can be added at the end of Basic program lines if they are preceded by a colon.

Examples:

```
10 REM THIS IS A REMARK
20 PI = 3.14 : REM APPROXIMATE VALUE
   OF PI
```

RESTORE

Purpose: To use DATA values again after they have been READ.

Format: RESTORE

Comments: After a RESTORE statement is executed, the next READ statement will read the first item of the very first DATA statement in your program.

Examples:

```
] 10 READ A
] 20 READ B
] 30 DATA 10, 20, 30
] 40 PRINT A, B
] 50 RESTORE
] 60 READ C, D, E
] 70 PRINT C, D, E

] RUN
10        20
10        20        30
```

RESUME

Purpose: To restart a program that has been halted due to an error.

Format: RESUME

Comments: This command is mainly used at the end of an error handling routine, and it causes the program to restart execution at the statement which caused the error. If an error occurs during the error handling, then RESUME will place your program in an infinite loop. You can get out of this only by pressing the RESET botton.

RIGHT$

Purpose: To return a specified number of characters from a string proceeding from the right

Format: RIGHT$(*X$,i*)

Comments: If *i* is greater than or equal to LEN(X$), then the whole string is returned. Integer *i* must be between 1 and 255. see LEN, LEFT$, and MID$ string functions.

Example:

```
] 10 X$="Laser 3000"
] 20 PRINT RIGHT$(X$,4)
] RUN
3000
]
```

ROT

Purpose: To specify the angle by which a shape drawn by either DRAW or XDRAW commands will ROTate.

Format: ROT=*angle*

Comments: For shapes drawn using the DRAW *shape* commands, the value given as *angle* can be anything between 0 and 255, with 255 representing a 360 rotation.

For shapes drawn using the shape table method, a value of 16 for *angle* will rotate a shape through one right angle (90°) ; twice 16 (32) will rotate it through two right angles (180°); 48 will rotate it throught three right angles; and 64 (=4 X 16) will perform a complete rotation, bringing it back to its original position.

Example: See SHLOAD command for an example on ROT

RUN

Purpose: To start a program execution.

Format: RUN *[linenumber]*

Comments: Unless *linenumber* is given, RUN always begins execution with the lowest numbered line.

When *linenumber* is specified, it starts at that line.

Example:

```
] 10 PRINT "FIRST LINE"
] 20 PRINT "SECOND LINE"
] 30 PRINT "ALL DONE"

] RUN
FIRST LINE
SECOND LINE
ALL DONE
]  RUN 30
ALL DONE
```

SAVE

Purpose: To write a program on to data cassette tape.

Format: SAVE

Comments: This command is the opposite to LOAD, which reads a program from cassette tape into the computer's memory.
SAVE does not check whether your cassette player is running in either play or record modes, but it does cause the Laser 3000 to issue a "Beep" at the start of SAVing, and to issue another at the end.
Your cassette player should be in *record* mode when you use SAVE.

SCALE

Purpose: To increase or decrease the size of shapes created by **DRAW** or **XDRAW.**

Format: **SCALE** = *size*

Comments: The *size* number must be between 1 and 255. If it equals 2, each straight line in the shape is doubled; if it equals 3, each line is tripled in size; and so on up to a multiplication factor of 255.

Example: See **SHLOAD** command for an example in **SCALE.**

SHLOAD

Purpose: To load a shape table in memory.

Format: SHLOAD

Comments: Once a shape table has been created, you can save it on tape.

To do this, you have to:

(1) store the length of the shape table into hex location 0 (for lower two digits) and 1 (for upper two digits)

(2) Type 0 . 1 W *start . end* W in kernel mode

start: the start address of the shape table.

end: end address of the shape table.

(3) Put your cassette in recording mode and press RETURN on your Laser 3000

Once the save process is completed, you can put the tape aside and use it later.

To load a shape table into the memory, rewind the tape to the start of data, type in SHLOAD and RETURN, then press PLAY on your cassette unit.

As in loading a Basic program from tape, you should hear two beep's during the whole loading process.

Example: starting address at the shape table = $ 1000
ending address at the shape table = $ 1200

length of the shape table = $ 200

To save this shape table on tape, you should enter the kernel mode and type the following:

```
]  CALL-151
*  00:00 02
*  0.1 W 1000 . 1200 W
*  E003G
]
```

To load the shape table, you must rewind the tape and type in:

```
SHLOAD
```

SOUND

Purpose:	To produce sounds through the internal sound generator.
Format:	SOUND *pitch, duration* [, *channelnumber, volume*] SOUND DEF SOUND TEMPO *duration* SOUND *note, no of beats length* [, *channel number, volume*] SOUND
Comments:	There are two types of SOUND statement available on the Laser 3000.

The first type uses *pitches* that can be varied almost continuously — between 1 and 63 — while the second uses *pitches* that relate to actual notes on the music scale.

The four parameters — *pitch, duration, channelnumber,* and *volume* can either be specified as constant numbers or as variables, or in any mixture of both.

The highest pitch in the first type of command is produced when *pitch* equals 1, and the lowest when it equals 63.

duration can range from 1 to 255, with each unit being 1 video frame time, ie $\frac{1}{50}$ or $\frac{1}{60}$ s. To get a duration of one second, set duration equal to 50 or 60.

channelnumber can be 1, 2, or 3, depending on where you want the sound produced. Only one channel can be active at any one time.

volume can range from 0 to 15, with 15 being the loudest.

The second type of SOUND command allows you to use all 3 tone channels to sound music notes simultaneously. First you declare that the BASIC statements thereon are for defining the notes of a piece of music composition, by entering:

linenumber SOUND DEF

Secondly, you define the duration of each beat: A *duration* of 1 is approximately 0.01s. The range of *duration* is from 1 to 255.

linenumber SOUND TEMPO *duration*

Thirdly, you define the notes for each tone channel:

linenumber SOUND *note, no of beat length,* [, *channel number, volume*]

And you execute the above definition:

linenumber SOUND

Notice that for each tone channel, you can define, at most, 16 notes. For a music piece longer than 16 notes per channel, you can repeatedly DEFine and SOUND the composition.

The notes

Each notes must be specified as *x1 x2* [*x3*] where:

x1 can be chosen from A, B, C, D, E, F, G:
x2 can be chosen from 1, 2, 3, 4, 5, 6, or 7 and
x3 can be chosen from# (sharp), or b or F (flat)
Seven octaves are thus provided. A4 is middle A, with a frequency of 440 Hz, the reference harmonic. C4 is middle C.

NUMBER OF BEATS LENGTH

The length of a beat can vary from 1 to 31. Usually the shortest note in a music piece should be taken as one beat.

Examples: Turn on channel 2 for 2 seconds

```
Sound 50 , 120, 2, 10
```

volume

channel 2

duration = $120 \times \frac{1}{60}$ S = 2S

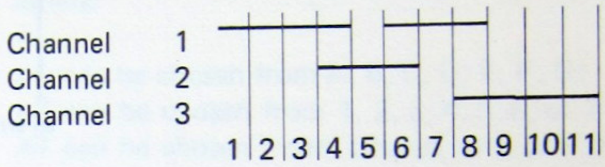
an artbitrary tone

Send out 10 different tones through channel 1

```
10      For I = 1 to 10
20      SOUND  I * 6, 60, 1, 15
30      NEXT I
40      END
```

the duration of each tone = 1 second

The following example demonstrates the usage of the three sound channels.



Example:

```
10    SOUND    DEF
20    SOUND    TEMPO    100
30    SOUND    C4, 4, 1, 15  ⎫
40    SOUND    C4, 1, 1, 0   ⎬— data for channel 1
50    SOUND    E4, 3, 1, 15  ⎭
60    SOUND    E4, 3, 2,  0  ⎫— data for channel 2
70    SOUND    D4, 3, 2, 15  ⎭
80    SOUND    D4, 7, 3, 0   ⎫
                             ⎬— data for channel 3
90    SOUND    F4, 4, 3, 15  ⎭
100   SOUND
```

Line 40, 60, and 80 are used to idle the channels by setting the channel volumn to 0, hence turning off the channel. The tone is meaningless and can be artbitrarily set when a channel is off.

SPC

Purpose: To separate two printed items by a specified number of spaces.

Format: PRINT SPC *(expression)*

Comments: This command, which is used in conjunction with the PRINT command, can be used to lay-out results printed by a program.

The value evaluated from *expression* must range between zero and 255.

SPEED

Purpose: To specify the rate at which characters are to be sent to an output device.

Format: SPEED = *rate*

Comments: The slowest *rate* is zero, and the fastest and the default rate is 255.

STOP

Purpose: To halt program execution and return to command level.

Format: STOP

Comments: This command is similar to END, except that STOP causes the message ''Break in nnnnn'' to be displayed, where nnnnn is the line number of the STOP statement.

The Laser 3000's Basic interpreter always returns to command level after a STOP is executed.

Example:

```
] 10 READ A
] 20 PRINT 7*A
] 30 STOP
] 40 DATA 7

] RUN
49
BREAK IN LINE 30
```

STORE

Purpose: To write a numeric array on to a data cassette tape.

Format: STORE *arrayname*

Comments: This command writes the values in an array to cassette tape, but it does not store the name of the array used. Thus it could be RECALLed with another array name. RECALL is the command that reads array values from cassette tape back into the Laser 3000's memory.

Example:

```
DIM EX (7, 6, 2)
STORE EX
```

The array element EX (0, 0, 0) Through EX (7, 6, 2) will be stored on to the cassette tape.

STR$

Purpose: To return a string representation of a numeric value.

Format: STR$(*x*)

Comments: This is a good means of checking the number of digits in a numeric constant, if it is used in conjunction with the LEN string function.

Example:

```
] 10  INPUT A
] 20  X$ = STR$ (A)
] 30  PRINT X$
] 40  PRINT " THE NUMBER HAS ";
      LEN (X$); " DIGITS "
] 50  GOTO 10
RUN
? 1234
1234
THE NUMBER HAS 4 DIGITS
?
```

SWAP

Purpose: To interchange the values of two variables.

Format: SWAP A, B

Comments: The variables may be of the same type, i.e. they must both be integer, floating point or string variables.

Example:

```
10   A = 10 : B = 20
20   PRINT A, B
30   SWAP  A, B
40   PRINT A, B

RUN

10        20
20        10
```

TAB

Purpose: To move the cursor a specified number of places to the right of the left margin.

Format: PRINT TAB *(expression)*

Comments: The TAB function only moves the cursor to the right. Hence, if the value evalvated from the expression is smaller then the column number of the current cursor position, the cursor will not move.

The value of *expression* must be form 0 to 255.

Example:

```
10    PRINT TAB (10)
20    PRINT "10 COLUMNS TO THE LEFT"
```

The computer will print out the string on statement 20 starting from the 10th column.

TEXT

Purpose: To set the display to full-screen text mode or to set character, background and border colours.

Format: TEXT
TEXT *character* [*, background, border*]
TEXT NORMAL

Comments: The full-screen text mode is made up of 24 lines of between 40 and 80 character each.
If no operand is given, TEXT set the display to full-screen text mode. Otherwise graphics and characters are set to have *character* colour with *background* surrounded by a *border* no matter what the original colours of the display are.

TEXT NORMAL will set the character and background to their default colour, i.e. white and black.

If the background colour is not specified, the character and border colours must be separated by 2 commas, for example:

TEXT NORMAL,, BLACK enables normal coloured display.

Example:

```
]    TEXT GREEN, BLUE, RED
```

All display graphics or characters will have green dots with blue background surrounded by a red border.

TROFF

Purpose: To stop program statement numbers from being displayed as a program executes

Format: **TROFF**

Comments: This turns off **TRON** . If **TRON** is not on, **TROFF** has no effect.

TRON

Purpose: To display line numbers of a program as they are executed.

Format: TRON

Comments: TRON is very useful in determining where a program may be going wrong (debugging). The line number of statements executed thereafter is displayed. TRON is turned off by the **TROFF** command.

Example:

```
] 10 FOR J = 1 TO 3
] 20 PRINT J*2
] 30 NEXT  J
] 40 END

] TRON
] RUN


LINE 10
LINE 20
2
LINE 30
LINE 20
4
LINE 30
LINE 20
6
LINE 30
LINE 40
]
```

USR

Purpose: This command specifies a parameter of an assembly language subroutine.

Format: USR (*n*).

Comments: *n* is arithmetic expression. When USR is encountered, the arithmetic expression is evaluated and placed in the floating point accumulator, and a JSR to location 0A is performed which must then contain a JMP to the beginning location of the machine-language subroutine. An RTS machine instruction should be executed at the end of the machine language subroutine.

Example:

```
] CALL-151
* 0A : 4C 10 03
* 310 : 60
* E003G
] PRINT USR (9) * 12
108
]
```

A JMP $310 instruction is placed at location $0A and RTS instruction at $ 310.

VTAB

Purpose: To move the cursor a given number of lines down the display screen.

Format: VTAB *number*

Comments: As there are only 24 lines on the display, number values outside of 1 to 24 will cause an error. The screen lines are numbered from top to bottom.

Example:

```
10   HOME
20   VTAB 10
30   PRINT " DOWN 10 ROWS"
```

WAIT

Purpose: To suspend a program's execution while watching the status of an input port.

Format: WAIT *portnumber*, *n* [,*m*]

Comments: This command suspends a program's execution until a specified input port develops an expected bit pattern.

The command loops around, reads the data at the port, XORs it with the integer value *m*, and then ANDs the result with the integer value *n*. If *m* is not specified, it is taken to be zero.

If the result at the end of the loop is zero, the loop starts over again.

If the result is not zero, then the program reumes execution at the next executable statement after the WAIT.

Careful: you can get into a continuous loop with the use of the WAIT command. Push the Laser 3000's Reset button if you believe this has happened. It will let you out of the loop, and return you to command level.

Example: WAIT 49152, 128
This will wait until a key is pressed which will set the most significant bit.

WIDTH

Purpose: To set the width of the text window.

Format: WIDTH *n*

Comments: On power up, the text window is set to 40 columns wide (or 80 columns if you have pressed the ESC key during power up).

You can shorten or extend the text window to *n* characters by WIDTH *n*, where *n* ranges from 1 to 80.

Example:

```
WIDTH 70
```

The screen will be cleared and the cursor returns to HOME position. Keyboard entry will be displayed from column 1 to column 70. The 71st character entered will appear on the next line.

XDRAW

Purpose: To erase a drawn shape

Format: XDRAW *shape no* AT *x,y*

Comments: This command allows you to erase a shape without erasing the whole screen.

Example:

```
10 DRAW 1 AT 100,100
20 FOR D = 1 to 1000 : NEXT D
30 XDRAW 1 AT 100,100
```

Assuming you have defined shape 1, this program will first draw it at co-ordinates (100,100), then wait for a while and finally erase the drawn shape.

CHAPTER 4

LASER 3000 BASIC FUNCTIONS

LASER 3000 BASIC FUNCTIONS

This chapter lists alphabetically and describes the intrinsic functions available for the Laser 3000's Basic.

The arguments — or parameters — for the functions are usually enclosed in parentheses.

The conventions followed for the arguments are as follows.

y and *x*	Represent any numeric expressions
i and *j*	Represent any integer expressions
X$* and *Y$	Represent any string expressions

ABS

Purpose: To give the absolute value of a numeric expression.

Format: ABS(*x*)

Comments: This function always returns a positive value, and can be used with either floating point or integer values.

Example:

```
] PRINT ABS(9*(-7))
63
]
```

ASC

Purpose: To return the ASCII code for the first character of the specified string.

Format: ASC(*X$*)

Comments: An error will result if the string specified is a null string.

Example:

```
] PRINT ASC("Laser")
76
]
```

ATN

Purpose: To calculate the arctangent of an angle.

Format: ATN(*x*)

Comments: This gives the arctangent of *x* in radians, with the result in the range - π/2 to π/2.

Example:

```
] PRINT ATN(8)
1.44644133
]
```

COS

Purpose: To calculate the cosine of an angle.

Format: COS(*x*)

Comments: The value of the angle must be given in radians, and not degrees.

Example:

```
] PRINT COS(2)
-.416146836
]
```

EXP

Purpose: To calculate the value of "e" – the base of natural logarithms – raised to a specified power.

Format: EXP(*x*)

Comments: The value of x should be less than 89, or an overflow error will result.

Example:

```
] PRINT EXP(9)
8103.08393
]
```

FRE

Purpose: Reports on the number of bytes in memory that are not being used by Basic.

Format: FRE *(expression)*

Comments: Because strings in Basic can have different lengths, and need to be manipulated. This frequently causes the memory to become very fragmented, using this statement with a dummy argument can force Basic to gather up all the loose fragments into contiguous wholes (garbage collection.)

This frees up areas of memory, and can often give you a surprising amount more.

Example: **X = FRE (0)**

This would lead to a garbage collection operation. It may take some time.

```
PRINT FRE(0)
```

In addition to a garbage collection operation, this would print out the amount in bytes of free user memory.

INT

Purpose: To round a fractional number down to a whole number.

Format: INT(*x*)

Comments: This function always returns an integer that is less than or equal to the number x.

Examples:

```
] PRINT INT(31.98)
31
]
PRINT INT(-31.98)
-32
]
```

LEN

Purpose: To return the number of characters in a string.

Format: LEN(X$)

Comments: This function counts **all** characters in the specified string, including blanks and non–printing characters.

Example:

```
] NEW
] 30 X$="Laser 3000"
] 40 PRINT LEN(X$)
]RUN
10
]
```

LOG

Purpose: To calculate the natural logarithm of a specified value.

Format: LOG(*x*)

Comments: The value *x* must be greater than zero.

Example:

```
] PRINT LOG(669)
6.50578406
]
```

POS

Purpose: To return the current horizontal cursor position.

Format: POS(*i*)

Comments: The leftmost cursor position is 0 on the Laser 3000 display screen. The argument *i* is a dummy.

Example:

```
] HTAB (10) : PRINT POS (1)
          9
]
```

RND

Purpose: To return a random number between 0 and 1.

Format: RND(*x*)

Comments: The value of the dummy argument x, determines how the random numbers are generated. If x is greater than zero then RND (x) generates a new random number everytime it is used.

If x is less than zero, then RND (x) generates the same random number everytime it is used with the same argument.

Example:

```
] 10 FOR I=1 TO 6
] 30 PRINT INT(RND(1)*1000)
] 50 NEXT
] RUN

 797
 584
 268
 397
  31
 932
]
```

SGN

Purpose: To return the sign of a number.

Format: SGN(*x*)

Comments: If the number is greater than zero, then SGN returns 1; if it is zero, SGN returns zero; and if it is negative, then SGN returns -1.

Example:

```
] 10 INPUTA
] 20 B=SGN(A)
] 30 IF B=0 THEN 90
] 40 IF B>0 THEN 70
] 50 PRINT "A IS NEGATIVE"
] 60 GOTO 10
] 70 PRINT "A IS POSITIVE"
] 80 GOTO 10
] 90 PRINT "A IS ZERO"
] 100 GOTO 10
] RUN
? 1
A IS POSITIVE
?-4
A IS NEGATIVE
? 0
A IS ZERO
?
```

SIN

Purpose: To calculate the sine of a specified angle.

Format: SIN(*x*)

Comments: The value of the angle must be given in radians, and not degrees.

Example:

```
] PRINT SIN(4)
-. 756802495
]
```

SQR

Purpose: To calculate the square root of a specified value.

Format: SQR(*x*)

Comments: A negative value for *x* will cause an error.

Example:

```
] 10 FOR I=1 TO 6
] 20 PRINT 2^(2*I), SQR(2^(2*I))
] 30 NEXT
] RUN
4                2
16               4
64               8
256              16
1024             32
4096             64
]
```

TAN

Purpose: To calculate the tangent of a specified angle.

Format: TAN(*x*)

Comments: The value of the angle must be given in radians, and not degrees.

Example:

```
] PRINT TAN(12)
-.635859926
]
```

VAL

Purpose: To return the numerical value of a specified string.

Format: VAL(*X$*)

Comments: The function ignores leading spaces of the specified string.

Example:

```
]   PRINT VAL ("          78")
    78
]
```

GLOSSARY

GLOSSARY

This section of the Basic manual explains the technical terms you may come across as you use your Laser 3000 computer.

address: The location of a register, a section of memory, or data in either RAM or ROM memory.

algorithm: A set of rules used to solve a problem in a finite series of steps.

alphabetic character: A letter of the alphabet.

alphameric or alphanumeric: Referring to a character set that can include both letters and digits.

application program: A program applied to a specific task.

argument: A value that is passed from one program to another.

array: Elements arranged in table form in one or more dimensions.

ASCII: American Standard Code for Information Interchange. One of the standard code used for exchanging information among computers and assoicated equipment.

asynchronous: The occurance of certain data signal, which has no definite relationship with time or other signals. Mainly refers to communications devices.

attribute: A property of an item.

background: The part of the display screen surrounding a character.

backup: A secure copy of a program or data that can be used if the live version is corrupted.

baud: A measure of a device's communication speed; equal to bits per second.

binary: Usually refer to the number system based on two.

bit: A part of the binary number system, either 1 or 0.

boolean value: A logical value that is either true or false.

bootstrap: Derived from the phrase "Pulling oneself up by one's bootstrap." A program which starts the computer's kernel or operating system running.

bps: Bits per second.

buffer: An area of memory in a computer peripheral, such as a printer, that is used to accomdate data. A buffer helps to match a fast input with a slow output, and is comparable to a cistern.

bug: An error in a program.

byte: A set of eight binary bits.

call: To bring a computer program or a subroutine into effect, usually by specifying the initial arguments and by starting at a given point.

carriage return character(CR): A character that causes the print or display position to move to the first position on the next line.

channel: A path along which signals are sent.

character: Either a letter, a number, or special symbol, represented as a **byte**.

clock: Device that generates regular signals used for matching electronic operations within the computer. Each signal is called a clock pulse.

communication: The transmission and reception of information.

complement: The binary number formed from another by changing its ones to zeros, and its zeros to ones.
ones.

concatenation: Bringing together two strings of characters.

constant: A fixed, invariable value.

control character: A special character which controls the action of a peripheral device such as a printer or display.

co-ordinates: A set of two or more numbers which determine the position of a point in two or more dimensional space.

cursor: A flashing element on the display that indicates the position of data or program entry.

debug: To find and remove mistakes in a program.

default: An assumed value when a selection is possible.

delimiter: A character that groups or separates values.

diagnostic: Detection and isolation of a malfunction or mistake.

dummy: A fictitious argument, having no effect, but necessary for the operation to start working.

duplex: In data communications, referring to a simultaneous two-way independent transmission in both directions.

echo: To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters shown on the display screen.

element: A member of a set, usually a value in an **array**.

field: In a file record, a specific section used for a particular type of data.

file: A set of related records which is treated as a unit by a computer program.

flag: Indicators used for determining program flow rather like flags for railways.

floppy disk: A flexible diskette.

font: A family of **characters** of a particular style.

foreground: The part of the dislplay area that is the character itself.

format: The particular arrangement or layout of data on a data medium, such as the screen or a diskette.

form feed (FF): A **control character** that causes the print or display position to move to the next page.

function: A procedure which returns a result depending on the value of one or more independent variables in a determined way. For example, sine and tangent are functions which relate to angles.

function key: One of the eight keys labeled F1 through F8 on the top row of the keyboard.

garbage collection: Gathering up loose scraps of memory into continuous wholes.

half duplex: In data communication, referring to a one way independent transmission, at any one time.

hard copy: A printed copy of computer output.

hertz(Hz): A unit of frequency equal to one cycle per second.

increment: A specified value used to update a counter in a loop.

initialise: To set counters, switches, addresses, or contents of memory to their starting values at the beginning of a computer routine.

instruction: An expression in a computer program that performs an operation, such as a add or jump.

integer: A whole number, whether positive, negative or zero.

interface: A medium between two devices.

interpret: In Basic, the translation and performance of a language statement.

interrupt: To halt a process in such a way that it can be restarted.

joystick: A lever that can pivot in all directions and is used to manipulate a cursor in games.

justify: To align characters so that they are all either to one side, or all positioned at either the top or bottom of a layout.

K, or kilo (byte): A measure of computer memory. 1KB is 1,024 bytes.

keyword: One of the predefined words of a programming language; a reserved word.

leading: The left-most part of a field.

light pen: A light sensitive input device that interacts with a display screen to either move its cursor or to cause it to generate shapes.

line feed (LF): A control character that causes the print or display position to move to the first position on the next line.

literal: A constant string value.

loop: A set of instructions that are executed repeatedly while a certain condition is true.

M, or mega (byte): A measure of computer memory. 1MB is 1,048,576 bytes.

machine infinity: The largest number that can be represented in a computer's internal format.

mantissa: The decimal part of a number expressed in exponential format.

mask: A pattern of bits that is used to control the formation of another pattern of bits.

matrix: An array with two or more dimensions.

dot-matrix printer: A printer in which each character is formed by a pattern of dots, which are produced by a matrix of fine wires.

menu: A list of options.

nest: To incorporate a program structure of some kind into another structure of the same kind. For example, you can put loops within other loops, i.e. nesting, or call subroutines from within other subroutines.

null: Empty, having no meaning. In particular, a string with no characters in it.

octal: Referring to the base 8 number system.

offset: The number of units from a starting point (in a record, control block, or memory).

operand: That which is operated on by an operator.

operating system: Software that controls the execution of application programs.

operation: A well-defined action that, when applied to any permissible combination of known entities, produces a new entity.

overflow: When the result of an operation exceeds the capacity of a register.

overlay: To use the same areas of memory for different parts of a computer program at different times.

overwrite: To record into an area of storage so as to destroy the data that was previously stored there.

pad: To fill a block with dummy data, usually zeros or blanks.

page: Part of the screen buffer that can be displayed and/or written on independently.

parameter: A name in a procedure that is used to refer to an argument passed to that procedure.

parity check: A means of ensuring that data is passed correctly. Usually the number of 1 bits in a byte is summed. If the number of them is even, a parity bit is not set; if odd, a parity bit is set.

pixel: A single point on a video display.

port: An access point for data entry or exit.

precision: A measure of the accuracy of a calculation, frequently the number of decimal places in a number.

prompt: A query from the computer. The] sign is an example of a prompt in Laser 3000 Basic.

queue: A line or list of items waiting for attention.

random access memory(RAM): Storage in which you can read and write to any desired location.

range: The extent of values that a variable quantity may assume.

read-only: A type of access to data that allows it to be read but not modified.

record: A collection of related information, treated as a unit.

recursive: Referring to a process in which each step makes use of the results of earlier steps.

reserved word: A word that is defined for a special purpose, and that you cannot use as a variable name.

resolution: In computer graphics, a measure of the sharpness of an image, expressed as the number of lines per unit of length.

routine: Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

scalar: A value or variable that is not an array.

scale: To change the representation of a quantity, expressing it in other units, so that its range is brought within a specified range.

scan: To examine sequentially, part by part.

scroll: To move all or part of the display image vertically or horizontally so that new data appears at one edge as old data disappears at the opposite edge.

segment: A particular 64K-byte area of memory.

sequential access: An access mode in which records are retrieved in the same order in which they were written. Each successive access to the file refers to the next record in the file.

stack: A method of temporarily storing data so that the last item stored is the first item to be processed.

statement: A meaningful expression that may describe or specify operations and is complete in the context of a programming language.

stop bit: A signal following a character or block that prepares the receiving device to receive the next character or block.

string: A set of characters.

subscript: A number that identifies the position of an element in an array.

syntax: The rules governing the structure of a language.

table: An arrangement of data in rows and columns, a two dimensional array.

terminal: A device, usually equipped with a keyboard and display, capable of sending and receiving information.

toggle: Referring to anything having only two stable states; to switch back and forth between the two states.

trailing: Located at the end of a string or number. For example, the number 2000 has three trailing zeros.

trap: A set of conditions that describe an event to be intercepted and the action to be taken after the interception.

truncate: To remove the trailing elements from a string or a number.

two's complement: A form for representing negative numbers in the binary number system.

variable: A quantity that can assume any of a given range of values.

vector: An ordered set of numbers.

APPENDICES

APPENDIX I
ERROR MESSAGES

LASER 3000 BASIC ERROR MESSAGES

In most cases, when an error occurs in a Laser 3000 Basic program, the Interpreter returns to command level.

This is designated by the "] " prompt and a flashing cursor. The program remains in memory, and the variables are set at the values they had assumed at the time the error was encountered.

You can use the PRINT command in direct mode to ascertain the values your program variables had at the time of the error.

To avoid your program stopping on coming across an error, you can use error trapping. See the ONERR GOTO statement for an explanation of how to use this technique.

LIST OF ERROR MESSAGES AND EXPLANATIONS:

CAN'T CONTINUE

This message will occur when you have halted a program (using **STOP** or **CTRL-C** or **BREAK**), then edited it, and tried to **CONT**inue. It will also arise when you try to **CONT**inue a program after an error has occurred.

DIVISION BY ZERO

This error will stop your program executing. You cannot divide a number by zero.

ILLEGAL DIRECT

This occurs when you try to use the following statements in the direct mode:

- GET
- DEF FN
- INPUT

ILLEGAL QUANTITY

The argument given to an arithmetic or string expression either does not match the type of expression or it is out of the expression's range. Possibilities are:

- using the **SQR** – square root – function with a negative argument.
- using a negative argument for the subscript of an array.
- using a negative or zero argument with the **LOG** – natural logarithm – function.
- mismatches either using the string functions with numeric variables, or other functions with the wrong variables.

NEXT WITHOUT FOR

Self explanatory. The variable given in a **NEXT** statement does not match the variable name given in a **FOR** statement which is in operation.

Alternatively, a **NEXT** does not correspond to any **FOR** statement which is in effect.

OUT OF DATA

This occurs when a **READ** statement is executed but either all the **DATA** statements have already been read, or there is not a match between the number of variables in the **READ** statement and the number of values given in the **DATA** statement.

OUT OF MEMORY

This message can arise from a number of conditions and errors:

- your program is too large for the available memory.
- your program has too many variables for the Basic interpreter to handle – a number in excess of 100 combined with a long program will cause this error.
- if you have **FOR...NEXT** loops nested to more than 10 levels.
- if you have **GOSUB...RETURN**s nested more than 24 levels.
- if an expression is too complicated for the interpreter to decipher.
- if parentheses are nested to more than 36 levels.

The last two possibilities are related, with the second giving an indication of the level of complexity permitted.

FORMULA TOO COMPLEX

A string expression is either too long or too complicated. Break the expression down into smaller expressions.

OVERFLOW

The result of a calculation exceeded 10E38, which is the Laser 3000's maximum number size. If a number is calculated as less than 10E-38 – the Laser's minimum number size – then the result becomes zero, and execution continues with no message being printed.

REDIM'D ARRAY

If an array has been used relying on the default **DIM**ensioning of an array, and then the array is explicitly **DIM**ensioned with another statement, this message will be displayed. Alternatively, it occurs when two different **DIM**ension statements exist for the same array.

RETURN WITHOUT GOSUB

Self explanatory. A **RETURN** statement exists without a corresponding **GOSUB** statement.

STRING TOO LONG

Trying to use the string concatenation operator (+) to bring together two strings whose added length is greater than 255 characters. 255 is the maximum length of a string in Laser 3000 Basic.

BAD SUBSCRIPT

Your program has tried to reference an array element which is greater than the size of the subscript given for the array in its **DIM** statement.

This can also occur if an array is referred to using the wrong number of dimensions.

For example, if array **ARRAY** has been **DIM**ensioned **DIM ARRAY** (10, 10, 10), and a subsequent statement like **ARRAY** (9, 8, 7, 6) = 54 is come across, then this error message will be displayed.

SYNTAX ERROR

The manner in which a statement, function, or expression has been put is incorrect. Things to look for are missing commas, spaces, parentheses, periods, or for illegal characters starting a variable name.

TYPE MISMATCH

This occurs when you try to assign a string value to a numeric value, or a numeric value to a string value, or if either a numeric function receives a string value, or a string function a numeric value.

UNDEF'D STATEMENT

A line referred to in a **GOTO**, **GOSUB**, or IF...**GOTO** statement does not exist in your program.

UNDEF'D FUNCTION

A reference is made to a user defined statement which does not exist in the Basic program.

APPENDIX II
KEYS AND THE ASSOCIATED CODES

KEY	CTRL	CTRL & SHIFT	SHIFT	CAP. ONLY	LOWER CASE ONLY
SPACE	2Ø	2Ø	2Ø	2Ø	2Ø
) 0	3Ø	29	29	3Ø	3Ø
! 1	31	21	21	31	31
@ 2	32	ØØ	4Ø	32	32
# 3	33	23	23	33	33
$ 4	34	24	24	34	34
% 5	35	25	25	35	35
^ 6	36	1E	5E	36	36
& 7	37	26	26	37	37
* 8	38	2A	2A	38	38
(9	39	28	28	39	39
_ –	2D	5F	5F	2D	2D
+ =	3D	2B	2B	3D	3D
{ [	5B	7B	7B	5B	5B
}]	5D	7D	7D	5D	5D
: ;	3B	3A	3A	3B	3B
" '	27	22	22	27	27
< ,	2C	3C	3C	2C	2C
> .	2E	3E	3E	2E	2E
? /	2F	3F	3F	2F	2F
A	Ø1	Ø1	41	41	61
B	Ø2	Ø2	42	42	62
C	Ø3	Ø3	43	43	63
D	Ø4	Ø4	44	44	64

KEY	CTRL	CTRL & SHIFT	SHIFT	CAP. ONLY	LOWER CASE ONLY
E	Ø5	Ø5	45	45	65
F	Ø6	Ø6	46	46	66
G	Ø7	Ø7	47	47	67
H	Ø8	Ø8	48	48	68
I	Ø9	Ø9	49	49	69
J	ØA	ØA	4A	4A	6A
K	ØB	ØB	4B	4B	6B
L	ØC	ØC	4C	4C	6C
M	ØD	ØD	5D	4D	6D
N	ØE	ØE	4E	4E	6E
O	ØF	ØF	4F	4F	6F
P	1Ø	1Ø	5Ø	5Ø	7Ø
Q	11	11	51	51	71
R	12	12	52	52	72
S	13	13	53	53	73
T	14	14	54	54	74
U	15	15	55	55	75
V	16	16	56	56	76
W	17	17	57	57	77
X	18	18	58	58	78
Y	19	19	59	59	79
Z	1A	1A	5A	5A	7A
↑	1B 44	1B 44	1B 44	1B 44	1B 44
↓	ØA	ØA	ØA	ØA	ØA

KEY	CTRL	CTRL & SHIFT	SHIFT	CAP. ONLY	LOWER CASE ONLY
←	Ø8	Ø8	Ø8	Ø8	Ø8
→	15	15	15	15	15
0	3Ø	3Ø	3Ø	3Ø	3Ø
1	31	31	31	31	31
2	32	32	32	32	32
3	33	33	33	33	33
4	34	34	34	34	34
5	35	35	35	35	35
6	36	36	36	36	36
7	37	37	37	37	37
8	38	38	38	38	38
9	39	39	39	39	39
+	2B	2B	2B	2B	2B
–	2D	2D	2D	2D	2D
.	3Ø 3Ø	3Ø 3Ø	3Ø 3Ø	3Ø 3Ø	3Ø 3Ø
RETURN	ØD	ØD	ØD	ØD	ØD
ESC	1B	1B	1B	1B	1B
TAB	1C	1C	1C	1C	1C
BREAK	7F	7F	7F	7F	7F
RUBOUT	Ø8 2Ø Ø8	Ø8 2Ø Ø8	Ø8 2Ø Ø8	Ø8 2Ø Ø8	Ø8 2Ø Ø8

APPENDIX III
DISPLAY CHARACTER

40 COLUMN DISPLAY CHARACTERS

		Inverse				Flashing				Normal							
										(Control)						(Lowercase)	
Decimal		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
	Hex	S00	S10	S20	S30	S40	S50	S60	S70	S80	S90	SA0	SB0	SC0	SD0	SE0	SF0
0	S0	@	P		0	@	P		0	@	P		0	@	P		p
1	S1	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	a	q
2	S2	B	R	"	2	B	R	"	2	B	R	"	2	B	R	b	r
3	S3	C	S	#	3	C	S	#	3	C	S	#	3	C	S	c	s
4	S4	D	T	$	4	D	T	$	4	D	T	$	4	D	T	d	t
5	S5	E	U	%	5	E	U	%	5	E	U	%	5	E	U	e	u
6	S6	F	V	&	6	F	V	&	6	F	V	&	6	F	V	f	v
7	S7	G	W	'	7	G	W	'	7	G	W	'	7	G	W	g	w
8	S8	H	X	(	8	H	X	(	8	H	X	(	8	H	X	h	x
9	S9	I	Y	)	9	I	Y	)	9	I	Y	)	9	I	Y	i	y
10	SA	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	j	z
11	SB	K	[	+	;	K	[	+	;	K	[	+	;	K	[	k	{
12	SC	L	\	,	<	L	\	,	<	L	\	,	<	L	\	l	}
13	SD	M	]	–	=	M	]	–	=	M	]	–	=	M	]	m	|
14	SE	N	^	.	>	N	^		>	N	^		>	N	^	n	~
15	SF	O	_	/	?	O	_	/	?	O	_	/	?	O	_	o	▩

80 COLUMN DISPLAY CHARACTERS																
	Inverse				Flashing				Normal							
									(Control)						(Lowercase)	
Decimal	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
Hex	S00	S10	S20	S30	S40	S50	S60	S70	S80	S90	SA0	SB0	SC0	SD0	SE0	SF0
0 S0	@	P		0	@	P		p	@	P		0	@	P		p
1 S1	A	Q	!	1	A	Q	a	q	A	Q	!	1	A	Q	a	q
2 S2	B	R	"	2	B	R	b	r	B	R	"	2	B	R	b	r
3 S3	C	S	#	3	C	S	c	s	C	S	#	3	C	S	c	s
4 S4	D	T	$	4	D	T	d	t	D	T	$	4	D	T	d	t
5 S5	E	U	%	5	E	U	e	u	E	U	%	5	E	U	e	u
6 S6	F	V	&	6	F	V	f	v	F	V	&	6	F	V	f	v
7 S7	G	W	'	7	G	W	g	w	G	W	'	7	G	W	g	w
8 S8	H	X	(	8	H	X	h	x	H	X	(	8	H	X	h	x
9 S9	I	Y	)	9	I	Y	i	y	I	Y	)	9	I	Y	i	y
10 SA	J	Z	*	:	J	Z	j	z	J	Z	*	:	J	Z	j	z
11 SB	K	[	+	;	K	[	k	{	K	[	+	;	K	[	k	{
12 SC	L	\	,	<	L	\	l	|	L	\	,	<	L	\	l	|
13 SD	M	]	–	=	M	]	m	}	M	]	–	=	M	]	m	}
14 SE	N	^	.	>	N	^	n	~	N	^		>	N	^	n	~
15 SF	O	-	/	?	O	-	o	▒	O	-	/	?	O	-	o	▒

APPENDIX IV
ASCII CHARACTER CODES

The following table lists all the ASCII codes (in decimal) and their associated characters. These characters can be displayed using PRINT CHR$(n), where n is the ASCII code. The column headed "Control Character" lists the standard interpretations of ASCII codes 0 to 31 (usually used for control functions or communications). They are all non-printing characters on the LASER 3000.

ASCII value	Control character	ASCII value	Character
000	NUL	032	(space)
001	SOH	033	!
002	STX	034	"
003	ETX	035	#
004	EOT	036	$
005	ENQ	037	%
006	ACK	038	&
007	BEL	039	'
008	BS	040	(
009	HT	041	)
010	LF	042	*
011	VT	043	+
012	FF	044	,
013	CR	045	-
014	SO	046	.
015	SI	047	/
016	DLE	048	0
017	DC1	049	1
018	DC2	050	2
019	DC3	051	3
020	DC4	052	4
021	NAK	053	5
022	SYN	054	6
023	ETB	055	7
024	CAN	056	8
025	EM	057	9
026	SUB	058	:
027	ESC	059	;
028	FS	060	<
029	GS	061	=
030	RS	062	>
031	US	063	?

ASCII value	Character	ASCII value	Character
064	@	095	_
065	A	096	’
066	B	097	a
067	C	098	b
068	D	099	c
069	E	100	d
070	F	101	e
071	G	102	f
072	H	103	g
073	I	104	h
074	J	105	i
075	K	106	j
076	L	107	k
077	M	108	l
078	N	109	m
079	O	110	n
080	P	111	o
081	Q	112	p
082	R	113	q
083	S	114	r
084	T	115	s
085	U	116	t
086	V	117	u
087	W	118	v
088	X	119	w
089	Y	120	x
090	Z	121	y
091	[	122	z
092	\	123	{
093	]	124	¦
094	∧	125	}

APPENDIX V
MATHEMATICAL FUNCTIONS

Functions that are not intrinsic to LASER 3000 Personal Computer BASIC may be calculated as follows.

Function	Equivalent
Secant	SEC(*x*) = 1/COS(*x*)
Cosecant	CSC(*x*) = 1/SIN(*x*)
Cotangent	COT(*x*) = 1/TAN(*x*)
Inverse sine	ARCSIN(*x*) = ATN(*x*/SQR(1–*x***x*))
Inverse cosine	ARCCOS(*x*) = 1.570796 –ATN(*x*/SQR(1–*x***x*))
Inverse secant	ARCSEC(*x*) = ATN(SQR(*x***x*–1)) +(*x*<0)*3.141593
Inverse cosecant	ARCCSC(*x*) = ATN(1/SQR(*x***x*–1)) +(*x*<0)*3.141593
Inverse cotangent	ARCCOT(*x*) = 1.57096–ATN(*x*)
Hyperbolic sine	SINH(*x*) = (EXP(*x*)–EXP(–*x*))/2
Hyperbolic cosine	COSH(*x*) = (EXP(*x*)+EXP(–*x*))/2
Hyperbolic tangent	TANH(*x*) = (EXP(*x*)–EXP(–*x*)) /(EXP(*x*)+EXP(–*x*))
Hyperbolic secant	SECH(*x*) = 2/(EXP(*x*)+EXP(–*x*))
Hyperbolic cosecant	CSCH(*x*) = 2/(EXP(*x*)–EXP(–*x*))
Hyperbolic cotangent	COTH(*x*) = (EXP(*x*)+EXP(–*x*)) /(EXP(*x*)–EXP(–*x*))
Inverse hyperbolic sine	ARCSINH(*x*) = LOG(*x*+SQR(*x***x*+1))
Inverse hyperbolic cosine	ARCCOSH(*x*) = LOG(*x*+SQR(*x***x*–1))
Inverse hyperbolic tangent	ARCTANH(*x*) = LOG((1+*x*)/(1–*x*))/2
Inverse hyperbolic secant	ARCSECH(*x*) = LOG((1+SQR(1–*x***x*))/*x*)
Inverse hyperbolic cosecant	ARCCSCH(*x*) = LOG((1+SGN(x) *SQR(1+*x***x*))/*x*)
Inverse hyperbolic cotangent	ARCCOTH(*x*) = LOG((*x*+1)/(*x*–1)/2

If you use these functions, a good way to code them would be using the DEF FN statement. For example, instead of typing the formula for inverse hyperbolic sine each time you need it, you could use a program line.

```
DEF FN INSINEH(x) = LOG (x+SQR(x*x+1))
```

then refer to it as

```
Z = FN INSINH(x)
```

APPENDIX VI

SUMMARY OF BASIC COMMANDS

COMMAND	*DESCRIPTION*
AMPERSAND COMMAND (&)	To jump into a machine language command starting at hex location $3F5.
CALL	To use an assembly language subroutine.
CHR$	Converts an ASCII code to its equivalent character.
CLR	To clear all variables, arrays and strings.
CONT	To start a program running again after it has been halted.
DATA	To store constant numbers and string values in your program so they can be used in it in conjunction with the READ statement.
DEF FN	Allows you to define and name a function.
DEL	Removes program lines.
DIM	This gives the values for the subscripts of arrays, and allocates enough storage to accomodate them.

DRAW	To draw pre-defined geometric shapes.
END	Finishes program execution and returns you to command level.
FLASH	To cause all computer message to alternate between character and background colour.
FOR...NEXT	Loops around a group of instructions a specified number of times.
FRE	Reports on the number of bytes in memory that are not being used by Basic.
GET	Reads a character from the keyboard without echoing it on the screen. No carriage return is necessary.
GOSUB...RETURN	To direct the program flow into, and to return from, a subroutine.
GOTO	To direct the program flow to another part of a Basic program.
HCOLOR	To set the colour of subsequently plotted graphics.

HGR HGR1 HGR2 HGR3 HGR4 HGR5 HGR6	To set up the different graphics modes.
HIMEM	To set the highest memory location available to a Basic program.
HPLOT	To draw either lines or dots.
HOME	To clear screen and position the cursor to the upper left corner of the display screen.
HTAB	To move the cursor a given number of places to the right of the left margin.
IF...GOTO and IF...THEN...	To direct program flow depending on the result of an evaluation.
IN #	To accept input from selected input device.
INPUT	Allows you to enter values from the keyboard while a program is executing.
INVERSE	To reverse the character and background colour of the video display.

LEFT$	Returns a specified number of characters from the left-hand side of a character string.
LET	To assign a value to a variable.
LIST	To display on the screen the Basic program that is currently in memory.
LOAD	To load a program from a data cassette tape into the computer.
LOMEM	To set the lowest memory location available to a Basic program.
NEW	Clears the current program from memory and clears all variables associated with it.
NOISE	To produce noise from the internal noise generator.
NORMAL	To return the video display from either inverse or flashing modes to the default mode.
ONERR GOTO	To avoid halting the program when an error is encountered.
ON...GOSUB ON...GOTO	To direct the program flow depending on the value of a expression.
PAINT	To fill a region on the screen with a selected colour.

POKE	To write a byte of data into a specified memory location.
POP	To change the action of a RETURN from a subroutine.
PRINT	To display characters on the display screen.
PRINT USING	To format output in a desired way.
PR#	To switch the output to the selected device.
READ	To read values from a DATA statement and to allocate them to variables.
RECALL	To read numeric array values that have been written to a data cassette tape.
REM	To let you REMind yourself by REMarks of what your program is doing.
RESTORE	To use DATA values again after they have been READ.
RESUME	To restart a program that has been halted due to an error.

ROT	To specify the angle at which a shape is rotated when drawn on the screen, used in conjuction with DRAW or XDRAW.
RUN	To start a program execution.
SAVE	To write a program on to a data cassette tape.
SCALE	To increase or decrease the size of shapes created by DRAW or XDRAW.
SHLOAD	To load a shape table in memory.
SOUND	To produce sounds through the internal sound generator.
SPC	To separate two printed items by a specified number of spaces.
SPEED	To specify the rate at which characters are to be sent to an output device.
STORE	To write a numeric array on to a data cassette tape.
STOP	To halt a program execution and return to command level.
SWAP	To interchange the values of two variables.

TAB	To move the cursor a specified number of places to the right of the left margin.
TEXT	To set the display to full-screen text mode or to set character, background and border colours.
TROFF	To stop program statement numbers from being displayed as a program executes.
TRON	To display line numbers of a program as they are executed.
USR	This command specifies a parameter of an assembly language subroutine.
VTAB	To move the cursor a given number of lines down the display screen.
WAIT	To suspend a program's execution while monitoring the status of an input port.
WIDTH	To set the width of the text window.
XDRAW	To erase a drawn shape.

APPENDIX VII

LIST OF RESERVED WORDS IN LASER 3000 BASIC

ABS	HIMEM	PEEK	STORE
AND	HOME	POKE	STEP
ASC	HPLOT	POP	SWAP
AT	HTAB	POS	TAB (
ATN	IF	PR	TAN
CALL	IN	PRINT	TEXT
CHR$	INPUT	READ	THEN
CLR	INT	RECALL	TO
CONT	INVERSE	REM	TROFF
COS	LEFT$	RESTORE	TRON
DATA	LEN	RETURN	USING
DEF	LET	RESUME	USR
DEL	LIST	RIGHT$	VAL
DIM	LOAD	RND	VTAB
DRAW	LOG	ROT =	WAIT
END	LOMEM :	RUN	WIDTH
EXP	MID$	SAVE	XDRAW
FLASH	NEW	SCALE =	&
FN	NEXT	SGN	+
FOR	NOISE	SHLOAD	–
FRE	NOT	SIN	*
GET	NORMAL	SOUND	/
GOSUB	ON	SPC (	>
GOTO	ONEER	SPEED	<
HCOLOR =	OR	SQR	=
HGR	PAINT	STOP	∧
HGR2	PDL	STR$	

LASER™
3000
PERSONAL COMPUTER