ATARI® 800™

ATARI MICROSOFT BASIC
INSTRUCTION MANUAL

Model CX8126
Use With
ATARI® 800™

ATARI® Microsoft BASIC Instructions

ATARI MICROSOFT BASIC INSTRUCTION MANUAL

ATARI®

A Warner Communications Company

PRINTED IN U.S.A.

PREFACE

In this manual you will find all the commands and statements used by **ATARI® Microsoft BASIC**. The INSTRUCTION list on the inside front cover is in alphabetical order with page numbers for your convenience.

BASIC was developed at Dartmouth College by John Kemeny and Thomas Kurtz. It was designed to be an easy computer language to learn and use. Many additions in recent years have made BASIC a complete and useful language for skilled programmers.

This reference manual does not teach BASIC. Those who wish to learn BASIC should read an introductory book. Helpful books are: *Computer Programming in BASIC for Everyone* by Dwyer and Kaufman, and *Basic BASIC* by James S. Coan.

CONTENTS

APPENDICES

ILLUSTRATIONS

TABLES

LOADING INSTRUCTIONS

Important: The disk-based release of **ATARI® Microsoft BASIC** requires that **all cartridges** (ATARI BASIC, Assembler Editor, games, and the like) **be removed from the front cartridge slots of your computer. You will need a blank diskette in addition to the ATARI Microsoft BASIC diskette on which to store programs.**

Warning: The ATARI Microsoft BASIC diskette is write-protected. Do not attempt to punch a notch in the corner in order to write on it. Attempting to make a read/write diskette out of your ATARI Microsoft BASIC diskette could destroy BASIC and void all warranties.

Use the following setup procedure to load ATARI Microsoft BASIC, format a blank diskette, write DOS files, create MEM.SAV, and transfer CIOUSR and DIR files (see Quick-Reference Guide for a list of timesaving steps).

1. Connect the **ATARI 800 Home Computer** to a television set and to a wall outlet as instructed in the operators manual.

   Note: ATARI Microsoft BASIC requires a minimum of 32K of RAM.
2. Connect the **ATARI 810™ Disk Drive** to the ATARI 800 Home Computer and to a wall outlet as instructed in the *ATARI 810 Disk Drive Operators Manual.*
3. Turn on your television set.
4. Turn the POWER (PWR) switch to ON for Disk Drive 1. Disk drive numbers are set by switches located in the back of your disk drive. Consult your *ATARI 810 Disk Drive Operators Manual* for drive numbers. Turn the POWER (PWR) switch to ON for any other disk drives you wish to use. Two red lights (the BUSY light and the PWR ON light) will come on.
5. When the BUSY light goes out on Disk Drive 1, open the drive door by pressing the door handle release lever.
6. Hold the ATARI Microsoft BASIC diskette with the label in the lower right corner and the arrow pointing towards the disk drive. Insert the diskette into the disk drive and close the disk drive door.
7. Switch the computer console POWER (PWR) to ON. ATARI Microsoft BASIC will load into the computer's memory automatically.
8. Type **DOS** RETURN. The Disk Operating System II version 2.0S will load into your computer's memory.
9. Remove your ATARI Microsoft BASIC Diskette from the disk drive and insert a blank diskette (CX8202).
10. Use the **I** DOS option to format the blank diskette.
11. Use the **H** DOS option to write DOS files onto the diskette.

12. Use the **N** DOS option to create MEM.SAVE. The MEM.SAV file is used to save the ATARI Microsoft BASIC program in memory when you use the DOS command. See the *ATARI Disk Operating System II Reference Manual* for more information on MEM.SAVE.

13. If you have two disk drives you can use the **C** DOS option to copy files from the ATARI Microsoft BASIC diskette. If you have one disk drive you must use the **O** DOS option.

    Copying files with two disk drives:

    - Put ATARI Microsoft BASIC in Drive 2.
    - Put formatted diskette in Drive 1.
    - Type **C** RETURN.
    - Respond to COPY—FROM, TO? by typing **D2:*.*,D1:*.*** RETURN.
    - Turn off the computer and reload ATARI Microsoft BASIC. MEM.SAV is now at work.

    Copying files with one disk drive:

    - Put ATARI Microsoft BASIC in disk drive.
    - Type **O** RETURN.
    - Respond to NAME OF FILE TO MOVE?
    - Press RETURN since source disk is in place.
    - Insert blank as DESTINATION DISK and press RETURN.
    - Repeat the **O** procedure with the file DIR.
    - Turn off computer and reload. ATARI Microsoft BASIC. MEM.SAV is now at work.

14. Remove your newly created program storage diskette and insert the ATARI Microsoft BASIC diskette. Turn your computer console off and then back on again to reload and reinitialize BASIC. To activate the MEM.SAV file you must remove BASIC and insert a program storage diskette. Put your program storage diskette back into the disk drive and press SYSTEM RESET. By pressing SYSTEM RESET with your program storage diskette in the disk drive, the MEM.SAV diskette file will save the correct return locations for future returns to BASIC.

15. If you wish to have duplicate program storage diskettes, now is the time to make them since you have not yet stored any programs. Use DOS option **I** to format the duplicate storage diskette. Then use the **H** option to write DOS files. Now use the **J** option to duplicate the program storage diskette.

You should now remove the ATARI Microsoft BASIC diskette and hereafter use the new program storage diskette(s) you have created. With a program diskette you can save and load the programs you write, and return to BASIC.

Pressing SYSTEM RESET with a program storage diskette in the disk drive brings you back to BASIC with a "warmstart," which means that the variables and your program will be just as you left it before you typed DOS RETURN.

QUICK-REFERENCE GUIDE

1. Boot* system with ATARI Microsoft BASIC Master Diskette.
2. Type **DOS** RETURN.
3. Remove BASIC Master Diskette.
4. Format blank diskette. (DOS 2.0S)
5. Write DOS files to the new diskette.
6. Create MEM.SAV on the diskette.
7. Copy from BASIC Master Diskette to your new diskette, CIOUSR and DIR.
8. Turn off your system and reboot* with ATARI Microsoft BASIC.
9. Insert newly created diskette into Drive 1.
10. Type **DOS** RETURN.
11. After DUP file is loaded, press SYSTEM RESET.
12. Use your newly created program storage diskette to make duplicate program storage diskettes (DOS option **J**).

Note: Steps 10, 11, and 12 write the correct Microsoft memory images into the MEM.SAV files on your Microsoft BASIC program storage diskette.

*BASIC loads into RAM automatically (boots) when you turn on the computer.

1 MICROSOFT OVERVIEW

ATARI® Microsoft BASIC is a customized and enhanced BASIC programming language. It was developed by Microsoft for the **ATARI 800™ Home Computer**, which uses the 6502 microprocessor and customized graphics and sound-integrated circuits.

In the development of ATARI Microsoft BASIC, the two primary considerations were processing speed and compatibility with other microcomputer BASIC languages. The fast ATARI 800 Computer clock rate of 1.8 MHz combines with the state-of-the-art Microsoft design to give high microprocessor throughput speed. ATARI Microsoft BASIC is a superset of the existing microcomputer languages. That is, ATARI Microsoft BASIC combines the capabilities of other microcomputer BASIC languages with some unique features. New graphics features have been added to take advantage of the hardware-supported player-missile graphics. Sound capabilities now include the ability to set the length of time a sound is heard. You can renumber and merge programs easily with Microsoft BASIC. This is a powerful language with software tools to fit a variety of needs.

WHAT IS A PROGRAM?

A program is a list of steps (statements) that you wish the computer to perform. Every statement stored in memory must have a line number. The lowest line number is 0 and the highest allowable line number is 63999. Statements are performed in line number order starting with the lowest numbered line. You can change the order in which the statements are performed by branching or jumping to other line numbers.

Line numbers always precede statements that you want stored in memory. Because the statements that have line numbers wait in memory until the command RUN is given, they are written in what is called the deferred mode.

To be exact, execution of a program waits until you type the word **RUN** and press the RETURN key. When ATARI Microsoft BASIC is first loaded, it is ready for you to write programs (deferred mode) or execute statements immediately (direct mode).

When the computer is ready to accept input, a prompt >appears on your television screen. When you see the >, you can enter statements with line numbers (deferred mode) or statements without line numbers for immediate execution.

Let's write a BASIC program in the deferred mode:

```
  >
  100 PRINT 7 * 7

RUN RETURN
  49
```

This single-line program does not execute immediately. The program waits to perform the statement until you type **RUN** and press RETURN. The word **RUN** typed without a line number, executes the program immediately after you press the RETURN key.

KEYWORDS

Keywords must be spelled out. Abbreviations are not legal syntax in ATARI Microsoft BASIC.

Keywords are words the computer recognizes. Each keyword tells the computer what you want done. The words IF, GOSUB, INPUT, and GOTO are keywords. Keywords can be thought of as the verbs in the vocabulary of your computer. If you write a statement that uses a keyword the computer does not recognize, BASIC will give you an ERROR statement when you run the program. ATARI Microsoft BASIC does not allow you to use keywords as variables, but does allow you to embed keywords in the variable names. That is, IF and GOSUB cannot be variables, but LIFE and RGOSUB are allowed. A complete list of keywords is given in Appendix L.

LINE CONSTRUCTION

The form of the BASIC statement looks like this:

Line
Number Statement

```
100 IF A<>B THEN 630 ELSE 210
```

Just as there are punctuation marks in the English language, so there are quotes, commas, semicolons, and colons in BASIC. The rules of punctuation are listed in this manual with the keywords that require them or have them as options. Following is a summary of punctuation use.

QUOTATION MARKS

The quotation marks are used to indicate where typed characters begin and end. Just as we use quotes in English to mark the beginning and end of a speaker's words, so it is with BASIC. The quote mark means that the material quoted constitutes a string variable or string constant; strings will be covered later in the text. For now it is enough to know that quotes tell the computer where to begin and end a string. The string in this example program will be told when to start and stop printing on the screen by quotes:

Example Program:

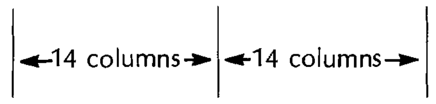
```
100 PRINT "START PRINTING ON SCREEN— — — — — —NOW STOP"

RUN RETURN

START PRINTING ON SCREEN— — — — — —NOW STOP
```

THE COMMA

The comma has three uses.

- Use the comma to separate required items after a keyword. The keyword SOUND has five different functions in ATARI Microsoft BASIC. Each parameter is separated by commas. For example, SOUND 2,&79,10,8,60 means voice 2, pitch hexadecimal 79 (middle C), noise 10, volume 8, and duration in jiffies (1/60 of a second) 60. Another example of the comma is the statement SETCOLOR 4,4,10 which means register 4, pink, bright luminance. The comma tells where one piece of information ends and the next begins. BASIC expects to find an exact order separated by commas.

- Use the comma to separate optional values and variable names. You can input any number of variable names on a single line with an INPUT statement. The variable names are of your own invention. You can have as many of them as you like as long as you separate them with a comma. For example, INPUT A,B,C,D,E tells the computer to expect five values from the keyboard.
- Use the comma to space advance to the next output field in a PRINT statement. When used in a PRINT statement at the end of a quoted string or between expressions, the comma will advance printing to the next column which is a multiple of 14. For example, if X is assigned the value of 25 then the statement 10 PRINT "YOU ARE", X, "YEARS OLD" will have the following spacing when you run it:



```
YOU ARE       25
YEARS OLD
```

USE OF SEMICOLON IN PRINT STATEMENT

The semicolon is used for PRINT statement output. The semicolon leaves one space after variables and constants separated by semicolons. A positive number printed with semicolons will have a leading blank space. Negative numbers will have a minus sign and no preceding blank space. For example, if X is assigned the value of 25, then the statement 10 PRINT "YOU ARE";X;"YEARS OLD" will have the following spacing when the program is run:

```
YOU ARE 25 YEARS OLD
```

If X is assigned the value of -25, then the statement 10 PRINT "YOU ARE";X;"YEARS OLD" will have the following spacing when the program is run.

```
YOU ARE-25 YEARS OLD
```

If you want more than one space left before and after the 25 you must leave the space in the string within the quotes. Thus,

```
10 PRINT "YOU ARE   ";25;"   YEARS OLD"
```

will give the following spacing when the program is run:

```
YOU ARE   25   YEARS OLD
```

The semicolon can also be used to bring two PRINT statements, string constants, or variables together on the same line of the television screen. For example:

```
100 PRINT "THE AMOUNT IS $";
120 AMOUNT=20
125 REM BOTH STRING CONSTANT AND VARIABLE
126 REM WILL PRINT ON THE SAME LINE
130 PRINT AMOUNT
```

THE COLON

The colon is used to join more than one statement on a line with a single line number. Thus, many statements can execute under the same line number. By joining more than one statement on a single line, the program requires less memory.

For example:

```
10 X=5:Y=3:Z=X+Y:PRINT Z:END
```

Many times this also helps the programmer organize the program steps. The same program with line numbers instead of colons uses more bytes of memory:

```
10 X=5
20 Y=3
30 Z=X+Y
40 PRINT Z
50 END
```

2 EDITING

KEYBOARD OPERATION

The ATARI 800 Computer keyboard has features that differ from those of an ordinary typewriter. To print lowercase letters on your television screen, press the CAPS LOWR key. The keyboard will now operate like a typewriter, with the SHIFT key giving uppercase letters. Since most BASIC programs are written in uppercase, you will normally want to return to the uppercase mode. Press the SHIFT key and hold it down while you press the CAPS LOWR key to return to uppercase letters.

SPECIAL FUNCTION KEYS

Inverse (Reverse) Video Key or ATARI logo key. Press this key to reverse the text on the screen (dark text on light background). Press key a second time to return to normal text.

CAPS LOWR

Lowercase Key. Press this key to shift the screen characters from uppercase (capitals) to lowercase. To restore the characters to uppercase , press the SHIFT key and the CAPS LOWR key simultaneously.

ESC

Escape Key. Press this key to enter a command to be entered into a program for later execution.

Example: To clear the screen, enter:

```
10 PRINT "ESC CTRL CLEAR"
```

and press RETURN. Then, whenever line 10 is executed the screen will be cleared.

ESC is also used in conjunction with other keys to print special graphics control characters. See the graphics in Appendix K for specific keys and their screen-character representations.

BREAK

Break Key. Press this key to stop your program. You may resume execution by typing **CONT** and pressing RETURN.

SYSTEM RESET

System Reset Key. This key is similar to BREAK in that it also stops program execution. Use this key to return the screen display to graphics mode 0, and to clear the screen.

SET CLR TAB

Tab Key. Press SHIFT and the SET CLR TAB keys simultaneously to set a tab. To clear a tab, press the CTRL and SET CLR TAB keys simultaneously. Used alone, SET CLR TAB advances the cursor to the next tab position. In deferred mode, set and clear tabs by adding a line number, the command PRINT, and a quotation mark, and pressing the ESC key.

Examples:

```
100 PRINT "ESC SHIFT SET CLR TAB
200 PRINT "ESC CTRL SET CLR TAB
```

If tabs are not set, they default to columns 7, 15, 23, 31, and 39.

INSERT

Insert Key. Press the SHIFT and INSERT keys simultaneously to insert a line. To insert a single character, press the CTRL and INSERT keys simultaneously.

CURSOR CONTROL KEYS

In addition to the special function keys, there are cursor control keys that allow immediate editing capabilities. These keys are used in conjunction with the SHIFT or CTRL keys. The keys that offer special editing features are described in the following paragraphs.

CTRL	Hold the control key down while pressing the arrow keys to produce the cursor control functions that allow you to move the cursor anywhere on the screen without changing any characters already on the screen. Other key combinations set and clear tabs, halt and restart program lists, and control the graphics symbols. Striking a key while pressing the CTRL key will produce the upper left symbol on those keys that have three functions.
CTRL ↑	Moves cursor up one line without changing the program or display.
CTRL →	Moves cursor one space to the right without disturbing the program or display.
CTRL ↓	Moves cursor down one line without changing the program or display.
CTRL ←	Moves cursor one space to the left without disturbing the program or display.
CTRL INSERT	Inserts one character space.
CTRL DELETE BACK S	Deletes one character or space.
CTRL **1.**	Temporarily stops and restarts screen display. You can use CTRL **1** while listing a program or while running a program.
CTRL **2.**	Rings buzzer.

Hold the SHIFT key down while pressing the numeric keys to display the symbols shown on the upper half of those keys.

SHIFT INSERT	Inserts one line.
SHIFT DELETE BACK S	Deletes one line.
SHIFT CAPS LOWR	Returns screen display to uppercase alphabetic characters.
BREAK	Stops program execution or program list, prints a > on the screen, and displays the cursor (█) underneath.

3 CONSTANTS, VARIABLES, AND NAMES

There are five types of constants in Microsoft BASIC: single-precision real, double-precision real, integer, string, and hexadecimal.

FORMING A VARIABLE NAME

In ATARI Microsoft BASIC a variable name can be up to 127 characters long. The allowable characters include the alphabet ABCDEFGHIJKLMNOPQRSTUVWXYZ, numbers 1234567890, and underscore (_). The underscore character (_) is a legal character in ATARI Microsoft BASIC. Numbers are allowed in variable names as long as the variable name starts with an alphabetic character. The variable name X9 is allowed, while 9X is not allowed.

SPECIFYING PRECISION OF NUMERIC VARIABLES

After you create a variable name, you can specify the precision of the variable in one of two ways. The variable name itself can have a variable-type identifier (none, #, %, $) as the last character or you can predefine the starting letter as a variable type using DEFSNG, DEFDBL, DEFINT, or DEFSTR.

PREDEFINING VARIABLE PRECISION

The advantage of predefining the variable type is that you can change all the variables from one type to another without having to go through your program changing all variable names. Changing DEFINT A to DEFDBL A, for example, changes all variables beginning with the letter A from integer type to double-precision type. Your other option is to use a type tag identifier: # (double precision), % (integer), and $ (string). Tag identifiers are attached to the end of the variable name itself. If variables should have both DEF identification of type and a tag identifier (#, %, $), the tag identifier has precedence.

Although DEFSNG, DEFDBL, DEFINT, and DEFSTR can be placed anywhere in a program, they are usually placed near the beginning. In all cases the DEF statement must precede the variable whose type it defines.

SINGLE-PRECISION REAL CONSTANTS

Examples: 65E12, 333335, .45E8, .33E-6

If you do not otherwise specify a constant (and it is outside the range -32768 to 32767), it is single-precision real.

SINGLE-PRECISION REAL VARIABLES

Examples: AMT, LENGTH, BUFFER

If you do not declare the precision of a variable, it becomes single-precision real by default. Numbers stored as single precision have an accuracy of six significant figures. The exponential range is -38 to +38.

DEFSNG

Format: DEFSNG letter,|beginning_letter-ending_letter|
Examples: 100 DEFSNG K, S, A-F
120 DEFSNG Y

Variable names beginning with the first letters identified in DEFSNG will be single-precision real variables. In DEFSNG K, S, A-F, the letter range A-F means ABCDEF will be single precision. Variable names starting with K and S will also be single precision in this example. Single letters and ranges of letters must be separated by commas.

Example Program:

```
10 DEFSNG A-F
20 COUNTER=COUNTER+1
30 PRINT COUNTER
40 GOTO 20
```

In the DEFSNG example program, all variable names beginning with the letter C will be single precision. Thus, COUNTER is single precision in this example because it starts with C. If counter were COUNTER# (# means double precision), it would have double precision even though it is defined as single precision. Keep in mind that the tag identifier in a variable name takes precedence.

Figure 3-1 illustrates how single-precision real numbers are represented in memory.

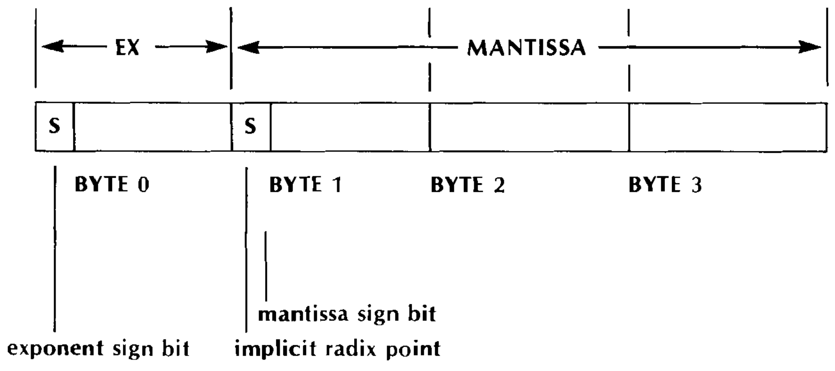


Figure 3-1 Machine Representation of Single-Precision Real

DOUBLE-PRECISION REAL CONSTANTS

Examples: 45D5, 23D-6, 8888888D-11

You can specify double-precision real in the constant by putting the letter D before the exponential part. Double-precision real numbers are stored in 8 bytes. Numbers are accurate to 16 decimal digits.

DOUBLE-PRECISION REAL VARIABLES

Examples: DBL#, X#, LGNO#

The pound sign (#) is the identifier for double-precision real variables. A double-precision real variable has 8 bytes. The exponent and sign are stored in the first byte. The range is the same as single precision -38 to +38. The accuracy is 16 significant figures in double-precision real. The pound sign (#) identifier is placed after the variable name.

DEFDBL

Format: DEFDBL letter,|beginning_letter-ending_letter|
Examples: 10 DEFDBL C-E, Z
20 DEFDBL R

Variable names starting with letters identified by the DEFDBL statement are double-precision real. In the example above CDE, Z, and R are all declared as double-precision. The variable name E1 would be a double-precision variable because the variable name begins with E.

Figure 3-2 illustrates how double-precision real numbers are represented in memory.

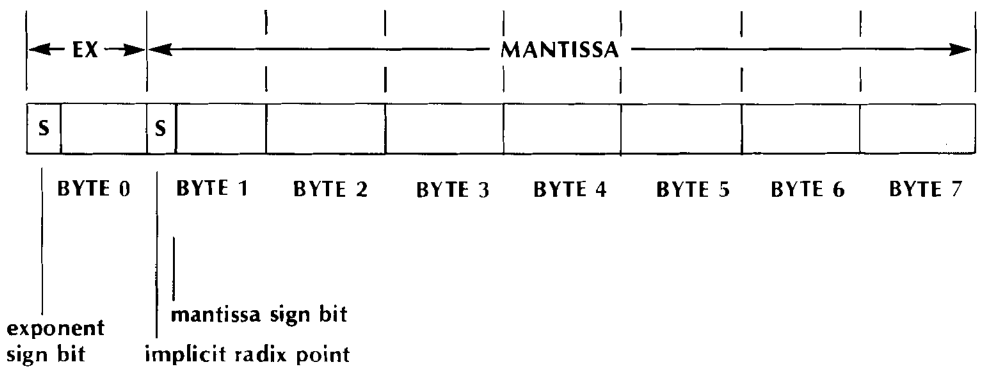


Figure 3-2 Machine Representation of Double-Precision Variable

INTEGER CONSTANTS

Examples: 23, -9999, 709, 32000

All numbers in ATARI Microsoft BASIC within the range -32768 to 32767 are stored as two bytes of binary. If an integer constant is multiplied with a single-precision real number, the product of the multiplication will be a single-precision real number. The results of mathematical operations are always stored in the higher level precision type.

INTEGER VARIABLES

Examples: SMALLNO%, J%, COUNT%

An integer can be identified by having a percent sign (%) as the last character in the variable name. An example of an integer identified by name is NO%. The 16-bit integer is stored as twos complement binary.

DEFINT

Format: DEFINT letter,|beginning_letter-ending_letter|
Examples: 10 DEFINT N, J, K-M
20 DEFINT I

The starting letters of variable names identified by the DEFINT statement are integers. Integer variables increase the speed of processing but can only accurately hold values between -32768 and +32767. Remember that tag identifiers have precedence. Even though N is defined by DEFINT as being an integer type, the pound sign appearing after the N identifies it as double precision. N#, N1#, NUMB# are all double precision.

Figure 3-3 illustrates how integers are represented in memory.

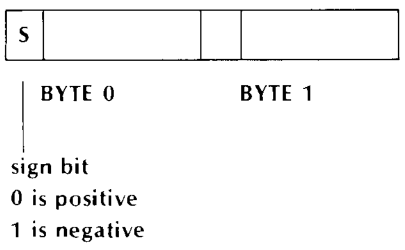


Figure 3-3 Machine Representation of Integer Variable

Negative integers are stored as twos complement binary.

STRING CONSTANTS

Examples: "AMOUNTS", "FILL IN NAME____________"

String constants are always enclosed in quotes. The string constant can be any length up to the maximum line length (127). Strings are composed of ANY keyboard characters: "!#$%&&'()00KJHGGFDS." A double-quote character ("") is also allowed. The double quote ("") will give you a single quote when the string is printed.

Example of a string constant used in a print statement:

```
10 PRINT "Strings and %&'$ ""things"""
20 A$="STRING CONSTANTS ASSIGNED TO VARIABLE NAME"
30 PRINT A$
```

STRING VARIABLES

Examples: A$, NINT$, ADDRESS$

String-variable names end with a dollar sign **$**. A string variable can be assigned a string up to 255 characters. The double-quote ("") character is a legal ATARI Microsoft BASIC way of getting a single quote (") within a string.

Examples of strings assigned to A$:

```
10 A$="a string"
20 A$="another ""string"""
```

DEFSTR

Format: DEFSTR letter,|beginning_letter-ending_letter|
Examples: 10 DEFSTR A, K-M, Z
20 DEFSTR F, J, I, O

A variable name can be defined as a string by declaring its starting letter in the DEFSTR statement. Strings can be up to the length of 255 characters. As in all variable name declarations, the tag identifier has precedence. A# or A% are their tag types even if their first letter is defined by DEFSTR.

Example Program:

```
10 DEFSTR A, M, Z
20 A="Employee Name      AMOUNT"
30 PRINT A
```

The example program will print the heading *Employee Name AMOUNT.*

HEXADECIMAL CONSTANTS

Examples: &76, &F3, &7B, &F3EB

It is often easier to specify locations and machine language code in hexadecimal (base 16) rather than decimal notation. By preceding a number with **&**, you declare it to be hexadecimal.

To jump to the machine language routine starting at hexadecimal location C305, you specify A=USR(&C305,0). A=PEEK (&5A61) will assign the contents of memory location 5A61 hex to the variable named A. Hexadecimal is useful in representing screen graphics—especially player-missile graphics.

Following is an equivalency table for decimal, hexadecimal, and binary numbers.

TABLE 3-1
DECIMAL, HEXADECIMAL, AND BINARY EQUIVALENTS

Decimal	Hexadecimal	Binary
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

4 NUMERIC AND STRING EXPRESSIONS

NUMERIC EXPRESSIONS

RELATIONAL OPERATORS

There is no real order of precedence for the relational operators =, < >, >, < =, > =. They are evaluated from left to right.

RELATIONAL AND LOGICAL SYMBOLS

Because the relational symbols are evaluated from left to right, you could say that their order of precedence is from left to right. The relational symbols =, < >, <, >, < =, > = have precedence over the logical operators NOT, AND, OR, and XOR. NOT has the highest precedence, AND ranks next, OR ranks next, and XOR ranks last.

The relational operators are combined to form expressions. For example: A>B AND C<D is an expression. The greater than (>) and less than (<) symbols are considered first, then the AND is evaluated. If the relationship is true, a nonzero number will result. If the relationship is not true, then zero will be the result. Nonzero is true and zero is false. In an IF statement this evaluation determines what happens next. The ELSE or the next line number is taken when an the expression formed with operators is false.

OPERATOR	**MEANING**
=	Equals. This is a true use of the equal sign. It asks if A= B. The B is not assigned to A.
< > or > <	Not Equal. Evaluates whether two expressions are not equal.
<	Is less than. A is less than B is represented by A<B.
>	Greater than. A is greater than B is represented by A>B.
> = or = >	Greater than or equal to. A is greater than or equal to B is represented by A> =B.
< = or = <	Less than or equal to. A is less than or equal to B is represented by < =.

ARITHMETIC SYMBOLS

The arithmetic symbols are: (), =, -, Λ, *, /, +, - (the first dash - means negation, the last dash means subtraction). The arithmetic symbols can be mixed with the logical operators in creating expressions. The expression A/C > D*A is legal. The arithmetic expressions represent mathematical symbols. The * symbol represents multiplication. The Λ is used in ATARI Microsoft BASIC to mean exponent. The order of precedence is:

SYMBOL	MEANING
()	Arithmetic within parenthesis is evaluated first.
=	Equals sign.
-	Negative number. This is not subtraction but a negative sign in front of a number. Example: -3, -A, -6.
Λ	Exponent.
*	Multiplication.
/	Division.
+	Addition.
-	Subtraction.

STRING EXPRESSIONS

RELATIONAL OPERATORS IN STRINGS

Relational operators in strings (=, <>, <, >, <=, >=) can accomplish useful tasks. Alphabetical order can quickly be achieved by an algorithm using the expression A$<B$. A match between names can be found by asking that A$=B$. The string variables are evaluated as numbers in ATASCII code and since the ATASCII is ordered alphabetically, the evaluation of string expressions is useful.

SYMBOL	MEANING
A$<B$	True (nonzero) if A$ has a lower ATASCII code number than B$.

Sort Example:

```
100 INPUT A$,B$
120 IF A$<B$ THEN 160
130 C$=A$
140 A$=B$
150 B$=C$
160 PRINT A$, B$
170 END
```

To experiment, type any two word combinations and separate them by commas. The words will be sorted into alphabetical order using the example above. Thus, you will see that BILL comes before BILLY, and CAT comes before DOG.

The logical operators have the following order of precedence:

OPERATOR	**MEANING**
NOT	Not. The 8 bits of the number are complemented. If it is a binary 1 it becomes a 0 after this logical operation.
AND	The bits of the number are logically ANDed. Example: A AND B. If A is 1 and B is 1 the result is 1. If A is 1 and B is 0 the result is 0. If A is 0 and B is 1 the result is 0. If A is 0 and B is 0 the result is 0.
OR	The bits of the number are logically ORed. Example: A OR B. If A is 1 and B is 1 the result is 1. If A is 1 and B is 0 the result is 1. If A is 0 and B is 1 the result is 1. If A is 0 and B is 0 the result is 0.
XOR	The bits of the number are logically eXclusive ORed. Example: A XOR B. If A is 1 and B is 1 the result is 0. If A is 1 and B is 0 the result is 1. If A is 0 and B is 1 then the result is 1. If A is 0 and B is 0 then the result is 0.

The logical operators can be used with string (A$) variables. Read Section 10 on string expressions.

5 COMMANDS

In ATARI Microsoft BASIC, statements are not evaluated for syntax errors until you type **RUN** and press the RETURN key.

NEW

Format: NEW
Examples: NEW
100 IF CODE<>642 THEN NEW

NEW clears your program to allow you to enter a new program. The NEW command does not destroy TIME$. All variables are cleared to zero and all strings are nulled when NEW is executed.

RUN

Format: RUN |"device:program__name"| |optional__starting__line__number|
Examples: RUN
RUN 120
200 RUN "D:TEST.BAS"
110 RUN 200

RUN without a line number starts executing your program with the lowest line numbered statement. RUN initializes all numeric variables to zero and nulls string variables before executing the first statement in the program.

RUN can be used in the deferred mode (with a line number). Refer to the program on the next page. It can also be used to enter a program from diskette or cassette. However, when RUN is used to run a program on diskette or cassette (i.e., RUN "D:SHAPES"), it cannot be used with |optional__starting__line__number|, which can only be used to run programs that are already in memory.

Example: 200 RUN "D:TEST

When statement line number 200 is executed, it will run the program called TEST.

RUN can be used to run tokenized (saved with the SAVE instruction) programs only.

RUN can be used to start executing a program at a particular line number.

Example: RUN 250

When RUN is executed in a program, as mentioned earlier, all numeric variables are set to zero and all strings are nulled.

Example Program:

```
100 X=55
110 Y=77
120 A$="A TEST"
130 PRINT X,Y,A$
140 RUN 150
150 PRINT X,Y,A$,"Variables are 0 and String is null"
160 END
```

DOS

Format: DOS
Example: DOS

The DOS command lets you leave BASIC and enter the DOS Menu. This makes available all of the DOS Menu items on programs and data stored on diskette. To return to ATARI Microsoft BASIC, press the SYSTEM RESET key. This method of exiting DOS will keep your program exactly as it was before you entered DOS.

LIST

Format: LIST |"device:program__name"| |m-n|
Examples:
```
100 LIST
150 LIST "C:
120 LIST "P:" 10-40
100 LIST "D:GRAFX.BAS
110 LIST 100-200
100 LIST -300
```

LIST writes program statements currently in memory onto the television screen or another device. If "device:program__name" is present, the program statement currently in memory is written onto the specified device.

Legal device names include: D: (for Disk), C: (for Cassette), P: (for Printer). If you do not follow LIST with a device name, the screen (S:) is assumed.

When you list programs on the screen, it is often convenient to freeze the list while it is scrolling. To freeze a listing, press both the CTRL and **1** key at the same time. To continue the listing, again press CTRL and **1** at the same time.

With the LIST command you can list just one statement or as many as you wish. A - (hyphen) is used to specify the range of statements:

LIST	Lists the whole program from lowest line number to the highest.
LIST n	Lists only the statement n (where n is a statement number).
LIST -m	Listing starts with the first statement in the program and stops listing with statement m. Statement m is listed.
LIST n-	Listing starts with statement number n and continues to the last statement number in the program.
LIST n-m	Listing starts with n and ends with m. Both statements n and m are included in the listing.

Example:

```
100 REM Example of the list
110 REM Command
120 PRINT "SHOWS WHICH STATEMENTS"
130 PRINT "OR GROUP OF STATEMENTS"
140 PRINT "GET LISTED"
```

LIST 110-130

```
110 REM Command
120 PRINT "SHOWS WHICH STATEMENTS"
130 PRINT "OR GROUP OF STATEMENTS"
```

Example of LIST used in deferred mode:

```
10 COUNT=1
20 COUNT=COUNT+1
30 PRINT COUNT
40 IF COUNT <> 30 THEN 20
50 LIST
```

Use LIST to list a program on a printer. This is done in direct mode.

```
LIST"P:
```

Use LIST to list a program in untokenized ASCII form onto a diskette. To list to diskette use:

```
LIST"D:name.ext
```

Use LOAD when you are entering untokenized (listed) programs into your computer. LOAD can be used to enter programs that have been listed or saved to cassette or diskette.

AUTO

Format: AUTO |n,i|
Examples: AUTO 200,20
AUTO

AUTO numbers your lines automatically. If you do not specify n,i (starting number, increment) you will get line numbers starting at 100 with an increment of 10. Use AUTO when you start writing a program. Type **AUTO**, then type a starting line number. (See the example on the following page.) Then type the amount you want the numbers to increase. After you start the AUTO numbering, you will automatically have a new line number every time you type a statement and press RETURN. To stop AUTO, press RETURN by itself without typing a statement. AUTO can also be stopped by pressing the BREAK key.

Example Program:

AUTO 300,20 RETURN	Starts numbering at 300 and increments by 20

```
300 PRINT "THIS SHOWS HOW"
320 PRINT "AUTO NUMBERING"
340 PRINT "WORKS"
360 RETURN
█
```

AUTO numbering ends when you press RETURN right after a line number. If there is an existing line at that number, the line will be displayed on your television screen.

DEL

Format: DEL n-m
Examples: DEL 450 -
DEL 250 - 350
DEL - 250

DEL deletes program statements currently in memory. With the DEL command you can delete just one statement or as many as you wish. A - (hyphen) is used to specify the range of statements:

DEL n	Deletes only the statement n (where n is a statement number).
DEL -m	Deletion starts with the first statement in the program and stops with statement m. Statement m is deleted.
DEL n-	Deletion starts with statement number n and continues to the last statement number in the program.
DEL n-m	Deletion starts with n and ends with m. Both statements n and m are deleted.

Example Program:

```
100 PRINT "AN EXAMPLE OF"
120 PRINT "HOW THE DELETE"
130 PRINT "COMMAND WORKS"
```

DEL 120- RETURN

Only statement 100 is left in memory.

LIST RETURN

```
100 PRINT "AN EXAMPLE OF"
```

If you want to delete a single statement from a program, simply type the statement number and press RETURN.

Example Program:

```
110 FOR X=1 TO 5000:NEXT
```

```
110 RETURN
```

SAVE

Format: SAVE "device:program__name"
Example: SAVE "D:GAME.BAS"

SAVE copies the program in memory onto the file named by *program__name*. Legal devices are D: (for disk), C: (for cassette). For example, the command SAVE "D:TEMP.BAS" will save the program currently in memory onto diskette. The program is recorded in "tokenized" form onto tape or diskette.

Example:

```
SAVE "D:PROGRAM"
```

Saves PROGRAM on diskette.

```
SAVE "C:
```

Saves the program on cassette.

SAVE...LOCK

Format: SAVE "device:program__name" LOCK
Example: SAVE "D:PROGRAM.EXA" LOCK

SAVE "device:program__name" LOCK saves a program onto tape or diskette and encodes it so that it cannot be edited, listed, merged, examined, or modified. LOCK is used to prevent program tampering and theft.

LOAD

Format: LOAD "device:program__name"
Examples: LOAD "D:EXAMPLE"
110 LOAD "C:"

LOAD "device:program__name" replaces the program in memory with the one located on **device:**. Disk drive or cassette can be specified for device:. Use LOAD "C:" to load data or listed cassette files. For programs that have been previously saved use CLOAD to increase loading speed. For diskette files, use "D:program__name" for listed programs or saved programs.

CLOAD

Format: CLOAD
Examples: CLOAD
440 CLOAD

Use CLOAD to load a program from cassette tape into RAM for execution. When you enter **CLOAD** and press RETURN, the in-cabinet buzzer sounds. Position the tape to the beginning of the program, using the tape counter as a guide, and press **PLAY** on the **ATARI 410™ Program Recorder**. Then press the RETURN key again. Specific instructions to CLOAD a program are contained in the *ATARI 410 Program Recorder Operators Manual.*

CSAVE

Format: CSAVE
Examples: CSAVE
330 CSAVE

CSAVE saves a RAM-resident program onto cassette tape. CSAVE saves the tokenized (compacted) version of the program. As you enter **CSAVE** and press RETURN, the in-cabinet buzzer sounds twice signaling you to press **PLAY** and **RECORD** on the Program Recorder. Then press RETURN again. Do not, however, press these buttons until the tape has been positioned. Saving a program with this command is speedier than with SAVE"C:" because short inter-record gaps are used. Use SAVE"C:" with LOAD"C:" or CSAVE with CLOAD but do not mix these paired statements — SAVE"C:" with CLOAD will give you an error message.

VERIFY

Format: VERIFY "device:program__name"
Examples: VERIFY "D:BIO.BAS"
VERIFY "C:

VERIFY compares the program in memory with the one named by "device:program__name". If the two programs are not identical, you get a TYPE MISMATCH ERROR.

MERGE

Format: MERGE "device:program__name"
Examples: MERGE "D:STOCK.BAS"
MERGE "C:

Use MERGE to merge the program stored at "device:program__name" with the program in memory. Only programs that have been saved using the LIST instruction to put them on diskette or cassette can be merged. If duplicate line numbers are encountered, the line on "device:program__name" will replace the one in memory. On the following page, you can see an example of merging programs.

Example Program:

```
100 REM THIS IS A PROGRAM
120 REM STORED ON DISKETTE
130 PRINT "MERGE TEST"
```

LIST "D:STOCK.BAS"

```
110 REM THIS PROGRAM IS
125 REM IN COMPUTER MEMORY
140 PRINT "RESULT"
```

MERGE "D:STOCK.BAS"

LIST

```
100 REM THIS IS A PROGRAM
110 REM THIS PROGRAM IS
120 REM STORED ON DISKETTE
125 REM IN COMPUTER MEMORY
130 PRINT "MERGE TEST"
140 PRINT "RESULT"
```

RENUM

Format: RENUM [m, n, i]
Example: RENUM 10,100,10

m = The line number to be applied to the first renumbered statement.

n = The first line number to be renumbered.

i = The increment between generated line numbers.

RENUM gives new line numbers to specified lines of a program. The line number to be applied to the first renumbered statement is the first parameter. The first line number to be renumbered is the next parameter. The increment or amount of increase between numbers is the last parameter.

The default of RENUM is 10, 0, 10.

Renumber changes all references following GOTO, GOSUB, THEN, ON...GOTO, ON...GOSUB, and ERROR statements to reflect the new line numbers.

Note: RENUM cannot be used to change the order of program lines. For example, RENUM 15, 30 would not be allowed when the program has three lines numbered 10, 20, and 30. Numbers cannot be created higher than 63999.

RENUM	Renumbers the entire program. The first new line number will be 10. Lines will increment by 10.
RENUM 10,100	The old program line number 100 will be renumbered 10. Lines increment by 10 (the default is 10).
RENUM 800,900,20	Renumbers lines from 900 to the end of the program. Line 900 now is 800. The increment is 20.

RENUM 300, 140, 20 gives number 300 to line 140 when it is encountered . The increment is 20.

BEFORE	**AFTER**
100	100
110	110
120	120
130	130
140	300
150	320
160	340
170	360

LOCK

Format: LOCK "device:file__name"
Example: LOCK "D:CHECKBK"

LOCK is the same LOCK that exists in the DOS Menu. LOCK ensures that you do not write over a program without first unlocking it. As a BASIC command, LOCK offers a measure of protection against accidental erasure.

UNLOCK

Format: UNLOCK "device:program__name"
Example: UNLOCK "D:GAME1.BAS"

The UNLOCK statement restores a file so that you can write to, delete, or rename it.

KILL

Format: KILL "device:program__name"
Example: KILL "D:PROG1.BAS"

KILL deletes the named program from a device.

NAME...TO

Format: NAME "device:program__name__1" TO "program__name__2"
Example: NAME "D:BALANCE" TO "CHECKBK"

NAME gives a new name to "device:program__name__1." The device (D1: through D8:) must be given for the old program, but the new program name enclosed in quotes is the only thing following the word TO.

TRON

Format: TRON
Examples: TRON
550 TRON

This command turns on the trace mechanism. When TRON is on, the number of each line encountered is displayed on your television screen before it is executed. Use TRON in direct or deferred mode.

TROFF

Format: TROFF
Example: 770 TROFF

This command turns off the trace mechanism. Use TROFF in direct or deferred mode.

6 STATEMENTS

REM or ! or '

Format: REM
Example:
```
10 REM THIS PROGRAM COMPUTES THE AREA OF A SPHERE
20 LET R=25 !Sets an initial value
30 GOSUB 225 'GO TO COMPUTATION SUBROUTINE
65 PRINT R:REM PRINTS RADIUS
```

Format: ! and '
Example:
```
10 PRINT "EXAMPLE" !TAIL COMMENTS
20 GOTO 10 ! USE ! and '
```

The exclamation point (!) and the accent (') are used after a statement for comments. REM must start right after the line number or colon, while ! and ' do not require a preceding colon.

REM, !, and ' are used to make remarks and comments about a program. REM does not actually execute. Although REM does use RAM memory, it is a valuable aid to reading and documenting a program.

LET

Formats: |LET| variable__name = |arithmetic__expression| or |string__expression|
variable__name = |arithmetic__expression| or |string__expression|
Example:
```
100 LET COUNTER = 55
120 D=598
```

LET assigns a number to a variable name. The equal sign in the LET statement means "assign," not "equal to" in the mathematical sense. For example, LET V=9, assigns a value of 9 to a variable named V. The number on the right side of the equal sign can be an expression composed of many mathematical symbols and variable names. Thus, LET V=(X+Y-9)/(W*Z) is a legal statement.

The word LET is optional in assignment. All that is necessary for assignment is the equal sign. Thus,

```
100 LET THIS = NUMBER * 5
```

is the same as:

```
100 THIS = NUMBER * 5
```

MOVE

Format: MOVE from__address, to__address, no.__of__bytes
Example: 20 MOVE MADDR1, MADDR2, 9

The MOVE statement moves bytes of memory from the area of memory whose lowest address is given by the first numeric expression (from__address) to the area whose lowest address is given by the second numeric expression (to__address). The third numeric expression specifies how many bytes are to be moved. The order of movement is such that the contents of the block of data are not changed by the move. MOVE's primary use is in player-missile graphics.

Example: MOVE 55,222,5

Five bytes with a starting low address at 55 (i.e., 55-60) will be moved to location 222-226.

STOP

Format: STOP
Example: 190 STOP

STOP is used to halt execution of a program at a place that is not the highest line number in the program. The STOP command prints the line number where execution of the program is broken. STOP is a useful debugging aid because you can use PRINT in the direct mode to show the value of variables at the point where execution halts. Also, you know that your program got as far as the STOP command.

CONT

Format: CONT
Example: CONT

CONT resumes program execution from the point at which it was interrupted by either STOP, the BREAK key, or a program error. This instruction is often useful in debugging a program. A breakpoint can be set using the STOP statement. You can check variables at the point where execution stops by using PRINT *variable__name* in the direct mode (without a line number). Then resume the program by using the CONT statement.

END

Format: END
Example: 990 END

END halts the execution of a program and is usually the last statement in a program. When END terminates a program, the prompt character appears on the screen. In ATARI Microsoft BASIC, it is not necessary to end a program with the END statement.

GOTO

Format: GOTO line__number
Example: 10 GOTO 110

GOTO tells which line number is executed next. Normally, statements are executed in order from the lowest to highest number, but GOTO changes this order. GOTO causes a branch in the program to the line number following GOTO.

Example: GOTO 55

Since this statement does not have a line number, it starts immediate execution of the program in memory starting at line number 55.

```
100 PRINT "THIS IS A COMPUTER"
120 GOTO 100
```
RUN RETURN

This program will cause endless branching to line number 100. Thus, the television screen quickly fills up with THIS IS A COMPUTER.

IF...THEN

Format: IF test__condition THEN goto__line__number or a__statement
Examples: 10 IF A=B THEN 290
20 IF J>Y AND J<V THEN PRINT "OUT OF STATE TAX"

If the result of an IF...THEN test is true, the next statement executed is *goto__line __number*. A test is made with the relational or mathematical operators. The test can be made on numbers or strings. The words GOTO after THEN are optional. If the statement test, *test__condition,* is false, the execution goes to the next numbered line in the program.

```
160 IF A__NUMBER > ANOTHER__NUMBER THEN 300
200 PRINT "ANOTHERNUMBER IS LARGER"
250 STOP
300 PRINT "ANUMBER IS LARGER"
450 END
```

IF...THEN...ELSE

Format: IF test__condition THEN goto__line__number or statement ELSE goto__line__number or statement
Example: 250 IF R<Y THEN 450 ELSE 200

This is the same as IF...THEN except that execution passes to the ELSE clause when the relational or mathematical test is untrue.

WAIT

Format: WAIT address, AND__mask__byte, compare__to__byte
Example: 330 &D40B,&FF,110 !WAIT FOR VBLANK

WAIT stops the program until certain conditions are met. Execution waits until the compare__to__byte, when ANDed with the AND__mask__byte, equals the byte contained in memory location address.

WAIT is ideal if you need to halt execution until VBLANK occurs. VBLANK occurs every 1/60 of a second. It consists of a number of lines below the visible scan area.You can make sure that your screen will not be interrupted halfway through its scan lines (causing the screen to blip) if you WAIT until a VBLANK occurs. This technique is used to animate characters as shown in Appendix C, Alternate Character Sets. See Appendix A for an example of the WAIT statement used to control the timing of vertical fine scrolling.

FOR...TO...STEP

Format: FOR starting_variable = starting_value TO ending-value STEP |increment|
Examples: 10 FOR X=1 TO 500 STEP 3
30 FOR Y=20 TO 12 STEP -2
20 FOR COUNTER=1 TO 250

The FOR/NEXT statement starts incrementing numbers by increment until *ending_number* is reached. When the ending number is counted, execution goes to the statement number after the NEXT statement.

FOR/NEXT determines how many times statements between the line numbers of the FOR...TO...STEP and the NEXT are executed repeatedly. If STEP is omitted, it is assumed to be 1. STEP can be a negative number or decimal fraction.

Example Program:

```
100 FOR X=1 TO 30
110 PRINT X, SQR(X)
120 NEXT
```

NEXT

Format: NEXT |variable_name|
Examples: 30 NEXT J,I
40 NEXT VB
120 NEXT

NEXT transfers execution back to the FOR..TO line number until the TO count is up. NEXT does not need to be followed by a variable name in Microsoft BASIC. When NEXT is not followed by a variable name, the execution is transferred back to the nearest FOR...TO statement.

Example Program:

```
100 FOR X=10 TO 100 STEP 10
110 PRINT X
120 NEXT
130 END
```

```
RUN RETURN

 10
 20
 30
 40
 50
 60
 70
 80
 90
 100
```

Two or more *starting-variables* can be combined on the same NEXT line with commas.

Example Program:

```
100 FOR X=1 TO 20
110 FOR Y=1 TO 20
120 FOR Z=1 TO 20
130 NEXT Z,Y,X
```

SUBROUTINES

A subroutine is a group of statements that you wish to use repeatedly in a program. The GOSUB statement gives execution to the group of statements. RETURN marks the end of the subroutine and returns execution to the statement after the GOSUB statement.

GOSUB

Format: GOSUB line_number
Example: 330 GOSUB 150

GOSUB causes *line_number* to be executed next. The statement starting with *line_number* is the start of a group of statements you wish to use a number of times in a program.

RETURN

Format: RETURN
Example: 550 RETURN

RETURN returns the program to the line number after the GOSUB statement which switched execution to this group of statements.

Example Program:

```
110 GOSUB 140
120 PRINT "THIS IS THE END"
130 STOP
140 PRINT "THIS IS THE START"
150 PRINT "OF CODE WHICH"
160 PRINT "IS EASY TO CALL"
170 PRINT "(EXECUTE) A NUMBER"
180 PRINT "OF TIMES IN A"
190 PRINT "PROGRAM"
200 RETURN ! EXECUTION CONTROL GOES TO LINE NUMBER 120
```

ON...GOTO

Format: ON *arithmetic_expression* GOTO line_number_1, line_number_2, line_number_3
Example: 400 ON X GOTO 550, 750, 990

ON...GOTO determines which line is executed next. It does this by finding the number represented by the *arithmetic_expression* and if the number is a 1, control passes to *line_number_1*. If the number is a 2, control passes to *line_number_2*. If the number is a 3, control passes to *line_number_3*, etc.

ON...GOSUB

Format: ON arithmetic__expression GOSUB line__number__1, line__number__2, line__number__3
Example: 220 ON X GOSUB 440, 500, 700

ON...GOSUB determines which line is executed next. It does this by finding the number represented by the *arithmetic__expression*. If the number is a 1 then execution passes to *line__number__1*. If the number is a 2, execution passes to *line__number__2*, or If the number is a 3, execution passes to *line__number__3*, etc.

RETURN is used to transfer execution back to the statement directly after the GOSUB.

Example Program:

```
110 ON X GOSUB 333, 440, 512, 620
...
...
333 B=B+C
340 RETURN
```

ON ERROR

Format: ON ERROR line__number
Example: ON ERROR 550

Program execution normally halts when an error is found and an error message prints on the television screen. ON ERROR traps the error and forces execution of the program to go to a specific *line number.*

The ON ERROR *line__number* statement must be placed before the error actually occurs in order to transfer execution to the specified *line__number.*

To recover normal execution of the program, you must use the RESUME statement. The RESUME statement transfers execution back into the program.

When RUN, STOP, or END is executed, the ON ERROR statement is terminated.

Example Program:

```
10 ON ERROR 1000
20 PRINT #3, "LINE"
30 STOP
1000 PRINT "DEVICE NOT OPENED YET"
1010 STOP
1020 RESUME
```

The ON ERROR *line__number* statement can be disabled by the statement: ON ERROR GOTO 0. If you disable the effect of ON ERROR within the error-handling routine itself, the current error will be processed in the normal way.

ERROR

Format: ERROR error__code
Example: 640 ERROR 162

ERROR followed by an *error__code* forces BASIC to evaluate an error of the specified *error__code* type. Forcing an error to occur is a technique used to test how the program behaves when you make a mistake. A complete listing of error codes is given in Appendix M. You can force both system errors and BASIC errors.

ERL

Format: ERL
Example: 100 PRINT ERL

ERL returns the line number of the last encountered error.

ERR

Format: ERR
Example:
```
120 PRINT ERR
150 IF ERR = 135 THEN GOTO 350
```

ERR returns the error number of the last encountered error.

AFTER

Format: AFTER (time_in_1/60_of_a_sec) |GOTO| *line_number*
Example: 100 AFTER (266) GOTO 220

When AFTER (...) is executed, a time count starts from 0 up to the number of 1/60 of a second (called jiffies). When the time is up, program execution transfers to *line_number*. AFTER can be placed anywhere in a program but it must be executed in order to start its count. A time period up to 24 hours is allowed.

When RUN, STOP, or END is executed the AFTER statement jiffie count is reset.

CLEAR STACK

Format: CLEAR STACK
Example: 100 CLEAR STACK

CLEAR STACK clears all current time entries. CLEAR STACK is a way to abort the AFTER statement. If certain conditions are met in a program, you may wish to cancel the AFTER statement.

Example Program:

```
100 AFTER (1333) GOTO 900
150 IF A=B THEN CLEAR STACK
900 PRINT "YOUR TURN IS OVER"
910 RESUME
```

STACK

Format: STACK
Examples:
```
120 PRINT STACK !Prints no. of stack entries available
310 IF STACK = 0 THEN PRINT "STACK FULL"
```

The STACK function gives the number of entries available on the time stack. The time stack can hold 20 jiffie entries. The STACK is used to hold the SOUND and AFTER jiffie times. This is a random stack since when a jiffie is up, time expires regardless of when the jiffies were put in the STACK.

RESUME

Formats:
```
RESUME |line_number|
RESUME |NEXT|
RESUME
```
Examples:
```
300 RESUME 55
440 RESUME NEXT
450 RESUME
```

RESUME is the last statement of the ON ERROR *line__number* error-handling routine. RESUME transfers control to the *line__number.*

RESUME *NEXT* transfers execution to the statement following the occurrence of the error.

RESUME transfers execution back to the originating (error causing) line number if you do not follow RESUME with *NEXT* or *line__number.*

OPTION BASE

Formats: OPTION BASE 0
OPTION BASE 1
Example: 150 OPTION BASE 1
200 DIM Z (25,25,25)!array element subscripts no. 1-25

OPTION BASE 1 declares that list and array subscript numbering will start with 1. The OPTION BASE (0/1) statement should be the first executable statement in a program. It states that you want the subscripted variables to begin with 0 or 1. If the OPTION BASE statement is omitted, lists' and arrays' subscript numbering starts at 0.

Example Program:

```
100 REM DEMONSTRATES OPTION BASE 1 STATEMENT
110 OPTION BASE 1
120 DIM ARRAY (15,15)
150 READ ARRAY (1,1), ARRAY (2,2), ARRAY (15,15)
165 DATA 32,33,34
180 PRINT ARRAY (1,1), ARRAY (2,2), ARRAY (15,15)
190 END
```

CLEAR

Format: CLEAR
Examples: CLEAR
550 CLEAR

CLEAR zeros all variables and arrays, and nulls all strings. If an array is needed after a CLEAR command, it must be redimensioned.

COMMON

Formats: COMMON variable__name,|variable__name|
COMMON ALL
Examples: 110 COMMON I, J, A$, H%, DEC, F()
100 COMMON ALL

Use COMMON to keep variable values the same across program runs. COMMON makes variables in two programs the same variable in fact as well as in name. If you name a variable COUNT in one short program and join that program with another program that has COUNT as a variable, the program will consider the COUNTs to be different variables. The COMMON statement says that you want both COUNTs to be considered the same variable. COMMON ALL keeps all previous variable values the same across the new program run.

Example Program:

```
100 COMMON X
110 X=4
120 RUN "D:PROG2"
```

BREAK

PRINT X RETURN

The value of X=4 after line 120 calls the new program is 4. If there is already a variable named X in the second program, then X gets its value from the new program.

RANDOMIZE

Format: RANDOMIZE |seed|
Examples: 10 RANDOMIZE
10 RANDOMIZE 55 !Sets a certain repeatable sequence

RANDOMIZE assures that a different random sequence of numbers will occur each time a program with the RND arithmetic function is run. RANDOMIZE gives a random seed to the starting point of the RND sequence.

Example Program:

```
100 RANDOMIZE
110 PRINT RND
120 END
```

Each time you run the above program, a unique number prints on the television screen.

The RND arithmetic function will repeat the same pseudo-random number each time a program is run without RANDOMIZE. In testing a program it is sometimes ideal to have an RND sequence that you know will be the same each time. In this case, use the RND function by itself without RANDOMIZE. Another way to produce a long sequence that will be the same each time, is to use RANDOMIZE |seed| (where |seed| is an arbitrary number). But if you wish to see a different set of cards each time you play the game, just use RANDOMIZE by itself somewhere near the start of your program.

Example of RND without RANDOMIZE:

```
100 PRINT RND
110 END
```

Each time you run this program, it prints the same number on the television screen.

OPTION PLM1, OPTION PLM2, OPTION PLM0

Format: OPTION PLM1
OPTION PLM2
OPTION PLM0
Example: 100 OPTION PLM1
100 OPTION PLM2
700 OPTION PLM0

OPTION PLM1 reserves 1280 bytes in memory for player-missiles (single-line resolution). OPTION PLM2 reserves 640 bytes in memory for player-missiles (double-line resolution). OPTION PLM0 releases all OPTION PLM reservations.

The GRAPHICS instruction (see Section 12) must always precede the OPTION PLMn statement. This is because the computer must first know the graphics mode before you reserve space.

Use OPTION PLM1 or OPTION PLM2 to reserve player-missile memory, clear the memory, and set PMBASE. You do not need to worry about the proper memory area to place player-missiles when you use the OPTION PLM statements. To find the exact memory location of the starting byte of your missiles, use VARPTR(PLM1) or VARPTR(PLM2).

You must poke decimal location 53277 with decimal 3 in order to enable player-missile graphics. You must also poke decimal location 559 with decimal 62 for single-line resolution or decimal 46 for double-line resolution. See Section 13 for an example of player-missile graphics.

OPTION CHR1, OPTION CHR2, OPTION CHR0

Format: OPTION CHR1
OPTION CHR2
OPTION CHR0

Examples:
```
110 OPTION CHR1
120 OPTION CHR2
130 OPTION CHR0
```

OPTION CHR1 reserves 1024 bytes in memory for character data. *OPTION CHR2* reserves 512 bytes in memory for character data. *OPTION CHR0* releases all OPTION CHR reservations.

Use OPTION CHR1 or OPTION CHR2 to reserve memory for a RAM character set. You can MOVE the ROM character set into the new RAM area you have reserved or you can define a totally new character set. VARPTR(CHR1) orVARPTR(CHR2) will point to the starting address of the zeroth character. It is necessary to POKE a new starting address into CHBAS. This can be done by determining the page to which VARPTR(CHR1) or VARPTR(CHR2) is pointing. One way to determine and POKE a new CHBAS is:

```
300 CHBAS=&2F4
310 ADDR%=VARPTR(CHR1)
320 POKE CHBAS,((ADDR%/256) AND &FF)
```

The GRAPHICS instruction (see Section 12) must always precede the OPTION CHRn statement. This is because the computer must first know the graphics mode before you reserve space.

This procedure will mask for the Most Significant Byte (MSB) of the VARPTR memory address and POKE that MSB into CHBAS so you will switch to the new character set. See Appendix C for an example of redefining the character set.

OPTION RESERVE

Format: OPTION RESERVE n
Example: 300 OPTION RESERVE 24

In the OPTION RESERVE n statement, n is a number representing the number of bytes reserved. For example, OPTION RESERVE 24 reserves 24 bytes. VARPTR(RESERVE) can be used to tell you the starting address of the 24 bytes in OPTION RESERVE 24. This statement allows you to reserve bytes for machine code or for another purpose.

VARPTR

Formats:
```
VARPTR(variable__name)
VARPTR(PLM1)
VARPTR(PLM2)
VARPTR(CHR1)
VARPTR(CHR2)
VARPTR(RESERVE)
```
Examples:
```
110 A=VARPTR(A$)
100 PRINT VARPTR(A$+1)
120 J=VARPTR(TOTAL)
120 T=VARPTR(CHR2)
155 POKE VARPTR(RESERVE),&FE
```

If the argument to this function is a variable name, the function returns the address of the variable's symbol table entry. When the variable is arithmetic,VARPTR returns the variable's 2-byte starting address (Most Significant Byte, Least Significant Byte) in memory. When the variable is a string, VARPTR returns the number of bytes in the string. Then the starting location of the string is given in VARPTR(A$)+1 **Least Significant Byte** and VARPTR(A$)+2 **Most Significant Byte**. Notice that only in the case of strings is the address given in the 6502 notation of low-memory byte before the high-memory byte. Except in the case of strings the whole address in high byte; low-byte format is returned with VARPTR. The following keywords can be used with VARPTR.

VARPTR(PLMn)	Returns the address (MSB, LSB) of the first byte allocated for PLMn.
VARPTR(CHRn)	Returns the address (MSB, LSB) of the first byte allocated for CHRn.
VARPTR(RESERVE)	Returns the address (MSB, LSB) of the first byte allocated for assembly language programs.

Use OPTION PLM1, OPTION PLM2, OPTION CHR1, OPTION CHR2, and OPTION RESERVE n to allocate space. Once OPTION has been used to set aside space, VARPTR can be used to point to the starting byte of that space.

7

INPUT/OUTPUT STATEMENTS

The keyboard, disk drive, program recorder, and modem are ways your computer gets information — **Input**. The ATARI Home Computer also gives information by writing it on the television screen, cassette tape, printer, or diskette — **Output**.

ATARI **input** and **output** devices have identifying codes:

K: Keyboard. Input-only device. The keyboard allows the computer to get information directly from the typewriter keys.

P: Line Printer. Output-only device. The line printer prints ATASCII characters, a line at a time.

C: Program Recorder. Input and output device. The recorder is a read/write device that can be used as either, but never as both simultaneously. The cassette has two tracks for sound and program recording purposes. The audio track cannot be recorded from the ATARI Computer system, but may be played back through the television speaker.

D1:,D2:,D3:,D4: Disk Drives. Input and output devices. If 32K of RAM is installed, the ATARI Computer can use four **ATARI 810™ Disk Drives**. The default is D1: if no drive is designated.

E: Screen Editor. Input and output device. This device uses the keyboard and television screen (see **S:** TV Monitor) to simulate a screen editing terminal. Writing to this device causes data to appear on the display starting at the current cursor position. Reading from this device activates the screen-editing process and allows the user to enter and edit data. Whenever the RETURN key is pressed, the entire line is selected as the current record to be transferred by Central Input/Output (CIO) to the user program.

S: TV Monitor. Input and output device. This device allows the user to read characters from and write characters to the display, using the cursor as the screen-addressing mechanism. Both text and graphics operations are supported.

R: Interface, RS-232. The **ATARI 850™ Interface Module** enables the ATARI Computer system to interface with RS-232 compatible devices such as printers, terminals, and plotters.

PRINT

Formats:
```
PRINT "string_constant"
? "string_constant", variable_name
PRINT variable_name_1, variable_name_2, variable_name_etc
PRINT#iocb, AT(s,b);X,Y
PRINT#6, AT(x,y);"string_constant";variable_name
```

Examples:
```
100 PRINT "SORTING PROGRAM";A$,X
500 ?#6, "ENTERING DUNGEON" !Print for GRAPHICS 1 and 2
```

PRINT puts string constants, string variables, or numeric variables on the television screen when executed. The PRINT statement will leave a blank line when executed alone. The question mark symbol (?) means the same thing as the word PRINT.

Example Program:

```
100 PRINT "SKIP A LINE"
120 PRINT
125 REM NOTE USE OF "" TO PRINT A QUOTE
130 ANOTHER_LINE$="PRINT ""ANOTHER"" LINE"
140 ? ANOTHER_LINE$
150 END
```

Line 120 leaves a blank line when this program is run:

```
SKIP A LINE

PRINT "ANOTHER" LINE
```

String constants, string variables, and numeric variables will all print on the same line when the line construction includes a comma or semicolon.

It is not necessary to use a closing quote if you wish to print a *string_constant* on your television screen:

```
100 PRINT "NO CLOSING QUOTE HERE
```

RUN RETURN

```
NO CLOSING QUOTE HERE
```

PRINT#iocb will print at a particular sector and byte if the disk drive has been opened as OUTPUT (see OPEN statement). The AT clause is quite versatile. If the device being addressed is a disk drive, AT(s,b) refers to the sector, byte. However, if the device being addressed is the screen, as in PRINT or PRINT#6, then the AT(x,y) refers to the x,y screen position.

An example of printing to a disk drive:

```
100 OPEN#3, "D:TEST.DAT" OUTPUT
110 X=5
120 PRINT#3, AT(7,1);"TEST";X
130 CLOSE#3
```

An example of printing to a screen location:

```
100 GRAPHICS 1
110 PRINT#6, AT(3,3);"PRINTS ON SCREEN"
```

TAB

Format: TAB(n)
Example: 120 PRINT TAB(5);"PRINT STARTS 5 SPACES IN"

TAB moves the cursor over the number of positions specified within the parentheses. This statement is used with PRINT to move characters over a number of tabbed spaces.

Example Program:

```
100 PRINT TAB (5);"THIS LINE IS TABBED RIGHT FIVE"
120 END
```

SPC

Format: SPC(n)
Example: 10 PRINT TAB (5);"XYZ";SPC (7);"SEVEN SPACES RIGHT OF XYZ"

SPC puts spaces between variables and constants in a line to be printed. The TAB always sets tabs from the left-hand margin. SPC counts spaces from where the last variable or constant ends.

PRINT USING

PRINT USING lets you format your output in many ways:

- Numeric variable digits can be placed exactly where you want them.
- You can insert a decimal point in dollar amounts.
- You can place a dollar sign ($) immediately in front of the first digit of a dollar amount.
- You can print a dollar sign ahead of an amount.
- Amounts can be padded to the left with asterisks (***$45.00) for check protection purposes.
- Numbers can be forced into exponential (E) or double-precision (D) format.
- A plus sign (+) causes output to print as a + for positive and a - for negative numbers.

PRINT USING #

The pound sign # holds a position for each digit in a number. Digits can be specified to the right or left of the decimal point with the pound sign #. Zeros are inserted to the right of the decimal, if needed, in the case where the amount is in whole dollars. Decimal points are automatically lined up when # is used. The # is convenient in financial programming.

Example Program:

```
10 X=246
20 PRINT USING "###";X
```

RUN RETURN
```
246
```

If a number has more digits than the number of pound signs, then a percent sign will print in front of the number.

Example Program:

```
100 X=99999 110 PRINT USING "###";X
120 END
```

RUN RETURN

```
%99999
```

PRINT USING .

Place the period anywhere within the # decimal place holders. The decimal in the amount will align with the decimal in the USING specification.

```
10 X=2.468 20 PRINT USING "##.##";X
```

RUN RETURN

```
2.47
```

Note that since only two digits were specified after the decimal point, the cents position was rounded up.

PRINT USING ,

Place a comma in any PRINT USING digit position. The comma symbol causes a comma to print to the left of every third digit in the result. Extra decimal position holders (#) must be used if more than one comma is expected in a result.

Example Program:

```
10 X#=2933604.53 !Double precision needed this # tag
20 PRINT USING "#######,.##";X#
30 END
```

RUN RETURN

```
2,933,604.53
```

PRINT USING **

Two asterisks in the first two positions fill unused spaces in the result with asterisks. The two asterisks count as two additional digit positions.

Example Program:

```
100 X=259
120 PRINT USING "**#######.##";X
```

RUN RETURN

```
******259.00
```

PRINT USING $

A dollar sign at the starting digit position causes a dollar sign to print at the left digit position in the result.

Example Program:

```
100 X=3.59631
110 PRINT USING "$###.##";X
120 END
```

RUN RETURN

```
$  3.60
```

PRINT USING $$

Two dollar signs ($$) in the first two positions give a floating dollar sign in the result. That is, the dollar sign will be located immediately next to the first decimal digit that is displayed.

Example Program:

```
100 X=3.5961
110 PRINT USING "$$###.##";X
120 END
```

RUN RETURN

```
$3.60
```

PRINT USING **$

If **$ is used in the first three positions the result will have asterisks filling unused positions and a dollar sign will float to the position immediately in front of the first displayed digit.

Example Program:

```
100 X=53.29
110 PRINT USING "**$########.##";X
120 END
```

RUN RETURN

```
********$53.29
```

PRINT USING ^^^^

Four exponentiation symbols after the pound sign (#) decimal place holder will cause the result to be in exponential (E or D) form.

Example Program:

```
100 X=500
110 PRINT USING "##ΛΛΛΛ";X
120 END
```

RUN RETURN

```
5E+02
```

PRINT USING +

The plus sign (+) prints a + for positive and a minus (-) for negative in front of a number. The plus sign (+) can be used at the beginning or end of the PRINT USING string.

Example Program:

```
100 A=999.55
110 PRINT USING "+####";A
120 END
```

RUN RETURN

```
+1000
```

PRINT USING -

The minus (-) sign following the PRINT USING string makes a —appear following a negative number. A trailing space will appear if the number is positive.

Example Program:

```
100 A=-998
110 PRINT USING "###-";A
120 END
```

RUN RETURN

```
998-
```

PRINT USING !

The exclamation sign (!) pulls the first character out of a string.

Example Program:

```
100 A$="B MATHEMATICS 1A"
110 PRINT USING "!";A$
120 END
```

RUN RETURN

```
B
```

PRINT USING %bbbb%

The percent signs (%) and blank spaces (b) will pull part of a string out of a longer string. The length of the string you pull out is 2 plus the number of spaces (b's) between the percent signs.

Example Program:

```
100 A$="Smith Fred"
120 PRINT USING "%bbb%";A$
130 END
```

RUN RETURN

```
Smith
```

INPUT

Format: INPUT|#iocb| |"prompt__string"|, |AT(s,b)|; variable__name, |variable__name|
INPUT#6 |"prompt__string"|, |AT(x,y)|; variable__name
Examples: 120 INPUT "TYPE YOUR NAME";A$
350 INPUT "ACCOUNT NO., NAME";NUM,B$
300 INPUT#5, AT(9,7);X

INPUT lets you communicate with a program by typing on the computer keyboard. You are also allowed to print character strings with the INPUT statement. This lets you write prompts for the user such as TYPE YOUR NAME. The typed characters are assigned to the variable names when you press the RETURN key or type a comma. The INPUT statement temporarily stops the the program until keyboard INPUT is complete. The INPUT statement automatically puts a question mark on the television screen.

If a disk drive has been opened as INPUT and assigned an IOCB#, then it can be used to input data. The input from the device is read AT(sector,byte) and assigned a variable name. INPUT#6, AT(x,y);X can be used to read a specific screen location.

LINE INPUT

Format: LINE INPUT|#iocb| |"prompt__string"| string__variable__name$
Example: 190 LINE INPUT ANS$

An entire line is input from the keyboard. Commas, colons, semicolons, and other delimiters in the line input from the keyboard are ignored. Mark the end of the line by pressing RETURN or its ASCII equivalent &9B for the End of Line (EOL).

Example Program:

```
100 LINE INPUT "WHAT IS YOUR NAME?"; N$
120 PRINT N$
130 END
```

DATA

Format: DATA arithmetic__constant,|arithmetic__constant|
DATA string__constant,|string__constant|
Example: 140 DATA 55,793,666,94.7,55
150 DATA ACCOUNT,AGE,""""NAME"""",SOCIAL SECURITY

The *arithmetic__constants* and *string__constants* in the DATA statement are assigned to variable names by the READ statement. Use a comma to separate the entries that you wish to input with DATA/READ. More than one DATA statement can be used. The first DATA item is assigned the first variable name encountered in READ; the second DATA item is assigned the second variable name, etc. When all the items are read and the program tries to read data when none exists — an "out-of- data" error occurs. The ERR statement can be used to test for the out-of-data condition.

If a comma is included in a string item in a data statement, then the whole string item must be enclosed in quotes. Otherwise, it could be mistaken as a comma used to separate items in the DATA statement. Quotes are not required if a string uses numeric values as string data.

READ

Format: READ variable__name__1,|variable__name__2,||variable__name__etc.|
Example: 150 READ A,B

READ assigns numbers or strings in the DATA statement to variable names in the READ statement. Commas separate variable names in the READ statement and items in the data statement. Hence, it is all right to leave extra spaces between items because the comma determines the end of items. READ A, B, C looks at the first three DATA items. If READ A, B, C is executed again, the next three numbers of the data statement are assigned to A, B, C respectively. The pairing of variables and data continues until all the data is read.

Example of DATA/READ:

```
100 FOR J = 1 TO 3
120 READ A$,A
130 PRINT A$,A
140 NEXT J
150 DATA FRED,50,JACK,20,JANE,200
900 PRINT "END OF DATA"
910 END
```

RESTORE

Format: RESTORE |line__number|
Examples: 440 RESTORE 770
550 RESTORE

The RESTORE statement is used if data items are to be used again in a program. That is, RESTORE allows use of the same DATA repeated a number of times. Without the RESTORE statement an out-of-data error results from the attempt to READ data a second time. The data can be restored starting with a particular line number using the optional |line__number|.

AT

Formats: PRINT#6, AT(x,y);variable__name,"string__constant"
PRINT AT(x,y);variable__name,"string__constant"
PRINT#iocb, AT(s,b);variable__name,"string__constant"
INPUT#iocb, AT(s,b);variable__name

AT can be added to either PRINT or INPUT. The numbers following AT refer to sector, byte if the proper disk #iocb has been opened. (See OPEN statement below.) The television screen is the output device when PRINT, or PRINT#6, are encountered. When the screen is the device, AT(x,y) gives the coordinates for printing.

OPEN

Format: OPEN #iocb, "device:program__name" file__access

Examples:

```
130 OPEN #4, "K:" INPUT
100 OPEN #3, "P:" OUTPUT
150 OPEN #4, "D:PROG.SAV" INPUT
120 OPEN #2, "D:GRAPH1.BAS" UPDATE
110 OPEN #5, "D:PROG.BAS" APPEND
```

#	Mandatory character entered by user.
#iocb,	Input/output control block (ICOB). Choose a number from 1 to 7 to identify a file and its file access. You must have a pound sign (#) followed by an IOCB number (1-7) and a comma.
"device:program__name"	Specifies the device and the name of the program. Devices are D: (disk), P: (printer), E: (screen editor), K: (keyboard), C: (cassette), S: (television monitor), and R: (RS-232-C). When you use D: your program name follows the colon. The name of your program can be up to eight characters long and have a three-character extension. Program names must begin with an alphabetic character. At the beginning of this section you will find a complete description of the device codes (K:, P:, C:, D:, E:, S:, R:).
file__access	Tells the type of operation: INPUT = input operation OUTPUT = output operation UPDATE = input and output operation APPEND = allows you to add onto the end of a file.

The idea behind the OPEN statement is to identify a number (the IOCB#) with the file access characteristics. After the **OPEN#n** statement is encountered in a program, you can use PRINT#2, INPUT#3, NOTE#5, STATUS#2, GET#4, and PUT#4. That is, you can use the IOCB# as an identifier.

The OPEN#n and PRINT#n statements now substitute for LPRINT (LINE PRINTING):

```
100 OPEN#3, "P:" OUTPUT
110 PRINT#3, "THIS IS A PRINTER TEST"
120 CLOSE#3
```

The following IOCB# identifiers have preassigned uses:

- #0 is used for INPUT and OUTPUT to E:, the screen editor.
- #6 is used for INPUT and OUTPUT to S:, the screen itself, in test modes GRAPHICS 1 and GRAPHICS 2.

An example of the use of IOCB #6 is:

```
100 GRAPHICS 2
110 PRINT#6, AT(5,5); "SCREEN PRINT TEST"
```

IOCBs #1 through #5 (and IOCB #7) can be used freely, but the preassigned IOCBs should be avoided unless a program does not use them for one of the preassigned uses mentioned above.

CLOSE

Format: CLOSE #iocb
Example: CLOSE #2

Use CLOSE after file operations are completed. The # sign is mandatory and the number itself identifies the IOCB.

#	Mandatory symbol
icob	The number of a previously opened IOCB

NOTE

Format: NOTE#iocb,variable__name,|variable__name|
Example: 120 NOTE#4, I,J

Use NOTE to store the current diskette sector number in the first variable__name and the current byte number within **byte**. This is the current read or write position in the specified file where the next byte to be read or written is located.

PUT/GET

Formats: PUT#iocb, |AT(sector,byte);| arithmetic__expression
GET#iocb, |AT(sector,byte);| variable__name
Examples: 100 PUT#6, ASC("A")
200 GET#1, X
330 GET#3, AT(8,2);J,K,L

PUT and GET are opposites. PUT outputs a single byte value from 0-255 to the file specified by *#iocb* (# is a mandatory character in both of these commands). GET reads 1-byte values from 0-255 (using #iocb to designate the file, etc. on diskette or elsewhere) and then stores the byte in the variable arithmetic__expression.

STATUS

Formats: STATUS (iocb__number)
STATUS ("device:program__name")
Examples: 100 A=STATUS (6)
120 A=STATUS ("D:MICROBE.BAS")

STATUS returns the value of the fourth byte of the iocb block (status byte). The Most Significant Bit (MSB) is a 1 for error conditions. A zero in the MSB indicates nonerror conditions. The remaining bits represent an error number.

TABLE 7-1
LIST OF STATUS CODES

Hex	Dec	Meaning
01	001	Operation complete (no errors)
03	003	End of file (EOF)
80	128	BREAK key abort
81	129	IOCB already in use (OPEN)
82	130	Nonexistent device
83	131	Opened for write only
84	132	Invalid command
85	133	Device or file not open
86	134	Invalid IOCB number (Y register only)
87	135	Opened for read only
88	136	End of file (EOF) encountered
89	137	Truncated record
8A	138	Device timeout (doesn't respond)
8B	139	Device NAK
8C	140	Serial bus input framing error
8D	141	Cursor out of range
8E	142	Serial bus data frame overrun error
8F	143	Serial bus data frame checksum error
90	144	Device-done error
91	145	Bad screen mode
92	146	Function not supported by handler
93	147	Insufficient memory for screen mode
A0	160	Disk drive number error
A1	161	Too many open disk files
A2	162	Disk full
A3	163	Fatal disk I/O error
A4	164	Internal file number mismatch
A5	165	Filename error
A6	166	Point data length error
A7	167	File locked
A8	168	Command invalid for disk
A9	169	Directory full (64 files)
AA	170	File not found
AB	171	Point invalid

EOF

Format: EOF(n)
Example: 120 IF EOF(4)=0 THEN GOTO 60

A value of true or false will be returned indicating the detection of an end-of-file condition on the last read of IOCB n.

8 ARRAYS

ABOUT ARRAYS

You are allowed up to 10 subscripted elements in a list or array without having to use the dimension (DIM) statement.

For example:

```
100 AN_ARRAY(1)=55
120 AN_ARRAY(2)=77
130 AN_ARRAY(3)=93
140 AN_ARRAY(4)=61
150 FOR X=1 TO 4
160 PRINT AN_ARRAY(X)
170 NEXT
180 END
```

An array with more than 10 elements must be dimensioned to reserve space for it in RAM.

DIM

Formats: DIM arithmetic_variable_name (number_of_elements), |list|
DIM string_variable_name$ (number_of_elements), |list|
Example: 10 DIM A$ (35), TOTAMT (50)

The DIM statement tells the computer the number of elements you plan to have in an array. If you enter more data elements into an array than you have allowed for in a dimension statement, you will get an error message.

The simplest array is the one-dimensional array. Let's say a teacher has 26 students in a class. He can record a numeric test score for each student by dimensioning:

```
10 OPTION BASE 1
20 DIM SCORE(26)
30 SCORE (1)=55
40 SCORE (2)=86
50 PRINT SCORE (1), SCORE (2)
RUN
```

Notice that the OPTION BASE statement begins the array subscripting with 1, thus SCORE (1) stores the numeric score of the first student. OPTION BASE 0 will allow you to begin subscripting with the number 0.

ATARI Microsoft BASIC allows you to have up to 255 array dimensions. Three-dimensional arrays allow you to make complex calculations easily.

Example Program:

```
110 X=20:Y=30:Z=25
120 DIM BOXES(X,Y,Z)
130 !Without an OPTION (0/1) the OPTION BASE defaults to 0
```

9

FUNCTION LIBRARY

ABS

Format: ABS (expression)
Example: ABS (-7)

ABS returns the absolute value of a number. The sign of a number will always be positive after this function is executed. If the number -7 (negative 7) is evaluated with ABS, the result will be 7 (positive 7).

INT

Format: INT (arithmetic__expression)
Examples: ? INT (5.3) *prints 5 on your television screen*
? INT (-7.6) *prints -8 on your television screen*

INT returns an integer for the arithmetic__expression. INT always rounds to the next lower integer.

SGN

Format: SGN (arithmetic__expression)
Example: ? SGN (-34) *prints -1 on your television screen*

SGN returns the sign of the *arithmetic__expression* enclosed in parentheses. The sign is +1 if the number within the parentheses is positive, 0 if the number is 0, or -1 if the number is negative.

SQR

Format: SQR (arithmetic__expression)
Example: ? SQR (25) *prints 5 on your television screen*

SQR returns the square root of a positive *arithmetic__expression* enclosed in parentheses. If the *arithmetic__expression* evaluated by SQR has a negative (-) sign, you will get an ILLEGAL QUANTITY ERROR.

RND

Formats: RND Returns a random single-precision value between 0 and 1.
RND (0) Same as RND above.
RND (integer) Returns an integer between 1 and the integer inclusive.
Examples: ? RND Prints 6 random digits after decimal point.
RND (37) Prints a number between and including 1 through 37.

RND returns random numbers. RND and RND (0) return random numbers between but not including 0 and 1. RND (integer) returns a positive integer between and including 1 and the (integer).

LOG

Format: LOG (arithmetic__expression)
Example: ? LOG (5) *prints the natural logarithm 1.60944*

LOG returns the natural logarithm (LOG_e) of a nonnegative *arithmetic__expression* in the parentheses. LOG (0) **will** give a FUNCTION CALL ERROR. LOG (1) is 1.61385904E-10.

EXP

Format: EXP (arithmetic__expression)
Example: ? EXP (3) *prints 20.0855*

EXP returns the Euler's number (e) raised to the power of the *arithmetic__expression* within the parentheses.

SIN

Format: SIN (arithmetic__expression)
Example: ? SIN (1) *prints the sine of 1 as .841471 radian*

SIN returns the trigonometric sine of the *arithmetic__expression.*

COS

Format: COS (arithmetic__expression)
Example: ? COS (.95) *prints cosine of .95 as .581683 radian*

COS returns the trigonometric cosine of the *arithmetic__expression.*

ATN

Format: ATN (arithmetic__expression)
Example: ? ATN (.66) *prints arctangent of .66 as .583373 radian*

ATN returns the arctangent of the *arithmetic-expression.*

TAN

Format: TAN (arithmetic__expression)
Example: ? TAN (.22) *prints the tangent of .22 as .223619 radian*

TAN returns the trigonometric tangent of the *arithmetic__expression.*

SPECIAL-PURPOSE FUNCTIONS

PEEK

Format: PEEK (address)
Examples: 110 PRINT PEEK(1034)
135 PRINT PEEK(ADDR)

PEEK (&FFF) looks at the address enclosed in the parentheses, in this case FFF hexadecimal. PEEK is used to discover the contents of a particular memory byte. You can examine ROM memory as well as RAM memory. All memory can be looked at with the PEEK instruction.

Examples:

```
PRINT PEEK(888)
```

Prints the byte in decimal at decimal memory location 888.

```
PRINT PEEK (&FFFF)
```

Prints the byte in decimal at memory location FFFF hex.

POKE

Format: POKE address,byte
Examples: POKE 2598,255

```
110 POKE ADDR3,&FF
120 POKE PLACE,J
```

POKE inserts a byte into an address location. The address and byte can be expressed as decimal or hexadecimal numbers. The address and byte can also be expressions. Thus, if X*Y-2 evaluates to a valid memory location or byte, it can be used.

Example:

```
POKE &FFF,43
```

Puts decimal 43 into hexadecimal location FFF.

```
X=22
Y=&8F

POKE X,Y
```

Puts hexadecimal 8F into memory location 22 decimal.

Note that decimal and hexadecimal are just two ways of assigning a number to the 8-bit byte. The highest number you are allowed to POKE, a byte, is FF in hexadecimal and 255 in decimal.

FRE (0)

Format: FRE (0)
Example: PRINT FRE(0)

This function gives you the number of RAM bytes that are free and available for your use. Its primary use is in direct mode with a dummy variable (0) to inform the programmer how much memory space remains for completion of a program. Of course FRE can also be used within a BASIC program in deferred mode. Using FRE (0) will release string memory locations that are not in use. This use of FRE (0) to pick up the string clutter is referred to as "garbage collection."

USR

Format: USR (address,n1)
Example: 550 A=USR(898,0)

USR passes the result of a machine language subroutine to a variable name. The USR function branches to a machine language routine address and can pass an optional value, n1. The value of n1 is usually the address of a data table used in the machine language routine.

During the execution of a USR routine, the programmer may use page zero RAM from &CD through &FF. The parameter passed will be stored in &E9 and &EA as data, and in &E3 and &E4 as an address. The parameter is assumed to be an integer or VARPTR.

Example Program:

```
10 ! ROUTINE TO TEST USR FUNCTION
20 ! THE ASSEMBLY ROUTINE IS:
30 !
40 ! LDA #35
50 ! STA 710
60 ! RTS
70 !
80 !
90 !
100 A=0:I=0:COL=0:C=0
110 OPTION RESERVE 10
120 ADDR = VARPTR(RESERVE) !STARTING ADDRESS
130 FOR I=0 TO 5
140 READ A
150 POKE ADDR+I,A
160 NEXT I
170 DATA &A9,&23,&8D,&C6,&02,&60
180 A=USR(ADDR,VARPTR(I))
190 STOP
```

TIME

Format: TIME
Example: 200 PRINT TIME

TIME gives the Real-Time Clock (RTCLOK) locations' contents. The decimal locations 18, 19, and 20 (RTCLOK) keep the system time in jiffies (1/60 of a second). Six decimal digits are returned by TIME. The difference between TIME$ and TIME is that TIME$ gives the time in standard hours, minutes, and seconds, while TIME gives the time as a jiffie count.

10 STRINGS

+ (Concatenation Operator)

Format: string + string
Example: 110 C$ = A$ + B$

Use the + symbol to bring two strings together.

Example Program:

```
110 A$="never"
120 B$="more"
130 Z$=A$+B$
140 PRINT Z$
```

RUN RETURN

```
nevermore
```

MID$

Format: MID$(string__expression__$,start,n)
Example: 100 A$="GETTHEMIDDLE"
110 PRINT MID(A$,4,3)

string__expression__$	String that will have characters pulled from its middle.
start	The character you wish to start with — counting from the left.
n	Number of characters you want to pull.

The string is identified by the first parameter of the function. The second parameter tells the starting character. The third parameter tells how many characters you want.

Example Program:

```
110 A$="AMOUNT OF INTEREST PAID"
120 B$=MID$(A$,11,8)! THIS CAUSES "INTEREST" TO BE PRINTED
130 PRINT B$
```

LEFT$

Format: LEFT$(string__expression__$,n)
Example: 100 A$="TOTALAMOUNT"
110 PRINT LEFT$(A$,5)

string__expression__$	String variable name or string expression.
n	Number of characters you want returned from the left side of the string.

RIGHT$

Format: RIGHT$(string__expression__$,n)
Example: A$="THERIGHT"

```
110 PRINT RIGHT$(A$,5)
```

string__expression__$	String variable name or string expression.
n	Number of characters to be taken from right side of the string.

LEN

Format: LEN (string__expression__$)
Example:

```
100 A$="COUNT THE"
120 ? LEN (A$+" CHARACTERS")!prints total number of
130 ! characters as 20
```

LEN returns the total number of characters in a *string__expression__$*. LEN stands for length. Spaces, numbers, and special symbols count as characters.

ASC

Format: ASC (string__expression__$)
Example: ? ASC("Smith")!prints 83 ATASCII decimal code for letter S

ASC gives the ATASCII code in decimal for the first character of the string enclosed in parentheses. See Appendix K for ATASCII Character Set.

VAL

Format: VAL (numeric__string__expression__$)
Example:

```
100 B$="309"
120 ? VAL (B$)!prints the number 309
130 END
```

VAL converts strings to numeric values. VAL returns the numeric value of the numeric constant associated with the *numeric__string__expression* in the parentheses. Leading and trailing spaces are ignored. Digits up to the first nonnumeric character will be converted. For example, PRINT VAL("123ABC") prints 123.If the first character of the string expression is nonnumeric, then the value returned will be 0 (zero).

CHR$

Format: CHR$ (ATASCII__code__number)
Examples:

```
110 PRINT CHR$ (123) !prints ATASCII club symbol
100PRINT CHR$(65) !PRINTS ATASCII CHARACTER A
```

CHR$ converts ATASCII values into one-character strings. CHR$ is the opposite of the ASC function. The *ATASCII__code__number* can be any number from 0 to 255. Appendix K gives a table of both the character set and the ATASCII__code__numbers.

INSTR

Format: INSTR (start,A$,B$)
Example: 110 HOLD = INSTR(5,C$,B$)

INSTR searches for a small string B$ within a larger string A$. The search can begin (start) a number of characters into the larger string. This starting position is assumed to be the first character if **start** is missing. The function returns the character position within A$, where B$ starts, or returns a 0 if B$ is not found.

STR$

Format: STR$ (arithmetic__expression)
Example:
```
100 A=999.02
110 PRINT STR$(A)
```

STR$ turns an *arithmetic__expression* into a string. String operations can be carried out on *arithmetic__expressions* with the STR$ function. Note that when the following two strings are brought together with the concatenation symbol, there is a space between them which represents the sign of the number.

Example Program:

```
100 NUM1=-22.344
120 NMU2=43.2
130 PRINT STR$ (NUM1) + STR$ (NUM2)
140 END
```

STRING$ (N,A$)

Format: STRING$ (N,A$)
Example: 100 A$=STRING$(20,"*")

STRING$(N,A$) returns a string composed of N repetitions of A$.

STRING$ (N,M)

Format: STRING$ (N,M)
Example: 110 PRINT STRING$(20,123)!prints 20 clubs

STRING$(N,M) returns a string composed for N repetitions of CHR$(M).

INKEY$

Format: INKEY$
Example: 110 A$=INKEY$

INKEY$ records the last key pressed. If no keys are currently being pressed on the keyboard, a null string is recorded. Statement 110 tests for a null string by representing it as two double quotes with no space between them. ATARI Microsoft BASIC does not recognize the space bar since leading and trailing blanks are trimmed for INKEY$.

Example Program:

```
100 A$=INKEY$
110 IF A$<>"" THEN PRINT "You typed a "; A$
120 GOTO 100
```

TIME$

Format: TIME$
Example: 100 PRINT TIME$

Set the time with the deferred mode statement:

```
190 TIME$ = "HH:MM:SS"
```

where HH = hours (up to 24)
MM = minutes
SS = seconds

Examples: 110 TIME$="22:55:05"
120 TIME$="05:30:09"

Note: Use leading zeros to make hours, minutes, and seconds into 2-digit numbers.

After TIME$ is set, you can use it in a program. TIME$ is continually updated to the current time, from your initial setting.

```
100 GRAPHICS 2
110 TIME$="11:59:05"
120 PRINT#6, AT(3,3);"DIGITAL CLOCK"
130 PRINT#6, AT(4,4);TIME$
140 GOTO 120
```

SCRN$

Format: SCRN$(x,y)
Example: 10 ? SCRN$(5,5)

The character at the X-coordinate and Y-coordinate is returned as the value of the function in character-graphics modes. In other graphics modes, SCRN$ returns the color register number being used by the pixel at location x,y.

Example of SCRN$(x,y):

```
10 GRAPHICS 1
20 COLOR 1
30 PRINT#6, AT(5,5);"A"
40 A$=SCRN$(5,5)
50 PRINT TAB(9);A$
60 END
```

11

USER-DEFINED FUNCTION

DEF

Format: DEF function__name (variable,variable) = function__definition
Example: 150 DEF MULT(J,K) = J*K

User-defined functions in the form DEF A(X) = X∧2, where A(X) represents the value of X, squared can be used throughout a program as if they were part of the BASIC language itself. Normally a user-defined function will be placed at the beginning of a program. The user-defined function can occupy no more than a single program line. String-defined functions are allowed. If the defined function is a string__variable__name, then the defined expression must evaluate to a string result. One or more parameters can be defined. Thus, DEF S$(A$,B$) = A$+B$ is legal.

Example Program:

```
100 DEF AVG(X,Y) = (X+Y)/2
120 PRINT AVG(25,35)
130 END
```

RUN RETURN
```
30
```

12 GRAPHICS

GRAPHICS OVERVIEW

The GRAPHICS command selects one of nine graphics modes. Graphics modes are numbered 0 through 8. The arithmetic expression following GRAPHICS must evaluate to a positive integer. Graphics mode 0 is a full-screen text mode. ATARI Microsoft BASIC defaults to GRAPHICS 0.

GRAPHICS 1 through 8 are split-screen modes. In the split-screen modes a 4-line text window is at the bottom of the television screen. The text window is actually 4 lines of GRAPHICS 0 mixed into the mode.

GRAPHICS 0, GRAPHICS 1, and GRAPHICS 2 display text and special characters of gradually increasing size. GRAPHICS 0 is regular text with special characters. GRAPHICS 1 is double-wide text and special characters. GRAPHICS 2 is double-wide, double-high text, and special characters. Note the keyboard representation of the text and special characters as an insert to this manual. The special characters that are not printed on your keyboard are called control characters because you must press the CTRL key to have them display on the television screen.

GRAPHICS 3 through GRAPHICS 8 are modes that plot points directly onto your television screen. The graphics mode dictates the size of the plot points and the number of playfield colors you can use. The maximum number of playfield colors in the point-plotting modes is four. But it is possible to get four more colors on your television screen by using players and missiles. For information on player-missile graphics, see Section 13.

GRAPHICS

Format: GRAPHICS arithmetic_expression
Examples: GRAPHICS 2

```
100 GRAPHICS 5+16
110 GRAPHICS 1+32+16
120 GRAPHICS 8
130 GRAPHICS 0
140 GRAPHICS 18
```

Use GRAPHICS to select one of nine graphics modes (0 through 8). Table 12-2 summarizes the nine modes and characteristics of each. GRAPHICS 0 is a full-screen text display. Characters can be printed in GRAPHICS 0 by using the PRINT statement without an IOCB# following the keyword PRINT. GRAPHICS 1 through GRAPHICS 8 are split-screen modes. These split-screen modes actually mix four lines of GRAPHICS 0 at the bottom of the television screen. This text window uses the PRINT statement. To print in the large graphics window in GRAPHICS 1 and GRAPHICS 2, use *PRINT#6,* . The following program will print in the graphics window in GRAPHICS 1 or GRAPHICS 2:

```
100 GRAPHICS 1
110 PRINT#6, AT(3,3);"GRAPHICS WINDOW TEST"
120 PRINT "TEXT WINDOW"
130 END
```

Adding +16 to GRAPHICS 1 through GRAPHICS 8 will override the text window and make a full screen graphics mode. If you run the following program without line 140, the screen will return to graphics mode 0. The screen returns to graphics mode 0 when STOP or END terminate the full screen graphics mode.

```
110 GRAPHICS 2 + 16
120 PRINT#6, AT(3,3);"WHOLE SCREEN IS"
130 PRINT#6, AT(4,4);"GRAPHICS 2"
140 GOTO 140
```

Normally the screen will be cleared of all previous graphics characters when a GRAPHICS n statement is encountered. Adding + 32 prevents the graphics command from clearing the screen.

Graphics modes 3 through 8 are point-plotting modes. To draw point graphics you need to use the COLOR n and PLOT statements. Use of the SETCOLOR statement will allow you to change the default colors to any one of 128 different color/luminance combinations. Point-plotting modes are explored in the example at the end of this section.

To return to GRAPHICS 0 in direct mode, type **GRAPHICS 0** and press the RETURN key.

COLOR

Format: COLOR n
Example: 100 COLOR 4

COLOR is paired with SETCOLOR to write up to four colors, called playfields, on the television screen. You must have a COLOR statement in GRAPHICS 3, 4, 5, 6, 7, and 8 in order to plot a color. When you use the COLOR statement without a SETCOLOR command you will get the default colors. For example, using Table 12-1, the default colors for GRAPHICS 3 are: SETCOLOR 4 is orange, SETCOLOR 5 is light green, SETCOLOR 6 is dark blue, and SETCOLOR 8 is black.

Shown below are the SETCOLOR - COLOR pairings by graphics mode:

GRAPHICS 3, 5, 7

SETCOLOR 4,hue,lum goes with COLOR 1
SETCOLOR 5,hue,lum goes with COLOR 2
SETCOLOR 6,hue,lum goes with COLOR 3
SETCOLOR 8,hue,lum goes with COLOR 0

GRAPHICS 4, 6

SETCOLOR 4,hue,lum goes with COLOR 1
SETCOLOR 8,hue,lum goes with COLOR 0

GRAPHICS 8

SETCOLOR 5,hue,lum goes with COLOR 1
SETCOLOR 6,hue,lum goes with COLOR 2

Note: You must always have a COLOR statement to plot a playfield point, but SETCOLOR is only necessary to make a color other than a default color.

SETCOLOR

Format: SETCOLOR register,hue,luminance
Example: 330 SETCOLOR 5,4,10

The SETCOLOR statement associates a color and luminance with a register.

register	Color registers 0,1,2,3 are for player-missiles 0,1,2,3 respectively. Color registers 4,5,6,7 are for playfield colors assignments. Register 8 is always the background register.
hue	Color hue number 0-15. (See table below.)
luminance	Color luminance (must be an even number between 0 and 14; the higher the number, the brighter the display; 14 is almost pure white).

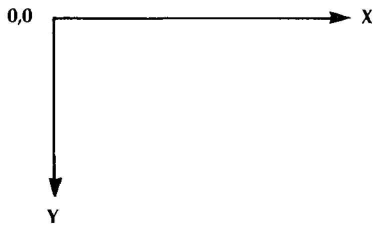
TABLE 12-1
THE ATARI HUE (SETCOLOR COMMAND) NUMBERS AND COLORS

Colors	SETCOLOR Hue Number Decimal	SETCOLOR Hue Number Hex
Gray	0	0
Light orange (gold)	1	1
Orange	2	2
Red-orange	3	3
Pink	4	4
Purple	5	5
Purple-blue	6	6
Azure blue	7	7
Sky blue	8	8
Light blue	9	9
Turquoise	10	A
Green-blue	11	B
Green	12	C
Yellow-green	13	D
Orange-green	14	E
Light orange	15	F

PLOT

Formats: PLOT X,Y
PLOT X,Y TO PLOT X,Y
Examples: 100 PLOT 12,9
112 PLOT 6,9 TO 3,3

Use PLOT to draw single-point plots, lines, and outline objects on the television screen. PLOT uses an X-Y coordinate system for specifying individual plot points. Give a number from 0 to whatever the maximum is for the current mode, X first then Y.



You can "chain" the PLOT instruction. That is, one plot point can be made to draw to the next plot point. The result of chaining two PLOT points is a straight line. It is also easy to outline an object using chained plots. To chain plots, use the word TO between PLOT X,Y's.

Example: 90 COLOR 1 !You must use a COLOR instruction before PLOT
100 PLOT 5,5 TO 5,15 !Draws a straight line
120 PLOT 5,5 TO 12,12 TO 2,12 TO 5,5 !Draws triangle outline

Here is an example program which shows PLOT, COLOR, and SETCOLOR at work:

```
100 GRAPHICS 3+16 !THE 16 GETS RID OF TEXT WINDOW
110 SETCOLOR 5,4,8 !PINK
120 SETCOLOR 6,0,4 !GRAY
130 SETCOLOR 8,8,6 !BLUE
140 COLOR 1 !COLOR 1 GOES WITH DEFAULT ORANGE
150 PLOT 5,5 TO 10,5 TO 10,10 TO 5,10 TO 5,5 !IN ORANGE
160 COLOR 2 ! PINK
170 PLOT 7,7 TO 12,12 TO 2,12 TO 7,7
180 COLOR 3 !GRAY
190 PLOT 2,7 TO 12,7
200 GOTO 200
```

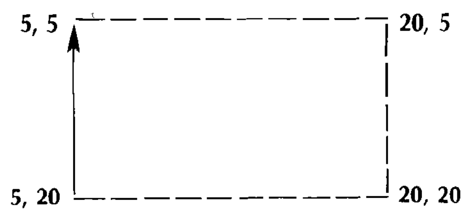
FILL

Format: FILL x,y TO x,y
Example: 550 FILL 10,10 TO 5,5

FILL fills an area with the color specified by the COLOR and SETCOLOR statements. The FILL process sweeps across the television screen from left to right. FILL stops painting and starts its next sweep when it bumps into a PLOT line or point. The line on the left-hand side of a filled object is specified by the FILL statement itself.

An example will show how FILL operates. First the outline of three sides of a box are specified. PLOT 5,5 TO 20,5 TO 20,20 TO 5,20 makes the top, right side, and bottom of the box. Make the left side and FILL with the statement FILL 5,5 TO 5,20.

Example:



The top, right, and bottom of the box (dashed lines) is formed with PLOT 5,5 TO 20,5 TO 20,20 TO 5,20. The box is filled with the statement FILL 5,5 TO 5,20.

```
10 GRAPHICS 5
20 SETCOLOR 4,12,8 !Register 4, green, medium brightness
30 COLOR 1 !COLOR 1 is paired with SETCOLOR 4 in GRAPHICS 5
40 PLOT 5,5 TO 20,5 TO 20,20 TO 5,20
50 FILL 5,5 TO 5,20
60 END
```

It is worthwhile to carefully review the FILL process. Line 40 in the above example makes three sides of a box. Then the FILL statement, line 50 draws the left side and fills the box. The FILL process scans from the FILL line to the right until it reaches the PLOT lines.

CLS

Format: CLS |background_register_option|
Example: CLS

```
110 CLS
220 GRAPHICS 3: CLS &C5
330 CLS 25
```

CLS clears screen text areas and sets the background color register to the indicated value, if present. In GRAPHICS 0 and GRAPHICS 8 the optional number after CLS determines the border color and luminance. In GRAPHICS 1, 2, 3, 4, 5, 6, 7 the optional number following CLS determines the background color and luminance.

TABLE 12-2
GRAPHICS MODES AND SCREEN FORMATS

Graphics Mode	Mode Type	Columns	ROWS-Split Screen	ROWS-Full Screen	Number of Colors	RAM Required (Bytes)
0	TEXT	40	-	24	2	992
1	TEXT	20	20	24	5	674
2	TEXT	20	10	12	5	424
3	GRAPHICS	40	20	24	4	434
4	GRAPHICS	80	40	48	2	694
5	GRAPHICS	80	40	48	4	1174
6	GRAPHICS	160	80	96	2	2174
7	GRAPHICS	160	80	96	4	4198
8	GRAPHICS	320	160	192	½	8112

TABLE 12-3
CHARACTERS IN GRAPHICS MODE 1 AND 2

POKE 756,224	POKE 756,226	SETCOLOR 4	SETCOLOR 5	SETCOLOR 6	SETCOLOR 7
	♥	32	0	160	128
!	├	33	1	161	129
"	▕	34	2	162	130
#	┘	35	3	163	131
$	┤	36	4	164	132
%	┐	37	5	165	133
&	╱	38	6	166	134
'	╲	39	7	167	135
(	◢	40	8	168	136
)	▗	41	9	169	137
*	◣	42	10	170	138
+	▝	43	11	171	139
,	▘	44	12	172	140
-	▔	45	13	173	141
.	▁	46	14	174	142
/	▖	47	15	175	143
0	♣	48	16	176	144
1	┌	49	17	177	145
2	─	50	18	178	146
3	┼	51	19	179	147
4	●	52	20	180	148
5	▄	53	21	181	149
6	▎	54	22	182	150
7	┬	55	23	183	151
8	┴	56	24	184	152
9	▌	57	25	185	153
:	└	58	26	186	154
;	␛	59	27	187	155
<	↑	60	28	188	156
=	↓	61	29	189	167
>	←	62	30	190	168

?	→	63	31	191	169
@	♦	64	96	192	224
A	a	65	97	193	225
B	b	66	98	194	226
C	c	67	99	195	227
D	d	68	100	196	228
E	e	69	101	197	229
F	f	70	102	198	230
G	g	71	103	199	231
H	h	72	104	200	232
I	i	73	105	201	233
J	j	74	106	202	234
K	k	75	107	203	235
L	l	76	108	204	236
M	m	77	109	205	237
N	n	78	110	206	238
O	o	79	111	207	239
P	p	80	112	208	240
Q	q	81	113	209	241
R	r	82	114	210	242
S	s	83	115	211	243
T	t	84	116	212	244
U	u	85	117	213	245
V	v	86	118	214	246
W	w	87	119	215	247
X	x	88	120	216	248
Y	y	89	121	217	249
Z	z	90	122	218	250
[	♣	91	123	219	251
\	▌	92	124	220	252
]	↖	93	125	221	253
^	◀	94	126	222	254
_	▶	95	127	223	255

The following short program demonstrates and confirms Table 12-3. This program prints the ATASCII code for a character in the text window and the character itself in the graphics window. Every time you press the RETURN key, a new character appears. The reason SETCOLOR 4,0,0 is the same as SETCOLOR 8,0,0 is to avoid a screen filled with hearts. Another way to accomplish this is to lower the character set into RAM (using MOVE) and redefine the heart character as 8 by 8 zeros. See Appendix C, Alternate Character Sets, for an example of lowering and redefining the character set. The special character set is shown in the program as it is now written. To see the standard character set, just delete line 20. The GRAPHICS 2 instruction automatically pokes 756,224.

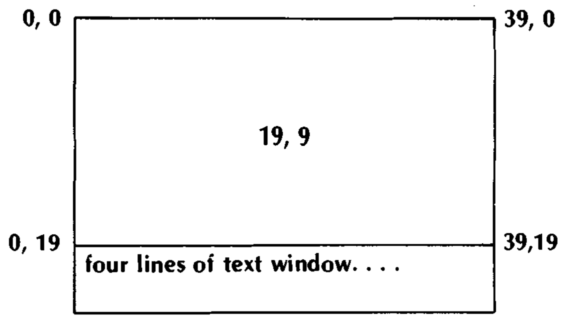
```
10 GRAPHICS 2
20 POKE 756,226
30 SETCOLOR 8,0,0
40 SETCOLOR 4,0,0!AVOID SCREEN HEARTS
50 SETCOLOR 5,4,6!PINK
60 SETCOLOR 6,12,2!GREEN+TEXT WINDOW
70 SETCOLOR 7,9,6!LIGHT BLUE
80 A$=INKEY$
90 IF A$="" THEN 80
100 ON ERROR GOTO 150
110 PRINT #6, AT(6,6);CHR$(X)
120 PRINT X
130 X=X+1
140 GOTO 80
150 RUN !REPEATS WHEN 256 REACHED
```

POINT-PLOTTING MODES

GRAPHICS 3 through 8 plot individual points on your television screen. The number following GRAPHICS determines the size of the points you plot. GRAPHICS 3 has the largest plot points. The following program can be used in GRAPHICS 3 through 8 by changing line number 10 to the appropriate graphics number. Note that you must include line 20 since it indicates that you are using COLOR 1 as a default (see Table 12-4 for default colors).

```
10 GRAPHICS 3 !CAN BE GRAPHICS 3 THROUGH 8
20 COLOR 1 !YOU WANT DEFAULT COLOR — ORANGE
30 PRINT "TYPE TWO NUMBERS — SEPARATE THE TWO"
40 PRINT "NUMBERS WITH A COMMA"
50 PRINT "PLOT X,Y"
60 INPUT X,Y
70 PLOT X,Y
80 GOTO 30
```

If you enter and run the above program you will see plot point 5,5 by typing **5,5** and pressing the RETURN key. The boundaries and middle of GRAPHICS 3 are as follows.



If you insert a new statement — statement 15 — 15 SETCOLOR 4,4,8 you will get large, pink dots instead of the default orange. This change to the original plotting program gives you pink plot points because SETCOLOR 4,x,x aligns with COLOR 1 in GRAPHICS 3. You can also make the text window at the bottom of the screen go away by changing statement 10 to 10 GRAPHICS 3+16.

TABLE 12-4
DEFAULT COLORS, MODE, SETCOLOR, AND COLOR

Default Colors	**Mode or Condition**	**Setcolor Register**	**Color n**	**Description and Comments**
	GRAPHICS 0	4	Register	
Light blue		5	holds	Character luminance
Dark blue		6	character	(same as background)
		7		Character
Black	**Text Mode**	8	Border	
Orange		4		Character
Light green	GRAPHICS 1,2	5		Character
Dark blue		6		Character
Red		7		Character
Black	**Text Modes**	8		Character
				Background, border
Orange		4	1	Graphics point
Light green	GRAPHICS 3,5,7	5	2	Graphics point
Dark blue		6	3	Graphics point
		7	-	— — —
Black	**4-color modes**	8	0	Background, border
Orange	GRAPHICS	4	1	Graphics point
	4 and 6	5	-	— — —
		6	-	— — —
		7	-	— — —
Black	**2-color modes**	8	0	Background, border
	GRAPHICS 8	4	-	— — —
Light blue		5	1	— — —
Dark blue		6	2	— — —
		7	-	— — —
Black	**1 color,2 lums.**	8	-	Border

Note: Player-missile graphics color is SETCOLOR register, color, luminance, where register=0,1,2,3 and determines color of player-missile 0,1,2,3, respectively. Player-missile graphics will work in all graphics modes.

The following programs will work in GRAPHICS 1 or GRAPHICS 2. The programs show the alternate basic character set and special character set (POKE 756,226). To restart these two programs, press the BREAK key and type **RUN** followed by RETURN.

```
2 REM KEYBOARD TYPEWRITER
10 GRAPHICS 2
20 SETCOLOR 4,0,0!to avoid screen full of hearts in lowercase
30 PRINT "TYPE Green/Blue/Red (G/B/R)"
40 INPUT "AND PRESS RETURN? "; C$
50 IF C$="G" THEN K=32
60 IF C$="B" THEN K=128
70 IF C$="R" THEN K=160
80 PRINT "TYPE UPPER/LOWER (U/L)"
90 INPUT "AND PRESS RETURN ? "; B$
100 IF B$="U" THEN 120
110 POKE 756,226
120 PRINT "NOW TYPE — ALPHA + CTRL KEYS"
130 A$=INKEY$
140 IF A$="" THEN 130
150 A=ASC(A$)+K!32 is green, 128 is blue, 160 is red
160 PRINT A
170 PRINT#6, CHR$(A);
180 GOTO 130
```

```
100 REM TWINKLE
110 GRAPHICS 16+2
120 X=RND(36)
130 ON ERROR GOTO 150
140 PRINT#6, TAB(X);"*"
150 GRAPHICS 32+16+2
160 RESUME
```

13 PLAYER-MISSILE GRAPHICS

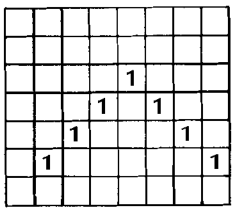
PLAYERS AND MISSILES

The following BASIC commands are tools to help you construct and move players and missiles:

MOVE instruction
OPTION (PLM1 or PLM2)
VARPTR (PLM1 or PLM2)
SETCOLOR 0 or 1 or 2 or 3

MAKING A PLAYER OUT OF PAPER

Cut a strip of paper about 2 inches wide from an 8 x 10 inch sheet of paper. Now draw an 8-bit-wide "byte" down the strip of paper.



Hex &08 drawn on 8-bit strip.
Hex &14 drawn on 8-bit strip.
Hex &22 drawn on 8-bit strip.
Hex &41 drawn on 8-bit strip.

An upside down V is shown on the strip in binary and hex. This strip of paper is like a player. If you take the player strip and lay it vertically down the middle of the television screen, you have "positioned it with the horizontal position register." When you move the strip right and left, you are "poking new locations into the horizontal position register" to get that movement.

The MOVE instruction is used to move the player-missile object up and down the player-missile strip. Your paper strip can serve to demonstrate how the MOVE instruction works. Let's say that you have put the upside down V on your paper strip with a pencil that has an eraser. To move the object it is necessary for you to erase the whole object and rewrite it elsewhere on the strip.

As you can imagine, vertical movement is slightly slower than horizontal movement. It is slower because it takes only a single poke to the horizontal position register for horizontal movement, but many erasures and redrawings are necessary to move an object vertically.

In the actual MOVE instruction you state the lowest address of the object you want to move; then state the lowest address of the new area to which you want to move the object; and lastly, state how many bytes you want moved. Hence the format: MOVE from__address, to__address, no.__of__bytes.

HOW THE ATARI MICROSOFT BASIC INSTRUCTIONS ASSIST PLAYER-MISSILE GRAPHICS

The OPTION (PLM1) zeros out and dedicates a single-line resolution player-missile area in RAM. OPTION (PLM2) is for double-line player-missile resolution.

VARPTR(PLM1 or PLM2) points to the beginning memory location of the player-missile area in RAM. This is the point from which you must figure your offset or displacement to poke your image into the correct area. For example, the starting address (top of television screen) for player 0 in double-line resolution is VARPTR(PLM2)+128. In double-line resolution each player is 128 bytes long. So if you wanted to poke a straight line in the middle of player 0, the poke would be POKE VARPTR(PLM2)+192,&FF.

The SETCOLOR instruction gives the register, color, and luminance assignments. In ATARI Microsoft BASIC the **registers** 0, 1, 2, and 3 are used for player-missiles 0, 1, 2, and 3. It is only necessary to specify SETCOLOR 0,5,10 to set player-missile 0; the COLOR instruction is not used.

Remember that you must poke decimal location 559 with decimal 62 for single-line resolution or with decimal 46 for double-line resolution. You must also poke decimal location 53277 with decimal 3 to enable player-missile display.

You can use player-missile graphics in all modes. Missiles consist of 2-bit-wide "strips." Missiles 0, 1, 2, 3 are assigned the same colors as their associated player. Thus, when SETCOLOR sets the color of player 1 to red, it also sets missile 1 to red.

The terms *player* and *missile* are derived from the animated graphics used in ATARI video games. Player-missile binary tables reside in player-missile graphics RAM. This RAM accommodates four 8-bit players and four 2-bit missiles (see Figure 13-1). Each missile is associated with a player, unless you elect to combine all missiles to form a fifth, independent player (see "Priority Control").

A player, like the spaceship shown in Figure 13-2, is displayed by mapping its binary table directly onto the television screen, on top of the playfield. The first byte in the table is mapped onto the top line of the screen, the second byte onto the second line, and so forth. Wherever 1's appear in the table, the screen pixels turn on; wherever 0's appear, the pixels remain off. The pattern of light and dark pixels creates the image.

You can display player-missile graphics with single-line resolution (use OPTION(PLM1)) or double-line resolution (OPTION(PLM2)). If you select single-line resolution, each byte of the player will be displayed on a single scan line. If you choose double-line resolution, each byte will occupy two scan lines and the player will appear larger than in single-line resolution. Each player is 256 bytes long with single-line resolution, or 128 bytes long with double-line resolution. Line resolution only needs to be programmed once. The resolution you choose will apply to all player-missile graphics in your program. The Player-Missile Graphics Demonstration Program included in this section is an example of double-line resolution programming.

Player-missile graphics give you considerable flexibility in programming animated video graphics. Registers are provided for player-missile color, size, horizontal positioning, player-playfield priority, and collision control.

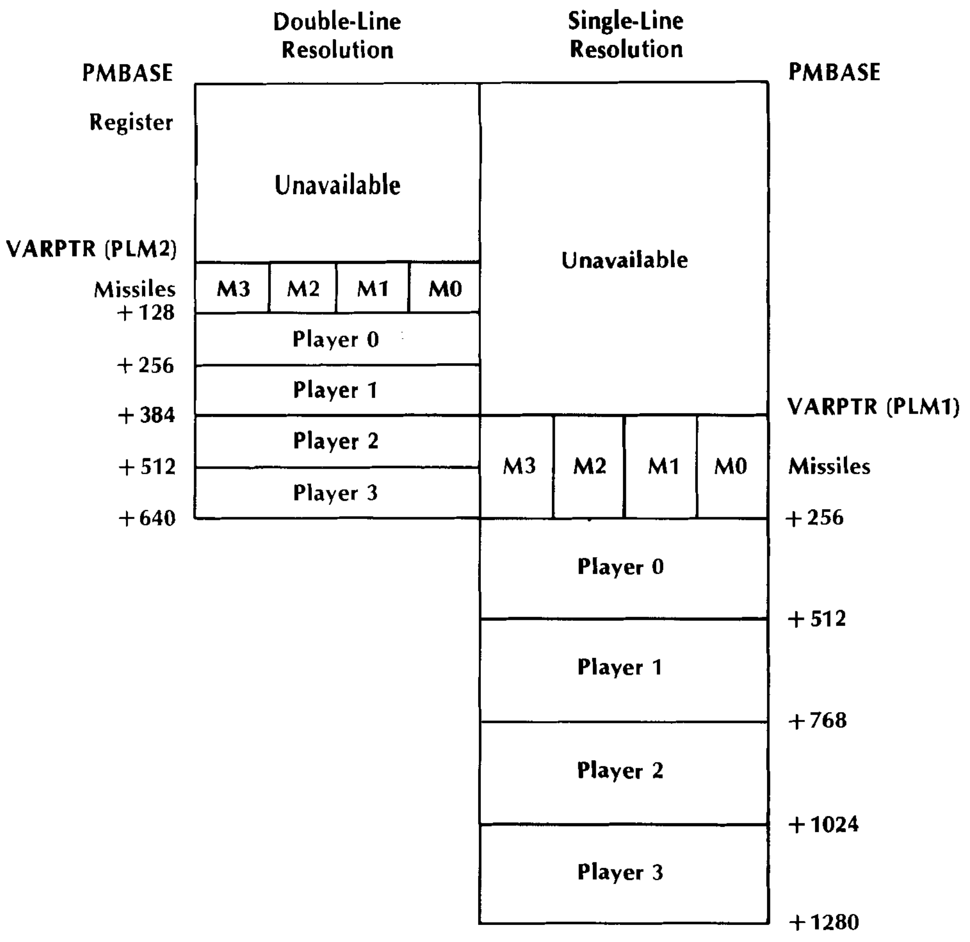


Figure 13-1 Player-Missile Graphics RAM Configuration

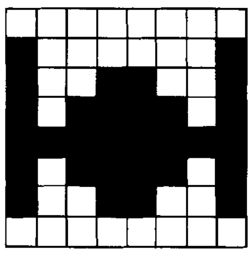
GRAPHIC REPRESENTATION	BINARY REPRESENTATION	HEXADECIMAL REPRESENTATION	DECIMAL REPRESENTATION
	00000000	0	0
	10000001	81	129
	10011001	99	153
	10111101	BD	189
	11111111	FF	255
	10111101	BD	189
	10011001	99	153
	00000000	0	0

Figure 13-2 Mapping the Player

COLOR CONTROL

The ATARI 400 and ATARI 800 Computers have nine registers for user control of player-missile, playfield, and background color:

TABLE 13-1
SETCOLOR REGISTER ASSIGNMENTS

SETCOLOR Register,Color,Luminance	Function
SETCOLOR 0,color,luminance	Color-luminance of Player-Missile 0
SETCOLOR 1,color,luminance	Color-luminance of Player-Missile 1
SETCOLOR 2,color,luminance	Color-luminance of Player-Missile 2
SETCOLOR 3,color,luminance	Color-luminance of Player-Missile 3
SETCOLOR 4,color,luminance	Color-luminance of Playfield 0
SETCOLOR 5,color,luminance	Color-luminance of Playfield 1
SETCOLOR 6,color,luminance	Color-luminance of Playfield 2
SETCOLOR 7,color,luminance	Color-luminance of Playfield 3
SETCOLOR 8,color,luminance	Color-luminance of background

Players are completely independent of the playfield and of each other. Missiles share color registers with their players and hence are the same color as their players. If you combine missiles to form a fifth player, they assume the color of playfield color-luminance register 3 (COLPF3).

To program color, specify the register, the hue, and the luminance. Use the SETCOLOR command. See lines 20 and 100 of the Player-Missile Graphics Demonstration Program for examples. See also "Graphics," Section 12.

Each color-luminance register is independent. Therefore, you could use as many as nine different colors in a program, depending upon the graphics mode selected. All registers cannot be used in all graphics modes (see "Graphics," Section 12).

SIZE CONTROL

Five size-control registers are provided—four for the players and one for all four missiles:

TABLE 13-2
REGISTERS CONTROLLING WIDTH OF PLAYER-MISSILES

Size	Address		
Register	Hex	Dec	Function
SIZEP0	D008	53256	Controls size of Player 0
SIZEP1	D009	53257	Controls size of Player 1
SIZEP2	D00A	53258	Controls size of Player 2
SIZEP3	D00B	53259	Controls size of Player 3
SIZEM	D00C	53260	Controls size of missiles

Size-control registers allow you to double or quadruple the width of a player or missile without altering its bit resolution. To double the width, poke a 1 into the size register; to quadruple the width, poke a 3; and to return a player or missile to normal size, poke a 0 or 2. An example is given in line 80 of the Player-Missile Graphics Demonstration Program.

POSITION AND MOVEMENT

VERTICAL

Vertical position is set when you specify the location of the player-missile in player-missile graphics RAM. The lower you place the player-missile in RAM, the higher the image will be on the television screen. A positioning technique is illustrated by lines 120 and 200 of the Player-Missile Graphics Demonstration Program at the end of this section.

To program vertical motion, use the MOVE command (see lines 350 and 390 of the Player-Missile Graphics Demonstration Program). Since the MOVE command does not zero the old location after the move, an extra zero at each end of the player is used to "cleanup" as the player is being moved. Give the current position of the player in RAM, the direction of the move through RAM (forward = +, backward = -), and the number of player bytes to be moved. Each byte of the player must be moved. Following the MOVE command, increment or decrement the vertical position counter (see lines 360 and 400 of the Player-Missile Graphics Demonstration Program).

HORIZONTAL

Each player and missile has its own horizontal position register, so players can move independently of each other, and missiles can move independently of their players.

TABLE 13-3
PLAYER-MISSILE HORIZONTAL POSITION REGISTERS

Position Register	Address Hex	Dec	Function
HPOSP0	D000	53248	Horizontal position of Player 0
HPOSP1	D001	53249	Horizontal position of Player 1
HPOSP2	D002	53250	Horizontal position of Player 2
HPOSP3	D003	53251	Horizontal position of Player 3
HPOSM0	D004	53252	Horizontal position of Missile 0
HPOSM1	D005	53253	Horizontal position of Missile 1
HPOSM2	D006	53254	Horizontal position of Missile 2
HPOSM3	D007	53255	Horizontal position of Missile 3

To set the position of a player or missile, poke its horizontal position register with the number of the position. To program horizontal movement, simply change the number stored in the register. See lines 100 and 180 of the Player-Missile Graphics Demonstration Program for examples.

A horizontal position register can hold 256 positions, but some of these are off the left or right margin of the television screen. A conservative estimate of the range of player visibility is horizontal positions 60 through 200. The actual range will depend upon the television set.

DIAGONAL

Horizontal and vertical moves can be combined to move the player diagonally. Set the horizontal position first, then the vertical position. See lines 270 through 390 of the Player-Missile Graphics Demonstration Program.

PRIORITY CONTROL

The Priority Control Register (PRIOR,&D01B; OS shadow GPRIOR,&26F) enables you to select player or playfield color register priority and to combine missiles to form a fifth player.

PRIORITY SELECT

You have the option to specify which image will have priority in the event player and playfield images overlap. This feature enables you to make players disappear behind the playfield and vice versa. To set the priority, poke one of the following numbers into the Priority Control Register:

1 = All players have priority over all playfields.
2 = Players 0 and 1 have priority over all playfields, and all playfields have priority over players 2 and 3.
4 = All playfields have priority over all players.
8 = Playfields 0 and 1 have priority over all players, and all players have priority over playfields 2 and 3.

ENABLE FIFTH PLAYER

Setting bit D4 of the Priority Control Register causes all missiles to assume the color of Playfield Register 3 (&2C7, dec. 711). You can then combine the missiles to form a fifth player. If enabled, the fifth player must still be moved horizontally by changing all missile registers (&D004 through &D007) together.

COLLISION CONTROL

Collision control enables you to tell when a player or missile has collided with another graphics object. There are 16 collision control registers.

TABLE 13-4
COLLISION CONTROL REGISTERS FOR PLAYER-MISSILES

Collision Register	Address Hex	Dec	Function
M0PF	D000	53248	Missile 0 to playfield
M1PF	D001	53249	Missile 1 to playfield
M2PF	D002	53250	Missile 2 to playfield
M3PF	D003	53251	Missile 3 to playfield
P0PF	D004	53252	Player 0 to playfield
P1PF	D005	53253	Player 1 to playfield
P2PF	D006	53254	Player 2 to playfield
P3PF	D007	53255	Player 3 to playfield
M0PL	D008	53256	Missile 0 to player
M1PL	D009	53257	Missile 1 to player
M2PL	D00A	53258	Missile 2 to player
M3PL	D00B	53259	Missile 3 to player
P0PL	D00C	53260	Player 0 to player
P1PL	D00D	53261	Player 1 to player
P2PL	D00E	53262	Player 2 to player
P3PL	D00F	53263	Player 3 to player

In each case, only the rightmost 4 bits of each register are used. They are numbered 0, 1, 2, and 3 from the right and designate, by position, which playfield or player the relevant player or missile has collided with. A one in any bit position indicates collision since the last HITCLR.

CLEARING COLLISION REGISTERS

All collision registers are cleared at once by writing a zero to the HITCLR register (&D01E, dec. 53278).

PLAYER-MISSILE GRAPHICS DEMONSTRATION PROGRAM

The following ATARI Microsoft BASIC program creates a player (spaceship) that shoots missiles and can be moved in all directions with the joystick. Connect a joystick controller to CONNECTOR JACK 1 on the front of your ATARI Home Computer.

```
05 !DOUBLE-LINE RESOLUTION PLAYER AND MISSILE
10 GRAPHICS 8
20 SETCOLOR 6,0,0
30 X = 130
40 Y = 70
50 STICK0 = &278
60 OPTION PLM2
70 POKE 559,46
80 POKE &D00C,1
90 POKE &D01D,3
100 POKE &D000,X
110 SETCOLOR 0,3,10
120 FOR J = VARPTR(PLM2)+128+Y TO VARPTR(PLM2)+135+Y:READ A:POKE
J,A
125 NEXT J
130 DATA 0,129,153,189,255,189,153,0
140 IF PEEK(&D010) = 1 THEN 220
150 SOUND 0,220,12,15,INT(X/30)
160 ZAP = X
170 POKE VARPTR(PLM2)+4+Y,3
180 POKE &D004,ZAP
190 ZAP = ZAP-12
200 IF ZAP <12 THEN POKE VARPTR(PLM2)+4+Y,0:GOTO 220 ELSE 180
210 !JOYSTICK MOVES
220 A = PEEK(STICK0): IF A = 15 THEN GOTO 140
230 IF A = 11 THEN X = X-1
240 IF A = 7 THEN X = X+1
250 POKE &D000,X
260 IF A = 14 THEN GOTO 350 !MOVE UP
270 IF A = 13 THEN GOTO 390 !MOVE DOWN
280 !MOVE DIAGONALLY
290 IF A =10 THEN X = X-1:POKE &D000,X:GOTO 350
300 IF A = 6 THEN X = X+1:POKE &D000,X:GOTO 350
310 IF A = 9 THEN X = X-1:POKE &D000,X:GOTO 390
320 IF A = 5 THEN X = X+1:POKE &D000,X:GOTO 390
330 GOTO 140
340 !MOVE UP
350 MOVE VARPTR(PLM2)+128+Y,VARPTR(PLM2)+128+(Y-1),8
```

```
360 Y = Y-1
370 GOTO 140
380 !MOVE DOWN
390 MOVE VARPTR(PLM2)+128+(Y-1),VARPTR(PLM2)+128+Y,8
400 Y = Y+1
410 GOTO 140
420 STOP
430 END
```

ANNOTATION

Line	
10	Sets a high-resolution graphics mode with no text window. You can program player-missile graphics in any graphics mode. See Section 12, "Graphics" and Table 12-4.

20	Sets the background color to black, as follows:
	6 = Background Color-Luminance Register (COLBK, &D01A);
	0 = Black (see Color Table 12-1);
	0 = Zero luminance. The luminance value is an even number between 0 and 14. The higher the number, the greater the luminance and the brighter the color.

30,40	Initializes player position variables X (horizontal) and Y (vertical).
50	Assigns the label STICK0 to joystick register 278.
60	Specifies double-line resolution RAM for the player-missile graphics (see Figure 13-1). PLM1 would specify single-line resolution.
70	Sets the Direct Memory Access Control Register (DMACTL, 559) for double-line resolution (46). A 62 would specify single-line resolution. **Note** When DMACTL is enabled, the player-missile graphics registers (GRAFP0-GRAFP3 and GRAFM) are automatically loaded with data from the player-missile RAM.
80	Doubles the width of the missile by poking the Size Control Register (SIZEM, &D00C) with 1. Poking the register with a 3 would quadruple the width.
90	Enables the Graphics Control Register (GRACTL, &D01D) to display player-missile graphics (3 enables, 0 disables).
100	Pokes the horizontal position of the player (X = 130 from line 30) into the player 0 Horizontal Position Register (HPOSP0, &D000).

110	Colors the player and missile bright red-orange as follows:
	0 = Player-missile 0 Color-Luminance Register (COLPM0, &D012);
	3 = Red-orange (see Color Table 12-1);
	10 = Luminance or brightness (see annotation of line 20).

120-125	Sets variable pointer VARPTR(PLM2) to the player-missile starting address in player-missile graphics RAM (see Figure 13-1). Pokes data from line 130 into the player area, VARPTR(PLM2)+128+Y to VARPTR(PLM2)+135+Y. The computer uses the data in line 130 to map the spaceship onto the screen (see Figure 13-2).
140	Tells the computer to read the joystick 0 trigger register (TRIG0, &D010). If the trigger button is not activated (&D010 = 1), the computer will go to line 220 and read the joystick position; if the button is activated (&D010 = 0), the computer will execute lines 150 through 200.
150	Generates sound each time the joystick button is pressed. Sound is programmed as follows: (1) Select voice. As many as four voices (0 to 3) can be used, but each voice requires a separate SOUND statement. (2) Choose pitch from Table 14-1. The larger the number, the lower the pitch. (3) Set distortion or noise level, using an even number between 0 and 14. A 10 gives a pure tone; 12 gives a buzzer effect. (4) Set volume, an odd number between 1 and 15. The larger the number, the louder the sound. (5) Set duration of sound per second (20 = 20/60 or ⅓ second).
160	Sets the horizontal position of the missile (ZAP) equal to the horizontal position of the player (X).
170	Turns on the screen pixels corresponding to the missile 0 RAM area [VARPTR(PLM2)+4+Y] to display the missile (3 = ON; 0 = OFF).
180	Pokes the horizontal position of the missile (ZAP = X from line 160) into the missile 0 horizontal position register (HPOSM0, &D004).
190	Decrements the missile 0 horizontal position counter by 12 to create a horizontal "line of fire" from the player.
200	If the missile's horizontal position is less than 12 (off the left side of the screen), the computer pokes 0's into the missile RAM area to clear it and goes to line 220. If the missile's horizontal position is 12 or greater, the computer pokes the new hrizontal position into HPOSM0 (register &D004 in line 180) and decrements the horizontal position counter by 12 (line 190).
220	Tells the computer to read the STICK0 register and find the position of the joystick (see Figure 13-3). If the position is 15 (neutral), the computer goes to line 140 and reads the joystick trigger register (&D010).
230/250	If the joystick is moved left (11), the computer decrements the horizontal position counter and pokes the spaceship's new horizontal position into the HPOSP0 register (&D000).

240/250	If the joystick is moved right (7), the computer increments the horizontal position counter and pokes the spaceship's new horizontal position into HPOSP0.
260	If the joystick is moved up (14), the computer moves the spaceship back one byte in player-missile RAM (line 350). Each of the 8 bytes that comprise the spaceship must be moved back. When the move is completed, the computer decrements the vertical position counter (line 360).
270	If the joystick is moved down (13), the computer advances the spaceship one byte in player-missile RAM (line 390) and increments the vertical position counter (line 400).
290 - 320	If the joystick is moved diagonally (10, 6, 9, or 5), the computer executes a horizontal move (after resetting the horizontal position register), makes a vertical move (line 350 or 390), and resets the vertical position counter (line 360 or 400).

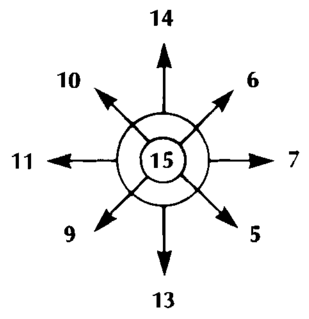


Figure 13-3 Joystick Controller Positions

14 SOUND

SOUND

Format: SOUND voice, frequency, distortion, volume, duration
Examples: 120 SOUND 2,204,10,12,244
100 SOUND 0,122,8,10

Voice. There can be up to four voices specified by the numbers 0 through 3.

Frequency. From 0-255 (see Frequency Chart, Table 14-1).

Distortion. The default is a pure tone. Even numbers between 0 and 14 define the distortion. A 10 is used to create a "pure" tone. A 12 gives a buzzer sound.

Volume. A number between 0 and 15. Use a 1 to create a sound that is barely audible. Use a 15 to make a loud sound. A value of 8 is considered normal. If more than one sound statement is being used, the total volume should not exceed 32. This will create an unpleasant "clipped" tone.

Duration. Duration is given in 1/60 of a second. The duration indicates how long a tone or noise will last. If you do not specify a number for the duration parameter, the tone will continue until the program reaches an END statement, another RUN statement, or until you type a second SOUND statement using the same voice number followed by 0,0,0. You can also stop the tone by pressing the BREAK key.

Example: SOUND 2,204,10,12
SOUND 2,0,0,0

TABLE 14-1
FREQUENCY CHART OF PITCH VALUES

	Notes	Hex	Decimal
HIGH NOTES	C	1D	29
	B	1F	31
	A# or B♭	21	33
	A	23	35
	G# or A♭	25	37
	G	28	40
	F# or G♭	2A	42
	F	2D	45
	E	2F	47
	D# or E	32	50
	D	35	53
	C# or D♭	39	57
	C	3C	60
	B	40	64
	A# or B	44	68
	A	4B	72
	G# or A♭	4C	76
	G	51	81
	F# or G♭	55	85
	F	5B	91
	E	60	96
	D# or E♭	66	102
	D	6C	108
	C# or D♭	72	114
MIDDLE C	C	79	121
	B	80	128
	A# or B♭	88	136
	A	90	144
	G# or A♭	99	153
	G	A2	162
	F# or G♭	AD	173
	F	B6	182
LOW NOTES	E	C1	193
	D# or E♭	CC	204
	D	D9	217
	C# or D♭	E6	230
	C	F3	243

Example Program:

NIGHT LAUNCH

```
10 GRAPHICS 2+16
20 SETCOLOR 4,8,4
30 PRINT#6, AT(3,3);"NIGHT LAUNCH"
40 FOR DELAY=1 TO 1000:NEXT
50 GRAPHICS 2+16
60 PRINT#6, AT(3,3);"AT THE CAPE"
70 FOR DELAY=1 TO 1000:NEXT
80 GRAPHICS 0
90 POKE 752,1
100 SETCOLOR 6,0,0
110 FOR T=1 TO 24:PRINT "":NEXT
120 PRINT TAB(11);CHR$(8);CHR$(10)
130 PRINT TAB(11);CHR$(22);CHR$(2)
140 PRINT TAB(11);CHR$(22);CHR$(2)
150 PRINT TAB(11);CHR$(13);CHR$(13)
160 PRINT TAB(11);CHR$(6);CHR$(7)
170 FOR VOL=15 TO 0 STEP -1
180 SOUND 2,77,8,VOL
190 PRINT CHR$(155)!MOVES ROCKET UP
200 FOR R=1 TO 200:NEXT R
210 NEXT VOL
220 END
```

The above program is a demonstration of the SOUND statement. It decreases (by a loop) the volume of a distorted sound. The sound effect resembles a rocket taking off into outer space.

15

GAME CONTROLLERS

In ATARI Microsoft BASIC, the game controllers are sensed with the PEEK instruction. The controllers are attached directly to the four controller jacks in the front of the ATARI Home Computer. The PEEK locations can be given the same names listed below or you can give them short variable names. A complete list of PEEK locations is given in Appendix E.

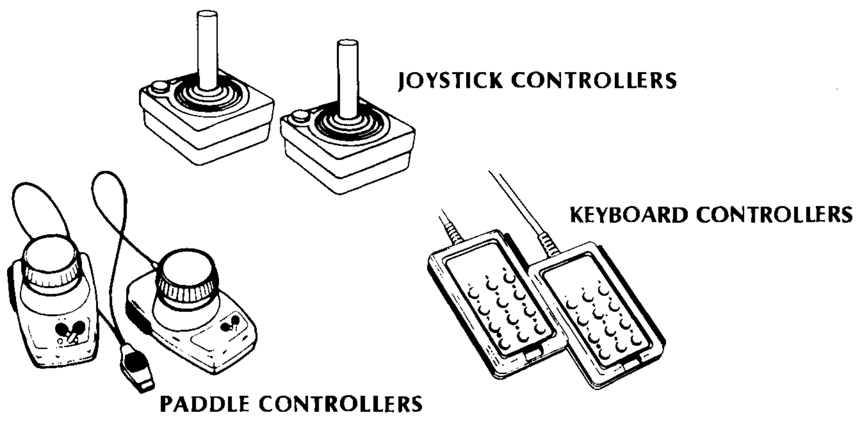


Figure 15-1 Game Controllers

PADDLE CONTROLLERS

The following example program senses and prints the status of paddle controller 0 (first paddle in leftmost port). This PEEK can be used with other functions or commands to "cause" further actions like sound, graphics controls, etc. An example is the statement IF PADDLE(0)>14 THEN GOTO 440. Peeking the paddle address returns a number between 1 and 228, with the number increasing in size as the knob on the controller is rotated counterclockwise (turned to the left).

Example of initializing and using PEEK for PADDLE(0):

```
10 PADDLE(0)=624
20 PRINT PEEK(PADDLE(0))
30 GOTO 20
```

PADDLE number and PEEK locations (decimal addresses):

```
PADDLE(0)=624
PADDLE(1)=625
PADDLE(2)=626
PADDLE(3)=627
PADDLE(4)=628
PADDLE(5)=629
PADDLE(6)=630
PADDLE(7)=631
```

KEYBOARD CONTROLLERS

Peeking the following addresses returns a status of 0 if you press the trigger button of the designated controller. Otherwise, it returns a value of 1.

Example of using paddle trigger (0):

```
10 PTRIG(0)=&27C
20 PRINT PEEK(PTRIG(0))
30 GOTO 20
```

PTRIG (paddle trigger) number and PEEK locations (decimal):

```
PTRIG(0)=636
PTRIG(1)=637
PTRIG(2)=638
PTRIG(3)=649
PTRIG(4)=640
PTRIG(5)=641
PTRIG(6)=642
PTRIG(7)=643
```

JOYSTICK CONTROLLERS

Peeking the joystick locations (addresses) works in the same way as for the paddle controllers, but can be used with the joystick controller. The joystick controllers are numbered 0-3 from left to right.

Example of using joystick (0):

```
10 STICK(0)=632
20 PRINT PEEK(STICK(0))
30 GOTO 20
```

STICK (joystick) number and PEEK (decimal) locations:

```
STICK(0)=632
STICK(1)=633
STICK(2)=634
STICK(3)=635
```

Figure 15-2 shows the PEEK number that will be returned for the various joystick positions:

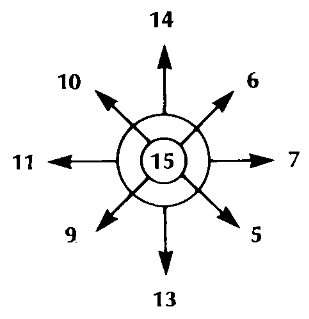


Figure 15-2 Joystick Triggers

Sensing the joystick triggers works the same way as for the paddle trigger buttons. It can be used with both the joystick and keyboard controllers.

Using joystick trigger (0):

```
10 STRIG(0)=644
20 PRINT PEEK(STRIG(0))
30 GOTO 20
```

STRIG (joystick) number and PEEK (decimal) locations:

```
STRIG(0)=644
STRIG(1)=645
STRIG(2)=646
STRIG(3)=647
```

```
5 REM THIS PROGRAM WILL SAY "BANG!" WHEN JOYSTICK RED BUTTON IS
6 REM PRESSED
10 IF PEEK(644)=0 THEN ? "Bang!"
20 IF PEEK(644)=1 THEN CLS
30 GOTO 10
```

CONSOLE KEYS

The following program reads the console keys on the right-hand side of the ATARI Computer:

```
10 POKE 53279,0
20 PRINT PEEK(53279)
30 GOTO 20
```

Peeking location 53279 (decimal) will return a number that indicates which key was pressed.

7= No key pressed
6= START key pressed
5= SELECT key pressed
3= OPTION key pressed

APPENDIX A

SAMPLE PROGRAMS

DISK DIRECTORY PROGRAM

Features used:

- User-callable CIO routines (CIOUSR) (See Appendix N.)
- Integers
- VARPTR function
- ON ERROR
- On-line comments

```
10 !                                    ROUTINE TO READ
20 !                                    DISK DIRECTORY
30 !
40 ON ERROR 350
50 OPTION RESERVE(200)                  !GET SPACE FOR CIOUSR ROUTINES
60 OPEN#1,"D:CIOUSR" INPUT              !OPEN FILE
80 ADDR=VARPTR(RESERVE)                 !GET STARTING ADDRESS OF RESERVED AREA
90 FOR I=0 TO 159                       !POKE IN CIOUSR ROUTINES
100 GET#1,D:POKE ADDR+I,D
110 NEXT I
120 CLOSE #1
130 PUTIOCB=ADDR                        !THESE ARE THE PROPER STARTING POINTS
140 CALLCIO=ADDR+61                     !FOR EACH OF THE
150 GETIOCB=ADDR+81                     !ROUTINES
160 DIM IOCB%(10)                       !DATA FOR ROUTINES TAKES 10 BYTES
170 IOCB%(0)=1                          !USE IOCB #1
180 IOCB%(1)=3                          !DO A CIO "OPEN" CALL
190 IOCB%(2)=6                          !FOR DIRECTORY INPUT
200 FSPEC$="D:*.*"                      !DIR FILE SPEC
210 !                                   !PUT ADDRESS OF FSPEC INTO BUFFER
220 Z=VARPTR(FSPEC$)                    !ADDRESS OF THE STRING FILESPEC
230 Y=VARPTR(IOCB%(3))                  !ADDRESS OF THE PROPER ARRAY POSITION
240 POKE Y,PEEK(Z+2)                    !HIGH ADDRESS BYTE
250 POKE Y+1,PEEK(Z+1)                  !LOW ADDRESS BYTE
260 !                                   PUTDATA INTO IOCB
270 Z=USR(PUTIOCB,VARPTR(IOCB%(0)))
280 !                                   THEN CALL CIO
290 Z=USR(CALLCIO,VARPTR(IOCB%(0)))
300 !                                   IOCB IS SETUP AND DISK
310 !                                   IS OPEN...READ DIRECTORY
320 INPUT #1,S$
330 PRINT S$
340 GOTO 320
350 CLOSE #1
360 END
```

EXPLOSION SUBROUTINE

Feature used: Sound

```
10 !TWO-LINE MAIN PROGRAM
20 !AND SUBROUTINE TO PRODUCE
30 !AN EXPLOSION
40 !
50 GOSUB 8000
60 STOP
8000 !
8010 ! EXPLOSION SUBROUTINE
8020 !
8030 SOUND 2,75,8,14
8040 ICR=0.79
8050 V1=15:V2=15:V3=15
8060 SOUND 0,NTE,8,V1
8070 SOUND 1,NTE+20,8,V2
8080 SOUND 2,NTE+50,8,V3
8090 V1 = V1 * ICR
8100 V2 = V2 * (ICR+.05)
8110 V3 = V3 * (ICR+.08)
8120 IF V3 > 1 THEN 8060
8130 SOUND 0,0,0,0,0
8140 SOUND 1,0,0,0,0
8150 SOUND 2,0,0,0,0
8160 RETURN
```

FANFARE MUSIC EXAMPLE

Feature used: Sound with duration

```
10 !ROUTINE TO GENERATE FANFARE MUSIC
20 !TWO-LINE MAIN PROGRAM
30 !
40 GOSUB 8000
50 STOP
8000 !
8010 !FANFARE MUSIC
8020 !
8030 DUR=20:V0=181:V1=144:V2=121:GOSUB 8200
8040 DUR=7:GOSUB 8200
8050 GOSUB 8200
8060 DUR=9:V0=162:V1=128:V2=108:GOSUB 8200
8070 DUR=15:V0=181:V1=144:V2=121:GOSUB 8200
8080 V0=162:V1=128:V2=108:GOSUB 8200
8090 V0=153:V1=128:V2=96:V3=193
8100 For I=2 TO 14
8110 SOUND 3,V0,10,I
```

```
8120 SOUND 1,V1,10,I
8130 SOUND 2,V2,10,I
8140 SOUND 0,V3,10,I
8150 FOR J=1 TO 100:NEXT J
8160 NEXT I
8170 FOR J=1 TO 200:NEXT J
8180 SOUND 0,0,0,0,0
8185 SOUND 1,0,0,0,0
8190 SOUND 2,0,0,0,0
8195 SOUND 3,0,0,0,0
8197 RETURN
8200 !SOUND GENERATOR
8210 SOUND 0,V0,10,8,DUR
8220 SOUND 1,V1,10,8,DUR
8230 SOUND 2,V2,10,8,DUR
8240 !
8250 !NOW STOP THE SOUND
8260 !
8270 SOUND 0,0,0,0,0
8280 SOUND 1,0,0,0,0
8290 SOUND 2,0,0,0,0
8295 FOR J=1 TO 250:NEXT J
8300 RETURN
```

EXAMPLE OF ATARI PIANO

Features used:
- OPEN statement
- String array
- INKEY$
- SOUND
- On-line comments

```
10 ! EXAMPLE PROGRAM TO
20 ! CONVERT YOUR ATARI
30 ! COMPUTER INTO A PIANO!
40 !
50 !
60 ! FIRST, SET UP A 2-OCTAVE
70 ! SCALE OF KEYS TO PRESS
80 ! AND NOTES TO PLAY
90 DIM SCALE$(15)
100 DIM PITCH(15)
110 ! NOW READ THESE INTO
120 ! THEIR RESPECTIVE TABLES
130 OPEN #1, "D:NOTES.DAT" INPUT
140 FOR I=1 TO 15
150 INPUT #1,S$,P
160 SCALE$(I)=S$:PITCH(I)=P
```

```
170 NEXT I
180 CLOSE #1
190 PRINT "PLAY, BURT, PLAY!"
200 !
210 ! BEGIN TESTING FOR KEYS
220 ! BEING PRESSED
230 !
240 N$=INKEY$
250 IF N$="" THEN GOTO 240 ELSE GOTO 320
260 !
270 ! WHEN A KEY IS PRESSED,
280 ! SEE IF ITS ONE ON OUR
290 ! PIANO KEYBOARD!
300 !
310 !
320 FOR I = 1 TO 15
330 IF N$ = SCALE$(I) GOTO 380
340 NEXT I
350 GOTO 240 !NOT A GOOD KEY, TRY AGAIN
360 ! FOUND A GOOD KEY, PROCESS IT
370 !
380 VOLUME = 8
390 SOUND 1,PITCH(I),10,VOLUME,15
400 GOTO 240
410 END
```

Sample NOTES.DAT FILE
First item is the key to be pressed.
Second item is the frequency to play.

NOTE.DAT CREATION PROGRAM

```
10 !PROGRAM TO CREATE NOTES.DAT FILE
20 !
30 DIM NOTES$(15),PITCH(15)
40 FOR I=1 TO 15
50 INPUT "ENTER KEY, FREQ. FOR KEY :";NOTES$(I),PITCH(I)
60 NEXT I
70 OPEN $1,"D:T" OUTPUT
80 FOR I=1, TO 15
90 PRINT $1,NOTES$(I);",";PITCH(I)
100 NEXT I
110 CLOSE $1
120 END
```

Enter the following values to get a 2-octave scale.

```
Z, 243
X, 217
C, 193
V, 182
B, 162
```

N,	144
M,	128
A,	121
S,	108
D,	96
F,	91
G,	81
H,	72
J,	64
K,	60

DECIMAL-TO-HEX CONVERSION ROUTINE

Features used:
- String array
- Integers
- On-line comments

```
20 !
30 ! D E C H E X
40 !
50 !
60 !
70 !PROGRAM TO CONVERT AN INPUT
80 !DECIMAL NUMBER TO ITS
90 !HEXADECIMAL EQUIVALENT
100 !
110 !
130 DIM HEX$(15):DIM HEXBASE(4)
140 FOR I=0 TO 15
150 READ HEX$(I)
160 NEXT I
170 FOR I=0 TO 4
180 READ HEXBASE(I)
190 NEXT I
200 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
210 DATA 0,4096,256,16,1
220 !
230 !GET THE DECIMAL NO.
240 !
250 INPUT "ENTER THE DECIMAL NO. :";DEC
260 IF DEC = 0 THEN 500 !STOP
270 !
280 !PROCESS EACH HEX DIGIT
290 !
300 FOR J = 1 TO 4
305 IF J=4 THEN ANS%=DEC:GOTO 350
310 ANS% = (DEC/HEXBASE(J)) - .5
320 IF ANS% < 1 THEN ANS% = 0
330 DEC = DEC - (ANS% * HEXBASE(J))
340 !
350 ! FIND THE HEX DIGIT FOR FIRST POSITION
```

```
360 FOR I% = 0 TO 15
370 IF ANS% = I% THEN GOTO 420
380 NEXT I%
390 !IF WE GOT HERE ITS AN ERROR!
400 PRINT " DECIMAL INPUT CAN'T BE COMPUTED"
410 PRINT "PLEASE TRY AGAIN": GOTO 250
420 HEXNO$ = HEXNO$ + HEX$(I%)
430 NEXT J
440 !
450 !PRINT THE HEX NO. AND GO FOR ANOTHER
460 !
470 PRINT "HEX NO. = ";HEXNO$
480 HEXNO$ = ""
490 GOTO 250
500 END
```

VERTICAL FINE SCROLLING

Features used:
- Fine scrolling
- VARPTR
- OPTION RESERVE and CHR
- User-defined display list

```
10 DEFINT A-Z
20 OPTION RESERVE(3000) !AREA FOR SCREEN RAM
30 OPTION CHR1 !AREA FOR DISPLAY LIST
40 ADDR = VARPTR(CHR1)
50 CADDR = VARPTR(RESERVE)
60 VSCROL = &D405 !VERTICAL SCROLL REGISTER
70 LCADDR = 0
80 HCADDR = ((CADDR AND &FF00)/256) AND &FF
90 FOR I=0 TO 99 !ZERO THE DISPLAY LIST AREA (1ST 100 BYTES)
100 POKE ADDR+I,0:NEXT I
110 LADDR = ADDR AND &FF
120 HADDR = ((ADDR AND &FF00)/256) AND &FF
130 LMSLO = ADDR+4 !ADDRESS OF LOAD
140 LMSHI = ADDR+5 !MEMORY SCAN BYTES (LMS)
150 FOR I=0 TO 18 !POKE IN NEW DISPLAY LIST
160 READ D !FROM DATA STMTS. 190-210
170 POKE ADDR+I,D
180 NEXT I
190 DATA &70,&70,&70,&67,&00,&00,&27,&27
200 DATA &27,&27,&27,&27,&27,&27,&27,&27
210 DATA &27,&07,&41
220 POKE ADDR+19,LADDR !LAST 2 BYTES POINT BACK
230 POKE ADDR+20,HADDR !TO TOP OF DISPLAY LIST
240 POKE LMSLO,LCADDR:POKE LMSHI,HCADDR !TELLS SCREEN RAM START
250 K=-1 !250 - 320 LOAD DATA INTO
260 FOR I=1 TO 300 !SCREEN RAM AREA, A PAGE FULL
270 K=K+1:POKE CADDR+K,33 !OF A's AND THEN THE ALPHABET
```

```
280 NEXT I
290 FOR I=34 TO 58
300 FOR J=1 TO 20
310 K=K+1:POKE CADDR+K,I
320 NEXT J,I
330 POKE &22F,0 !TURN OFF ANTIC
340 POKE &230,LADDR !TELL IT WHERE MY DISPLAY
350 POKE &231,HADDR !LIST IS, AND ...
360 POKE &22F,&22 !TURN ANTIC BACK ON
370 REM HERE IS THE REAL PROGRAM
380 FOR I=1 TO 15 !380 - 410 DO THE VERTICAL
390 POKE VSCROL,I !FINE SCROLL
400 FOR W=1 TO 30:NEXT W
410 NEXT I
420 CADDR=CADDR+20 !CALCULATE WHERE NEXT LINE OF
430 LCADDR=CADDR AND &FF !SCREEN RAM STARTS
440 HCADDR = ((CADDR AND &FF00)/256) AND &FF !FOR THE COARSE SCROLL
450 WAIT &D40B,&FF,96 !WAIT UNTIL TV VERTICAL LINE COUNTER HITS 96
460 POKE VSCROL,0 !THEN SET CHARACTERS BACK TO ORIGINAL POSITION
470 POKE LMSLO,LCADDR !AND COARSE
480 POKE LMSHI,HCADDR !SCROLL BY CHANGING LMS BYTE IN DISPLAY LIST
490 GOTO 380
```

APPENDIX B

GRAPHICS MODES PROGRAMS

MICROBE INVASION EXAMPLE

```
10 REM MICROBE INVASION
15 REM SPIRAL CREATURES TAKE OVER SCREEN
16 REM 10 PERCENT CHANCE SCREEN CHANGES MODE
17 REM WHEN CREATURE GOES OUT OF BOUNDS
30 RANDOMIZE
40 MODE=RND(8)
50 GRAPHICS MODE+16
60 PIX=RND(15)
70 SETCOLOR 0,PIX,6
80 COLOR 1
90 BAK=RND(255)
100 POKE 712,BAK
110 X=RND(150):Y=RND(100)
120 IF X>140 THEN 40
130 Z=2
140 NUM=NUM+1
150 FOR DOTS=1 TO Z
160 IF NUM=5 THEN NUM=1
170 ON ERROR GOTO 230
180 PLOT X,Y
190 ON NUM GOSUB 250,270,290,310
200 NEXT
210 Z=Z+1
220 GOTO 140
230 GRAPHICS MODE+32+16!NO TEXT WINDOW, NO SCREEN CLEAR
240 RESUME 60
250 X=X+1:Y=Y+1
260 RETURN
270 X=X+1:Y=Y-1
280 RETURN
290 X=X-1:Y=Y-1
300 RETURN
310 X=X-1:Y=Y+1
320 RETURN
```

The following short program makes use of RANDOMIZE and RND to print three-letter words and three-letter abbreviations of government agencies.

```
10 RANDOMIZE !Seeds the RND function
20 GRAPHICS 2+16
30 X=RND(26)+96 !Make first letter
40 Y=RND(5) !Make a vowel for middle letter
50 IF Y=1 THEN Y=97 !Make an A
60 IF Y=2 THEN Y=101 !Make an E
70 IF Y=3 THEN Y=105 !Make an I
80 IF Y=4 THEN Y=111 !Make an O
90 IF Y=5 THEN Y=117 !Make a U
100 Z=RND(26)+96 !Make last letter
110 PRINT#6, AT(9,3);CHR$(X);CHR$(Y);CHR$(Z)
120 FOR DELAY=1 TO 2000:NEXT
180 GOTO 30
```

APPENDIX C

ALTERNATE CHARACTER SETS

ATARI Home Computers support several standard character sets that are stored as part of the Operating System (OS) ROM. These include all the upper- and lowercase alphabet, numbers, special characters, and a special graphics character set. At times, however, it is very useful to be able to define your own character set. Applications for this capability that immediately come to mind include character-driven animation, foreign language word processing, and background graphics for games (for instance, a map or special playfield).

ATARI Computers and ATARI Microsoft BASIC readily support this ability. This is easy for the ATARI Home Computer because the OS data base contains a pointer (CHBAS) at hex location 2F4 (decimal location 756) which points to the character set to be used. Normally this points at the standard character set in the OS ROM. But in BASIC, you can POKE your own character set into a free area of RAM (set aside with the OPTION CHR1 or OPTION CHR2 statement) and then reset the OS pointer, CHBAS, to point to your new character set. The computer will instantly begin using the new characters.

There are several important things to keep in mind when redefining the character set:

- Graphics mode 0 needs 128 characters defined (OPTION CHR1). Graphics modes 1 and 2 allow only 64 characters (OPTION CHR2).
- All 64 or 128 characters need to be defined even though you may only wish to change and use one character; this is easily accomplished by transferring the ROM characters into your RAM area and then changing the desired character to its new configuration.
- The 64-character set requires 512 bytes of memory (8 bytes per character) and must start on a ½ K boundary. The 128-character set requires 1024 bytes of memory and must start on a 1K boundary. The programmer need not worry about these restrictions when using the CHR1 and CHR2 options; the area is allocated to begin on the proper boundary.
- The value that is poked into CHBAS after the character set is defined is the page number in memory where the character set begins. This value can be computed with the following statement:

  ```
  CHBAS% = (VARPTR(CHRn)/256) AND &FF
  ```

  Where "n" is either 1 or 2. This value is then poked into location &2F4 (decimal 756).

The most time-consuming process in using an alternate character set is creating the characters. Each character consists of 8 bytes of memory, stacked one on top of the other (see Figure C-1). Visualize each character as an 8x8 square of graph paper. Darken the necessary square on the graph paper to create a character (see Figure C-2). Then, each row of the 8x8 square is converted from this binary representation (where each darkened square is a 1 and each blank square is a zero) to a hex or decimal number (see Figure C-2). These numbers are then poked into the appropriate bytes of the RAM area, from top to bottom in these figures, to define the character in RAM. The first 8 bytes of the reserved (OPTION CHR1 or CHR2) area define the zeroth character; the next 8 bytes define the first character, and so on. After transferring the standard character set from its ROM location to the reserved CHR1 or CHR2 area, any character can be redefined by finding its starting position in the area, then poking the new bytes into the starting byte and the next 7 bytes. After all necessary characters are redefined, poke the new page number into CHBAS and the new character will immediately be active. Use BASIC PRINT statements to display the new characters; for instance, if you have redefined the "A" to be a solid block and use the statement,

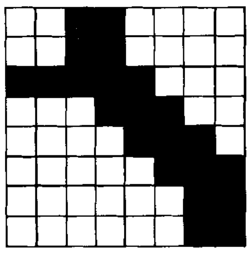
```
PRINT "A",
```

the new character will be printed.

A little experimentation with this process will quickly show you how powerful this capability can be. The program on the following page is an example of character set redefinition.

Byte 1
Byte 2
Byte 3
Byte 4
Byte 5
Byte 6
Byte 7
Byte 8

Figure C-1 Amount of Memory per Character



Byte No.	**Binary**	**Hex**	**Decimal**
1	00110000 =	30 =	48
2	00110000 =	30 =	48
3	11111000 =	F8 =	248
4	00011100 =	1C =	28
5	00001110 =	OE =	14
6	00000111 =	07 =	07
7	00000011 =	03 =	03
8	00000011 =	03 =	03

Figure C-2 Redefining a Character

SAMPLE PROGRAM

```
10 !
20 !PROGRAM TO DEMONSTRATE
30 !ALTERNATE CHARACTER SET
40 !DEFINITION
50 !
60 !THE PROGRAM REDEFINES THE
70 !CHARACTERS A,B,C,D,E,F,G,H
80 !
90 CHBAS = &2F4 !CHR. SET POINTER
100 OPTION CHR1 !ALLOCATE CHARACTER SET AREA
110 ADDR%=VARPTR(CHR1) !FIND STARTING ADDRESS
120 PAGENO% = (ADDR%/256) AND &FF !CALCULATE PAGE
130 !
140 MOVE 57344,ADDR%,1024 !MOVE CHR. SET DOWN INTO RAM
150 !
160 OFFSET=33*8 !OFFSET TO "A"
170 FOR I=0 TO 63 !GET NEW CHARACTERS
180 READ C
190 POKE ADDR%+OFFSET+I,C !AND INSERT
200 NEXT I
210 !
220 !DATA STATEMENTS ARE BY CHARACTER
230 !
240 DATA &07,&0F,&1F,&3F,&7F,&FF,&FF,&FF
250 DATA &E0,&F0,&F8,&FC,&FE,&FF,&FF,&FF
260 DATA &FF,&FF,&FF,&7F,&3F,&1F,&0F,&07
270 DATA &FF,&FF,&FF,&FE,&FC,&F8,&F0,&E0
280 DATA &00,&00,&00,&3F,&7F,&FF,&FF,&FF
290 DATA &00,&00,&00,&FC,&FE,&FF,&FF,&FF
300 DATA &FF,&FF,&FF,&7F,&3F,&00,&00,&00
310 DATA &FF,&FF,&FF,&FE,&FC,&00,&00,&00
320 !
330 POKE CHBAS.PAGENO% !SWITCH TO NEW CHARACTER SET!
340 !
350 POKE &2F0,1 !TURN OFF CURSOR
360 SETCOLOR 6,2,6 370 X=20
380 FOR Y=10 TO 20
390 WAIT &D40B,&FF,110
400 CLS: PRINT AT...
410 PRINT AT(X,Y+1):"CD"
420 FOR W=1 TO 30:NEXT W
430 NEXT Y
440 CLS: PRINT AT...
450 PRINT AT(X,22);"GH"
460 SOUND 0,79,10,8,4
470 FOR W=1 TO 80:NEXT W
480 FOR Y=20 TO 10 STEP -1
490 WAIT &D40B,&FF,110
500 CLS: PRINT AT...
510 PRINT AT(X,Y+1);"CD"
520 FOR W=1 TO 30:NEXT W
530 NEXT Y
540 GOTO 380
```

APPENDIX D

DERIVED FUNCTIONS

The following trigonometric functions can be derived by the calculations shown.

Derived Functions	**Derived Functions in Terms of Microsoft**
Secant	SEC(X)=1/COS(X)
Cosecant	CSC(X)=1/SIN(X)
Inverse sine	ARCSIN(X)=ATN(X/SQR(-X*X+1))
Inverse cosine	ARCCOS(X)=-ATN(X/SQR(-X*X+1) +CONSTANT))
Inverse secant	ARSEC(X)=ATN(SQR(X*X-1)) +(SGN(X-1)*CONSTANT)
Inverse cosecant	ARCCSC(X)=ATN(1/SQR(X*X-1)) +(SGN(X-1)*CONSTANT)
Inverse contangent	ARCCOT(X)=ATN(X)+CONSTANT
Hyperbolic sine	SINH(X)=(EXP(X)-EXP(-X))/2
Hyperbolic cosine	COSH(X)=(EXP(X)+EXP(-X))/2
Hyperbolic tangent	TANH(X)=-EXP(-X)/(EXP(X)+EXP(-X))*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	ARCSINH(X)=LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine	ARCCOSH(X)+LOG(X+SQR(X*X-1))
Inverse hyperbolic tangent	ARCTANH(X)=LOG((1+X)/(1-X))/2
Inverse hyperbolic secant	ARCSECH(X)=LOG((SQR(-X*X+1)+1)/X)
Inverse hyperbolic cosecant	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
Inverse hyperbolic cotangent	ARCCOTH(X)=LOG((X+1)/(X-1))/2

APPENDIX E

MEMORY LOCATIONS

Memory locations are expressed in hexadecimal, with decimal equivalents in parentheses. For additional information, see the *ATARI Personal Computer System Technical Users Notes* (part number C016555).

MEMORY MAP

The 6502 Microprocessor is divided into four basic memory regions: RAM, cartridge area, I/O chip region, and resident OS ROM. Memory regions and their address boundaries are listed below:

RAM (minimum required for operation):	0000-1FFF (0-8191)
RAM expansion area:	2000-7FFF (8192-32767)
Cartridge B (left cartridge) or 8K RAM:	8000-9FFF (32768-40959)
Cartridge A (right cartridge) or 8K RAM:	A000-BFFF (40960-49151)
Unused:	C000-CFFF (49152-53247)
I/O chips:	D000-D7FF (53248-55295)
OS floating point package:	D800-DFFF (55296-57343)
Resident Operating System ROM:	E000-FFFF (57344-65535)

RAM REGION

The RAM region, shared by the OS and the program in control, is divided into the following subregions:

- 6502 Microprocessor Page 0 Address Mode Region: 0000 through 00FF (0-255) allocated as follows:

  0000 through 007F (0-127): OS
  0080 through 00FF (128-255): User applications
  00D4 through 00FF (212-255): Floating point package, if used.

- Page 1, 6502 Hardware Stack Region: 0100 through 01FF (256-511).

Note: At power up or SYSTEM RESET, the stack location points to address 01FF (511) and the stack then pushes downward toward 0100 (256). The stack wraps around from 0100 to 01FF if a stack overflow occurs.

- Pages 2-4, OS Data Base (working variables, tables, data buffers): 0200 through 047F (512-1151).

- Pages 7-XX, User Boot Area: 0700 (1792) to start of free RAM area, where XX is a function of the screen graphics mode and the amount of RAM installed.

Note: When initial diskette startup is completed, the data base variable points to the next available location above software loaded. When no software is entered by the the initial diskette startup, the data base variable points to location 0700.

- Screen Display List and Data: Page XX to top of RAM. Data base pointer contains address of last available location below the screen area.

CARTRIDGE AREA

Cartridge B is the RIGHT CARTRIDGE on the ATARI 800 Home Computer. Cartridge A is the LEFT CARTRIDGE on the ATARI 800 Home Computer and the only cartridge on the ATARI 400 Home Computer.

- Cartridge B: 8000 through 9FFF (32768-40959)
- Cartridge A: A000 through BFFF (40960-49151) for 8K cartridges; 8000 through BFFF (32768-49151) for 16K cartridges (optional)

Note: On the ATARI 800 Home Computer, if a RAM module plugged into the last slot overlaps any of these cartridge addresses, the installed cartridge will disable the conflicting RAM module in 8K increments.

I/O CHIPS

The 6502 Microprocessor performs input/output operations by addressing the following external support chips as memory:

- CTIA D000 through D01F (53248-53279)
- POKEY D200 through D21F (53760-53791)
- PIA D300 through D31F (54016-54047)
- ANTIC D400 through D41F (54272-543030)

Some of the chip registers are read/write; others are read only or write only. Table E-2 lists the registers and their addresses by chip. For additional information, see the *ATARI Personal Computer System Technical Users Notes.*

RESIDENT OS ROM

The region from D800 through FFFF (55296-65535) permanently contains the OS and the floating point package:

- Floating point package: D800 through DFFF (55296-57343)
- Operating System ROM: E000 through FFFF (57344-65535)

The OS contains many vectored entry points, all fixed, at the end of the ROM and in RAM. The floating point package is not vectored, but all documented entry points will be fixed. See the Appendix of the *ATARI Personal Computer System OS Users Manual* (part of the *ATARI Personal Computer System Technical Users Notes*) for listings of the fixed ROM vectors and entry points.

TABLE E-1
USEFUL OS DATA BASE ADDRESSES

Address Hex	Dec	Name	Byte Size	Function
MEMORY CONFIGURATION (See Sections 4 and 7, *ATARI Personal Computer System OS Users Manual,* part of *ATARI Personal Computer System Technical Users Notes.*)				
000E	14	APPMHI	2	User-free memory screen lower limit
006A	106	RAMTOP	1	Display handler top of RAM address (MSB)
02E4	740	RAMSIZ	1	Top of RAM address (MSB)
O2E5	741	MEMTOP	2	User-free memory high address
02E7	743	MEMLO	2	User-free memory low address

TEXT/GRAPHICS SCREEN (See Section 5, *OS Users Manual*.)

Screen Margins (text modes; text window)

0052	82	LMARGN	1	Left screen margin (0-39; default 2)
0053	83	RMARGN	1	Right screen margin (0-39; default 39)

Cursor Control

0054	84	ROWSCRS	1	Current cursor row
0055	85	COLCRS	2	Current cursor column
005A	90	OLDROW	1	Prior cursor row
005B	91	OLDCOL	2	Prior cursor column
0290	656	TXTROW	1	Current cursor row in text window
0291	657	TXTCOL	2	Current cursor column in text window
02F0	752	CRSINH	1	Cursor display inhibit flag (0 = cursor on, 1 = cursor off)

Color Control

02C0	704	PCOLR0	4	Color-luminance Player-Missile 0
02C1	705	PCOLR1	4	Color-luminance of Player-Missile 1
02C2	706	PCOLR2	4	Color-luminance of Player-Missile 2
02C3	707	PCOLR3	4	Color-luminance of Player-Missile 3
02C4	708	COLOR0	5	Color-luminance of Playfield 0
02C5	709	COLOR1	5	Color-luminance of Playfield 1
02C6	710	COLOR2	5	Color-luminance of Playfield 2
02C7	711	COLOR3	5	Color-luminance of Playfield 3
02C8	712	COLOR4	5	Color-luminance of background

Attract Mode

004D	77	ATRACT	1	Attract mode timer and flag (Value 128 = on; turns on every 9 minutes)

Tabbing

02A3	675	TABMAP	15	Tab stop bit map (default: 7, 15, 23, etc. to 119)

Screen Memory

0058	88	SAVMSC	2	Upper left corner of screen

Split-Screen Memory

0294	660	TXTMSC	2	Upper left corner of text window

DRAW/FILL Function

0060	96	NEWROW	1	Destination point; initialized to value in ROWCRS.
0061	97	NEWCOL	2	Destination point; initialized to value in COLCRS.
02FD	765	FILDAT	1	Fill data for graphics FILL command.

Internal Character Code Conversion

02FA	762	ATACHR	1	Contains last ATASCII character or plot point.

Display Control Characters

02FE	766	DSPFLG	1	Display control character flag. (1 = display control characters)

KEYBOARD (See Section 5, *OS Users Manual.*)

Key Reading

02FC	764	CH	1	Contains value of last keyboard character in FIFO or $FF if FIFO is empty.

Special Functions

0011	17	BRKKEY	1	BREAK key flag (normally nonzero; set to 0 by BREAK)
02B6	694	INVFLG	1	Inverse video flag (norm = 0; set by key)
02BE	702	SHFLOK	1	Shift/control lock control flag ($00 = no lock (norm); $40 = caps lock; $80 = control lock)
02FF	767	SSFLAG	1	Start/stop flag (norm = 0; set by CTRL 1). Set to $40 on power up and SYSTEM RESET; reset by CAPS LOWR, CAPS LOWR SHIFT, or CAPS LOWR CTRL.

CENTRAL I/O (CIO) ROUTINE (See Section 5, *OS Users Manual.*)

I/O Control Block

0340-034F (832-847)	IOCB	16	I/O Control Block 0
0350-035F (848-863)	IOCB	16	I/O Control Block 1
0360-036F (864-879)	IOCB	16	I/O Control Block 2
0370-037F (880-895)	IOCB	16	I/O Control Block 3
0380-038F (896-911)	IOCB	16	I/O Control Block 4
0390-039F (912-927)	IOCB	16	I/O Control Block 5
03A0-03AF (928-943)	IOCB	16	I/O Control Block 6
03B0-03BF (944-959)	IOCB	16	I/O Control Block 7

0340	832	ICHID	1	Handler I.D. (See Section 5; Initialized to $FF at power up and SYSTEM RESET.)
0341	833	ICDNO	1	Device number
0342	834	ICCMD	1	Command byte
0343	835	ICSTA	1	Status
0344	836	ICBAL/ICBAH	2	Buffer address
0346	838	ICPTL/ICPTH	2	PUT BYTE vector (Points to CIO's "IOCB not OPEN" at power up and SYSTEM RESET.)
0348	840	ICBLL/ICBLH	2	Buffer length/byte count
034A	842	ICAX1/ICAX2	2	Auxiliary information
034C	844	ICAX3/ICAX6	4	Spare bytes for handler use

Zero Page IOCB

0020	32	ZIOCB	16	Zero page IOCB (Only the first 12 bytes (IOCBs) are moved by the CIO utility.)
0020	32	ICHIDZ	1	Handler index number (set to $FF on CLOSE)
0021	33	ICDNOZ	1	Device drive number
0022	34	ICCOMZ	1	Command byte
0023	35	ICSTAZ	1	Status byte
0024	36	ICBALZ,ICBALH	2	Buffer address
0026	38	ICPTLZ,ICPTHZ	2	PUT BYTE vector (Points to CIO's "IOCB not OPEN" on CLOSE.)
0028	40	ICBLLZ,ICBLHZ	2	Buffer length/byte count
002A	42	ICAX1Z,ICAX2Z	2	Auxiliary information
0002C	44	ICSPRZ	4	CIO working variables
		(ICIDNO,ICOCHR)		CIDNO = ICSPRZ+2; ICOCHR = ICSPRZ+3 (See Sections 5 and 9 of the *OS Users Manual.*)

DEVICE STATUS

02EA	746	DVSTAT	4	Device status

DEVICE TABLE (See Section 9, OS Users Manual.)

O31A	749	HATABS	38	Device handler table

SERIAL I/O (SIO) ROUTINE (See Section 9, OS Manual.)

Device Control Block

0300-030B	(768-779)	D C B	12	Device control block
0300	768	DDEVIC	1	Device bus I.D.
0301	769	DUNIT	1	Device unit number

0302	770	DCOMND	1	Device command
0303	771	DSTATS	1	Device status
0304	772	DBUFLO,DBUFHI	2	Handler buffer address
0306	774	DTIMLO	1	Device timeout (See Section 9, *OS Users Manual.*)
0308	776	DBYTLO,DBYTHI	2	Buffer length/byte count (See Section 9, *OS Users Manual.*)
030A	778	DAUX1,DAUX2	2	Auxiliary information

BUS SOUND CONTROL

0041	65	SOUNDR	1	Quiet/noisy I/O flag (0 = quiet)

ATARI CONTROLLERS (See Appendix L, *OS Users Manual.*)

Joysticks

0278	632	STICK0-STICK3	4	Joystick position port
0284	644	STRIG0-STRIG3	4	Joystick trigger port

Paddles

0270	624	PADDL0-PADDL7	8	Paddle position port
027C	636	PTRIG0-PTRIG7	8	Paddle trigger port

Light Pen

0234	564	LPENH	1	Light pen horizontal position code
0235	565	LPENV	1	Light pen vertical position code
0278	632	STICK0-STICK3	4	Light pen button port

FLOATING POINT PACKAGE (See Section 8, *OS Users Manual.*)

00D4	212	FR0	6	Floating point register 0
00E0	224	FR1	6	Floating point register 1
00F2	242	CIX	1	Character index
00F3	243	INBUFF	1	Input text buffer pointer
00FB	251	DEGFLG/RADFLG	1	Degrees/radians flag (0 = DEGFLG; 6 = degrees; DEGFLG = 0)
00FC	252	FLPTR	2	Pointer to floating point number
0580	1408	LBUFF	96	Text buffer

POWER UP AND SYSTEM RESET (See Section 7, *OS Users Manual.*)

Diskette/Cassette Boot

0002	2	CASINI	2	Cassette boot initialization vector
000C	12	DOSINI	2	Diskette boot initialization vector

Environment Control

0008	8	WARMST	1	Warmstart flag (= 0 on power up; $FF on SYSTEM RESET)
000A	10	DOSVEC	2	Noncartridge control vector (See Section 10, *OS Users Manual.*)

INTERRUPTS (See Secton 6, OS Users Manual.)

0010	16	POKMSK	1	POKEY interrupt mask
0042	66	CRITIC	1	Critical code section flag (nonzero = executing code is critical)

Real Time Clock

0012	18	RTCLOK	3	Real time frame counter (1/60 sec) (+0 = MSB; +1 = NSB; +2 = LSB)

System VBLANK Timers

0218	536	CDTMV1	2	System timer 1 value
021A	538	CDTMV2	2	System timer 2 value
021C	540	CDTMV3	2	System timer 3 value
021E	542	CDTMV4	2	System timer 4 value
0020	544	CDTMV5	2	System timer 5 value
0226	550	CDTMA1	2	System timer 1 jump address
0228	552	CDTMA2	2	System timer 2 jump address
022A	554	CDTMF3	2	System timer 3 flag
022C	556	CDTMF4	1	System timer 4 flag
022E	558	CDTMF5	2	System timer 5 flag

NMI Interrupt Vectors

0200	512	VDSLST	2	Display list interrupt vector (not used by the OS)
0222	546	VVBLKI	2	Immediate VBLANK vector
0224	548	VVBLKD	2	Deferred VBLANK vector

IRQ Interrupt Vectors

0202	514	VPRCED	2	Serial I/O bus proceed signal
0204	516	VINTER	2	Serial I/O bus interrupt signal
0206	518	VBREAK	2	BREAK instruction vector
0208	520	VKEYBD	2	Keyboard interrupt vector
020A	522	VUSERIN	2	Serial I/O bus receive data ready
020C	524	VSEROR	2	Serial I/O bus transmit ready
020E	526	VSEROC	2	Serial I/O bus transmit complete
0210	528	VTIMR1	2	POKEY timer vector (not used by OS)
0212	530	VTIMR2	2	POKEY timer vector (not used by OS)
0214	532	VTIMR4	2	POKEY timer vector (not used by OS)
0216	534	VIMIRQ	2	General IRQ vector

Hardware Register Updates

0230	560	SDLSTL	1	Screen display list address
0231	561	SDLSTH	1	Screen display list address
02C0	704	PCOLRx	4	Color register
02C4	708	PCOLORx	5	Color register
02F3	755	CHACT	1	Character control
02F4	756	CHBAS	1	Character address base register ($E0=uppercase, number set; $E2=lowercase, special graphics set; default=$E0)

USER AREAS (See Section 4, *OS Users Manual.*)

Note: The following areas are available to the user in a nonnested environment.

0080	128	128
0480	1152	640

Note: For additional information refer to the *ATARI Personal Computer System Hardware Manual* (part of the ATARI Personal Computer System Technical Notes).

TABLE E-2
HARDWARE ADDRESSES

Address Hex	Dec	Register Name	Function	OS Hex	Shadow Dec	Name
ANTIC CHIP						
D400	54272	DMACTL	Direct memory access (DMA) control (WRITE)	22F	559	SDMCTL
D401	54273	CHACTL	Character control (WRITE)	2F3	755	CHART
D402	54274	DLISTL	Display list pointer low byte (WRITE)	230	560	SDLSTL
D403	54275	DLISTH	Display list pointer high byte (WRITE)	231	561	SDLSTH
D404	54276	HSCROL	Horizontal scroll (WRITE)			
D405	54277	VSCROL	Vertical scroll (WRITE)			
D407	54279	PMBASE	Player-missile base address (WRITE)			
D409	54281	CHBASE	Character base address (WRITE)	2F4	756	CHBAS
D40A	54282	WSYNC	Wait for horizontal sync (WRITE)			
D40B	54283	VCOUNT	Vertical line counter (READ)			
D40E	54286	NMIEN	Nonmaskable interrupt (NMI) enable (WRITE)			
D40F	54287	NMIRES	Reset NMIST (WRITE)			
D40F	54287	NMIST	NMI status (READ)			
D410-D4FF (54288-54527) Repeat ANTIC addresses D400 through D40F.						
CTIA CHIP						
PLAYER-MISSILE GRAPHICS CONTROL						
Horizontal Position Control (WRITE)						
D000	53248	HPOSP0	Horizontal position Player 0			
D001	53249	HPOSPI	Horizontal position Player 1			
D002	53250	HPOSP2	Horizontal position Player 2			
D003	53251	HPOSP3	Horizontal position Player 3			
D004	53252	HPOSMO	Horizontal position Missile 0			
D005	53253	HPOSM1	Horizontal position Missile 1			
D006	53254	HPOSM2	Horizontal position Missile 2			
D007	53255	HPOSM3	Horizontal position Missile 3			

Collision Control (READ)

D000	53248	M0PF	Missile 0 to playfield
D001	53249	M1PF	Missile 1 to playfield
D002	53250	M2PF	Missile 2 to playfield
D003	53251	M3PF	Missile 3 to playfield
D004	53252	P0PF	Player 0 to playfield
D005	53253	P1PF	Player 1 to playfield
D006	53254	P2PF	Player 2 to playfield
D007	53255	P3PF	Player 3 to playfield
D008	53256	M0PL	Missile 0 to player
D009	53257	M1PL	Missile 1 to player
D00A	53258	M2PL	Missile 2 to player
D00B	53259	M3PL	Missile 3 to player
D00C	53260	P0PL	Player 0 to player
D00D	53261	P1PL	Player 1 to player
D00E	53262	P2PL	Player 2 to player
D00F	53263	P3PL	Player 3 to player

Collision Clear (WRITE)

D01E	53278	HITCLR	Collision clear

Size Control (WRITE)

Note: 0 = normal, 1 = double, 3 = quadruple size.

D008	53256	SIZEP0	Size of Player 0
D009	53257	SIZEP1	Size of Player 1
D00A	53258	SIZEP2	Size of Player 2
D00B	53259	SIZEP3	Size of Player 3
D00C	53260	SIZEM	Sizes of all missiles

Graphics Registers (WRITE)

D00D	53261	GRAFP0	Graphics for Player 0
D00E	53262	GRAFP1	Graphics for Player 1
D00F	53263	GRAFP2	Graphics for Player 2
D010	53264	GRAFP3	Graphics for Player 3
D011	53265	GRAFM	Graphics for all missiles

Joystick Controller Triggers (READ)

D010	53264	TRIG0	Read Joystick 0 trigger	284	644	STRIG0
D011	53265	TRIG1	Read Joystick 1 trigger	285	645	STRIG1
D012	53266	TRIG2	Read Joystick 2 trigger	286	646	STRIG2
D013	53267	TRIG3	Read Joystick 3 trigger	287	647	STRIG3

Color-Luminance Control (WRITE)

D012	53266	COLPM0	Color-lum. Player-Missile 0	2C0	704	COLR0
D013	53267	COLPM1	Color-lum. Player-Missile 1	2C1	705	PCOLR1
D014	53268	COLPM2	Color-lum. Player-Missile 2	2C2	706	PCOLR2
D015	53269	COLPM3	Color-lum. Player-Missile 3	2C3	707	PCOLR3
D016	53270	COLPF0	Color-lum. Playfield 0	2C4	708	COLOR0
D017	53271	COLPF1	Color-lum. Playfield 1	2C5	709	COLOR1
D018	53272	COLPF2	Color-lum. Playfield 2	2C6	710	COLOR2
D019	53273	COLPF3	Color-lum. Playfield 3	2C7	711	COLOR3
D01A	53274	COLBK	Color-lum. background	2C8	712	COLOR4

Priority Control (WRITE)

D01B	53275	PRIOR	Priority selection	26F	623	GPRIOR

Graphics Control (WRITE)

D01D	53277	GRACTL	Graphics control

MISCELLANEOUS I/O FUNCTIONS

PAL/NTSC Systems

D014	53268	PAL	Read PAL/NTSC bits

Console Switches (set to 8 during VBLANK)

D01F	53279	CONSOL	Write console switch port
D01F	53279	CONSOL	Read console switch port

POKEY CHIP

Audio (WRITE)

D200	53760	AUDF1	Audio Channel 1 frequency			
D201	53761	AUDC1	Audio Channel 1 control			
D202	53762	AUDF2	Audio Channel 2 frequency			
D203	53763	AUDC2	Audio Channel 2 control			
D204	53764	AUDF3	Audio Channel 3 frequency			
D205	53765	AUDC3	Audio Channel 3 control			
D206	53765	AUDF4	Audio Channel 4 frequency			
D207	53767	AUDC4	Audio Channel 4 control			
D208	53768	AUDCTL	Audio control			

Start Timer (WRITE)

D209	53769	STIMER	Resets audio-frequency dividers to AUDF values			

Pot Scan (Paddle Controllers)

D200	53760	POT 0	Read Pot 0	270	624	PADDL0
D201	53761	POT 1	Read Pot 1	271	625	PADDL1
D202	53762	POT 2	Read Pot 2	272	626	PADDL2
D203	53763	POT 3	Read Pot 3	273	627	PADDL3
D204	53764	POT 4	Read Pot 4	274	628	PADDL4
D205	53765	POT 5	Read Pot 5	275	629	PADDL5
D206	53766	POT 6	Read Pot 6	276	630	PADDL6
D207	53767	POT 7	Read Pot 7	277	631	PADDL7
D208	53768	ALLPOT	Read 8-line pot-port state			
D20B	53771	POTGO	Start pot scan sequence (written during VBLANK)			

Keyboard Scan and Control (READ)

D209	53769	KBCODE	Keyboard code	2FC	764	CH

Random Number Generator (READ)

D20A	53770	RANDOM	Random number generator			

Serial Port

D20A	53770	SKRES	SKSTAT reset (WRITE)			
D20D	53773	SERIN	Serial port input (READ)			
D20D	53773	SEROUT	Serial port output (WRITE)			
D20F	53775	SKCTLS	Serial Port 4-keyboard control (WRITE)	232	562	SSKCTL
D20F	53775	SKSTAT	Serial Port 4-keyboard status register (READ)			

IRQ Interrupt

D20E	532774	IRQEN	IRQ interrupt enable (WRITE)	10	16	POKMSK
D20E	532775	IRQST	IRQ interrupt status (READ)			

D210-D2FF (53776-54015) Repeat D200-D20F (53760-53775)

PIA CHIP

Joystick Read/Write Registers

D300	54016	PORTA	Writes or reads data from Controller Jacks 1 and 2 if bit 2 of PACTL = 1. Writes to direction control register if bit 2 of PACTL = 0.	278 279	632 633	STICK0 STICK1
D301	54017	PORTB	Writes or reads data from Controller Jacks 3 and 4 if bit 2 of PBCTL = 1. Writes to direction control register if bit 2 of PBCTL = 0.	27A 27B	634 635	STICK2 STICK3
D302	54018	PACTL	Port A control (set to $3C by IRQ code).			
D303	54019	PBCTL	Port B control (set to $3C by IRQ code).			

D304-D3FF (54020-54271) Repeat D300-D303 (54016-54019)

APPENDIX F

PROGRAM CONVERSIONS

COPYRIGHT NOTICE

Computer programs are protected in general by the Copyright Law. While the Copyright Law expressly permits the owner of the copyright for a computer program to adapt the program as necessary for utilization on a machine, such adaptation or translation is otherwise generally prohibited. ATARI recommends that you only convert programs purchased from the copyright owner or in accordance with a software license.

CONVERTING PROGRAMS TO ATARI MICROSOFT BASIC

The COMMODORE PET*® BASIC, APPLE**® APPLESOFT**® BASIC, and RADIO SHACK***® LEVEL II BASIC were all written by Microsoft. The overall approach and syntax of these BASIC languages has been kept compatible whenever possible to allow both programs and programmers to easily move from machine to machine. This appendix reviews the differences and indicates how to work around them when converting to ATARI Microsoft BASIC.

Microsoft divided its original BASIC into several different levels: 4K, 8K, Extended, and Full. Each successive level was a superset of the previous level and required more memory. When a manufacturer requested BASIC, the specific level to start from was determined by the memory constraints of the target machine. One source of incompatibility is due to starting at different levels. PET BASIC and APPLE APPLESOFT BASIC are based on the 8K level. RADIO SHACK LEVEL II and ATARI Microsoft BASIC are based on the full language level. Fortunately, this makes conversion into ATARI Microsoft BASIC easy. The key language differences between 8K and Full BASIC are the following:

- DATA TYPES: In 8K BASIC, double precision is not supported. Only 9 digits of accuracy are available. Integers can be used but they are converted to single precision before any arithmetic is done, so their only advantage is small storage requirements — not speed.
- PRINT USING is not available, so the user has to format his own numbers.
- The advanced statements: IF...THEN...ELSE, DEFINT, DEFSNG, DEFDBL, DEFSTR, TRON, TROFF, RESUME, and LINE INPUT are not supported.
- The functions, INSTR and STRING$, are not supported.
- Arrays can only be single dimensioned.
- User-defined functions can only have one argument.

By far the most difficult areas for conversion are machine-dependent features such as graphics and machine language use. In all programming it is important to isolate the uses of the features and document the assumption made about the machine.

*PET is a registered trademark of Commodore Business Machines, Inc.
**APPLE and APPLESOFT are registered trademarks of APPLE COMPUTER.
***RADIO SHACK is a registered trademark of TANDY CORPORATION.

APPENDIX G

CONVERSION FROM COMMODORE (PET) BASIC VERSION 4.0

Most of the difficulty in converting from Commodore (PET) BASIC (used on Commodore PET computers) comes from specific hardware features rather than the BASIC language since it is a strict implementation of the 8K level. Some of the conversion problems are:

- The Commodore PET character set has been extended to 256 characters. These characters are block graphics characters. In order to emulate this feature of the Commodore PET, an ATARI Computer user should set up a RAM-based character set.
- Commodore PET BASIC has built-in constants as follows: TI$ (TIME$ for ATARI Computers) and TI (TIME for ATARI Computers), ST for the STATUS of the last I/O operation and a pi symbol for the constant pi.
- Commodore PET I/O is done with special statements that control its IEEE bus. The arguments to OPEN are completely different from other machines and must be completely changed. The exact format of sending the characters is done by specifying a channel number with PRINT and INPUT statements, which is the same as ATARI Microsoft BASIC, so only the OPEN and control statements need to be reprogrammed.
- The display size of the Commodore PET is 40 by 25. If menus are designed for this layout, they will need to be reprogrammed.
- PEEKs and POKEs are always very machine dependent. Commodore PET programs often use PEEK and POKE to control cursor positioning because there is no direct way to change the cursor position. Each PEEK and POKE must be examined and reprogrammed.
- Commodore PET programs often embed cursor control characters in literal text strings. The ATARI Microsoft BASIC also supports this feature but the character codes are different and must be changed.
- The Commodore PET calls CLEAR, CLR.
- Any use of machine language through the Commodore PET EXEC statement will have to be carefully examined because although the microprocessor is the same, the layout of memory and the way of passing arguments to BASIC and receiving them from BASIC are quite different.
- Since the Commodore PET does not support sound or true graphics there is no conversion problem in these areas.
- RND is different. RND with a positive argument (generally 1) returns a number between 0 and 1.

Overall, if a special character set is set up identical to the Commodore PET's, it should be quite easy to convert programs that do not make heavy use of machine language or PEEK and POKE.

CONVERSION TO ATARI MICROSOFT BASIC

Use the following table to convert a software program developed under Commodore (PET) BASIC 4.0.

Note: For simplicity, those universal BASIC commands such as RUN, CONT, and POKE have been omitted. In those cases, no conversion is necessary.

The following table can also be used to perform diskette-based functions. Commodore (PET) BASIC 4.0 is a diskette-based language that must be supported by the ATARI ComputerDOS options.

(Also see Appendix A.)

COMMODORE (PET) COMMAND	**Equivalent ATARI Computer DOS OPTION**	**ATARI Microsoft BASIC**
DIRECTORY	**A** RETURN DIRECTORY—SEARCH SPEC, LIST FILE? RETURN	
COPY	**C** RETURN COPY—FROM,TO? **D1:***fn*,**D2:***fn* RETURN	
RENAME	**E** RETURN RENAME,GIVE OLD NAME,NEW **D2:***old fn, new fn* RETURN	NAME
SCRATCH	**D** RETURN DELETE FILESPEC **D2:***fn* RETURN TYPE "Y" TO DELETE *fn* **Y** RETURN	KILL
HEADER	**I** RETURN WHICH DRIVE TO FORMAT? **1** RETURN TYPE "Y" TO FORMAT DRIVE 1 **Y** RETURN	
BACKUP D0 TO D1	**J** RETURN DUP DISK—SOURCE,DEST DRIVES? **1,1** RETURN TYPE "Y" IF OK TO USE PROGRAM AREA? **Y** RETURN INSERT SOURCE DISK,TYPE RETURN RETURN INSERT DESTINATION DISK,TYPE RETURN RETURN	

Keep in mind that the Commodore (PET) BASIC 4.0 is a diskette-supported language. Therefore, when converting to run the Commodore (PET) program on your ATARI Computer, you must be aware of the peripherals involved.

DLOAD	LOAD "Dn:filename"
LOAD	CLOAD
DCLOSE	CLOSE *filenumber*
DOPEN	OPEN *filenumber*
DSAVE	SAVE *filename*
SAVE	CSAVE

Some of the Commodore (PET) BASIC 4.0 commands cannot be easily supported. As an example, use the following conversion:

APPEND#

```
OPEN #1, "filespec" INPUT
OPEN #2, "filespec" OUTPUT
LINE INPUT#1, A$
PRINT #2, A$
CLOSE #1
KILL "filename"
INPUT "filename";N$
LINE INPUT "      ";A$
LINE INPUT "      ";B$
PRINT#2, N$
PRINT#2, A$
PRINT#2, B$
CLOSE
NAME "filename2" AS "filename"
```

Check the logical flow of the software that you wish to convert to determine the direction of these commands. You will have to program around their use, depending upon the results you wish to accomplish with your software application.

APPENDIX H

CONVERTING RADIO SHACK TRS-80 PROGRAMS TO ATARI MICROSOFT BASIC

Radio Shack BASIC is based on Full Microsoft BASIC, so converted programs will make much better use of the features of ATARI Microsoft BASIC than APPLE or Commodore PET programs. ATARI Microsoft BASIC does have some additional features, such as COMMON, because it was written later and because the memory limitation for storing BASIC itself is not as restrictive on the ATARI Computer as it is on the Radio Shack Computer. The term Radio Shack BASIC refers to the BASIC built into the Model I and Model III computers, and called "Level II" BASIC. The BASIC on the Model II is very similar, but it is not specifically covered here.

- The Radio Shack display size poses the greatest problem in converting TRS-80 BASIC programs, because it is 16 by 64. Programs that use the full 64 characters for tables or menus will need to be changed.
- Radio Shack supports a form of graphics that allow black and white displays of 128 by 48 pixels intermixed with characters. The only statements for manipulation of the graphics are: CLS (clear screen), SET (turn a point on), RESET (turn a point off), and POINT (test the value of a point on the screen).
- Radio Shack does not store the up-arrow character in the standard ASCII position, so it has to be translated when moving programs onto the ATARI Computer.
- Radio Shack PRINTER I/O is done with LPRINT and LLIST without opening a device. Radio Shack cassette I/O is done with PRINT or INPUT to channels 1 and 2 (two drives can be supported). The format of files on cassette is completely different.
- Calls to machine language are done with USR. Because Radio Shack Computers use the Z-80 processor instead of the 6502, machine language routines will have to be completely rewritten.
- PEEKs and POKEs cannot be directly converted. PEEK and POKE are not heavily used on the Radio Shack Computers.
- The cursor positioning syntax is an @ after PRINT in Radio Shack BASIC and "AT" in ATARI Microsoft BASIC.
- The error codes returned by ERR are completely different.

TRS-80	ATARI	DEFINITION
AUTO mm-nn	AUTO mm,nn	Generates line numbers automatically.
CDBL(exp)	————	Returns double-precision representation of expression.
CINT(exp)	————	Returns largest integer not greater than the expression.
CLOAD	CLOAD LOAD"C:"	Loads a BASIC program from tape.
CLOAD?	VERIFY"C:filespec"	Verifies BASIC program on tape to one in memory.
CSNG(X)	Automatically truncates	Returns single-precision representation of the expression.
EDIT ln	AUTO line number	Lets you edit specified line number. Use cursor control keys.
FIX(x)	SGN(X)*INT(ABS(X))	Truncates all digits to the right of the decimal point.
INPUT#-1	OPEN#5, "C:"INPUT INPUT#5	INPUT reads data from cassette tape.
LIST mm-nn	LIST mm-nn	Lists the program in memory onto the printer.
LLIST	LIST "P:" mm-nn	Lists program to printer.
LPRINT	OPEN#4, "P:" OUTPUT PRINT#4, "TEST"	Prints a line on printer.
MEM	PRINT FRE (0)	
POINT (x,y)	OPEN#5, "D:" INPUT or GET#iocb, AT(s,b) INPUT#5, AT(sector,byte) or PUT#iocb, AT(s,b)	
PRINT @ n, list	PRINT#6, AT(x,y);list	
PRINT	CLOAD	Writes data to cassette.
RANDOM	RANDOMIZE	
SYSTEM	DOS	

APPENDIX I

CONVERTING APPLESOFT PROGRAMS TO ATARI MICROSOFT BASIC

Applesoft started from exactly the same BASIC source as PET BASIC, so once again there are very few pure language issues in converting programs to ATARI Microsoft BASIC.

- Apple added two language features to Applesoft to enhance compatibility with their integer BASIC. They are: ONERR for error trapping and POP for eliminating GOSUB entries. ONERR can be easily converted to ON ERROR in ATARI Microsoft BASIC. POP has no equivalent since it allows a very unstructured form of programming where subroutines aren't really subroutines. To convert, add a flag, change the POP to set the flag, RETURN, and then have a statement at the RETURN point check the flag and clear it and branch if it is set.
- The Apple default display size is different from the ATARI display (actual screen size is the same). Menus and tables laid out to use the full display will have to be edited.
- The Apple disk and peripheral I/O scheme is unique. Prints to specific channels are used to activate plug-in peripheral cards. The prints for the cards all have to be reprogrammed.
- The most difficult conversion task is changing the graphics and sound statements. The overall Apple high-resolution display size is 280 by 192. The color control is fairly unusual because each pixel cannot independently take on all color values. The sound port is a single bit.
- A variety of CALL statements are used in Applesoft to trigger machine-specific features. Use of PEEK and POKE is much rarer but also must be changed.
- Use of machine language generally will depend on the exact memory layout of the Apple Computer. Since the microprocessor is the same, machine language can be converted when the source is available except for references to the Apple Operating System.
- RND is different. Apple RND with a positive argument (generally 1) returns a number between 0 and 1.

The following list of commands, statements, and functions illustrates how to convert Applesoft programs to ATARI Microsoft.

APPLESOFT	ATARI
CALL	USR (addr.)
ctrl C	BREAK
DEF FN name(x)=	DEF name(x)=
HLIN	PLOT x,y To x,y
HOME	CLS
HPLOT	PLOT
HTAB	PRINT AT(x,y)
INVERSE	▨
NORMAL	SHIFT ▨
LOAD	LOAD "D:"
NOTRACE	TROFF
ONERR GOTO n	ON ERROR GOTO
PDL	PEEK(address)
POP	add flag
	check flag
RECALL	OPEN#n, "C:" OUTPUT
SAVE	SAVE "D:"
TEXT	GRAPHICS 0
TRACE	TRON
VLIN	PLOT x,y TO x,y
VTAB	PRINT AT(x,y)

APPENDIX J

CONVERTING ATARI 8K BASIC TO ATARI MICROSOFT BASIC

ATARI Microsoft BASIC has improved graphics capabilities. You should consider rewriting graphics sections to take advantage of player-missile graphics. The SETCOLOR registers have been changed so that registers 0, 1, 2, and 3 now refer to player-missiles. What was SETCOLOR 0,cc, and 11 is now SETCOLOR 4,cc, and 11. SETCOLOR numbers have changed so that what was 0, 1, 2, 3, and 4 for the register assignment is now 4, 5, 6, 7, and 8. Other graphics changes include a FILL instruction and a "chained" PLOT that replaces DRAWTO.

Microsoft has improved string-handling capabilities. If your initial program occupies too much RAM you might consider compacting it by rewriting it in Microsoft.

The are minor differences in the RND() and other instructions when converting to ATARI Microsoft BASIC. The RND() can be made to work identically to the 8K BASIC's if you include a RANDOMIZE statement as part of your program. Programs that you have listed in 8K BASIC onto diskette can be loaded with ATARI Microsoft BASIC, and with a few changes should run.

ATARI 8K BASIC	**ATARI MICROSOFT BASIC**	**COMMENTS**
ADR(s$)	VARPTR(s$)	
CLR	CLEAR	
DEG	——-	
DRAWTO	PLOT x,y TO x,y	
LIST mm,nn	LIST mm-nn	
LOCATE x,y,var	var=SCRN$(x,y)	
LPRINT	OPEN#7, "P:" OUTPUT PRINT#7,	
OPEN#iocb, aexp1,aexp2, filespec filespec filespec	OPEN#iocb, filespec INPUT	
POINT#iocb sector, byte	INPUT#iocb, AT (sector, byte)	

ATARI 8K BASIC	ATARI MICROSOFT BASIC	COMMENTS
POP	——-	Use the USR function to call a machine-language routine. POP stack in 6502 code.
POSITION x,y	PRINT|#6,| AT(x,y)	
SOUND voice, pitch,noise,vol.	SOUND voice, pitch,noise,vol., duration	The duration is a new option. Duration is given in 1/60 of a second called jiffies. Thus, SOUND will work the same as when converting programs to Microsoft BASIC.
TRAP exp	ON ERROR exp	
USR(addr,list)	USR(addr,pointer)	You pass only one argument to the ATARI Microsoft BASIC rather than an argument list.
XIO	FILL x,y TO x,y	Microsoft's FILL plots points from x,y TO x,y. It scans to the right as it fills the area from x,y TO x,y. The sweep rightward stops and a new filling scan begins when a solid plotted line is met.

For other XIO commands, see Appendix N.

PADDLE, PTRIG, STICK, STRIG are done with PEEKs and POKEs in ATARI Microsoft. See the Section 15, "Game Controllers," for detailed description.

APPENDIX K

ATASCII CHARACTER SET

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
0	0	♥
1	1	├
2	2	▕
3	3	┘
4	4	┤
5	5	┐
6	6	╱
7	7	╲
8	8	◢
9	9	▗
10	A	◣
11	B	▝
12	C	▘
13	D	▔
14	E	▁
15	F	▖
16	10	♣
17	11	┌
18	12	─
19	13	┼
20	14	●
21	15	▄
22	16	▎
23	17	┬
24	18	┴
25	19	▌
26	1A	└

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
27	1B	EC
28	1C	↑
29	1D	↓
30	1E	←
31	1F	→
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	–
46	2E	
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	[illegible]
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	[illegible]
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	[illegible]
94	5E	^

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
95	5F	_
96	60	◆
97	61	[illegible]
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	♠
124	7C	|
125	7D	↖
126	7E	◀
127	7F	▶

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
128	80	▼
129	81	┣
130	82	▮
131	83	┛
132	84	┫
133	85	┓
134	86	╱
135	87	╲
136	88	◢
137	89	▗
138	8A	◣
139	8B	▝
140	8C	▘
141	8D	▔
142	8E	▁
143	8F	▖
144	90	♣
145	91	┏
146	92	━
147	93	╋
148	94	●
149	95	▬
150	96	▌
151	97	┳
152	98	┻
153	99	▊
154	9A	┗
155	9B	Ɛ
156	9C	↑
157	9D	↓
158	9E	←
159	9F	→
160	A0	☐

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
161	A1	!
162	A2	"
163	A3	#
164	A4	●
165	A5	%
166	A6	&
167	A7	'
168	A8	(
169	A9	)
170	AA	*
171	AB	+
172	AC	,
173	AD	−
174	AE	.
175	AF	/
176	B0	●
177	B1	●
178	B2	2
179	B3	3
180	B4	4
181	B5	5
182	B6	6
183	B7	7
184	B8	8
185	B9	●
186	BA	:
187	BB	;
188	BC	<
189	BD	=
190	BE	>
191	BF	?
192	C0	@
193	C1	●
194	C2	B

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
195	C3	$\boxed{\text{C}}$
196	C4	$\boxed{\text{D}}$
197	C5	■
198	C6	$\boxed{\text{F}}$
199	C7	$\boxed{\text{G}}$
200	C8	$\boxed{\text{H}}$
201	C9	$\boxed{\text{I}}$
202	CA	$\boxed{\text{J}}$
203	CB	■
204	CC	$\boxed{\text{L}}$
205	CD	$\boxed{\text{M}}$
206	CE	■
207	CF	$\boxed{\text{O}}$
208	D0	$\boxed{\text{P}}$
209	D1	$\boxed{\text{Q}}$
210	D2	■
211	D3	$\boxed{\text{S}}$
212	D4	$\boxed{\text{T}}$
213	D5	■
214	D6	$\boxed{\text{V}}$
215	D7	$\boxed{\text{W}}$
216	D8	■
217	D9	$\boxed{\text{Y}}$
218	DA	$\boxed{\text{Z}}$
219	DB	■
220	DC	$\boxed{\backslash}$
221	DD	$\boxed{]}$
222	DE	$\boxed{\wedge}$
223	DF	$\boxed{\_}$
224	E0	$\boxed{\blacklozenge}$
225	E1	$\boxed{\text{a}}$
226	E2	$\boxed{\text{b}}$
227	E3	$\boxed{\text{c}}$
228	E4	$\boxed{\text{d}}$

DECIMAL CODE	HEXADECIMAL CODE	CODE CHARACTER
229	E5	e
230	E6	f
231	E7	g
232	E8	h
233	E9	i
234	EA	j
235	EB	k
236	EC	l
237	ED	m
238	EE	n
239	EF	o
240	F0	p
241	F1	q
242	F2	r
243	F3	s
244	F4	t
245	F5	u
246	F6	v
247	F7	w
248	F8	x
249	F9	y
250	FA	z
251	FB	♠
252	FC	|
253	FD	↖
254	FE	◀
255	FF	▶

APPENDIX L

ALPHABETICAL DIRECTORY OF BASIC RESERVED WORDS

RESERVED WORD	**BRIEF SUMMARY OF BASIC STATEMENT**
ABS	Function returns absolute value (unsigned) of the variable or expression. **Example:** Y=ABS(A+B)
AFTER	Causes the placement of an entry on a time-interrupt list. The elapsed time to be associated with time interrupt is specified by the numeric expression in units of jiffies (1/60 of a second). **Example:** AFTER (180) GOTO 1000
AND	Logical operator: Expression is true only if both subexpressions joined by AND are true. **Example:** IF A=10 AND B=30 THEN END
ASC	String function returns the numeric ATASCII value of a single string character. **Example:** PRINT ASC(A$)
AT	Use to position disk or screen output via PRINT statement. **Example:** PRINT AT(S,B);"START HERE"
ATN	Function returns the arctangent of a number or expression in radians. **Example:** PRINT ATN(A)
AUTO	A command generating line numbers automatically. **Example:** AUTO 100,50
BASE	Use with OPTION statement to set minimum value for array subscripts. **Example:** OPTION BASE 1
CHR	Use with OPTION statement to allocate RAM for alternate character sets, where: CHR1 = 1024 bytes are allocated (128 characters), CHR2 = 512 bytes are allocated (64 characters), CHR0 = free the allocated RAM **Example:** OPTION CHR1
CHR$	String function returns a single string character equivalent to a numeric value between 0 and 255 in ATASCII code. **Example:** PRINT CHR$(48)

CLEAR	Use to set all strings to null and set all variables to zero. **Example:** CLEAR
CLEAR STACK	Resets all entries on the time stack to zero. **Example:** CLEAR STACK
CLOAD	Use to put programs on cassette tape into computer memory. **Example:** CLOAD
CLOSE	I/O statement used to close a file at the conclusion of I/O operations. **Example:** CLOSE #6
CLS	Erases the text portion of the screen and sets the background color register to the indicated value, if present. **Example:** CLS 35
COLOR	Establishes the color register or character to be produced by subsequent PLOT and FILL statements. **Example:** COLOR 2
COMMON	A program statement passing variables to a chained program. **Example:** COMMON A,B,C$
CONT	Continues program execution after a BREAK or STOP. **Example:** CONT
COS	Function returns the cosine of the variable or expression (degrees or radians). **Example:** A=COS(2.3)
CSAVE	Used to put programs that are in computer memory onto cassette tape. **Example:** CSAVE
DATA	I/O statement lists data to be used in a READ statement. **Example:** DATA 2.3,"PLUS",4
DEF	Statement having two applications: 1) Define an arithmetic or string function. **Example:** DEF SQUARE (X,Y)=SQR(X*X+Y*Y) 2) Define default variable of type INT, SNG, DBL, or STR. **Example:** DEFINT I-N
DEL	Delete program lines. **Example:** DEL 20-25
DIM	Reserves the specified amount of memory for matrix, array, or string array. **Example:** DIM A(3), B$(10,2,3)

END	Stop program, close all files, and return to BASIC command level. **Example:** END
EOF	Returns true (-1) if file is positioned at its end. **Example:** IF EOF(1)GOTO 300
ERL	Error line number. **Example:** PRINT ERL
ERR	Error code number. **Example:** IF ERR=62 THEN END
ERROR	Generate error of code (see table). May call user ON ERROR routine or force BASIC to handle error. **Example:** ERROR 17
EXP	Function raises the constant e to the power of expression. **Example:** B=EXP(3)
FILL	Fills in area between two plotted points with a color. **Example:** FILL 10,10 TO 20,20
FOR...TO...STEP	Use with NEXT statement to repeat a sequence of program lines. The variable is incremented by the value of STEP. **Example:** FOR DAY=1 TO 5 STEP 2
FRE(0)	Gives memory free space available to programmer. **Example:** PRINT FRE(0)
GET	Reads a byte from an input device. **Example:** GET#1,D
GOSUB	Branch to a subroutine beginning at the specified line number. **Example:** GOSUB 210
GOTO	Branch to a specified line number. **Example:** GOTO 90
GRAPHICS	Establishes which of the display lists and graphics modes, contained in the operating system are to be used to produce the screen display. **Example:** GRAPHICS 5
IF...THEN	If exp is true, the THEN clause is executed. Otherwise, the next statement is executed. **Example:** IF ENDVAL>0 THEN GOTO 200
IF...THEN...ELSE	If exp is true, the THEN clause is executed. Otherwise, the ELSE clause or next statement is executed. **Example:** IF X<Y THEN Y=X ELSE Y=A

INKEY$	Returns either a one-character string read from terminal or null string if no character pending at terminal. **Example:** A$ = INKEY$
INPUT	Read data from a device. **Example:** INPUT #1,A,B Read data from the keyboard. Semicolon after INPUT suppresses echo of carriage return/line feed. If a prompt is given, it will appear as written; if not, a question mark will appear in its place. **Example:** INPUT "VALUES";A,B
INSTR	Returns the numeric position of the first occurrence of string2 in string1 scanning from position exp. **Example:** INSTR(3,X$,Y$)
INT	Evaluates the expression for the largest integer less than expression. **Example:** C = INT(X + 3)
KILL	Delete a disk file. **Example:** KILL "D:INVEN.BAS"
LEFT$	Returns leftmost length characters of the string expression. **Example:** B$ = LEFT$(X$,8)
LEN	String function returns the length of the specified string in bytes or characters (1 byte contains 1 character). **Example:** PRINT LEN(B$)
LET	Assigns a value to a specific variable name. **Example:** LET X = I + 5
LINE INPUT	Read an entire line from the keyboard. Semicolon after LINE INPUT suppresses echo of carriage return/line feed. See INPUT. **Example:** LINE INPUT "NAME";N$
LIST	Display or otherwise output the program list. **Example:** LIST 100-1000
LOAD	Load a program file. **Example:** LOAD "D:INVEN"
LOCK	Sets the file locked condition for the file named in the string expression. **Example:** LOCK "D1:TEST.BAS"
LOG	Function returns the natural logarithm of a number. **Example:** D = LOG(Y-2)
MERGE	Merge program on disk with program in memory by line number. **Example:** MERGE "D:SUB1"

MID$	Returns characters from the middle of the string starting at the position specified to the end of the string or for length characters. **Example:** A$=MID$(X$,5,10)
MOVE	Moves bytes of memory from one area to another so that the block is not changed. **Example:** MOVE 45000,50000,6
NAME	Change the name of a disk file. **Example:** NAME "D:SUB1" AS "SUB2"
NEW	Delete current program and variables. **Example:** NEW
NEXT	Causes a FOR/NEXT loop to terminate or continue depending on the particular variables or expressions. **Example:** NEXT I
NOT	Unary operator used in logical comparisons evaluates to 0 if expression is non-zero; evaluates to 1 if expression is 0. **Example:** IF A=NOT B
NOTE	Causes the current disk sector number to be stored into the first variable and the byte number into the second variable for the file associated with the IOCB#. **Example:** NOTE #1,S,B
ON ERROR	Enables error trap subroutine beginning at specified line. If line=0, disables error trapping. If line=0 inside error trap routine, forces BASIC to handle error. **Example:** ON ERROR GOTO 1000
ON...GOSUB	GOSUB to statement specified by expression. (If exp=1, to 20; if exp=2, to 20; if exp=3, to 40; otherwise, error.) **Example:** ON DATE%+1 GOSUB 20,20,40
ON...GOTO	Branch to statement specified by exp. (If exp=1, to 20; if exp=2, to 30; if exp=2, to 40; otherwise, error.) **Example:** ON INDEX GOTO 20,30,40
OPEN	Open a device. Mode must be one of:INPUT, OUTPUT, UPDATE, and APPEND. **Example:** OPEN #1, "D:INVEN.DAT", OUTPUT
OPTION BASE	Declare the minimum value for array subscripts; n is 0 or 1. **Example:** OPTION BASE 1
OPTION CHR	Allocates space for alternate character sets. **Example:** OPTION CHR1
OPTION PLM	Allocates space for player-missile graphics. **Example:** OPTION PLM1

OPTION RESERVE	Allocates free space for programmer's use in assembly language program. **Example:** OPTION RESERVE(50)
OR	Logical operator used between two expressions. If either one is true, a "1" is evaluated. A "0" results only if both are false. **Example:** IF A=10 OR B=30 THEN END
PEEK	Function returns decimal form of contents of specified memory location. **Example:** PRINT PEEK (&2000)
PLM	Used with OPTION statement to allocate RAM for player-missile graphics, where: PLM1= single-line resolution PLM2= double-line resolution PLM0= free the allocated RAM **Example:** OPTION PLM2
PLOT	Plots a single point on the screen or draws from one point to another. **Example:** PLOT 10,10 TO 20,20
POKE	Insert the specified byte into the specified memory location. **Example:** POKE &2310,255
PRINT	I/O command causes output from the computer to the specified output device. **Example:** PRINT USING "!";A$,B$
PUT	Write byte-oriented data to a data file. **Example:** PUT #3,4
RANDOMIZE	Reseed the random number generator. **Example:** RANDOMIZE
READ	Read the next items in the DATA list and assign to specified variables. **Example:** READ I,X,A$
REM	Remarks. Allows comments to be inserted in the program without being executed by the computer on that program line. Alternate forms are exclamation point (!) and apostrophe ('). **Example:** REM DAILY FINANCES
RENUM	Renumber program lines. **Example:** RENUM 100,,100
RESERVE	Used with OPTION statement to reserve a specified number of bytes for the programmer's use. **Example:** OPTION RESERVE (512)

RESTORE	Resets DATA pointer to allow DATA to be read more than once. **Example:** RESTORE
RESUME	Returns from ON ERROR or time-interrupt routine to statement that caused error. RESUME NEXT returns to the statement after error causing statement and RESUME line number returns to statement at line number. **Example:** RESUME
RETURN	Return from subroutine to the statement immediately following the one in which GOSUB appeared. **Example:** RETURN
RIGHT$	Returns rightmost length characters of the string expression. **Example:** C$ = RIGHT$(X$,8)
RND	Generates a random number. If parameter = 0, returns random between 0 and 1. If parameter >0, returns random number between 0 and parameter. **Example:** E = RND(10)
RUN	Executes a program starting with the lowest line number. **Example:** RUN
SAVE	Save the program in memory with name "filename." ,A saves program in ASCII. ,P protects file. Also, SAVE "filename" LOCK encrypts the program as it writes to disk. **Example:** SAVE"D:PROG"
SCRN$	The character or color number of the pixel at an x-coordinate and a y-coordinate is returned as the value of the function. **Example:** A = SCRN$ (23,5)
SETCOLOR	Associates a color and luminance with a color register. **Example:** SETCOLOR 0,5,5
SGN	1 if expression > 0 0 if expression = 0 -1 if expression < 0 **Example:** B = SGN(X + Y)
SIN	Function returns trigonometric sine of given value in degrees. **Example:** B = SIN(A)
SOUND	Statement initiates one of the sound generators. **Example:** SOUND 1,121,8,10,60

SPC	Use in PRINT statements to print spaces. **Example:** PRINT SPC(5),A$
SQR	Function returns the square root of the specified value. **Example:** C=SQR(D)
STACK	Returns the number of entries available on time stack. **Example:** A=STACK
STATUS	Function accepts a single argument as either a numeric or string then returns status of logical unit number or file. **Example:** ST=STATUS(2)
STOP	Causes execution to stop, but does not close files. **Example:** STOP
STR$	Function returns a character string equal to numeric value given. **Example:** PRINT STR$(35)
STRING$	Returns a string composed of a specified number of replications of A$. **Example:** X$=STRING$(100,"A") Returns a string 100 units long containing CHR$(65). **Example:** Y$=STRING$(100,65)
TAB	Use in PRINT statements to tab carriage to specified position. **Example:** PRINT TAB(20),A$
TAN	Tangent of the expression (in radians). **Example:** D=TAN(3.14)
TIME	Returns numeric representation of time from the real time clock. **Example:** ATM=TIME
TIME$	The time of day in a 24-hour notation is returned in the string. The format is HH:MM:SS. **Example:** TIME$="08:55:05" PRINT TIME$
TROFF	Turn trace off. **Example:** TROFF
TRON	Turn trace on. **Example:** TRON
UNLOCK	Statement terminates the LOCK condition. **Example:** UNLOCK "D1:DATA.OUT"
USING	Provides string format for printed output. **Examples:** PRINT USING "###.##";PDOLLARS

USR	Function returns results of a machine-language subroutine. **Example:** X=USR(SVBV, VARPTR(ARR(0)))
VAL	Function returns the equivalent numeric value of a string. **Example:** PRINT VAL("3.1")
VARPTR	Returns address of variable or graphics area in memory, or zero if variable has not been assigned a value. **Example:** I=VARPTR(X)
VERIFY	Compares the program in memory with the one on filename. If the two programs are not found to be identical, it returns an error. **Example:** VERIFY "D1:DATA.OUT"
WAIT	Equality comparison, pauses execution until result equals third parameter. **Example:** WAIT &E456,&FF,30
XOR	Bitwise exclusive OR (integer). **Example:** IF A XOR B=0 THEN END

APPENDIX M

ERROR CODES

CODE	ERROR
1	NEXT without FOR. NEXT was used without a matching FOR statement. This error may also happen if NEXT variable statements are reversed in a nested loop.
2	Syntax. Incorrect punctuation, open parenthesis, illegal characters, and misspelled keywords will cause syntax errors.
3	RETURN without GOSUB. A RETURN statement was placed before the matching GOSUB.
4	Out of data. A READ or INPUT # statement was not given enough data. DATA statement may have been left out or all data read from a device (diskette, cassette).
5	Function call error. Attempted to execute an operation using an illegal parameter. Examples: square root of a negative number, or negative LOG.
6	Overflow. A number that is too large or small has resulted from a mathematical operation or keybord input.
7	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.
8	Undefined line. An attempt was made to refer or branch to a nonexistent line.
9	Subscript out of range. A matrix element was assigned beyond the dimensioned range.
10	Redefinition error. Attempt to dimension a matrix that had already been dimensioned using the DIM statement or defaults.
11	Division by zero. Using zero in the denominator is illegal.
12	Illegal direct. The use of INPUT, GET or DEF in the direct mode.
13	Type mismatch. It is illegal to assign a string variable to a numeric variable and vice-versa.

15	Quantity too big. String variable exceeds 255 characters in length.
16	Formula too complex. A mathematical or string operation was too complex. Break into shorter steps.
17	Can't continue. A CONT command in the direct mode cannot be done because program encountered an END statement.
18	Undefined user function. The USR function cannot be carried out. User code has an error in logic or USR start points to wrong memory address.
19	No RESUME. End of program reached in error-trapping mode.
20	RESUME without error. RESUME encountered before ON ERROR GOTO statement.
21	FOR without NEXT. NEXT statement encountered before a FOR statement.

For an explanation of the following error codes, see *ATARI Disk Operating System II Manual.*

128	BREAK abort
129	IOCB
130	Nonexistent device
131	IOCB write only
132	Invalid command
133	Device or file not open
134	Bad IOCB number
135	IOCB read-only error
136	EOF
137	Truncated record
138	Device timeout
139	Device NAK
140	Serial bus

141	Cursor out of range
142	Serial bus data frame overrun error
143	Serial bus data frame checksum error
144	Device-done error
145	Read after write-compare error
146	Function not implemented
147	Insufficient RAM
160	Drive number error
161	Too many OPEN files
162	Disk full
163	Unrecoverable system data I/O error
164	File number mismatch
165	File name error
166	POINT data length error
167	File locked
168	Command invalid
169	Directory full
170	File not found
171	POINT invalid

APPENDIX N

USE OF THE CIO CALLING USR ROUTINES

There are three, prewritten USR routines provided on the ATARI Microsoft BASIC diskette for your use. These routines provide a flexible way to interact with the Central Input/Output (CIO) facilities of your ATARI Home Computer. These routines (or similar routines if you prefer to write your own) allow the BASIC program to send or retrieve data directly to or from an Input/Output Control Block (IOCB). The IOCB's are discussed in detail in the *ATARI Operating System Users Manual* (part of *ATARI Personal Computer System Technical Users Notes*). Refer to that document for a complete description of CIO capabilities.

These routines allow the BASIC programmer to perform such tasks as retrieving a disk directory, formatting a diskette, or conditioning a specific IOCB and its associated logical unit number to interface with RS-232 devices. Following is a brief description of how to read these routines into your own program and how to use them.

STEP 1. Inserting the Routines Into a BASIC Program.

All three routines are contained in the file **CIOUSR** on the ATARI Microsoft BASIC diskette. They are in a machine-readable format, ready to be poked directly into RAM. To allocate RAM for this purpose, use the OPTION RESERVE n statement where n should be at least 160. Get the starting address of the reserved area with the statement ADDR=VARPTR(RESERVE). Then, the following code can be used to put the routines into the BASIC program:

```
OPEN #1, "D:CIOUSR"
INPUT FOR I = 0 TO 159
GET #1,
POKE ADDR+I,:NEXT I
CLOSE #1
```

STEP 2. Setting Up the Data Array

The routines are now in the reserved area of the BASIC program. There are three routines called PUTIOCB, CALLCIO, and GETIOCB. PUTIOCB starts at RAM location ADDR. CALLCIO starts at ADDR+61. GETIOCB starts at ADDR+81.

The GETIOCB routine retrieves the user-alterable bytes from a specified IOCB and puts them into an integer array of length 10. The programmer may alter any of these parameters and then put the new values back into the IOCB with the PUTIOCB routine. When the proper parameters have been set, the use of CALLCIO will cause the

IOCB values to be executed by the CIO facility. The next step is to dimension an integer array to use for retrieval and storage of the IOCB parameters. This array should be dimensioned to 10 using a BASE option of zero. Following is a list of the elements of the array and what each is used for:

Element Number	IOCB Parameter
0	This element is the number of the IOCB to be used (1 to 8).
1	COMMAND CODE
2	STATUS — returned
3	BUFFER ADDRESS
4	BUFFER LENGTH
5-10	AUX byte 1 - 6

Each element of an integer array has two bytes of data storage, so the buffer address in element 3 will fit into a single integer element.

STEP 3. Calling the USR Routines

A USR call is used to execute the CIOUSR routines. The GETIOCB routine will return to the program the current values of the specified IOCB's parameters. After changing these parameters in the array, to effect some CIO function (i.e., setting the baud rate on an RS-232 port), the PUTIOCB routine is called to put the desired values into the specified IOCB. Then the CALLCIO routine is called to execute the CIO facility. Following is the syntax necessary to call each of the routines:

```
nvar = USR(addr,VARPTR(array(0)))
```

where:

nvar — a numeric variable which will receive the status of the CIO function in the case of a CALLCIO call, otherwise it will not be specifically affected by these routines.

addr — the starting address of the proper CIOUSR routine, in our current example these would be ADDR for PUTIOCB,ADDR+61 for CALLCIO and ADDR+81 for GETIOCB.

array(0) — the array will be the integer array the program uses for data retrieval and storage for the routines. Passing the VARPTR of element zero of this array to the routines tells them where to begin retrieving the data from, starting with the IOCB number.

Following is an example program to set up and use an RS-232 port for telecommunications. Also see the "Disk Directory Program" in Appendix A for another example of the use of these routines.

```
10 !
20 !ROUTINE TO DEMONSTRATE
30 !CIOUSR ROUTINES...
40 !
50 !PROVIDES TELECOMMUNICATIONS
60 !WITH RS-232 DEVICES
70 !
80 DIM CIO%(10),S%(10)
90 CIO%(0) = 2
100 S%(0)=5:S%(1)=&0D
110 OPTION RESERVE 200
120 ADDR=VARPTR(RESERVE)
130 PUTIOCB=ADDR
140 CALLCIO=ADDR+61
150 GETIOCB=ADDR+81
160 OPEN #1,"D:CIOUSR" INPUT
170 FOR I=9 TO 159
180 GET #1,D:POKE ADDR+I,D
190 NEXT I
200 CLOSE #1
210 OPEN #1,"K:" INPUT
220 CIO%(0)=2
230 CIO%(1)=3
240 FSPEC$="R:"
250 Z=VARPTR(FSPEC$)
260 Y=VARPTR(CIO%(3))
270 POKE Y,PEEK(Z+2)
280 POKE Y+1,PEEK(Z+1)
290 Y=VARPTR(S%(3))
300 POKE Y,PEEK(Z+2)
310 POKE Y+1,PEEK(Z+1)
320 CIO%(5)=13
330 A=USR(PUTIOCB,VARPTR(CIO%(0)))
340 A=USR(CALLCIO,VARPTR(CIO%(0)))
350 A=USR(GETIOCB,VARPTR(CIO%(0)))
360 CIO%(1)=40
370 CIO%(5)=0:CIO(6)=0
380 A=USR(PUTIOCB,VARPTR(CIO%(0)))
390 A=USR(CALLCIO,VARPTR(CIO%(0)))
400 X=USR(PUTIOCB,VARPTR(S%(0)))
410 !
420 !SHOULD BE READY TO GO NOW
430 PRINT "STARTING LOOP"
440 !
450 GET #1,A:PUT #2,A:POKE 764,255
460 X=USR(CALLCIO,VARPTR(S%(0))):IF PEEK(747)=0 THEN 480
470 GET #2,D:IF D<>10 THEN PRINT CHR%(D);
480 IF PEEK(764)<> 255 THEN 450
490 GOTO 460
```

APPENDIX O

ACTIONS TAKEN WHEN PROGRAM ENDS

	ACTIONS TAKEN		
Key Pressed or Statement Executed	**Close All Files**	**Run Out the Stack**	**Clear Sound**
STOP ERRORS BREAK	NO	NO	YES
Running off the last statement or "END"	YES	YES	YES
After a direct mode statement	NO	YES	NO
RUN	YES	NO	YES

INDEX