LIMITED WARRANTY

Radio Shack warrants for a period of 90 days from the date of delivery to customer that the computer hardware described herein shall be free from defects in material and workmanship under normal use and service. This warranty shall be void if the computer case or cabinet is opened or if the unit is altered or modified. During this period, if a defect should occur, the product must be returned to a Radio Shack store or dealer for repair. Customer's sole and exclusive remedy in the event of defect is expressly limited to the correction of the defect by adjustment, repair or replacement at Radio Shack's election and sole expense, except there shall be no obligation to replace or repair items which by their nature are expendable. No representation or other affirmation of fact, including but not limited to statements regarding capacity, suitability for use, or performance of the equipment, shall be or be deemed to be a warranty or representation by Radio Shack, for any purpose, nor give rise to any liability or obligation of Radio Shack whatsoever.

EXCEPT AS SPECIFICALLY PROVIDED IN THIS AGREEMENT, THERE ARE NO OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL RADIO SHACK BE LIABLE FOR LOSS OF PROFITS OR BENEFITS, INDIRECT, SPECIAL, CONSEQUENTIAL OR OTHER SIMILAR DAMAGES ARISING OUT OF ANY BREACH OF THIS WARRANTY OR OTHERWISE.

IMPORTANT NOTICE

ALL RADIO SHACK COMPUTER PROGRAMS ARE DISTRIBUTED ON AN "AS IS" BASIS WITHOUT WARRANTY

Radio Shack shall have no liability or responsibility to customer or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by computer equipment or programs sold by Radio Shack, including but not limited to any interruption of service, loss of business or anticipatory profits or consequential damages resulting from the use or operation of such computer or computer programs.

NOTE: Good data processing procedure dictates that the user test the program, run and test sample sets of data, and run the system in parallel with the system previously in use for a period of time adequate to insure that results of operation of the computer or program are satisfactory.

This Reference Manual and You

We've prepared this Reference Manual with the assumption that you – the user – already have considerable experience with programming in BASIC. Our LEVEL I User's Manual was written for the total beginner – and has been greeted with wide acclaim. We freely admit this Manual has not been written from the same perspective.

But by the time you recognize a desire (or need) for a LEVEL II BASIC, we expect that you've gone through our LEVEL I Manual and have a solid foundation in programming.

If this is your first experience with programming micro-computers, we very strongly urge you to spend time with a LEVEL I TRS-80 first – and the Manual we prepared for it.

If you've had experience with other forms of the BASIC language (other micro-computers or time share systems) then you should be ready for our Reference Manual for LEVEL II.

LEVEL II is a far more powerful version of BASIC than was LEVEL I. If you have been working with LEVEL I for some time, be prepared for some pleasant surprises – and some differences that might throw you for awhile (for example, LEVEL I programs won't run as-is on a LEVEL II machine... you'll have to modify them). This Manual is a complete reference guide – it is not intended to be a complete step-by-step training manual or an applications book (that will come later).

If you have some suggestions... criticisms... additions... concerning this Manual – we'd be glad to hear from you.

FIRST EDITION — 1978

Printed in the United States of America

CONTENTS

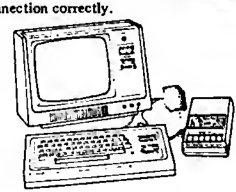
Appendices

Setting Up The System

Carefully unpack the system. Remove all packing material. Be sure you locate all cables, papers, tapes, etc. Save the packing material in case you need to transport the system.

Connecting the Video Display and Keyboard:

1. Connect the power cord from the Video Display to a source of 120 volts, 60 Hz AC power. Note that one prong of the AC plug is wider than the other – the wide prong should go into the widest slot of the AC socket.
   NOTE: If you use an AC extension cord, you may not be able to plug the Display's power cord in. Do not attempt to force this wide prong into the extension cord; use a wall outlet if at all possible.
2. Connect the power cord of the Power Supply to a source of 120 volts, 60 Hz AC power.
3. Connect the gray cable from the front of the Video Monitor to the VIDEO jack on the back of the Keyboard Assembly. Take care to line up the pins correctly (the plug fits only one way).
   NOTE: Before the next step, be sure the POWER switch on the back of the Keyboard is off (button out).
4. Connect the gray cable from the Power Supply to the POWER jack on the back of the Keyboard Assembly. Again, take care to mate the connection correctly.



Connecting The Cassette Recorder:

NOTE: You do not need to connect the Cassette Recorder unless you plan to record programs or to load taped programs into the TRS-80.

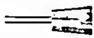
1. Connect the CTR-41 to a source of 120 volt AC power. (Batteries are not recommended for using Recorder with TRS-80.)
2. Connect the short cable (DIN plug on one end and 3 plugs on the other) to the TAPE jack on the back of the Keyboard Assembly. Be sure you get the plug to mate correctly.
3. The 3 plugs on the other end of this cable are for connecting to the CTR-41.

A. Connect the black plug into the EAR jack on the side of the CTR-41. This connection provides the output signal from the CTR-41 to the TRS-80 (for loading Tape programs into the TRS-80).

B. Connect the larger gray plug into the AUX jack on the CTR-41. This connection provides the recording signal to record programs from the TRS-80 onto the CTR-41's tape.

Also, plug the Dummy Plug (provided with the CTR-41) into the MIC jack (this disconnects the built-in Mic so it won't pick up sounds while you are loading tapes).

NOTE: Be sure you always use the Dummy Plug when saving programs on tape (Recording).



Dummy Plug

C. Connect the smaller gray plug into the REM jack on the CTR-41. This allows the TRS-80 to automatically control the CTR-41's motor (turn tape motion on and off for recording and playing tapes).

Notes On Using The Recorder

There are a number of things you should be aware of as you use the Cassette Tape System:

1. To play a tape (load a taped program into the TRS-80), you must have the CTR-41's Volume control set to middle to upper levels, (approximately 4 to 6). Then press the CTR-41's PLAY key and then type CLOAD on the TRS-80 and ENTER this command. This will start the tape motion. An * will appear on the top line of the Monitor; a second * will blink, indicating the program is loading. When loading is done, the TRS-80 will automatically turn the CTR-41 off and flash READY on the screen. You are then ready to RUN the program (type in RUN and hit ENTER).
2. To record a program from the TRS-80, press the CTR-41's RECORD and PLAY keys simultaneously. Then type CSAVE followed by a one-letter "file-name" in quotes and ENTER this command. When the program has been recorded the TRS-80 will automatically turn the CTR-41 off and display READY on the screen. Now you have your program on tape (it still is in the TRS-80 also). Many computer users make a second or even a third recording of the tape, just to be sure they have a good recording.
3. Use the CTR-41's Tape Counter to aid you in locating programs on tapes.
4. For best results, use Radio Shack's special 10 minute per side Computer Tape Cassettes (especially designed for recording computer programs). If you use standard audio tape cassettes, be sure to use top quality, such as Realistic SUPERTAPE. Keep in mind that audio cassettes have lead-ins on both ends (blue non-magnetic mylar material) — you can not record on the leader

portion of the tape. Advance the tape past the leader before recording a program.

5. When you are not going to use a CTR-41 for loading or recording programs, do not leave RECORD or PLAY keys down (press STOP).
6. To REWIND or FAST-Forward a cassette, place Recorder in REWIND or FAST-Forward, then type CLOAD and hit ENTER When tape has reached the desired position, push the Reset button inside the Expansion Port access door (rear left of TRS-80). (Instead of using this CLOAD/Reset sequence, you could remove the REMote plug from its jack: however, repeated insertion/removal tends to wear out any plug and is not recommended.)
7. If you want to save a taped program permanently, break off the erase protect tab on the cassette (see CTR-41 Manual).
8. Do not expose recorded tapes to magnetic fields. Avoid placing your tapes near the Power Supply.
9. To check if a tape has a program recorded on it, you can disconnect the plug from the EAR jack (also disconnect the REM plug so you can control the CTR-41 with the keys) and Play the tape; you'll hear the program material from the speaker.
10. For the best results when using a Recorder with the Computer, you should keep the Recorder's heads and tape handling mechanism very clean. A new Recorder should be cleaned before it is used the first time, and cleaned again after every four hours' use. In addition, the tape heads should be demagnetized periodically.

    A complete line of recorder accessories (cleaning solution, cotton-tipped swabs, demagnetizer-cassettes, etc.) is available at your local Radio Shack store.

Special Note:

Before attempting to load a program from tape into the Computer, be sure the cassette is rewound to a blank portion of the tape preceding the program. If you try to start the load in the middle of a preceding program, you probably will get the Computer "hung up" (in which case you'll have to press Reset and start over).

The same rule applies when you're using the CLOAD? command to compare a taped program with one stored in the Computer.

Tips On Loading Cassette Programs

There are many factors which will affect the performance of a cassette system. The most significant one is volume. Too low a volume may cause some of the information to be missed. Too high a volume may cause distortion and result in the transfer of background noise as valid information. Both of these situations will cause errors.

The recommended volume settings* for loading from cassette tape are:

	USER GENERATED	PRE-RECORDED RADIO SHACK
LEVEL II	4-6	5 1/2 - 6 1/2
LEVEL I	7-8	7 1/2 - 8 1/2

If the asterisks do not appear during a load, try lowering the volume. It is also a good idea to unplug the EARphone (black) plug and listen for the start of the program. This will tell you exactly where the program starts. If the asterisks appear, but one is not flashing, try increasing the volume setting. If higher volume setting doesn't solve the problem, clean the head.

Handling Load-Errors

There is a very rare case in which only a minor error may occur in loading a program and no error message will be printed. The best way to check for this, is to List the program. If the program looks OK, use the CLOAD? command to compare the tape version with the one you loaded. If they are not exactly the same, a "BAD" message will be printed. Such a case normally can be remedied with a minor adjustment in the volume setting (usually a slight increase).

*Numbers refer to markings on the Radio Shack CTR-41 Recorder, which run from 0 to 10 (full volume). For different models of Recorders, numbers recommended may not be appropriate. Do a little experimenting.

1 / General Information

This chapter will provide you with an overview of LEVEL II BASIC – what some of its special features are, how it differs from LEVEL I, and generally, what you need to get going. In addition, there's a short glossary at the end of the chapter.

Power-Up

Connect Keyboard-Computer, Video Display and Power Supply as explained in the previous section. Plug Video Display and Power Supply into 120-volt AC outlets. Press POWER buttons on Video Display and at the back of the Keyboard. Give the video tube a few seconds to warm up.

MEMORY SIZE? – will appear on the screen. This is your chance to protect a segment of memory so that machine-language programs may be loaded, using a special command, SYSTEM. For normal applications, you won't want to protect any memory, so just press the ENTER key without typing in any numbers. This will allow you to write BASIC programs using the full memory capacity of your Computer (for 4K LEVEL II machines, that's 3284 bytes; for 16K LEVEL II machines, it's 15,572 bytes).

NOTE: In general, whenever you have typed something in via the keyboard and you want the Computer to "act" on your input, you must first hit the ENTER key just as you did with the Level I TRS-80. There are ways to have the Computer respond as soon as you hit a key (without ENTER), but these will be covered later.

```
RADIO SHACK LEVEL II BASIC
READY
>_
```

will appear on the screen. You are now ready to use LEVEL II BASIC.

Operating Modes

There are four operating modes: Command, Execute, Edit and Monitor. Command and Execute Modes are just like LEVEL I BASIC. In the Command Mode, the Computer responds to commands as soon as they are entered. This is the level you use to write programs and perform computations directly ("calculator mode" of LEVEL I). Whenever the >_ appears on the Display, you're in the Command Mode.

The Execute Mode is usually entered by typing RUN: this causes BASIC programs to be executed. Unlike LEVEL I, LEVEL II initializes all numeric variables to zero and sets all strings to null when you enter the command RUN.

The Edit Mode is a real time-saving feature of LEVEL II. It allows you to edit (alter, add to or delete) the contents of program lines. Instead of retyping an entire program line, you change just the part that needs changing.

NOTE: Whenever Computer encounters a Syntax error during execution, it will go into Edit Mode for that line. To get out of Edit Mode, type "Q" (without quotes).

The Monitor Mode lets you load machine language "object files" into memory. These routines or data can then be accessed by your BASIC programs, or they may be completely independent programs.

Special Function Keys

LEVEL II BASIC offers the same special function keys as LEVEL I – plus a few extras. The function of the key depends on what mode the Computer is in.

Command Mode:

ENTER	Effects a carriage return; Computer "looks at" line just typed in and acts accordingly. If line just typed in has no line number, Computer will interpret and execute the statements contained in the line. If Line has a line number, Computer stores the line in program memory.
←	Backspaces the cursor and deletes last character typed in.
SHIFT ←	Deletes the line you are typing in, and returns cursor to beginning of logical line.
↓	Linefeed; moves cursor down to next physical line on the Display.
:	Separates BASIC statements contained on the same logical line, to allow multi-statement lines. E.g., PRINT "FIRST STATEMENT":PRINT "SECOND STATEMENT"
→	Moves cursor over to the next tab stop. Tab stops are at positions 0, 8, 16, 24, 32, 40, 48 and 56.
SHIFT →	Converts display to 32 character-per-line format.
CLEAR	Clears the Display and returns it to 64 character-per-line format.

Execute Mode:

SHIFT @	Pause; stops program execution. Hitting any key causes execution to be resumed. Hitting SHIFT @ also freezes the Display during a LIST so you can examine program lines.
BREAK	Stops execution. Resume execution by typing CONT.
ENTER	When Computer is awaiting input from the keyboard, ENTER causes Computer to "look at" what you've typed in.

For Edit Mode special function keys, see Chapter 9.

Variable Names

Variable names must begin with a letter (A-Z) and may be followed by another letter or digit (0-9). So the following are all valid and distinct variable names:

A A2 AA AZ G9 GP M MU ZZ Z1

Variable names may be longer than two characters, but only the first two characters will be used by the computer to distinguish between variables. For example "SUM", "SUB" and "SU" will be treated as one and the same variable by LEVEL II BASIC.

As you can imagine, this gives you plenty of variable names to use in LEVEL II (in the neighborhood of 900). However, you cannot use variable names which contain words with special meaning in the BASIC language. For example, "XON" cannot be used as a variable name, since it contains the BASIC keyword "ON". The complete list of "reserved words" which cannot be used in variable names appears in Appendix A of this Manual.

Variable Types

There are four types of variables in LEVEL II: integer, single precision, double precision, and string variables. The first three types are used to store numerical values with various degrees of precision; the last type stores strings (sequences) of characters — letters, blanks, numbers and special symbols - up to 255 characters long. LEVEL I only allowed two string variables, A$ and B$ – but LEVEL II allows you to use any variable name for strings, simply by adding the string declaration character, $, to the variable name. There are declaration characters for the other variable types, too: Here's a complete listing:

Variable Type	Declaration Character	Examples	Typical values stored
integer (whole numbers greater than −32769 and less than +32768	%	A%, B9%	−30, 123, 3, 5001
single precision (6 significant figures)	!	A!, AA!, Z1!	1, −50, .123456, 353421
double precision (16 significant figures)	#	A#, ZZ#, C#	−300.12345678, 3.141592653589, 1.00000000000001
double precision with scientific notation (for entering constants or during output of large or small numbers)	D	"A#=1.2345678901D+12"	$1.2345678901 \times 10^{12}$
string (up to 255 characters)	$	A1$, GT$, HI$	"JOHN Q. DOE", "WHISTLE-STOP" "1 + 2 = ?"

The same variable name may be used for different variable types, and the Computer will still keep them distinct, because of the type declaration character: For example, A$, A%, A!, A# are distinct variable names.

Variables without declaration characters are assumed to be single-precision; this assumption can be changed with DEFine statements (Chapter 4).

Arrays

Any valid variable name can be used to name an array in LEVEL II BASIC; and arrays are not limited to one dimension. The DIMension statement is used to define arrays at the beginning of a program. Depending on the variable type used, an array may contain strings, integers, double precision values, etc. A whole chapter of this Manual is devoted to arrays:

Examples: A$ (X,Y,Z) would be a three-dimensional array containing string values
G3 (I,J) would be a two-dimensional array containing numerical single-precision values
G#(I) would be a one dimensional array of double precision values.

Arithmetic Operators

LEVEL II uses the same arithmetic operators as LEVEL I: + (addition), − (subtraction), * (multiplication) and / (division). And there's a new, very handy operator: ↑ (exponentiation: 2 ↑ 3 = 8).

For example, to compute $6 * 2^{1/3}$ `PRINT 6*2 ↑ (1/3)`

NOTE: Some TRS-80's generate a [character instead of the ↑ arrow.

Relational Operators

These are the same as LEVEL I.

< (less than) > (greater than) =(equal to)
<>(not equal to) <=(less than or equal to) >=(greater than or equal to)

These operators are useful both for IF . . . THEN statements and for logical arithmetic.

Example: `100 IF C<=0 THEN C=127`

Logical Operators

In LEVEL I BASIC, * and + were used to represent the logical operators AND and OR. In LEVEL II, we don't use symbols, we use AND and OR directly. We also have another operator, NOT.

Examples:

```
80 IF Q = 13 AND R2 = 0 THEN PRINT "READY"
```

`100 Q = (G1<0) AND (G2<L)`	Q = −1 if both expressions are True; otherwise Q = 0
`200 Q = (G1<0) OR (G2<L)`	Q = −1 if either expression is True; otherwise Q = 0
`300 Q = NOT(C>3)`	Q = −1 if the expression is False; Q = 0 if it is True

```
400 IF NOT (P AND Q) THEN PRINT "P AND O ARE NOT BOTH EQUAL TO —1"

500 IF NOT (P OR Q) THEN PRINT "NEITHER P NOR Q EQUALS —1"
```

String Operators

Strings may be compared and concatenated ("strung together") in LEVEL II. A whole chapter of this Manual is devoted to string manipulations.

Symbol	Meaning	Example
<	precedes alphabetically	"A" < "B"
>	follows alphabetically	"JOE" > "JIM"
=	equals	B$ = "WIN"
<>	does not equal	IF A$<>B$ THEN PRINT A$
<=	precedes or equals	IF A$<=A2$ PRINT "DONE"
>=	follows or equals	IF L1$>="SMITH" PRINT L1$
+	concatenate the two strings	A$ = C$+C1$ A$ ="TRS–" + "80"

Order of Operations

Operations in the innermost level of parentheses are performed first, then evaluation proceeds to the next level out, etc. Operations on the same nesting level are performed according to the following hierarchy:

Exponentiation: A ↑ B

Negation: −X

*, / (left to right)

+, − (left to right)

<, > , =, < =, > =, < > (left to right)

NOT

AND

OR

Intrinsic Functions

Most of the subroutines in the LEVEL I manual are built-in to LEVEL II. They are faster, more accurate, and much easier to use.

Graphics

Level II has the same SET, RESET and POINT functions as LEVEL I for turning graphics blocks on and off and determining whether an individual block is on or off. (There are a few differences – see Chapter 8.)

A big feature of LEVEL II is the selectable display – either 64 characters per line or 32 characters per line (c/l). When the machine is turned on it is in the 64 c/l mode; hit SHIFT and ← simultaneously to change to 32 c/l. Display will return to 64 c/l whenever a CLS or NEW is executed or CLEAR key is hit. You can also shift to 32 c/l by executing a PRINT CHR$ (23). More on this in Chapter 5.

Error Messages

LEVEL I pointed out errors by printing HOW?, WHAT? or SORRY along with the offending program line with a question mark inserted at the point of error. LEVEL II gives you much more specific information about what type of error occurred, using a set of Error Codes (see Appendix). The offending program line is also pointed out, but it's up to you to locate the error in the line.

Abbreviations

Very few abbreviations are allowed in LEVEL II. Ex-LEVEL I users will have to forget about R., L., P., etc. Although LEVEL II doesn't allow these short-forms, it stores the programs more efficiently than LEVEL I did, so you can still pack a lot of program into a small amount of memory space.

The abbreviations are:

? for PRINT, and
' for :REM
. for last line entered, listed, edited, or in which an error occurred.

Keyboard Rollover

With the LEVEL I TRS-80 (and many other computers) you have to release one key before the Computer will allow entry of another key. LEVEL II lets you hit the second key before you have released the first key. This is great for you touch typists.

Glossary for LEVEL II BASIC

address a value specifying the location of a byte in memory; decimal values are used in LEVEL II

alphanumerics the set of letters A-Z, the numerals 0-9, and various punctuation marks and special characters

argument the value which is supplied to a function and then operated on to derive a result

array an arrangement of elements in one or more dimensions

ASCII American Standard Code for Information Interchange; in LEVEL II BASIC, decimal values are used to specify ASCII codes

assembler a program that converts a symbolic-language program into a machine-language program

BASIC Beginners All-purpose Symbolic Instruction Code

baud signaling speed in bits per second; LEVEL II's cassette interface operates at 500 baud (500 bits per second)

binary number a number represented in the base-two number system using only binary digits "0" and "1"

bit binary-digit, the smallest memory cell in a computer

byte the smallest memory unit that can be addressed in BASIC, consisting of 8 consecutive bits

decimal number a number represented in the base-ten number system using the digita 0-9

expression a combination of one or more operations, constants and variables

file an organized collection of related data

hexadecimal number a number represented in the base-16 number system using the digits 0-9 plus A, B, C, D, E, F

intrinsic function a function (usually a complicated function) that may be "built-in" to the Computer's ROM and may be used directly in a BASIC statement

logical expression an expression which is either True or False: if True, −1 is returned; if False, 0 is returned

machine language the language used directly by the Computer, written as binary-coded instructions

port one of 256 channels through which data can be input to or output from the Computer

RAM Random Access Memory; memory available to the user for writing programs and storing data

ROM Read Only Memory; memory which is permanently programmed and may be read but not written into; LEVEL II BASIC is stored in ROM

routine a sequence of instructions to carry out a certain function

statement a complete instruction in BASIC

string a sequence of alphanumeric characters ranging in length from zero (the "null" string) to 255

subroutine a sequence of instructions for performing a desired function; may be accessed many times from various points in a program

variable a quantity that can take on any of a given set of values

variable name the label by which a given variable is addressed

2/Commands

Whenever a prompt > is displayed, your Computer is in the Command Mode. You can type in a command, **ENTER** it, and the Computer will respond immediately. This chapter describes the commands you'll use to control the Computer – to change modes, begin input and output procedures, alter program memory, etc. All of these commands – except CONT – may also be used inside your program as statements. In some cases this is useful; other times it is just for very specialized applications.

The commands described in this chapter are:

AUTO	CONT	EDIT	SYSTEM
CLEAR	CSAVE	LIST	TROFF
CLOAD	DELETE	NEW	TRON
CLOAD?		RUN	

AUTO *line number, increment*

Turns on an automatic line numbering function for convenient entry of programs – all you have to do is enter the actual program statements. You can specify a beginning line number and an increment to be used between line numbers. Or you can simply type AUTO and hit **ENTER**, in which case line numbering will begin at 10 and use increments of 10. Each time you hit **ENTER**, the Computer will advance to the next line number.

Examples:	to use line numbers
AUTO	10, 20, 30, . . .
AUTO 5,5	5, 10, 15, . . .
AUTO 100	100, 110, 120, . . .
AUTO 100,25	100, 125, 150, . . .

To turn off the AUTO function, hit the BREAK key. (Note: When AUTO brings up a line number which is already being used, an asterisk will appear beside the line number. If you do not wish to re-program the line, hit the BREAK key to turn off AUTO function.)

CLEAR *n*

When used without an argument (e.g., type CLEAR and hit ENTER), this command resets all numeric variables to zero, and all string variables to null. When used with an argument (e.g., CLEAR 100), this command performs a second function in addition to the one just described: it makes the specified number of bytes available for string storage.
Example: CLEAR 100 makes 100 bytes available for strings. When you turn on the Computer a CLEAR 50 is executed automatically.

CLOAD *"file name"*

Lets you load a BASIC program stored on cassette. Place recorder/player in Play mode (be sure the proper connections are made and cassette tape has been re-wound to proper position).

NOTE: In LEVEL II, CLOAD and CSAVE operate at a transfer rate of 500 baud. This is twice as fast as LEVEL I's cassette transfer rate. Therefore the Volume setting used during CLOAD should be correspondingly lower. For example, if you're using Radio Shack's CTR-41 Cassette Recorder, try a setting of between 4 and 6 on the Volume control when loading programs or data you placed on the tape. For loading pre-recorded programs, a higher Volume level may be required. Do a little experimenting.

Entering CLOAD will turn on the cassette machine and load the first program encountered. LEVEL II also lets you specify a desired "file" in your CLOAD command. For example, CLOAD "A" will cause the Computer to ignore programs on the cassette until it comes to one labeled "A". So no matter where file "A" is located on the tape, you can start at the beginning of the tape: file "A" will be picked out of all the files on the tape and loaded. As the Computer is searching for file "A", the names of the files encountered will appear in the upper right corner of the Display, along with a blinking "*".

Only the first character of the file name is used by the Computer for CLOAD, CLOAD?, and CSAVE operations.

Loading a program from tape automatically clears out the previously stored program. See also CSAVE.

CLOAD? *"file name"*

Lets you compare a program stored on cassette with one presently in the Computer. This is useful when you have dumped a program onto tape (using CSAVE) and you wish to check that the transfer was successful. If you labeled the file when you CSAVEd it, you may specify CLOAD? *"file-name"*. Otherwise, if you don't specify a file-name, the first program encountered will be tested. During CLOAD?, the program on tape and the program in memory are compared byte for byte. If there are any discrepancies (indicating a bad dump), the message "BAD" will be displayed. In this case, you should CSAVE the program again. (CLOAD?, unlike CLOAD, does not erase the program memory.)

CONT

When program execution has been stopped (by the BREAK key or by a STOP statement in the program), type CONT and **ENTER** to continue execution at the point where the stop or break occurred. During such a break or stop in execution, you may examine variable values (using PRINT) or change these values. Then type CONT and **ENTER** and execution will continue with the current variable values. CONT, when used with STOP and the BREAK key, is primarily a debugging tool.

NOTE: You cannot use CONT after EDITing your program lines or otherwise changing your program. CONT is also invalid after execution has ended normally.
See also STOP.

CSAVE *"file name"*

Stores the resident program on cassette tape. (Cassette recorder must be properly connected, cassette loaded, and in the Record mode, before you enter the CSAVE command.) You must specify a file-name with this command. This file-name may be any alpha-numeric character other than double-quotes ("). The program stored on tape will then bear the specified file-name, so that it can be located by a CLOAD command which asks for that particular file-name. You should always write the appropriate file-names on the cassette case for later reference.

Examples:

`CSAVE "1"`	dumps resident program and attaches label "1"
`CSAVE "A"`	dumps resident program and attaches label "A"

See also CLOAD.

DELETE *line number-line number*

Erases program lines from memory. You may specify an individual line or a sequence of lines, as follows:

DELETE *line number*	erases one line as specified
DELETE *line number-line number*	erases all program lines starting with first line number specified and ending with last number specified
DELETE-*line number*	erases all program lines up to and including the specified number

The upper line number to be deleted must be a currently used number.

Examples:

DELETE 5	erases line 5 from memory (error it line 5 not used)
DELETE 11-18	erases lines 11, 18 and every line in between

If you have just entered or edited a line, you may delete that line simply by entering DELETE. (use a period instead of the line number).

EDIT *line number*

Puts the Computer in the Edit Mode so you can modify your resident program. The longer and more complex your programs are, the more important EDIT will be. The Edit Mode has its own selection of subcommands, and we have devoted Chapter 9 to the subject.

LIST *line number-line number*

Instructs the Computer to display all program lines presently stored in memory. If you enter LIST without an argument, the entire program will scroll continuously up the screen. To stop the automatic scrolling, press SHIFT and @ simultaneously. This will freeze the display. Press any key to release the "pause" and continue the automatic scrolling.

To examine one line at a time, specify the desired line number as an argument in the LIST command. To examine a certain sequence of program lines, specify the first and last lines you wish to examine.

Examples:

LIST 50	displays line 50
LIST 50-150	displays line 50, 150 and everything in between
LIST 50-	displays line 50 and all higher-numbered lines
LIST.	displays current line (line just entered or edited)
LIST -50	displays all lines up to and including line 50

NEW

Erases all program lines, sets numeric variables to zero and string variables to null. It does not change the string space allocated by a previous CLEAR *number* statement.

RUN *line number*

Causes Computer to execute the program stored in memory. If no line number is specified, execution begins with lowest numbered program line. If a line number is specified, execution begins with the line number. (Error occurs if you specify an unused line number.) Whenever RUN is executed, Computer also executes a CLEAR.

Examples:

RUN	execution begins at lowest-numbered line
RUN 100	execution begins at line 100

RUN may be used inside a program as a statement; it is a convenient way of starting over with a clean slate for continuous-loop programs such as games.

SYSTEM

Puts the Computer in the Monitor Mode, which allows you to load object files (machine-language routines or data). Radio Shack offers several machine-language software packages, such as the IN-MEMORY INFORMATION SYSTEM. You can also create your own object files using the TRS-80 EDITOR/ASSEMBLER, which is itself an object file.

To load an object file: Type SYSTEM and ENTER

*? will be displayed. Now enter the file-name (no quotes are necessary) and the tape will begin loading. When loading is complete, another

*? will be displayed. Type in a slash-symbol / followed by the address (in decimal form) at which you wish execution to begin. Or you may simply hit the slash-symbol and ENTER without any address. In this case execution will begin at the address specified by the object file.

TROFF

Turns off the Trace function. See TRON.

TRON

Turns on a Trace function that lets you follow program-flow for debugging and execution analysis. Eacb time the program advances to a new program line, that line number will be displayed inside a pair of brackets.

For example, enter the following program:

```
10 PRINT "START"
20 PRINT "GOING"
30 GOTO 20
40 PRINT "GONE..
```

Now type in TRON, ENTER , and RUN, ENTER

```
<10>   START
<20>   GOING
<30>   <20> GOING
<30>   <20> GOING
```

etc.

(Press SHIFT and @ simultaneously to pause execution and freeze display. Press any key to continue with execution.)
As you can see from the display, the program is in an infinite loop.

The numbers show you exactly what is going on. (To stop execution, hit BREAK key.)

To turn off the Trace function, enter TROFF. TRON and TROFF may be used inside programs to help you tell when a given line is executed.

For example

```
50  TRON
60  X=X*3.14159
70  TROFF
```

might be helpful in pointing out every time line 60 is executed (assuming execution doesn't jump directly to 60 and bypass 50). Each time these three lines are executed, <60> <70> will be displayed. Without TRON, you wouldn't know whether the program was actually executing line 60. After a program is debugged, TRON and TROFF lines can be removed.

3/Input-Output

The statements described in this chapter let you send data from Keyboard to Computer, Computer to Display, and back and forth between Computer and the Cassette interface. These will primarily be used inside programs to input data and output results and messages.

Statements covered in this chapter:

PRINT
- @ (PRINT modifier)
- TAB (PRINT modifier)
- USING (PRINT formatter)

INPUT
DATA
READ
RESTORE
PRINT # (Output to Cassette)
INPUT # (Input to Cassette)

PRINT *item list*

Prints an item or a list of items on the Display. The items may be either string constants (messages enclosed in quotes), string variables, numeric constants (numbers), variables, or expressions involving all of the preceding items. The items to be PRINTed may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next print zone before printing the next item. If semi-colons are used, no space is inserted between the items printed on the Display.

Examples:

```
50  X=5
100 PRINT 25;"IS EQUAL TO";X ↑ 2
RUN

 25 IS EQUAL TO 25
```

```
10  A$="STRING"
20  PRINT A$;A$,A$;" ";A$
RUN
STRINGSTRING      STRING STRING
```

Positive numbers are printed with a leading blank (instead of a plus sign); all numbers are printed with a trailing blank; and no blanks are inserted before or after strings (you can insert them with quotes as in line 20.

```
10 PRINT "ZONE 1","ZONE 2","ZONE 3","ZONE 4","ZONE 1 ETC"
RUN
ZONE 1          ZONE 2          ZONE 3          ZONE 4
ZONE 1 ETC
```

There are four 16-character print zones per line.

```
10 PRINT "ZONE 1",,"ZONE 3"
RUN
ZONE 1                          ZONE 3
```

The cursor moves to the next print zone each time a comma is encountered.

```
10 PRINT "PRINT STATEMENT #10 ";
20 PRINT "PRINT STATEMENT #20"
RUN
PRINT STATEMENT #10 PRINT STATEMENT #20
```

A trailing semi-colon over-rides the cursor-return so that the next PRINT begins where the last one left off (see line 10).

If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

PRINT @ *position, item list*

Specifies exactly where printing is to begin. (AT was used in LEVEL I BASIC.) The @ modifier must follow PRINT immediately, and the location specified must be a number from 0 to 1023. Refer to the Video Display worksheet, Appendix E, for the exact position of each location 0-1023:

```
100 PRINT @ 550, "LOCATION 550"
```

RUN this to find out where location 550 is.

Whenever you PRINT @ on the bottom line of the Display, there is an automatic line-feed, causing everything displayed to move up one line. To suppress this, use a trailing semi-colon at the end of the statement.

Example:

```
100 PRINT @ 1000, 1000;
```

PRINT TAB (*expression*)

Moves the cursor to the specified position on the current line (or on succeeding lines if you specify TAB positions greater than 63). TAB may be used several times in a PRINT list.

The value of *expression* must be between 0 and 255 inclusive.

Example:

```
10 PRINT TAB(5) "TABBED 5";TAB(25) "TABBED 25"
```

No punctuation is required after a TAB modifier.

```
5  X=3
10 PRINT TAB(X) X: TAB(X ↑ 2) X ↑ 2; TAB(X ↑ 3) X ↑ 3
```

Numerical expressions may be used to specify a TAB position. This makes TAB very useful for graphs of mathematical functions, tables, etc. TAB cannot be used to move the cursor to the left. If cursor is beyond the specified position, the TAB is ignored.

PRINT USING *string; item list*

PRINT USING – This statement allows you to specify a format for printing string and numeric values. It can be used in many applications such as printing report headings, accounting reports, checks . . . or wherever a specific print format is required.

The PRINT USING statement uses the following format:
PRINT USING *string ; value*

String and *value* may be expressed as variables or constants. This statement will print the expression contained in the string, inserting the numeric value shown to the right of the semicolon as specified by the field specifiers.

The following field specifiers may be used in the string:

\# This sign specifies the position of each digit located in the numeric value. The number of # signs you use establishes the numeric field. If the numeric field is greater than the number of digits in the numeric value, then the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.

The decimal point can be placed anywhere in the numeric field established by the # sign. Rounding-off will take place when digits to the right of the decimal point are suppressed.

The comma – when placed in any position between the first digit and the decimal point – will display a comma to the left of every third digit as required. The comma establishes an additional position in the field.

** Two asterisks placed at the beginning of the field will cause all unused positions to the left of the decimal to be filled with asterisks. The two asterisks will establish two more positions in the field.

$$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is, it will occupy the first position preceding the number.

**$ If these three signs are used at the beginning of the field, then the vacant positions to the left of the number will be filled by the * sign and the $ sign will again position itself in the first position preceding the number.

+ When a + sign is placed at the beginning or end of the field, it will be printed as specified as a + for positive numbers or as a – for negative numbers.

– When a – sign is placed at the end of the field, it will cause a negative sign to appear after all negative numbers and will appear as a space for positive numbers.

% spaces % To specify a string field of more than one character, *% spaces %* is used. The length of the string field will be 2 plus the number of spaces between the percent signs.

! Causes the Computer to use the first string character of the current value.

The following program will help demonstrate these format specifiers:

```
10 INPUT A$, A
20 PRINT USING A$;A
30 GOTO 10
```

RUN this program and try various specifiers and strings for A$ and various values for A.

For Example:

```
RUN
? ##.##, 12.12
12.12
? ###.##, 12.12
 12.12
? ##.##, 121.21
% 121.21
```

The % sign is automatically printed if the field is not large enough to contain the number of digits found in the numeric value. The entire number to the left of the decimal will be displayed preceded by this sign.

```
? ##.##, 12.127
12.13
```

Note that the number was rounded to two decimal places.

```
? +##.##,12.12
+12.12
? +##.##, -12.12
-12.12
? ##.##+,12.12
12.12+
? ##.##+, -12.12
12.12-
? ##.##-, 12.12
12.12
? ##.## -, -12.12
12.12-
? **##, 12.12
**12
? **##.##, 1212.12
1212.12
? $$##.##, 12.12
  $12.12
? " ##,####", 12121.2
  12,121.2
? "####,#", 12121.2
  12,121
? ###, 1212
% 1212
```

Another way of using the PRINT USING statement is with the string field specifiers "!" and *% spaces %.*

Examples:

```
PRINT USING "!"; string
PRINT USING "%     %"; string
```

The "!" sign will allow only the first letter of the string to be printed. The "*% spaces %*" allows spaces +2 characters to be printed. Again, the *string* and specifier can be expressed as string variables. The following program will demonstrate this feature:

```
10 INPUT A$, B$
20 PRINT USING A$; B$
30 GOTO 10
```

and RUN it:

```
? !, ABCDE
  A
? %%, ABCDE
  AB
? %   %, ABCD
```

Multiple strings or string variables can be joined together (concatenated) by these specifiers. The "!" sign will allow only the first letter of each string to be printed. For example:

```
10 INPUT A$, B$, C$
20 PRINT USING "!"; A$; B$; C$
```

And RUN it . . .

```
? ABC,DEF,GHI
 ADG
```

By using more than one "!" sign, the first letter of each string will be printed with spaces inserted corresponding to the spaces inserted between the "!" signs. To illustrate this feature, make the following change to the last little program:

```
20 PRINT USING "! ! !"; A$, B$, C$
```

And RUN it . . .

```
? ABC,DEF,GHI
  A D G
```

Spaces now appear between letters A, D and G to correspond with those placed between the three "!" signs.

Try changing "! ! !" to "%%" in line 20 and run the program.

The following program demonstrates one possible use for the PRINT USING statement.

```
 10 CLS
 20 A$ = "**$##,######.## DOLLARS"
 30 INPUT "WHAT IS YOUR FIRST NAME"; F$
 40 INPUT "WHAT IS YOUR MIDDLE NAME"; M$
 50 INPUT "WHAT IS YOUR LAST NAME"; L$
 60 INPUT "ENTER THE AMOUNT PAYABLE"; P
 70 CLS: PRINT "PAY TO THE ORDER OF  ";
 80 PRINT USING "!! !! "; F$; "."; M$; ".";
 90 PRINT L$
100 PRINT: PRINT USING A$; P
110 GOTO 110
```

RUN the program. Remember, to save programming time, use the "?" sign for PRINT. Your display should look something like this:

```
WHAT IS YOUR FIRST NAME? JOHN
WHAT IS YOUR MIDDLE NAME? PAUL
WHAT IS YOUR LAST NAME? JONES
ENTER AMOUNT PAYABLE? 12345.8

PAY TO THE ORDER OF J. P. JONES

*******$12,345.80 DOLLARS
```

If you want to use an amount greater than 999,999 without rounding off or going into scientific notation, then simply add the double precision sign (#) after the variable P in Lines 60 and 100. You will then be able to use amounts up to 16 decimal places long.

INPUT *item list*

Causes Computer to stop execution until you enter the specified number of values via the keyboard. The INPUT statement may specify a list of string or numeric variables to be input. The items in the list must be separated by commas.

```
100 INPUT X$, X1, Z$, Z1
```

This statement calls for you to input a string-literal, a number, another string literal, and another number, in that order. When the statement is encountered, the Computer will display a

```
?_
```

You may then enter the values all at once or one at a time. To enter values all at once, separate them by commas. (If your string literal includes leading blanks, colons, or commas, you must enclose the string in quotes.)

For example, when line 100 (above) is RUN and the Computer is waiting for your input, you could type

```
JIM,50,JACK,40        (ENTER)
```

The Computer will assign values as follows:

```
X$="JIM"      X1=50      Z$="JACK"      Z1=40
```

If you ENTER the values one at a time, the Computer will display a

```
??_
```

. . . indicating that more data is expected. Continue entering data until all the variables have been set, at which time the Computer will advance to the next statement in your program.

Be sure to enter the correct type of value according to what is called for by the INPUT statement. For example, you can't input a string-value into a numerical variable. If you try to, the Computer will display a

```
?REDO
?_
```

and give you another chance to enter the correct type of data value, starting with the *first* value called for by the INPUT list.

NOTE: You cannot input an expression into a numerical value – you must input a simple numerical constant. (LEVEL I allowed you to input an expression or even a variable into a numerical variable.)

Example:

```
100 INPUT X1, Y1$
200 PRINT X1, Y1$
RUN
?_ [you type:] 7+3 ( ENTER )
? REDO
?_ [you type:] 10  ( ENTER )
??_ [you type:] "THIS IS A COMMA: ,"
   10             THIS IS A COMMA:,
```

It was necessary to put quotes around "THIS IS A COMMA:," because the string contained a comma.

If you ENTER more data elements than the INPUT statement specifies, the Computer will display the message

```
?EXTRA IGNORED
```

and continue with normal execution of your program.

You can also include a "prompting message" in your INPUT statement. This will make it easier to input the data correctly. The prompting message must immediately follow "INPUT", must be enclosed in quotes, and must be followed by a semi-colon.

Example:

```
100 INPUT "ENTER YOUR NAME AND AGE (NAME,AGE)";N$,A
(RUN)
ENTER YOUR NAME AND AGE (NAME,AGE)?_
```

DATA *item list*

Lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last Item in the last DATA statement. Items in a DATA list may be string or numeric constants – no expressions are allowed. If your string values include leading blanks, colons or commas, you must enclose these values in quotes.

It is important that the data types in a DATA statement match up with the variable types in the corresponding READ statement. DATA statements may appear anywhere it is convenient in a program. Generally, they are placed consecutively, but this is not required.

Examples:

```
500    READ N1$,N2$,N1,N2
1000   DATA "SMITH, J.R.","WILSON, T.M."
2000   DATA 150,175
```

See READ, RESTORE.

READ *item list*

Instructs the Computer to read a value from a DATA statement and assign that value to the specified variable. The first time a READ is executed, the first value in the first DATA statement will be used; the second time, the second value in the DATA statement will be read. When all the items in the first DATA statement have been read, the next READ will use the first value in the second DATA statement; etc. (An Out-of-Data error occurs if there are more attempts to READ than there are DATA items.) The following program illustrates a common application for READ/DATA statements.

```
50     PRINT "NAME","AGE"
100    READ N$
110    IF N$="END" PRINT "END OF LIST":END
120    READ AGE
130    IF AGE < 18 PRINT N$,AGE
140    GOTO100
150    DATA "SMITH, JOHN",30,"ANDERSON,T.M.",20
160    DATA "JONES, BILL",15,"DOE,SALLY",21
170    DATA "COLLINS,W.P.",17,END
RUN

NAME          AGE
JONES, BILL    15
COLLINS,W.P.   17
END OF LIST


READY
>_
```

The program locates and prints all the minors' names from the data supplied. Note the use of an END string to allow READing lists of unknown length.

See DATA, RESTORE

RESTORE

Causes the next READ statement executed to start over with the first item in the first DATA statement. This lets your program re-use the same DATA lines.

Example:

```
100   READ X
110   RESTORE
120   READ Y
130   PRINT X,Y
140   DATA 50,60
RUN
 50             50
READY
>_
```

Because of the RESTORE statement, the second READ statement starts over with the first DATA item.

See READ, DATA

PRINT #-1, *item list*

Prints the values of the specified variables onto cassette tape. (Recorder must be properly connected and set in Record mode when this statement is executed.) The PRINT # statement must always specify a device number. This is because the TRS-80 can actually input/output to two cassette machines, once you've added the Expansion Interface described in Chapter 10. For normal use with just one recorder connected, the device number must be -1, e.g., PRINT #-1 (followed by a comma and then the item list).

Example:

```
5   A1=-30.334:B$="STRING VALUE"
10  PRINT#-1,A1,B$,"THAT'S ALL"
```

This stores the current values of A1 and B$, and also the string-literal "THAT'S ALL". The values may be input from tape later using the INPUT# statement. The INPUT# statement must be identical to the PRINT# statement in terms of number and type of items in the PRINT#/INPUT# lists. See INPUT#.

Special Note:

The values represented in *item list* must not exceed 255 characters total; otherwise all characters after the first 255 will be truncated. For example, `PRINT#-1, A$,B$,C$,D$,E$,F$,G$,H$,I$,J$,A$` will probably exceed the maximum record length if A$ is longer than about 75 characters. In such a case, A$ would not be recorded, and when you try to INPUT#-1 the data, an OD (Out of Data) error will occur.

INPUT #-1, *item list*

Inputs the specified number of values stored on cassette and assigns them to the specified variable names. Like the PRINT# statement, INPUT# requires that you specify a device number. (This will make more sense when you have added the Expansion Interface and are using a dual cassette system. See Chapter 10.) Use Device number -1 for normal applications without the Expansion Interface. e.g., INPUT#-1, *list*.

Example:

```
50   INPUT #-1,X,P$,T$
```

When this statement is executed, the Computer will turn on the tape machine, input values in the order specified, then turn off the tape machine and advance to the next statement. If a string is encountered when the INPUT list calls for a number, a bad file data error will occur. If there are not enough data items on the tape to "fill" the INPUT statement, an Out of Data error will occur.

The Input list must be identical to the Print list that created the taped data-block (same number and type of variables in the same sequence.)

Sample Program

Use the two-line program supplied in the PRINT# description to create a short data file. Then rewind the tape to the beginning of the data file, make all necessary connections, and put cassette machine in Play mode. Now run the following program.

```
10   INPUT#-1,A1,B$,L$
20   PRINT A1,B$,L$
30   IF L$="THAT'S ALL"END
40   GOTO 10
```

This program doesn't care how long or short the data file is, so long as:

1) the file was created by successive PRINT# statements identical in form to line 10
2) the last item in the last data triplet is "THAT'S ALL".

4/Program Statements

LEVEL II BASIC makes several assumptions about how to run your programs. For example:

* Variables are assumed to be single-precision (unless you use type declaration characters – see Chapter 1, "Variable Types").
* A certain amount of memory is automatically set aside for strings and arrays – whether you use all of it or not.
* Execution is sequential, starting with the first statement in your program and ending with the last.

The statements described in this chapter let you over-ride these assumptions, to give your programs much more versatility and power.

NOTE: All LEVEL II statements except INPUT and INPUT# can be used in the Command Mode as well as in the Execute Mode.

Statements described in this chapter:

Type Definition	Assignment & Allocation	Sequence of Execution	Tests (Conditional Statements)
DEFINT	CLEAR *n*	END	IF
DEFSNG	DIM	STOP	THEN
DEFDBL	LET	GOTO	ELSE
DEFSTR		GOSUB	
		ON . . . GOTO	
		ON . . . GOSUB	
		FOR-NEXT-STEP	
		ERROR	
		ON ERROR GOTO	
		RESUME	
		REM	

This chapter also contains a discussion of data conversion in LEVEL II BASIC; this will let you predict and control the way results of expressions, constants, etc., will be stored – as integer, single precision or double precision.

DEFINT *letter range*

Variables beginning with any letter in the specified range will be stored and treated as integers, unless a type declaration character is added to the variable name. This lets you conserve memory, since integer values take up less memory than other numeric types. And integer arithmetic is faster than single or double precision arithmetic. However, a variable defined as integer can only take on values between −32768 and +32767 inclusive.

Examples:

```
10  DEFINT A,I,N
```

After line 10, all variables beginning with A, I or N will be treated as integers. For example, A1, AA, I3 and NN will be integer variables. However, A1#, AA#, I3# would still be double precision variables, because of the type declaration characters, which always over-ride DEF statements.

```
10  DEFINT I-N
```

Causes variables beginning with I, J, K, L, M or N to be treated as integer variables.

DEFINT may be placed anywhere in a program, but it may change the meaning of variable references without type declaration characters. Therefore it is normally placed at the beginning of a program.

See DEFSNG, DEFDBL, and Chapter 1, "Variable Types".

DEFSNG *letter range*

Causes any variable beginning with a letter in the specified range to be stored and treated as single precision, unless a type declaration character is added. Single precision variables and constants are stored with 7 digits of precision and printed out with 6 digits of precision. Since all numeric variables are assumed to be single precision unless DEFined otherwise, the DEFSNG statement is primarily used to re-define variables which have previously been defined as double precision or integer.

Example:

```
100  DEFSNG I, W-Z
```

Causes variables beginning with the letter I or any letter W through Z to be treated as single precision. However, I% would still be an integer variable, and I# a double precision variable, due to the use of type declaration characters.

See DEFINT, DEFDBL, and Chapter 1, "Variable Types".

DEFDBL *letter range*

Causes variables beginning with any letter in the specified range to be stored and treated as double-precision, unless a type declaration character is added. Double precision allows 17 digits of precision; 16 digits are displayed when a double precision variable is PRINTed.

Example:

```
10   DEFDBL S-Z, A-E
```

Causes variables beginning with one of the letters S through Z or A through E to be double precision.

DEFDBL is normally used at the beginning of a program, because it may change the meaning of variable references without type declaration characters.

See DEFINT, DEFSNG, and Chapter 1, "Variable Types".

DEFSTR *letter range*

Causes variables beginning with one of the letters in the specified range to be stored and treated as strings, unless a type declaration character is added. If you have CLEARed enough string storage space, each string can store up to 255 characters.

Example:

```
10   DEFSTR L-Z
```

Causes variables beginning with any letter L through Z to be string variables, unless a type declaration character is added. After line 10 is executed, the assignment L1 = "WASHINGTON" will be valid.

See CLEAR *n*, Chapter 1, "Variable Types", and Chapter 5.

CLEAR *n*

When used with an argument *n* (*n* can be a constant or an expression), this statement causes the Computer to set aside *n* bytes for string storage. In addition all variables are set to zero. When the TRS-80 is turned on, 50 bytes are automatically set aside for strings.

The amount of string storage CLEARed must equal or exceed the greatest number of characters stored in string variables during execution; otherwise an Out of String Space error will occur.

Example:

```
10   CLEAR 1000
```

Makes 1000 bytes available for string storage.

By setting string storage to the exact amount needed, your program can make more efficient use of memory. A program which uses no string variables could include a CLEAR 0 statement, for example. The CLEAR argument must be non-negative, or an error will result.

DIM *name (dim1, dim2, . . ., dimK)*

Lets you set the "depth" (number of elements allowed per dimension) of an array or list of arrays. If no DIM statement is used, a depth of 11 (subscripts 0-10) is allowed for each dimension of each array used.

Example:

```
10   DIM A(5),B(2,3),C$(20)
```

Sets up a one-dimension array A with subscripted elements 0-5; a two-dimension array B with subscripted elements 0,0 to 2,3; and a one-dimension string array C$ with subscripted elements 0-20. Unless previously defined otherwise, arrays A and B will contain single-precision values.

DIM statements may be placed anywhere in your program, and the depth specifier may be a number or a numerical expression.

Example:

```
40   INPUT "NUMBER OF NAMES";N
50   DIM NA(N,2)
```

To re-dimension an array, you must first use a CLEAR statement, either with or without an argument. Otherwise an error will result.

Example Program:

```
10   AA(4) = 11.5
20   DIM AA(7)
RUN
?DD  ERROR IN 20
```

See Chapter 6, ARRAYS.

LET *variable = expression*

May be used when assigning values to variables. RADIO SHACK LEVEL II does not require LET with assignment statements, but you might want to use it to ensure compatibility with those versions of BASIC that do require it.

Examples:

```
100 LET A$="A ROSE IS A ROSE"
110 LET B1=1.23
120 LET X=X-Z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

END

Terminates execution normally (without a BREAK message). Some versions of BASIC require END as the last statement in a program; with LEVEL II it is optional. END is primarily used to force execution to terminate at some point other than the logical end of the program.

Example:

```
10   INPUT S1,S2
20   GOSUB 100
.
.
.
99   END
100 H=SQR(S1*S1 + S2*S2)
110 RETURN
```

The END statement in line 99 prevents program control from "crashing" into the subroutine. Now line 100 can only be accessed by a branching statement such as 20 GOSUB 100.

STOP

Interrupts execution and prints a BREAK IN *line number* message. STOP is primarily a debugging aid. During the break in execution, you can examine or change variable values. The command CONT can then be used to re-start execution at the point where it left off. (If the program itself is altered during a break, CONT cannot be used.)

Example:

```
10   X=RND(10)
15   STOP
20   GOSUB 1000

RUN

BREAK IN 15
READY
>_
```

Suppose we want to examine what value for X is being passed to the subroutine beginning at line 1000. During the break, we can examine X with PRINT X. (You can delete line 15 after the program is debugged.)

GOTO *line number*

Transfers program control to the specified line number. Used alone, GOTO *line number* results in an unconditional (or automatic) branch; however, test statements may precede the GOTO to effect a conditional branch.

Example:

```
200 GOTO 10
```

When 200 is executed, control will automatically jump back to line 10.

You can use GOTO in the Command Mode as an alternative to RUN. GOTO *line number* causes execution to begin at the specified line number, without an automatic CLEAR. This lets you pass values assigned in the Command Mode to variables in the Execute Mode.

See IF,THEN,ELSE,ON... GOTO.

GOSUB *line number*

Transfers program control to the subroutine beginning at the specified line number. When the Computer encounters a RETURN statement in the subroutine, it will then return control to the statement which follows GOSUB. GOSUB, like GOTO may be preceded by a test statement. See IF,THEN,ELSE,ON... GOSUB.

Example Program:

```
100 GOSUB 200
110 PRINT "BACK FROM SUBROUTINE": END
200 PRINT "EXECUTING THE SUBROUTINE"
210 RETURN
```

(RUN)

```
EXECUTING THE SUBROUTINE
BACK FROM THE SUBROUTINE
```

Control branches from line 100 to the subroutine beginning at line 200. Line 210 instructs Computer to return to the statement immediately following GOSUB, that is, line 110.

RETURN

Ends a subroutine and returns control to statement immediately following the most recently executed GOSUB. If RETURN is encountered without execution of a matching GOSUB, an error will occur. See GOSUB.

ON *n* GOTO *line number, . . ., line number*

This is a multi-way branching statement that is controlled by a test variable or expression. The general format for ON *n* GOTO is:

ON *expression* GOTO *1st line number, 2nd line number, . . . , Kth line number*

expression must be between Ø and 255 inclusive.

When ON . . . GOTO is executed, first the expression is evaluated and the integer portion . . . INT(expression) . . . is obtained. We'll refer to this integer portion as J. The Computer counts over to the Jth element in the line-number list, and then branches to the line number specified by that element. If there is no Jth element (that is, if J > K in the general format above), then control passes to the next statement in the program.

If the test expression or number is less than zero, an error will occur. The line-number list may contain any number of items.

For example,

```
100 ON MI GOTO 150, 160, 170, 150, 180
```

says "Evaluate MI. If integer portion of MI equals 1 then go to line 150;
If it equals 2, then go to 160;
If it equals 3, then go to 170;
If it equals 4, then go to 150;
If it equals 5, then go to 180;
If the integer portion of MI doesn't equal any of the numbers 1 through 5, advance to the next statement in the program."

Sample Program Using ON *n* GOTO

```
100 INPUT "ENTER A NUMBER":X
200 ON SGN(X)+2 GOTO 220,230,240
220 PRINT "NEGATIVE":END
230 PRINT "ZERO":END
240 PRINT "POSITIVE":END
```

SGN(X) returns −1 for X less than zero; 0 for X equal to zero; and +1 for X greater than 0. By adding 2, the expression takes on the values 1, 2, and 3, depending on whether X is negative, zero, or positive. Control then branches to the appropriate line number.

ON *n* GOSUB *line number, . . ., line number*

Works like ON *n* GOTO, except control branches to one of the subroutines specified by the line numbers in the line-number list.

Example:

```
100 INPUT "CHOOSE 1, 2 OR 3";I
105 ON I GOSUB 200,300,400
110 END
200 PRINT "SUBROUTINE #1":RETURN
300 PRINT "SUBROUTINE #2":RETURN
400 PRINT "SUBROUTINE #3":RETURN
```

The test object *n* may be a numerical constant, variable or expression. It must have a non-negative value or an error will occur.

See ON *n* GOTO.

FOR *name = exp* **TO** *exp* **STEP** *exp*
NEXT *name*

Opens an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times. The general form is (brackets indicate optional material):

line # FOR *counter-variable = initial value* TO *final value* [STEP *increment*]
.
. [*program statements*]
.
line # NEXT [*counter-variable*]

In the FOR statement, *initial value*, *final value* and *increment* can be constants, variables or expressions. The first time the FOR statement is executed, these three are evaluated and the values are saved; if the variables are changed by the loop, it will have no effect on the loop's operation. However, the counter variable must not be changed or the loop will not operate normally.

The FOR-NEXT-STEP loop works as follows: the first time the FOR statement is executed, the counter is set to the "initial value." Execution proceeds until a NEXT statement is encountered. At this point, the counter is incremented by the amount specified in the STEP *increment*. (If the *increment* has a negative value, then the counter is actually decremented.) If STEP *increment* is not used, an increment of 1 is assumed.

Then the counter is compared with the *final value* specified in the FOR statement. If the counter is greater than the *final value*, the loop is completed and execution continues with the statement following the NEXT statement. (If *increment* was a negative number, loop ends when counter is less than *final value*.) If the counter has not yet exceeded the *final value*, control passes to the first statement after the FOR statement.

Example Programs:

```
10  FOR I=10 TO 1 STEP -1
20  PRINT I;
30  NEXT

RUN

 10  9  8  7  6  5  4  3  2  1
READY
>_
```

```
10  FOR K=0 TO 1 STEP .3
20  PRINT K;
30  NEXT
```

```
RUN

 .0   .3   .6   .9
READY
>_
```

After K=.9 is incremented by .3, K=1.2. This is greater than the *final value* 1, therefore loop ends without ever printing *final value.*

```
10   FOR = 4 TO 0
20   PRINT K;
30   NEXT

RUN

 4
READY
>_
```

No STEP is specified, so STEP 1 is assumed After K is incremented the first time, its value is 5. Since 5 is greater than the *final value* 0, the loop ends.

```
10   J=3 : K=8 : L=2
20   FOR I=J TO K+1 STEP L
25   J=0 : K=0 : J=0
30   PRINT I;
40   NEXT

RUN

  3  5  7  9
READY
>_
```

The variables and expressions in line 20 are evaluated once and these values become constants for the FOR-NEXT-STEP loop. Changing the variable values later has no effect on the loop.

FOR-NEXT loops may be "nested":

```
10   FOR I=1TO3
20   PRINT "OUTER LOOP"
30       FOR J=1 TO 2
40       PRINT "   INNER LOOP"
50       NEXT J
60   NEXT I
```

```
RUN

OUTER LOOP
   INNER LOOP
   INNER LOOP
OUTER LOOP
   INNER LOOP
   INNER LOOP
OUTER LOOP
   INNER LOOP
   INNER LOOP
```

Note that each NEXT statement specifies the appropriate counter variable; however, this is just a programmer's convenience to help keep track of the nesting order. The counter variable may be omitted from the NEXT statements. But if you do use the counter variables, you must use them in the right order; i.e., the counter variable for the innermost loop must come first.

It is also advisable to specify the counter variable with NEXT statements when your program allows branching to program lines outside the FOR-NEXT loop.

Another option with nested NEXT statements is to use a counter variable list.

Delete line 50 from the above program and change line 60:

```
60   NEXT J,I
```

Loops may be nested 3-deep, 4-deep, etc. The only limit is the amount of memory available.

ERROR *code*

Lets you "simulate" a specified error during program execution. The major use of this statement is for testing an ON ERROR GOTO routine. When the ERROR *code* statement is encountered, the Computer will proceed exactly as if that kind of error had occurred. Refer to Appendix B for a listing of error codes and their meanings.

Example Program:

```
100  ERROR 1

RUN

?NF ERROR
READY
>_
```

1 is the error code for "attempt to execute NEXT statement without a matching FOR statement".

See ON ERROR GOTO, RESUME.

ON ERROR GOTO *line number*

When the Computer encounters any kind of error in your program, it normally breaks out of execution and prints an error message. With ON ERROR GOTO, you can set up an error-trapping routine which will allow your program to "recover" from an error and continue, without any break in execution. Normally you have a particular type of error in mind when you use the ON ERROR GOTO statement. For example, suppose your program performs some division operations and you have not ruled out the possibility of division by zero. You might want to write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

Example:

```
5   ON ERROR GOTO 100
10  C = 1/0
```

In this "loaded" example, when the Computer attempts to execute line 10, a divide-by-zero error will occur. But because of line 5, the Computer will simply ignore line 10 and branch to the error-handling routine beginning at line 100.

NOTE: The ON ERROR GOTO must be executed before the error occurs or it will have no effect.

The ON ERROR GOTO statement can be disabled by executing an ON ERROR GOTO 0. If you use this inside an error-trapping routine, BASIC will handle the current error normally.

The error handling routine must be terminated by a RESUME statement. See RESUME.

RESUME *line number*

Terminates an error handling routine by specifying where normal execution is to resume.

RESUME without a line number and RESUME 0 cause the Computer to return to the statement in which the error occurred.

RESUME followed by a line number causes the Computer to branch to the specified line number.

RESUME NEXT causes the Computer to branch to the statement following the point at which the error occurred.

Sample Program with an Error Handling Routine

```
5   ON ERROR GOTO 100
10  INPUT "SEEKING SQUARE ROOT OF";X
20  PRINT SQR(X)
30  GOTO 10
100 PRINT "IMAGINARY ROOT:"; SQR(-X);"*I"
110 RESUME 10
```

RUN the program and try inputting a negative value.

REM

Instructs the Computer to ignore the rest of the program line. This allows you to insert comments (REMarks) into your program for documentation. Then, when you (or someone else) look at a listing of your program, it'll be a lot easier to figure out. If REM is used in a multi-statement program line, it must be the last statement.

Examples Program:

```
10 REM  ** THIS REMARK INTRODUCES THE PROGRAM **
20 REM  ** AND POSSIBLY THE PROGRAMMER, TOO.    **
30 REM  **                                      **
40 REM  ** THIS REMARK EXPLAINS WHAT THE        **
50 REM  ** VARIOUS VARIABLES REPRESENT:         **
60 REM  ** C = CIRCUMFERENCE  R = RADIUS        **
70 REM  ** D = DIAMETER                         **
80 REM
90 INPUT "RADIUS";R : REM  THIS IS FIRST EXECUTABLE LINE
```

The above program shows some of the graphic possibilities of REM statements. Any alphanumeric character may be included in a REM statement, and the maximum length is the same as that of other statements: 255 characters total.

IN LEVEL II BASIC, an apostrophe '(SHIFT 7) may be used as an abbreviation for :REM.

```
100 ' THIS TOO IS A REMARK
```

IF *true/false expression action-clause*

Instructs the Computer to test the following logical or relational expression. If the expression is True, control will proceed to the "action" clause immediately following the expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line.

In numerical terms, if the expression has a non-zero value, it is always equivalent to a logical True.

Examples:

```
100 IF X >127 PRINT "OUT OF RANGE": END
```

If X is greater than 127, control will pass to the PRINT statement and then to the END statement. But if X is not greater than 127, control will jump down to the next line in the program, skipping the PRINT and END statements.

```
100 IF 0<=X AND X<=90 THEN Y=X+180
```

If both expressions are True then Y will be assigned the value X+180. Otherwise control will pass directly to the next program line, skipping the THEN clause.

NOTE: THEN is optional in the above and similar statements. However, THEN is sometimes required to eliminate an ambiguity. For example, `400 IF Y=M THEN M=0` won't work without THEN.

```
500 INPUT A$ : IF A$="YES" THEN 100
600 INPUT A$ : IF A$="YES" GOTO 100
```

The two statements have the same effect. THEN is not optional in line 500 and other `IF` *expression* `THEN` *line number* statements.

```
100 IF A>0 AND B>0 PRINT "BOTH POSITIVE"
```

The test expression may be composed of several relational expressions joined by logical operators AND and OR.

See THEN, ELSE.

THEN *statement or line number*

Initiates the "action clause" of an IF-THEN type statement. THEN is optional except when it is used to specify a branch to another line number, as in `IF A<0 THEN 100`. THEN should also be used in IF-THEN-ELSE statements.

ELSE *statement or line number*

Used after IF to specify an alternative action in case the IF test fails. (When no ELSE statement is used, control falls through to the next program line after a test fails.)

Examples:

```
100 INPUT A$ : IF A$="YES" THEN 300 ELSE END
```

In line 100, if A$ equals "YES" then the program branches to line 300. But if A$ does not equal "YES", program skips over to the ELSE statement which then instructs the Computer to end execution.

```
200 IF A<B PRINT"A<B"ELSE PRINT "B<=A"
```

If A is less than B, the Computer prints that fact, and then proceeds down to the next program line, skipping the ELSE statement.
If A is not less than B, Computer jumps directly to the ELSE statement and prints the specified message. Then control passes to the next statement in the program.

```
200 IF A>.001 THEN B=1/A : A = A/5 : ELSE 260
```

If A >.001 is True, then the next two statements will be executed, assigning new values to B and A. Then the program will drop down to the next line, skipping the ELSE statement. But if A >.001 is

False, the program jumps directly over to the ELSE statement, which then instructs it to branch to line 260. Note that GOTO is not required after ELSE.

IF-THEN-ELSE statements may be nested, but you have to take care to match up the IFs and ELSEs.

```
10  INPUT "ENTER TWO NUMBERS";A,B
20  IF A< =B THEN IF A<B PRINT A;:ELSE PRINT "NEITHER";:ELSE PRINT B;
30  PRINT"IS SMALLER"
```

RUN the program, inputting various pairs of numbers. The program picks out and prints the smaller of any two numbers you enter. Note that the THEN statements and the colons may be omitted from line 20.

Data Conversion

Every number used during execution must be typed as either integer, single precision or double precision. Often this typing involves converting a number from one form to another. This may produce unexpected, confusing results -- unless you understand the rules governing such automatic typing and type conversion.

Type Conversion

Constants are the actual numbers (not the variable names) used by LEVEL II BASIC during execution. They may appear in your program (as in X=1/3, the right side of the equation) or they may be temporary (intermediate) constants created during the evaluation of an expression. In any case, the following rules determine how a constant is typed:

I. If a constant contains 8 or more digits, or if D is used in the exponent, that number is stored as double precision. Adding a # declaration character also forces a constant to be stored as double precision.

II. If the number is not double-precision, and if it is outside the range −32768 to +32767 or if it contains a decimal point, then the number is stored as single-precision. If number is expressed in exponential notation with E preceding the exponent, the number is single precision.

III. If neither I nor II is true of the constant, then it is stored as an integer.

Example Program:

```
10  PRINT 1.234567, 1.2345678
RUN

 1.23457          1.2345678
READY
>_
```

The first constant contains 7 digits; so by Rules I and II, it becomes a single-precision number. Single precision numbers are printed as 6 digits with the least significant digit properly rounded. But the second constant contains 8 digits, therefore by Rule I it becomes a double precision number, stored internally as 1.2345678000000000. The number is printed out with all eight significant digits showing, and all the trailing zeros suppressed.

Typing of Constants

When operations are performed on one or two numbers, the result must be typed as integer, double or single-precision.

When a +, −, or * operation is performed, the result will have the same degree of precision as the most precise operand. For example, if one operand is single-precision, and the other double-precision, the result will be double precision. Only when both operands are integers will a result be integer. If the result of an integer *, −, or + operation is outside the integer range, the operation will be done in single precision and the result stored as single precision.

Division follows the same rules as +, * and −, except that it is never done at the integer level: when both operators are integers, the operation is done in single precision with a single-precision result.

During a compare operation (<, >,=,etc.) the operands are converted to the same type before they are compared. The less precise type will always be converted to the more precise type.

If you are using logical operators for bit manipulations or Boolean operations (see Chapter 8, "Logical Operators"), you'll need to read the next paragraph; otherwise, skip it.

The logical operators AND, OR and NOT first convert their operands to integer form. If one of the operands is outside the allowable range for integers (−32768 to +32767) an overflow error occurs. The result of a logical operation is always an integer.

Effects of Type Conversions on Accuracy

When a number is converted to integer type, it is "rounded down"; i.e., the largest integer which is not greater than the number is used. (This is the same thing that happens when the INT function is applied to the number.)

When a number is converted from double to single precision, it is "4/5 rounded" (the least significant digit is rounded up if the fractional part > =5. Otherwise it is left unchanged).

In the following examples, keep in mind that single precision variables are stored with 7 digits of precision, but printed out with 6 digits (to allow for proper rounding). Similarly, double precision values are stored with 17 digits but printed out with only 16.

Example Programs:

```
10   A#=1.666666666666667
20   B!=A#
30   C%=A#
40   PRINT B!,C%
RUN

 1.66667        1
READY
>_
```

When a single precision number is converted to double precision, only the seven most significant digits will be accurate. And if the single precision number didn't contain seven significant digits, watch out!

Examples:

```
10   A!=1.3
20   A#=A!
30   PRINT A#

RUN

1.299999952316284
READY
>_
```

```
10   A#=2/3
20   PRINT A#

RUN

.6666666865348816
READY
>_
```

2/3 is converted to a single precision constant; therefore only the first seven digits of A# are accurate.

```
10   A#=2/3#
20   PRINT A#

RUN

.6666666666666667
READY
>_
```

Since the expression 2/3# is evaluated as a double precision constant, all 16 digits of A# are accurate, with the least significant properly 4/5– rounded.

When assigning a constant value to a double precision variable, be sure to include as many significant digits as possible (up to 17). If your constant has seven or less significant digits, you might as well use single precision.

Examples:

```
10   PI#=3.1415926535897932
20    E#=2.7182818284590452
```

5/Strings

"Without string-handling capabilities, a computer is just a super-powered calculator." There's an element of truth in that exaggeration; the more you use the string capabilities of LEVEL II, the truer the statement will seem.

LEVEL I BASIC offered two string variables which could be input and output to make your programs look "friendly" (as in HELLO, BOB!). In LEVEL II you can do much more than that. First of all, you're not limited to two strings – any valid variable name can be used to contain string values, by the DEFSTR statement or by adding a type declaration character to the name. And each string can contain up to 255 characters.

Moreover, you can compare strings in LEVEL II, to alphabetize them, for example. You can take strings apart and string them together (concatenate them). For background material to this chapter, see Chapter 1, "Variable Types" and "Glossary", and Chapter 4, DEFSTR.

Subjects and functions covered in this chapter:

"String Input/Output"	FRE (*string*)	MID$
"String Comparisons"	INKEY$	RIGHT$
"String Operations"	LEN	STR$
ASC	LEFT$	STRING$
CHR$		VAL
		INSTRING Subroutine

String Input/Output

String constants – sequences of alphanumeric characters – may be input to a program just as numeric constants are input, using INPUT, READ/DATA, and INPUT # (input from cassette). They may generally be input without quotes:

```
10  INPUT "YES OR NO";R$
20  IF R$="YES"PRINT"THAT'S BEING POSITIVE!":END
30  PRINT "WHY NOT?"

RUN

YES OR NO?_[you type] YES ENTER
THAT'S BEING POSITIVE!
READY
>_
```

However, to input a string constant which contains commas, colons, or leading blanks, the string must be enclosed in quotes.

```
10   INPUT "LAST NAME, FIRST NAME";N$
20   PRINT N$
```

```
RUN
```

```
LAST NAME, FIRST NAME? _ [you type:] "SMITN, JONN"
ENTER
SMITN, JOHN
READY
>_
```

The same rule regarding commas, colons and leading blanks applies to values input via DATA statements and INPUT # statements.

```
10   READ T$, N$, D$
20   PRINT T$;N$;D$
30   DATA "TOTAL IS:  ","ONE TNOUSAND, TWO HUNDRED  "
40   DATA  DOLLARS.
```

T$ requires quotes because of the colon;
N$ requires quotes because of the comma.

String Comparisons

Strings may be compared for equality or alphabetic precedence. When they are checked for equality, every character, including any leading or trailing blanks, must be the same or the test fails.

```
600 IF Z$="END"TNEN999
```

Strings are compared character-for-character from left to right. Actually the ASCII codes for the characters are compared, and the character with the lower code number is considered to precede the other character. (See Appendix C, ASCII Codes.)

For example, the constant "A!" precedes the constant "A#", because "!" (ASCII code: decimal 33) precedes "#" (ASCII code: decimal 35). When strings of differing lengths are compared, the shorter string is precedent if its characters are the same as those in the longer string. For example, "A" precedes "A   ".

The following relational symbols may be used to compare strings;

```
=     <>     <     <=     >     >=
```

Note: Whenever a string constant is used in a comparison expression or an assignment statement, the constant must be enclosed in quotes:

```
A$="CONSTANT"
IF A$ = "CONSTANT" PRINT A$
```

(The quotes are required in both cases.)

String Operations

Not including the functions described below, there is only one string operation – concatenation, represented by the plus symbol +.

Example Programs:

```
10   CLEAR 75
20   A$="A ROSE"
30   B$=" IS A ROSE"
40   C$=A$+B$+B$+B$+"."
50   PRINT C$

RUN

A ROSE IS A ROSE IS A ROSE IS A ROSE.
READY
>_
```

In line 40, the strings are concatenated – strung together.

```
10   T$="100"
20   SUB$="5"
30   CODE$="32L"
40   LC$=T$+"."+SUB$+CODE$
50   PRINT LC$

RUN

100.532L
READY
> _
```

ASC (*string*)

Returns the ASCII code (in decimal form) for the first character of the specified string. The string-argument must be enclosed in parentheses. A null-string argument will cause an error to occur.

```
100 PRINT ASC("A")
110 T$="AB": PRINT ASC(T$)
```

Lines 100 and 110 will print the same number.

The argument may be an expression involving string operators and functions:

```
200 PRINT ASC(RIGHT$(T$,1))
```

Refer to the ASCII Code Table, Appendix C. Note that the ASCII code for a lower-case letter is equal to that letter's upper-case ASCII code plus 31. So ASC may be used to convert upper-case values to lower-case values – useful in case you have a line printer with lower-case capabilities and the proper interfacing hardware/software).

ASC may also be used to create coding/decoding procedures (see example at end of this chapter).

CHR$ (*expression*)

Performs the inverse of the ASC function: returns a one-character string whose character has the specified ASCII, control or graphics code. The argument may be any number from 0 to 255, or any variable expression with a value in that range. Argument must be enclosed in parentheses.

```
100 PRINT CHR$(35)
```

prints a pound-sign #

Using CHR$, you can even assign quote-marks (normally used as string-delimiters) to strings. The ASCII code for quotes " is 34. So A$=CHR$(34) assigns the value " to A$.

```
100 A$=CHR$(34)
110 PRINT"HE SAID, ";A$;"HELLO.";A$

RUN

HE SAID, "HELLO."
READY
>_
```

CHR$ may also be used to display any of the 64 graphics characters. (See Appendix C, Graphics Codes.)

```
10  CLS
20  FOR I=129 TO 191
30  PRINT I;CHR$(I),
40  NEXT
50  GOTO 50
```

(RUN the program to see the various graphics characters.)

Codes 0-31 are display control codes. Instead of returning an actual display character, they return a control character. When the control character is PRINTed, the function is performed. For example, 23 is the code for 32 character-per-line format; so the command, PRINT CHR$(23) converts the display format to 32 characters per line. (Hit CLEAR, or execute CLS, to return to 64 character-per-line format.)

FRE (*string*)

When used with a string variable or string constant as an argument, returns the amount of string storage space currently available. Argument must be enclosed in parentheses.

```
500 PRINT FRE(A$), FRE(L$), FRE("Z")
```

All return the same value.

The string used has no significance; it is a dummy variable. See Chapter 4, CLEAR *n*.

INKEY$

Returns a one-character string determined by an instantaneous keyboard strobe. If no key is pressed during the strobe, a null string (length zero) is returned. This is a very powerful function because it lets you input values while the Computer is executing – without using the ENTER key. The popular video games which let you fire at will, guide a moving dot through a maze, play tennis, etc., may all be simulated using the INKEY$ function (plus a lot of other program logic, of course).

Characters typed to an INKEY$ are not automatically displayed on the screen.

Because of the short duration of the strobe cycle (on the order of microseconds) INKEY$ is invariably placed inside some sort of loop, so that the Keyboard is scanned repeatedly.

Example Program:

```
10  CLS
100 PRINT @ 540,INKEY$ : GOTO 100
```

RUN the program; notice that the screen remains blank until the first time you hit a key. The last key hit remains on the screen until you hit another one. (Whenever you fail to hit a key during a keyboard strobe, a null string, i.e., "nothing", is PRINTed at 540. This "nothing" has no effect on the currently displayed character at 540.)

INKEY$ may be used in sequences of loops to allow the user to build up a longer string.

Example:

```
90  PRINT "ENTER THREE CHARACTERS"
100 A$=INKEY$ : IF A$=""THEN 100 ELSE PRINT A$;
110 B$=INKEY$ : IF B$=""THEN 110 ELSE PRINT B$;
120 C$=INKEY$ : IF C$=""THEN 120 ELSE PRINT C$;
130 D$=A$+B$+C$
```

A three-character string D$ can now be entered via the keyboard without using the ENTER key.

NOTE: The statement `IF A$=""` compares A$ to the null string.

LEFT$ (*string, n*)

Returns the first *n* characters of *string*. The arguments must be enclosed in quotes. *string* may be a string constant or expression, and *n* may be a numeric expression.

Example Program:

```
10   A$="TIMOTHY"
20   B$=LEFT$(A$,3)
30   PRINTB$;"—THAT'S SHORT FOR ";A$

RUN

TIM—THAT'S SHORT FOR TIMOTHY
READY
>_
```

LEN (*string*)

Returns the character length of the specified string. The string variable,expression, or constant must be enclosed in parentheses.

```
10   A$=""
20   B$="TOM"
30   PRINT A$,B$,B$+B$
40   PRINT LEN(A$),LEN(B$),LEN(B$+B$)

RUN

                  TOM               TOMTOM
 0                 3                 6
READY
>_
```

MID$ (*string,p,n*)

Returns a substring of *string* with length *n* and starting at position *p*. The string name, length and starting position must be enclosed in parentheses. *string* may be a string constant or expression, and *n* and *p* may be numeric expressions or constants. For example, MID$(L$,3,1) refers to a one-character string beginning with the 3rd character of L$.

Example Program:

The first three digits of a local phone number are sometimes called the "exchange" of the number. This program looks at a complete phone number (area code, exchange, last four digits) and picks out the exchange of that number.

```
10   INPUT "AREA CODE AND NUMBERS (NO HYPHENS, PLEASE)";PH$
20   EX$=MID$(PH$, 4, 3)
30   PRINT "NUMBER IS IN THE ";EX$;" EXCHANGE."
```

If no argument is specified for the length *n*, the entire string beginning at position *p* is returned.

RIGHT$ (*string,n*)

Returns the last *n* characters of *string*. *string* and *n* must be enclosed in parentheses. *string* may be a string constant or variable, and n may be a numerical constant or variable. If LEN(*string*) is less than or equal to *n*, the entire *string* is returned.

RIGHT$(ST$,4) returns the last 4 characters of ST$.

STR$ (*expression*)

Converts a numeric expression or constant to a string. The numeric expression or constant must be enclosed in parentheses. STR$(A), for example, returns a string equal to the character representation of the value of A. For example, if A=58.5, then STR$(A) equals the string " 58.5". (Note that a leading blank is inserted before "58.5" to allow for the sign of A). While arithmetic operations may be performed on A, only string operations and functions may be performed on the string "58.5".

PRINT STR$(X) prints X without a trailing blank; PRINT X prints X with a trailing blank.

Example Program:

```
10  A=58.5 : B=-58.5
20  PRINT STR$(A)
30  PRINT STR$(B)
40  PRINT STR$(A+B)
50  PRINT STR$(A)+STR$(B)

RUN

 58.5
-58.5
 0
 58.5-58.5
READY
>_
```

Note that the leading blank is filled by the minus sign in STR$(B).

STRING$ (*n, character or number*)

Returns a string composed of *n character*-symbols. For example, STRING$(30,"*") returns "******************************"
STRING$ is useful in creating graphs, tables, etc.

character can also be a number from 0-255; in this case, it will be treated as an ASCII, control, or graphics code.

Example:

STRINGS(64,191) returns a string composed of 64 graphics blocks.

VAL (*string*)

Performs the inverse of the STR$ function: returns the number represented by the characters in a string argument. For example, if A$="12" and B$="34" then VAL(A$+"."+B$) returns the value 12.34. VAL(A$+"E"+B$) returns the value 12E34, that is 12×10^{34}.

VAL operates a little differently on mixed strings – strings whose values consist of a number followed by alphanumeric characters. In such cases, only the leading number is used in determining VAL; the alphanumeric remainder is ignored.

For example: VAL ("100 DOLLARS") returns 100.

This can be a handy short-cut in examining addresses, for example.

Example Program:

```
10  REM "WHAT SIDE OF THE STREET?"
15  REM EVEN=NORTH. ODD=SOUTH
20  INPUT "ADDRESS: NUMBER AND STREET"; AD$
30  C=INT(VAL(AD$)/2)*2
40  IF C=VAL(AD$) PRINT "NORTH SIDE"; GOTO 20
50  PRINT "SOUTH SIDE": GOTO 20
```

RUN the program, entering street addresses like "1015 SEVENTH AVE".

Coding/Decoding Program for Illustration Only

```
5   CLS: PRINT CHR$(23)
10  CLEAR 1000
20  INPUT "ENTER MESSAGE": M$
30  FOR K=1 TO LEN(M$)
40  T$=MID$(M$, K, 1)
60  CD=ASC(T$)+5: IF CD>255 CD=CD-255
70  NU$=NU$ + CHR$(CD)
80  NEXT
90  PRINT "THE CODED MESSAGE IS"
100 PRINT NU$
110 FOR K=1 TO LEN(NU$)
120 T$=MID$(NU$, K, 1)
130 CD=ASC(T$)-5: IF CD<0 CD=CD+255
140 OLD$=OLD$+CHR$(CD)
150 NEXT
160 PRINT "THE DECODED MESSAGE IS"
170 PRINT OLD$
```

RUN the program.

Lines 30-80 and 110-150 demonstrate how you can "peel off" the characters of a string for examination. Lines 60 and 130 demonstrate manipulation of ASCII codes.

Instring Subroutine

Using the intrinsic string functions MID$ and LEN, it's easy to create a very handy string-handling subroutine, INSTRING. This function takes two string arguments and tests to see whether one is contained in the other. When you are searching for a particular word, phrase or piece of data in a larger body of text or data, INSTRING can be very powerful. Here's the subroutine:

```
999    END ' THIS IS A PROTECTIVE END-BLOCK
1000   FOR I=1TO LEN(X$)-LEN(Y$)+1
1010   IF Y$=MID$(X$,I,LEN(Y$)) RETURN
1020   NEXT : I=0 : RETURN
```

To use the subroutine, first assign the value of the larger string (the "search area") to X$, and the value of the desired substring to Y$. Then call the subroutine with GOSUB. The subroutine will return a value of I which tells you the starting position of Y$ in the larger string X$; or if Y$ is not a substring of X$, I is returned with a value of zero.

Here's a sample program using the INSTRING subroutine. (Type in the above lines 999-1020 plus the following.)

```
5    CLEAR 1000 : CLS
10   INPUT "ENTER THE LONGER STRING"; X$
20   INPUT "NOW ENTER THE SHORTER STRING"; Y$
30   GOSUB 1000
40   IF I=0 THEN 70
50   PRINT Y$;" IS A SUBSTRING OF ";X$
55   PRINT "STARTING POSITION:";I,
60   PRINT "ENDING POSITION:";I+LEN(Y$)-1
65   PRINT : PRINT : GOTO 10
70   PRINT Y$;" IS NOT CONTAINED IN ";X$
80   GOTO 10
```

RUN the program, entering the string to be searched and then the desired substring.

6/Arrays

An array is simply an ordered list of values. In LEVEL II these values may be either numbers or strings, depending on how the array is defined or typed. Arrays provide a fast and organized way of handling large amounts of data. To illustrate the power of arrays, this chapter traces the development of an array to store checkbook data: check numbers, dates written, and amounts for each check.

In addition, several matrix manipulation subroutines are listed at the end of this chapter. These sequences will let you add, multiply, transpose, and perform other operations on arrays.

Note: Throughout this chapter, zero-subscripted elements are generally ignored for the sake of simplicity. But you should remember they are available and should be used for the most efficient use of memory. For example, after DIM A(4), array A contains 5 elements: A(0), A(1), A(2), A(3), A(4).

For background information on arrays, see Chapter 4, DIM, and Chapter 1, "Arrays".

A Check-Book Array

Consider the following table of checkbook information:

Check #	Date Written	Amount
025	1-1-78	10.00
026	1-5-78	39.95
027	1-7-78	23.50
028	1-7-78	149.50
029	1-10-78	4.90
030	1-15-78	12.49

Note that every item in the table may be specified simply by reference to two numbers: the row number and the column number. For example, (row 3, column 3) refers to the amount 23.50. Thus the number pair (3,3) may be called the "subscript address" of the value 23.50.

Let's set up an array, CK, to correspond to the checkbook information table. Since the table contains 6 rows and 3 columns, array CK will need two dimensions: one for row numbers, and one for column numbers. We can picture the array like this:

A(1,1)=025	A(1,2)=1.0178	A(1,3)=10.00
.	.	.
.	.	.
.	.	.
.	.	.
A(6,1)=030	A(6,2)=1.1578	A(6,3)=12.49

Notice that the date information is recorded in the form *mm ddyy*, where *mm*=month number, *dd*=day of month, and *yy* = last two digits of year. Since CK is a numeric array, we can't store the data with alpha-numeric characters such as dashes.

Suppose we assign the appropriate values to the array elements. Unless we have used a DIM statement, the Computer will assume that our array requires a depth of 10 for each dimension. That is, the Computer will set aside memory locations to hold CK(7,1), CK(7,2), ..., CK(9,1), CK(9,2) and CK(9,3). In this case, we don't want to set aside this much space, so we use the DIM statement at the beginning of our program:

```
10   DIM CK(6,3)   'SET UP A 6 BY 3 ARRAY (EXCL. ZERO SUBSCRIPTS)
```

Now let's add program steps to read the values into the array CK:

```
20   FOR ROW=1 TO 6
30        FOR COL=1 TO 3
40             READ CK(ROW,COL)
50   NEXT COL, ROW
90   DATA 025, 1.0178, 10.00
91   DATA 026, 1.0578, 39.95
92   DATA 027, 1.0778, 23.50
93   DATA 028, 1.0778, 149.50
94   DATA 029, 1.1078, 4.90
95   DATA 030, 1.1578, 12.49
```

Now that our array is set up, we can begin taking advantage of its built-in structure. For example, suppose we want to add up all the checks written. Add the following lines to the program:

```
100 FOR ROW=1 TO 6
110      SUM=SUM+CK(ROW,3)
120 NEXT
130 PRINT"TOTAL OF CHECKS WRITTEN";
140 PRINT USING"$$###.##";SUM
```

Now let's add program steps to print out all checks that were written on a given day.

```
200 PRINT "SEEKING CHECKS WRITTEN ON WHAT DATE (MM.DDYY)";
210 INPUT DT
230 PRINT :PRINT"ANY CHECKS WRITTEN ARE LISTED BELOW:"
240 PRINT"CHECK #","AMOUNT" : PRINT
250 FOR ROW=1 TO 6
260       IF CK(ROW,2)=DT PRINT CK(ROW,1), CK(ROW,3)
270 NEXT
```

It's easy to generalize our program to handle checkbook information for all 12 months and for years other than 1978.

All we do is increase the size (or "depth") of each dimension as needed. Let's assume our checkbook includes check numbers 001 through 300, and we want to store the entire checkbook record. Just make these changes:

```
10  DIM CK(300,3) 'SET UP A 300 BY 3 ARRAY
20  FOR ROW=1 TO 300
```

and add DATA lines for check numbers 001 through 300. You'd probably want to pack more data onto each DATA line than we did in the above DATA lines.

And you'd change all the ROW counter final values:

```
100 FOR ROW=1 TO 300
.
.
250 FOR ROW=1 TO 300
```

Other Types of Arrays

Remember, in LEVEL II the number of dimensions an array can have (and the size or depth of the array), is limited only by the amount of memory available. Also remember that string arrays can be used. For example, C$(X) would automatically be interpreted as a string array. And if you use DEFSTR A at the beginning of your program, any array whose name begins with A would also be a string array. One obvious application for a string array would be to store text material for access by a string manipulation program.

```
10  CLEAR 1200
20  DIM TXT$(10)
```

would set up a string array capable of storing 10 lines of text. 1200 bytes were CLEARed to allow for 10 sixty-character lines, plus 600 extra bytes for string manipulation with other string variables.

Array/Matrix Manipulation Subroutines

To use this subroutine, your main program must supply values for N1 (rows) and N2 (columns).

```
30100 REM MATRIX INPUT SUBROUTINE (2 DIMENSION)
30110 FOR I=1 TO N1
30120 PRINT "INPUT ROW";I
30130    FOR J=1 TO N2
30140    INPUT A(I,J)
30160 NEXT J,I
30170 RETURN
```

To use this subroutine, your main program must supply values for N1 (dim#1), N2 (dim#2) and N3(dim#3).

```
30200 REM MATRIX READ SUBROUTINE (3 DIMENSION)
30205 REM REQUIRES DATA STMTS.
30210 FOR K=1 TO N3
30220    FOR I=1 TO N1
30230       FOR J=1 TO N2
30240       READ A(I,J,K)
30270 NEXT J,I,K
30280 RETURN
```

Main program supplies values for N1, N2, N3, etc.

```
30300 REM MATRIX ZERO SUBROUTINE (3 DIMENSION)
30310 FOR K=1 TO N3
30320    FOR J=1 TO N2
30330       FOR I=1 TO N1
30340       A(I,J,K)=0
30370 NEXT I,J,K
30380 RETURN
```

Main program supplies values for N1, N2, N3.

```
30400 REM MATRIX PRINT SUBROUTINE (3 DIMENSION)
30410 FOR K=1 TO N3
30420    FOR I=1 TO N1
30430       FOR J=1 TO N2
30440       PRINT A(I,J,K),
30450       NEXT J:PRINT
30460    NEXT I:PRINT
30470 NEXT K:PRINT
30480 RETURN
```

Main program supplies values for N1, N2, N3.

```
30500 REM MATRIX INPUT SUBROUTINE (3 DIMENSION)
30510 FOR K=1 TO N3
30520 PRINT "PAGE";K
30530   FOR I=1 TO N1
30540   PRINT "INPUT ROW";I
30550     FOR J=1 TO N2
30560     INPUT A(I,J,K)
30570     NEXT J
30580   NEXT I
30590 PRINT:NEXT K
30595 RETURN
```

Multiplication by a Single Variable: Scalar Multiplication (3 Dimensional)

```
30600 FOR K=1 TO N3:'N3=3RD DIMENSION
30610   FOR J=1 TO N2:'N2=2ND DIMENSION (ROWS)
30620     FOR I=1 TO N1:'N1=1ST DIMENSION (ROWS)
30630     B(I,J,K)=A(I,J,K)*X
30640     NEXT I
30650   NEXT J
30660 NEXT K
30670 RETURN
```

Multiplies each element in MATRIX A by X and constructs matrix B

Transposition of a Matrix (2 Dimensional)

```
30700 FOR I = 1 TO N1
30710   FOR J=1 TO N2
30720   B(J,I)=A(I,J)
30730   NEXT J
30740 NEXT I
30750 RETURN
```

Transposes matrix A into matrix B

Matrix Addition (3 Dimensional)

```
30800 FOR K=1 TO N3
30810   FOR J = 1 TO N2
30820     FOR I=1 TO N1
30830     C(I,J,K)=A(I,J,K)+B(I,J,K)
30840     NEXT I
30850   NEXT J
30850 NEXT K
30870 RETURN
```

Array Element-wise Multiplication (3 Dimensional)

```
30900 FOR K= 1 TO N3
30910    FOR J=1 TO N2
30920       FOR I=1 TO N1
30930       C[I,J,K]=A[I,J,K]*B[I,J,K]
30940       NEXT I
30950    NEXT J
30960 NEXT K
```

Multiplies each element in A times its corresponding element in B.

Matrix Multiplication (2 Dimensional)

```
40000 FOR I=1 TO N1
40010    FOR J=1 TO N2
40020    C[I,J]=0
40030       FOR K=1 TO N3
40040       C[I,J]=C[I,J]+A[I,K]*B[K,I]
40050       NEXT K
40060    NEXT J
40070 NEXT I
```

A must be an N1 by N3 matrix; B must be an N3 by N2 matrix. The resultant matrix C will be an N1 by N2 matrix. A, B, and C must be dimensioned accordingly.

7/Arithmetic Functions

LEVEL II BASIC offers a wide variety of intrinsic ("built-in") functions for performing arithmetic and special operations. The special-operation functions are described in the next chapter.

All the common math functions described in this chapter return single-precision values accurate to six decimal places. ABS, FIX and INT return values whose precision depends on the precision of the argument. The conversion functions (CINT, CDBL, etc.) return values whose precision depends on the particular function. For all the functions, the argument must be enclosed in parentheses. The argument may be either a numeric variable, expression or constant.

Functions described in this chapter:

ABS	COS	INT	SGN
ATN	CSNG	LOG	SIN
CDBL	EXP	RANDOM	SQR
CINT	FIX	RND	TAN

ABS (*x*)

Returns the absolute value of the argument. ABS(X)=X for X greater than or equal to zero, and ABS(X)=−X for X less than zero.

```
100 IF ABS(X)<1E-6 PRINT "TOO SMALL"
```

ATN (*x*)

Returns the arctangent (in radians) of the argument; that is, ATN(X) returns "the angle whose tangent is X". To get arctangent in degrees, multiply ATN(X) by 57.29578.

```
100 Y=ATN(B/C)
```

CDBL (*x*)

Returns a double-precision representation of the argument. The value returned will contain 17 digits, but only the digits contained in the argument will be significant.

CDBL may be useful when you want to force an operation to be done in double-precision, even though the operands are single precision or even integers. For example CDBL (I%)/J% will return a fraction with 17 digits of precision.

```
100 FOR I%=1 TO 25 : PRINT 1/CDBL(I%), : NEXT
```

CINT (*x*)

Returns the largest integer not greater than the argument. For example, CINT (1.5) returns 1; CINT(-1.5) returns -2. For the CINT function, the argument must be in the range -32768 to + 32767.

CINT might be used to speed up an operation involving single or double-precision operands without losing the precision of the operands (assuming you're only interested in an integer result).

```
100 K%=CINT(X#)+CINT(Y#)
```

COS (*x*)

Returns the cosine of the argument (argument must be in radians). To obtain the cosine of X when X is in degrees, use COS(X*.0174533).

```
100 Y=COS(X+3.3)
```

CSNG (*x*)

Returns a single-precision representation of the argument. When the argument is a double-precision value, it is returned as six significant digits with "4/5 rounding" in the least significant digit. So CSNG(.6666666666666667) is returned as .666667; CSNG(.3333333333333333) is returned as .333333.

```
100 PRINT CSNG(A#+B#)
```

EXP (*x*)

Returns the "natural exponential" of X, that is, e^X. This is the inverse of the LOG function, so X=EXP(LOG(X)).

```
100 PRINT EXP(-X)
```

FIX (*x*)

Returns a truncated representation of the argument. All digits to the right of the decimal point are simply chopped off, so the resultant value is an integer. For non-negative X, FIX(X)=INT(X). For negative values of X, FIX(X)=INT(X)+1. For example, FIX(2.2) returns 2, and FIX(-2.2) returns -2.

```
100 Y=ABS(A-FIX(A))
```

This statement gives Y the value of the fractional portion of A.

Page Was Blank When Provided

INT(x)

Returns an integer representation of the argument, using the largest integer that is not greater than the argument. Argument is not limited to the range -32768 to +32767. INT(2.5) returns 2; INT(-2.5) returns -3; and INT(1000101.23) returns 100101.

```
100 Z=INT(A*100+.5)/100
```

Gives Z the value of A rounded to two decimal places (for non-negative A).

LOG(x)

Returns the natural logarithm of the argument, that is, $\log_e$(*argument*). This is the inverse of the EXP function, so X=LOG(EXP (X)). To find the logarithm of a number to another base *b*, use the formula $\log_b(X) = \log_e(X)/\log_e(b)$. For example, LOG(32767)/LOG(2) returns the logarithm to base 2 of 32767.

```
100 PRINT LOG(3.3*X)
```

RANDOM

RANDOM is actually a complete statement rather than a function. It reseeds the random number generator. If a program uses the RND function, you may want to put RANDOM at the beginning of the program. This will ensure that you get an unpredictable sequence of pseudo-random numbers each time you turn on the Computer, load the program, and run it.

```
10  RANDOM
20  C=RND(6)+RND(6)
.
.
.
80  GOTO 20  'RANDOM NEEDS TO EXECUTE JUST ONCE
```

RND(x)

Generates a pseudo-random number using the current pseudo-random "seed number" (generated internally and not accessible to user). RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND(0) returns a single-precision value between 0 and 1. RND(*integer*) returns an integer between 1 and *integer* inclusive (*integer* must be positive and less than 32768). For example, RND(55) returns a pseudo-random integer greater than zero and less than 56. RND(55.5) returns a number in the same range, because RND uses the INTeger value of the argument.

```
100 X=RND(2) : ON X GOTO 200,300
```

SGN(x)

The "sign" function : returns –1 for X negative, 0 for X zero, and +1 for X positive.

```
100  ON SGN(X)+2 GOTO 200,300,400
```

SIN(x)

Returns the sine of the argument (argument must be in radians). To obtain the sine of X when X is in degrees, use SIN(X*.0174533).

```
100  PRINT SIN(A*B-B)
```

SQR(x)

Returns the square root of the argument. SQR(X) is the same as X↑(1/2), only faster.

```
100  Y=SQR(X↑2-H↑2)
```

TAN(x)

Returns the tangent of the argument (argument must be in radians). To obtain the tangent of X when X is in degrees, use TAN(X*.0174533).

```
100  Z=TAN(2*A)
```

NOTE: A great many other functions may be created using the above functions. See Appendix F, "Derived Functions".

8/Special Features

LEVEL II BASIC offers some unusual functions and operations that deserve special highlighting. Some may seem highly specialized; as you learn more about programming and begin to experiment with machine-language routines, they will take on more significance. Other functions in the chapter are of obvious benefit and will be used often (for example, the graphics functions). And then there are a couple of features, INP and OUT, that will be used primarily with the TRS-80 Expansion Interface.

Functions, statements and operators described in this chapter:

Graphics:	Error-Routine Functions:	Other Functions and Statements:
SET	ERL	INP
RESET	ERR	MEM
CLS		PEEK
POINT	**Logical Operators:**	POKE
		POS
	AND	OUT
	OR	USR
	NOT	VARPTR

SET(*x*,*y*)

Turns on the graphics block at the location specified by the coordinates *x* and *y*. For graphics purposes, the Display is divided up into a 128 (horizontal) by 48 (vertical) grid. The *x*-coordinates are numbered from left to right, 0 to 127. The *y*-coordinates are numbered from top to bottom, 0 to 47. Therefore the point at (0,0) is in the extreme upper left of the Display, while the point at (127,47) is in the extreme lower right corner. See the Video Display Worksheet in Appendix E.

The arguments *x* and *y* may be numeric constants, variables or expressions. They need not be integer values, because SET(*x*,*y*) uses the INTeger portion of *x* and *y*. SET (*x*,*y*) is valid for:

$0 <= x < 128$
$0 <= y < 48$

Examples:

```
100 SET(RND(128)-1,RND(48)-1)
```

Lights up a random point on the Display.

```
100 INPUT X,Y: SET(X,Y)
```

RUN to see where the blocks are.

RESET(x,y)

Turns off a graphics block at the location specified by the coordinates x and y. This function has the same limits and parameters as SET(x,y).

```
200 RESET (X,3)
```

CLS

"Clear-Screen" – turns off all the graphics blocks on the Display and moves the cursor to the upper left corner. This wipes out alphanumeric characters as well as graphics blocks. CLS is very useful whenever you want to present an attractive Display output.

```
5   CLS
10  SET(RND(128)-1,RND(48)-1)
20  GOTO 10
```

POINT(x,y)

Tests whether the specified graphics block is "on" or "off". If the block is "on" (that is, if it has been SET), then POINT returns a binary True (-1 in LEVEL II BASIC). If the block is "off", POINT returns a binary False (0 in LEVEL II BASIC). Typically, the POINT test is put inside an IF-THEN statement.

```
100 SET(50,28) : IF POINT(50,28) THEN PRINT "ON" ELSE PRINT "OFF"
```

This line will always print the message, "ON", because POINT(50,28) will return a binary True, so that execution proceeds to the THEN clause. If the test failed, POINT would return a binary False, causing execution to jump to the ELSE statement.

ERL

Returns the line number in which an error has occurred. This function is primarily used inside an error-handling routine accessed by an ON ERROR GOTO statement. If no error has occurred when ERL is called, line number 0 is returned. However, if an error has occurred since power-up, ERL returns the line number in which the error occurred. If error occurred in direct mode, 65535 is returned (largest number representable in two bytes).

Example Program using ERL

```
5       ON ERROR GOTO 1000
10      CLEAR 10
20      INPUT"ENTER YOUR MESSAGE";M$
30      INPUT"NOW ENTER A NUMBER";N : N=1/N
40      REM  REST OF PROGRAM BEGINS HERE
.
.
999 END
1000    IF ERL=20 THEN 1010 ELSE IF ERL=30 THEN 1020
1005    ON ERROR GOTO 0
1010    PRINT "TRY AGAIN-KEEP MESSAGE UNDER 11 CHARACTERS"
1015    RESUME 20
1020    PRINT"FORGOT TO MENTION: NUMBER MUST NOT BE ZERO"
1025    RESUME 30
```

RUN the program. Try entering a long message; try entering zero when the program asks for a number. Note that ERL is used in line 1000 to determine where the error occurred so that appropriate action may be taken.

ERR/2+1

Similar to ERL, except ERR returns a value related to the code of the error rather than the line in which the error occurred. Commonly used inside an error handling routine accessed by an ON ERROR GOTO statement. See Appendix B, "Error Codes."

ERR/2+1 = true error code
(true error code −1)*2=ERR

Example Program

```
10      ON ERROR GOTO 1000
20      DIM A(15) : I=1
30      READ A(I)
40      I=I+1 : GOTO 30
50      REM  REST OF PROGRAM
.
.
100     DATA 2,3,5,7,1,13
999     END
1000    IF ERR/2+1=4 RESUME 50
1010    ON ERROR GOTO 0
```

Note line 1000: 4 is the error code for Out of Data.

INP(*port*)

Returns a byte-value from the specified port. The TRS-80 Expansion Interface is required to use INP effectively (with user-supplied peripheral hardware). There are 256 ports, numbered 0-255. For example

```
100 PRINT INP(50)
```

inputs a byte from port 50 and prints the decimal value of the byte.

MEM

Returns the number of unused and unprotected bytes in memory. This function may be used in the Command Mode to see how much space a resident program takes up; or it may be used inside the program to avert OM (Out of Memory) errors by allocating less string space, DIMensioning smaller array sizes, etc. MEM requires no argument.

Example:

```
100 IF MEM < 80 THEN 900
110 DIM A(15)
.
.
.
```

Enter the command PRINT MEM (In Command Mode) to find out the amount of memory not being used to store programs, variables, strings, stack, or reserved for object-files.

OUT *port, value*

Ouputs a byte value to the specified port. OUT is not a function but a statement complete in itself. It requires two arguments separated by a comma (no parenthesis): the port destination and the byte value to be sent.

Example:

```
OUT 250,10
```

sends the value "10" to port 250. Both arguments are limited to the range 0-255.

OUT, like INP, becomes useful when you add the TRS-80 Expansion Interface. See INP.

PEEK(*address*)

Returns the value stored at the specified byte address (in decimal form). To use this function, you'll need to refer to two sections of the Appendix: the Memory Map (so you'll know where to PEEK) and the Table of Function, ASCII and Graphics Codes (so you'll know what the values represent).

If you're using PEEK to examine object files, you'll also need a microprocessor instruction set manual (one is included with the TRS-80 Editor/Assembler Instruction Manual).

PEEK is valuable for linking machine language routines with LEVEL II BASIC programs. The machine language routine can store information in a certain memory location, and PEEK may be used inside your BASIC program to retrieve the information. For example,

```
A = PEEK (17999)
```

returns the value stored at location 17999 and assigns that value to the variable A.

Peek may also be used to retrieve information stored with a POKE statement. Using PEEK and POKE allows you to set up very compact, byte-oriented storage systems. Refer to the Memory Map in the Appendix to determine the appropriate locations for this type of storage. See POKE, USR.

POKE *address, value*

Loads a value into a specified memory location. POKE is not a function but a statement complete in itself. It requires two arguments: a byte address (in decimal form) and a value. The value must be between 0 and 255 inclusive. Refer to the Memory Map in the Appendix to see which addresses you'd like to POKE.

To POKE (or PEEK) an address above 32767, use the following formula. -1*(desired address - 32767) = POKE or PEEK address.

POKE is useful for LEVEL II graphics. Look at the Video Display Worksheet in the Appendix. In each of the 1024 PRINT locations there are 6 subdivisions. If we call each PRINT position a byte, then the smaller boxes are bits. We know that there are 8 bits per byte; so what happened to the other 2? One is used to identify the byte as a graphics or ASCII code. The other bit is not used. The remaining 6 bits contain either an ASCII, graphics or control code.

We can use POKE to turn on the entire PRINT position (6 bits) at one time. When we use SET, only 1 bit is turned on. Therefore POKE is about 6 times faster than SET. The following program demonstrates this speed.

```
10  CLS
20  FOR X=15360 TO 16383
30  POKE X, 191
40  NEXT
50  GOTO 50
```

RUN the program to see how fast the screen is "painted" white. (191 is the code for "all bits on". 15360 to 16383 are the Video Display memory addresses.)

Since POKE can be used to store information anywhere in memory, it is very important when we do our graphics to stay in the range for display locations. If we POKE outside this range, we may store the byte in a critical place. We could be POKEing into our program, or even in worse places like the stack. Indiscriminate POKEing can be disastrous. You might have to reset or power off and start over again. Unless you know where you are POKEing – don't.

See PEEK, USR, SET, and Chapter 4, CHR$ for background material.

POS(*x*)

Returns a number from 0 to 63 indicating the current cursor position on the Display. Requires a "dummy argument" (any numeric expression).

```
100 PRINT TAB(40) POS(0)
```

prints 40 at position 40. (Note that a blank is inserted before the "4" to accommodate the sign; therefore the "4" is actually at position 41.) The "0" in "POS(0)" is the dummy argument.

```
100 PRINT "THESE" TAB(POS(0)+5) "WORDS" TAB(POS(0)+5) "ARE";
110 PRINT TAB(POS(0)+5) "EVENLY" TAB(POS(0)+5) "SPACED"

RUN

THESE     WORDS     ARE     EVENLY     SPACED
READY
>_
```

USR (*x*)

Calls a machine language subroutine and passes the argument to the subroutine (you may not need it, in which case it is a dummy argument). Such a subroutine could be loaded from tape, or created by POKEing microprocessor instructions into the appropriate memory locations. To use the USR function, you should be familiar with the machine-language programming (as explained in the TRS-80 Editor/Assembler Instruction Manual or any Z-80 Programming Manual). Playing around with the USR function can be disastrous to any programs you may have resident in the TRS-80; so do some studying before you attempt to use it.

There is only one allowable USR call in LEVEL II BASIC. In LEVEL II DISC BASIC, there will be up to 10: USR0 through USR9.

Example:

```
100 X=USR(N)
```

would cause the Computer to branch to the routine beginning at the location POKEd into the USR(N) addresses 16526-16527. *n* is also stored at 2687 as a 2-byte integer. Upon return from the routine, the variable X would be given the value passed back from the routine. If no value is passed, X is assigned the value of the argument N.

N must be an integer between -32768 and 32767.

To create a machine language subroutine for access by USR, you must protect an area in high memory. (See Appendix D, "Memory Map"). First determine how many bytes your routine will require Then subtract that number from your Computer's highest Memory address (depending on whether your TRS-80 has 4K or 16K bytes of memory). The resultant number will be the address where your protected memory should begin. Turn on the TRS-80, and answer the MEMORY SIZE question by entering the address where protected memory should begin. Addresses above that number will now be reserved for machine language data and routines.

Load the machine language routine, using POKE or via the cassette interface using the SYSTEM command (see Chapter 2, SYSTEM). Then, at the point where you want your BASIC program to branch to the machine language routine, insert a statement which calls USR(0). For example,

```
50  PRINT USR(N)
```

or

```
50  A = USR(I%) + B
```

To pass the argument to the subroutine, the subroutine should immediately execute a CALL 0A7F(hex) i,e., call 2687(dec).

There are two ways to return to your BASIC program from the machine-language subroutine:

1) If you don't wish to pass any values from the subroutine back to the BASIC program, a machine-language RET instruction can be used.

2) To return a value, load the value into the HL register pair as a two-byte signed integer and execute a JUMP to location 0A9A (HEX) [2714 (DEC)]. HL will be returned as a signed 2-byte integer.

The last thing you need to do is tell your BASIC program what address to branch to in the machine language routine. This two-byte address must be POKEd into memory locations 16526 and 16527. POKE the least significant byte into the lower (16526) memory location.

For example, if your routine begins at 32000: in hexadecimal this is 7D00. Therefore we POKE 00 (HEX) into 16526, and 7D (HEX) into 16527. Since POKE requires arguments in decimal form, we use:

```
POKE 16526,0 : POKE 16527,208
```

(208 decimal = 7D hex).

After you have executed the above line, when you use the USR(0) function, the Computer will branch to the instruction stored at 32000.

Note: locations 16526-16527 contain the address of the Illegal Function Call routine unless modified by POKE.

USR routines are automatically allocated up to 8 stack levels or 16 bytes (a high and low memory byte for each stack level). If you need more stack space, you can save the BASIC stack pointer and set up your own stack. However, this gets complicated; be sure you know what you're doing. See Chapter 2, SYSTEM, and this chapter, PEEK, POKE.

VARPTR (*variable name*)

Returns an address-value which will help you locate where the variable name and its value are stored in memory. If the variable you specify has not been assigned a value, an FC error will occur when this function is called.

If VARPTR(*integer variable*) returns address K:
Address K contains the least significant byte (LSB) of 2-byte integer K (two's complement form).
Address K+1 contains the most significant byte (MSB) of integer K.

If VARPTR(*single precision variable*) returns address K:
(K)* = LSB of value
(K+1) = Next most sig. byte (Next MSB)
(K+2) = MSB
(K+3) = exponent of value

If VARPTR(*double precision variable*) returns K:
(K) = LSB of value
(K+1) = Next MSB
(K+...) = Next MSB
(K+6) = MSB
(K+7) = exponent of value

IF VARPTR(*string variable*) returns K:
(K) = length of string
(K+1) = LSB of string value starting address
(K+2) = MSB of string value starting address

For single and double precision values, the number is stored in normalized exponential form, so that a decimal is assumed before the MSB. 128 is added to the exponent. Furthermore, the high bit of MSB is used as a sign bit. See examples below.

Examples:

A! = 4 will be stored as follows:
4 = 10 Binary, normalized as .1E2
So exponent of A is 128+2 = 130
MSB of A is 10000000;
however, the high bit is changed to zero since the value is positive.
So A! is stored as

Exponent	MSB	Next MSB	LSB
130	0	0	0

A! = -.5 will be stored as

Exponent	MSB	Next MSB	LSB
128	128	0	0

* (K) signifies "contents of address K"

A! = 7 will be stored as

Exponent	MSB	Next MSB	LSB
131	96	0	0

A! = -7 :

Exponent	MSB	Next MSB	LSB
131	224	0	0

Zero is simply stored as a zero-exponent. The other bytes are insignificant.

Logical Operators

In Chapter 1 we described how AND, OR and NOT can be used with relational expressions. For example,

```
100 IF A=C AND NOT (B>40) THEN 60 ELSE 50
```

AND, OR and NOT can also be used for bit manipulation, bitwise comparisons, and Boolean operations. In this section, we will explain how such operations can be implemented using LEVEL II BASIC. However, we will not try to explain Boolean algebra, decimal-to-binary conversions, binary arithmetic, etc. If you need to learn about these subjects, Radio Shack's **Understanding Digital Computers** (Catalog Number 62-2027) would be a good place to start.

AND, OR and NOT convert their arguments to sixteen-bit, signed two's-complement integers in the range -32768 to +32767. They then perform the specified logical operation on them and return a result within the same range. If the arguments are not in this range, an "FC" error results.

The operations are performed in bitwise fashion; this means that each bit of the result is obtained by examining the bit in the same position for each argument.

The following truth tables show the logical relationship between bits:

OPERATOR	ARGUMENT 1	ARGUMENT 2	RESULT
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0

OPERATOR	ARGUMENT 1	ARGUMENT 2	RESULT
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0

OPERATOR	ARGUMENT	RESULT
NOT	1	0
	0	1

EXAMPLES:

(In all of the examples below, leading zeroes on binary numbers are not shown.)

63 AND 16=16	Since 63 equals binary 111111 and 16 equals binary 10000, the result of the AND is binary 10000 or 16.
15 AND 14=14	15 equals binary 1111 and 14 equals binary 1110, so 15 and 14 equals binary 1110 or 14.
-1 AND 8=8	-1 equals binary 1111111111111111 and 8 equals binary 1000, so the result is binary 1000 or 8 decimal.
4 AND 2=0	4 equals binary 100 and 2 equals binary 10, so the result is binary 0 because none of the bits in either argument match to give a 1 bit in the result.
4 OR 2=6	Binary 100 OR'd with binary 10 equals binary 110, or 6 decimal.
10 OR 10=10	Binary 1010 OR'd with binary 1010 equals binary 1010, or 10 decimal.
-1 OR -2=-1	Binary 1111111111111111 (-1) OR'd with binary 1111111111111110 (-2) equals binary 1111111111111111, or -1.
NOT 0=-1	The bit complement of binary 0 to 16 places is sixteen ones (1111111111111111) or -1. Also NOT -1=0.
NOT X	NOT X is equal to -(X+1). This is because to form the sixteen bit two's complement of the number, you take the bit (one's) complement and add one.
NOT 1=-2	The sixteen bit complement of 1 is 1111111111111110, which is equal to -(1+1) or -2.

A typical use of the bitwise operators is to test bits set in the TRS-80's inport ports which reflect the state of some external device. This requires the TRS-80 Expansion Interface.

Bit position 7 is the most significant bit of a byte, while position 0 is the least significant.

For instance, suppose bit 1 of I/O port 5 is 0 when the door to Room X is closed, and 1 if the door is open. The following program will print "Intruder Alert" if the door is opened:

```
10 IF INP(5) AND 2 THEN PRINT "INTRUDER ALERT":GOTO 100
20 GOTO 10
```

See Chapter 1, "Logical Operators".

9/Editing

LEVEL I users undoubtedly spent lots of time retyping long program lines, all because of a typo, or maybe just to make a minor change. Once a line had been entered, there was no way to alter the line – without starting all over and retyping it.

LEVEL II's editing features eliminate much of this extra work. In fact, it's so easy to alter program lines, you'll probably be able to do much more experimenting with multi-statement lines, complex expressions, etc.

Commands, subcommands, and special function keys described in this chapter:

EDIT	L	*n*D
ENTER	X	*n*C
*n*Space-Bar	I	*n*Sc
n →	A	*n*Kc
SHIFT ↑	E	
	Q	
	H	

EDIT *line number*

This command puts you in the Edit Mode. You must specify which line you wish to edit, in one of two ways:

EDIT *line-number* ENTER	Lets you edit the specified line. If line number is not in use, an FC error occurs
or	
EDIT.	Lets you edit the current program line – last line entered or altered or in which an error has occurred.

For example, type in and ENTER the following line:

```
100 FOR I = 1 TO 10 STEP .5 : PRINT I, I↑2, I↑3 : NEXT
```

This line will be used in exercising all the Edit subcommands described below.

Now type EDIT. and hit ENTER . The Computer will display:

```
100 _
```

You are now in the Edit Mode and may begin editing line 100.

ENTER key

Hitting **ENTER** while in the Edit Mode causes the Computer to record all the changes you've made (if any) in the current line, and returns you to the Command Mode.

*n*Space-bar

In the Edit Mode, hitting the Space-bar moves the cursor over one space to the right and displays any character stored in the preceding position. For example, using line 100 entered above, put the Computer in the Edit Mode so the Display shows:

```
100 _
```

Now hit the Space-Bar. The cursor will move over one space, and the first character of the program line will be displayed. If this character was a blank, then a blank will be displayed. Hit the Space-Bar until you reach the first non-blank character:

```
100 F_
```

is displayed. To move over more than one space at a time, hit the desired number of spaces first, and then hit the space-bar. For example, enter 5 and hit Space-bar, and the display will show something like this (may vary depending on how many blanks you inserted in the line):

```
100 FOR I=_
```

Now type 8 and hit the Space-bar. The cursor will move over 8 spaces to the right, and 8 more characters will be displayed.

n ← (Backspace)

Moves the cursor to the left by *n* spaces. If no number *n* is specified, the cursor moves back one space. When the cursor moves to the left, all characters in its "path" are erased from the display, but they are **not deleted from the program line.** For example, assuming you've used *n*Space-Bar so that the Display shows:

```
100 FOR I=1 TO 10 _
```

type 8 and hit the ← key. The Display will show something like this.

```
100 FOR I=_          (will vary depending on number of blanks in
                      your line 100 )
```

SHIFT ↓

Hitting SHIFT and ↓ keys together effects an escape from any of the Insert subcommands listed below: X, I and H. After escaping from an Insert subcommand, you'll still be in the Edit Mode, and the cursor will remain in its current position. (Hitting ENTER is another way to exit these Insert subcommands).

L (List Line)

When the Computer is in the Edit Mode, and is not currently executing one of the subcommands below, hitting L causes the remainder of the program line to be displayed. The cursor drops down to the next line of the Display, reprints the current line number, and moves to the first position of the line. For example, when the Display shows

```
100 _
```

hit L (without hitting ENTER key) and line 100 will be displayed:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I↓2, I↓3 : NEXT
100 _
```

This lets you look at the line in its current form while you're doing the editing.

X (End of Line and Insert)

Causes the rest of the current line to be displayed, moves cursor to end of line, and puts Computer in the Insert subcommand mode so you can add material to the end of the line. For example, using line 100, when the Display shows

```
100 _
```

hit X (without hitting ENTER) and the entire line will be displayed; notice that the cursor now follows the last character on the line:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I↓2, I↓3 : NEXT _
```

We can now add another statement to the line, or delete material from the line by using the ← key. For example, type `: PRINT"DONE"` at the end of the line. Now hit ENTER .
If you now type LIST 100, the Display should show something like this:

```
100 FOR I=1 TO 10 STEP .5 : PRINT I, I↓2, I↓3 : NEXT : PRINT"DONE"
```

I (Insert)

Allows you to insert material beginning at the current cursor position on the line. (Hitting → will actually delete material from the line in this mode.) For example, type and ENTER the EDIT 100 command, then use the Space Bar to move over to the decimal point in line 100. The Display will show:

```
100 FOR I=1 TO 10 STEP _
```

Suppose you want to change the increment from .5to .25. Hit the I key (don't hit ENTER) and the Computer will now let you insert material at the current position. Now hit 2 so the Display shows:

```
100 FOR I=1 TO 10 STEP .2_
```

You've made the necessary change, so hit SHIFT ↓ to escape from the Insert Subcommand. Now hit L key to display remainder of line and move cursor back to the beginning of the line:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I↓2, I↓3 : NEXT : PRINT "DONE"
100 _
```

You can also exit the Insert subcommand and save all changes by hitting ENTER . This will return you to Command mode.

A (Cancel and Restart)

Moves the cursor back to the beginning of the program line and cancels editing changes already made. For example, if you have added, deleted, or changed something in a line, and you wish to go back to the beginning of the line and cancel the changes already made: first hit SHIFT ↓ (to escape from any subcommand you may be executing); then hit A. (The cursor will drop down to the next line, display the line number and move to the first program character.

E (Save Changes and Exit)

Causes Computer to end editing and save all changes made. You must be in Edit Mode, not executing any subcommand, when you hit E to end editing.

Q (Cancel and Exit)

Tells Computer to end editing and cancel all changes made in the current editing session. If you've decided not to change the line, type Q to cancel changes and leave Edit Mode.

H (Hack and Insert)

Tells Computer to delete remainder of line and lets you insert material at the current cursor position. Hitting → will actually delete a character from the line in this mode. For example, using line 100 listed above, enter the Edit Mode and space over to the last statement, PRINT"DONE". Suppose you wish to delete this statement and insert an END statement. Display will show:

```
100 FOR I=1 TO 10 STEP .25 : PRINT I, I↑2, I↑3 : NEXT : _
```

Now type H and then type END. Hit ENTER key. List the line:

```
100 FOR I=1 TO 10 : STEP .25 : PRINT I, I↑2, I↑3 : NEXT : END
```

should be displayed.

*n*D (Delete)

Tells Computer to delete the specified number *n* characters to the right of the cursor. The deleted characters will be enclosed in exclamation marks to show you which characters were affected. For example, using line 100, space over to the PRINT command statement:

```
100 FOR I=1 TO 10 : STEP .25 : _
```

Now type 19D. This tells the Computer to delete 19 characters to the right of the cursor. The Display should show something like this:

```
100 FOR I=1 TO 10 : STEP .25 : !PRINT I, I↑2, I↑3 :!_
```

When you list the complete line, you'll see that the PRINT statement has been deleted.

*n*C (Change)

Tells the Computer to let you change the specified number of characters beginning at the current cursor position. If you type C without a preceding number, the Computer assumes you want to change one character. When you have entered *n* number of characters, the Computer returns you to the Edit Mode (so you're not in the *n*C Subcommand). For example, using line 100, suppose you want to change the final value of the FOR-NEXT loop, from "10" to "15". In the Edit Mode, space over to just before the "0" in "10".

```
100 FOR I=1 TO 1_
```

Now type C. Computer will assume you want to change just one character. Type 5, then hit L. When you list the line, you'll see that the change has been made.

```
100 FOR I=1 TO 15 STEP .25 : NEXT : END
```

would be the current line if you've followed the editing sequence in this chapter.

*n*S*c* (Search)

Tells the Computer to search for the *n*th occurrence of the character *c*, and move the cursor to that position. If you don't specify a value for *n*, the Computer will search for the first occurrence of the specified character. If character *c* is not found, cursor goes to the end of the line. Note: The Computer only searches through characters to the right of the cursor.

For example, using the current form of line 100, type EDIT 100 (ENTER) and then hit 2S: . This tells the Computer to search for the second occurrence of the colon character. Display should show:

```
100 FOR I=1 TO 15 STEP .25 : NEXT _
```

You may now execute one of the subcommands beginning at the current cursor position. For example, suppose you want to add the counter variable after the NEXT statement. Type I to enter the Insert subcommand, then type the variable name, I. That's all you want to insert, so hit SHIFT ↓ to escape from the Insert subcommand. The next time you list the line, it should appear as:

```
100 FOR I=1 TO 15 STEP .25 : NEXT I: END
```

*n*K*c* (Search and "Kill")

Tells the Computer to delete all characters up to the *n*th occurrence of character *c*, and move the cursor to that position. For example, using the current version of line 100, suppose we want to delete the entire line up to the END statement. Type EDIT 100 (ENTER), and then type 2K: . This tells the Computer to delete all characters up to the 2nd occurrence of the colon. Display should show:

```
100 !FOR I=1 TO 15 STEP .25 : NEXT I!_
```

The second colon still needs to be deleted, so type D . The Display will now show:

```
100 !FOR I=1 TO 15 STEP .25 : NEXT I!!:!_
```

Now hit ENTER and type LIST 100 (ENTER).

Line 100 should look something like this:

```
100 END
```

10/Expansion Interface

An Expansion Interface is available for the TRS-80 LEVEL II Computer. This interface will allow the use of additional Input/Output devices. There is also a provision for adding RAM memory. The Interface will allow four major additions to the TRS-80:

1. An additional cassette deck
2. A TRS-80 Line Printer
3. Up to four Mini-Disks
4. Up to 48K bytes of RAM Memory (32K in the Expansion Interface)

These devices are available from your Radio Shack store or dealer. To set up the Expansion Interface and any of the external devices, see the Expansion Interface instructions.

When the Expansion Interface is hooked up to the TRS-80, the Computer assumes that a Mini-Disk is interfaced. The Mini Disk will allow the use of additional commands and statements listed later. Even if you don't have a Mini Disk, the Computer will assume you do (because of the presence of the Disk Controller) and will try to input special instructions from the Disk Controller. Therefore, to use the Interface without a Mini Disk, hold down the BREAK key as you turn on the TRS-80. This will override the mini-disk mode and allow normal LEVEL II operation. Whenever you need to press the Reset button, you must also hold down the BREAK key.

Dual Cassettes

The use of two cassettes will allow a much more efficient and convenient manner of updating data stored on tape. For example, If you have payroll data stored on tape, the information can be read in one item at a time from cassette #1 then changed or added and written out on cassette #2, one item at a time. The routine might look like this:

```
10  INPUT #-1,A,B,C,D
20  PRINT "MAKE CORRECTIONS HERE: RETYPE LINE"
30  INPUT A,B,C,D
40  PRINT "THE LINE NOW READS:" A,B,C,D
50  PRINT "STORING ON TAPE #2..."
60  PRINT #-2, A,B,C,D
70  GOTO 10
```

This is a very simple application; however, very powerful routines can be constructed to allow input and output of data using two tapes simultaneously.

See Chapter 3, PRINT.

Codes

Several codes are used to control the output of the line printer. The codes and their functions are listed below. The CHR$ function is used to call up these function codes. For example:

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

will generate a line feed.

CODE	FUNCTION
10	line feed with carriage return
11	line feed with carriage return
12	Move carriage to top of form (page)
13	carriage return

NOTE: At the end of a line, a line feed is automatically generated unless a semi-colon is used at the end of the PRINT statement.

The line printer will print 6 lines per inch and 66 lines per page. If this format is not suitable, the number of lines per page can be changed by POKEing the new number of lines into memory location 16424.

Example:

```
POKE 16424, 40
```

This instructs the Line Printer to print 40 lines per page.

Mini-Disks – (LEVEL II DISK BASIC)

The TRS-80 Mini Disk System is a small version of a floppy disk. The disk allows vast file storage space and much quicker access time than you get with tape storage. Disc 0 will contain about 80,000 bytes of free space for files. Each additional disk will have 89,600 bytes of file space. The disk system has its own set of commands which allow manipulations of files and expanded abilities in file use. The TRS-80 Mini Disk System allows both sequential and random access. The disks will also allow use of several additional BASIC commands and functions:

Commands:

CLOSE	LSET	PUT
FIELD	NAME	RSET
GET	OPEN	MERGE
KILL	PRINT	LOAD
		SAVE

I/O Functions

CVD	LOF
CVI	MKD$
CVS	MKI$
EOF	MKS$
LOC	DSKF

Additions to LEVEL II

Ten USR calls – USRØ through USR9	INSTR	(performs function of INSTRING subroutine – see Chapter 4)
&H (hex constants)	TIME$	(Date and 24-Hr. Real-Time Clock.)
&O (octal constants)	DEF FN	(User Defined Functions)
DEFUSR		
LINE INPUT		
MID$ (on left side of equation)		

For explanation of these commands functions, see the TRS-80 Disk Operating System Manual.

Expansion of RAM Memory

The TRS-80 Expansion Interface has provisions for adding extra RAM memory. This is done by adding RAM memory chips. You can add up to 32,768 additional bytes of memory. For price information and installation, see your Radio Shack store or dealer.

11/Saving Time and Space

Most LEVEL II programs are faster and take up less memory space than their LEVEL I counterparts. But even with its inherently more efficient features, LEVEL II can be further streamlined by following a few simple guidelines when constructing your program.

Saving Memory Space

1) When your program is operating properly, delete all unnecessary REM statements from your running version.

2) Do not use unnecessary spaces between statements, operators, etc.

3) When possible, use multiple-statement program lines (with a colon between each two statements). Each time you enter a new line number it costs you 5 bytes.

4) Use integer variables whenever possible, for example,

```
FOR I% = 1 TO 10
```

Integers take only two bytes. Single precision takes 7 and double precision takes 11 bytes.

5) Using subroutines will save program space if the operation is called from different places several times. If a routine is always called from the same place, use unconditional branches (GOTO's). Each active GOSUB takes 6 bytes; a GOTO takes none at Run time.

6) Structure your calculations so as to use as few parentheses as possible (refer to Chapter 1, "Arithmetic Operators"). It takes 4 bytes to process parentheses. And since these operations inside parentheses are done first, the result of each parenthetical expression must be stored (this takes 12 bytes).

7) Dimension arrays sparingly. When you set up a matrix, the Computer reserves 11 subscript addresses for each DIMension, even if the space is not filled. Use the zero subscripted elements, since they are always available.

8) Use DEF statements when you will be working with values other than single precision (strings, integers and double precision). A DEF statement takes 6 bytes but this is made up for fairly quickly since you don't need to use type declaration characters with the variable names.

Speeding Up Execution

The speed at which a program is processed will depend on the complexity of the operations and the number of instructions. In most simple programs, speed will not be a factor. It will seem as though the answer is returned the moment you enter RUN. However, as you begin writing longer and more intricate programs, speed will become a significant factor. Here are some suggestions to guide you in designing speedier programs.

1) Delete all unnecessary lines in the program (REM statements, etc.)

2) Combine multi-statement program lines when practical.

3) Use variables rather than constants in operations (very important). Your TRS-80 normally operates using floating decimal point values. It takes a lot less time to access a variable than to convert a constant to floating point representation. For example: if you will use π a lot in a program, define π as a variable (PI=3.14159) and use the variable (PI) in the operations.

4) Use POKE graphics. This can speed up your graphics displays by a factor of 6.

5) Define the most commonly used variables first. When a variable is defined it is located at the top of the variable table. The second will be just below that. When variables are accessed, the table will be searched to find the variable. Therefore, you will save time by locating frequently used variables at the top of the table (by defining them first). The Computer will not have to look as far to find them.

6) Use integer variables, especially in FOR-NEXT loops, when possible. This is most important of all.

A/Level II Summary

Special Characters and Abbreviatons

Command Mode	
ENTER	Return carriage and interpret command
←	Cursor backspace and delete last character typed
SHIFT ←	Cursor to beginning of logical line; erase line
↓	Linefeed
:	Statement delimiter; use between statements on same logical line
→	Move cursor to next tab stop. Tab stops are at positions 0, 8, 16, 24, 32, 40, 48, and 56.
SHIFT →	Convert display to 32 characters per line
CLEAR	Clear Display and convert to 64 characters per line

Execute Mode	
SHIFT @	Pause in execution; freeze display during LIST
BREAK	Stop execution
ENTER	Interpret data entered from Keyboard with INPUT statement

Abbreviations	
?	Use in place of PRINT.
'	Use in place of :REM
.	"Current line"; use in place of line number with LIST, EDIT, etc.

Type Declaration Characters

Character	Type	Examples
$	String	A$, ZZ$
%	Integer	A1%, SUM%
!	Single-Precision	B!, N1!
#	Double-Precision	A#, 1/3#
D	Double-Precision (exponential notation)	1.23456789D-12
E	Single-Precision (exponential notation)	1.23456E+30

Arithmetic Operators

+ add − subtract * multiply / divide
↑ exponentiate (e.g., 2 ↑ 3 = 8)

String Operator

+ concatenate (string together) "2" + "2" = "22"

Relational Operators

Symbol	meaning in numeric expressions	in string expressions
<	is less than	precedes
>	is greater than	follows
=	is equal to	equals
<= or =<	is less than or equal to	precedes or equals
>= or =>	is greater than or equal to	follows or equals
<> or ><	does not equal	does not equal

Order of Operations (operators on same line have same precedence)

↑ (Exponentiation)
- (Negation)
*, /
+, -
Relational operators
NOT
AND
OR

Commands

Command	Function	Examples
AUTO *mm,nn*	Turn on automatic line numbering beginning with *mm*, using increment of *nn*.	AUTO AUTO 10 AUTO 5,5 AUTO .,10
CLEAR	Set numeric variables to zero, strings to null.	CLEAR
CLEAR *n*	Same as CLEAR but also sets aside *n* bytes for strings.	CLEAR 500 CLEAR MEM/4
CONT	Continue after BREAK or STOP in execution.	CONT
DELETE *mm-nn*	Delete program lines from line *mm* to line *nn*.	DELETE 100 DELETE 10-50 DELETE .
EDIT *mm*	Enter Edit Mode for line *mm*. See Edit Mode Subcommands below.	EDIT 100 EDIT .
LIST *mm-nn*	List all program lines from *mm* to *nn*.	LIST LIST 30-60 LIST 30- LIST -90 LIST .

NEW	Delete entire program and reset all variables, pointers etc.	NEW
RUN *mm*	Execute program beginning at lowest numbered line or *mm* if specified.	RUN RUN 55
SYSTEM	Enter Monitor Mode for loading of machine-language file from cassette.	See Chapter 2
TROFF	Turn off Trace	TROFF
TRON	Turn on Trace	TRON

Edit Mode Subcommands and Function Keys

Subcommand/Function Key	Function
ENTER	End editing and return to Command Mode.
SHIFT ↑	Escape from subcommand and remain in Edit Mode.
*n*Space-Bar	Move cursor *n* spaces to right.
n ←	Move cursor *n* spaces to left.
L	List remainder of program line and return to beginning of line.
X	List remainder of program line, move cursor to end of line, and start Insert subcommand.
I	Insert the following sequence of characters at current cursor position; use Escape to exit this subcommand.
A	Cancel changes and return cursor to beginning of line.
E	End editing, save all changes and return to Command Mode.
Q	End editing, cancel all changes made and return to Command Mode.

H	Delete remainder of line and insert following sequence of characters; use Escape to exit this subcommand.
*n*D	Delete specified number of characters *n* beginning at current cursor position.
*n*C	Change (or replace) the specified number of characters *n* using the next *n* characters entered.
*n*S*c*	Move cursor to *n*th occurrence of character *c*, counting from current cursor position.
*n*K*c*	Delete all characters from current cursor position up to *n*th occurrence of character *c*, counting from current cursor position.

Input/Output Statements

Statement*	Function	Examples
PRINT *exp*	Output to Display the value of *exp*. *Exp* may be a numeric or string expression or constant, or a list of such items.	PRINT A$ PRINT X+3 PRINT "D=" D
	Comma serves as a PRINT modifier. Causes cursor to advance to next print zone.	PRINT 1,2,3,4 PRINT "1","2" PRINT 1,,2
	Semi-colon serves as a PRINT modifier. Inserts a space after a numeric item in PRINT list. Inserts no space after a string item. At end of PRINT list, suppresses the automatic carriage return.	PRINT X;"=ANSWER" PRINT X;Y;Z PRINT "ANSWER IS";

**exp* may be a string or numeric constant or variable, or a list of such items.

PRINT @*n*	PRINT modifier; begin PRINTing at specified display position *n*.	PRINT @ 540,"CENTER" PRINT @ N+3,X*3
TAB *n*	Print modifier: moves cursor to specified Display position *n* (expression).	PRINT TAB(N) N
PRINT USING *string;exp*	PRINT format specifier: output *exp* in form specified by *string* field (see below).	PRINT USING A$;X PRINT USING "#.#";Y+Z
INPUT *"message";variable*	Print message (if any) and await input from Keyboard.	INPUT"ENTER NAME":A$ INPUT"VALUE": X INPUT"ENTER NUMBERS" ;X,Y INPUT A,B,C,D$
PRINT #−1	Output to Cassette #1.	PRINT #—1,A,B,C,D$
INPUT #−1	Input from Cassette #1.	INPUT #—1,A,B,C,D$
DATA *item list*	Hold data for access by READ statement.	DATA 22,33,11,1.2345 DATA "HALL","SMITH","DOE"
READ *variable list*	Assign value(s) to the specified variable(s), starting with current DATA element.	READ A,A1,A2,A3 READ A$,B$,C$,D
RESTORE	Reset DATA pointer to first item in first DATA statement.	RESTORE

Field Specifiers for PRINT USING statements

Numeric Character	Function	Example
#	Numeric field (one digit per #).	###
.	Decimal point position.	##.###
+	Print leading or trailing sign (plus for positive numbers, minus for negative numbers).	+#.### #.###+ -#.### #.###-
-	Print trailing sign only if value printed is negative.	###.##-
**	Fill leading blanks with asterisk.	**###.##
$$	Place dollar sign immediately to left of leading digit.	$$####.##
**$	Dollars sign to left of leading digit and fill leading blanks with asterisks.	**$####.##
↑↑↑↑	Exponential format, with one significant digit to left of decimal.	#.#######↑↑↑↑

String Character	Function	Example
!	Single character.	!
%spaces%	String with length equal to 2 plus number of spaces between % symbols.	% %

Statement	Function	Examples
(Type Definition)		
DEFDBL *letter list or range*	Define as double-precision all variables beginning with specified letter, letters or range of letters.	DEFDBL J DEFDBL X,Y,A DEFDBL A-E,J
DEFINT *letter list or range*	Define as integer all variables beginning with specified letter, letters or range of letters.	DEFINT A DEFINT C,E,G DEFINT A-K
DEFSNG *letter list or range*	Define as single-precision all variables beginning with specified letter, letters or range of letters.	DEFSNG L DEFSNG A-L, Z DEFSNG P,R,A-K
DEFSTR *letter list or range*	Define as string all variables beginning with specified letter, letters or range of letters.	DEFSTR A,B,C DEFSTR S,X-Z DEFSTR M
(Assignment and Allocation)		
CLEAR *n*	Set aside specified number of bytes *n* for string storage.	CLEAR 750 CLEAR MEM/10 CLEAR 0
DIM *array(dim*1, . . . ,dim*k)*	Allocate storage for *k*-dimensional *array* with the specified size per dimension: dim*1, dim*2, . . ., etc. DIM may be followed by a list of arrays separated by commas.	DIM A(2,3) DIM A1(15), A2(15) DIM B(X+2),C(J,K) DIM T(3,3,5)

Statement	Function	Examples
LET *variable=expression*	Assign value of *expression* to *variable*. LET is optional in LEVEL II BASIC.	LET A$="CHARLIE" LET B1=C1 LET A%=I#
(Sequence of Execution)		
END	End execution, return to Command Mode.	99 END
STOP	Stop execution, print Break message with current line number. User may continue with CONT.	100 STOP
GOTO *line-number*	Branch to specified *line-number*.	GOTO 100
GOSUB *line-number*	Branch to subroutine beginning at *line-number*.	GOSUB 3000
RETURN	Branch to statement following last-executed GOSUB.	RETURN
ON *exp* GOTO *line*1, . . . ,line*k*	Evaluate expression, if INT(exp) equals one of the numbers 1 through *k*, branch to the appropriate line number. Otherwise go to next statement.	ON K+1 GOTO 100,200,300
ON *exp* GOSUB *line*1,. . .,line*k*	Same as ON . . . GOTO except branch is to subroutine beginning at *line*1*, *line*2*, . . . , or *line*k*, depending on *exp*.	ON J GOSUB 330,7000

Statement	Function	Examples
FOR *var=exp* TO *exp* STEP *exp*	Open a FOR-NEXT loop. STEP is optional, if not used, increment of one is used. See Chapter 4.	FOR I=1 TO 50 STEP 1.5 FOR M%=J% TO K-1%
NEXT *variable*	Close FOR-NEXT loop. *Variable* may be omitted. To close nested loops, a variable list may be used. See Chapter 4.	NEXT NEXT I NEXT I,J,K
ERROR (*code*)	Simulate the error specified by *code* (see Error Code Table).	ERROR (14)
ON ERROR GOTO *line-number*	If an error occurs in subsequent program lines, branch to error routine beginning at *line-number.*	ON ERROR GOTO 999
RESUME *n*	Return from error routine to line specified by *n*. If *n* is zero or not specified, return to statement containing error. If *n* is "NEXT", return to statement following error-statement.	RESUME RESUME 0 RESUME 100 RESUME NEXT
RANDOM	Reseeds random number generator	RANDOM
REM	REMark indicator, ignore rest of line	REM A IS ALTITUDE

Statement	Function	Examples
(Tests – Conditional Statements)		
IF *exp–1* THEN *statement–1* ELSE *statement–2*	Tests *exp–1*: If True, execute *statement -1* then jump to next program line (unless statement - 1 was a GOTO). If *exp–1* is False, jump directly to ELSE statement and execute subsequent statements.	IF A=0 THEN PRINT "ZERO" ELSE PRINT "NOT ZERO"
(Graphics Statements)		
CLS	Clear Video Display.	CLS
RESET(*x,y*)	Turn off the graphics block with horizontal coordinate *x* and vertical coordinate *y*. 0<=X<128 and 0<=Y<48	RESET(8+B,11)
SET (*x,y*)	Turn on the graphics block specified by coordinates *x* and *y*. Same argument limits as RESET.	SET(A*2,B+C)
(Special Statements)		
POKE *location,value*	Load *value* into memory *location* (both arguments in decimal form) 0<=*value*<=255.	POKE 15835,34 POKE 17770,A+N
OUT *port,value*	Send *value* to *port* (both arguments between 0 and 255 inclusive)	OUT 255,10 OUT 55,A

String Functions

Function	Operation	Examples
ASC(*string*)	Returns ASCII code of first character in string argument.	ASC(B$) ASC("H")
CHR$(*code exp*)	Returns a one-character string defined by *code*. If *code* specifies a control function, that function is activated.	CHR$(34) CHR$(I)
FRE(*string*)	Returns amount of memory available for string storage Argument is a dummy variable.	FRE(A$)
INKEY$	Strobes Keyboard and returns a one-character string corresponding to key pressed during strobe (null string if no key is pressed).	INKEY$
LEN(*string*)	Returns length of *string* (zero for null string).	LEN(A$+B$) LEN("HOURS")
LEFT$(*string,n*)	Returns first *n* characters of *string*.	LEFT$(A$,1) LEFT$(L1$+C$,8) LEFT$(A$,M+L)
MID$(*string,p,n*,)	Returns substring of *string* with length *n* and starting at position *p* in *string*.	MID$(M$,5,2) MID$(M$+B$,P,L—1)
RIGHT$(*string,n*)	Returns last *n* characters of *string*.	RIGHT$(NA$,7) RIGHT$(AB$,M2)
STR$(*numeric exp*)	Returns a string representation of the evaluated argument.	STR$(1.2345) STR$(A+B*2)
STRING$(*n,char*)	Returns a sequence of *n char* symbols using first character of *char*.	STRING$(30, ".") STRING$(25, "A") STRING$(5,C$)
VAL(*string*)	Returns a numeric value corresponding to a numeric-valued string.	VAL("1"+A$+"."+C$) VAL(A$+B$) VAL(G1$)

**string* may be a string variable, expression, or constant.

Arithmetic Functions*

Function	Operation (unless noted otherwise, $-1.7E+38 <= exp <= 1.7E+38$)	Examples
ABS(*exp*)	Returns absolute value.	ABS(L*.7) ABS(SIN(X))
ATN(*exp*)	Returns arctangent in radians.	ATN(2.7) ATN(A*3)
CDBL(*exp*)	Returns double-precision representation of *exp*.	CDBL(A) CDBL(A+1/3#)
CINT(*exp*)	Returns largest integer not greater than *exp*. Limits: $-32768 <= exp < +32768$.	CINT(A#+B)
COS(*exp*)	Returns the cosine of *exp*; assumes *exp* is in radians.	COS(2*A) COS(A/57.29578)
CSNG(*exp*)	Returns single-precision representation, with 5/4 rounding in least significant decimal when *exp* is double-precision.	CSNG(A#) CSNG(.33*B#)
EXP(*exp*)	Returns the natural exponential, e^{exp} =EXP(*exp*).	EXP(34.5) EXP(A*B*C-1)
FIX(*exp*)	Returns the integer equivalent to truncated *exp* (fractional part of *exp* is chopped off).	FIX(A-B)
INT(*exp*)	Returns largest integer not greater than *exp*.	INT(A+B*C)
LOG(*exp*)	Returns natural logarithm (base e) of *exp*. Limits. *exp* must be positive.	LOG(12.33) LOG(A↑B+B)
RND(0)	Returns a pseudo-random number between 0.000001 and 0.999999 inclusive.	RND(0)
RND(*exp*)	Returns a pseudo-random number between 1 and INT(*exp*) inclusive. Limits: $1 <= exp < 32768$.	RND(40) RND(A+B)
SGN(*exp*)	Returns -1 for negative *exp*; 0 for zero *exp*; +1 for positive *exp*.	SGN(A*B+3) SGN(COS(X))

*exp is any numeric-valued expression or constant.

SIN(*exp*)	Returns the sine of *exp*; assumes *exp* is in radians.	SIN(A,B) SIN(90/57.29578)
SQR(*exp*)	Returns square root of *exp*. Limits: *exp* must be non-negative.	SQR(A*A - B*B)
TAN(*exp*)	Returns the tangent of *exp*; assumes *exp* is in radians.	TAN(X) TAN(X*.0174533)

Special Functions

Function	Operation and Limits	Examples
ERL	Returns line number of current error.	ERL
ERR	Returns a value related to current error code (if error has occurred). ERR = (error code-1)*2. Also: (ERR/2)+1 = error code.	ERR/2+1
INP(*port*)	Inputs and returns the current value from the specified *port*. Both argument and result are in the range 0 to 255 inclusive.	INP(55)
MEM	Returns total unused and unprotected bytes in memory.	MEM
PEEK(*location*)	Returns value stored in the specified memory byte. *location* must be a valid memory address in decimal form (see Memory Map in Appendix D).	PEEK(15370)
POINT (*x*,*y*)	Checks the graphics block specified by horizontal coordinate *x* and vertical coordinate *y*. If block is "on", returns a True (-1); if block is "off", returns a False (0). Limits: 0<=x<128,0<=y<48.	
POS(0)	Returns a number indicating the current cursor position. The argument "0" is a dummy variable.	POS(0)
USR(*n*)	Branches to machine language subroutine. For LEVEL II BASIC, *n* must equal 0. See Chapter 8.	USR(0)
VARPTR(*var*)	Returns the address where the specified variable's name, value, and pointer are stored. *var* must be a valid variable name. Returns 0 if *var* has not been assigned a value.	VARPTR(A$) VARPTR(N!)

LEVEL II Reserved Words*

@	FIX	OUT
ABS	FOR	PEEK
AND	FRE	POINT
ASC	GET	POKE
ATN	GOSUB	POS
CDBL	GOTO	PRINT
CHR$	IF	PUT
CINT	INKEY$	RANDOM
CLEAR	INP	READ
CLOSE	INPUT	REM
CLS	INSTR	RESET
CMD	INT	RESTORE
CONT	KILL	RESUME
COS	LEFT$	RETURN
CSNG	LET	RIGHT$
CVD	LSET	RND
CVI	LEN	SAVE
CVS	LINE	SET
DATA	LIST	SGN
DEFDBL	LOAD	SIN
DEFFN	LOC	SQR
DEFINT	LOF	STEP
DEFSNG	LOG	STOP
DEFUSR	MEM	STRING$
DEFSTR	MERGE	STR$
DELETE	MID$	TAB
DIM	MKD$	TAN
EDIT	MKI$	THEN
ELSE	MKS$	TIME$
END	NAME	TROFF
ERL	NEW	TRON
ERR	NEXT	USING
ERROR	NOT	USR
EXP	ON	VAL
FIELD	OPEN	VARPTR

* Many of these words have no function in LEVEL II BASIC; they are reserved for use in LEVEL II DISK BASIC. None of these words can be used inside a variable name.

Program Limits and Memory Overhead

Ranges

Integers	-32768 to +32767 inclusive
Single Precision	-1.701411E+38 to +1.701411E+38 inclusive
Double Precision	-1.701411834544556E+38 to +1.701411834544556E+38 inclusive

String Range: Up to 255 characters

Line Numbers Allowed: 0 to 65529 inclusive

Program Line Length: Up to 255 characters

Memory Overhead

Program lines require 5 bytes minimum, as follows:
- Line Number – 2 bytes
- Line Pointer – 2 bytes
- Carriage Return – 1 byte

In addition, each reserved word, operator, variable name, special character and constant character requires one byte.

Dynamic (RUN-time) Memory Allocation

Integer variables: 5 bytes each
(2 for value, 3 for variable name)

Single-precision variables: 7 bytes each
(4 for value, 3 for variable name)

Double-precision variables: 11 bytes each
(8 for value, 3 for variable name)

String variables: 6 bytes minimum
(3 for variable name, 3 for stack and variable pointers, 1 for each character)

Array variables: 12 bytes minimum
(3 for variable name, 2 for size, 1 for number of dimensions, 2 for each dimension, and 2, 3, 4 or 8 [depending on array type] for each element in the array)

Each active FOR-NEXT loop requires 16 bytes.

Each active (non-returned) GOSUB requires 6 bytes.

Each level of parentheses requires 4 bytes plus 12 bytes for each temporary value.

B/LEVEL II Error Codes

CODE	ABBREVIATION	ERROR
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	Subscript out of range
10	DD	Redimensioned 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	NR	NO RESUME
19	RW	RESUME without error
20	UE	Unprintable error
21	MO	Missing operand
22	FD	Bad file data
23	L3	Disk BASIC only*

Explanation of Error Messages

NF NEXT without FOR: NEXT is used without a matching FOR statement. This error may also occur if NEXT *variable* statements are reversed in a nested loop.

SN Syntax Error: This usually is the result of incorrect punctuation, open parenthesis, an illegal character or a mis-spelled command.

RG RETURN without GOSUB: A RETURN statement was encountered before a matching GOSUB was executed.

OD Out of Data. A READ or INPUT # statement was executed with insufficient data available. DATA statement may have been left out or all data may have been read from tape or DATA.

FC Illegal Function Call: An attempt was made to execute an operation using an illegal parameter. Examples: square root of a negative argument, negative matrix dimension, negative or zero LOG arguments, etc. Or USR call without first POKEing the entry point.

OV Overflow: A value input or derived is too large or small for the computer to handle.

OM Out of Memory: All available memory has been used or reserved. This may occur with very large matrix dimensions, nested branches such as GOTO, GOSUB, and FOR-NEXT Loops.

UL Undefined Line: An attempt was made to refer or branch to a non-existent line.

BS Subscript out of Range: An attempt was made to assign a matrix element with a subscript beyond the DIMensioned range.

DD Redimensioned Array: An attempt was made to DIMension a matrix which had previously been dimensioned by DIM or by default statements. It is a good idea to put all dimension statements at the beginning of a program.

/0 Division by Zero: An attempt was made to use a value of zero in the denominator.

ID Illegal Direct: The use of INPUT as a direct command.

TM Type Mismatch: An attempt was made to assign a non-string variable to a string or vice-versa.

OS Out of String Space: The amount of string space allocated was exceeded.

LS String Too Long: A string variable was assigned a string value which exceeded 255 characters in length.

ST String Formula Too Complex: A string operation was too complex to handle. Break up the operation into shorter steps.

CN Can't Continue: A CONT was issued at a point where no continuable program exists, e.g., after program was ENDed or EDITed.

NR No RESUME: End of program reached in error-trapping mode.

RW RESUME without ERROR: A RESUME was encountered before ON ERROR GOTO was executed.

UE Unprintable Error: An attempt was made to generate an error using an ERROR statement with an invalid code.

MO Missing Operand: An operation was attempted without providing one of the required operands.

FD Bad File Data: Data input from an external source (i.e., tape) was not correct or was in improper sequence, etc.

L3 DISK BASIC only: An attempt was made to use a statement, function or command which is available only when the TRS-80 Mini Disk is connected via the Expansion Interface.

C/Control, Graphics, and ASCII Codes

Control Codes: 1-31

Code	Function
0-7	None
8	Backspaces and erases current character
9	None
10-13	Carriage returns
14	Turns on cursor
15	Turns off cursor
16-22	None
23	Converts to 32 character mode
24	Backspace → Cursor
25	Advance ← Cursor
26	Downward ↑ linefeed
27	Upward ↓ linefeed
28	Home, return cursor to display position(0,0)
29	Move cursor to beginning of line
30	Erases to the end of the line
31	Clear to the end of the frame

ASCII Character Codes 32-128

Code	Character
32	space
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	Ø
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	↑ or [
92	↓
93	←
94	→
95	_
96-127	lower case for codes 64-95
128	Space

Graphics Codes 129-191

You can examine these codes using:

```
10 FOR X = 129 TO 191
20 PRINT X;:PRINT CHR$(X),
30 NEXT
```

Space Compression Codes: 192 TO 255

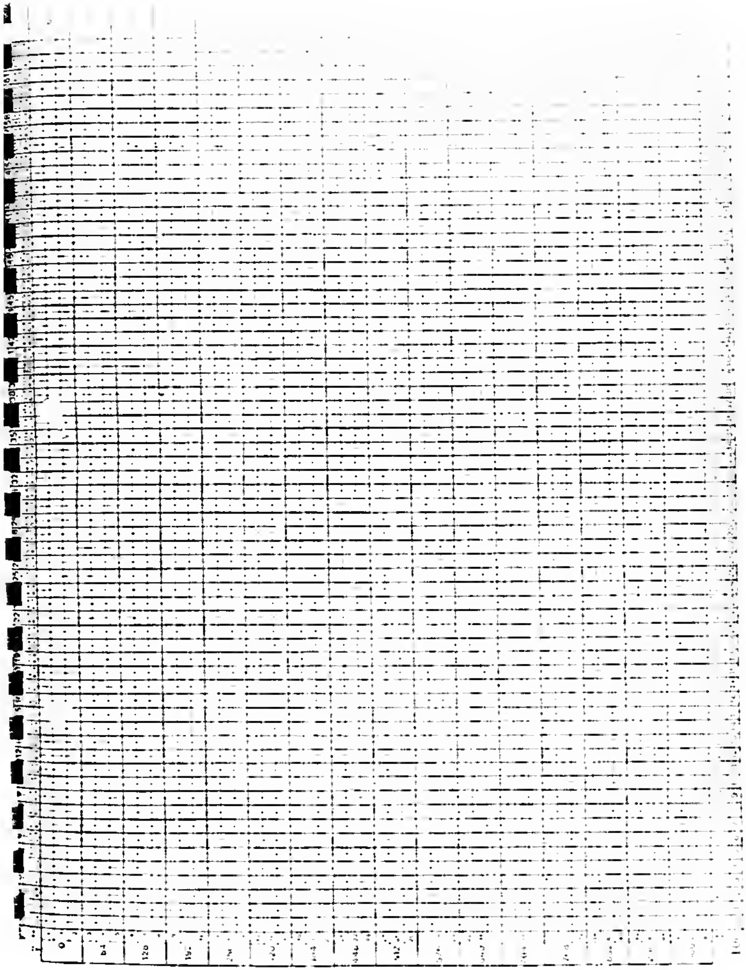
Code	Function
192-255	Tabs for 0 TO 63 spaces, respectively

D/LEVEL II TRS-80 MEMORY MAP

ADDRESS

DECIMAL	HEXIDECIMAL	
0	0000	LEVEL II BASIC ROM
12288	3000	RESERVED
14302	37DE	COMMUNICATION STATUS ADDRESS
14303	37DF	COMMUNICATION DATA ADDRESS
14304	37E0	INTERRUPT LATCH ADDRESS
14305	37E1	DISK DRIVE SELECT LATCH ADDRESS
14308	37E4	CASSETTE SELECT LATCH ADDRESS
14312	37E8	LINE PRINTER ADDRESS
14316	37EC	FLOPPY DISK CONTROLLER ADDRESS
14336	3800	TRS-80 KEYBOARD MEMORY
15360	3C00	TRS-80 CRT VIDEO MEMORY
16383	3FFF	
16384	4000	LEVEL II BASIC FIXED RAM VECTORS (RST'S 1 THROUGH 7)
16402	4012	
16405	4015	KEYBOARD DEVICE CONTROL BLOCK DCB + 0 = DCB TYPE + 1 = DRIVER ADDRESS + 2 = DRIVER ADDRESS + 3 = 0 + 4 = 0 + 5 = 0 + 6 = 'K' + 7 = 'I'
16413	401D	VIDEO DISPLAY CONTROL BLOCK DCB + 0 = DCB TYPE + 1 = DRIVER ADDRESS (LSB) + 2 = DRIVER ADDRESS (MSB) + 3 = CURSOR POS'N (LSB) + 4 = CURSOR POS'N (MSB) + 5 = CURSOR CHARACTER + 6 = 'D' + 7 = 'O'
16421	4025	LINE PRINTER CONTROL BLOCK DCB + 0 = DCB TYPE + 1 = DRIVER ADDRESS (LSB) + 2 = DRIVER ADDRESS (MSB) + 3 = LINES/PAGE + 4 = LINE COUNTER + 5 = 0 + 6 = 'P' + 7 = 'R'

Decimal	Hex	Contents
16429	402D	RESERVED
16463	404F	
16464	4050	FDC INTERRUPT VECTOR
16466	4052	COMMUNICATIONS INTERRUPT VECTOR
16468	4054	RESERVED
16476	405C	
16478	405E	25 MSEC HEARTBEAT INTERRUPT
		RESERVED
16512	4080	LEVEL II BASIC RAM
		RESERVED
16870	41E6	I/O BUFFER
17127	42E7	
17128	42E8	ALWAYS ZERO
17129	42E9	
		↓ PROGRAM TEXT
		↓ SIMPLE VARIABLES
		↓ ARRAYS
		FREE MEMORY
		↑ STACK
		↑ STRING SPACE
		SPACE RESERVED FOR MACHINE LANGUAGE ROUTINES TO BE ACCESSED FROM BASIC – IF MEMORY SIZE SET
20479 (4K)	4FFF (4K)	
32767 (16K)	7FFF (16K)	END OF ACTUAL MEMORY



Σ/l

F/Derived Functions

Function	Function Expressed in Terms of Level II Basic Functions
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(-X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = -ATN (X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPOBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X)*2+1
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))*2+1
INVERSE HYPERBOLIC SINE	ARGSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X) = LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X) = LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X) = LOG((X+1)/(X-1))/2

H/User Programs

Space-Ship Lander

This challenging program lets you simulate a landing sequence on any of four planetary bodies: Earth, Moon, Mars, and the asteroid Vesta. Before each 10-second "burn" interval, you are given the following information:

Elapsed Time (seconds)
Altitude (kilometers)
Velocity (kilometers/hour – negative amount indicates motion away from planetary body)
Remaining Fuel (kilograms)

Using this information, you select a "burn rate" (kilograms of fuel/second). For example, a 10 Kg/sec burn rate consumes 100 Kg during the 10-second mandatory burn interval. Burns must be in the range 0-100 Kg/sec (over 100 Kg/sec would cause the "G" force to become too great.)

Hints:

- A negative velocity indicates you burned too much fuel and are moving away from the planetary body.
- Fuel burns may include fractional parts (e.g., 15.5 Kg/sec)
- As you consume fuel, the weight of the lander decreases. Therefore subsequent burns will be increasingly effective.
- Landing conditions are different for each planetary body because each has its own particular gravity: Earth = 980 cm/sec^2; Moon = 162 cm/sec^2; Mars = 372 cm/sec^2; Vesta = 17.5 cm/sec^2.
- The up arrow ↑ appears as a left bracket [in this printout. Remember to enter it as an up arrow (for exponentiation).

Good luck, Commander!

```
100 CLS
110 PRINT@ 20, "* * * LANDER * * *"
120 PRINT : PRINT "TYPE 1 FOR EARTH, 2 FOR MOON, 3 FOR MARS, 4 FOR VESTA"
140 INPUT X : ON X GOTO 500, 600, 700, 300
145 CLS
147 PRINT@ 980, A$;
150 G2 = G1/35
160 G3 = SQR(G2) * 120 : G3=FIX(G3) : IF G3<175 THEN G3=175
170 G4 = G3 * 55 : G4=FIX(G4) : IF G4>10000 THEN G4=10000
180 G5 = G4 + [illegible] * 10000
190 A1 = -7400 : A2 = 3300 : A3 = 17000 : A4 = 10
200 B4 = A4 : B3 = A2 : [illegible] : [illegible] = G4
```

```
205 PRINT@ 0, "ELAPSED     ALTITUDE     VELOCITY     PEMAINING    INPUT FUEL",
206 PRINT@ 64, "TIME         (KM)          (KM/HR)        FUEL        BURN (KG/SEC)",
210 PRINT@ 128+Q , T1; TAB(10) N3, TAB(24) B2, TAB(39) N4, TAB(53) ,. INPUT F
250 IF F=0 GOTO 280
260 IF F<0 OR F>100 GOTO 320
270 T = N4/F : IF T<10 THEN B4=T
280 N4 = N4 - (F*B4)
285 V1=03
286 T1=T1+B4
290 B5 = (G2+ ((G2 * N3)/(A5 * -2))) - ((F * G5)/(A3 + N4))
295 03 = B2 + (B5 * B4)
298 N5=N3
300 N3 = N3+ (((B3 + B2) / A1) * B4)
305 B2=03
307 IF N3<0 GOTO 450
310 IF N4 <= 0 GOTO 400 : GOTO 210
312 Q=Q+64 . IF Q + 128 > 960 THEN Q=032
315 GOTO 205
320 PRINT " -->> ILLEGAL FUEL BURN - DUMMY!--TRY AGAIN (0 TO 100)" : GOTO 210
400 V2 = SQR (B2[2 + N3 * G2 * 5650) : PRINT "OUT OF FUEL AT"; T1 ;"SECONDS"
410 V3 = ABS(V2) * 10000 / 3600
420 T1 = T1 + LOG ((V3 * N3 * 10000) / G1)
430 GOTO 1000
450 V2 = SQR (ABS (N5 / (26 * B5))) * (26 * B5) + V1 : GOTO 1000
460 T1=T1-(10-B4)
500 01 = 980.7 : A5 = 6371 : A$="EARTH" : GOTO145
600 G1 = 162 : A5 = 1738 : A$="MOON" : GOTO 145
700 01 = 374 : A5 = 3380 : A$="MARS" : GOTO 145
800 G1 = 17 5   A5 = 195 : A$="VESTA" : GOTO 145
1000 PRINT   PRINT "YOU HAVE ",
1010 IF V2<20 PRINT "LANDEO"   GOTO 1100
1020 IF V2<100 PRINT "CPASHED"   GOTO 1140
1030 IF V2<250 PRINT "BEEN OBLITURATED"   GOTO 5000
1040 IF V2<5000 PRINT "MADE A NEW CRATER"   GOTO 5000
1050 IF V2>4999 PRINT "BORED A HOLE INTO THE PLANET" . GOTO 5000
1100 IF V2<1 PRINT "NICE TOUCH--VERY GOOD"   GOTO 5000
1110 IF V2<5 PRINT "NOT TOO BAO"   GOTO 5000
1120 PRINT "KIND OF ROUGH"   GOTO 5000
1140 IF V2<30 PRINT "YOU WILL NOT BE ABLE TO TAKE OFF" : GOTO 5000
1150 IF V2<45 PRINT "YOU ARE INJUREO, THE LANDER IS ON FIRE" : GOTO 5000
1160 PRINT "THERE ARE NO SURVIVORS"
5000 PRINT "VELOCITV AT IMPACT * * *" TAB(40), ABS(V2) ,"KM/HR"
5010 FRINT "ELAPSED TIME * * * * * *" TAB(40); T1 ,"SECONDS"
5020 END
```

Customer Information

This program allows you (or your customers) to store information in a file for future reference. It stores Name, Address and Phone Number; the file can be recalled, modified, etc., by specifying the desired action using the "Menu" (Table of Commands).

This would be a handy way to create a mailing/phone list.

```
10 CLEAR 1000 :CLS :DIM N$(50) :DIM A$(50) :DIM P$(50)
20 CLS :PRINT@ 10,"* * MENU * *" :PRINT :PRINT
30 PRINT "TO BUILD A FILE TYPE 1
40 PRINT "TO SEE THE ENTIRE FILE TYPE 2
50 PRINT "TO SEE AN INDIVIDUAL NAME TYPE 3
60 PRINT "TO MAKE CORRECTIONS TYPE 4
70 PRINT "TO SAVE THE CURRENT FILE ON TAPE TYPE 5
80 PRINT "TO INPUT A FILE FROM TAPE TYPE 6
90 INPUT Q :ON Q GOTO 100,200,300,400,500,600
100 INPUT"WHEN READY, NIT ENTER (TO CLOSE THE FILE TYPE 9999 FOR NAME)";X
110 FOR I=1 TO 50 :CLS :PRINT"ENTER YOUR NAME (LAST FIRST, NO COMMAS PLEASE)
112 PRINT"THEN NIT THE 'ENTER' KEY" ; :INPUT N$(I)
115 IF N$(I)="9999" THEN P1=I :GOTO150
120 INPUT"ENTER YOUR ADDRESS (NO COMMAS)"; A$(I)
130 INPUT"ENTER YOUR PHONE # "; P$(I)
135 IF FRE(X$) < 100 GOTO150
140 NEXT
150 PRINT"FILE CLOSED —" :INPUT"TO SEE THE MENU, HIT ENTER"; X
160 GOTO20
200 CLS :FOR I=1 TO P1 :PRINT N$(I), A$(I), P$(I) :NEXT
210 INPUT"TO SEE THE MENU, NIT ENTER"; X :GOTO20
300 CLS :INPUT"ENTER THE NAME, LAST FIRST (NO COMMAS)"; N$
310 FOR I=1 TO P1 :IF N$(I)=N$ THEN330
315 NEXT
320 PRINT"NAME NOT IN FILE" :GOTO340
330 PRINT N$(I), A$(I), P$(I)
340 PRINT :PRINT"FOR ANOTHER NAME TYPE 1, OTHERWISE 0"; :INPUT X
350 IF X=1 GOTO300 ELSE20
400 CLS :PRINT"ENTER THE NAME FOR THE LINE YOU WISH TO CHANGE (NO COMMAS)"
405 INPUT N$
410 FOR I=1 TO P1 :IF N$=N$(I) GOTO430
415 NEXT
420 PRINT"NAME NOT IN FILE" :GOTO460
430 PRINT"ENTER THE CORRECTED INFO. : NAME, ADDRESS, PHONE"
440 INPUT N$(I), A$(I), P$(I)
450 PRINT"THE LINE NOW READS :" :PRINT N$(I), A$(I), P$(I)
460 INPUT"FOR ANOTHER CORRECTION TYPE 1, OTHERWISE 0"; X
470 IF X=1 GOTO400
490 GOTO20
500 CLS :INPUT "MAKE PREPARATIONS FOR CASSETTE, WHEN READY NIT ENTER"; X
510 PRINT"COPYING..."
520 PRINT #-1, P1
530 FOR I=1 TO P1 :PRINT #-1, N$(I), A$(I), P$(I) :NEXT
540 PRINT"COMPLETE — NOTE TAPE LOCATION"
550 INPUT"TO SEE THE MENU, NIT ENTER"; X :GOTO20
600 CLS :INPUT"WHEN READY, NIT ENTER"; X
610 PRINT"INPUTING...
620 INPUT #-1, P1
630 FOR I=1 TO P1 :INPUT #-1, N$(I), A$(I), P$(I) :NEXT
640 PRINT"COMPLETE":INPUT"TO SEE MENU, HIT ENTER"; X :GOTO20
```

Triangle Computation with Graphics

This program illustrates the use of math functions as well as graphics. It's a great way to investigate the geometry of triangles (might be good for high-school students). (Note: up arrow ↑ = [in this printout.)

```
10 CLS
100 PRINT"THIS PROGRAM CALCULATES THE AREA OF A TRIANGLE
110 PRINT"GIVEN 3 PARAMETERS AND DRAWS THE TRIANGLE TO SCALE
120 PRINT PRINT"FOR 3 SIDES TYPE  SSS, FOR 2 SIDES AND 1 ANGLE TYPE  SAS,
130 PRINT"FOR 1 SIDE AND 2 ANGLES TYPE  ASA
140 INPUT A$  IF A$="SAS" GOSUB300
150 IF A$="ASA" GOSUB400
200 'SSS
210 PRINT"ENTER 3 SIDES. (LONGEST SIDE FIRST).
220 INPUT L1, L2, L3
225 IF L2>L1 OR L3>L1 PRINT "* * * LONGEST FIRST PLEASE    "   PRINT   GOTO 210
230 S=(L1+L2+L3)/2
235 IF S <= L1 PRINT " * * * NOT A TRIANGLE * * *"   PRINT : GOTO 210
240 Y3 = 2 * SQR( S * (S-L2) * (S-L1) * (S-L3)) / L1
250 A = Y3/L2 :A = ATN( A / SQR(-A * A+1))
260 X3 = COS(A) * L2
270 AR = (L1 * Y3) /2
280 GOTO500
300 'SAS
310 PRINT"ENTER 2 SIDES AND 1 ANGLE: AB, AC, THETA (LARGEST SIDE FIRST)
320 INPUT L1, L2, T
325 T = (T * 3 14159) / 180
330 Y3 = L2 * SIN(T)
340 X3 = COS(T) * L2
350 AR = (L1 * Y3) /2
360 GOTO500
400 'ASA
410 PRINT"ENTER 2 ANGLES AND 1 SIDE  THETA1, THETA2, AB"
420 INPUT T1, T2, L2
425 T1 = (T1 * 3 14159) / 180 : T2 = (T2 * 3 14159) / 180
430 Y3 = L2 * SIN(T1)
440 B1 = COS(T1) * L2
450 B2 = Y3 / TAN(T2)
460 L1 = B1 + B2  X3 = B1
470 AR = (L2 * Y3) / 2
500 CLS   F=1   IF L1>50 OR Y3>10 OR L2>50 THEN GOSUB700
510 VC = (3 14159 * ( L1 * F - X3 * F)  * (Y3 * F) [2) / 3
520 VS = (3 14159 * (X3 * F) * (Y3 * F) [2) / 3   VT = VC + VS
525 IF F=6 GOTO610
530 S1=Y3 / X3   S2=Y3 / (X3 - L1)
532 IF INT(X3) = 0 THEN1100
533 IF INT(X3) = INT(L1) THEN1000
534 IF X3<0 THEN1209
535 IF X3>L1 THEN1109
537 IF X3=L2 THEN1000
540 FOR X=20 TO L1 * 2+20 STEP 2   SET (X , Y3+5)   NEXT
550 FOR X=0 TO X3   SET (X * 2+20 , S1 *  (3-X) +5)   NEXT
560 FOR X=X3 TO L1   SET (X * 2+20 , Y3+ (S2 * (L1-X) +5))   NEXT
590 PRINT@ 64 * INT((Y3+5) / 3) + 69, "A (0,0)" , TAB(L1) , "B(", L1 * F , ",0)"
```

```
600 PRINT@ (X3 + 20) / 2, "C ("; X3 * F ;","; Y3 * F ;")" ;
610 PRINT@ 832, "AREA ="; AR ;" SQ. UNITS" ;
620 PRINT@ 896, "THE VOLUME OF THE SOLID CREATED BY REVOLVING THE TRIANGLE ";
625 PRINT"ABOUT THE X AXIS (LINE AB) =";VT;"CUBIC UNITS";
630 PRINT@ 768, "*" ;: INPUT "TO RUN AGAIN, TYPE 1"; B6 : IF B6=1 THEN10
640 STOP : GOTO10
700 IF L1<100 THEN F=2 : GOTO750
710 IF L1<150 THEN F=3 : GOTO750
720 IF L1<200 THEN F=4 : GOTO750
730 IF L1<250 THEN F=5 : GOTO750
740 PRINT "SORRY, SCALE TOO LARGE TO BE DRAWN" ;: F=6 : GOTO510
750 L1=L1/F : Y1=Y1/F : Y2=Y2/F : Y3=Y3/F : X1=X1/F : X2=X2/F : X3=X3/F
760 RETURN
1000 FOR Y=5 TO Y3+5 : SET(X3 * 2 + 20 , Y) : NEXT : GOTO540
1100 FOR Y=5 TO Y3+5 : SET(20 , Y) : NEXT : GOTO540
1199 IF X3>127 GOSUB700
1200 FOR X=L1 TO X3 : SET(X * 2 + 20 , Y3 + (S2 * (L1-X) +5)) : NEXT : GOTO540
1299 IF X3 < -10 GOSUB700
1300 FOR X=X3 TO 0 : SET(X * 2 + 20 , Y3+ (S1 * (0-X) +5)) : NEXT : GOTO540
```

Target Practice

This program uses the INKEY$ function to simulate one of the popular "video games". Notice how few lines are required. This program could easily be "dressed up" — let the user choose a Fast Target, Slow Target; keep score, print special messages, etc. To change the speed of the target, change line 40 as follows: instead of "RND(10)/10", use "RND(0)*S1". For a slow-moving target, let S1 be small (less than 1); for a faster target, let S1 be greater than 1. S1 should not exceed 1.5 or the target will advance to the next line.

```
1 CLS:PRINT : PRINT CHR$(23) ; "HIT 'Z' KEY TO AIM LEFT."
2 PRINT "HIT '/' KEY TO AIM RIGHT."
3 PRINT "HIT SPACE BAR TO FIRE."
4 FOR I = 1 TO 5000 : NEXT
10 CLS : CA=928 : I=1 : PRINT @ CA, "*" ; : PRINT @ 991,"***";
20 F=0
30 IF I >= 15 PRINT @ 124, "          "; : I=1
40 PRINT @ 64 + I * 4, "     "; : I=I+RND(10)/10 : PRINT@ 64+I*4," --> ";
50 IF F=0 THEN 200
60 RESET(MX,MY) : MX=MX-MD : MY=MY-0 : IF MX<=0 OR MX>=127 THEN20
70 IF MY>2 SET(MX,MY) : GOTO 30
80 IF ABS( I*8-MX)>4 THEN 20
90 FOR J=1 TO 6 : PRINT@ 64+4*I,"****"; : FOR K=1 TO 50 : NEXT
95 PRINT@ 64+4*I, "    " : FOR K=1 TO 50 : NEXT K,J
100 GOTO10
200 Y$=INKEY$
205 IF F=1 STOP
210 IF Y$<>"Z" THEN 250
```

```
220 IF CA < 922 THEN 30
230 PRINT@ CA, "  ", : CA=CA-1 : GOTO 280
250 IF Y$<>"/" THEN 300
260 IF CA>934 THEN 30
270 PRINT@ CA, "  ", : CA=CA+1
280 PRINT@ CA, "*", : GOTO30
300 IF Y$<>" " THEN 30
310 F=1 : MD=928-CA : MY=40 : MX=64-3*MD : SET(MX,MY) : GOTO30
311 END
```

Ready-Aim-Fire (Bouncing Dot Revisited)

Remember the LEVEL I Bouncing Dot program? This program takes that idea and turns it into a game for one or more players by means of the INKEY$ function. The object is to enter the correct 3-digit combination that will cause your missile to destroy the bouncing dot. (The 3-digit number corresponds to the X-axis of the display and therefore should be in the range 001 to 126 – and be sure to enter leading zeros for 1- or 2-digit numbers.)

The Computer always takes the first shot; then it's Player Number 1's turn.

```
5 DIM N$(4)
6 CLS : INPUT "ENTER THE NO. OF PLAYERS"; X1 : PRINT"ENTER"; X1 ;"1ST NAMES :"
7 FOR XI=1 TO X1 : INPUT N$(XI) : NEXT : XI=1
10 CLS
20 FOR M=0 TO 127 : SET(M,0) : SET(M,47) : NEXT
30 FOR M=0 TO 47 : SET(0,M) : SET(127,M) : NEXT
35 FOR X=1 TO 121 STEP 10 : RESET(X,0) : NEXT
40 RANDOM : Y= RND(40) +1 : X= RND(110) +4
50 D=1 : Q=1 : Z=64
60 RESET (Z,Y-D) : RESET (X- Q * 4, 24)
70 SET(Z,Y) : SET(X,24) : GOSUB 500
80 Y=Y+D : X=X+Q
90 IF X=123 OR X=4 THEN GOSUB 700
100 IF Y=47 THEN 120
105 IF Y=0 GOSUB 900
110 IF Y <> -1 OR X <> -1 THEN 60
120 Y= Y- 2 * D : D= -D : GOTO 60
500 IF X=Z OR X=Q+Z OR X=2 * Q+Z OR X=3 * Q+Z OR X=Q * 4+Z THEN IF Y=24 GOSUB 600
510 IF Y=23 OR Y=24 OR Y=25 THEN IF X=Z  GOSUB 600
520 RETURN
600 X=1
610 FOR Z=1 TO 50 : PRINT@ 550,"HIT !!!"; : NEXT
620 FOR Z=1 TO 25 : PRINT@ 550,"       " ; : NEXT
630 X=X+1 : IF X<5 GOTO 610
640 GOTO 2000
700 X=X-2 * Q : Q= -Q : RETURN
900 T$ = INKEY$ : A$ = "" : B$ = "" : C$ = ""
1000 A$= INKEY$ : IF LEN(A$) = 0  THEN 1000
1005 PRINT@ 0, A$;
```

```
1010 B$= INKEY$ : IF LEN(B$)=0 THEN 1010
1015 PRINT@ 1, B$;
1020 C$= INKEY$ : IF LEN(C$)=0 THEN 1020
1025 PRINT@ 2, C$;
1030 RESET(Z,1) : X$= A$+B$+C$ : Z=VAL(X$) : IF Z>126 GOTO 1100
1033 PX=PX+1
1035 GOTO120
1040 RETURN
1100 FOR X=1 TO 50 : PRINT@ 70, "TOO LARGE, TRY AGAIN" : NEXT
1110 PRINT@ 70, "                    " : Z=1 : GOTO 1000
2009 IF PX=0 GOSUB 3000
2010 CLS : PRINT "                    * * * "; N$(XI) ;" * * *" : PRINT : PRINT
2017 PX(XI) = PX+PX(XI) : PH(XI) = PH(XI)+1
2020 PRINT, "SHOTS         HITS           PERCENTAGE"
2030 PRINT : PRINT "THIS ROUND  "; TAB(17) PX; TAB(28)"1"; TAB(42) (1/PX) * 100
2035 IFPX(1)=0THENPX(1)=1
2040 PRINT : PRINT "TOTAL       "; TAB(17) PX(XI);
2042 PRINT TAB(28) PH(XI); TAB(42) (PH(XI) / PX(XI)) * 100
2045 FOR X=1 TO 2500 : NEXT
2050 XI=XI+1
2060 IF XI>X1  THEN XI=1
2065 PX=0
2070 GOTO10
2115 IF PX=0 GOSUB 3000
3000 PRINT@ 0, "WHAT LUCK !!!" : PX=1 : RETURN
```

Things You Should Know – LEVEL II TRS-80

1. After executing an INPUT#-*n* (input from cassette), some TRS-80's will not READ properly from DATA statements. Instead a RESTORE will automatically be performed before each READ, so that only first DATA item will be read.

   If your TRS-80 operates this way (depends on a few IC's from one supplier), there is a simple fix. Insert the statement,

   POKE 16553,255

   immediately after every INPUT#-*n* statement.

2. A PRINT#-*n* statement can put no more than 248 bytes on the tape. If you have a lengthy PRINT# list, only the first 248 bytes will be saved on tape; the rest will be lost. Therefore you should break up such lists into two or more PRINT# statements.

3. If you have an Expansion Interface connected and you need to Reset the Computer, hold down the BREAK key and press Reset. This will return you to the MEMORY SIZE question. Any BASIC program in memory will be lost by this Reset sequence.

4. If you stop a BASIC program during execution, and then alter the program itself, all variables will be reset to zero. You will not be able to continue execution where you left off. RUN it again. Note: If a syntax error is encountered and BASIC puts you in the Edit mode, type Q to return to the Command mode. You can then examine variable values, if you wish, before fixing the syntax error.

5. If you attempt to execute an LPRINT or an LLIST when a line printer is not connected (or is turned off), the computer will "freeze up". Either turn on the line printer, or, if one is not connected, Reset the Computer (see 3 above).

6. All the built-in mathematical functions in LEVEL II BASIC return single-precision results (6-7 digits of accuracy). Trig functions use or return radians, not degrees. A radian-degree conversion is given in the LEVEL II Reference Manual.

7. Hard-to-find program errors:

Shift characters are not always interchangeable with their unshifted counterparts. For example, PRINT@ will not work if you use a shifted @, even though it will look ok on the screen. If you can't find anything wrong with a line which causes a syntax error message, try retyping the line, watching out for the shift key.

Spaces are sometimes important in LEVEL II BASIC. The following line is incorrect:

```
IFD<0D=0
```

because 0D is interpreted to mean "double-precision zero".

Change it to:

```
IFD<0 THEN D=0
```

8. To use the CLOAD? with cassette #2, use this format:

```
CLOAD#-2,?"filename"
```

9. If you frequently get "double-entries" when pressing a particular key, remove the plastic key cap, and carefully clean the contacts, using a stiff piece of paper. Insert the paper between the contacts, press the key down to pinch the paper, and pull the paper out while the contacts are pinching it.

10. If you have other questions regarding operation of your TRS-80, call Customer Service, (817) 390-3583, or write:

    TRS-80 Customer Service
    Radio Shack
    P. O. Box 185
    Fort Worth, TX 76102

11. The maximum TAB for an LPRINT statement in 63. The Line Printer won't tab past column 63. There's a simple way around this limitation, using the STRING$ function to simulate tabs past column 63.

Example:

```
LPRINT TAB(5)"NAME"TAB(30)"ADDRESS"STRING$(63,32)"BALANCE"
```

will print "NAME" at column 5, "ADDRESS" at column 30, and "BALANCE" at column 100.