TANDY 102
Applications and BASIC Reference Guide

Cat. No. 26-3803

Contents

Part 1/
Applications Reference Guide

Part 2/
BASIC Reference Guide

PART 1/
Applications Reference Guide

This part is a quick reference guide to the information presented in the *Tandy 102 Owner's Manual.*

1/ Tandy 102

NUM	Turns on and off the numeric keypad.
PRINT	Prints the display.
LABEL	Turns on and off the function-key display.
SHIFT BREAK	Stops the current operation.

To set the day, date, and time: Enter BASIC and use the DAY$, DATE$, and TIME$, commands. (See "BASIC Keywords" in Part 2 of this guide.)

To rename or kill a file: Enter BASIC and use the NAME and KILL commands. (See "BASIC Keywords" in Part 2 of this guide.)

2/ SCHEDL/ADDRSS

F1 Finds records from NOTE.DO (SCHEDL) or ADRS.DO (ADDRSS) and lists them on the display.

F5 Finds records from NOTE.DO (SCHEDL) or ADRS.DO (ADDRSS) and prints them on the printer.

F8 Returns to the Main Menu.

3/ Text

TEXT Cursor Movement

→	Moves right 1 character.
←	Moves left 1 character.
↑	Moves up 1 character.
↓	Moves down 1 character.
SHIFT →	Moves to the next word.
SHIFT ←	Moves to the previous word.
CTRL →	Moves to the right end of the line.
CTRL ←	Moves to the left end of the line.
SHIFT ↑	Moves to the top of the display.
SHIFT ↓	Moves to the bottom of the display.
CTRL ↑	Moves to the top of the file.
CTRL ↓	Moves to the bottom of the file.

TEXT Editing

SHIFT DEL	Deletes a character.
BKSP	Backspaces and erases.
SHIFT PRINT	Prints a text file.
PASTE	Pastes the contents of the paste buffer.
F1	Finds text.
F2	Loads a text file from cassette tape, RS-232C or modem.
F3	Saves a text file on cassette tape, line printer, RS-232C or modem.
F5	Copies text into the paste buffer.
F6	Cuts text into the paste buffer.
F7	Selects text to cut or copy into the paste buffer.
F8	Returns to the Main Menu.

4/ TELCOM

TELCOM Interactive Mode

F1	Finds and autodials a number stored in ADRS.DO. (The number must follow a colon (:).)
F2	Autodials the number that you type and enter.
F3	Enters or displays TELCOM's parameters.
F4	Enters the terminal mode.
F8	Returns to the Tandy 102 Main Menu.

TELCOM Terminal Mode

F1 Displays the previous screen.

F2 Saves all transmitted information into a text file.

F3 Sends the information stored in a text file.

F4 Switches between the full-duplex and half-duplex echo modes.

F5 Prints all transmitted information on the printer.

F8 Disconnects from the terminal mode and returns to the interactive mode.

Part 2/
BASIC Reference Guide

This part is a reference to *Tandy 102 BASIC*. It assumes you already know how to program in BASIC and need to find out how BASIC is implemented on the Tandy 102.

To learn how to program in BASIC, we suggest the following book:

> *The TRS-80 Model 100 Portable Computer*, David A. Lien, CompuSoft Publishing, 1983.

5/ BASIC Operation

BASIC Modes

BASIC lets you operate it in the immediate mode, program execution mode, and edit mode:

To use the immediate mode: Type and enter any statement, for example, **NEW** ENTER. This causes the statement to immediately execute.

To use the execution mode: Type and enter the RUN statement or press F4. This causes the current BASIC program to run.

To use the edit mode: Type and enter the EDIT statement. This causes BASIC to load program lines into the Tandy 102 TEXT program. To return the lines to BASIC from TEXT, press the F8 key.

BASIC Keys

BASIC recognizes these special keys:

F1	same as typing FILES ENTER
F2	same as typing LOAD "
F3	same as typing SAVE "
F4	same as typing RUN ENTER
F5	same as typing LIST ENTER

F6	not used
F7	not used
F8	same as typing MENU ENTER
PRINT	same as typing LCOPY ENTER
SHIFT PRINT	same as typing LLIST ENTER
PAUSE	pauses execution of a BASIC program
SHIFT BREAK	breaks execution of a BASIC program

You can redefine the 8 of these keys—the 8 function keys—with the KEY statement.

BASIC Programs

BASIC lets you execute programs that contain:

- Up to 65529 lines
- Up to 255 characters per line.
- 1 or more BASIC statements per line, separated by colons (:).

These are examples of simple BASIC program lines. As these examples show, the spaces between the keywords are optional:

```
10 CLEAR : CLS : PRINT @ 35, "MENU";

20 PRINT@75,"1.Enter Data";:PRINT@115,
"2.Update Data";
```

6/ BASIC Data

BASIC lets you enter data into a program as a string or as a number. A string can contain any kind of characters; BASIC can store up to 255 characters in a string.

A number can contain only numeric characters; BASIC can store a number in 1 of 3 levels of precision. (More precision requires more memory.)

- Double precision numbers—These numbers range between +/− 10^{62} to +/− 10^{-64} and consist of up to 14 significant digits, plus a decimal point. To represent a double precision in exponential form, use the E notation. Examples:

  1.3402100054   3.1415926535898
  1.44343455331E-40

- Single precision numbers—These numbers range between +/− 10^{62} to +/− 10^{-64} and consist of up to 6 significant digits, plus a decimal point. To represent a single precision in exponential form, use the E notation. Examples:

  100.003   −23.4212   4.552E−14

- Integer numbers—These numbers range between −32768 to 32767 and include whole numbers only (no decimal numbers). Examples:

  1 32000 −2 500 −12345

Many statements let you enter data as an expression. An expression can consist of constants, variables, operations, and BASIC functions.

These are examples of numeric expressions:

```
52    N    N+2    TAN(N)+5
```

These are examples of string expressions:

```
"FRANK"  A$  A$+"FRANK"  A$+CHR$(13)
```

Constants

BASIC lets you use constants in string or numeric expression. To use a constant in a string expression, enclose the value in quotes. Example: "Enter Check 123"

To use a constant in a numeric expression, omit the quotes. Example: 1234. BASIC treats all numeric constants as double-precision numbers.

Variables

BASIC lets you use variables in any kind of expression. To use a variable, first equate the variable's name to a constant (for example, N=17); then use the variable name to refer to the constant.

A variable name can consist of any number of characters—the first of which needs to be a letter—however, BASIC recognizes only the first 2 characters in the variable name. For example, BASIC treats all the following variable names as 1 name—the variable name SU:

```
SU        SUPER        SUPERLATIVE
```

BASIC initially assumes that all variables are double precision numbers. To change this assumption, you can use these type definition statements:

```
DEF1NT   DEFSNG   DEFSTR   DEFDBL
```

You can also use any of these type declaration tags:

%	integer variable
!	single precision variable
#	double precision variables
$	string variables

A type declaration tag takes precedence over a type definition statement. For example, DEFINT L defines the variable L1 as an integer variable; however, L1$ is a string variable.

A variable name can be simple or subscripted. These are examples of simple variable names:

```
SU        D1        VA
```

These are examples of subscripted variable names (often called array variables):

SU(3,5,9)   DATA(2,5,3,5,5,)

When using subscripted variable names, you need to include a DIM statement at the start of the program to dimension enough memory. The only limit on the number of subscripts you can use is the amount of free memory.

Operations

BASIC lets you use only 1 operator in a string expression:

+ concatenate

BASIC lets you use any of the following operators in a numeric expression:

+	positive
−	negative
+	addition
−	subtraction
*	multiplication
/	division
\	integer division (enter the "\" by pressing GRAPH − at the same time)
^	exponentiation
MOD	modulus
<	less than
>	greater than

=	equal to
<> or ><	not equal to
=< or <=	less than or equal to
=> or >=	greater than or equal to
AND	logical AND
OR	logical OR
XOR	logical XOR
EQV	logical EQV
IMP	logical IMP
NOT	logical NOT

When you use more than one operator, BASIC performs the operations according to this hierarchy:

^

+, - (positive or negative)

*,/

MOD, \

+, -

<,>,=,=>,<=,><

NOT

AND

OR

XOR

EQV

IMP

You can override this hierarchy by enclosing operations in parentheses—BASIC works from the inner parentheses outwards. For example,
C=(A+B)/5+3

Functions

BASIC lets you use any of these functions in a string expression:

CHR$ DATE$ DAY$ INSTR LEFT$
MID$ RIGHT$ SPACE$ STR$ STRING$
TIME$

BASIC lets you use any of these functions in a numeric expression:

ABS ASC ATN CDBL CINT
COS CSNG ERL ERR EXP
FIX FRE INT LEN LOG
RND SGN SIN SQR TAN
VAL VARPTR

7/ BASIC Input/Output

BASIC has statements and functions that let you input and output to 7 devices. These statements and functions are listed on Table 1.

Screen Positions

BASIC lets you use the LINE, PSET, and PRESET statements to produce graphics on 15,360 screen positions (240 *x* positions and 64 *y* positions). The graphics screen positions are shown on Figure 1.

BASIC lets you use the PRINT @ statement, and the POS and CSRLIN functions, to control the cursor's location on the 320 positions. The cursor screen positions are shown in Figure 2.

Sound Frequencies

BASIC lets you use the SOUND statement to produce music, using the sound-generator frequency chart shown in Table 2.

RAM

CLOSE EOF INPUT# INPUT$ IPL KILL
LINE INPUT# LOAD LOADM MERGE
NAME OPEN PRINT# PRINT# USING
RUN RUNM SAVE SAVEM TAB

Cassette

CLOAD CLOAD? CLOADM CLOSE
CSAVE CSAVEM EOF INPUT# INPUT$
LINE INPUT # LOAD LOADM
MERGE MOTOR OPEN PRINT # USING
PRINT# RUN RUNM SAVE SAVEM TAB

Modem and RS-232

TAB MDM CLOSE EOF INPUT#
INPUT$ LOAD MERGE RUN SAVE
ON MDM GOSUB OPEN PRINT#
PRINT # USING COM ON COM GOSUB

Screen

CLS CSRLIN LIST POS PRINT PRINT @
PRINT USING SCREEN PRINT # TAB
PRINT USING # CLOSE OPEN

Line Printer

LCOPY LLIST LPOS LPRINT
LPRINT USING CLOSE OPEN PRINT #
SAVE PRINT # USING TAB

Keyboard

INKEY$ INPUT INPUT$ KEY KEY LIST
KEY LINE INPUT ON KEY GOSUB

Sound generator

BEEP SOUND

Table 1. BASIC Device Statements and Functions

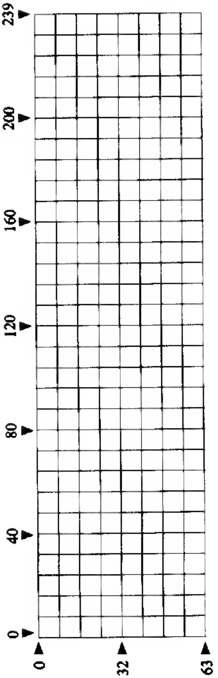

Figure 1.
Graphic Screen
Positions

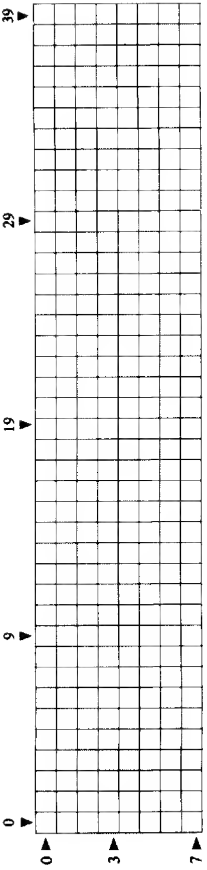

Figure 2.
Cursor Screen
Positions

Octave					
Note	1	2	3	4	5
G	12538	6269	3134	1567	783
G#	11836	5918	2959	1479	739
A	11172	5586	2793	1396	698
A#	10544	5272	2636	1318	659
B	9952	4976	2488	1244	622
C	9394	4697	2348	1174	587
C#	8866	4433	2216	1108	554
D	8368	4184	2092	1046	523
D#	7900	3950	1975	987	493
E	7456	3728	1864	932	466
F	7032	3516	1758	879	439
F#	6642	3321	1660	830	415

Table 2. Sound frequencies.

8/ BASIC Files

BASIC has many statements and functions that let you input and output to "device files," and, in many cases, these statements and functions are "device generic." For example, PRINT # is a device generic statement—It lets you output to files on 6 devices: RAM, cassette tape, modem, RS-232, screen, and printer.

Using device generic statements makes it easy to modify a program for a different device. For example, assume a program uses device generic statements to output to the screen. You can easily modify this program to output to the printer, rather than the screen, simply by changing the screen file specifications to printer file specifications.

File Specifications

When inputting or outputting to a device file, you need to give a file specification. The formats for file specifications are:

RAM files:	"RAM:*name*"
Cassette files:	"CAS:*name*"
Modem files:	"MDM:*wpbs*"
RS-232 files:	"COM:*rwpbs*"
Screen files:	"LCD:"
Line printer files:	"LPT:"

name can contain 1-6 characters. With RAM files, BASIC will add the following 2-letter extensions: ".BA", if the file is a BASIC program, or ".DO", if the file is ASCII data.

rwpbs specifies the following communication parameters:

r baud rate (omit if the device is MDM)
1 = 75; 2 = 110; 3 = 300; 4 = 600; 5 = 1200;
6 = 2400; 7 = 4800; 8 = 9600; 9 = 19200.

w word length
6 = 6 bits; 7 = 7 bits; 8 = 8 bits.

p parity
O = Odd; I = Ignore; N = None; E = Even.

b stop bits
1 = 1 stop bit; 2 = 2 stop bits.

s start/stop (XON/XOFF) enablement
E = enable;D = disable.

Examples of using the same statement to open a file for outputting data to RAM, cassette tape, the modem line, the RS-232 line, the screen, and the line printer:

```
OPEN "RAM:ACCTS" FOR OUTPUT AS 1
OPEN "CAS:DATA1" FOR OUTPUT AS 1
OPEN "COM:37E1E" FOR OUTPUT AS 1
OPEN "MDM:7E1E" FOR OUTPUT AS 1
OPEN "LCD:" FOR OUTPUT AS 1
OPEN "LPT:" FOR OUTPUT AS 1
```

File Types

BASIC uses 2 kinds of files: BASIC program files (which contain BASIC's compressed codes) or ASCII data files (which contain standard ASCII codes). In both cases, BASIC can access the file only 1 way—using sequential access.

When inputting or outputting to a BASIC program file, you need to use only 1 BASIC statement. For example:

```
SAVE "RAM:PROG"
LOAD "MDM:7E1E"
```

When inputting or outputting to an ASCII data file, you need to use a combination of BASIC statements:

1. Use the OPEN statement to open a file buffer for input, output, or appending to a file. (On startup, BASIC lets you use only 1 file buffer, but you can reset this with the MAXFILES statement.)

2. If outputting to a file, use either the PRINT # or PRINT # USING statement, depending on how you want to format the data. (See PRINT and PRINT USING for information on the 2 kinds of formats.)

If inputting from a file, use either the INPUT #, INPUT$, or LINE INPUT # statements depending how you want to input the data. (See INPUT, INPUT$, and LINE INPUT for information on the 3 ways of inputting data.) You may also need to use the EOF function to test whether you have reached the end of the file.

3. Use the CLOSE statement to close the file buffer.

This is an example of a program that outputs data to an ASCII file:

```
10 MAXFILES=1
20 OPEN "NAMES" FOR OUTPUT AS I
30 FOR I%=1 TO 10
40 INPUT "ENTER A NAME";A$
50 PRINT #1, A$;",";
60 NEXT I%
70 CLOSE #I
```

This is an example of a program that updates an ASCII file:

```
I0 MAXFILES=2
20 OPEN "NAMES" FOR INPUT AS 1
30 OPEN "UPDATE" FOR OUTPUT AS 2
40 IF EOF(1) THEN 100
50 INPUT #1, A$
60 PRINT A$
```

```
70 INPUT "PRESS ENTER OR ENTER
   NEW NAME";B$
80 IF B$<>"" THEN PRINT #2,
   B$;",";ELSE PRINT #2 A$;",";
90 GOTO 40
100 CLOSE 1,2
```

9/ BASIC Program Flow

BASIC executes the statements in a BASIC program sequentially. You can alter this program flow with these statements:

CALL   END   FOR/NEXT   GOSUB   GOTO
IF/THEN   ON GOTO   ON GOSUB   RESUME
RETURN   ON TIME$ GOTO
ON KEY GOTO   ON MDM GOTO
ON COM GOTO   ON ERROR GOTO

Interrupt-Handling Routines

BASIC lets you use the ON TIME$ GOSUB, ON KEY GOSUB, ON MDM GOSUB, and ON COM GOSUB to set an interrupt condition which causes BASIC to branch to an interrupt-handling subroutine.

For example, the statement ON TIME$ = "11:30:00" GOSUB 1000 sets an interrupt condition to occur when the time is 11:30—At 11:30, BASIC will go to the subroutine at line 1000.

Before BASIC can recognize an interrupt condition, you need to "turn on" the appropriate interrupt with the TIME$, KEY, MDM, COM statement. For example, TIME$ ON tells BASIC to start watching the time so that it can handle the interrupt set at 11:30.

You can also "turn off" or "stop" an interrupt using the same statements. For example, T1ME$ OFF tells BAS1C to quit watching the time. T1ME$ STOP tells BASIC to keep watching the time, but not to handle the 11:30 interrupt until it encounters another T1ME$ ON statement.

This is an example of a program using an interrupt-handling subroutine:

```
     10 ON T1ME$="20:00:00" GOSUB 1000
     20 TIME$ ON
     •
     •
   1000 T1ME$ = "19:00:00"
   1010 T1ME$ OFF
   1020 RETURN
```

The first time that the clock reaches 20:00:00, BAS1C jumps to line 1000, resets the clock, and returns to what it was doing before the subroutine call. The next time the clock reaches 20:00:00, nothing happens because the interrupt was disabled in line 1010.

Error Handling Routines

Another of the above statements—ON ERROR GOTO—causes BAS1C to interrupt program flow if an error occurs and goto an error-handling portion of the program. To return to the main portion of the program, you need to use the RESUME statement.

This is an example of a program using an error-handling routine:

```
   100 ON ERROR GOTO 1000
   •
   •
   200 X = 10000 / Y
   •
   300 X = 300 / Y
   •
   •
   1000 IF ERR<>11 THEN PRINT “Error
Code“;ERR;” in line ”:ERL : STOP ELSE
X=100000: RESUME NEXT
```

If an error occurs, BASIC jumps to line 1000. If the error is error 11 (division by zero), X is set to a high value, 100000, and execution returns to the line following the error line. If some other error occurs, BASIC prints out the message and stops.

10/ BASIC Keywords

ABS(*numeric expression*)

returns the absolute value of numeric expression.

    ABS(−5)

returns the number 5.

ASC(*string expression*)

returns the ASCII code for the first character in string expression. (See BASIC codes.)

ATN *(numeric expression)*

returns the arctangent of number (in radians). The resulting value ranges from $-\pi$ to π.

```
10 AN = ATN(.5)
```

sets AN to 0.46364760900081.

BEEP

causes the sound generator to beep for about 1/2 second.

```
10 BEEP
```

CALL *entry address, expression1, expression2*

calls a machine level subroutine beginning at *entry address. expression1* and *expression2* are optional; if used, Register A will contain *expression1* (a value from 0 to 255) and Register HL will contain *expression2* (a value from -32768 to 65535).

```
10 CALL 60000,10,VARPTR(A%)
```

calls a subroutine beginning at address 60000. Upon entry to the subroutine, register A contains 10, and register HL contains the address of the variable A%.

CDBL ***(numeric expression)***

converts the value of *numeric expression* to a double-precision number.

```
    10 A# = CDBL (A%)
```

If A% contains 344, then A# contains 344.

CHR$ ***(numeric expression)***

returns the ASCII character for the value of *numeric expression*. *numeric expression* must lie in the range of 0 to 255. CHR$ is the inverse of the function ASC. See the Appendices for a list of ASCII codes.

```
    10 PRINT CHR$(65)
```

prints the character A.

CINT ***(numeric expression)***

truncates the decimal portion of *numeric expression*. The resulting value must lie in the range -32768 to 32767.

```
    10 A% = CINT(45.67)
```

sets A% equal to 45.

CLEAR ***string space, high memory***

clears the values in all numeric and string variables and closes all open files. Also allocates memory for *string space* and sets *high memory* (the highest address BASIC can access). If you omit *string space*, BASIC allocates 256 bytes. If you omit *high memory*, BASIC uses all memory up to the top of RAM.

```
10 CLEAR
```

clears all variables, closes open files, sets the available string space to 256 bytes and releases all available memory to BASIC.

```
CLEAR 100,50000
```

clears all variables, closes open files, sets the available string space to 100 bytes, and sets 50000 as the highest memory address available to BASIC.

```
CLEAR 0
```

clears all memory.

CLOAD "*file*",R

clears the current BASIC program and loads *file*, a BASIC program, from cassette tape. If you omit *file*, BASIC loads the first BASIC program it finds. If **R** is used, BASIC executes the new program as soon as the load is complete.

```
CLOAD "ACCT",R
```

loads and runs the BASIC program ACCT stored on tape.

```
CLOAD
```

loads the first BASIC program found on the cassette tape.

CLOAD? *file*

compares *file* with the BASIC program currently in memory. If there are any differences, BASIC displays the message VERIFY FAILED; otherwise BASIC simply prints OK.

```
CLOAD? "ACCT"
```

compares the cassette file ACCT with the program currently in memory.

CLOADM *"file"*

loads the machine-code program called *file* from cassette into memory, at the addres specified when it was written to the cassette tape.

```
CLOADM "MEMTST"
```

loads the machine program MEMTST from the cassette.

CLOSE *file buffer*

closes the specified *file buffer*. If omitted, BASIC closes all open file buffers. (See OPEN.)

```
CLOSE 1, 2, 3
```

closes file buffers 1, 2, and 3.

CLS

clears the screen and moves the cursor to the upper-left corner.

```
CLS: PRINT "The old screen is gone!"
```

COM ON / COM OFF / COM STOP

turns on, turns off, or temporarily stops the ON COM interrupt. (See ON COM GOSUB.)

```
COM ON
```

turns on the ON COM interrupt.

CONT

resumes execution of a program after you have pressed (BREAK) or after BASIC has encountered a STOP statement in the program.

```
CONT
```

resumes execution of the BASIC program.

COS *(numeric expression)*

returns the cosine of angle given by *numeric expression*. You must give this angle in radians.

```
   10 Y = COS(60*0.01745329)
```

assigns Y the value 0.50000013094004.

CSAVE *"file"*,A

stores the current BASIC program on cassette tape using the specified *file* .A is optional; if used, BASIC saves the program as an ASCII file— Otherwise, BASIC stores the program as a BASIC program file.

```
   CSAVE "TANDY"
```

saves the current program on cassette tape as a compressed BASIC file under the name "TANDY."

```
   CSAVE "TANDY" ,A
```

saves the current program on cassette tape as an ASCII file.

CSAVEM *"file", start address, end address, entry address*

writes the machine-code program stored from *start address* to *end address* on cassette tape using the specified *file. entry address* is optional; if omitted, BASIC assumes that the program entry address is the same as the start address.

```
   CSAVEM "MEMTST" ,50000,50305,50020
```

writes the program stored from addresses 50000 to 50305 with the entry point at 50020 on cassette tape, giving the file the name "MEMTST."

CSNG *(numeric expression)*

returns the single-precision form of *numeric expression.*

```
    10 A! = CSNG(0.66666666666)
```

sets A! equal to 0.666667.

CSRLIN

returns the vertical position (line number) of the cursor where 0 is the top line and 5 is the bottom line.

```
    10 CLS: A% = CSRLIN
```

clears the screen and assigns A% the value 0.

DATA *constant list*

defines a set of constants (numeric and/or string) to be accessed by a READ command elsewhere in the program. See also READ and RESTORE.

```
    DATA 10,25,50,15,“Probabilities”,“Total”
```

stores the given values.

DATE$

returns the date. The date has the form MM/DD/YY.

```
    DATE$ = “11/02/84”
```

sets the date to November 02, 1984.

```
    PRINT DATE$
```

prints the date.

DAY$

returns the day. The day is a 3-letter abbreviation: ‘Mon'', “Tue'', “Wed'', “Thu'', “Fri'', “Sat'', or “Sun''.

```
DAY$ = “Fri''
```

sets the day to Friday.

```
PRINT DAY$
```

prints the day.

DEFDBL *letter list*

defines all the variables which begin with the letters in *letter list* as double-precision variables. *letter list* consists of individual letters and/or letter ranges of the form *letter1—letter2.*

```
100 DEFDBL D, X-Z
```

defines as double-precision all variables beginning with the letters D, X, Y, and Z.

DEFINT *letter list*

defines all the variables which begin with the letters in *letter list* as integer variables. *letter list* consists of individual letters and/or letter ranges of the form *letter1—letter2.*

```
120 DEFINT D, X-Z.
```

defines as integer type all variables beginning with the letters D, X, Y, and Z.

DEFSNG *letter list*

defines all the variables which begin with the letters in *letter list* as single precision variables. *letter list* consists of individual letters and/or letter ranges of the form *letter1—letter2.*

```
100 DEFSNG D, X-Z
```

defines as single precision all variables beginning with the letters D, X, Y, and Z.

DEFSTR ***letter list***

defines all the variables which begin with the letters in *letter list* as string variables. *letter list* consists of individual letters and/or letter ranges of the form *letter1* – letter2.

```
100 DEFSTR D, X-Z
```

defines as string all variables beginning with the letters D, X, Y, and Z.

DIM ***variable name (dimensions)***

defines *variable name* as an array with one or more dimensions. The number of dimensions you can use depends on the amount of available memory. To redimension an array, you must first use the CLEAR command to clear the array.

```
DIM A$(10), BAL%(10,10)
```

defines a string array, A$, which consists of 11 elements, A$(0) through A$(10), and an integer array, BAL%, which consists of 121 elements, BAL%(0,0) through BAL%(10,10).

EDIT ***line number range***

enters the TEXT program so that you can edit the specified lines. To return to BASIC, press F8.

```
EDIT
```

lets you edit the entire program.

```
EDIT 100-500
```

lets you edit lines 100 through 500

```
EDIT.
```

lets you edit the current line.

```
    EDIT 100-
```

lets you edit from line 100 to the end of the program.

END

terminates execution of the BASIC program. If omitted, BASIC executes up to the physical end of the program.

```
    END
```

EOF *(file buffer)*

checks to see if the file assigned to the specified *file buffer* has reached the end of the file. If so, EOF returns a -1; if not EOF returns a 0.

```
    IF EOF(1) THEN 200
```

checks the file assigned to buffer 1 for end of file. If it has reached the end of file, the program jumps to line 200.

ERL

returns the line number of the last error. If the last error is not from a program line but from a direct mode command, ERL returns the value 65535. ERL is useful in an error-handling routine. (See ON ERROR and ERR.)

```
    2000 IF ERR = 23 THEN RESUME ELSE
    PRINT “Error”;ERR; “in line”;ERL:
        STOP
```

If the error is an I/O error (ERR = 23), BASIC simply retries the I/O(RESUME). If there is some other error, such as a syntax error, BASIC displays “Error 2 in line 1000” and stops the program.

ERR

returns the error code number of the last error. ERR is useful in an error-handling routine. (See ON ERROR and ERL.)

```
    2000 IF ERR = 18 THEN PRINT "I/O Error"
        ELSE STOP
```

ERROR *numeric expression*

simulates the error specified by *numeric expression*. BASIC acts as if your program has committed the error. ERROR is useful in an error-handling routine. (See ON ERROR.)

```
    100 ERROR 10
```

prints DD Error in 100 and stops execution of the program.

EXP *(numeric expression)*

returns the exponential (or natural antilog) of *numeric expression*. *numeric expression* must be in the range +145.062860858624/−147.365445951624 or an overflow error occurs. EXP is the opposite of the function LOG.

```
    PRINT EXP(14)
```

prints 1202604.2841644, the natural antilog of 14.

FILES

causes BASIC to display all the files currently stored in RAM, without exiting BASIC. BASIC will display an asterisk (*) next to the program that is currently running.

FIX *(numeric expression)*

returns the whole number portion of *numeric expression*.

```
    10 A = FIX(1440.43)
```

sets A equal to 1440.

```
    I0 A + FIX(-33494123.4442)
```

sets A equal to -33494123.

FOR *variable=initial value* TO *final value* STEP *increment*
NEXT *variable*

executes the statements between the FOR and NEXT loop repetitively, from *initial value* to *final value* using the specified STEP *increment*. STEP *increment* is optional; if omitted, BASIC assumes STEP I.

```
    I0 FOR I=I0 TO 1 STEP -I
    20 PRINT I;
    30 NEXT I
```

prints the numbers 10 through 1.

FRE (0)

returns the current amount of unused memory.

```
    PRINT FRE(0)
```

FRE ("")

returns the current amount of unused string space.

```
    PRINT FRE("")
```

GOSUB *line number*

transfers program control to the subroutine beginning at *line number*. You must terminate the subroutine with a RETURN command.

```
    GOSUB 1000
```

goes to the subroutine beginning at line 1000.

GOTO *line number*

goes to the specified *line number*.

```
    GOTO 10
```

HIMEM

returns the top address of memory available to BASIC. You may change this value with the CLEAR statement.

```
    PRINT HIMEM
```

IF *relational* or *logical expression* THEN *command(s)* ELSE *command(s)2*

tests a relational or logical expression. If true, BASIC executes *command(s)1*. If false, BASIC executes *command(s)2*. ELSE *command(s)2* is optional; if omitted, BASIC assumes the ELSE clause is the next line.

```
    10 IF A < 90 THEN GOTO 100
```

tests A < 90. If true, BASIC goes to line 100; if false, BASIC continues with the next line.

```
    10 IF A = 10 OR A = 20 THEN B$ =
        "PAID" ELSE B$ = "Not Paid"
```

tests the condition A=10 OR A=20. If true, BASIC assigns B$ the string "Paid"; if false, BASIC assigns B$ the string "Not Paid."

INKEY$

returns the value of the key currently pressed, if any. If no key is pressed, the function returns a null character (“”). If you press an undefined function key, or the LABEL key, INKEY$ returns an ASCII 0 with a length of 1.

```
10 A$ = INKEY$: IF A$ = “” THEN 10
```

INP *(port number)*

returns a byte from the specified port. *port number* must be a numeric expression in the range of 0 to 255. INP is the complement function to the OUT command. (See the Tandy 102 Technical Reference Manual for information about ports.)

```
A% = INP(5)
```

sets A% equal to the byte value at Port 5.

INPUT *“prompt”;variable list*

awaits input from the keyboard and assigns the input to the variables in variable list. “prompt” is optional.

```
10 INPUT “Enter your name”;A$
```

prompts the operator with “Enter your name”; then assigns the value entered to A$.

INPUT *#file buffer,variable list*

inputs data sequentially from the file opened under *file buffer*. (See OPEN.)

```
10 INPUT #1,A$,B$,C
```

inputs values for A$, B$ and C from the file opened as file #1.

INPUT$ *(numeric expression)*

returns a string of *numeric expression* characters from the keyboard. *numeric expression* must be in the range of 1 to 255. INPUT$ does not display the input.

```
    A$ = INPUT$(5)
```

waits for the operator to input 5 characters and assigns this value to A$.

INPUT$ *(numeric expression,file buffer)*

Same as INPUT$ except input is from the specified *file buffer.*

```
    A$=INPUT$(5,1)
```

inputs the next 5 characters from file buffer 1.

INSTR *(start position, search string, match string)*

searches *search string* for *match string* and, if found, returns the position of *match string*; otherwise, returns a 0. *start position* is optional; if omitted, INSTR starts the search at position 1.

```
    PRINT INSTR("dimethylsulfate","sulfate")
```

displays 9 ("sulfate" starts at position 9).

INT *(numeric expression)*

returns the whole number representation of *numeric expression*, not greater than *numeric expression.*

```
    A# = INT(214441113.443)
```

sets A# equal to 2144433113.

```
    A# = INT(-214.995)
```

sets A# equal to -215.

IPL *"file"*

defines a RAM file named *file* as the startup program. After executing this command, the program named *file* runs whenever you turn on the computer. IPL executes properly only if the computer is turned off while in BASIC.

```
IPL "TIMSET.BA"
```

To disable auto startup type IPL (ENTER).

KEY *function key, string expression*

defines *function key* as *string expression*. *string expression* must be 15 or less characters.

```
KEY 6,"PRINT TIME$" + CHR$(13)
```

defines function key 6 as PRINT TIME$ followed by a carriage return. Now whenever you press function key 6, BASIC returns the time. (See also ON KEY and KEY ON.)

To reset the function keys to the cold start default, you must call two subroutines:

```
CALL 23164,0,23366
CALL 27795
```

These calls reset the function keys to their original value.

KEY *(function key)* ON/OFF/STOP

enables, disables, or temporarily stops the ON KEY interrupt. (See ON KEY GOSUB.)

```
100 KEY (2) ON
```

enables function key 2.

```
100 KEY ON
```

enables all function keys.

```
100 KEY (4) OFF
```

disables function key 4.

KEY LIST

lists on the display the current definitions for the function keys in the format:

```
    key 1          key 2
    key 3          key 4
    key 5          key 6
    key 7          key 8
```

KILL *"file"*

deletes a RAM file. You *must* include the file's extension.

```
    KILL "BILLS.BA"
```

deletes the RAM file BILLS.BA.

> ***Note:** If you have 200 bytes or less of free memory, KILL may not delete a file. If this situation occurs, delete program lines manually or go to TEXT, select a file, and put it in the PASTE buffer. Then return to BASIC and KILL the unwanted files.*
>
> *Also, please note that you cannot kill a BASIC program while it is in the work area.*

LCOPY

prints the text on the display. LCOPY ignores non-text data.

LEFT$ *(string expression, portion)*

returns the left *portion* of *string expression*. *portion* is a numeric expression.

```
    10 AC$ = LEFT$("817552161",3)
```

.sets AC$ to "817".

LEN *(string expression)*

returns the number of characters in *string expression*.

```
    PRINT LEN("HELLO")
```

prints 5.

LET *variable name=value*

equates a *variable name* with *value*. The word LET can be omitted; it is included to be compatible with older forms of BASIC.

```
    LET A$ = "The"
```

assigns "The" to A$.

```
    A$ = "The"
```

assigns "The" to A$.

LINE *(x1,y1)—(x2,y2), switch,* BF

draws a line from coordinates *x1,y1* to *x2,y2*. (See "Graphic Screen Locations" in "BASIC Input/Output".) If *(x1,y1)* is omitted, BASIC starts the line from the *x,y* coordinates of the last LINE command, or from 0,0 if this is the first LINE command.

switch tells BASIC whether to set or reset the pixels: odd values tell BASIC to set the pixels; even values tell BASIC to reset the pixels. If omitted, BASIC uses set.

B tells BASIC to draw a box, rather than a line. BF tells BASIC to fill in the box. Both B and BF require that you specify *switch*.

```
    10 LINE (20,20)—(50,63)
    20 LINE—(30,30)
```

draws lines from (20,20) to (50,63), and from (50,63) to (30,30).

```
10 LINE (20,20)—(50,63) ,0
```

resets (erases) all points on a line from (20,20) to (50,63)

```
10 LINE (0,0)—(239,63) ,1,B
```

draws a box with corners at (0,0) and (239,63).

```
10 LINE (0,0)—(239,63) ,1,BF
```

draws a box with corners at (0,0) and (239,63) and then sets all the points inside the box.

LINE INPUT *“prompt”; string variable*

awaits for the operator to enter a line of data from the keyboard; then, when the operator presses the carriage return (ENTER), assigns that string to *string variable. prompt* is optional.

```
10 LINE INPUT “ENTER NAME AND
   ADDRESS:”;NA$
```

displays “ENTER NAME AND ADDRESS” and waits for the operator to enter this information. If the operator enters “John “Rocky” Smith, 5641 Lancaster, East Pearoe, Ohio‘, this entire value is assigned to NA$.

LINE INPUT# *file buffer, string variable*

reads the next line (all text up to the carriage return) from the specified *file buffer* and assigns the value of this line to *string variable.*

```
LINE INPUT #1,Z$
```

reads the next line from the file assigned to buffer #1, and assigns the value of this line to Z$.

LIST *lines*

displays the specified program *lines*.

```
LIST
```

displays the entire program.

```
LIST 100-300
```

displays from line 100 to line 300.

```
LIST .-
```

displays from the current line to the end of the program. (The period (.) represents the current line.)

LLIST *lines*

Same as list except output is to the *printer*. (See also LIST.)

```
LLIST
```

LOAD *"file"*, R

loads a BASIC program file from RAM, cassette, the RS-232, or the modem. If you omit the device in the file specification, BASIC assumes RAM.

```
LOAD "RAM:TIMSET"
```

loads the BASIC program TIMSET.BA from RAM.

```
LOAD "CAS:ACCT" ,R
```

loads and runs the BASIC program ACCT from cassette tape.

```
LOAD "COM:78N1E"
```

loads a BASIC program from the RS-232C line, using 4800 baud, 8-bit words, no parity, 1 stop bit, and start/stop enabled.

```
LOAD "MDM:702E" ,R
```

loads a BASIC program from the modem, using 7-bit words, odd parity, 2 stop bits, and start/stop enabled.

LOADM *"file"*

loads a machine-language program *file* from RAM or cassette tape. If you omit the device in the file specification, BASIC assumes RAM. When BASIC loads the file, it prints out its start address, end address, and entry point, if any.

```
LOADM "MEMTST"
```

loads the machine-language program called MEMTST.CO from RAM.

```
LOADM "CAS:MEMTST"
```

loads the machine-language program called MEMTST from cassette tape.

LOG *(numeric expression)*

returns the natural logarithm (base "E") of *numeric expression. numeric expression* must be greater than zero.

```
10 A = LOG(10)
```

sets A equal to 2.302585092994.

LPOS (0)

returns the current position of the printer print head within the printer buffer.

```
LPRINT "Printer head positon:"LPOS(0)
```

prints the message followed by the number.

LPRINT *expression list*

same as print except output is to the line printer. (See PRINT.)

```
LPRINT "The total for ";A$;" was "; TT
```

If A$ contains the string April and TT contains the value 1332.44, this statement prints: The total for April was 1332.44

```
LPRINT X,Y,Z
```

prints the value of X beginning in column 0, Y in column 14, and Z in column 28.

```
LPRINT X,,,Z
```

prints the value of X beginning in column 0, and Z in column 42 (two columns are skipped because of the two commas.)

LPRINT USING *"format string";expression list*

same as PRINT USING except output is to the line printer. (See PRINT USING.)

MAXFILES

stores the maximum number of file buffers that you can have open at the same time. On startup, MAXFILES equals 1.

```
10 MAXFILES = 5
```

sets MAXFILES to 5.

```
PRINT MAXFILES
```

prints the current value of MAXFILES.

MAXRAM

contains the memory size of Tandy 102.

```
CLEAR 1000,MAXRAM
```

clears 1000 bytes for string storage and sets the high memory to the maximum amount for the Tandy 102.

MDM ON/OFF/STOP

enables, disables, or stops the ON MDM interrupt. (See ON MDM.)

```
10 MDM ON
```

enables the ON MDM interrupt.

MENU

exits BASIC and returns to the Tandy 102 Main Menu. If you are editing a current RAM file, BASIC rewrites the file before returning to the Menu.

```
MENU
```

MERGE *"file"*

loads a BASIC program stored as an ASCII file from RAM, cassette tape, the RS-232, or the modem, and merges it with the current program. If BASIC finds a duplicate line number, the line from *file* replaces the current line. If you omit the device from the file, BASIC assumes RAM.

```
MERGE "RAM:ACT.DO"
```

loads ACT.DO from RAM and merges it with the current program.

```
MERGE "CAS:ACCT"
```

loads ACCT from cassette tape and merges it with the current program.

```
    MERGE "COM:78E1E"
```

loads the the file coming in on the RS-232C line using the TELCOM parameter settings of "78E1E" and merges it with the current program.

MID$ *(string expression,position,length)*

returns a portion of *string expression* that starts at the specified *position* and continues for the specified *length*. *length* is optional.

```
    10 HASH$ = MID$(A$,2,2)
```

If A$ contains the string 003449953, then this statement assigns string 03 to HASH$.

MID$ *(string expression1,position,length)= string expression2*

replaces characters of *string expression1*, starting at *position*, with *string expression2*. *length* is optional and, if present, it is ignored.

```
    10 MID$(A$,5) = "FF"
```

If A$ contains the string 00000000, this statement changes A$ to 0000FF00.

```
    1000 MID$(A$,4) = "ABCDEF"
```

If A$ contains the string 000ABCDE, this statement changes A$ to 000ABCD.

MOTOR ON or OFF

turns on or off the cassette recorder motor.

```
    MOTOR ON
```

turns on the cassette recorder motor.

NAME "RAM:*old file*" AS "RAM:*new file*"

renames a RAM file. You must include the extensions in the files.

```
NAME "ACCTS.DO" AS "OLDACT.DO"
```

renames the RAM file ACCTS.DO to OLDACT.DO.

NEW

erases the current program, sets numeric variables equal to zero, and sets string variables equal to null(""). NEW does not change the string space allocation.

```
NEW
```

ON COM GOSUB *line number*

tells BASIC to go to the subroutine at *line number* when it receives data from the RS-232. The COM interrupt must be on. (See COM ON.)

```
10 ON COM GOSUB 1000
20 COM ON
•
•
1000 OPEN "COM:78N1E" FOR INPUT AS
I
I0I0 OPEN "1MPDAT.DO" FOR OUTPUT
   AS 2
I020 LINE INPUT -1, A$
I030 PRINT -2, A$
I040 IF NOT EOF(I) THEN GOTO 1020
1050 CLOSE 1,2
1060 RETURN
```

When data comes in on the RS-232C line, control transfers to line 1000, where it copies the input into a RAM file called "IMPDAT.DO".

ON ERROR GOTO *line number*

tells BASIC to go to an error-handling routine at *line number* when an error occurs. To return from the error-handling routine, use RESUME.

```
    100 ON ERROR GOTO 1000
```

when an error occurs, goes to Line 1000.

ON *numeric expression* GOSUB *line number list*

goes to subroutine starting at the *n*th line number. *n* is specified by the numeric expression.

```
    10 ON X GOSUB 100,200,300
```

calls the subroutine beginning at line 100, 200, or 300, if X equals 1, 2, or 3, respectively.

ON *numeric expression* GOTO *line number list*

goes to the *n*th line number. *n* is specified by the numeric expression.

```
    10 ON X GOTO 100,200,300
```

branches to 100, 200, or 300, if X equals 1, 2, or 3, respectively.

ON KEY GOSUB *line number list*

tells BASIC to go to a subroutine beginning at one of the *line numbers* when a function key is pressed. The function key interrupt must be on. (See KEY ON.)

10 ON KEY GOSUB 1000,2000,3000,,5000
tells BASIC to go to the following subroutines if a function key is pressed: Function Key 1—Line 1000, Function Key 2—Line 2000, Function Key 3—Line 3000, Function Key 4 —not defined in this statement, Function Key 5—Line 5000, Function Key 6, 7, and 8—not defined in this statement.

ON MDM GOSUB *line number*

tells BASIC to go to a subroutine at *line number* when it receives data over the modem line. The modem interrupt must be on. (See MDM ON.)

10 ON MDM GOSUB 1000
defines a modem interrupt routine beginning at line 1000.

ON TIME$ = *"time"* GOSUB *line number*

tells BASIC to go to a subroutine starting at *line number* when TIME$ = *time*. *time* is a string expression of the form HH:MM:SS. The TIME$ interrupt must be enabled. (See TIME$ ON.)

10 ON TIME$ = "14:20:00" GOSUB 1000
tells BASIC to go to Line 1000 at 2:20PM (14:20:00).

OPEN *"file"* FOR *mode* AS *file buffer*

opens a file buffer for accessing a *file* in RAM, cassette tape, the RS-232, the modem, the screen, or the line printer using any of these *modes*: OUTPUT-sequential output, starting at the file's beginning INPUT—sequential input, starting at the file's beginning APPEND—sequential output, starting at the file's end.

```
10 OPEN "RAM:ACCT.DO" FOR APPEND
AS 1
```

opens a RAM file called ACCT.DO for appending, and assigns it the file buffer 1.

```
10 OPEN "CAS:" FOR OUTPUT AS 3
```

opens an output file on cassette and assigns it to file buffer 3.

```
10 OPEN "MDM:6E1E" FOR INPUT AS 4
```

opens a modem file for input as file buffer 4, using the "6E1E" TELCOM parameters.

```
10 OPEN "LCD:" FOR OUTPUT AS 1
```

opens a screen file as file buffer 1.

OUT *port number, byte value*

outputs *byte value* to *port number*. *port number* and *byte value* are numeric expressions in the range 0 to 255. (See the Tandy 102 Technical Manual for information about ports.)

```
10 OUT 55, 100
```

outputs 100 to CPU port 55.

PEEK *(memory address)*

returns the byte value stored at *memory address*. *memory address* and the returned value are both in decimal form.

```
10 A$ = PEEK(16999)
```

assigns the byte value at address 16999 to A%.

POKE *memory address,byte value*

loads *memory address* with *byte value*. Both must be expressed as decimal numeric expressions.

```
100 POKE 60000, 104
```

loads 104 into address 60000.

POS *(dummy numeric expression)*

returns the current horizontal screen position of the cursor.

```
100 OP% = POS(0)
```

assigns OP% the current horizontal cursor position.

POWER *numeric expression*

changes the Tandy 102's automatic power down period to *numeric expression* X 0.1 minutes. (See also the *Tandy 102 Owner's Manual.*)

```
10 POWER 10
```

resets the automatic power down period to one minute (10 X 0.1).

POWER CONT

disables the automatic power down feature of the Tandy 102.

```
10 POWER CONT
```

POWER OFF,RESUME

immediately turns off the power. RESUME is optional; if present when you turn the power back on, the Tandy 102 resumes execution of the program at the statement following the POWER OFF,RESUME. If not present, the Tandy 102 returns to the Main Menu upon power up.

```
    10 IF TIME$>"11:30:00" THEN POWER
    OFF
```

turns off the power if the clock is past 11:30 A.M.

PRESET *(x-coordinate,y-coordinate)*

turns off the LCD pixel at *(x-coordinate,y-coordinate)*. *x-coordinate* may range from 0 to 239, and *y-coordinate* may range from 0 to 63. (See also PSET.)

```
    10 PRESET (55,10)
```

turns off the pixel at (55,10).

PRINT *expression list*

prints *expression list* on the display. You can separate the data in *expression list* with a comma or a semi-colon. A comma causes BASIC to move to the next print zone, a semi-colon causes it to stay in the same position. No punctuation at the end of *expression list* causes BASIC to move to the next line.

BASIC prints positive numbers with leading blanks, all numbers (positive and negative) with trailing blanks, and strings without any leading or trailing blanks.

PRINT "JOHN", "DOE"

prints JOHN DOE on the display; then moves the cursor to the next line.

PRINT "JOHN";",";"DOE";

prints JOHN,DOE on the display and leaves the cursor in the position immediately following the E in DOE.

PRINT @ *screen position, expression*

prints expression at the specified screen position. (See BASIC Input/Output.)

PRINT @ 140, "MENU"

prints "MENU" at screen position 140.

PRINT *#file buffer, expression list*

same as PRINT except the output is to a file.

200 PRINT #1,"JOHN";",";"DOE";

prints JOHN,DOE to file buffer #1 and leaves the file pointer at the position immediately after the E in DOE.

PRINT *#file number,* USING *"format"; expression list*

same as PRINT USING except output is to a file buffer. (See PRINT USING.)

PRINT USING *"format";expression list*

prints the data in *expression list* using the specified *format*. The data in *expression list* may be separated either by commas or semi-colons.

format consists of *field specifiers* which describe the type and the format to use in printing the data. If there is more data in *expression list* than *field specifiers*, BASIC reuses the *field specifiers*.

The string *field specifiers* are:

"!" prints the first character in a string.

```
PRINT USING "!":"Tandy" T
```

"\n\\" prints *n*+2 characters in a string. *n* is any number of blank spaces. (To enter the "\\", press GRAPH –.)

```
PRINT USING "\  \";"Tandy" Tand
```

The numeric *field specifiers* are:

prints a number, right justified, in the specified digit positions. If the number is larger than the field, BASIC precedes the number with %.

```
PRINT USING "#####";5
         5
```

+ inserts the algebraic sign of a number.

```
PRINT USING "+#####";-13
       -13
PRINT USING "##### +";14
       14 +
```

– if negative, inserts a minus sign in a number; if positive, inserts a blank space in a number.

```
PRINT USING "-#####";14
       -   14
PRINT USING "#####.## -";-0.45
       -0.45
```

** replaces a number's leading spaces with asterisks.

```
PRINT USING "**#####";145
****145
```

$$ precedes a number with a dollar sign.

```
PRINT USING "$$#####";450
          $450
```

**$ precedes a number with asterisks and then a dollar sign.

```
PRINT USING "**$###";12
***$12
```

• inserts a decimal point in a number.

```
PRINT USING "#####.##";14.5
          14.50
PRINT USING "#####.##";0.588
          0.59
```

, inserts commas in a number.

```
PRINT USING "#########,";14432
          14,432
```

^^^^ prints number in exponential format. (To enter "^", press SHIFT 6.)

```
PRINT USING "###^^^^";
150000
          15E + 04
```

PSET ***(x-coordinate,y-coordinate)***

turns on the graphics pixel at specified x,y coodinate. (See "Graphic Screen Coordinates" in the "BASIC Input/Output Section.")

```
   10 PSET (40,45)
```

turns on the pixel at 40,45.

READ *variable list*

reads the next constants in a DATA statement and assigns them to the variables in *variable list*. (See also DATA and RESTORE.)

```
100 DATA 0,4, 0.2 “Trinity River”
120 READ A,B%,C$
```

assigns A the value 0.4, B% the value 0.2, and C$ the string Trinity River.

REM *comment statement*

tells BASIC that the remainder of the line is a comment. You may abbreviate REM with an apostrophe. If the comment follows another BASIC command, you must either use the apostrophe or precede REM with a colon.

```
10 REM This program finds the standard
   deviation
10 ’ This program finds the standard
   deviation
100 AVE = SUM / TT ’Calculate the average
100 AVE = SUM / TT :REM Calculate the
average
```

RESTORE *line number*

resets the DATA statement pointer to the first item in the DATA statement on *line number*. *line number* is optional; if omitted, BASIC uses the first DATA statement. (See also DATA and READ.)

```
100 DATA “Nuts”, “Bolts”, “Screws”,
   “Hammers”
•
•
300 READ ITEM$(1),ITEM$(2),ITEM$(3),
   ITEM$(4)
•
•
600 RESTORE 100
610 READ CT$(1),CT$(2),CT$(3),CT$(4)
```

Line 300 assigns the strings of the DATA statement in line 100 to ITEM$ 1 through 4. Line 600 resets the DATA pointer so that line 610 reassigns the strings to CT$ 1 through 4.

RETURN

ends subroutine and returns to the statement immediately following the last GOSUB statement.

```
RETURN
```

RESUME *line number*

ends an error handling routine by branching to *line number* where BASIC begins normal execution. If *line number* is omitted, BASIC returns to the line which caused the error. You can specify NEXT as the line number, in which case BASIC returns to the line immediately following the error-causing line.

```
1000 IF ERR = 18 THEN PRINT @0,
“Printer Not Ready!!!”:RESUME
```

If an I/O error occurs, BASIC prints the message and resumes execution at the offending statement.

```
RESUME NEXT
```

BASIC proceeds to the next statement.

RIGHT$ *(string expression,portion)*

returns the right *portion* of *string expression*. *portion* is a numeric expression.

```
10 SEC$ = RIGHT$(TIME$,2)
```

assigns the current second count to SEC$.

RND *numeric expression*

returns a pseudo-random number between 0 and 1. If *numeric expression* is non-zero, RND returns a new random number. If *numeric expression* equals 0, RND returns the last random number generated.

```
20 PRINT RND(1)
30 PRINT RND(0)
```

prints the same random number twice.

RND always generates the same random number series. If your application requires a different random number starting the sequence each time, you can use the clock to establish a starting point in the sequence. For example, the following routine points the random number generator to one of 60 starting points in the generator:

```
10 SEC = VAL(RIGHT$(TIME$,2))
20 FOR I = 1 TO SEC
30 DUMMY = RND(1)
40 NEXT I
```

RUN *line number*,R

clears all variables, closes all open files, and executes the current program, starting at *line number*. *line number* is optional; if omitted, BASIC starts execution at the first line of the program. R is also optional; it tells BASIC to leave current files open.

```
RUN 100
```

clears all variable values and starts executing the program at line 100.

RUN *"file"*,R

same as RUN, except the program is loaded from the specified file before BASIC runs it.

```
1000 RUN "PART2.BA",R
```

loads and executes the RAM file PART2.BA, keeping all open files open.

```
100 RUN "MDM:7E2E"
```

loads and executes the BASIC program coming in over the modem lines.

RUNM *"file"*

closes all open files; then loads and executes file, an executable machine-code program stored in RAM or cassette tape. If the file does not include a device specification, RAM is assumed.

RUNM “MEMTST”

loads the program MEMTST.CO from RAM and executes it.

RUNM “CAS:”

loads and runs the first machine-language program found on the cassette tape.

SAVE *“file”*,A

writes the current BASIC program to a file in RAM, cassette tape, the RS-232, the modem, the screen, or the printer. A is optional; if used, the program is saved as an ASCII data file.

SAVE “TIMSET”

writes the current BASIC program to the RAM file TIMESET.BA.

SAVE “PART3” ,A

writes the current BASIC program to the RAM file PART3.DO. The file is stored in ASCII format.

SAVE “CAS:CLOCK”

writes the current program to cassette tape naming the file CLOCK (identical to the command CSAVE“CLOCK”).

SAVE “MDM:7N1E”

sends the current program out the modem, using the configuration 7 bit words, no parity check, 1 stop bit, and stop/start enable.

(You do not need to save the changes that you make to a program stored in RAM. BASIC automatically does this for you.)

SAVEM "*file, start address, end address, entry address*

writes the machine-code program stored from *start address* to *end address* to cassette tape or RAM under the name *file. entry address* is optional; if not present, BASIC assumes the program *entry address* is the same as the *start address.*

SAVEM "CAS:MEMTST",50000,50305,50020

writes the program stored from addresses 50000 to 50305 with the entry point at 50020 to cassette tape, giving the file the name MEMTST.

SAVEM "MEMTST",50000,50305,50020

writes the program stored from addresses 50000 to 50305 with the entry point at 50020 to RAM, giving the file the name MEMTST.CO.

SCREEN *on/off*

locks or unlocks the bottom (LABEL) line on the display for scrolling. *on* is 0,0 and *off* is 0,1.

SCREEN 0,0

causes LABEL line to disappear and allows you to scroll with all eight lines.

SCREEN 0,1

causes LABEL line to reappear.

SGN *(numeric expression)*

returns a -1 for negative numbers, 0 for zero, and 1 for positive numbers.

200 TTL = 10 * SGN(CR)

sets TTL equal to either 10, 0, or -10, depending on whether CR is positive, zero, or negative.

SIN *(numeric expression)*

returns (in radians) the trigonometric sine of *numeric expression.*

```
    100 Y = SIN(1.5)
```

assigns Y the value 0.99749498660406.

SOUND *pitch,length*

"plays" a given pitch for the given length. *length* ranges from 0 to 255. Dividing length by 50 gives the approximate length in seconds. *pitch* ranges from 0 to 16383, with the smaller values corresponding to higher pitches. (See "Sound Frequencies" in the "BASIC Input/Output" section for the frequencies to use for musical notes.)

SOUND ON or OFF

Turns on or off the beep BASIC uses when: (1) You load a file from cassette, and (2) the Tandy 102 is waiting for a carrier signal from the telephone modem lines. This statement has no affect on the BEEP or SOUND statement.

SPACE$ *(length)*

returns a string of *length* spaces.

```
    100 B$ = SPACE$(20) + A$
```

sets B$ equal to a string of 20 spaces followed by the string stored in A$.

SQR *(numeric expression)*

returns the square root of *numeric expression*. *numeric expression* must be a positive number.

```
    I0 C = SQR(A^2 + B^2)
```

sets C equal to the square root of the sum of A^2 and B^2.

STOP

stops execution of a BASIC program. You can continue execution with the CONT statement. STOP and CONT are useful for debugging a program.

```
    100 STOP
```

stops execution at line I00.

STR$ *(numeric expression)*

converts *numeric expression* to its string representation. This function is the inverse of VAL.

```
    B$ = "$" + STR$(BAL) + ".00"
```

If BAL contains the value I33, this statement sets B$ equal to $ 133.00.

STRING$ *(n,character)*

returns a string in which *character* is repeated *n* times. *n* can be between 0 to 255. *character* can be a string or an ASCII code. ("See BASIC Codes.")

```
    PRINT STRING$(20,"*")
```

prints a string of 20 asterisks.

```
    PRINT STRING$(40,239)
```

prints a string of 40 solid blocks (239 is the ASCII code for a solid block.)

TAB *(numeric expression)*

skips *numeric expression* spaces before printing the next data item. *numeric expression* ranges between 0 and 255.

```
   10 PRINT TAB(30);"Table 1"
```

prints "Table 1" starting in column 30.

```
   20 LPRINT TAB(10);"Total";TAB(20);
      "Number";TAB(30);"Balance"
```

skips 10 spaces and prints Total on the printer, skips another 20 spaces and prints Number, and finally skips another 10 spaces and prints Balance.

TAN *(numeric expression)*

returns the tangent of *numeric expression*. *numeric expression* must be in radians.

```
   10 SLOPE = TAN(THETA)
```

assigns SLOPE the value of the tangent of THETA.

TIME$

sets or returns the time, using the format HH:MM:SS.

```
   TIME$ = " 10:00:00"
```

sets the time to 10:00 AM.

```
   PRINT TIME$
```

prints the current time.

TIME$ ON / TIME OFF / TIME STOP

turns on, turns off, or stops the ON TIME$ GOSUB interrupt. (See "BASIC Program Flow.")

```
   10 ON TIME$="20:00:00" GOSUB 1000
```

At 8:00 P.M., BASIC goes to the subroutine at Line 1000.

VAL *(string expression)*

converts *string expression* to a numeric representation of the string. If *string expression* contains non-numeric characters, VAL returns only the value of the leading number, if any. VAL is the inverse of the function STR$.

```
   5 B$ = “100.44824
  10 A = VAL(B$)
```

sets A equal to 100.44824.

```
   5 B$ = “no balance”
  10 A = VAL(B$)
```

sets A equal to 0.

```
   5 B$ = “3.00313354E33”
  10 A = VAL(B$)
```

sets A equal to 3.00313354 X 10^{33}.

VARPTR *(file buffer)*

returns the memory address that points to the first byte of data in *file buffer*.

```
  LINK=VARPTR(1)
```

returns the first address of the data stored in file buffer 1.

VARPTR *(variable)*

returns a memory address that points to an individual *variable*: either a simple variable or a subscripted variable.

If *variable* is numeric, this address points to the actual variable. If *variable* is string, this address points to the variable's string descripter.

```
LINK=VARPTR(A%)
```

sets LINK equal to the first address of A%.

```
LINK=VARPTR(A$(1))
```

sets LINK equal to the first address of the string descripter which points to of array element A$(1).

11/ Machine-Code Calls

BASIC includes statements and functions that you can use to call machine-code routines. These statement and functions are for technical applications.

Calling a Machine-Code Routine

BASIC lets you call a Tandy 102 machine-code routine stored in ROM or your own machine-code routine.

To call a ROM machine-code routine:

Use the CALL statement to call the routine at the specified address. For the addresses of the ROM routines, you need to purchase the *Tandy 102 Technical Manual*.

To call your own machine-code routine:

1. Use the CLEAR statement to reserve an area in high memory that BASIC cannot destroy.

2. Insert a machine-code routine into this area of high memory. You can do this in 2 ways: (a) by using the BASIC POKE statement to directly insert the machine-code routine into high memory, or (b) by using the BASIC CLOADM statement to load a preassembled machine-code routine into high memory. (To preassemble a routine, you need a Model 100 or a Tandy 102 assembler product.)

3. Use the CALL statement to jump to this area of high memory that contains the machine-code routine.

4. Return from the machine code routine by using the following machine-code instruction:

```
RTS
```

Passing Values to a Machine-Code Routine

The CALL statement lets you pass two values to a machine-code routine. The first value must be in the range of 0-255; you can use this value to pass a character's code to the routine.

The second value must be in the range of -32768 to 65535; you can use this value to pass the address of a variable or a file buffer to the routine. (You can obtain this address by using the BASIC VARPTR statement.)

Before returning from the routine, you can pass a value back to BASIC. To do so, insert this value in the address of a BASIC variable or in the address of a BASIC file buffer.

(See also, "BASIC Codes," the VARPTR statement and "BASIC Variable Storage.")

12/ BASIC Variable Storage

Integer Variables

BASIC stores integer variables in 2 bytes of memory using two's complement notation: The first byte is the least significant byte (LSB); the second, the most significant byte (MSB).

With two's complement notation, the highest bit of the MSB is indicates the sign of the number. If this bit is 1, the number is negative; if this bit is 0, the number is positive.

If the number is negative it is stored as its binary inverse plus 1. BASIC calculates a number's inverse by changing all the bits that are 1's to 0's and all the bits that are 0's to 1's.

For example, this is how BASIC stores the integer 513:

Byte	Binary	Decimal	Meaning
0	00000001	1	1
1	00000010	2	512

This is how BASIC stores the integer -513

Byte	Binary	Decimal	Meaning
0	11111111	255	the binary inverse of 1 plus 1
1	11111101	253	the binary inverse of 512

Single and Double Precision Variables

BASIC stores single- and double-precision numbers in 4 bytes (single-precision) or 8 bytes (double-precision) of memory using floating-point notation.

With floating point notation, BASIC converts a number to a mantissa and an exponent. The mantissa is the significant digits in the number represented as a decimal fraction. For example, the mantissa of -51.25 is .5125.

The exponent is whatever power of 10, when multiplied by the mantissa, will produce the number's actual value. For example, the exponent of -51.25 is 2. (.5125 times 10 to the power of 2 equals -51.25).

The first byte of a single- or double-precision variable contains the number's sign (in bit 7) and exponent (in bits 0-5). The 6th bit of this byte always contains a 1.

For example, the number -51.25 is a negative number with an exponent of 2. BASIC uses a binary 11000010 to store this information.

The remaining bytes contain the variable's mantissa stored in binary-coded decimal (BCD) notation. BCD format uses 4 bits to store each mantissa digit.

For example, the first two digits of the mantissa of -51.25 is 51. BASIC uses binary 01010001 to store this information. (Binary 0101 represents the digit 5; 0001 represents the digit 1).

The number -51.25 is stored as follows:

Byte	Binary	Decimal	Meaning
0	11000010	194	negative number exponent = 2
1	01010001	81	digits 51
2	00100101	37	digits 25
3	00000000	0	digits 00

String Variables

BASIC uses a 3-byte string descriptor to indicate where in memory a string is stored. The string descriptor contains:

Byte	Meaning
0	Length of the string
1	LSB of string address
2	MSB of string address

Array Variables

BASIC stores arrays in memory using an array descriptor, which is immediately followed by each of the array elements. The array descriptor is in this format:

Byte	Meaning
0 and 1	ASCII code for the array name
2 and 3	Length of the array
4	Number of dimensions in the array
5 and 6	Number of elements in first dimension
7 and 8	Number of elements in the second dimension

BASIC reserves bytes 7 and 8 for the number of elements in the second dimension, even if the array has only 1 dimension. If the array has 3 or more dimensions, each of the next groups of 2 bytes store the number of elements in each additional dimension.

For example, if the array has 4 dimensions, bytes 9 and 10 contain the number of elements in the third dimension, and bytes 11 and 12 contain the number of elements in the fourth dimension.

The array descriptor is followed immediately by each of the array elements. For example, in a 2-dimensional array, Byte 9 contains the first byte of the first element in the array.

BASIC stores the array elements in the same way it stores simple variables. For example, in a 2-dimensional integer array, Bytes 9 and 10 contains two's complement notation of the first element; in a 2-dimensional string array, Bytes 9-11 contain the string descriptor of the first element.

13/ BASIC Codes

These tables list the codes that you can use with the CHR$ and ASC statements in a BASIC program. For example, you could use PRINT CHR$(128) to display a small telephone, PRINT CHR$(27);"A" to move the cursor up 1 line, and PRINT CHR$(ASC("A")+32) to display the letter "a".

Decimal	Hex	Binary	Printed Character	Keyboard Character
0	00	00000000		CTRL @
1	01	00000001		CTRL A
2	02	00000010		CTRL B
3	03	00000011		CTRL C
4	04	00000100		CTRL D
5	05	00000101		CTRL E
6	06	00000110		CTRL F
7	07	00000111	bell	CTRL G
8	08	00001000	backspace	CTRL H
9	09	00001001	tab	CTRL I
10	0A	00001010	line feed	CTRL J
11	0B	00001011	cursor home	CTRL K
12	0C	00001100		CTRL L
13	0D	00001101	carriage return	CTRL M
14	0E	00001110		CTRL N
15	0F	00001111		CTRL O
16	10	00010000		CTRL P
17	11	00010001	XON	CTRL Q
18	12	00010010		CTRL R
19	13	00010011	XOFF	CTRL S
20	14	00010100		CTRL T
21	15	00010101		CTRL U
22	16	00010110		CTRL V
23	17	00010111		CTRL W
24	18	00011000		CTRL X
25	19	00011001		CTRL Y
26	1A	00011010	EOF	CTRL Z

Decimal	Hex	Binary	Printed Character	Keyboard Character
27	1B	00011011		ESC
28	1C	00011100		→
29	1D	00011101		←
30	1E	00011110		↑
31	1F	00011111		↓
32	20	00100000		SPACEBAR
33	21	00100001	!	!
34	22	00100010	"	"
35	23	00100011	#	#
36	24	00100100	$	$
37	25	00100101	%	%
38	26	00100110	&	&
39	27	00100111	'	'
40	28	00101000	(	(
41	29	00101001	)	)
42	2A	00101010	*	*
43	2B	00101011	+	+
44	2C	00101100	,	,
45	2D	00101101	–	–
46	2E	00101110	.	.
47	2F	00101111	/	/
48	30	00110000	0	0
49	31	00110001	1	1
50	32	00110010	2	2
51	33	00110011	3	3
52	34	00110100	4	4
53	35	00110101	5	5
54	36	00110110	6	6

Decimal	Hex	Binary	Printed Character	Keyboard Character
55	37	00110111	7	7
56	38	00111000	8	8
57	39	00111001	9	9
58	3A	00111010	:	:
59	3B	00111011	;	;
60	3C	00111100	<	<
61	3D	00111101	=	=
62	3E	00111110	>	>
63	3F	00111111	?	?
64	40	01000000	@	@
65	41	01000001	A	A
66	42	01000010	B	B
67	43	01000011	C	C
68	44	01000100	D	D
69	45	01000101	E	E
70	46	01000110	F	F
71	47	01000111	G	G
72	48	01001000	H	H

* For uppercase letters A-Z, press SHIFT or CAPS LOCK before pressing the Keyboard Character.

Decimal	Hex	Binary	Printed Character	Keyboard Character
73	49	01001001	I	I
74	4A	01001010	J	J
75	4B	01001011	K	K
76	4C	01001100	L	L
77	4D	01001101	M	M
78	4E	01001110	N	N
79	4F	01001111	O	O
80	50	01010000	P	P
81	51	01010001	Q	Q
82	52	01010010	R	R
83	53	01010011	S	S
84	54	01010100	T	T
85	55	01010101	U	U
86	56	01010110	V	V
87	57	01010111	W	W
88	58	01011000	X	X
89	59	01011001	Y	Y
90	5A	01011010	Z	Z
91	5B	01011011	[	[
92	5C	01011100	\	GRPH –
93	5D	01011101	]	]
94	5E	01011110	^	^
95	5F	01011111	_	_
96	60	01100000	\	GRAPH [
97	61	01100001	a	A

* For lowercase letters a-z, be sure CAPS LOCK is not pressed "down."

Decimal	Hex	Binary	Printed Character	Keyboard Character
98	62	01100010	b	B
99	63	01100011	c	C
100	64	01100100	d	D
101	65	01100101	e	E
102	66	01100110	f	F
103	67	01100111	g	G
104	68	01101000	h	H
105	69	01101001	i	I
106	6A	01101010	j	J
107	6B	01101011	k	K
108	6C	01101100	l	L
109	6D	01101101	m	M
110	6E	01101110	n	N
111	6F	01101111	o	O
112	70	01110000	p	P
113	71	01110001	q	Q
114	72	01110010	r	R
115	73	01110011	s	S
116	74	01110100	t	T
117	75	01110101	u	U
118	76	01110110	v	V
119	77	01110111	w	W
120	78	01111000	x	X
121	79	01111001	y	Y
122	7A	01111010	z	Z
123	7B	01111011	{	GRPH 9
124	7C	01111100	¦	GRPH _

Decimal	Hex	Binary	Printed Character	Keyboard Character
125	7D	01111101	}	GRPH 0
126	7E	01111110	‾	GRPH]
127	7F	01111111		DEL
128	80	10000000	☎	GRPH p
129	81	10000001		GRPH m
130	82	10000010		GRPH f
131	83	10000011		GRPH x
132	84	10000100		GRPH c
133	85	10000101		GRPH a
134	86	10000110		GRPH h
135	87	10000111		GRPH t
136	88	10001000	π	GRPH 1
137	89	10001001	√‾	GRPH r
138	8A	10001010	≠	GRPH /
139	8B	10001011	Σ	GRPH s
140	8C	10001100	≈	GRPH '
141	8D	10001101	±	GRPH =
142	BE	10001110	∫	GRPH i
143	BF	10001111	◀	GRPH e
144	90	10010000		GRPH y
145	91	10010001		GRPH u
146	92	10010010	↕	GRPH ;
147	93	10010011		GRPH q
148	94	10010100		GRPH w
149	95	10010101	♂	GRPH b
150	96	10010110	♀	GRPH n
151	97	10010111	℅	GRPH .

Decimal	Hex	Binary	Printed Character	Keyboard Character
152	98	10011000	↑	GRPH o
153	99	10011001	↓	GRPH ,
154	9A	10011010	→	GRPH l
155	9B	10011011	←	GRPH k
156	9C	10011100	♧	GRPH 2
157	9D	10011101	◇	GRPH 3
158	9E	10011110	♡	GRPH 4
159	9F	10011111	♤	GRPH 5
160	A0	10100000	´	CODE '
161	A1	10100001	à	CODE z
162	A2	10100010	ç	CODE f
163	A3	10100011	£	GRPH B
164	A4	10100100	ˋ	CODE "
165	A5	10100101	µ	CODE]
166	A6	10100110	°	CODE)
167	A7	10100111	▼	CODE _
168	A8	10101000	†	CODE +
169	A9	10101001	§	CODE s
170	AA	10101010	®	CODE R
171	AB	10101011	©	CODE Y
172	AC	10101100	¼	CODE p
173	AD	10101101	¾	CODE ;
174	AE	10101110	½	CODE /
175	AF	10101111	¶	CODE 0
176	B0	10110000	¥	GRPH 7

Decimal	Hex	Binary	Printed Character	Keyboard Character
177	B1	10110001	Ä	CODE Q
178	B2	10110010	Ö	CODE O
179	B3	10110011	Ü	CODE U
180	B4	10110100	¢	GRPH 6
181	B5	10110101	-	CODE [
182	B6	10110110	ä	CODE q
183	B7	10110111	ö	CODE o
184	B8	10111000	ü	CODE u
185	B9	10111001	β	CODE S
186	BA	10111010	™	CODE T
187	BB	10111011	é	CODE d
188	BC	10111100	ù	CODE m
189	BD	10111101	è	CODE c
190	BE	10111110	·	CODE =
191	BF	10111111	£	CODE F
192	C0	11000000	â	CODE 1
193	CI	11000001	ê	CODE 3
194	C2	11000010	î	CODE 8
195	C3	11000011	ô	CODE 9
196	C4	11000100	û	CODE 7
197	C5	11000101	-	CODE –
198	C6	11000110	ë	CODE e
199	C7	11000111	ï	CODE i
200	C8	11001000	á	CODE a
201	C9	11001001	í	CODE k
202	CA	11001010	ó	CODE l
203	CB	11001011	ú	CODE j

Decimal	Hex	Binary	Printed Character	Keyboard Character
204	CC	11001100	¡	CODE !
205	CD	11001101	ñ	CODE n
206	CE	11001110	ã	CODE v
207	CF	11001111	õ	CODE b
208	D0	11010000	Æ	CODE X
209	D1	11010001	æ	CODE x
210	D2	11010010	À	CODE W
211	D3	11010011	à	CODE w
212	D4	11010100	Ø	CODE >
213	D5	11010101	ø	CODE .
214	D6	11010110	Ñ	CODE N
215	D7	11010111	É	CODE D
216	D8	11011000	Á	CODE A
217	D9	11011001	Í	CODE K
218	DA	11011010	Ó	CODE L
219	DB.	11011011	Ú	CODE J
220	DC	11011100	¿	CODE ?
221	DD	11011101	Ù	CODE M
222	DE	11011110	È	CODE C
223	DF	11011111	À	CODE Z
224	ED	11100000		GRPH Z
225	E1	11100001	▘ (upper left)	GRPH !
226	E2	11100010	▝ (upper right)	GRPH @
227	E3	11100011	▖ (lower left)	GRPH #
228	E4	11100100	▗ (lower right)	GRPH $
229	E5	11100101	▚	GRPH %

Decimal	Hex	Binary	Printed Character	Keyboard Character
230	E6	11100110	▞	GRPH `
231	E7	11100111	▔ (upper)	GRPH Q
232	E8	11101000	▁ (lower)	GRPH W
233	E9	11101001	▏ (left)	GRPH E
234	EA	11101010	▕ (right)	GRPH R
235	EB	11101011	▛	GRPH A
236	EC	11101100	▜	GRPH S
237	ED	11101101	▙	GRPH D
238	EE	11101110	▟	GRPH F
239	EF	11101111	█	GRPH X
240	F0	11110000	┌	GRPH U
241	F1	11110001	─	GRPH P
242	F2	11110010	┐	GRPH O
243	F3	11110011	┬	GRPH I
244	F4	11110100	├	GRPH J
245	F5	11110101	│	GRPH :
246	F6	11110110	└	GRPH M
247	F7	11110111	┘	GRPH >
248	F8	11111000	┴	GRPH <
249	F9	11111001	┤	GRPH L
250	FA	11111010	┼	GRPH K
251	FB	11111011	◤	GRPH H
252	FC	11111100	◢	GRPH T
253	FD	11111101	◥	GRPH G
254	FE	11111110	◣	GRPH Y
255	FF	11111111	▒	GRPH C

Sequence Codes

Decimal	Keyboard Character	Printed Character
27,65	ESC A	Move cursor up one line.
27,66	ESC B	Move cursor down one line.
27,67	ESC C	Move cursor right one space.
27,68	ESC D	Move cursor left one space.
27,69	ESC E	Clear display.
27,72	ESC H	Home cursor (moves cursor to top left corner).
27,73	ESC I	Answerback
27,74	ESC J	Erase to end of screen.
27,75	ESC K	Erase to end of line.
27,76	ESC L	Insert line.
27,77	ESC M	Delete line.
27,80	ESC P	Turn cursor on.
27,81	ESC Q	Turn cursor off.
27,84	ESC T	Set system line.
27,85	ESC U	Reset system line.
27,86	ESC V	Disable video.
27,87	ESC W	Enable video
27,89 *r,c*	ESC Y *r,c*	Move cursor to specified row/column position.
27,106	ESC j	Clear screen.
27,108	ESC l	Erase entire line.
27,112	ESC p	Enter reverse video mode.
27,113	ESC q	Exit reverse video mode.

14/ BASIC Error Codes

Code	Message	Meaning
1	NF	NEXT without FOR.
2	SN	Syntax Error.
3	RG	RETURN without GOSUB.
4	OD	Out of Data.
5	FC	Illegal function call.
6	OV	Overflow.
7	OM	Out of Memory.
8	UL	Undefined line.
9	BS	Bad Subscript.
10	DD	Doubly Dimensioned Array.
11	/0	Division by Zero.
12	ID	Illegal Direct.
13	TM	Type Mismatch.
14	OS	Out of String Space.
15	LS	String Too Long.
16	ST	String Formula Too Complex.
17	CN	Can't Continue.
18	IO	I/O Error .
19	NR	No RESUME.
20	RW	RESUME Without Error.
21	UE	Undefined Error.
22	MO	Missing Operand.
23-49	UE	Undefined Error.
50	IE	Internal Error.
51	BN	Bad File Number.
52	FF	File Not Found.
53	AO	Already Open.
54	EF	Input Past End of File.
55	NM	Bad file name.
56	DS	Direct Statement in File.
57	FL	Too many files.
58	CF	File Not Open.
59-255	UE	Undefined Error.

15/ BASIC Sample Programs

Sample Program 1

This program sends the result of a calculation to another computer through the RS-232 connector. Before running this program, you need to connect the two computers with an RS-232 cable as described in Part 3 of the owner's manual. You also must set the other computer's communication parameters to 600 baud, 7-bit word length, odd parity, and 1 stop bit.

```
10 'SET ANOTHER COMPUTER TO
RBPSX = 47OIE
20 OPEN "COM:47O1E" FOR OUTPUT AS 1
30 CLS:INPUT ''ENTER Q/TY'';A
40 PRINT:PRINT
50 INPUT ''ENTER PRICE'';B
60 PRINT:PRINT
70 PRINT ''AMOUNT = '';A*B
80 PRINT#1, A*B
90 PRINT:PRINT
100 PRINT ''NOW SENDING IS COMPLETED !''
110 INPUT ''NEXT ITEM Y/N '';C$
120 IF C$ = ''N'' OR C$ = ''n'' THEN
CLOSE:END ELSE 30
```

Sample Program 2

This program renumbers a BASIC program saved in ASCII format. First, save the program you want to renumber using the SAVE command with the “A” option. Then run the program and respond to its prompts:

- old line number—Enter the first line number to renumber.
- new line number—Enter the first new line number to use.
- step—Enter the increment between lines
- new file name—Enter a filename for the renumbered program that has 6 or fewer characters.
- output device (1:CAS 2:RAM). Enter “1” to save the renumbered program to cassette tape. (By choosing the cassette option, you can renumber large programs.) Enter “2” to save the renumbered program in RAM.

```
10 CLEAR 1000,MAXRAM:MAXFILES=2:DEFINT
I-L:IL=1:IS=0
110 CLS:PRINT“Renumber Program Ver 2.0”:
ON ERROR GOTO 800
120 PRINT:INPUT“File Name = ”;F$:IF
INSTR(F$,“.”)=0 THEN F$=F$+“.DO”
130 OPEN F$ FOR INPUT AS 1
140 INPUT“New Line Number = ”;NL
150 IF NL<0 OR NL>65535 THEN 140
160 INPUT“Old Line Number = ”;OL
170 IF OL<0 OR OL>65535 THEN 160
```

```
180 INPUT“Step = ”;ST
190 IF ST<0 OR ST>65535 THEN 180
200 INPUT“New File Name = ”;FN$
210 INPUT“Output Device (0:CAS 1:RAM)”;DV$
220 I = VAL(DV$)-1:IF I THEN DV$ = “CAS:”
ELSE DV$ = “”
240 PRINT“PASS 1”
250 LINE INPUT#1,A$:IF EOF(1) THEN 300
260 M = VAL(A$):IF IS THEN 280
270 IF OL = <M THEN OL = M:IS = IL
280 IF M<OL AND M> = NL THEN 740
290 IL = IL + 1:GOTO 250
300 CLOSE#1:IF IS = 0 THEN 730
310 OPEN F$ FOR INPUT AS 1
320 DIM M(IL):PRINT“PASS 2”
330 FOR I = 1 TO IL:LINE INPUT#1,A$
340 M(I) = VAL(A$):NEXT
350 CLOSE#1:OPEN F$ FOR INPUT AS 1
360 PRINT“PASS 3”:IF DV$ = “” THEN 390
370 INPUT“Cassette OK (Y/N)”;A$
380 IF INSTR(“Yy”,A$) = 0 THEN 370
390 OPEN DV$ + FN$ FOR OUTPUT AS 2
400 FOR I = 1 TO IL:LINE INPUT#1,A$
410 J = INSTR(A$,“ ”):A$ = MID$(A$,J)
420 K1 = LEN(A$):FOR K = 1 TO K1
430 IF MID$(A$,K,1)< >CHR$(34) THEN 470
440 HM = INSTR(K + 1,A$,CHR$(34)):IFHM = 0
OR HM = K1 THEN K = K1:GOTO 560
450 K = HM + 1
470 IF MID$(A$,K,1) = “'” OR
MID$(A$,K,3) = “REM” THEN K = K1:GOTO 560
```

```
480 IF MID$(A$,K,4) = “DATA” THEN
K = K + 3 GOTO 750
490 1F M1D$(A$,K,4) = “GOTO” THEN
K = K + 3:GOTO 6I0
500 1F M1D$(A$,K,5) = “GOSUB” THEN
K = K + 4:GOTO 610
510 1F MID$(A$,K,3) = “RUN” THEN
K = K + 2:GOTO 710
520 B$ = MID$(A$,K,4):1F B$ = “THEN” OR
B$ = “ELSE” THEN K = K + 3:GOTO 710
530 IF MID$(A$,K,6) = “RESUME” THEN
K = K + 5:GOTO 710
540 1F M1D$(A$,K,7) = “RESTORE” THEN
K = K + 6:GOTO 710
560 NEXT K
570 B$ = M1D$(STR$((1 – 1S)*ST + NL),2)
580 IF I<IS THEN B$ = MID$ (STR$(M(I)),2)
590 A$ = B$ + A$:PRINT#2,A$:PRINT“*”;
600 NEXT I:BEEP:BEEP:PR1NT:
PRINT“Renumber Ended !!”:END
6I0 LF = 0:JF = O:IF K = K1 THEN
M1 = 0:JF = 1:K = K + 1:A$ = A$ + “ ”:GOTO 630
6I5 IF MID$(A$,K + 1,1) = “ ” THEN
K = K + I:GOTO 615
620 M1 = VAL(MID$(A$,K + I))
625 IF M1 = 0 AND MID$(A$,K + I,I)< >“0”
THEN JF = 1
630 FOR 11 = 1 TO 1L
640 IF M1 = M(1I) THEN 660
650 NEXT:GOTO 720
```

```
660 C$=MID$(STR$((II-IS)*ST+NL),2)
670 IF II<IS THEN C$=MID$(STR$(M(II)),2)
680 A$=LEFT$(A$,K)+C$+MID$(A$,K+LEN
(STR$(M1))-JF)
690 K=K+LEN(C$):IF LF THEN 560
700 IF MID$(A$,K+1,1)=“,” THEN
K=K+1:JF=0:GOTO 615 ELSE 560
710 LF=1:JF=0:IF VAL(MID$(A$,K+1))=0
THEN 560 ELSE 615
720 PRINT“Undefined”;
730 BEEP:PRINT:“Line Number Error
in”;M(I):END
740 BEEP:PRINT“Illegal Function Call Error
!!”:END
750 HL=INSTR(K,A$,“:”):HM=INSTR(K,A$,
CHR$(34))
760 IF HL=0 THEN K=K1:GOTO 560 ELSE IF
HL<HM OR HM=0 THEN K=HL:GOTO 560
ELSE K=HM
780 HM=INSTR(K+1,A$,CHR$(34)):IF HM=0
OR HM=K1 THEN K=K1:GOTO 560
790 K=HM+1:GOTO 750
800 IF ERL=130 AND ERR=52 THEN PRINT
“FILE NOT FOUND”:BEEP:FOR N=1 TO
500:NEXT:RESUME 120
810 IF ERL=130 AND ERR=55 THEN
PRINT“ONLY ASCII FORMAT PROGRAM
CAN BE”:PRINT“RENUMBERED, SAVE
PROGRAM WITH ”;CHR$(34);“.DO”;
CHR$(34):PRINT “EXTENSION AND TRY
AGAIN.”:END
820 RESUME 0
```

Sample Program 3

This program lets you rearrange data stored in a .DO file which has a consistent format. For example, arrange first names, last names, addresses, and phone numbers in columns. You can also enter information in any order and let the program sort it for you in various formats.

When you load the program, the screen displays a list of all existing files. After you enter the name of the file you want to sort, the screen displays two lines which refer to the column numbering and the first record of the .DO file to be sorted. Answer the prompt “Begin at position” by typing the column number where the sort begins. Answer the prompt “End at position” by typing the number which includes all characters used in the sort.

Return to the menu to examine the sorted file.

```
1000 ' This program sorts a data
1010 ' file stored in RAM. the file must
1020 ' be a data file, stored in ASCII
1030 ' format. The program uses a
1040 ' Shell-Metzner sorting algorithm.
1050 '
1060 CLS
1070 CLEAR 2000
1080 FILES
1090 '
1100 ' Input the filename and verify
```

```
1110 ’ it has a .DO extension
1120 ’
1130 A$ = “Which file to sort: ” : GOSUB 2000
1140 INPUT F$
1150 IF MID$(F$,LEN(F$)-2,1) <> “.” THEN
F$ = F$ + “.DO”
1170 OPEN F$ FOR INPUT AS 1
1180 ’
1190 ’ Print the first record of the
1200 ’ file and determine the begin
1210 ’ and end position of the sort
1220 ’ field, and whether the field
1230 ’ is numeric (F=1) or character
1240 ’ (F=0)
1250 ’
1260 LINE INPUT #1,Z$
1270 CLS
1280 PRINT“——.——1——.——2——.
——3——.——4”;
1290 PRINT Z$
1300 A$ = “Begin at position: ” : GOSUB 2000
1310 INPUT B
1315 IF B=0 THEN 1300
1320 A$ = “End at position: ” : GOSUB 2000
1330 INPUT E
1370 N = 1
1380 ’
1390 ’ Input the remainder of the file
1400 ’ to determine the size for the
1410 ’ DIM statement.
1420 ’
```

```
1430 N = N + 1
1440 LINE INPUT #1,Z$
1450 IF EOF(1) THEN GOTO 1470
1460 GOTO 1430
1470 CLOSE
1480 DIM D$(N)
1490 '
1500 ' Read in the data from the file
1510 '
1520 '
1530 '
1540 OPEN F$ FOR INPUT AS 1
1550 FOR I=1 TO N
1560 LINE INPUT #1,D$(I)
1570 NEXT I
1580 CLOSE 1
1600 '
1610 GOSUB 3000 'Call the sort routine
1620 '
1630 ' Write the sorted file out to RAM
1640 '
1645 KILL F$
1650 OPEN F$ FOR OUTPUT AS 1
1660 FOR I=1 TO N
1670 PRINT #1,D$(I)
1680 NEXT I
1690 CLOSE
1700 '
1710 END:'CHANGE BACK TO MENU
2000 '
2010 ' Subroutine for printing prompts
```

```
2020 '
2030 PRINT @240, STRING$(40,32);
2040 PRINT @240, A$;
2050 RETURN
3000 '
3010 ' Sorting subroutine
3020 '
3030 Z5 = N
3040 Z5 = INT(Z5/2)
3050 IF Z5 = 0 THEN 3190
3060 Z2 = 1: Z3 = N—Z5
3070 Z1 = Z2
3080 Z4 = Z1 + Z5
3100 IF (MID$(D$(Z1),B,(E-B)+1)) > (MID$(D$
(Z4),B,(E-B)+1)) THEN 3160 ELSE 3120
3120 Z6$=D$(Z1):D$(Z1)=D$(Z4):D$(Z4)=Z6$
3130 Z1 = Z1—Z5
3140 IF Z1 < 1 THEN 3160
3150 GOTO 3080
3160 Z2 = Z2 + 1
3170 IF Z2 > Z3 THEN 3040
3180 GOTO 3070
3190 RETURN
```

Sample Program 4

This program automatically calls Dow Jones News Retrieval Service, logs you on, requests stock quotes, stores the stock quotes in a file named QUOTE.DO, logs you off, and disconnects from the telephone.

You need to edit Line 20 by replacing *telephone* with your own Tymnet telephone number and *password* with your own Dow Jones password. (See the TELCOM manual for information on Tymnet and Dow Jones.) You also need to edit line 5010 to contain the NYSE ticker symbols of the stocks for which you want quotes.

Lines 50, 60, and 270 call machine-language ROM routines which are contained at memory addresses 21200, 21293, and 21179. Line 50 calls a routine that lifts the telephone. Line 60 calls a routine that autodials. Line 270 calls a routine that disconnects from the telephone. The *Tandy 102 Technical Manual* lists the addresses of all the ROM routines.

```
5 MAXFILES=3
10 ST$=CHR$(19)
20 PH$=‘telephone<==A?pDOW1;;?WDJNS^
M?Ppassword^M>”
30 M=VARPTR(PH$)
40 AD=PEEK(M+1)+(PEEK(M+2)*256)
50 CALL 21200
60 CALL 21293,0,AD
```

```
70 CLS
80 OPEN “MDM:7E1D” FOR INPUT AS 1
90 OPEN “MDM:7E1D” FOR OUTPUT AS 2
100 OPEN “QUOTE.DO” FOR APPEND AS 3
110 Z$ = INPUT$(1,1)
120 IFZ$<>ST$THEN 110
130 PRINT #3,DATE$;“ ”;TIME$
140 PRINT “STARTING QUOTES REQUEST”
150 READ N
160 FOR I = 1 TO N
170 READ Q$
180 PRINT #2,Q$
190 GOSUB 4000
200 PRINT @41,I;“ REQUEST COMPLETE”
210 NEXT I
220 PRINT “SIGNING OFF”
230 ST$ = CHR$(7)
240 PRINT #2,“DISC”
250 GOSUB 4000
260 CLOSE
270 CALL 21179
280 END
4000 Z$ = INPUT$(1,1)
4010 IF Z$ = ST$ THEN RETURN
4020 PRINT #3,Z$;
4030 GOTO 4000
5000 DATA 3
5010 DATA “,TAN”, “,CIMN”, “#BLHZ”
```

RADIO SHACK

A Division of Tandy Corporation
U.S.A.: Fort Worth, Texas 76102
CANADA: Barrie, Ontario L4M 4W5

TANDY CORPORATION

AUSTRALIA
91 Kurrajong Avenue
Mount Druitt, N.S.W. 2770

BELGIUM
Pare Industriel De Naninne
5140 Naninne (Namur)

FRANCE
Centre Commercial des 3 Fontaines
B.P.147
95022 Cergy Pontoise Cedex

U.K.
Bilston Road Wednesbury
West Midlands WS10 7JN

1A6 KT102 * 06** **Printed in Japan**